|  |
| --- |
| 2802ICT Intelligent systems |
| N-Queens Problem |
|  |
|  |
| **Tomas Sheiles-Hutton s5132012** |
| **Trimester 1 2019** |

|  |
| --- |
|  |

Contents

[Introduction 2](#_Toc4410627)

[Part A: Uninformed searches 3](#_Toc4410628)

[Breadth First Search 3](#_Toc4410629)

[Solutions using BFS: 3](#_Toc4410630)

[Depth First Search 4](#_Toc4410631)

[Solutions using DFS: 4](#_Toc4410632)

[Predicting Larger board sizes 5](#_Toc4410633)

[BFS: 5](#_Toc4410634)

[DFS: 6](#_Toc4410635)

[Pruning the search space 7](#_Toc4410636)

[Part B: Informed Local Searches 8](#_Toc4410637)

[Hill Climbing algorithm: 8](#_Toc4410638)

[Simulated Annealing: 9](#_Toc4410639)

[Hill Climbing results 9](#_Toc4410640)

[Simulated annealing 10](#_Toc4410641)

[Comparative study: 10](#_Toc4410642)

# Introduction

Originally developed in the 1940’s by Max Bezzel, the 8 queens problem has since become a staple problem in computer science. The original problem involved determining the number of permutations of a chess board populated with eight individual queens such that no queen could take another in a single move. While a relatively simple concept, due to the large number of possible arrangements of such a board combined with the large freedom of movement allowed by the queen piece; determining the total number of solutions for a given size of chess board can quickly become too time and memory expensive for computers to produce a solution using only brute force methods. In order to effectively compute the problem, an algorithm must be used to reduce the state space of the problem and effectively traverse the possible solutions. It is for this reason that the problem can be used to evaluate the effectiveness of a particular search algorithm and quantitatively measure its performance when solving problems with very large search spaces. For this report four separate algorithms were implemented using python for the purpose of analysing how each distinct approach performs when solving the queens problem.

# Part A: Uninformed searches

Uninformed search strategies are characterized by the fact that they generate and traverse a problem without any additional or specific information relating to the current state of any particular graph. The order in which uninformed algorithms traverse a problem is not altered by any heuristic specific to the problem being solved. Because of this, in order to ensure all possible goal states are reached, every node in the graph must eventually be explored to produce the optimal solution. Dramatically increasing the computational cost needed to complete the search.

## Breadth First Search

Breadth first search is an uninformed method of traversing a tree graph. The method used in this algorithm comprehensively explores the problem set by expanding the children of an origin node and then exploring each in order of its addition to the queue before exploring the children of the second level in the graph (First in First out).

In terms of time and space complexity, this algorithm increases exponentially in both cases ( where b is the width of the graph and d is the depth of each branch) because of this exponential growth, the algorithm either quickly reaches the limits of the computer’s memory or completion times far exceeding reasonable levels. It is however complete and will eventually return a solution to the problem assuming that one exists in a finite graph.

For this specific implementation of the algorithm, an array of equal length to that of the board was used to represent its current state; where the vertical columns of the board are represented by the array’s value and the horizontal rows represented by the position of each value in the array; this was used to both reduce memory requirements of other implementations and to eliminate the need to verify the position of the queens on the horizontal axis. The algorithm then iteratively generates all child nodes of the origin node and adds them to a FIFO queue. Once the nodes are removed from the queue, the validity of each one is individually verified by first ensuring that no two nodes share the value (and thus the same vertical axis) or the same diagonal by calculating the gradient between each pair of queens on the board () and comparing to 1 or -1 to determine if a straight diagonal line could pass between them. All solutions are then appended to a separate array of goal states and the program continues until all nodes are explored.

### Solutions using BFS:

|  |  |  |
| --- | --- | --- |
| **N** | **Solutions** | **Time to complete in seconds** |
| 1 | N/a | N/a |
| 2 | 0 | 0 |
| 3 | 0 | 0 |
| 4 | 2 | 0.0009 |
| 5 | 10 | 0.007 |
| 6 | 4 | 0.10 |
| 7 | 40 | 2.18 |
| 8 | 92 | 50.12 |
| 9 | N/a | N/a |

N = 1: while technically N=1 has a single solution of a single piece on the only available space, as no other queens can be placed, it is not counted as a valid solution.

N = 2 & 3: For both of these sizes, no valid solution exists where all pieces can be placed on the board.

N = 9: Once the algorithm reaches this size, the exponential amount of memory required to hold the graph exceeds the capabilities of the computer’s hardware.

## Depth First Search

Similar to Breadth first search, depth first search is also an uninformed method of searching a graph. However it has a number of key distinctions that separates it from Breadth First Search. The main differentiation between the two is the order in which the nodes in the tree are explored. Depth first algorithms use a Last in First Out stack to store and traverse the problem, meaning the entire extent of the first added nodes children is explored before the second original child is.

Because of this differentiation, the complexity of the algorithm is different. While it maintains an exponential time complexity the space complexity becomes linear where b is the width and d is the depth of the graph. The implications of this are that depth first algorithms have the potential to be run for longer periods of time as the amount of memory used does not increase significantly overtime. This however forgoes the completeness of the breadth first solution as it has the possibility to become stuck in infinite loops.

This implementation of the depth first search uses similar methods to that of the breadth first. The main difference is the order in which nodes are added to the stack. The nodes are removed from the beginning of the stack and they are, each child node is appended to replace its position.

### Solutions using DFS:

|  |  |  |
| --- | --- | --- |
| **N** | **Solutions** | **Time to complete in seconds** |
| 1 | N/a | N/a |
| 2 | 0 | 0 |
| 3 | 0 | 0 |
| 4 | 2 | 0.0009 |
| 5 | 10 | 0.007 |
| 6 | 4 | 0.10 |
| 7 | 40 | 1.857 |
| 8 | 92 | 39.07 |
| 9 | 352 | 936.73 |
| 10 | 724 | 25473.99 |

N = 9 & 10: Due to the fact depth first solutions are linearly space complex, these two solutions could be determined but the exponential time constraints limited higher number solutions.

## Predicting Larger board sizes

Using the results obtained from both the breadth and depth first algorithms, educated predictions can be used to estimate the number of solutions as well as the time it would take to compute the problem for significantly larger sized boards. This estimate can be used to overcome the limitations placed on the algorithm due to time and memory constraints and obtain a relatively accurate answer for these larger problems. Because both of these algorithms exponentially expand both in time to complete and number of valid solutions an exponential curve can be added to the data and then used to calculate the estimated answers to larger problems.

### BFS:

Using the trend lines in combination with their formulas, the values for any board size can theoretically be calculated within a certain margin of error. This is caused by the fact that with so few data points, exponential trends are difficult to calculate.

**Time to complete BFS:**

For this formula, n – 2 must be calculated as zero values are not accepted for exponential trend lines

**N = 8 (Actual value = 50.12):**

**N = 30:**

From this calculation it is evident that it is a physiological impossibility to find all answers to the n-queens problem for a board this large.

**Number of solutions BFS:**

Similarly, the number of solutions can be obtained in a similar manner using the exponential trend line.

**N = 8 (Actual value = 92):**

**N = 30:**

Or ~170,000,000,000

### DFS:

**Time to complete DFS:**

Due to the trend line not accepting value of zero, n – 3 must be used to calculate the value

**N = 8 (Actual value = 39.07):**

**N = 30:**

**Number of solutions**:

While theoretically the same as BFS, due to more data points being available DFS should be more accurate

**N =8 (Actual value = 92):**

**N = 30:**

From these results a number of conclusions can be reached. Due to the fact that the DFS algorithm has the potential to reach a higher number of solutions, the predicted time completion and number of solutions for larger boards can be assumed to be more accurate. As well as this, due to the faster completion time, while trivial at low board sizes, the difference in time to complete dramatically decreases at large sizes assuming the necessary pruning is possible for BFS to solve these larger problems. Although due to the incompleteness of the DFS algorithm, it can never be proven with certainty that all solutions have been found.

## Pruning the search space

There are a number of strategies that can be applied to prune the total number of nodes that must be traversed using uninformed strategies. The benefits of this are a vast increase in time to complete any particular search as well as a reduction in the total amount of memory in certain situations.

The strategy employed to prune this specific implementation of the search algorithm reduces the amount of possible nodes in the graph traversed by forcing only one queen to be placed per horizontal row. This entirely eliminates all permutations of the board with two queens in same row and was required in order to find viable solutions to even small board sizes in a reasonable time.

However for problems larger than n=10, another method must be used to prune the results as completion time and memory usage still remain high. One method to achieve this is to store all rotated variation of each node, removing the need to explore their viability once reached in the tree; While speeding up completion time, this method could prove too memory expensive for deeper levels of larger problems. Similarly to the method used, the search tree could avoid placing queens in columns that already contain a certain piece; this method could dramatically speed up completion time. Due to the nature of the problem requiring n number of pieces on a n sized board, all solutions that do not contain n number of pieces could be discounted in order to reduce overall tree size; this strategy however changes the nature of the employed search algorithm and may lead to more computationally expensive methods to maintain the correct search order.

# Part B: Informed Local Searches

Informed or heuristic search algorithms differ from uninformed search in one key aspect; for each unique problem the algorithm traverses, a heuristic is calculated to find the solution. A heuristic is a value that is used to evaluate the fitness or optimality of the current iteration in the search thereby allowing the algorithm to continuously make moves towards a global optimum solution for the current problem without holding all heuristics in memory at one time. In theory, heuristic searches can produce a solution in both a faster time and with less resource occupation due to the fact that most permutations that are not optimal can be ignored and as such only a small neighborhood of nodes must be stored at any one time.

## Hill Climbing algorithm:

Hill climbing algorithms are an informed method of reaching an optimal solution to a particular problem using a numeric heuristic. This method involves moving towards a global optimum solution by calculating the heuristic cost of all surrounding neighbor nodes to the current iteration and moving to the neighbor with the most desirable value and then repeating until no more moves can increase the heuristic value. While one the more simple informed search methods, hill climbing often becomes stuck in ‘local maximums’ where a higher optimal value exists but cannot be reached as the algorithm must take a step down in heuristic value before reaching it.

For heuristic searches time and space complexity cannot be as objectively stated, as every input can lead to a different run time or problem size; however it can be estimated. Because of the fact that hill climbing alone can never guarantee it has reached a global optimum, it is neither complete or optimal and can infinitely attempt to search for a more fitting solution meaning O(∞) and only terminates once a specified condition is met. In terms of space, because of the fact that only the neighbor nodes need to be stored at any one time, the space complexity is O(n) where n is the size of the board.

For this specific implementation, the heuristic used to conduct the search was calculated based on how many queens can directly or indirectly attack each other with the current board permutation, meaning that a solution to the n – queens problem is any board with a heuristic cost of zero and any lower neighbor values to the current should be taken. A list of all possible moves is generated as well as the heuristic costs for all of them, any move that shares the current lowest value move is randomly selected between and the process repeats until the global minimum is reached.

## Simulated Annealing:

Simulated annealing is similar in many ways to hill climbing algorithms. It however uses a slightly altered method in order to avoid many of the problems caused by the fact that hill climbing only ever accepts the most optimal current move. The algorithm instead selects a neighbor at random and compares the cost of both, if the newly selected neighbor is closer to the optimal solution it is selected and the process repeats. However if the selected neighbor has a lower than optimal cost, it is also selected at a certain probability based on a temperature value. Once a defined number of iterations have been completed and the goal state has not been reached, the temperature is then decreased at a certain rate which in turn decreases the probability of accepting a worse move as the algorithm reaches a value closer to the global optimum.

Due to the changes made between hill climbing and simulated annealing, because of its ability to take bad moves in order to move towards a global maximum, the algorithm is complete and will find the most optimal solution given a finite search space and sufficient values for temperature, decay rate and number of iterations.

For this implementation, a random starting board is generated and all neighbors are added to an array. A random neighbor is then selected and its heuristic cost is compared. If the cost is worse the probability to accept is calculated using the formula . A random choice is then made with the calculated probability and if accepted the worse move is made.

### Hill Climbing results

|  |  |
| --- | --- |
| **N** | **Time to complete in seconds** |
| 10 | 0.08 |
| 20 | 0.24 |
| 30 | 5.76 |
| 40 | 21.26 |
| 50 | 127.56 |
| 60 | 68.44 |
| 70 | 187.69 |
| 80 | 370.5 |
| 90 | 783.77 |
| 100 | 1303.92 |

### Simulated annealing results:

**K** = 10,000, **Temperature** = 1,000, **temperature decay** = 0.999

|  |  |
| --- | --- |
| **N** | **Time to complete in seconds** |
| 4 | 0.005 |
| 5 | 0.003 |
| 6 | 0.164 |
| 7 | 1.037 |
| 8 | 0.98 |
| 9 | 21.25 |
| 10 | 89.85 |

## Comparative study:

By recording the time taken for each algorithm to calculate a single answer for the problem and comparing them, conclusions and comparisons can be made about the performance of each of them.

The most immediate difference between the performance of the two algorithms is the size of board that remained able to be computed within a reasonable time. While simulated annealing is a complete algorithm and will always find the optimal solution to a problem, this comes at the cost of an exponential time to compute compared to the hill climbing’s fixed power scaling.

As well as this, simulated annealing requires additional arbitrary variables in the form of temperature, number of iterations (K) and temperature decay rate that each affect the performance of the algorithm. Knowledge of the optimal values for these variables is required in order to optimize the completion time of the algorithm, a limitation not experienced with Hill Climbing.

However, despite Hill Climbing’s lower complexity and faster run time. Its nature to only accept moves that improve upon the current step cause frequent cases where seemingly no optimal answer is present. These local maximums and its incompleteness are the largest downfalls of the algorithm.

For these reasons, for any problem where an exact answer is not required hill climbing’s fixed power completion time can dramatically reduce the time to find an answer. If however the exact answer to a problem is required regardless of completion time, the avoidance of local maximums present with simulated annealing becomes the optimal choice for searching the problem space.