2801ICT – Assignment 1

Table of Contents

Introduction

This report will be presented in two parts: Part A will be an investigation into using a breadth first search algorithm to find all solutions to the *n*-queens problem for *n* = 1 to *n* = 20. The results from these approaches will be used to approximate the results for *n* = 30.

Part B will be a comparative analysis of find just one solution to the *n*-queens problem for *n* = 1 to *n* = 30 using a random-restart hill climbing search algorithm and a simulated annealing algorithm. The problem is to place *n* number of queens on a chess board of size *n*×*n* without any of them attacking each other.

Although different algorithms and different functions are used throughout the analysis there are some data structures and functions that remain constant. A description of these will be given now and will be referred to in the coming sections of the report.

The Board

It was decided to use a tuple of integers where each value shows a queen's column and the index of that value in the tuple is its row. *Figure 1* shows a board using is "physical" programmatic tuple representation above its logical depiction. If B is a tuple with elements ($column_1$, $column_2$, $column_3$) then the co-ordinates of a queen as (row, column) is equal to B[row] = $column_n$.

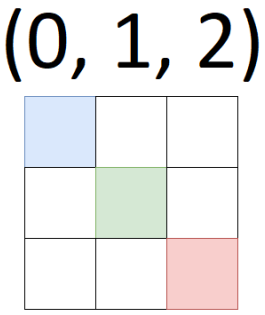

***Figure 1**: A board as a tuple and how it can be imagined.*

Checking for Attacks

A queen can attack any other queen on the board in three different ways; horizontally, vertically, and diagonally. This can be checked easily with simple mathematics. For each queen in the board it is checked against every other queen using the following techniques:

- If two queens are on the same row, then the difference (Δ*x*) in the value of those row positions will be equal to 0. If queen *A* is at (2, 1) and queen *B* is at (2, 0) then they are attacking one another as 2 – 2 = 0
- The same logic can be applied to two or more queens in the same column. If the change in columns, Δ*y*, is equal to 0 then an attack is happening
- To find a diagonal attack, both of the above points need to be combined. If the absolute difference in *both* of the queens' rows and columns, Δ*x* and Δ*y,* are equal then the change in position from queen *A* to queen *B* is a constant and therefore the two are attacking each other.

Goal State and Heuristic Cost Functions

Although a checking for a goal state and calculating a heuristic cost are different things they are implemented in a very similar way. Put simply, checking for a goal state is checking whether there are any queens that are being attacked. If no queens are being attacked, then the current state is a goal state. Calculating the heuristic cost of a state works in the same way apart from a small alteration; the *number* of queens that are being attacked is recorded.

Part A

Part A will be broken into two subsections, the first of which will explore the implementation and results of a breadth first search (henceforth referred to as B.F.S.) algorithm without any pruning of the search tree. The second subsection will focus on the same algorithm but with some pruning of the search tree considered as well.

B.F.S. – Unpruned Approach

Implementation

In this version of the search tree, every single possible node will be explored (but not more than once). Hence no pruning or "thought" goes into how the agent moves through the state space to find a goal state. For each *n* there will be

$$\frac{n!}{r! \times (n-r)!}$$

states in the state space to explore and some nodes in that search tree will generate the same children as other nodes. Therefore, a list of explored states is used to make sure that no time is wasted on checking the same states repeatedly. As per *Figure 2*, a child is generated from a current node by moving *each* queen into *every* available space on the board (i.e. one that is not currently occupied or is not the space that it started from). Without considering that some of these child states will have been explored, each state will generate the following number of children for *n:*

$$n^2(n\text{-}1)$$

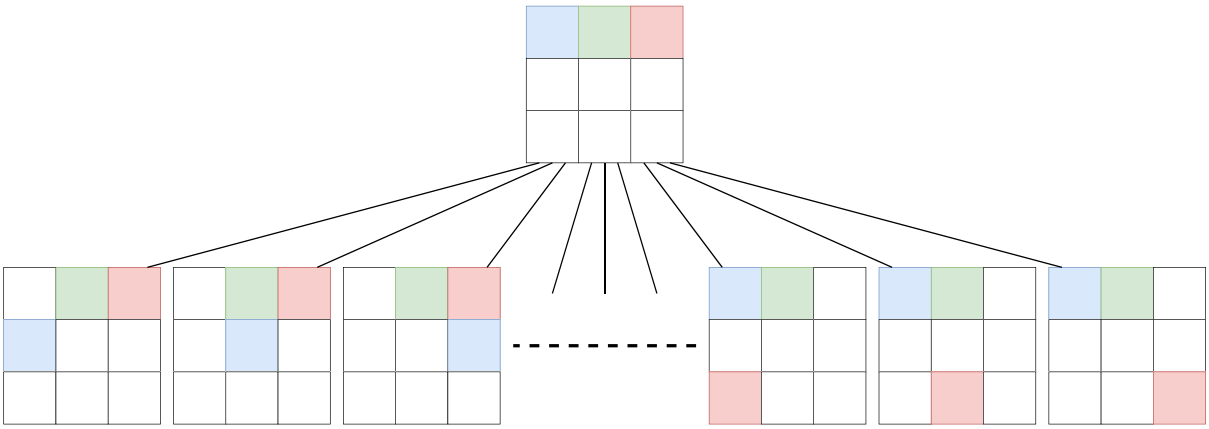

Figure 2*: A diagram of how a parent state will generate its children with a board size of n = 3. Each queen is represented by an individual colour.*

Lastly, a queue called the *frontier* is used so that the agent can move through the state space tree in an ordered fashion. A "first in, first out" (FIFO) approach was used for the frontier because this allows for a search to be conducted for each child in a particular level of the search tree. In other words, for each level of the tree its entire *breadth* is searched before moving on.

Results

Although the curve is less severe in *Figure 4* than *Figure 3* there is a clear correlation between the number of nodes that the B.F.S. algorithm is checking, and the time taken to perform it in its entirety for each *n* number of queens. One of the goals of this testing was too reach a value of *n* = 20 but one can see from *Figure 4* that running the algorithm for this length of time is not feasible and hence it was stopped somewhere between *n* = 8 and *n* = 9.

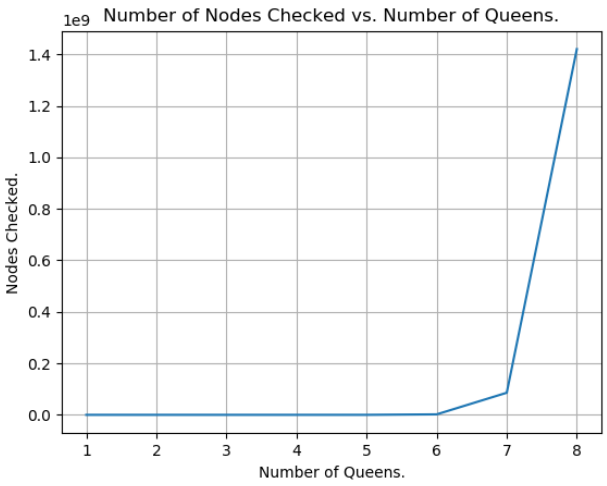

Figure 3

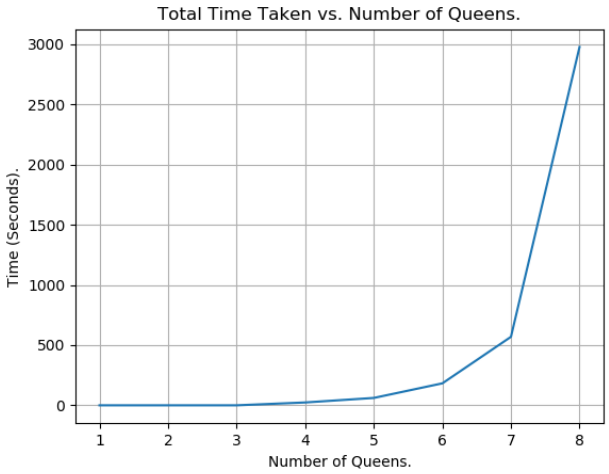

Figure 4

In any case, the algorithm performed as intended; the correct number of solutions were found for each of the board sizes that were able to be run.

Number of Queens	Solutions Found
1	1
2	0
3	0
4	2
5	10
6	4
7	40
8	92

Given these results we can conclude that given the right amount of time and memory the B.F.S. algorithm will be able to accurately generate all solutions for *n* = 30. Therefore, although incomplete, the results were used to perform a 4th order polynomial regression to predict the time taken to solve *n = 9* to *n* = 30.

$$\mathbf{y = 12.941x^4 - 190.99x^3 + 962.24x^2 - 1873.7x + 1121.4}$$

4th order was chosen because that gives an accuracy of 0.9909% (*Figure 5*) .

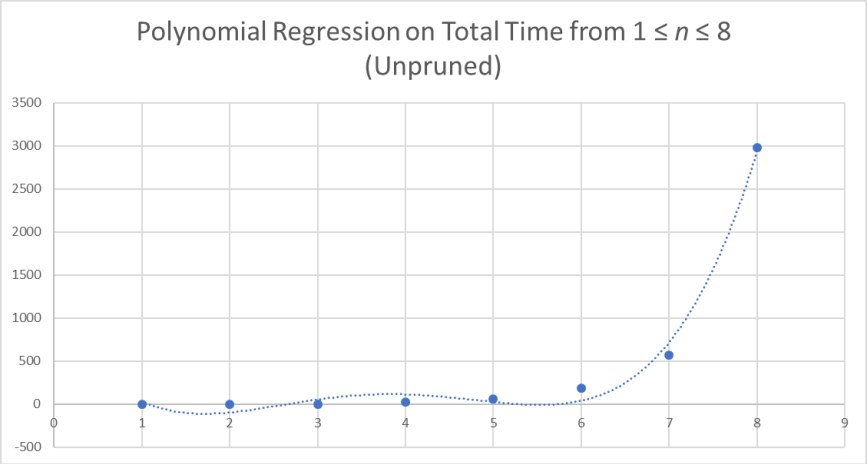

Figure 5

Figure 6 uses the function, generated by performing this regression to predict what further results would look like. It is made quite clear that using an un-pruned B.F.S. approach for this problem is extremely inefficient.

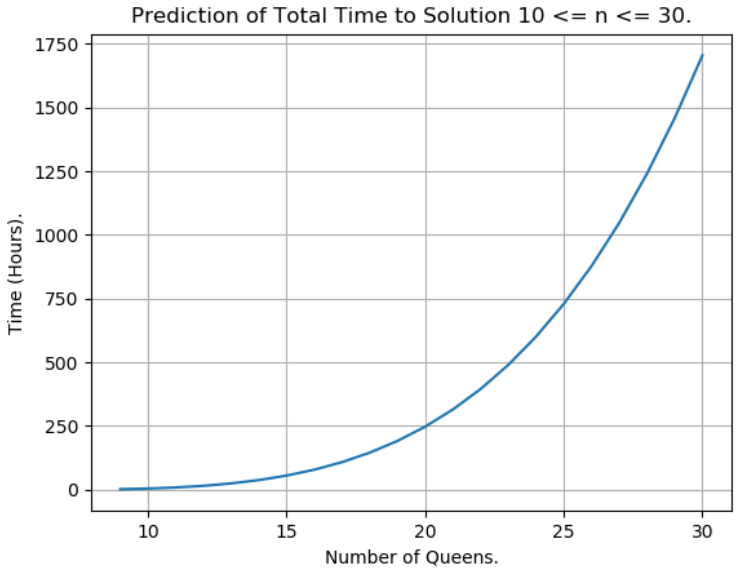

Figure 6

B.F.S. – Pruned Approach

Implementation

Three ways of pruning were considered in this approach and two were applied. When creating each from the initial state, it was ensured that no new child would be placed into a row or column that is already occupied by another queen. The state space tree is generated by starting from an empty board and placing one queen at a time, row by row, until each row is being occupied by only one queen at each level. As they are being placed, the algorithm also makes sure to place a new queen in a column that has no queens inhabiting it. Hence, each child of a node will have $n + 1$ queens and the depth of the tree will be no greater than $n + 1$. This process is shown in *Figure 7*.

Worth noting is pruning of diagonal clashes was not applied. This was because logically the end of each path in the state space tree would just yield a goal state and no real searching of its breadth would take place. Therefore, the search could be further sped up with this.

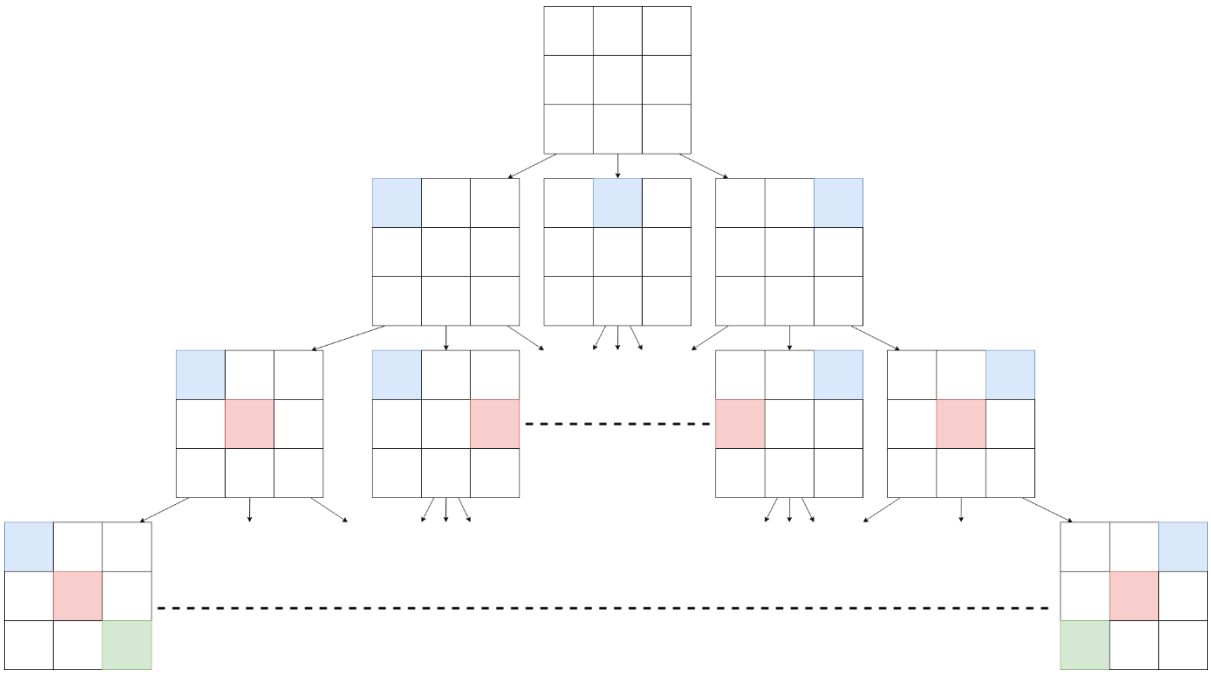

***Figure 7**: How the pruned search tree is generated for the pruned approach.*

Another small change was made to the function to check for a goal state; in this case, if the number of queens present on the board is not equal to n (i.e. not all queens have been placed) then it is immediately not a goal state and the queen positions aren't checked. This further increased the speed of finding results.

Results

Pruning the tree allowed the B.F.S. algorithm to perform significantly better (*Figure 8*). Achieving the goal of reaching *n* = 20 was still not feasible, however improvements were made.

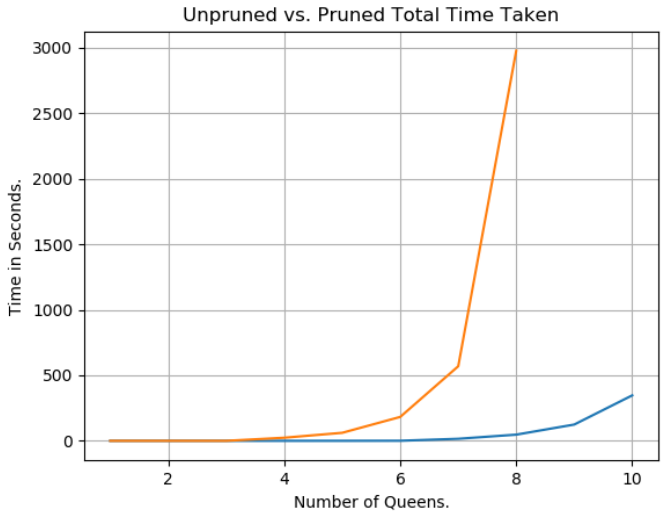

Figure 8: *Showing the difference in time taken between an unpruned (orange) and a pruned (blue) version of B.F.S.*

Another notable difference is in the count of nodes that were generated/checked as a goal state using pruning (*Figure 9*). This is an even greater change that shown in *Figure 8*.

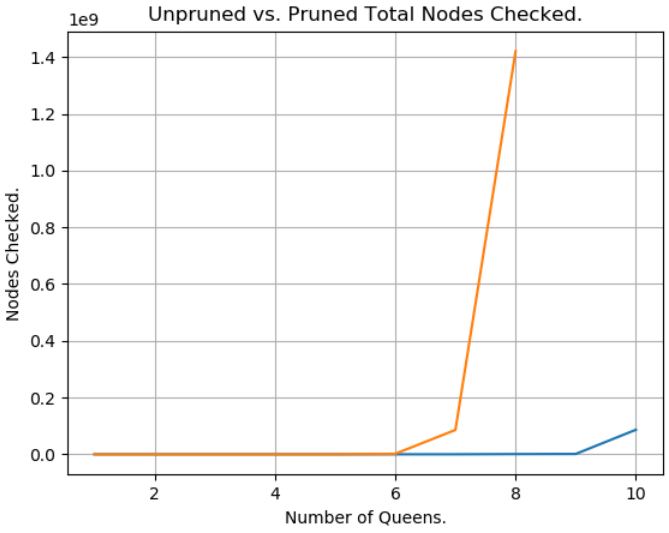

Figure 9: *Showing the difference in nodes generated between an unpruned (orange) and a pruned (blue) version of B.F.S.*

The same (accurate) results for each queen were given when testing for all of the possible solutions for *n.* Therefore, another polynomial regression was performed to predict the amount of time it would take to reach all possible goal states for *n* = 30. With an accuracy value of 0.9954, the following 4th order polynomial is used (plot shown in *Figure 10*):

$$\mathbf{y = 0.4031x^4 - 6.7917x^3 + 39.099x^2 - 86.5x + 57.26}$$

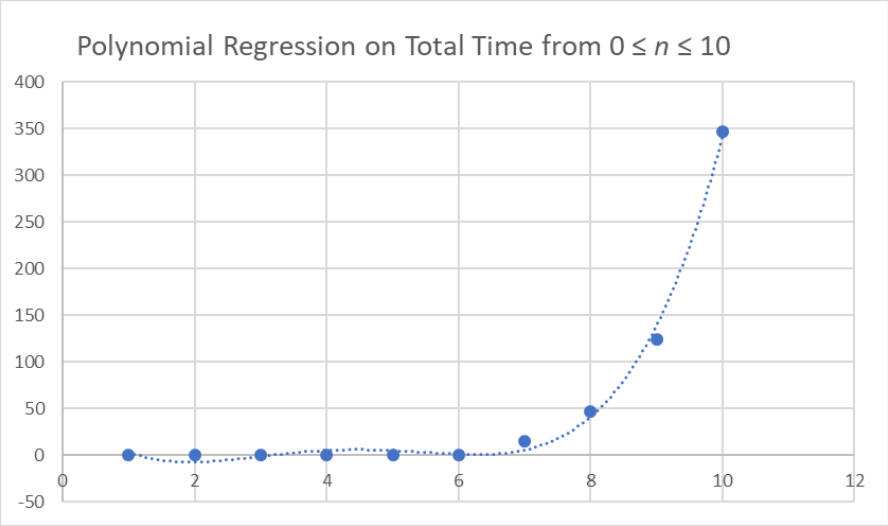

Figure 10: *Pruned regression analysis.*

Although how long each board size takes are radically different, the trends in the predicted regression functions between the unpruned (*Figure 6*) and pruned trees are strikingly similar (*Figure 11*).

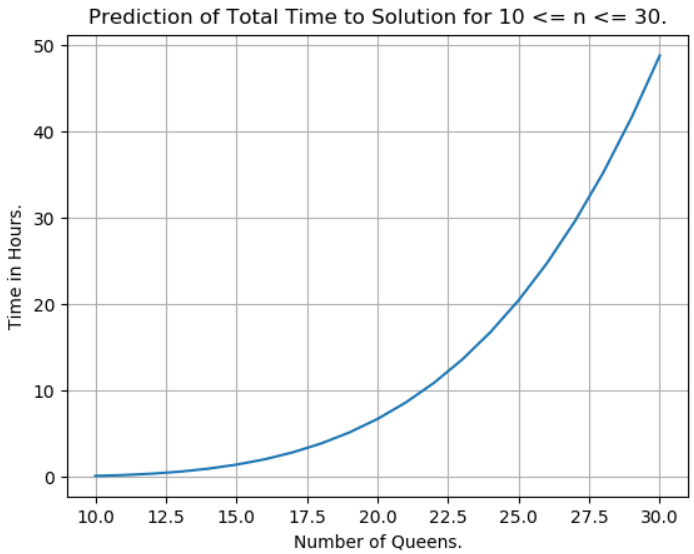

Figure 11: *Pruned regression.*

Conclusion

Solving this kind of problem using breadth first search is not ideal. Although time and memory aren't of concern for very low counts of queens there is a steady increase in time and space needed to accurately calculate every possible solution for $1 \leq n \leq 30$. Speeding up the process by pruning the state space tree had significant results, however, and this should be taken into consideration upon deciding whether to use B.F.S. Whether pruned or not, breadth first search has proved to be inefficient for anything other than the most trivial of tasks.

Part B

Now for a comparative analysis on solving for one solution of the *n* queens problem using simulated annealing and random restart hill climbing algorithms (henceforth referred to as S.A. and H.C., respectively).

Implementation

Both algorithms seek to find a solution by "climbing" down to states with lower and lower costs (based on the heuristic cost function mentioned earlier in this report). H.C. is a blind, unnuanced approach to this philosophy and will move to the very next state that has a better cost than itself. It never looks back, it doesn't accept anything that has a worst cost than itself, and if it cannot find a path downwards it just gives up and resets itself. Failure is inherent in such an algorithm and we will see just how often it fails and restarts because no "better" states were found. By its very nature, it will get stuck in areas that are a local minimum, but not a goal state.

S.A., on the other hand, adds an important step to this greedy method. It allows itself to make "uphill" moves to states that have a higher cost than the current state. This permits the agent to move up and down local minima and maxima with a better chance of reaching an optimal solution eventually. We will see how this small addition changes the way solution states are found from initial states.

Results

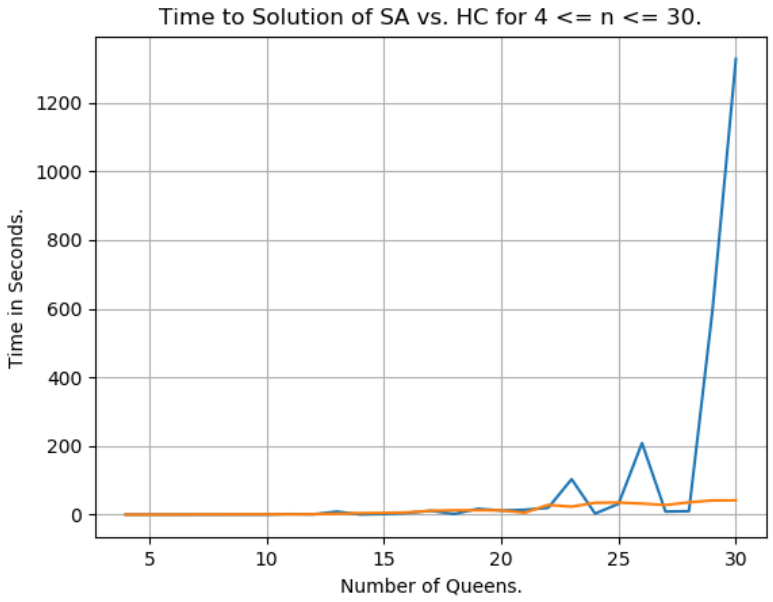

Figure 12: *The difference in how long it takes for a solution to be found in S.A. (orange) and H.C. (blue).*

As *Figure 12* demonstrates, the difference in *n* for how long it takes using H.C. can be very erratic. With *n* = 24 and *n* = 27 taking closer to 30 seconds to find a solution, *n* = 26 took just over 200 seconds. There is a definite exponential trend towards the larger board sizes, however. This is in start comparison to S.A. which does increase over time but with a very steady and slight curve.

One may be able to see why H.C. has such unpredictable behaviour when comparing it's running time, verses the amount of times it has had to stop and restart itself (*Figure 13*).

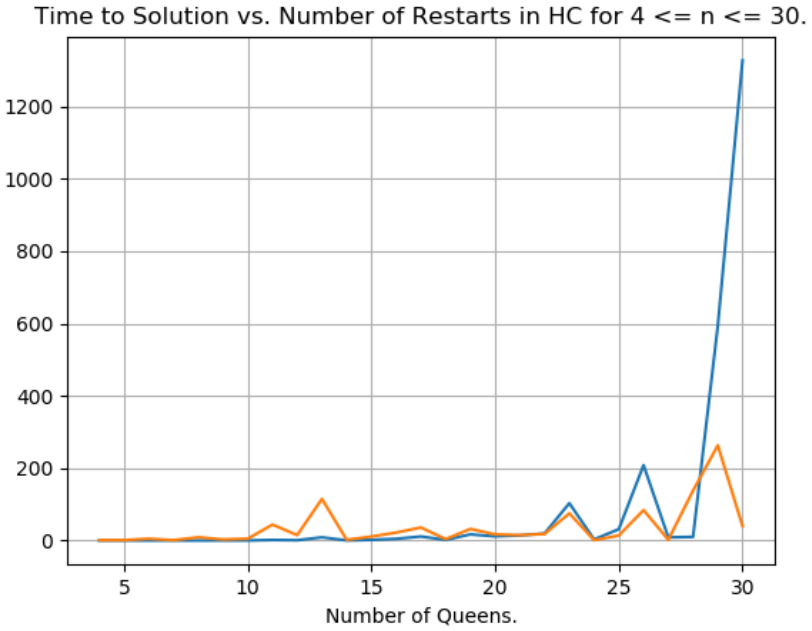

Figure 13: *The trend in number of restarts (orange) and how long the algorithm took to find a solution (blue) for H.C.*

There is a clear relationship between the amount of times H.C. had to restart and the length of time it took to finally arrive at a solution. Although the rises in time to solution are sharper, for each of them there is a correlated rise in the number of restarts that round of H.C. had to take.

This is also apparent when examining the relationship between the number of restarts a value of *n* had to take and how many overall moves it made before finding a solution. Again, almost every dip or rise in the trend in the number of restarts made for a value of *n* has a direct effect on the number of states it had to traverse to find a solution (*Figure 14*). Although, much more severe.

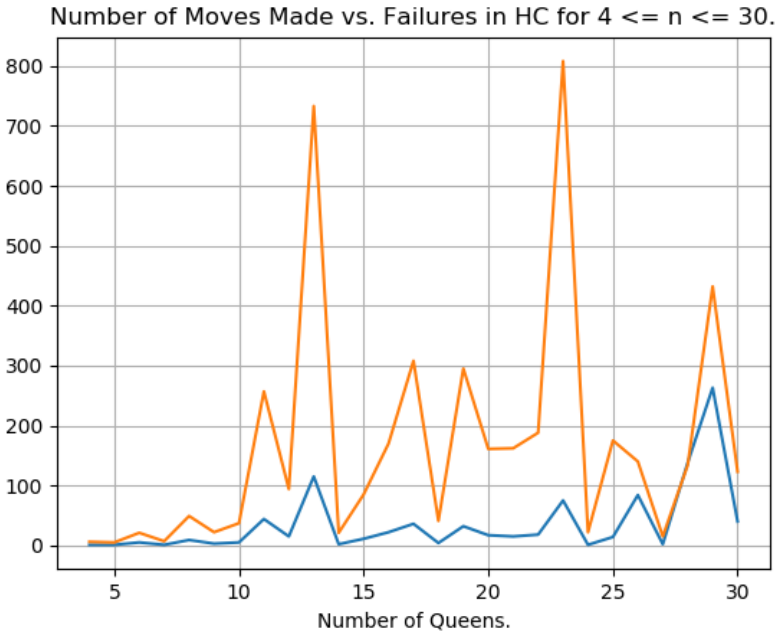

Figure 14: *How the number of restarts (blue) effects the amount of states traversed (orange) in H.C.*

The volatility of H.C. becomes even more obvious when relating the amount of "moves" it does before finding a goal state with the same metric in S.A, as seen in *Figure 15*. A "move" is when the agent verifies a neighbouring state as a valid change and transfers to it. As with comparing the amount of time taken for both algorithms, here we see a similar trend. H.C. is completely irregular with almost no clear overall tendency while S.A. is much steadier and clearer in which direction it is heading.

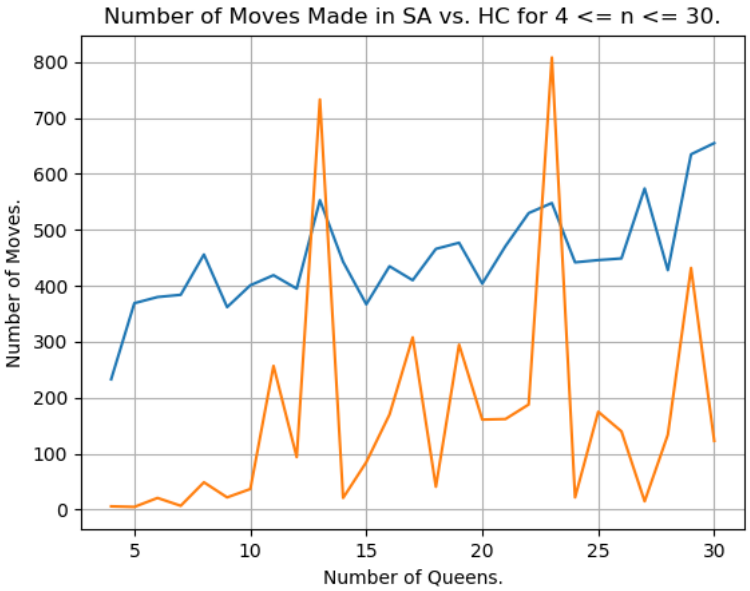

***Figure 15**: Difference in moves with S.A (blue) and H.C. (orange).*

Conclusion

If all parameters are right and H.C. doesn't have to restart itself many times, it is very fast and actually quite reliable in the result it gives. But it is seldom perfect. Inherent in its design is failures, long stretches of moves, and unclear running times. A steadier approach to the problem in using S.A. ensures that each step is more consistent in the way it works towards a goal. A further nuanced method that ensures more uniformity, but at the cost of greater memory usage per *n*, is easier to foresee solutions with and analyse.