N-Queens Problem Documentation

N-Queens is the problem of placing *n* number of queens on a chess board of size $n \times n$ in which no queens can attack each other. Queens in chess can travel within the board along the same row, column, left diagonally, or right diagonally. The goal of solving this problem is to find all possible solutions or a single solution such that no queens occupy the same row, column, or diagonal.

There are many ways to approach this problem to find its solution. This document covers some artificial intelligence approaches with heuristics, that being uninformed search and local search. More specifically, breadth-first search (BFS), hill-climbing search, and simulated annealing (SA) search.

Uninformed search

Uninformed search is a blind approach to solving the n-queens problem to find all possible solutions. This strategy may cover all possible positions each queen can be placed on the board ($(\frac{(n \times n)!}{(n \times (n-1))! \times n!}$) (where *n* is the number of queens)) to find every goal state. The search is accomplished by using a tree data structure, where the initial state is an empty board and branches off the root along every possible path, finding all branches and leaves. However, there are many different approaches using this method.

Breadth-first search (BFS)

Breadth-first search's strategy to solving the n-queens problem is that an agent travels across all depths of the tree finding each nodes' children before travelling to the next layer. In n-queens, the initial state (root node) is an empty board, *n* children stem off the root node creating all possible children given the agents constraints. The next step is removing the root node, the roots children then become the parent of their own child nodes, each child branches off *n* children. This process repeats until the agent is *n* layers deep. Once the agent reaches the end of the tree, it must check if one of the children is a goal state using a goal checking function.

BFS is implemented for the n-queens problem by traversing down the tree placing a new queen row by row for each layer of the tree, different columns for each child, such that no two queens occupy the same row and each child is different to its siblings. By doing this, it cuts down the possible placements of queens to $\sum_{i=0}^{n} n^i$ (where *n* is the number of queens).

The primary data structure in the code is a queue of a vector of 8-bit integers. A first-in, first-out (FIFO) queue is necessary when implementing BFS, this allows for quick popping of parent nodes and quick pushing of children. A one-dimensional vector of 8-bit integers is used to represent the chess board. The index of the vector represents the row, and the value in each index represents the

occupied column. A vector is dynamically allocated to ensure minimal space is occupied and an 8-bit integer is of one byte in contrast to a regular 32-bit integer being four bytes.

Pruning BFS for the n-queens problem significantly improves performance. The steps involved in pruning this solution is simple; create a basic boolean valid move checking function that checks if a move is valid for a given child nodes queen placement, if the function returns true, the child is added to the queue, if false, the child is not created and never continues down that child's path. In the big picture this saves a great deal of memory space and vastly speeds up the algorithm. Table 1 and 2 illustrates this difference in the time elapsed for each n value of BFS and BFS with pruning.

Table 1: Breadth-first search without pruning

Number of queens (*n*)	Number of solutions	Time elapsed (in seconds)
1	1	0.0003
2	0	0.0006
3	0	0.0009
4	2	0.0028
5	10	0.0259
6	4	0.0251
7	40	0.2375
8	92	4.9644

Table 2: Breadth-first search with pruning

Number of queens (*n*)	Number of solutions	Time elapsed (in seconds)
1	1	0.0004
2	0	0.0006
3	0	0.0009
4	2	0.0027
5	10	0.0249
6	4	0.0125
7	40	0.0020
8	92	0.0026
9	352	0.0043
10	724	0.0103
11	2,680	0.0399
12	14,200	0.1950
13	73,712	1.1300
14	365,596	6.7162
15	2,279,184	43.7340
16	14,772,512	875.9430

The system used to run the algorithm is only capable of getting to $n = 8$ without pruning and $n = 16$ with pruning due to lack of memory space. For example, without pruning, $n = 9$, the memory needs to store at least $9^9 \times 9$ queen states.

To predict the approximate number of solutions or time taken to achieve $n = 30$, a graph is generated in Microsoft Excel for the time elapsed and number of solutions. The graph requires an exponential trendline for the growth rate, to do this, Excel can generate the growth rate formula.

BFS without pruning, predicting $n = 30$:

Time: $y = 0.0004e^{1.7177n}$ (where n is the number of queens)

$y = 0.0004e^{1.7177\times30}$

$\approx 9.5871 \times 10^{18}\ seconds$

Solutions: $y = 0.3088e^{0.6336n}$

$y = 0.3088e^{0.6336\times30}$

$\approx 55{,}558{,}025\ solutions$

Local search

Local search in n-queens starts off with a randomly generated chess board. Each local search algorithms agent has its own set of constraints on choosing the next move to be made. The primary difference for n-queens in local search and uninformed search is that unlike uninformed search being effective at finding all possible moves, local search is effective at finding a single solution.

Hill-climbing search

The goal of hill climbing is the algorithm continually chooses the best possible next move. Hill-climbing search's strategy to finding its goal state in n-queens is simple: first, a random board state is generated, with this board state the agent is to find all its neighbouring board states, that being each queen moves to every position on the current board. The neighbour with the lowest number of conflicts is chosen as the agents following state. This repeats until the number of conflicts between all queens is zero. The primary issue with this algorithm is that it can get stuck at a local maximum. If no move can be made from the boards current position that has less conflicts than the current board state, this is a local maximum. To correct this issue, the algorithm simply regenerates a random board to start again until a board of $n \times n$ size is solved. There is a random element to this algorithm which can affect the time taken to complete.

Table 3: Hill-climbing

Number of queens (*n*)	**Time elapsed (in seconds)**
5	0.0034
10	0.0075
15	0.0094
20	0.0202
25	0.5632
30	0.0716
35	2.0663
40	5.4796
45	4.2277
50	5.5722
55	7.1761
60	16.7508

Simulated Annealing

Simulated annealing's search strategy is like hill-climbing, however, allows for downhill movement when stuck at a local maximum. It starts off by generating a random board, the agent then chooses the next position randomly from the neighbourhood, evaluates the number of conflicts. If the number of conflicts if less than of the current state, it moves to the next state, however, this is where the algorithm can get a little complicated. The simulated annealing uses what's called a p value ($p = e^{(-(current\ state\ conflicts - next\ state\ conflicts)\ /\ temperature)}$) to determine the chance of making a downhill move. The closer p is to 0, the higher the chance is that the agent moves to the next board state, even if the next state has a greater number of conflicts. The temperature is set to n^2 where it is decreased by a given cooling rate of 5%. As the temperature approaches 0.001 the chances of the algorithm moving downhill decreases. There is a random element to this algorithm which can affect the time taken to complete.

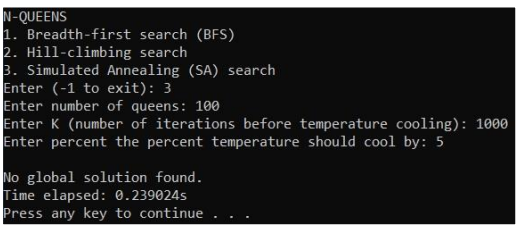
The algorithm has a K value, the K value determines the time of how long the algorithm runs. Higher K values (e.g. $1{,}000{,}000{,}000$) gives the algorithm a more consistent chance of finding a solution for *n* when the value for *n* is high with the trade-off of being slower. When K is set to a low value (e.g. $1{,}000$) the algorithm cannot consistently find high *n* values but is much faster.

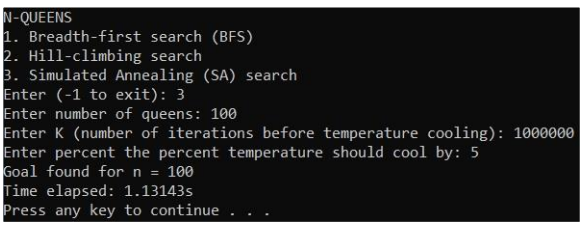
Table 4: Simulated annealing

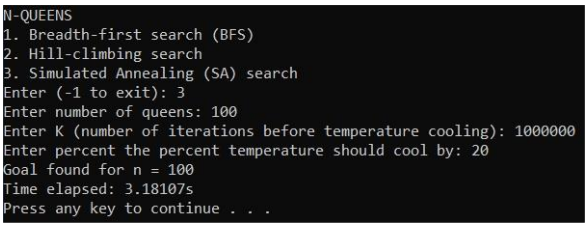
Number of queens (*n*)	Time elapsed (in seconds)
10	0.0015
20	0.0073
30	0.0195
40	0.1208
50	0.0972
60	0.2115
70	0.4773
80	0.5751
90	0.7742
100	0.9877
110	1.7298
120	2.2910

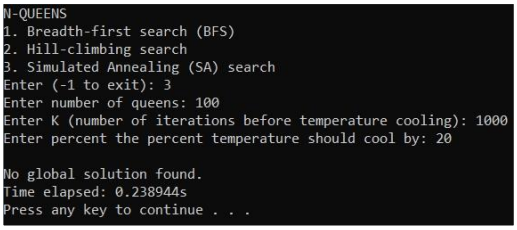
Comparing Hill-climbing and Simulated annealing search

Given the two local search algorithms, it stands out that simulated annealing performs drastically better than hill-climbing ((*table 3*) and (*table 4*)). Due to its ability to manoeuvre downhill when necessary, allowing it to quickly find the global maximum as opposed to hill-climbing requiring to do a random restart when it reaches a local maximum.









As it shows, higher K values give more consistent results. And reducing the temperature by more after each K iterations takes longer to find the goal. Depending on the algorithm, constraints, and implementation, the user much find a sweet spot in which attains to their needs.