Experimental Comparison of Uninformed and Heuristic AI Algorithms for N Puzzle and 8 Queen Puzzle Solution

Kuruvilla Mathew and Mujahid Tabassum
Swinburne University of Technology(Sarawak Campus), Jalan Simpang Tiga, 93350, Kuching, Malaysia
kmathew@swinburne.edu.my and mtabassum@swinburne.edu.my

ABSTRACT

This paper compares the performance of popular AI techniques, namely the Breadth First Search, Depth First Search, A* Search, Greedy Best First Search and the Hill Climbing Search in approaching the solution of a N-Puzzle of size 8, on a 3x3 matrix board and for the solution of the classic 8 Queen Puzzle. It looks at the complexity of each algorithm as it tries to approaches the solution in order to evaluate the operation of each technique and identify the better functioning one in various cases. The N Puzzle and the 8 Queen is used as the test scenario. An application was created to implement each of the algorithms to extract results for each of the cases. The paper also depicts the extent each algorithm goes through while processing the solution and hence helps to clarify the specific cases in which a technique may be preferred over another.

KEYWORDS

Artificial Intelligence; N Puzzle Solution; 8 Queens Puzzle; Uninformed and Heuristic AI Techniques;

1 INTRODUCTION

Artificial Intelligence (AI) attempts replicating the human ways of reasoning in computing. As a full replication may not be approachable at once due to is magnitude and complexity, research now targets commercialisable aspects of AI towards providing “intelligent” assistive services to the human users [1]. Decision making in this paradigm involves evaluating a number of alternatives in different spatial configurations, environments and circumstances and to find better of the alternatives. It also involves decision making even when an ideal alternative is not derivable. This paper is therefore limited to comparing the implementations of the popular AI algorithms, namely Breadth First Search, Depth First Search, A*, Best First Search and Hill climbing algorithms for solving a sliding n-puzzle in an attempt to look at the better efficient of the algorithms for this case. To solve the sliding n puzzle problem, one moves a set of square tiles arranged randomly in a square board to arrive at a pre-determined order. The board has only one blank square and each tile can only move to the blank space adjacent to itself. Our aim in this paper is to apply the AI approaches to the case and compare their performances in the problem solving.

This paper is organized as follows. Section II looks at related work in the areas of research. Section III outlines the problem and our approach towards the solution. Section IV details the experiment and results whereas section 5 presents the summary and conclusion based on results observed in section IV.

2 RELATED WORKS

Brooks discussed about intelligence without perception [1] in a case where intelligence is not about individual sub-system decompositions but about parallel activity decomposers that interact directly with the world, with notions of peripheral and central systems fading away.

Droguil and Debreuil presented a distributed approach in solving the N Puzzle [2], an approach based on decomposition of a problem into independent sub-goals, which is in turn decomposed into agents that satisfy the sub-goals. This approach is used to demonstrate emergent solutions and for solving for very large N Puzzles.

Kumar et. all presented summary of results from a parallel best first search of state-space graphs [3]. The paper looks at several formulations of A* Best First
algorithm and discussed how certain searches are better or lesser suited for some search problems.

Blai Bonet and Héctor Geffner study a family of heuristic planners [4], applying them in the context of Hill Climbing and Best First search algorithms and tested on a number of domains to analyse the best planners and the reasons why they do well.

Korf, R. E. discussed a study on Depth First search as asymptotically optimal exponential tree searches [5]. They discuss that the Depth First iterative-deepening algorithm as capable of finding optimal solution for randomly generated 15 puzzle.

Korf, R. E. presented a Linear Best First Algorithm [6], exploring nodes in the best first order, and expands fewer nodes. This works on the sliding puzzle with reduced computation time, but with a penalty on solution cost.

Russel, S and Norvig, P explored the detailed concepts of AI using intelligent agents and multi-agent systems search algorithms etc. in the distributed problem solving approach in AI in their book[7]. The book gives good insight towards the modern approaches in AI.

3 THE PROBLEM

3.1 The N Puzzle

The sliding puzzle is a simple but challenging case for demonstrating artificial intelligence concepts. It involves having a set dimension of puzzle space (usually 3x3 for an 8-puzzle) and denoting the dimensions as N being the columns and M being the rows. In the puzzle space, there is a random arrangement of cells/blocks, with one empty space that enables adjacent cells/block to slide into them. Since the pieces are square blocks, only the top, bottom, left and right blocks adjacent to the empty space may slide into its place. The cells can either be numbered or printed with a fragment of the whole picture that the rearranged puzzle should show[2].

3.2 The 8 Queen Problem

The classic 8 Queen puzzle is a very similar puzzle as the N puzzle. This involves an 8 x 8 game board, commonly used in the game of chess. The challenge is to place 8 queens of a chess game so that no queen is facing another, i.e. not in the same line, either diagonally or directly. The initial state of the puzzle starts with an empty board and the goal state has all the 8 queens placed on the board, not facing each other, as shown in the figure.

![Figure 1. The 8 Puzzle, initial and goal state](image)

3.3 The Problem Formulation

Problem formulation is done by analysing the environment of the puzzle and deriving its characteristics using PEAS, as in Table 1.

3.4 Approaching the Solution

3.4.1 N Puzzle

For solution searching, it would be most useful to distil the possible arrangements of tiles as individual States. Thus, each State shows a possible combination of tile positions within the given puzzle space. The collection of all possible States is called the State Space. With the increase of N or M of the puzzle, the size of the State Space shall increase exponentially.

In every state, the empty space position determines which States can be transitioned to. For instance, when the empty space is in the middle of a 3x3 puzzle, tiles at the Top, Bottom, Left or Right can move into it. But if the empty space is at the top left corner, only the right or bottom tiles can slide into it.

Thus, after each slide, a new State is transitioned into. If puzzle is to begin with an Initial State of tile arrangements, then its subsequent transitions into other States can be represented by a Graph. An example of this can be seen in Figure 2. A search attempt will need to begin with an Initial State and a Goal State to
achieve. As puzzle traversal can often pass through the same state at different intervals. We will consider the instances of decisions as nodes. By aligning the node arrangements to start from the Initial Node to possible routes leading to the Goal nodes, a search tree is formed, as we see in Figure 3. The algorithms explored in this paper will traverse the Search Tree in different ways to find the Goal State from the Initial State.

### TABLE I. ENVIRONMENT ANALYSIS – N PUZZLE

<table>
<thead>
<tr>
<th>Sl No</th>
<th>Environment Characteristics of Puzzle</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Performance</td>
<td>Arrangement of tiles/cells/blocks in the whole puzzle space. Main performance gauge is from the least number of moves to solve the puzzle.</td>
</tr>
<tr>
<td>2.</td>
<td>Environment</td>
<td>Puzzle space determined by N (columns) and M (rows), always with a single empty space for tiles to slide into. Numbers range from 1 to ((N\times M) - 1). Initial state arrangements must be derived from Goal state arrangement or else there will not be possible solutions.</td>
</tr>
<tr>
<td>3.</td>
<td>Actuators</td>
<td>Tiles are moved into the empty space, either from Top, Bottom, Left or Right of the empty space.</td>
</tr>
<tr>
<td>4.</td>
<td>Sensors</td>
<td>Fully software, so the agent will have full view of the puzzle space.</td>
</tr>
</tbody>
</table>

### TABLE II. ENVIRONMENT ANALYSIS – 8 QUEEN PUZZLE

<table>
<thead>
<tr>
<th>Sl No</th>
<th>Environment Characteristics of Puzzle</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.</td>
<td>Performance</td>
<td>Arrangement of 8 queens in an 8x8 puzzle space of an 8x8 game board. Main performance gauge is from the least number of moves to solve the puzzle.</td>
</tr>
<tr>
<td>6.</td>
<td>Environment</td>
<td>Puzzle space determined by the 8x8 game board. Initial state is an empty board and final state is the board with 8 queens placed, none facing another.</td>
</tr>
<tr>
<td>7.</td>
<td>Actuators</td>
<td>Each queen is placed, one after the other on a square on the board in a position that does not face any other.</td>
</tr>
<tr>
<td>8.</td>
<td>Sensors</td>
<td>Fully software, so the agent will have full view of the puzzle space.</td>
</tr>
</tbody>
</table>

### 3.5 The Ecosystem

This paper attempts to demonstrate the implementation of Breadth First Search, Depth First Search, A*, Best First Search and Hill Climbing algorithms for solving a sizeable sliding puzzle and for solving an 8 queen puzzle. An Object Oriented Approach is applied for the N Puzzle in order to modularize the application so that different search algorithms (encapsulated in different classes) can be made to work with the same interface. For the N puzzle, the input parameters and output results are to be presented in a Text File. We will also assume that the input parameters contain only valid configurations, implying that the Initial State is solvable, only if it can be done in

---

**Figure.** 2. Puzzle transition graph for a 3x3 N puzzle

**Figure.** 3. The Search Tree

### 3.4.2 The 8 Queen Puzzle

The 8 queen is very similar to the N puzzle, the solution space defined by the 8x8 game board on which queens can be placed. Queens can move vertically, horizontally or diagonally, both in the forward and backward directions and the objective is to place all the 8 queens on the board so that they don’t face each other. Unlike the N puzzle, the state space in this puzzle is limited to the 8x8 board and its combinations. A solution state will start from the initial state and transition through one of the possible states and will be considered complete when it reaches the goal state. An example search tree is shown in figure 4. The algorithms explored in this paper will traverse the Search Tree in different ways to find the Goal State from the Initial State.
reverse, to reach the initial state starting from the Goal State. Otherwise, the N puzzle is unsolvable. The 8 Queen

Search Algorithms will have their own unique flowcharts, which will be detailed during their own specific sections of this paper.

A separate file reading and writing facility will help in parsing the input parameters as well as outputting the process logs as a new results file. This design enables the different search methods to use the same interface for extracting parameters, processing and generating the results log. The application was developed in C# using the Visual Studio .NET 2013 IDE.

4 The Algorithm Implementations and Experiments

There are 5 different search algorithms that are explored in this paper, namely the Breadth First Search, Depth First Search, A*, Best First Search and Hill Climbing. The first two are Uninformed techniques while the others are Heuristics based. The first part will discuss the N Puzzle solution and the second part will discuss 8 Queen Puzzle solution. The theory, software implementation and results of each of

Figure 4. Breadth First Search – 8 Queens

Figure 5. Flow chart test application application

3.6 System Design Overview

The flow chart in Figure 4 illustrates the top-level design of the application used to run the tests. The

146
the AI techniques in consideration are explained in the following sections.

4.1 N Puzzle

4.1.1 Breadth First Search

The Breadth First Search is an uninformed type algorithm, so it does not start with full knowledge of the entire State Space. Instead, it builds its own memory of the State Space by remembering all the explored nodes that it passes through. In addition, the only way for the BFS to know when to stop is by finally arriving at a node that has the same state as the Goal Node. The BFS traverses the Search Tree by uncovering the frontier one level at a time. The sequence is illustrated in Figure 5.

![Breadth First Tree Traversal – N puzzle](image)

The algorithm strategy can be explained plainly in the following steps:

**Step 1**: Begin by making the Root Node as the Active Node.
**Step 2**: Add the Active Node into the Solutions List.
**Step 3**: Check if the Active Node is the Goal Node.
**Step 4**: If it is the Goal Node, then go to Final Step.
**Step 5**: If not, derive all unexplored successors and add them into the Queue.
**Step 6**: If the queue is empty, go to Final Step.
**Step 7**: Else, take the next element on the Queue as Active Node, and repeat Step 2.

**Final Step**:
Trim all Solution List nodes that do not lie between the Goal Node and the Root Node.

Examining the steps, it can be concluded that the Queue used to arrange the order of nodes examined, determines the horizontal-first motion of expansion (First In, First Out). The configuration input parameters are maintained for all 5 algorithm tests to ensure a level testing condition. An excerpt from the results (shown as the following) shows that for a 4 move solution, it has expanded 23 nodes. The reason for the 22 drops was that the Successor trimming also removes the state that the active node is transitioning from. This shows that while BFS can find the shortest path to the Goal State, it has to expand quite a lot of nodes and remember them in the process.

**Result Summary**:
- Nodes Expanded = 23
- Nodes Dropped = 22
- Solution Length = 4

4.1.2 Depth First Search

The Depth First Search algorithm shares the similar mode for Goal identification and Search Tree traversal, but not in its motion of expansion. Instead of going Horizontal first, the DFS method chooses a branch and expands it continuously until it reaches a dead end. If a goal has not been found, it will backtrack to the next available alternate branch and repeat the process. This motion is illustrated in Figure 6.

![Depth First Search Tree Traversal](image)

As mentioned in the BFS strategy, the motion of expansion is influenced by the type of data structure used. Instead of First-In-First-Out, DFS will use a Stack (First-In, Last Out). The rest of the algorithm is exactly the same as BFS. But pure DFS has a high possibility of infinite depth, so it will need a depth limit to force the agent to backtrack. This will transform the DFS into Depth-Limited Search.
As the difference between DFS and BFS is only the frontier data structure, the programmatic flow chart for DFS is exactly the same as BFS, except that the Frontier is now a Stack where BFS uses a queue. While the Depth Limited Search do not consume as much memory as BFS when it expanded 4743 but dropped 4832 nodes, it takes variable amount of time to search, depending on whether it found a branch with the goal at the end. Also, the DFS is not a complete method because it can get lost expanding a branch without reaching an end or goal. In this experiment, it took much longer than BFS with 4743 nodes expanded and the solution is inefficient with 4730 moves. The summary result, using the same configuration as the BFS test is as the follows:

Nodes Expanded = 4743
Nodes Dropped = 4832
Solution Length = 4730

4.1.3 A* Search

The A* Search is a Heuristics-based Informed search algorithm. The term ‘Informed’ refers to the use of State Space awareness in the form of the Heuristics Function. All informed search methods rely on knowing how much a particular node have in common with the Goal Node. Put simply, Heuristics $H(n) = \text{Difference (Goal State,Current State)}$. The higher $H(n)$ is, the less ideal the alternative.

A heuristic search algorithm will basically use the Breadth First Search’s horizontal motion of expansion, but will only expand leaves that are most ‘ideal’, i.e. having the smallest $H(n)$. The A* algorithm takes this further by incorporating the Path Cost into the Heuristics function for every node. So now, not only are nodes chosen according how closely they resemble the Goal State, but also how far they are from the Root Node. An example of the A* traversal can be seen as Figure 7. The Blue numbers are the Path Costs and the Green numbers are the Heuristics function.

The overlying structure of the A* search algorithm is the BFS, but significant changes has been made to the generation of successors and queue management which involves the heuristics and path cost calculations. These details are reflected in the flowchart shown in Figure 8.

![Figure 8. A* traversal of the 8-Puzzle search tree](image)

It is observed that the A* algorithm managed to find the best solution like BFS in 4 steps, but has managed to do so with only 4 nodes being expanded. Also, it has managed this with only generating 9 nodes and dropping 3, presumably due to them being too far different from the Goal State. It is more memory efficient compared to DFS and is faster than BFS but requires more computation power as each node is not only evaluated for it matching the Goal State, but also how much difference between them and how far down the tree it has gone.
The summary of the test results, using the same configuration script for BFS and DFS, are as follows:

<table>
<thead>
<tr>
<th></th>
<th>Nodes Generated</th>
<th>Nodes Expanded</th>
<th>Nodes Dropped</th>
<th>Solution Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best First Search</td>
<td>9</td>
<td>4</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Hill Climbing Search</td>
<td>9</td>
<td>4</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

4.1.4 Best First Search

The Best First Search is also a type of informed search method, in that it relies on the Heuristics Function for its working. Unlike the A* method, which requires higher computational capacity, the Best First Search only compares the heuristic value of nodes, ignoring the path cost. Theoretically, this will cut down the required computations of the A* by half, in expense of the possibility of having endless loops or very expensive total path cost of solutions. Other than that, it operates over a Breadth First Search’s motion of expansion.

The Best First Search is also known as the Greedy Best First Search. When it expands nodes and gets a list of possible successors that were not explored before, it will derive the heuristics value of each of the successors and pick the best one to expand. The other leaves will be unexplored. It is not optimized, as the cheapest path may involve going through heuristically suboptimal nodes but yet have less path cost. Other than this, the Greedy Best First Search has the same mode of operation as the A*.

The Greedy Best First Search only differs from the A* Search at the point of determining the fate of possible Successors. Instead of taking into account calculations of path cost, it directly decides on which leaf to expand by using only the H function. In this particular instance, its findings and performance match the A* test, because the solution only spans 4 steps. On longer solutions and more complex Search Trees, the same cannot be expected of the Greedy BFS search as there is a higher possibility of it getting stuck on a lost branch with initially low H values. By supplying the same configurations as the previous tests, the following results are observed:

<table>
<thead>
<tr>
<th></th>
<th>Nodes Generated</th>
<th>Nodes Expanded</th>
<th>Nodes Dropped</th>
<th>Solution Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best First Search</td>
<td>9</td>
<td>4</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Hill Climbing Search</td>
<td>9</td>
<td>4</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

4.1.5 Hill Climbing Search

Similar to the Greedy Best First Search, the Hill Climbing method borrows from the A* Search but simplifies it even further. Where the Greedy search only bases its decisions on the Heuristics Function, Hill Climbing works in the same way but totally disregards memory of explored nodes. Therefore, it travels down the Search Tree by selecting the successor with the cheapest heuristics value, without retaining memory of explored states.

This will ensure that the heuristics technique functions with minimal use of memory, least computation possible but still retain the advantage of an informed method of solution finding. The downside of Hill Climbing is that due to the absence of memory, resulting in the possibility of repeating the same states and getting stuck in some state of local maxima.

![Hill Climbing Search Algorithm](image)

**Figure. 10. Hill Climbing Search Algorithm**

While the changes between the Greedy BFS and A* are only in the selection of successors, Hill Climbing
greatly simplifies the Successor generation and selection process, as illustrated in the flow chart shown in Figure 9.

From the results, it seems that the Hill Climbing technique manages to find the solution in the same time as Greedy BFS and A*, but dropped more nodes than the other two. Also, it shares even more risk of failure than Greedy search in cases of longer solutions or more complex problems. By supplying the same configurations script, the following results are observed:

- Nodes Generated = 5
- Nodes Expanded = 4
- Nodes Dropped = 7
- Solution Length = 4

The following conclusions can be made based on the results as we observe in table II and figure 10.

**4.1.6 Conclusion and Future work**

A summary of the comparison of the 5 AI techniques are as observed in table II. The findings listed in the table are plotted in a comparison chart in figure 10. The values of Depth first search has been avoided in the chart since the value is well out of range.

**TABLE III. SUMMARY OF RESULTS**

<table>
<thead>
<tr>
<th>SI No</th>
<th>Summary of AI Algorithm Results on the N Puzzle</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Algorithm</td>
</tr>
<tr>
<td>1</td>
<td>Breadth First Search</td>
</tr>
<tr>
<td>2</td>
<td>Depth First Search</td>
</tr>
<tr>
<td>3</td>
<td>A* Search</td>
</tr>
<tr>
<td>4</td>
<td>Greedy BFS Search</td>
</tr>
<tr>
<td>5</td>
<td>Hill Climbing</td>
</tr>
</tbody>
</table>

**Figure. 11.** Comparisson of the Search Algorithm Observations

**Figure. 12.** Depth First Search – 8 Queen Puzzle

- **Breadth First Search:** More complete and concise uninformed technique that manages to find the best solution using minimal
computation but suffers from intensive memory use. BFS will be most recommended for small dimension sliding puzzles, but for the expandable N*M type with exponentially scaling complexity, BFS becomes less efficient and can run out of memory before completing long solutions.

- **Depth First Search**: Explores branches individually before backtracking. It is actually good for long solutions, but only if it is lucky enough to start on a branch with a possible Goal State. The sliding puzzle has single definite solutions, so it will not be optimal to use DFS. A complete search method is preferable.

- **A* Search**: By utilizing both heuristics values and path costs, the A* search manages to find the shortest solutions with moderate memory and time performance. A lighter method will be preferable if it was a small puzzle. However, this dynamically up-scaling puzzle will benefit more from the A*'s complete and heuristic approach.

- **Greedy BFS**: The Best First Search also uses the A*'s heuristic method, but negates the path cost. For this particular problem, the Best First Search is quite effective, matching the performance of the A* search on shorter solutions. For longer solutions, the A* is safer but requires more computations.

- **Hill Climbing**: This is the most memory efficient heuristics approach but has high risks of failure due to local maxima issue and disregard of memory, especially for moderate to long solutions. This is not recommended for the sliding puzzle problem as there is a high chance for the failure conditions to present themselves.

In conclusion, the best approaches to apply to this dynamically scaling sliding puzzle will either be the A* or the Greed Best First Search. Greedy BFS is more memory efficient and matches the performance of A* for shorter solutions. For longer and more complex solutions, the A* is the best choice, in order to avoid possibilities of getting a suboptimal solution due to the Greedy BFS’s disregard for path cost computations.

### 4.2 The 8 Queen Puzzle

Additional study on the 5 AI techniques was carried out to generate the solution for the classic 8 Queen Puzzle. Since the algorithms have already been explained in the sections above, this section will focus on the application on the 8 Queen Puzzle. The experiments will compare the convergence of the algorithms on the 8 Queens puzzle to extend the results of the AI techniques on the solution for the N puzzle.

**4.2.1 Breadth First Search (BFS)**

Breadth first search's initial state is an empty board. After the initial step, the next step would be to place...
The first queen. This is done row by row. For example, the first initial queen will be placed at row 1 column 1. The next initial queen is placed at row 1 column 2 etc. The algorithm then checks each square row by row to see if there's any squares which do not crash with the initial queen placed or any other queens on the board, whether upwards, downwards, left, right or diagonally. When that square is discovered, a queen will be placed there and the algorithm continues to search for such squares until all 8 queens are placed. If 8 queens fail to be placed on the board, the board will be cleared and a new initial queen will be placed. The goal state is reached when 8 queens have been placed on the board without any queen facing any of the other queens on the board.

4.2.2 Depth First Search (DFS)

Depth first search is similar to breadth first search. The only difference is that when placing the initial queen, in goes column by column. And also, the search for squares with no crashes is done column by column.

In the diagram for Depth first search (figure 12), there are three examples given ach with a different initial queen. For the experiment, the depth first search produces a board with eight queen when the initial queen is placed on row 1, column 6.

4.2.3 Best First Search

For best first search, the $h(n)$ will be the total number of crashes for the board. In this algorithm, the crash value for each square is calculated. For example, if there are no queens that can attach that square, it will be 0. If there are two queens, one above it and one to the left of it, the crash value for that square will be 2.

The best first search will insert an initial queen. Then, it will check to see if there are any squares containing 0 crash value for each column. If there is a square containing 0 crash value ($h=O$), it will insert queen in that square and move on to the next column. However, if after checking all the squares in that column and there are no squares containing 0 crash value, the algorithm will reset the board and start anew with a new initial queen position. Therefore, the algorithm can obtain 8 queens by making sure that the value remains as $O$(lowest possible value) by having no crashes between each queens at all. Figure 13 shows the success path where there are no crashes between queens as well as a fail path. As shown, in column six of the latest state, there are no squares with $h=O$ and choosing a square with crash value higher than 0 will cause the board’s number of crashes to increase. Therefore, the board is discarded.

4.2.4 Hill Climbing Search

Hill climbing search for this 8-Queen puzzle, in order to reach the goal state where $h = 0$, it will continue to loop to find moves in the direction of decreasing $h(n)$.  

![Figure 1. Hill Climbing Search – 8 Queen Puzzle](image-url)
It will terminates when there is no lower \( h(n) \) than the previous ones. By looking at the diagram (Figure 14), the hill climbing search will randomly generate 8 random placement of the queen in the 8x8 board after the initial state it will then calculate the \( h(n) \) and then during the next state it will swap the Q position column by column in search of the \( h(n) \) that is lesser than the previous \( h(n) \) until it reach \( h=0 \).

Hence, based on the evaluation function \( f(n)=h(n) \), so the results will be \( f(n)1=0 \). The board will terminate if there is no \( h(n) \) that is less than the previous \( h(n) \). When board is clear, a new random placement of the queens is placed again and the process is repeated until it reaches the goal state.

### 4.2.5 A* Search

The goal state for every a-Queen solution is \( h(n) \) is equal to 0. Thus, for A* search, the algorithm used can generate 8 random placement of the queen in the 8x8 board without any conflict between the Queens. As shown in the diagram below, once the A* search found the first solution that has \( h=0 \), it will choose that as the goal state even though there are other solution available.

According to the evaluation function, \( f(n)=g(n)+h(n) \). \( h(n) \) is the number of conflicts from goal state and \( g(n) \) is the number of conflicts from initial state. So at the goal state, when \( h(n)=0 \), and since A* search places every Q without any conflict between each queen, hence \( g(n)=0 \). Based on the evaluation function, \( f(n) \) will be 0.

### 4.2.6 Conclusion and Summary

After exploring the 5 algorithms to solve the 8-Queen puzzle, we have come to the conclusion that A* is the best search algorithm for the 8 Queen Solution. Breadth First Search, Depth First Search and Best First Search can only generate one board which same each time it is run. Hill Climbing on the other hand, attempts to swap the position of the queens on the board and may discard many boards before it discover correct solution. A* algorithm is able to reach the goal state of 8 queens by positioning 8 queens on the board so that there is only one queen per column and one queen per row where the total crashes between queens is 0. In addition to this, it will guarantee to produce a board which solves the 8-Queen puzzle. The board generated can also be different each time the A* algorithm is used and need not be static like those generated using BFS, DFS and Best First Search. The summary of the result is as shown in Table IV.

<table>
<thead>
<tr>
<th>Comparison between algorithms</th>
<th>BFS</th>
<th>DFS</th>
<th>Best First</th>
<th>Hill Climbing</th>
<th>A*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min Cost by Back-tracking</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Complete</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Time</td>
<td>( O(b^{d+1}) )</td>
<td>( O(b^m) )</td>
<td>( O(b^m) )</td>
<td>( O(\infty) )</td>
<td>( O(b) )</td>
</tr>
<tr>
<td>Space</td>
<td>( O(b^{d+1}) )</td>
<td>( O(b^m) )</td>
<td>( O(b^m) )</td>
<td>( O(b) )</td>
<td>Stores all nodes</td>
</tr>
</tbody>
</table>

**Summary, Conclusion and Future Work**

As we have seen in the studies of on the N Puzzle as well as the 8 Queens puzzle, the A* is seen to perform...
best, with its heuristics and faster convergence at cheaper cost and therefore conclude that the A* search is best suited for solving issues of this nature. In both cases, A* gave the best results and also has the possibilities to be applied towards issues of growing dimensionalities.

Further studies can be carried out to measure the rate of convergence and do a parallel comparison for both puzzles. Also, further improvement can be implemented in the form of morphing algorithm that can opt to repeat the search in A* if the initial attempt with Greedy BFS turns out unusually long solution. Also, a fail-safe function to switch from A* to Greedy BFS can be implemented, to kick in during events of critically reduced memory conditions.

5 Acknowledgements

We acknowledge the contributions of Mark Tee Kit Tsun Carmella Sim Lee Yoong, Carmen and Lo Xiang for the contributions in coding for the algorithm implementations.

6 References