n-Queens problem

Site Difficulty: 1 kyu / Perceived difficulty: Very Hard

The eight queens puzzle is the problem of placing eight chess queens on an `8 × 8` chessboard so that no two queens threaten each other. Thus, a solution requires that no two queens share the same row, column or diagonal. The eight queens puzzle is an example of the more general `n` queens problem of placing `n` non-attacking queens on an `n × n` chessboard. You can read about the problem on its Wikipedia page: Eight queens puzzle.

You will receive a (**possibly large**) number `n` and have to place `n` queens on a `n × n` chessboard, so that no two queens attack each other. This requires that no two queens share the same row, column or diagonal. You will also receive the mandatory position of one queen. This position is given 0-based with `0 <= row < n` and `0 <= col < n` `{row, col}`. The coordinates `{0, 0}` are in the top left corner of the board. For many given parameters multiple solutions are possible. You have to find one of the possible solutions, all that fit the requirements will be accepted.

You have to return the solution board as a string, indicating empty fields with `'.'` (period) and Queens with `'Q'` (uppercase Q), ending each row with `'\n'`.

If no solution is possible for the given parameters, return `""` (empty string).

input parameters are:

- `int n`
- `std::pair<int mandatory_row, int mandatory_column>`

Tests

- there are 8 tests for very small boards $(n \leq 10)$
- there are 8 tests for cases without solution
- there are 5 tests for small boards $(10 < n \leq 50)$
- there are 5 tests for medium boards $(100 < n \leq 500)$
- there are 5 tests for large boards $(500 < n \leq 1000)$

Example

For input of `size = 8`, mandatory `column = 3` and mandatory `row = 0`, your solution could return:

`"...Q....\n......Q.\n..Q.....\n.......Q\n.Q......\n....Q...\nQ.......\n.....Q..\n"`

giving the following board:

```
. . . Q . . . .
. . . . . . Q .
. . Q . . . . .
. . . . . . . Q
. Q . . . . . .
. . . . Q . . .
Q . . . . . . .
. . . . . Q . .
```

(Other solutions to this example are possible and accepted. The mandatory queen has to be in its position, in the example in the first row at `col=3`, `row=0`.)

Problem solution

This problem is classically used in compsci to teach beginners the concept of algorithmic backtracking.

Although this approach is quite intuitive, going about solving the problem this way carries with it a horribly bad time complexity of $O(n!)$. This is fine for small n but since in this problem n ranges from $0 \leq n \leq 1000$, this approach is not feasible since even for $n = 20 \implies 20! = 2.432902 \times 10^{18}$ iterations.

There are quite a few ways of optimizing backtracking for n-Queens such as pruning, memoization, symmetry reduction, etc... . However, even with these optimizations the average time complexity will still be a not very spectacular $O(c^n)$ where $c \approx [1.5; 2]$ and worst case $O(n \cdot 2^n)$. Again, this makes large n infeasible.

Because of these initial observations I required a different approach.

Attempt one: Simulated Annealing

Simulated Annealing (SA) is a metaheuristic technique inspired by metallurgy where controlled entropy is utilised to selectively temper out defects and converge to an optimal solution. For a given CSP, such as in this case the n-Queens problem, SA is able to relatively quickly converge, especially compared to a brute-force apporach.

Viewed at a high level, SA works as follows: The goal is to bring the system (n-Queens), from an arbitrary initial state, to a state with the minimum possible energy (conflicts). First define a neighbor and an energy function with these goals:

- The neighbor function takes the current state of the board, makes a small change at random and returns the result.
- The energy function calculates the amount of conflicts. The algorithm starts with a specific temperature, which cools by a specified factor each iteration. The temperature controls the likelyhood of a counter-productive neighbor being accepted. This is so that the solution space is sufficiently traversed in the hopes of arriving at an optimal solution.

More specifically:

- Let T be temperature
- Let T_{min} be minimum temperature
- Let α be cooling factor
- Let $E_{current}$ be current board energy
- Let $E_{neighbor}$ be neighbor board energy
- Let $\Delta E = E_{neighbor} - E_{current}$

Each iteration, T is multiplied by α $(T := T\alpha)$ to cool the temperature and lower the chances of a worse neighbor being accepted the closer the algorithm gets to a solution. More specifically:

$$\text{if } \left(\Delta E < 0 \vee \exp\left(\frac{-\Delta E}{T}\right) > x \text{ such that } x \in [0,1] \right)$$

$$\text{Then accept the neighbor.}$$

Where $\exp(x) = e^x$, $e =$ Euler's number and x is randomized each iteration.

To elaborate on the acceptance criteria, there are basically three paths leading to two options:

1. $\Delta E < 0$. If the difference in energy is negative, the neighbor is better. Accept neighbor.
2. $\vee \exp\left(\frac{-\Delta E}{T}\right) > x \text{ such that } x \in [0, 1]$. If the first criteria is not met, calculate the chance this neighbor will be accepted based on the difference in energy, the current temperature and if that chance is greater than a randomly generated number x between 0 and 1 $(x \in \mathbb{R})$, accept neighbor.
3. If both criteria fail, discard the neighbor.

Unfortunately there are two main glaring issues with using simulated annealing for n-Queens with large n.

Critical issue 1: getting stuck on local minima (stagnation)

SA has the tendancy to stagnate on almost-but-not-really optimal solutions. Usually within a 2-3% range of an optimal $E = 0$ solution. This is acceptable for some problems such as the TSP, where complete accuracy isn't a requirement for the solution to be practical, but that isn't the case for n-Queens.

This could be solved by implementing a stagnation detection and resolving algorithm.
The problem with doing this is that you have to define when something counts as stagnation, what actions to take when stagnation has been detected, how much action to take and when to take it. Meaning I'd have to introduce quite a few new variables which need adjusting on top of the very sensitive annealing variables. These variables also can't be constant and have to be in function of n.

Profiling and solving this very delicate balancing act is a large time investment with no real guarantee of success in the end, especially when combined with the next issue.

Critical issue 2: converging to a perfect state

As previously stated, SA is efficient at getting close to an optimal solution. Actually converging from a board with e.g. $E = 8$ to $E = 0$ is a different story. This isn't very efficient since SA is basically a glorified random number generator with no real idea of what it's working toward. The amount of time required to fully converge gets very steep as $n \to 1000$.

Conclusion

Because of the two major issues stated above, I deemed SA to be a poor fit for this particular problem.

However, the concept of dynamically adjusting agressiveness of action based on how close to a solution the algorithm currently is was implemented in my final solution.

Attempt two: min-conflicts

Min-conflicts is a heuristics-based search algorithm which shines in quickly converging from a semi-high energy state to a perfect state ($E = 0$).

The algorithm creates a list of all queens with the very highest E values (most conflicts), picks out one of the queens from that list at random and places it in a new location which has been calculated to have lowest possible E value out of all moves which can be made with that queen. If more than one lowest energy location exists, pick one at random as to more effectively explore the entire solution space.

This approach is much more efficient and simpler for this particular problem compared to SA since there aren't any tempermental variables to tangle with.

There are however still problems.

Critical issue 1: stagnation

As with SA, stagnation is a big issue. Especially with small $n < 10$, where this algorithm has a tendancy to take tens of minutes over something which could conceivably be calculated in a few milliseconds. The algorithm stagnates about 50% of the time for small n. Min-conflicts doesn't have very sensitive variables as with SA, so I felt much more comfortable putting in the time of running test-suites to find the optimal formulae to calculate the variables in order to fine-tune a sophisticated stagnation D/R (detection and resolution) system.

Here follows a more in-depth explaination of the stagnation D/R algorithm.

- Let $\mathrm{iterationCount}$ be the amount of times the algorithm has looped.
- Let $\mathrm{stagnationCheckInterval}$ be the amount of iterations between checks, calculated as $10n$. $(\mathrm{stagnationCheckInterval} \in \mathbb{Z}^+)$
  In other words, every $\mathrm{iterationCount} \equiv 0 \pmod{10n}$, stagnation D/R runs.

To resolve stagnation, the computer first has to be able to detect stagnation, so stagnation has to be defined. I have defined it as "if three samples of the board's total energy, taken at specified points within $stagnationCheckInterval$, and the current total energy are all equal, the board has stagnated".

The algorithm periodically takes energy samples of the entire board.
Here are the criteria for when to take each sample:

- S_1: at $\left\lfloor \frac{\text{stagnationCheckInterval}}{1.5} \right\rfloor$
- S_2: at $\text{stagnationCheckInterval}$
- S_3: at $\left\lfloor \frac{\text{stagnationCheckInterval}}{3.0} \right\rfloor$

When dividing as with S_1 and S_3, the $\text{floor}()$ function is utilised so that modulo checks with iterationCount, which is $\mathbb{Z}^+$, don't consistently fail.

By using tri-sampling instead of only one sample, the chance of false positives is lowered significantly. This and the fact that all samples have to match up perfectly without margin for error makes it so there are significantly less false positives. I've found that when the algorithm is actually stuck energy fluctuations are bascially non-existent, so I'm able to get away with these strict requirements.

Once detected, a stagnation is resolved by perturbation. In essence this means randomly making a pre-specified amount of moves not taking into account if the move is an overall benifit or detriment to the solution, in an attempt to escape local minima.

The perturbation amount must be defined in function of n, since using a constant wouldn't scale. This could be defined as something static such as $\frac{n}{4}$, but this way you increase the chances of wasting computation time the closer a stagnation occurs towards the end since you now have higher chances of wasting a carefully calculated correct queen placement each perturbation.

My method of resolving this potential waste of resources is by calculating the perturbation amount dynamically using a modified version of SA. Instead of using temperature, this version is based on the ratio of the total board energy compared to the energy at the start of the entire problem where the closer the current board energy is to the start, the more agressive the algorithm is. By being more gentle toward the end, there is much less chance of loosing a good board setup and having to start over.

Stagnation amount calculation:

$$\left\lceil p_{min} + \frac{E_{current}}{E_{begin}} \left(p_{max} - p_{min}\right) \right\rceil$$

Where:

- p_{min}: minimum amount of perturbations
- p_{max}: maximum amount of perturbations
- E_{begin}: board energy at initialization
- $E_{current}$: current board energy

In practice these numbers were used:

$$\left\lceil 1 + \frac{E_{current}}{E_{begin}} \left(\frac{n}{10} - 1\right) \right\rceil$$

This way, even when $E \to 0$, a positive stagnation check will result in a positive stagnation check will result in at least one perturbation, and at most $\frac{n}{10}$ perturbations.
Making too many alterations, e.g. $\frac{n}{4}$, is detrimental to the carefully computed board min-conflicts has constructed. Local minima in later stages can be escaped by smaller, non-overzealous perturbations.

Since this algorithm makes such extensive use of E-values, this seems like a good time to explain how these are calculated.

On the computation of E

The E-value or energy of a is queen the amount of conflicts that specific queen currently has with other queens, in other words how many other queens she can attack from her current position. How a queen can attack is defined in the rules of chess.

To compute this I utilised the following method:
In a matrix (such as a chessboard), each diagonal, anti-diagonal, horizontal and vertical line can be given an index, which could be seen as a name of sorts.

E.g. here is a table constructed by the formula $i = x - y$ to illustrate the concept of indexed diagonals:

/	0	1	2	3	4	5
0	0	1	2	3	4	5
1	-1	0	1	2	3	4
2	-2	-1	0	1	2	3
3	-3	-2	-1	0	1	2
4	-4	-3	-2	-1	0	1
5	-5	-4	-3	-2	-1	0

As seen above, diagonals can be indexed or "named" e.g. " -1 " or "3".
By using this concept you can compute if any diagonals or horizontals are being shared relatively cheaply and then by summing up the results you calculate E.

Due to the nature of this problem a valid solution means that there can be only one queen per row, so there is no need to represent the chessboard with an actual 2D-array.
I've defined this datatype to represent the board: `using Board = std::vector<int>`. This way the index of the array represents the row and the value represents the column.
Limiting a queen's movements to one column also significantly improves the chances of a change made being positive and this cuts the amount of required compares per queen per iteration by a factor of n.

Knowing this, a formula to compute the board's energy can be constructed:

$$E = \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} \left[(b[i] = b[j]) \vee (|b[i] - b[j]| = |i - j|) \right]$$

This formula iterates over all pairs (i, j) where $0 \leq i \leq j \leq n$ and $b := board_{current}$.
If this pair satiesfies one of the constraints, E is incremented by one.
Per pair one and only one constraint can be satisfied at a time so there is no loss in accuracy by compressing all constraints into one large constraint.

Do this for all values and you have calculated the energy value E of the entire board.

Critical issue 2: initial convergence

The second large problem is initial convergence. In other words going from the initial energy of the arbitrary starting board (e.g. $E = 1800$) to a lower energy which min-conflicts can comfortably work with (e.g. $E = 100$).

This isn't a big issue with smaller n but as $n \to 1000$ this starts getting very time-consuming. Enough so that this algorithm is incapable of solving `nQueens(1000)` withing codewars' time constraint of `160000 ms`, making this a critical issue.

Although there are many possible solutions for this issue, after (a hellish amount of) trial-and-error I settled on altering the function which generated the starting layout arbitrarily to make use of a heuristic so that it creates a starting-board with energy-levels much closer to the eventual solution.

The heuristic I selected to impelement is the greedy algorithm heuristic.
Basically by making locally-optimal choices, choiches which only serve to minimize energy at that moment and not globally, the algorithm can place the queens such that the starting energy is much more managable.

Implementation initially caused a spike in stagnations and average execution times since this approach eliminated any randomness and concequently also caused the solution space to be less effectively explored. The simple solution for this minor issue was to introduce a small amount of randomness into the function.

Since this function didn't seem to be named I've dubbed it "gan-init" (Greedy-Random Initialization).

Gan-init exponentially improved average execution times causing me to mark this issue as solved.

Final phase

Since the algorithm was now capable of efficiently calculating n-Queens with large n in a timely manner, there were a few more constraints to which have to be added in order to comply with the problem definition.

Mandatory queen

The problem definition specified an argument

`std::pair<int mandatory_row, int mandatory_column>`, which specifies a location for a queen which must be present in the final solution. This is mostly there to improve the difficulty of the issue since this comes with new poblems and conciderations.

To implement this one can simply modify gan-init to place the mandatory queen as the first queen and then construct the initial board arround that queen. For min-conflicts you simply skip the row where the mandatory queen is located so it doesn't get included in any lists and by extention also doesn't get altered.

The problem this causes is that now, especially for smaller boards $4 \leq n \leq 7$, you can have queen placements which block all possible solutions. These so called "queen-blockades" cannot be resolved. In other words they have to be detected and then the appropriate response for "invalid" has to be given.

E.g. there are only two valid configurations for $n = 4$:

		$\mathbf{Q_3}$	
$\mathbf{Q_1}$			
			$\mathbf{Q_4}$
	$\mathbf{Q_2}$		

$[(0,1),(1,3),(2,0),(3,2)]$

	$\mathbf{Q_2}$		
			$\mathbf{Q_4}$
$\mathbf{Q_1}$			
		$\mathbf{Q_3}$	

$[(0,2),(1,0),(2,3),(3,1)]$

If a mandatory queen position doesn't match any of these positions, that queen will block any possible solution.

The best way to detect if any given input is valid is by looking at $\mathrm{iterationCount}$. Because of earlier test-suite results, the average number of iterations for any given n is known, so now it's only a matter of introducing a stopping mechanism whenever the algorithm goes over the ± average amount of iterations plus some extra margin to minimize false negatives in case of bad luck.

Mandatory queens are now correctly implemented.

Base cases

The only thing left to take into account are the base cases. Cases in which the algorithm is not applicable or cannot be used in order to determine the result based on the given arguments.

- $n = 1$: Trivial solution. Simply one queen in the one free cell.
- $\{n \mid n < 4 \land n \neq 1\}$: No solutions possible.

With that the complete solution is constructed.

Personal notes

I imposed the constraint on myself to solve this problem only with general methods for CSP's, without looking at whitepapers outlining custom n-Queens solving algorithms. This to force myself to learn more universally applicable concepts.

This problem wasn't easy in the slightest but I managed to get it in the end and I had some fun along the way.