2802ICT Programming Assignment 1

N-Queen Problem

By Candice Smith
S5168666

The N-queen is the problem where one must place N number of queens on a chessboard the size of N*N and the queens must be placed in a configuration such that no queen can attack another. This means that no two queens can be placed on the same row or column, or diagonal to each other. Given the number of queens, N, the problem requires one to find all distinct board configurations where the number of attacking queens is zero. There exist solutions to N-queens where N = all natural numbers excluding 2 and 3.

There exist many algorithms and heuristics that can solve the N-Queen problem, some more effective and optimal than others. Some commonly used algorithms include breadth-first search, depth-first search, backtracking, genetic algorithm, simulated annealing, and hill climbing. This report examines several of these, the uninformed search algorithm – breadth first search and breadth first search with pruning – and the local search algorithms – hill climbing and simulated annealing.

Part A – Uninformed Search

Uninformed search, also known as blind search or brute force, is an algorithm that produces a search tree exploring every possible solution without any specific knowledge or heuristic to guide it towards the goal. Examples of uninformed search are breadth-first search, depth-first search, depth-limited search, depth-first iterative deepening, and uniform cost search. The uninformed search algorithm used is breadth-first search.

Breadth-first Search (BFS)

BFS generates a search tree by always exploring the shallowest node first. This is implemented using a queue; when the first node (empty board) is explored its children (a queen placed in each spot on the first row of the board) will be pushed to the queue, and before exploring the children the next node in that layer will be explored, subsequently pushing its children to the queue. Once a node is explored it is popped from the queue. This process is repeated until the last layer is reached. For the N-queen problem the last layer in the tree has only boards with N number of queens on the board and the queue contains all possible solutions. The possible solutions are sent to the evaluation function to produce a fitness score. A solution is found when the fitness score is zero; i.e. the total number of attacking queens is zero.

BFS is memory heavy, without pruning the number of nodes stored at each depth grows exponentially. Minimising memory usage is vital for solving for a higher number of queens. A 1-dimensional vector is used where each value in the vector is the column the queen is placed, and the corresponding vector index value is the row the queen is placed. The following is a breakdown of the performance of each implementation – BFS without pruning and BFS with pruning – to show the difference in the time taken to find all solutions to the N-queen problem.

Table 1: BFS without pruning

N (number of queens)	Solutions found	Time (seconds)
1	1	1e-07
2	0	N/A
3	0	N/A
4	2	0.0069814
5	10	0.357104
6	4	0.212954
7	40	0.332186
8	92	6.07999

Table 2: BFS with pruning

N (number of queens)	Solutions found	Time (seconds)
1	1	1e-07
2	0	N/A
3	0	N/A
4	2	0.0093756
5	10	0.0383576
6	4	0.0461371
7	40	0.0002828
8	92	0.0010471
9	352	0.0043887
10	724	0.0193324
11	2,680	0.0808934
12	14,200	0.290704
13	73,712	1.53587
14	365,596	9.13993
15	2,279,184	79.7848
16	14,772,512	2103.44

Due to RAM constraints the solutions for N > 8 is not recorded for BFS without pruning. With the values taken from BFS without pruning larger N values can be predicted using a graph. A graph is produced in Excel for the time taken and solutions produced (Figures 1 & 2). Adding an exponential trendline in Excel the function for the growth rate of both times taken and solutions found can be generated. This function is then used to predict the time taken and number of solutions when N = 30.

BFS without pruning:

$$\begin{aligned} solultions &= 0.3088e^{0.6336N} \\ &= 0.3088e^{0.6336*30} \\ &\approx 55{,}558{,}025\ solutions \end{aligned}$$

$$\begin{aligned} time\ taken\ (seconds) &= 8E-08e^{2.4301N} \\ &= 8E-08e^{2.4301*30} \end{aligned}$$

$$\approx 2.733 \times 10^{24}\ seconds$$

Based off the values for each N value found it can be predicted that when N = 30 there exists 55,558,025 solutions and the time it takes to find all solutions would be approximately 2.733 x 10^{24} seconds.

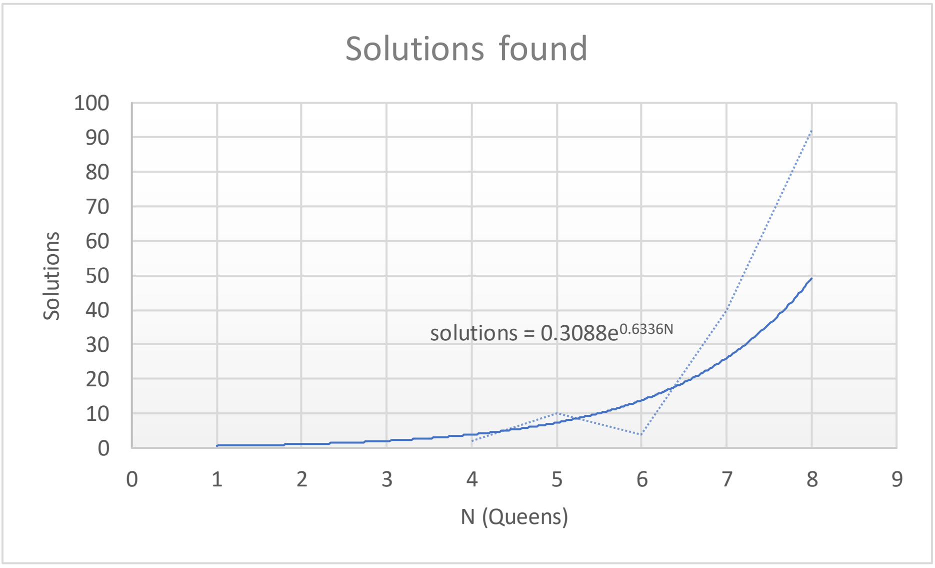

Figure 1: BFS without pruning solutions

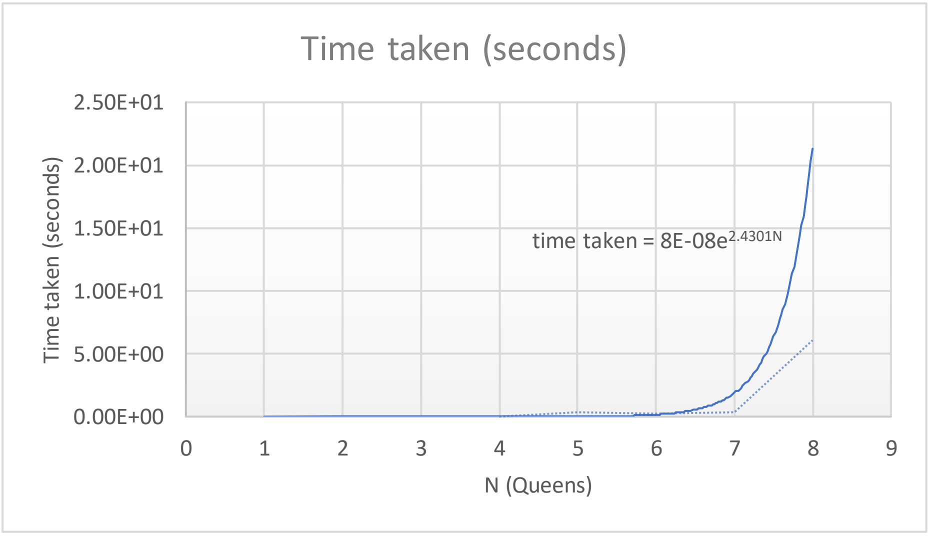

Figure 2: BFS without pruning time taken

BFS on its own is very inefficient as it will produce and store every possible board configuration including when there are less than N queens on the board. This is extremely taxing on the system memory and can be addressed using pruning. Pruning is essentially pruning off branches of the search tree that will not lead to the goal. When a node is created it is checked for any attacking queens, if it finds any it will not store that node in the queue, effectively cutting off that branch and not allowing its traversal. This cuts down the number of nodes that will be stored in memory, making it able to find solutions to a higher N value.

Part B – Local Search

Local search algorithms start with a randomly generated initial state and moves to another state based on some predefined cost function. Unlike other search algorithms, like BFS and greedy, local search algorithms do not guarantee a solution. However, because of the use of some heuristic they are more efficient at finding or coming close to a solution.

Hill Climbing Search

Hill climbing search is a local search algorithm which uses the heuristic of when deciding the next move always chooses a neighbouring node which is closer to the goal, continuously moving up the mountain to reach the global maximum. This uses an evaluation function which calculates a fitness score – how many changes need to be applied to the board (in the case of N-queen problem) to reach a goal state (fitness score of zero) – to determine how far up the mountain the node is. The problem with this algorithm is it often gets stuck in a local maximum where it reaches a node which is not the goal, but its neighbours are further away from the goal. To address this hill climbing with random restart is used. If a local maximum is reached and it cannot move further up the mountain a new board is randomly generated and the search starts again – this is repeated until it reaches the global maximum.

Hill climbing relies on an initial state which will lead to the global maximum and will not lead to a local maximum or plateau (all neighbouring nodes of equal fitness score to the current node). Therefore, combining hill climbing with random restarts is much more effective at finding a solution than hill climbing alone. Hill climbing can find a solution for at least N = 50. Due to time constraints no higher N value was tested. Since hill climbing is not constrained by a predetermined number of iterations it can be run for infinite time and therefore can possibly find a solution for a very large range of N values. The times recorded (Table 3) are somewhat inconsistent when compared with BFS because it depends largely on the initial configuration.

Table 3: Hill climbing time taken

N (number of queens)	Time (seconds)
1	1e-07
2	N/A
3	N/A
4	0.0323625
5	0.0417959
6	0.0923809
7	0.0003373
8	0.0001832
9	0.0005965
10	0.0022337
11	0.0057167
12	0.0012646
30	10.5219
31	5.28376
50	19.8415

Simulated Annealing Search

Simulated annealing search is an algorithm like hill climbing but it will allow some movement downhill which addresses the problem of getting stuck in a local maximum. Given a random initial board configuration a neighbour node is generated randomly which is sent to the evaluation function to calculate its fitness score. If the fitness score of the neighbour node is greater than the current node it will accept this move. If it is less than the current node it will also accept this move but with a probability function relative to how close the current and neighbour nodes are to the goal. This probability is calculated using the formula $p = e^{-(current\ node\ fitness\ score - neighbour\ fitness\ score)/T}$, where T = the current temperature. The temperature is initialised at 100, which is slowly cooled by a rate of $1 - \alpha$ where α is 0.01. This means that as T decreases the probability of the algorithm choosing a downhill move decreases.

The algorithm is run K times, meaning that the value K is set to initially will affect how long the program will have to find a solution. When K is set to a very large number the algorithm has a higher chance of producing a solution to a larger N value. When K = 1,000,000,000 the algorithm can produce a solution for up to ~N = 32 consistently in ~125 seconds. When K = 10,000 it can only consistently get to ~N = 15. Much like hill climbing, simulated annealing depends heavily on the initial board generated. This means the times it takes to find a solution can vary particularly for large N values.

Based on the results produced BFS with pruning drastically improves the efficiency of BFS finding all solutions to the N-queen problem. Due to the memory consumption of BFS it is very difficult to find solutions to high N values. The hill climbing algorithm performs better than both BFS and BFS with pruning in both time and memory use but does not guarantee a solution. Simulated annealing also performs better than BFS and BFS with pruning and seems to produce faster results than hill climbing but could not reach a N value as high as hill climbing but also cannot guarantee a solution. Due to the constraints of the K value for simulated annealing, hill climbing seems to be more consistent at producing a solution. Each algorithm has its strengths and weaknesses, but due to the heuristic nature of hill climbing and simulated annealing they overall seem to be more optimal algorithms for solving the N-queen problem.