**CSC3206 Artificial Intelligence**

**Assignment 2**

By

Saurabh Varughese M. Kovoor **[REDACTED]**

Pierre Corazo Cesario **[REDACTED]**

Roger Jia Sien Boon **[REDACTED]**

## Table of Contents

**Introduction** **3**

**Table Seating Plan Problem** **3**

- Introduction 3
- Goal and problem formulation 3
- Assumption 4
- Implementation 5
- Results 6
- Discussion 8

**8-Queens Problem** **8**

- Introduction 8
- Goal and Problem Formulation 9
- Assumptions 10
- Implementation 12
- Results 16
- Discussion 16

**Conclusion** **17**

**References** **19**

## I. Introduction

Previously, we were tasked to propose searching algorithms to find solutions to the two problems: table seating problem and the 8-queens problem. This report once again provides a brief introduction to the problem then discusses the actual development and implementation for our proposed solution to the problems as well as their evaluation results from measuring their performance.

## II. Table Seating Plan Problem

### *A. Introduction*

The table seating algorithm is an algorithm that helps determine the best seating position which allows everyone to sit as comfortably as they can when seated next to each other. It takes in each person's personality and closeness with each other and assigns them a number, known as the comfort value, where the higher the number, the better the compatibility. This type of algorithm can be very useful for crowd control and seating arrangement for large events and functions such as wedding receptions to best assign guests that you may want to avoid or be seated with each other since it can get rather complicated and time-consuming when done manually due to the sheer number of possible configurations.

From what we have gathered, there are two approaches to solving this problem. The first approach is to use brute force. While it is very easy to create a brute force algorithm, the program execution time can be excessive and very time-consuming, especially when the number of people to arrange increases. Not to mention, the sheer number of resources required to run the algorithm. Another approach is to create a search algorithm where there is a cost function, in this case, the overall comfort value. Rather than testing every possible combination like in brute force, it will choose the best pairing of persons at each step.

In our implementation, we developed an algorithm based on Uniform Cost Search (UCS) where the highest path cost – or comfort value in this case – is prioritised. An extra function was also added to our initial proposal. The section below will discuss more how we developed and implemented our algorithm and how the additional functionalities were added.

### *B. Goal and problem formulation*

The goal of this algorithm is to determine seating arrangements for 5 people with the highest overall comfortability value on the table.

We will start by defining the state representation and actions to consider for this search problem. So, the goal state in this case, is an arrangement of 5 people in the round table with the highest overall/summed comfortability value. This is implemented with the bestSeating list in the findSeatingArrangements function which obtains the list of best seating arrangements when different starting person is used returned from the ucs() function, then sorts and extracts only the best arrangement(s) (if multiple arrangements compute the same highest overall comfort value, then multiple are returned). So, the goal test is the test after which all the agents have been placed to determine if they have been placed in a manner in which the overall summed comfort value is the highest. In our implementation this can be seen in line 127-128, where the program checks the length of the seated list (already seated agents) if it is equivalent to the number of persons to be seated. If so, a solution has been found, and the looping stops to record/ return the names of the members of the seated list or that have been seated,

A state in this case, is an arrangement of 5 people in the round table, no matter whether it has the overall summed comfort value. In our implementation the *ucs()* function returns a list of such possible arrangements through the *seatedNames* list with its associated comfort value (*overallComfortValue*). An initial state is an empty table seating arrangement where no agents have been seated yet, as seen in the *seated* list which is declared at line 123 of the *ucs()* function.

Actions in this case is the act of seating an actor at the table. This can be seen through the *seated.append(frontier[0])* command where the program will add an agent to the seated list if it is at the front of the frontier list (*frontier[0]*). For this purpose we also implemented the actions attribute to keep track of the number of actions taken to find the solution or goal state.

A transitional model is defined as the sum of the current overall comfort value and the comfort value of the newly made pair of persons, while ensuring the list of possible people to be seated does not include any already seated persons. This is performed by the *getChildren()* function which returns the summed comfort value of the initial node/person and the newly added person/child. The frontier is reset by the caller function, *in ucs()*, this ensures the newly added child is not considered to be placed again in the next loop (not part of the frontier).

For this problem, the path cost is defined as the sum of all the comfort values of the people seated in an arrangement or also known as the overall/summed comfort value. In our program, this is tracked by the overallComfortValue attribute for every arrangement (*seatedNames*) that is created by the *ucs()* function. So, in the *ucs()* function, this is seen in line 143-145, where the *overallComfortValue* is calculated based on the last person in the seated list, where their summed comfort value is equivalent to the sum of all the previous pair comfort values. Then, to ensure that this list is circular, where the last person in the list is adjacent to the first person and to account for that pair's comfort value as well, the comfort value is retrieved (*getComfortValue()*) and added. This value is shown to the user together with the optimal solution(s).

## C. *Assumption*

Given the information we know about the problem, assumptions regarding the environment can be made. In this case, the agent is aware of who is seated beside who, thus making the environment observable. Furthermore, the environment is also discrete because once a person is seated, they cannot be seated again as seen from lines 122-126 of the *ucs()* function where the program checks to see if a particular agent is already seated at the table. If so, the placement will be skipped and the next child/agent will be tested, else the child/agent can be seated (part of the frontier). Since the algorithm is aware of the path that provides the highest overall comfort value, the environment is said to be known. This is demonstrated by the *findSeatingArrangements()* function which is able to return one or more lists of seated people along with their associated overall comfort value that have the highest summed overall comfortability value after generating all the possible best seating arrangements with different starting agents from the *ucs()* function and then sorting and extracting the list(s) with the highest comfort value (lines 59-61). Finally, the environment is deterministic since choosing one person to sit down will always result in that person sitting as seen in the ucs function with the *ucs()* function where from lines 136 to 141 a person will be appended to the frontier if they are not placed already in the seating arrangement.

## D. *Implementation*

The seating arrangement is to be arranged in a circular pattern clockwise. After a person is seated, the next seat will be chosen from the remaining people and once they are seated they cannot be seated again. After that, comfortability levels are assigned to each of them where 5 is the most comfortable and -5 is the least comfortable. To mimic a person's affinity for another in real life, while A's comfortability level with B could be 5, B's comfortability with A could be different. This is also known as the one-way comfortability value. After obtaining user input on the number of people to be seated at the table, the program will start by randomly generating these one-way values for every agent against each other. Then the program will add the values between two agents to compute the two-way comfort values. For instance, the one-way comfort value of A to B and B to A is 4 and -2 respectively. Then, the two-way comfort value (A and B) will be the summed value which is 2. This is the basis in which the algorithm is able to choose which persons to seat at every step – the pairing with the highest comfort values.

With the permutation of 5P5, we can deduce that there are roughly 120 different ways to sit 5 people. This can also be seen as the number of actions that would be taken if a brute force search was applied. This, of course, increases when the number of people to be seated in a single table increases. The first seated person will get the chance to be paired with other actors in the combination trials and only then the placement producing the highest comfortability level is taken. The newly seated person is then removed from the available pool of persons to be seated next. This loop repeats until all the seats are filled.

We will be using a modified UCS for the problem formulation. In our UCS, the *getOneWayComfortMatrix()* will assign randomised values to each one way comfort pairing. Using that matrix, the algorithm will sum the cost of each pairing with the *getPairComforts()* function. Then, to start the UCS, one of the persons to be seated is chosen to be the first. To illustrate, in the first run, person A is chosen to start the UCS. With the two way pair comfort cost values determined, the highest cost pair at each expansion will be seated with each other. This is the first modification of our UCS algorithm since it prioritises the highest cost values, not the lowest cost values at each expansion. So, the people in the frontier are arranged in a manner in which the child that produces a pair with the current node with the highest comfort value is positioned first (line 129 of *ucs.py*). Then, the front person in the frontier is appended to the seated list, signifying that the person has been seated. The process continues until all the people are seated as checked by part of the goal test, which is line 127-128. This process repeats for all the people to be seated. For example, in this second run, person B is chosen to start the next UCS. This is the second modification to the typical UCS algorithm. It repeats the UCS with a different starting person for all persons to be seated. This is because we noticed that some seating arrangements provide a higher overall comfortability value depending on the starting person.

The program will print out the best arrangement, the overall comfort level, the number of actions and the time taken to run the program. The number of actions and time taken can be used as a measurement to demonstrate the efficiency of our program, especially when compared to brute force.

Since the main algorithm is done, extra functions were added to increase the functionality of the program. A simple and light-weight user interface through the python terminal is created to allow a user to interact with the program. Simple and clear messages were also used while developing the algorithm to keep the interface user-friendly. The initial proposal limit of the number of people seating was also increased to accommodate 4 to 52 people in a single table. 1, 2 and 3 people were not considered for this algorithm since no matter the permutation, the overall comfort values would not change, so any arrangement would be the best arrangement.

## *E. Results*

In the following table, the number shows the amount of action and time taken to calculate each number of people to be seated. Computers with different specifications may have varying results. For our test, we used a computer with an i7-8750H (6 cores/12 threads) and 8GB of DDR4 RAM.

**Table 1**

*Number of Actions and Execution Time per Number of People to be Seated*

| Number of People to be Seated, N | Number of Actions | Execution Time (ms) |
|---|---|---|
| 4 | 12 | ≈0.0 |
| 5 | 20 | ≈0.0 |
| 6 | 30 | 0.99512~ |
| 7 | 42 | 1.01245~ |
| 8 | 56 | 1.99561~ |
| 9 | 72 | 2.01587~ |
| 10 | 90 | 2.01147~ |
| 11 | 110 | 4.98975~ |
| 12 | 132 | 5.99219~ |
| 13 | 156 | 4.98633~ |
| 14 | 182 | 9.00537~ |
| 15 | 210 | 10.96265~ |
| 16 | 240 | 13.96191~ |
| 17 | 272 | 16.9502~ |
| 18 | 306 | 17.92358~ |
| 19 | 342 | 19.94409~ |
| 20 | 380 | 23.97021~ |
| 21 | 420 | 31.91113~ |
| 22 | 462 | 34.9021~ |
| 23 | 506 | 44.87866~ |
| 24 | 552 | 44.90161~ |
| 25 | 600 | 49.84692~ |
| 26 | 650 | 56.86646~ |

### *F. Discussion*

Based on the results, our algorithm is able to generate the seating algorithm in less than a second. Looking at the trend of the time taken, it is safe to assume that when the amount of people increases the time needed to generate also increases. We also observed that the program will execute the code faster after having been executed several times. We suspect this could be due to cache, whether it be from Python or the system. Tests were limited to 26 people in the interest of time and as the results are merely to demonstrate the efficiency of the program. Furthermore, 15 or more people seated in a table is rather unrealistic, thus the remainder is left to the user for them to experiment with.

Similarly, in terms of number of actions we can see that the number of actions that our UCS approach takes for seating 5 people is 12, in contrast to our aforementioned unsuitable algorithm, brute-force search which would have to consider and check all 120 (5P5) different seating arrangements to find the one with the highest overall comfort value. So, from this we can see that our UCS approach requires far less number of actions to find the solution for the table seating problem of 5 agents. This translates to a lesser time and space complexity which will be even more evident for larger iterations of this problem, where the number of people to be seated is higher.

## III. 8-Queens Problem

### *A. Introduction*

The 8 queens problem is a classical combinatorial problem, which is part of the more generalised N-Queens problem, where N represents the number of queens to arrange or the size of the chessboard the problem is concerned with. For example, the problem can be implemented or demonstrated as the 4-queen problem or 5-queen problem, which is concerned with finding the arrangement of 4 or 5 queens in a mutually non-attacking manner on a 4x4 or 5x5 chessboard respectively. Therefore, solutions exist for cases where N is 1 or more than or equals 4. So, there are only no solutions for the 2-queens or 3-queens problem.

In the area of artificial intelligence (AI) and algorithm design, this problem is typically explored, to study the implementation and analyse the effectiveness/efficiency of search algorithms. More specifically, it is especially influential for practical applications concerned with attaining the correct arrangement of agents (solution/goal state) with completeness(able to find a solution) and optimality (finding the solution path in the least amount of steps), for instance for VLSI layout problems and testing and traffic control problems.

In our implementation, we developed an algorithm that is based on depth-first search (DFS) with backtracking to find solutions to the N-Queens or 8-Queens problem as we found from our initial planning, that this approach is more suited to this problem domain. In the implementation section, the reasoning for choosing this approach as well as how we developed this algorithm will be explored.

## B. *Goal and Problem Formulation*

The aim is to solve the problem of placing N chess queens (8) on an NXN chessboard (8X8), in an arrangement where no two queens can attack each other or be on each other's same line of attack. So, this arrangement can be seen as one where no two queens share the same column, row, or diagonal (45°, 135°) row. This is because of the queen piece's freedom of movement and attack, where it can attack pieces on the same row, column and diagonals. This freedom of movement contributes to the difficulty in finding a solution to this N-Queens problem, as the lines of attack should be managed and designed to allow for the placement of N queens on the NXN chessboard where none are attacking or attacked by each other.

Therefore, it is understandable that as the number of queens or chessboard size increases, the number of actions/steps to reach the solution increases, the number of solutions/goal states increase, thus, requiring more time (more time complexity) and computational resources. So, the N-Queen problem becomes intractable for large values of "N" and thus classifies it as a non-deterministic polynomial (NP) class problem [1].

Next, for the problem formulation, we will define the actions and states to consider for this search problem.

First, the goal state in this case, is the arrangement of 8 queens on the 8x8 chessboard in a configuration where no queens are attacking each other or are sharing the same row, column or diagonal. In our implementation this is seen as a list, called boardState where all rows have a queen, so the index represents the row whereas the value represents the column.

So, the goal test is the test after the 8 queens have been placed to determine if they have been placed in a manner where there is no collision or threat/opportunity of attack. In our implementation, before choosing a queen placement (column in a particular row), we check if there exists a queen in the previous rows which are in the same column or same diagonal, using the checkCorrectPlacement function in the Chessboard class.

Next, the states in this case, is a combination of 8 queens on the 8x8 chessboard, whether it is correct or incorrect. Hence, it is also represented by boardStates in our implementation, however, this list typically only holds solutions as we avoid the occurrence of incorrectly placed 8 queens on an 8x8 board beforehand as we are progressively adding queens we make sure its not colliding with previously placed queens and rejecting such collisions or placements with the checkCorrectPlacement function called by the getPlacementOpp function. So, by the time we reach the 8th row of the boardState, we can be sure that all the queens that are placed don't collide with one another in the boardState.

The initial state is an empty 8x8 chessboard where no queens are placed yet. We represent this blank state with -1 multiplied with the list with the size of boardSize (N) or 8 for the 8 queens problem (line 3-6, Chessboard.py).

Actions, in this case, are considered to be the act of placing a queen onto any available square on the chessboard. In our implementation, this is represented by the placeQueen function, which updates the chessboard/boardState list by adding a queen at the specified row and column. We also declared a variable "actions" which increments every time the function is called, to track the number of actions required to find each solution or the total number of actions for finding every solution. So, in our case, an action is only considered the placement of a queen on the chessboard, and doesn't include the removal of pieces, checking for collisions, resetting or initialization of the chessboard, etc.

Moving on, the transitional model is the description of what each action achieves, so in this case that would be the function to return possible queen placement opportunities and validating whether a queen can be placed in the particular coordinate without having collisions with any other queen, if so, that cell will be abandoned, and the next cell will be checked for placement and this will progress recursively. For this, we use a function, *getPlacementOpportunity* which will return possible queen placement options by iterating over every column in a particular row, and checking if it is possible to place without colliding with previous row queen pieces, using the checkCorrectPlacement function. Hence, with the input of the boardState and current row attribute, the getPlacementOpportunity function can determine all the positions where queens can be placed correctly (without collision) in that particular row.

The path cost in this case is the number of steps/actions (placement of queens) required to reach a particular solution (placement of 8 queens on the chessboard in a mutually non-attacking manner). This is especially an important consideration for our depth-first search with a backtracking approach, whereas backtracking is involved, this adds on to the typical 8 steps required to reach one state. So, the number of steps is not constant for reaching every solution. Therefore, we track this with the "actions" attribute in the *placeQueen* function that increments every time the function is called. So, this "actions" attribute which tracks the number of actions or placements of queens to find a solution or all solutions, indicates the path cost of our DFS with a backtracking approach to solving the N-queens problem.

### C. *Assumptions*

Based on the information about the environment and the problem domain, we can deduce some other assumptions about the environment to assist us in the problem formulation. For this case, the environment is considered "observable", as the agent is always aware of the current state, in this case, from the placement of the queen on the chessboard, we know its position/coordinates on the board and its lines of attacks to prevent other pieces from being placed there. Secondly, the environment is considered "discrete", as at any given state, there are only a finite number of actions to choose, which in this case, a queen can be placed anywhere on the 8x8 chessboard, so there are 64 positions to choose from. This number decreases as more queens are placed on the board, and there are lesser positions to choose from without sharing the same row, column, or diagonal as another queen. Thirdly, the environment is "known" as the agent "knows" which states are achievable by performing each action. Thus, in this case, this can be seen as placing a queen on a certain cell on the chessboard, we can know whether it will cause a collision with another queen, and if it can be placed correctly (*checkCorrectPlacement* function), we can know its lines attack to determine the placement of subsequent queens. Finally, the environment is "deterministic", in that each action has one outcome, so by placing a queen on the chessboard, we can determine whether the placement is correct (there are no collisions with other queens) and so the placement can be kept, or if the placement is incorrect (collision occurs as the same row, column or diagonal is shared between 2 queens), the position is rejected and the queen is placed on the next cell (*getPlacementOpp* function).

One assumption that we considered during the implementation of this N-queens simulator, since we found from our preliminary research that even with the inherent complexity of current search algorithms, even the most efficient and sophisticated search algorithms can only find and return solutions for n values up to 27. Thus, to prevent the user's python terminal from crashing or hanging and to avoid the program from being overloaded with values and not being able to complete execution, we restricted the simulator to only allow N input values between 1 to 26, except 2 and 3. We performed this by implementing error handling when users enter a bigger or smaller integer and requesting the user to input another integer between 1 to 26

Another assumption we have made in our implementation is that, since there are no solutions for the 2-queens and 3-queens problem, we prevent users from entering 2 and 3 for the size of the chessboard or number of queens by performing error handling and requesting the user to input another integer between 1-26.

An assumption that we made and kept in consideration during our implementation, as seen in the, *checkCorrectPlacement()* function is the constraint or rule where when determining possible positions where queen can be placed on the next row, we eliminate positions that share the same row, column or diagonal as another queen on any of the previous rows. Hence, we can use this to further narrow down the possible cells to place the queens. So, the problem space can be reduced to just 8! = 40,320. Thus, giving us significantly lesser combinations to have to try and explicitly examine. This optimisation can substantially speed up the computation because a highly reduced board space can be especially effective for chessboards of larger sizes (larger values of N, in N-queen problem). Also, validity checking after placing a queen in a cell becomes simpler as we only have to check collisions in the diagonals and anti-diagonal direction.

Another constraint that is enforced in our implementation is the fixed positioning of the first queen or starting point which is on the bottom-left cell, A1. After that the subsequent cells where the queen is placed is determined based on our DFS with backtracking approach.

Overall this shows that as the constraints increase, the possible solution set is narrower for the problem domain, thus lesser time and computation is needed to reach a solution [2].

## *D. Implementation*

From conducting the problem and goal formulation, with the assumptions in mind and thoroughly understanding the problem at hand, we decided to implement the depth-first search (DFS) algorithm with backtracking for searching the state space and finding solutions to this N-Queens problem.

This is because, in DFS, which is an uninformed search algorithm, from a root node, it expands the deepest node in the current frontier of the search tree. So, it traverses vertically instead of horizontally (breadth-first search), recursively traversing the subtree and stops when it reaches either a non-solution state or goal node. Usually this would not be ideal for cases where the tree depth is unknown and could extend to infinity until leaf node is reached as the adjacent subtrees would not be traversed or visited. However, in our case, the tree depth will be a maximum of 8 for the 8 queens problem or N for the N-Queens problem, so it is fine to traverse vertically to get a state where all 8 rows of the chessboard have one queen placed on it. So, starting from a root node, in our case, the bottom left corner coordinate A1, it traverses every possible queen placement in the subsequent rows. So, we utilise DFS to dictate or determine the order in which we "search" or find a set of steps (placement of 8 queens on the board) to find a solution or goal state (8 queens on board without hitting one another) and to find the subsequent solution.

By implementing backtracking, we can further ensure that only subtrees that lead to a goal state are explored, overall, reducing the number of actions or steps needed to reach a solution. Backtracking is at the core of DFS so it integrates well [3]. This is because with backtracking partial solution candidates that do not lead to a solution are foregone and ignored, and based on the DFS, queens are continued to be added to the state space in an incremental manner row-by-row while obeying the predetermined rules and constraints.

The following sections provide brief explanations on the implementation of the main stages of our N-Queens simulator program. The source code is divided into 2 files, the Main.py file which is the main driver file and the *Chessboard.py* file which contains the Chessboard class which can be instantiated to create a Chessboard object and to access its six methods/functions which are required to simulate to modify the Chessboard object for the solution-finding of the N-Queens problem. So, in order to execute this program, one will have to navigate to the Chessboard directory containing these two python files through their python terminal, then execute or run the Main.py file to access the output correctly within the python terminal. From there, they can simply follow the guided steps to using the n-queens simulator within the python terminal.

### Gathering User Input for boardSize and simOption, ***menu()***

The program starts executing the menu() function which requests input from the user on the size of the chessboard/number of queens and then the simulation mode or option they would like to display the output. For the simulation option, they can either choose to display each action (placement of queen) one-by-one (choose 1), each solution one-by-one (choose 2), every solution at once  (choose 3), or to quit the program and end execution (choose 4). Error handling is performed to ensure the user enters the correct input within the specified range (board size: 1-26, except 2,3 & simulation option: 1-4).

With the inputs obtained and before starting the main program, now the tracking of the execution time begins by recording the time at start time (time.time()). The Chessboard object is instantiated based on the board size that the user entered. When instantiated, a blank chessboard is created in the form of a list signified with -1 values with the size of the entered board size. This chessboard is also called a *boardState* which keeps track of where queens have been placed and dictates where the next queens can be placed. It is in the form of a tuple list where the index represents the row and the value represents the column of the queen. So, if the value is -1, the queen for that row has not been placed yet. This will also be used to visualise the chessboard (states and solutions) later in the Python terminal. The actions variable is instantiated to keep track of the number of actions taken by this approach to finding solutions.

**Generating Chessboard States, *getBoardStates()***

Then, the object's *getBoardStates()* function is called with the chosen simulation option as an input. The simulation option input allows for 2 forms of output from, a generator object with list of boardstates for every action performed (mode 1) and one with boardstates for only the solutions (mode 2,3). This allows mode 1 to give the impression of a simulator where the user can witness all the actions that are taken through depth-first search and backtracking to find a solution, whereas the other options are for users who only want to see the goal states. The getBoardStates function starts the depth-first search approach from depth 1 or the root node, and row 0 or the first row of the chessboard. The purpose of this function is to generate the chessboard states and solutions.

**Generating Potential Queen Placement Opportunities, *getPlacementOpp()***

So, it will then call the *getPlacementOpp* which will generate all the possible queen placements options on a col (returns col, where queen can be placed) in the current row where no collision occurs. It does this iteratively for every column in that particular row (using for loop) by calling another function called *checkCorrectPlacement* for checking the columns in that row, if placing a queen at that position/cell will cause a collision with any queens placed at previous rows.

**Checking Queen Placement and Collisions, *checkCorrectPlacement()***

The *checkCorrectPlacement* function does this by iteratively checking (while loop) at every previous row (*prevRow*), if there's a queen at a previous row (*prevCol*), the current queen should not be in the same col (*col = prevCol*) or on the same diagonal (*abs(row - prevRow) == abs(col - prevCol)*). If not the function will *return True* and that column will be returned as a potential placement opportunity in that particular row. Else, the function will return false and the next column is checked in case a queen can be placed there.

If a queen cannot be placed on any columns in that row, backtracking will occur by recursively calling the initial *getPlacementOpp()* function and the queen on the previous row will have to shift to the next col. The *getPlacementOpp()* function yields all the columns for that row which allows the placement of queens without colliding with any previous queens which can be explored.

**Placing Queen on a Particular Row, *placeQueen()***

With this list of columns the program will place a queen at that particular row and column (cell) using the *placeQueen()* function. The purpose of this function is to simply update the current boardState, by adding a column value to the index of the *boardState* (row). This function will increment the actions attribute, which signifies that an action (placement of a queen) has taken place.

If this is mode 1 or the mode which simulates every action, a copy of the board state is returned to be visualised to the user.

**Iteratively Generating Board States for Next Tree Depth, *getBoardStates()***

All these steps will be iterated for every row in the chessboard or as long as the current tree depth is smaller than the board size or the final row has not been reached yet. So, the program will recursively call the *getBoardStates()* function for the next tree depth. If the final row has been reached or the last tree depth (8 for the 8 queens) is the same size as the user input board size, the *boardStates* that have been yielded this far, together form a solution. This will be copied and yielded to the caller.

Overall, it is clear to see the importance of our utilisation of the generator function in our implementation of this program, through the use of commands such as "yield", which in contrast to "return", yield will suspend and return the functions output to the caller but will retain the state and allow for the function to proceed from the previously suspended state. This will allow for the implementation to proceed smoothly, in this case, after finding a solution, the next solution can be found by continuing to iteratively move and queens to a new position in the boardState of the chessboard to generate unique solutions. So, the return datatype of these generator functions *getBoardStates()* and *getPlacementOpp()* is a generator object of lists.

So, at the end, all the *boardStates* will have been yielded and the end time is recorded (*time.time*) to compute the duration of the execution time, which will be displayed at the end if the chosen simulation option is mode 3.

**Visualising Board States Based on Chosen Simulation Option, *chooseSimOption()***

With the *boardStates* yielded from the *getBoardStates* function we can visualise them in a clearly formatted form that mimics the appearance of that of a chessboard using the *displayChessboard* function. Before this, the *chooseSimOption* function is called which will generate/visualise the *boardStates* or chessboard according to the chosen simulation option by the user. So if the user chose option 1, the program will display boardstates that contain -1 (which signifies rows without queens) as an action along with the action number and if the *boardstates* do not contain -1 it is considered a solution and an appropriate message will be shown indicating the solution number. Whereas for option 2, the *boardStates* displayed will be solutions only, but the subsequent solution chessboard will only be displayed after the user presses Enter (*input()*). The program will also display the number of actions needed to reach the particular solution. Then for the 3rd option, the program will display every solution in one go, along with the solution number as well as the execution time to compute every board state. After all the *boardStates* have been displayed, for all modes, the total number of actions taken for the problem is displayed.

**Displaying Board State in the Python Terminal, *displayChessboard()***

Another important function in this program is the *displayChessboard()* method, which as mentioned before, has the primary purpose to display chessboard states and goal states in the Python terminal. It starts by printing the top edge of the chessboard, using box unicode characters according to the specified chessboard size. Then, using for loop, the body of the chessboard is printed row-by-row in a reverse manner to ensure that A1 appears at the bottom left, to follow that of a traditional chessboard and to ensure that it aligns well with our computed boardStates. So, the program checks to see a position in the row in the chessboard for column that doesn't have value -1 (not empty), and if it is not -1, a queen unicode icon will be printed there. For the rest of the boxes an empty leading and trailing blank squares are printed, before finally the right edge of the chessboard is printed. Then, the bottom border lines are printed. So, this progresses till all the rows in the chessboard state according to the board size is visited. Afterwards, the board column coordinates are printed with 4 blank spaces printed between them, and they are shifted 4 cells to the right so they are in line with the chessboard. Once all the columns have been visited or the loop index has reached the size of the board size, the printing of the column coordinates stops and a blank line is printed. After the chessboard, there is also an area where messages can be printed, indicating the solution number.

## *E. Results*

There are 2 metrics that we track to measure the performance of our DFS with Backtracking approach so that it can be easily compared with any other approaches. This includes, the number of actions and the program's execution time (in seconds)

Therefore, the following table displays the results of the simulation that was conducted on a machine with an i7-8750H a 6 core 12 threads CPU and 8GB of DDR4 ram, detailing the number of actions taken and the length of the execution time to find all the solutions to the N-Queens problem:

**Table 2**

*Number of Actions and Execution Time per Number of Queens (N) for the N-Queens Problem*

| Number of Queens, N | Number of Actions | Execution Time (in seconds) |
|---|---|---|
| 1 | 1 | 0.00296s |
| 4 | 16 | 0.02054s |
| 5 | 53 | 0.11237s |
| 6 | 152 | 0.03131s |
| 7 | 551 | 0.53022s |
| 8 | 2056 | 1.51273s |
| 9 | 8393 | 5.04636s |
| 10 | 35538 | 12.28245s |
| 11 | 66925 | 66.91628s |
| 12 | 856188 | 329.9463s |
| 13 | 4674889 | 1987.98385s |

## *F. Discussion*

### Actions

As mentioned earlier in the problem formulation, for this search problem, we are only considering a placement of a queen on the chessboard as an action. So, with each new queen that is added to chessboard, the *boardstate* is updated at that particular row and column, with the queens placement and the actions attribute is incremented to signify an action was performed.

Based on the results, we can see that for the 8-Queens problem, which is specifically highlighted for our study, a total of 2056 actions are needed. Comparing to our earlier proposed unsuitable search algorithm, which is the brute force search, that would have to compute 4,426,165,368 combinations (or actions in this case) based on the formula C(n,r) = n!/(r! (n−r)!), where n=64 and r=8, to find all 92 distinct solutions as this algorithm blindly tests every possible combination of 8 queens on all 64 squares while checking if the combination fulfils the rules/constraints to be considered a solution. Hence, it is clear to see that our DFS with backtracking approach is a significant improvement on the brute force search algorithm in generating solutions for the 8-Queens and N-Queens problems.

Additionally, from analysing our results, we can see a pattern where the total number of actions needed increases drastically when the N increases. It proves our earlier assumption, where the N-Queens problem becomes intractable and the space and time complexity drastically increases for larger values of N and thus classifies it as a non-deterministic polynomial (NP) class problem. So, we can assume that for even larger values of N, N>13, the number of actions would be even more substantial.

One more thing to note is that after all the solutions have been found, one might notice that there are more actions performed. For example, for the 8 queens problem it takes a total of 1951 actions to find all 92 solutions, but there are 2056 actions performed. This is because the program is searching for any other possible solution after the 92nd based on the DFS and backtracking approach. So, the additional actions are meaningful to the program to ensure that all solutions have been found, in the event we are not aware of the total number of solutions that exist for a particular problem.

**Execution Time**

The execution time tracking begins right before the chessboard object has been initialised and then solutions or goal states to the N-Queens problem are being developed or computed. Then, the execution time tracking ends right after the chessboards/combinations have been displayed in the python terminal. So, it's taking into account the entire program execution time.

Based on our results, for the 8-Queens problem, our approach took a total of 1.51273s. Like for the number of actions, we can see a pattern in the results where the time taken grows dramatically as the N value increases. In fact for n values more than and equal to 14, the time taken is far too significant, and would require the program to be left executing for a substantial time period. So, it is safe to assume for even larger values of N, N>13, the execution time period would be even more substantial.

## IV. Conclusion

In conclusion, this paper discussed the implementation of our search algorithms for the table seating (UCS) and 8-queens problem (DFS with backtracking) along with the evaluation metrics (number of actions, execution time) we used to measure its performance and the results from this experimental study. Hence, based on our experimental study of our proposed algorithms, the results and evaluation have indicated that our proposed implemented programs perform far more efficiently than the aforementioned unsuitable algorithms in the initial planning documentation.

## V. References

[1] VikasThada and D. Shivali, “Performance Analysis of N-Queen Problem using Backtracking and Genetic Algorithm Techniques,” International Journal of Computer Applications (0975 – 8887), Volume 102, No.7, 2014.

[2] S. Farhad, S. Bahareh and F. Golriz, “A New Solution for N-Queens Problem using Blind Approaches: DFS and BFS Algorithms," International Journal of Computer Applications (0975 – 8887), Volume 53, No.1, 2012.

[3] A. Bruce and Y. Mordecai, “ Construction Through Decomposition: A Linear Time Algorithm for the N-queens Problem,” 2013.