Heuristic Approach for Efficiently Solving Large-Scale n-Queens Problems with a Pre-Specified Mandatory Queen

Gilles Van pellicom

September 13, 2024

Abstract

The n-Queens problem tasks us with placing n queens on an $n \times n$ chessboard such that no two queens attack each other. This includes ensuring no queens share the same row, column, diagonal or anti-diagonal. Additionally, to make this version of the problem more challenging, one queen's position is fixed. The mandatory queen placement introduces challenges in certain cases, leading to unsolvable configurations, which are also handled. This explanation explores different approaches to solving the n-Queens problem for varying board sizes up to $n = 1000$.

This explanation recognizes the broad scope of the material used and as such will include links to articles explaining specific concepts each time they are initially mentioned. These links are embedded in the text for the digital version but can also be found in the references section on the last page if you are reading the printed version.

Contents

1 Introduction

This problem is classically used in computer science to teach beginners the concept of algorithmic backtracking[1]. Although this approach is quite intuitive, solving the problem this way carries a horribly bad time complexity[2] of $\mathcal{O}(n!)$. This is acceptable for small n, but since in this version of the problem $0 \leq n \leq 1000$, this approach is not feasible.

E.g. even for $n = 20 \implies \mathcal{O}(20!) \approx 2.432902 \cdot 10^{18}$ iterations.

There are quite a few ways of optimizing backtracking for the n-Queens problem, such as pruning[3], memoization[4], and symmetry reduction[5]. However, even with these optimizations, the average time complexity will still be a not very spectacular $\mathcal{O}(c^n)$ where $c \approx [1.5; 2]$ and worst case $\mathcal{O}(n \cdot 2^n)$. Again, this makes large n infeasible. Because of these initial observations, I realised I required a different approach.

2 Attempt One: Simulated Annealing

Simulated annealing[6] (SA) is an example of a randomized algorithm[7]. SA is a metaheuristic[8] technique inspired by metallurgy[9] where controlled entropy[10] is utilised to selectively temper out defects and converge to an optimal solution. For a given CSP[11], such as in this case the n-Queens problem, SA is able to relatively quickly converge, especially compared to a brute-force[12] approach.

2.1 Simulated Annealing, Generalised

Viewed at a high level, SA works as follows: The goal is to bring the system (in this case n-Queens), from an arbitrary initial state, to a perfect state with the minimum possible energy level (conflicts). To utilise simulated annealing two main functions have to be defined:

- **Neighbor Function**: This function takes the current state of the board, makes a small random change, and returns the resulting state.
- **Energy Function**: This function computes the energy of a given state, which in this context corresponds to the number of conflicts.

The algorithm starts with a specific temperature, which cools by a specified factor each iteration. The temperature controls the likelihood of a counter-productive neighbor being accepted. This is done in order to the sufficiently traverse the solution space in the hopes of quickly arriving at an optimal solution.

2.1.1 Notation

- $\boldsymbol{T}$: current temperature and initially starting temperature.
- $\boldsymbol{T_{\text{min}}}$: minimum temperature.
- $\boldsymbol{\alpha}$: cooling factor.
- $\boldsymbol{E_{\text{current}}}$: current board energy.
- $\boldsymbol{E_{\text{neighbor}}}$: neighbor board energy.
- $\boldsymbol{\Delta E} = E_{\text{neighbor}} - E_{\text{current}}$

2.2 Main algorithm

Each iteration, T is multiplied by α, $T \coloneqq T \cdot \alpha$, to cool the temperature and lower the chances of accepting a worse neighbor as the algorithm gets closer to a solution. More specifically:

$$\text{If } \left(\Delta E < 0 \vee \exp\left(\frac{-\Delta E}{T}\right) > x \text{ where } 0 \leq x \leq 1 \right)$$
$$\text{Then accept the neighbor.}$$

Where $\exp(a) = e^a$, e is Euler's number[13] and x is randomized each iteration within given constraints. $(x \in \mathbb{R})$

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

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

2.3 Critical Issues

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

2.3.1 Critical Issue 1: Stagnation

SA has the tendancy to stagnate (get stuck in local minima). Meaning it gets stuck 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 Travelling Salesman Problem[14], 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 quite a few new variables which need adjusting would have to be introduced 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.

2.3.2 Critical Issue 2: Convergence

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 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 $n \to 1000$.

2.4 Conclusion

Because of the two major issues stated above, SA was deemed 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.

3 Attempt Two: Min-Conflicts

Min-conflicts[15] 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.

3.1 Critical Issues

Even though this approach should be much better suited to this problem, there are still issues.

3.1.1 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.

3.1.1.1 Notation

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

3.1.1.2 Solution

To resolve stagnation, the computer first has to be able to detect stagnation. For this purpose stagnation has to be defined, and I have defined it as such: "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:

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

When dividing as with S_2 and S_3, the $floor()$ 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. Also the fact that in this implementation all samples have to match up perfectly without margin for error results in significantly less false positives. I've found that when the algorithm is actually stuck energy fluctuations are bascially non-existent, therefore I'm able to get away with these strict requirements.

Once detected, a stagnation is resolved by perturbation[16]. 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 multiple carefully calculated correct queen placements each perturbation.

My method for 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. 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 losing a good board setup and having to start over.

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

Where:

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

In practice the following numbers were used:

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

This way, even when $E \to 0$, a positive stagnation check will result in at least one perturbation so that an attempt to solve the stagnation is always made, and at most $\frac{n}{10}$ perturbations. Making too many alterations to the board, 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.

Following is the C++ implementation of this formula:

```
1     for (int i = 0; i < ceil(1.0 + (static_cast<double>(E_board) / E_begin)
          * (n / 10.0 - 1)); ++i) {
2         // perform perturbation
3     }
```

3.1.2 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 the given time constraint of 160000 ms, making this a critical issue.

Although there are many possible solutions for this issue, after a large amount 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 which was selected to be impelemented is the greedy algorithm[17] heuristic. Basically by making locally-optimal choices, choices 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 but 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.

3.2 On the Computation of E

The E-value or energy of a queen is 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[18].

To compute E, the following methods were utilised:

3.2.1 Line Indexing

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. following is a table constructed using the formula $i = x - y$ to illustrate the concept of indexed diagonals:

	0	**1**	**2**	**3**	**4**	**5**	**n**	$x \rightarrow$
0	0	1	2	3	4	5	$\cdots$	
1	-1	0	1	2	3	4	$\cdots$	
2	-2	-1	0	1	2	3	$\cdots$	
3	-3	-2	-1	0	1	2	$\cdots$	
4	-4	-3	-2	-1	0	1	$\cdots$	
5	-5	-4	-3	-2	-1	0	$\cdots$	
n	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	
$y \downarrow$								

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.

3.2.2 Datatype

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. The datatype to represent the board is defined as follows:

```
1 using Board = std::vector<int>;
```

This C++ code defines a list of integers as a new datatype named "`Board`".

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 being productive. This cuts the number of required comparisons per queen per iteration by a factor of n.

3.2.3 Calculation

Using these insights, definitions and formulas can be constructed. A conflict is defined as:

$$conflict = (b[i] = b[j]) \vee (|b[i] - b[j]| = |i - j|)$$

Where $b =$ `Board` (zero-indexed) and [ ], indexing operator.

Using the definition of a conflict, a formula to compute the board's total energy can be constructed:

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

This formula iterates over all pairs (i, j) where $i, j \in \mathbb{Z}^+$, $i \leq n \wedge j \leq n$. If this pair satisfies 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.

Following is the C++ implementation of the energy formulae:

```
1   inline int calculateBoardE(const Board& b) {
2   const int n = static_cast<int>(b.size())-1; // Get board size (n)
3   int E = 0; // start with E := 0
4   for (int i = 0; i < n; ++i) {
5     // Dont count i == j
6     for (int j = i + 1; j < n; ++j) {
7       // if (not in same col) or (not in same diagonal)
8       // row doesn't need to be checked since the code doesn't allow two
            queens to generate in one row.
9       if (b[i] == b[j] || std::abs(b[i] - b[j]) == std::abs(i - j)) {
10        ++E; // increment E
11      }
12    }
13  }
14  return E;
15  }
```

3.3 Final Constraints

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 be handeled order to comply with the problem definition.

3.3.1 Base Cases

Base cases are cases in which the algorithm is not applicable or cannot be used in order to determine the result based on the given arguments.

- $\boldsymbol{n = 1}$: Trivial solution. Only one location so only one queen possible. Only one queen so no possible attacks. Handle by always returning a queen in cell $(0, 0)$.
- $\boldsymbol{\{n \in \mathbb{Z}^+ \mid n < 4 \wedge n \neq 1\}}$: No solutions possible. Handle by always returning failiure.

3.3.2 Mandatory Queen

The problem definition specifies an argument:

```
1 std::pair<int mandatoryRow, int mandatoryColumn>
```

This C++ code can be translated as: "a pair (x, y) which specifies the location for a queen which must be present in the final solution." This is mostly there to increase the difficulty of the problem, as it introduces new challenges and considerations.

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

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

For example, there are only two valid configurations for $n = 4$:

Solution 1:

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

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

Solution 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 does not match any of these positions, that queen will block any possible solution. Thus, the appropriate response for "invalid" has to be returned.

The best way to detect if any given input is valid is by examining the *iterationCount*. Due to earlier test-suite results, the average number of iterations for any given n is known. Thus, it is only a matter of introducing a stopping mechanism whenever the algorithm exceeds the average number of iterations plus some extra margin to minimize false negatives in case of very bad luck.

3.4 Conclusion

Although pure min-conflicts isn't able to handle this problem, by implementing the modifications and solutions outlined above, this method is very efficient at computing the n-Queens problem.

4 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 and I got some entertainment out of it.

References

In case you are viewing a printed version of this document and cannot click on the embedded links, the URLs are provided below for manual entry:

1. Algorithmic backtracking: https://en.wikipedia.org/wiki/Backtracking
2. Time complexity: https://en.wikipedia.org/wiki/Time_complexity
3. Decision tree pruning: https://en.wikipedia.org/wiki/Decision_tree_pruning
4. Memoization: https://en.wikipedia.org/wiki/Memoization
5. Symmetry reduction: https://www.khoury.northeastern.edu/home/wahl/Publications/ew05b.pdf
6. Simulated Annealing: https://en.wikipedia.org/wiki/Simulated_annealing
7. Randomized algorithm: https://en.wikipedia.org/wiki/Randomized_algorithm
8. Metaheuristic: https://en.wikipedia.org/wiki/Metaheuristic
9. Metallurgy: https://en.wikipedia.org/wiki/Metallurgy
10. Entropy: https://en.wikipedia.org/wiki/Entropy
11. Constraint Satisfaction Problem: https://en.wikipedia.org/wiki/Constraint_satisfaction_problem
12. Brute-force attack: https://en.wikipedia.org/wiki/Brute-force_attack
13. Euler's number: https://en.wikipedia.org/wiki/E_(mathematical_constant)
14. Travelling Salesman Problem: https://en.wikipedia.org/wiki/Travelling_salesman_problem
15. Min-Conflicts algorithm: https://en.wikipedia.org/wiki/Min-conflicts_algorithm
16. Perturbation: https://en.wikipedia.org/wiki/Perturbation_theory
17. Greedy algorithm: https://en.wikipedia.org/wiki/Greedy_algorithm
18. Queen movement rules: https://en.wikipedia.org/wiki/Queen_(chess)#Placement_and_movement