Intelligent Systems 2802ICT

N-Queens Problem

Jesse Schneider

2020

Contents

[Problem Overview 2](#_Toc35893438)

[Part A – Uninformed Search 2](#_Toc35893439)

[Algorithm Design – Breadth-First Search 2](#_Toc35893440)

[Pseudocode – BFS Search 2](#_Toc35893441)

[Results 3](#_Toc35893442)

[Pruned BFS 3](#_Toc35893443)

[N = 30 - BFS 4](#_Toc35893444)

[Part B – Informed Search 5](#_Toc35893445)

[Hillclimb 5](#_Toc35893446)

[Algorithm Design – Hill-climb 5](#_Toc35893447)

[Pseudocode – Hill-climb 5](#_Toc35893448)

[Results 5](#_Toc35893449)

[6-Queen Output – Hill-climb 6](#_Toc35893450)

[Simulated Annealing 7](#_Toc35893451)

[Algorithm Design – Simulated Annealing 7](#_Toc35893452)

[Pseudocode – Simulated Annealing 7](#_Toc35893453)

[Results 7](#_Toc35893454)

[6-Queen Output – Simulated Annealing 8](#_Toc35893455)

[Informed Search – Hill-climb vs Simulated Annealing Summary 9](#_Toc35893456)

[Simulated Annealing Parameters 9](#_Toc35893457)

# Problem Overview

The point of this report is to report on the findings of using different searching strategies to try and solve the N-Queens problem – both uninformed and informed strategies have been used.

The N-Queens problem is a chess problem – it asks the question: how many unique ways can a number of queens (*n* number of queens) be placed on a chess board that is *n* x *n* in size, where they are unable to attack each other? (i.e. how many ways can 8 queens be placed on an 8 x 8 chess board where they cannot attack each other?). For context – a queen can attack vertically, horizontally or diagonally any number of spaces – so each queen must have a clear horizontal, vertical and diagonals.

# Part A – Uninformed Search

### Algorithm Design – Breadth-First Search

To perform the uninformed search on the N-Queens problem, a Breadth First Search (BFS) algorithm has been implemented. This is a well-known search strategy for uninformed searching.

The ultimate strength of the BFS – the fact that is will always deliver the smallest cost – is lost on this problem, as it is not stopping upon finding a solution; rather it is continuing until all possible solutions have been exhausted. This does not make it any less valid to use though – cost is just simply irrelevant.

### Pseudocode – BFS Search

The below code does not relate to the class is the source code – rather only the BFS implementation:

Number of solutions = 0

add starting board to explored list

create a queue

Add starting board to queue

while queue is not empty:

get next value from the queue

for each column in the board:

if our current row is >= n:

skip

copy our current board

add our new queen to column

store new queen coordinates

if this board has already been explored:

skip to next board

create a new board -> pass in current board and its related details

if the current row = n:

check if board == goal state

if goal state:

number of solutions ++

put the board back into the queue for further searching

add the current board to explored list

return the number of solutions

### Results

Here are the average running times for N Queens 1 – 7:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| N Queens | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| Time (seconds) | 0.00 | 0.001 | 0.001 | 0.012 | 0.184 | 3.124 | 69.694 |
| Number of Solutions | 1 | 0 | 0 | 2 | 10 | 4 | 40 |

(These averages are across 5 runs)

To solve the 8 x 8 board for 8 Queens using the BFS search method takes approximately 27 minutes for my hardware to complete – but it did complete successfully.

Because of this, I was unable to continue testing due to time constraints on the assignment.

# Pruned BFS

To prune the BFS – I added 2 new lines of code:

If board already has collisions:

skip to next board permutation

The thinking behind this is: If 1 or more queens are already clashing, there is no way a further branch of this board is going to equal a goal state. This removed some redundant branches from ever being placed back on the Queue – entire useless branches are now ignored.

The result of this knocks nearly 20 seconds off the 7 Queens solution!

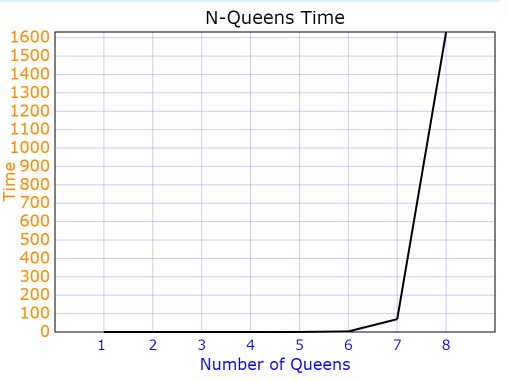
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| N Queens | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| Time (seconds) | 0.00 | 0.0 | 0.0 | 0.010 | 0.145 | 2.358 | 50.120 |
| Number of Solutions | 1 | 0 | 0 | 2 | 10 | 4 | 40 |

With the number of solutions matching to the previous solutions, it is safe to assume this pruning technique is still providing correct solutions – in a reduced time period.

This proves that the branches of the search tree that we are now skipping are completely redundant.

## N = 30 - BFS

Here we can see the relation of time to number of queens is exponential up to 8 queens:



We know that the time complexity of a BFS = O(bd);

b = number of branches at each layer, d = number of child node layers from the root (depth)

If N = 30, then there will be 30 branches at each layer, and 30 layers depth from root node to bottom

So: b = 30, d = 30

When n = 30, Time complexity = O(3030)

As for number of solutions – since I did not manage to get past 10 Queens, I do not have enough data to theorise how many solutions N = 30 has.

# Part B – Informed Search

2 different informed searching strategies have been developed to solve this problem – Hill-climb search, and Simulated Annealing.

## Hillclimb

### Algorithm Design – Hill-climb

This implementation of the hill-climbing search algorithm is the random restart hill-climb. It proved to be relatively effective in solving the N-Queens problem.

When the algorithm finds a local maximum, it will randomly restart to a new point in the state space, hoping this time to be able to progress to find the global maximum.

**Important: when generating a new N-Queens board – each column has been allocated 1 queen – since we can assume N columns = 1 Queen per column**

### Pseudocode – Hill-climb

while the queens are clashing:

boards = array of all local neighbours

for every neighbour in the array of neighbours:

store <- neighbour’s cost

if neighbour’s cost is better than previous cost:

neighbour is our new board

if the board has not been changed to a neighbour at all:

exit the hill climb and restart

if cost == 0:

solution has been found

### Results

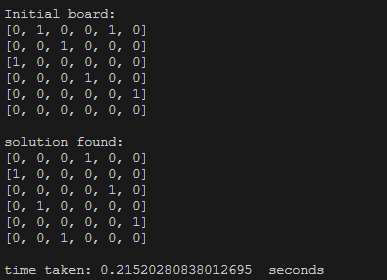
Average running time of Hill-climb for N-Queens:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| N Queens | 4 | 5 | 6 | 7 | 8 | 9 |
| Time (seconds) | 0.007 | 0.013 | 0.167 | 0.295 | 2.473 | 9.659 |

(These averages are across 5 runs)

Beyond 9 Queens, it was found to be excessive to be running to get the average times.

### 6-Queen Output – Hill-climb



Here it can be seen the basic output for the Hill-climb search:

* The initial board
* The solution board that has been found
* The computation time it took to complete

## Simulated Annealing

### Algorithm Design – Simulated Annealing

The Simulated Annealing algorithm was very similar to the previously implemented Hill-climbing algorithm; however, it differs in the way it deals with local maximums.

Where the hill-climb would restart to avoid a local maximum, the simulated annealing algorithm will use a mathematical algorithm to take a random move that will allow the algorithm to traverse past the local maximum – it never restarts.

**Important: again, when generating a new N-Queens board – each column has been allocated 1 queen – since we can assume N columns = 1 Queen per column**

### Pseudocode – Simulated Annealing

while Temperature is greater than 0:

for n times:

board <- a single random neighbour of our board

neighbour cost <- board’s cost

if cost == 0:

A solution has been found

Else if the neighbour\_cost is less than original cost:

Our Board <- Neighbour’s board

else:

delta = cost – neighbour cost

p = edelta / Temperature

prob <- random value between 0 and 1

if prob < p:

accept a random move

T <- Temperature \* alpha

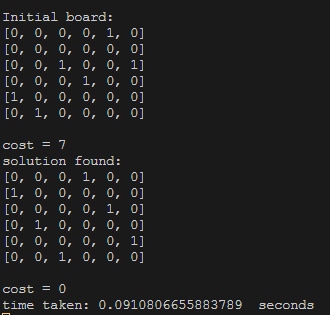
### Results

Average running time of Simulated Annealing for N-Queens:

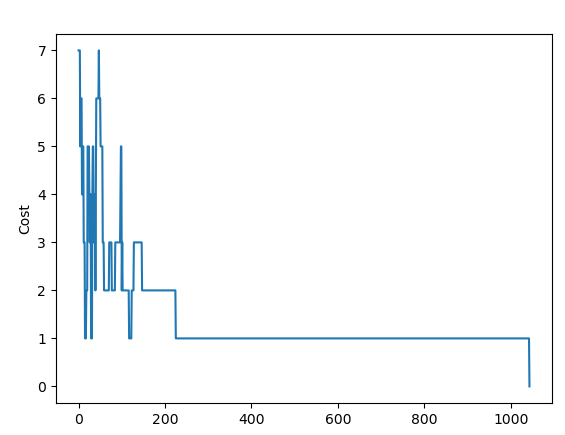
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| N Queens | 4 | 5 | 6 | 10 | 20 | 30 |
| Time (seconds) | 0.01 | 0.0144 | 0.153 | 0.517 | 3.176 | 7.202 |

(These averages are across 5 runs)

### 6-Queen Output – Simulated Annealing



The basic output of the Simulated Annealing Search as above:

* The Initial board
* The initial cost
* The Solution Board
* Ensure the solution cost = 0
* The time taken to find the solution

This graph represents the value of the cost growing and falling over the simulated annealing search above – with it growing meaning that it is traversing a local maximum/minimum.

# Informed Search – Hill-climb vs Simulated Annealing Summary

The first thing to notice in this comparison – the simulated annealing wins by a mile. Because no matter where it starts, it will always keep progressing towards finding that global maximum. The hill-climb using a random restart, potentially can find every single local maximum before progressing to find the global (unlikely, but not impossible).

This dominance is easy to see in the results – with Simulated Annealing able to complete N = 30 quicker than Hill-climb completes N = 9.

This of course, makes absolute sense, because the Simulated Annealing is an improved version of the Hill-climb search – they both search nearby neighbourhoods, both take a better move if it is available – but because of how the Simulated Annealing deals with the situation of not finding a better move without resetting the board – it performs far better.

### Simulated Annealing Parameters

When working on the Simulated Annealing implementation, deciding on a starting temperature value was a bit of a challenge – I ended up trying out many different values to trial and error and see what happens.

I decided on using a temperature of 3 – starting with a really high temperature, such as 1000+ had adverse effects on time – it took slightly longer to complete.

Using a temperature that was < 3 also had a different effect – it seemed to make the running time more randomised, less predictable.

The decision to go with 3 resulted in the most consistent running times – at least in my own testing.

As for choosing an alpha value, the decision was made to go for 0.9 – as it allows for a small temperature change, nothing too significant, but also not wasting time on iterations with little to no change.