Comparative Analysis of AI-Based Algorithms for Solving the N-Queens Problem: A Study on Efficiency and Scalability

Ahmed Khaled 236664 | Hassan Ahmed 229591 | Hazem Hassan 232504 | Ahmed Sameh 236322 | Ahmed Momen 237046

Faculty of Informatics & Computer Science, The British University In Egypt
Cairo, Egypt

Submitted to Dr Amr S. Ghoneim

ABSTRACT

The N-Queens problem is a classical combinatorial challenge that involves placing N queens on an N×N chessboard so that no queens threaten each other. A solution requires that no two queens share the same row, column or diagonal. This report presents a comprehensive comparative study of four of the most popular artificial intelligence techniques. Such as Backtracking, Breadth-First Search, Hill-Climbing, and Genetic Algorithm are used for solving the N-Queens problem. This paper includes detailed analyses of algorithmic strategies, pseudocode, flowcharts, and experimental results. The study does not only compare solution quality and computational efficiency across different board sizes but also investigates parameter tuning effects. The results demonstrate that while traditional methods such as backtracking guarantee solution completeness for small N, heuristic and evolutionary methods offer significant performance advantages as problem complexity increases [2], [3], [7].

KEYWORDS

N-Queens Problem, Breadth-First Search, Depth-First Search, Best-First Search, Hill-Climbing, Backtracking, Brute Force, Heuristic Methods, Genetic Algorithm, Combinatorial Optimization, Artificial Intelligence.

I. INTRODUCTION AND OVERVIEW

A computer science search algorithm is any algorithm employed to find a solution to the search problem, i.e., visiting possible solutions in a search space. Computational problems are classified as tractable and intractable meaning that they can be solved in polynomial time or exponential time, respectively [8]. Since the N-Queens problem is assumed to be intractable it requires advanced algorithms due to its complexity. For that, several data structures are used to solve this problem, including arrays, graphs and trees, and priority queues [4].

There have been some studies recently that considered various methods and algorithms specifically to tackle the N-Queens problem. Ensuring computation completeness and speed. These are breadth-first search (BFS), depth-first search (DFS), greedy best-first search, and hill climbing [1]. In addition to classical algorithms such as backtracking and brute force that provide a guaranteed optimal solution every time but are less efficient than necessary as they become computationally irrelevant in practice for large N. In addition to a heuristic and evolutionary method such as genetic algorithms (GA) that offer faster solutions by way of intelligent search in the problem space. Such methods greatly increase efficiency, making them suitable for large-scale problems of the N-Queens Problem [4], [7].

This paper presents a detailed comparative analysis of the above methods and their theoretical backgrounds, implementation details, computational cost, and performance measures.

I.I PROBLEM STATEMENT

N-Queens problem is the position of N queens on an N×N chessboard so that no queen is attacked by another, i.e., no two queens are placed in the same column, row, or diagonal. Being discovered in the 19th century, it has been used as a benchmark to compare search and optimization algorithms in artificial intelligence because it is NP-hard in nature [7]. The problem becomes computationally expensive with the rise in the

exponential number of search spaces with the rise in N because there are exactly 4,426,165,368 possible N = 8 Queens positions on an 8×8 board but only 92 solutions [4]. This is the reason for the need for the development of effective solution strategies. However, the problem has solutions if and only if N > 3.

II. LITERATURE REVIEW

The N-Queens problem literature will be a large collection of diverse methodologies and techniques as discussed in the overview above. Within this review, the major techniques and strategies followed while solving the problem will be given a detailed explanation. The following sections provide a detailed explanation of these techniques:

Breadth-First Search (BFS):

BFS is an uninformed search algorithm. The algorithm traverses nodes level by level, ensuring the first solution to be the earliest. All the paths are traversed breadthwise, and hence it is complete and optimal in unweighted graphs. BFS uses the First In, First Out (FIFO) principle using the implementation of a queue to store nodes, in which all the nodes of the current depth are processed before processing the next level. BFS requires huge memory space as it stores and keeps all the generated nodes, and hence it results in exponential space complexity, and hence not practical in large search space. Time complexity of BFS is O(V+E), where V is vertices and E is edges. In spite of the limitation and drawback, it is a basic algorithm in artificial intelligence and graph traversal, used in shortest path finding, and AI-based pathfinding in computer games [5], [6], [7].

Depth-First Search (DFS):

Depth First Search (DFS) is a simple graph traversal algorithm that goes as far as possible along a branch before returning. It begins at a source vertex, DFS goes to the closest unvisited neighbors recursively, then goes to it, until all vertices are visited. DFS requires less memory and is used in pathfinding and connected components. The time complexity of the DFS algorithm is O(V + E), where V is the number of vertices and E is the number of edges. [4], [5].

Best-First Search (BeFS):

Heuristic-based strategy that explores the paths by selecting the best choice based on the heuristic function. It finds the goal quickly by selecting the nodes with the minimum estimated cost. Though quicker than uninformed search, its efficiency depends on the heuristic's quality; an inefficient heuristic can lead to inefficiency and yield suboptimal solutions. Utilization of heuristics like the number of attacking pairs helps in the ordering of the solutions in the optimization problems. A good heuristic is one that is able to prune the search space well though accuracy is also important since misleading heuristics can mislead the search which may yield suboptimal solutions [5].

Hill-Climbing Search (HCS):

Hill Climbing is a simple heuristic search optimization method starting with a beginning solution and changing from the existing solution by taking improvements in the objective function. The method is effective in getting a local optimum but weak in getting a global optimum. This is because the system will remain on plateau regions or steep hills. To attempt to remove this issue, Stochastic Hill Climbing introduces a randomization method in order to leave local optima [4], [6]. Additional exploration can be made if methods like random restarts are used. Hill Climbing methods are vastly used despite their flaws and are being widely implemented in real-time applications such as game AI, robotics, and combinatorial optimization [1].

Backtracking Algorithms (BT):

Backtracking is a systematic search technique that constructs solutions incrementally and eliminates bad decisions, thus trimming the search and being more efficient than brute force methods. It is most appropriate for problems like the N-Queens problem, where queens are added row by row, and row, column, or diagonal conflicts are backtracked on the spot. Unlike brute force, which attempts to search for all permutations in an exhaustive manner. Its time complexity grows exponentially, and thus it is not feasible for large instances. While it guarantees all possible solutions by systematically exploring feasible configurations, backtracking remains computationally intensive [3], [8]. To avoid this, much research has been concentrated on optimizing the algorithm with the use of heuristic improvements. This new approach includes adding queens one by one from the first row and backtracking the moment there is a conflict. This clarifies conflicts earlier and minimizes recursions. The improved backtracking method derives a solution at a lower computational cost than the computational cost of a genetic algorithm for the average N values in question [10].

Brute Force (BF):

Brute Force is a simple algorithmic solution that selects an acceptable solution by trying and checking every and any possible solution. Brute Force attempts every queen position on an N×N board to find a suitable position in the N-Queens problem, the only condition being that no queen is attacking another queen.But at the cost of time complexity. The exponential increase in the search space doesn't make it feasible for large N. For small problem sizes, i.e., N≤12, BF can easily find solutions. But for larger N's render them inferior to heuristic-based methods like Genetic Algorithms for medium and large problem sizes. Though inefficient for hard problems, BF is still a benchmark to check solutions due to its deterministic and exhaustive nature [7], [8].

Genetic Algorithms (GAs):
GAs are natural selection-based optimization techniques that provide solutions through selection, crossover, and mutation. Candidate solutions are coded similar to chromosomes to represent the positions of queens [2], [9]. Permutation encoding is commonly used in order to avoid putting two queens in the same row or column. Fitness functions score candidates according to the number of non-attacking pairs, with optimal chromosomes being selected using techniques like tournament selection. Partially Mapped Crossover combines genetic material, with mutation introducing diversity to avoid local optima [3], [8]. GAs are especially suitable for medium to large values of N because they can balance exploration and exploitation. GAs is suitable for NP-hard problems like N-Queens, using permutation-based representations to minimize conflict, with parallelization being used for scalability in combinatorial optimization [2], [8], [7].

III. Methodology

This section will include details about each selected approach and technique. Mentioning the steps required to execute them and describing the expected input and output of each method. This will also include pseudocode, flowcharts, and diagrams that illustrate how each method works for this specific topic being N-Queens.

III.I Breadth-First Search (BFS)

The way this algorithm works is that it visits all nodes traversing in a horizontal approach. This can be explained through a few straightforward steps. However, before the explanation, let's first examine the structure of a tree. Figure 1 will serve as the reference for our explanation.

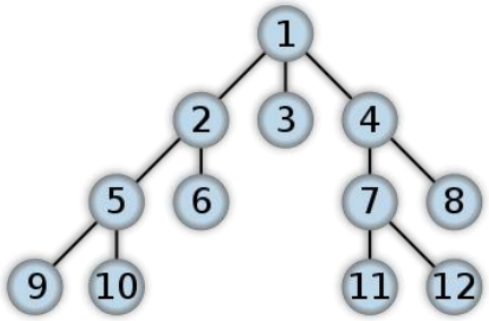

Figure 1. Breadth First Search – Tree Example [6].

Step 1: Set the root node as the active node.
Step 2: Add the active node to the solution array.
Step 3: Check if the active node is the goal node, then proceed to the final step if this condition is true.
Step 4: if not, find all unvisited child nodes from the current node and add them to a queue for further exploration.
Step 5: if the queue is empty, proceed to the final step.
Step 6: if not, set the next node from the queue as the active node and repeat again starting from Step 2.

Final Step: Only keep the nodes that are connected from the root to the goal, and remove other nodes.

By following these steps, we can see that the queue determines the order by which the nodes are explored using a FIFO approach. This means that nodes at the same level are explored first before moving deeper into the tree. This method ensures that BFS will always find the shortest path, however it is memory intensive as it requires storing a large number of nodes inside an array and a queue [6].

III.II Depth-First Search (DFS)

DFS uses a similar approach as BFS. However, the main difference is the expansion strategy, unlike horizontal expansion, DFS uses vertical expansion meaning that it keeps diving deeper into the tree until it reaches a leaf node, then backtracks and continues exploring. Another difference between BFS and DFS is that DFS uses a stack featuring Last In, First Out. A limitation that DFS faces is the infinite depth problem, meaning that it could keep expanding indefinitely, however, this could be fixed by introducing a depth limit [6].

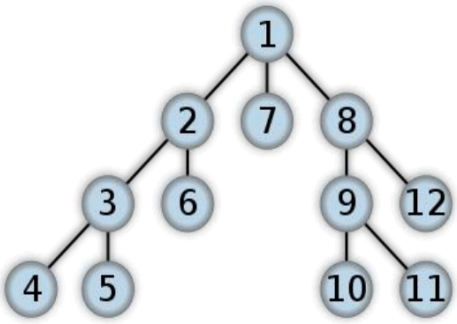

Figure 2. Depth First Search – Tree Example [6].

III.III Best-First Search (BeFS)

Best First Search makes use of the heuristic function to choose the most promising node at each step but does not consider the actual path cost. Using this method could look like this:

Step 1: Set the root node as the active node.
Step 2: Check if the current node is the goal state. If yes, stop.
Step 3: if not, expand the current node and generate its successors.
Step 4: Evaluate each successor using the heuristic function.
Step 5: Select the successor with the lowest heuristic value.
Step 6: Move to the selected node and repeat from Step 2.

Step 7: If no more nodes can be expanded, terminate (may result in failure) [6].

III.IV Hill-Climbing Search (HCS)

HCS further simplifies the search process by not keeping track of previously visited states. While this reduces memory usage, it significantly increases the risk of getting stuck in local maxima, flat regions, or sharp slopes, leading to possible failure in more complex search spaces. This could be implemented by following the same exact steps of the BeFS, the only difference is in step 6, here after moving to the selected node it should delete the memory of past nodes [6].

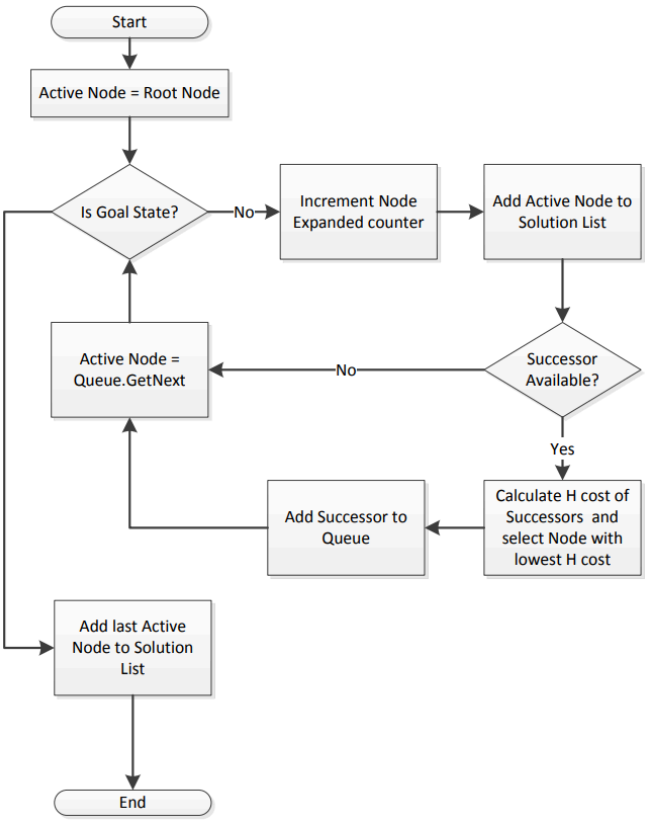

Figure 3. Hill Climbing Search Algorithm [6].

III.V Backtracking Algorithms (BT) & Improved Version

Backtracking solves N-Queens problem by placing Queens row by row and backtracking to solve conflicts. Here is what an algorithm for solving N-Queens Problem by backtracking could look like:

Step 1: Place a queen in the top row, check which column and diagonal it covers.
Step 2: In the next row, place a new queen, while making sure it doesn't cover the same column or diagonal with the first one, once again, check the columns and diagonals it covers and move to the next row to place another queen.
Step 3: In the case that no position is free in the next row, go back to the previous row and put the queen in the next available position in its row, the process starts over again until reaching a solution for the correct arrangements for all queens [8].

More optimized versions exist, they use heuristics to decrease the number of recursions, Here is what an algorithm for solving N-Queens Problem by an imporved backtracking could look like:

Step 1: Begin with an empty chessboard and start placing queens from the first row.
Step 2: For each row, a queen is placed in a column such that it does not conflict with queens already placed, using shortcuts (like lists for columns and diagonals) to check quickly.
Step 3: If no valid placement is possible, the algorithm backtracks to the previous row and tries an alternative position.
Step 4: Continue until a complete and valid configuration is achieved [10].

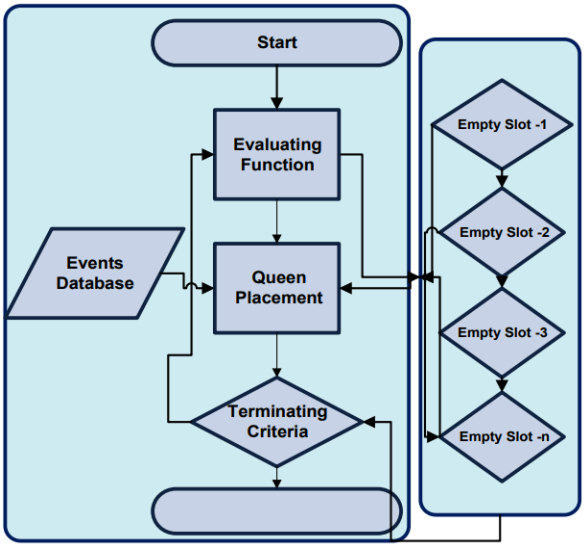

Figure 4. Logical Flow of Improved Backtracking Search [10].

```
1. Procedure BTS
2. Repeat
3. IF All The Placement are Assigned
   Then Exit
4. For All the Periods
5. IF Evaluation (QList, New ) == TRUE
6. THEN
7. Place the New Queen
8. Call BTS(QList[Counter + 1])
9. ELSE
10. Select Succeeding Period
11. End For
12. Until True
```

Figure 5. pseudocode Improved Backtracking Search [10].

III.VI Brute Force (BF)

Brute Force checks every single possible Queen placement, it guarantees that a solution is found as long as one exists. Here is what an algorithm for solving N-Queens Problem by brute force could look like:

Step 1: Put the queen in the highest row.
Step 2: Place another queen in the following row.
Step 3: If the last queen added is sharing the same column or same diagonal with the first one, place the queen in the next free place in that row, otherwise move on to the next row to place the next queen.
Step 4: If there is no free place in the next row, move to the previous row and move the queen to the next available place in its row. The process starts again and loops until reaching a solution.

This solution is effective but becomes very impractical for large cases [8].

III.VII Genetic Algorithms (GAs)

GAs solve optimization problems through selection, crossover, and mutation. In N-Queens, permutation encoding prevents conflicts, while fitness functions and genetic operators enhance solution quality. These can be done using the following steps:

Step 1: Generate 92 random solutions on an empty chess board.
Step 2: Evaluate each candidate using a fitness function that calculates the number of non-attacking queen pairs as shown in Figure 8.
Step 3: Rank the chromosomes based on their fitness values.
Step 4: Crossover is applied to produce offspring, and mutation is applied based on adaptive rates to introduce diversity.
Step 5: The process is iterated until a solution is found or a stopping condition is reached.

```
Function fitness (chromosome) {

t1 = 0;   //number of repetitive queens in one diagonal while seen from left corner
t2 = 0;   //number of repetitive queens in one diagonal while seen from right corner
size = length (chromosome);
for i= 1 to size:
   f1 (i) = (chromosome (i)-i);
   f2 (i) = ((1+size)-chromosome (i)-i);
end
f1=sort (f1);
f2=sort (f2);
for i=2 to size:
   if(f1 (i) == f1 (i-1)) //checks whether two Queens are in same diagonals seeing from left corner or not
      t1 = t1+1;
   end
   if(f2 (i) == f2(i-1)) //checks whether two Queens are in same diagonals seeing from right corner or not
      t2 = t2+1;
   end
end
fitness_value = t1 + t2;
return fitness_value;
}
```

Figure 6. pseudocode for the fitness function used in the Adaptive Genetic Algorithm [9].

IV. EXPERIMENTS

IV.I Tree-Based Experiment

Tree-based trials attempt algorithms that search within a structured search tree in which every node represents a distinct problem state. This was used in the 8 Queen problems by starting the search from a beginning state and progressing step by step from nodes to build candidate solutions. Trials were conducted under controlled test conditions in which all the algorithms were provided with the same input parameters for comparative purposes. The number of nodes generated, built, and rejected, as well as the solution length, were some of the areas that were utilized in testing the performance of each algorithm.

IV.I.I Breadth-First Search (BFS):

The experiment was conducted using a 3x3 board as described in figure 1, the algorithm was implemented to explore the search tree level by level starting from the root node moving all the way till the goal node [6].

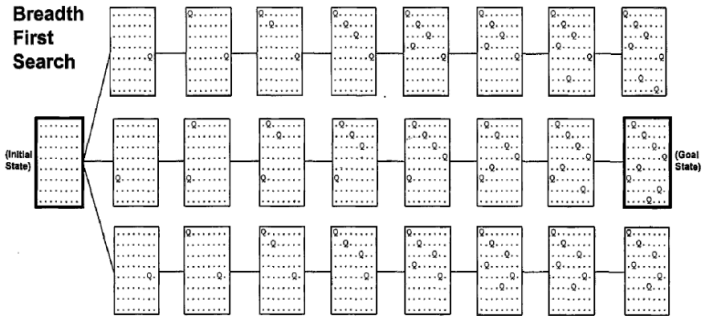

Figure 7. Breadth First Search – 8 Queens Problem [6].

The results that were measured were the number of nodes expanded, dropped, and the solution length.

IV.I.II Depth-First Search (DFS):

The experiment used the exact same layout as the BFS algorithm, here the difference is that DFS searches by going as deep as possible and backtracking after reaching a leaf node as shown in figure 2. [6].

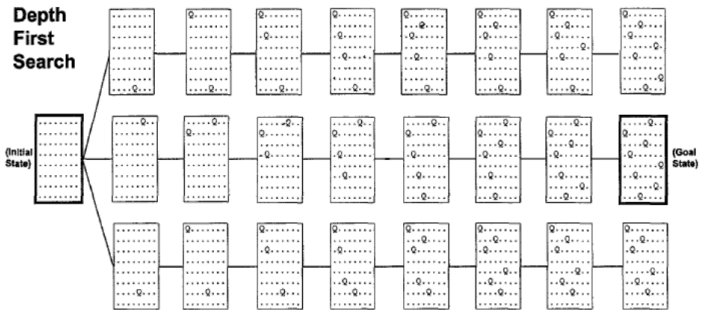

Figure 8. Depth First Search - 8 Queens Problem [6].

The results that were measured were the number of nodes expanded, dropped, and the solution length.

IV.I.III Best-First Search (BeFS):

The experiment was conducted on the N-Puzzle using a tree-based search approach similar to DFS and BFS. However, unlike BFS, BeFS uses a heuristic function to determine the most promising node for expansion. The results were as follows [6]:

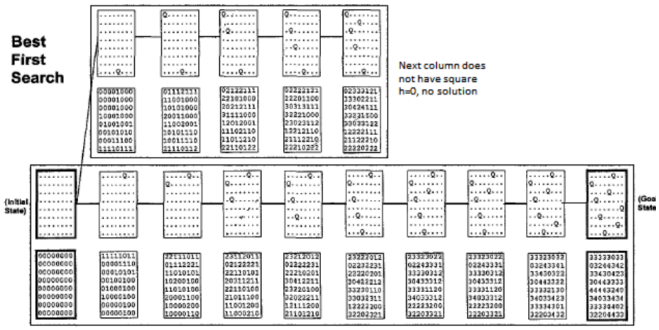

Figure 9. Best-First Search - 8 Queens Problem [6].

The results that were measured were the number of nodes generated, expanded, dropped, and the solution length.

IV.I.IV Hill-Climbing Search (HCS):

Hill Climbing was also tested on the same N-Puzzle as shown in figure 2. Although similar to Best-First Search in using a heuristic function, Hill Climbing differs by completely discarding memory of previously visited nodes. The results were as follows [6]:

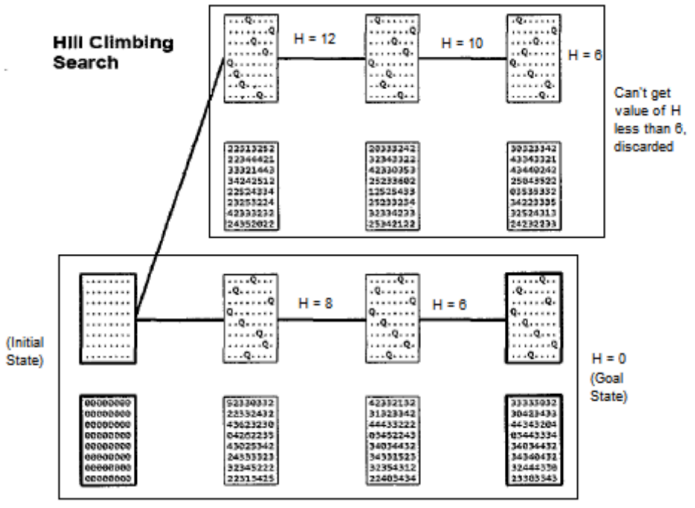

Figure 10. Hill Climbing Search - 8 Queens Problem [6].

The results that were measured were the number of nodes generated, expanded, dropped, and the solution length.

IV.II Board-Based Experiments

Board-based experiments focus on such algorithms which operate by modifying directly the board configuration and not by traversing a search tree. In board-based experiments, we are tackling the problem by starting with a random or blank board and then modifying the board step by step to construct a solution. We are using various values of N (Board size) in these experiments.

IV.II.I Backtracking Algorithms (BT) & Improved Version:

The implementations of the algorithm were tested on the N-Queens problem as a test case. The Normal Backtracking and the Improved Backtracking implementations were tested on various board sizes such as 4×4, 6×6, and 8×8 to observe scalability. In each test, the initial configuration began with an empty board where queens were placed one row at a time. Each algorithm was tested on different hardware as follows [8], [10]:

Regular BT: Intel Core 2 Duo processor and 8 GB of RAM.
Improved BT: Intel Core i3 processor and 2 GB of RAM.

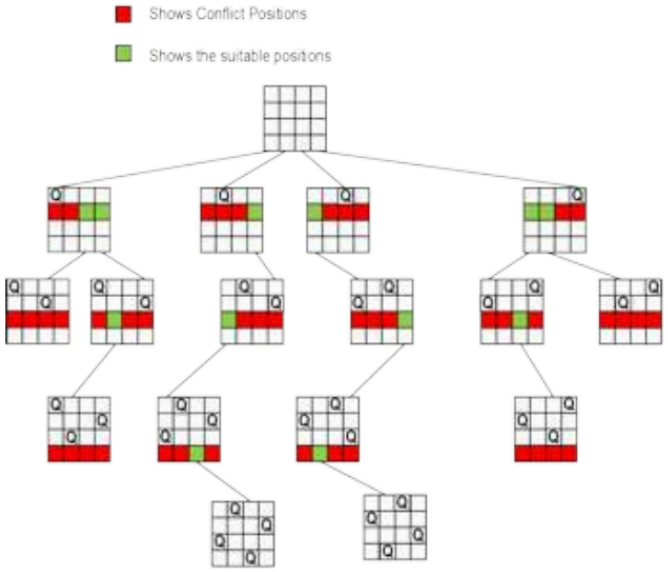

Figure 11. Backtracking Search - 4 Queens Problem [10]

The results that were measured were execution time, accuracy, and scalability.

IV.II.II Brute Force (BF):

The BF algorithm was executed to determine solutions to the N-Queens problem by systematic generation and checking of all possible arrangements of queens on an N×N chessboard. The experiments were performed for small board sizes such as 4×4, 6×6, and 8×8. In every experiment, a starting input of an empty board was used, and the algorithm put queens in every possible

position in a systematic manner to generate all possible arrangements [8].

The results that were measured were execution time, accuracy, and scalability.

IV.II.III Genetic Algorithms (GA):

The study employed a GA to attain the 8-Queen problem's 92 valid solutions. The algorithm was applied to an 8×8 chessboard where each chromosome is a 1D array of 8, where each index is a column and each value is a pointer to the location of the queen in the row. The setup consisted of the formation of an initial population, each with its chromosomes formed through a fitness function of pairs of non-attacking queens, choosing the parents through tournament selection, crossover in offspring generation, mutation prior to repeated attempts until a terminal condition or when all possible solutions are obtained [2].

The results were measured in terms of accuracy, iterations required, execution time, and fitness values.

V. RESULTS

In this section we will cover all the results that were found from conducting the previous experiments on each algorithm and will compare some of the algorithms with each other.

V.I Results of Tree-Based algorithms:

These are the results of BFS, DFS, HCS, and BeFS respectively [6].

BFS and DFS Algorithm Results Table [6]

BFS	**DFS**
Nodes Expanded: 23	Nodes Expanded: 4743
Nodes Dropped:22	Nodes Dropped: 4832
Solution Length: 4 moves	Solution Length: 4730 moves

HCS and BeFS Algorithm Results Table [6]

HCS	**BeFS**
Nodes Generated: 5	Nodes Generated: 9
Nodes Expanded: 4	Nodes Expanded: 9
Nodes Dropped: 7	Nodes Dropped: 4
Solution Length: 4 moves	Solution Length: 4 moves

V.II Results of Board-Based Algorithms:

These are the results of BF [8], BT [8], Improved BT [10], and GA [1] respectively.

Brute Force Algorithm Results Table [8]

Number of Queens	Number of Solutions	Execution Time:
1	1	0.0 sec
2	0	0.0 sec
3	0	0.0 sec
4	2	0.0 sec
5	10	0.0 sec
6	4	0.0 sec
7	40	0.054945 sec
8	92	0.164835 sec
9	352	0.714286 sec
10	742	1.978022 sec
11	2680	9.450549 sec
12	14200	58.736264 sec

Backtracking Search Algorithm Results Table [8]

Number of Queens	Number of Solutions	Execution Time:
1	1	0.0 sec
2	0	0.0 sec
3	0	0.0 sec
4	2	0.0 sec
5	10	0.0 sec
6	4	0.0 sec
7	40	0.054945 sec
8	92	0.109890 sec
9	352	0.384615 sec
10	742	1.153846 sec
11	2680	5.164835 sec
12	14200	32.252747 sec

Improved Backtracking Search Algorithm Results Table [10]

Number of Queens	Number of Solutions	Execution Time:
1	1	0.0 sec
2	0	0.0 sec
3	0	0.0 sec
4	2	0.0 sec
5	10	0.0 sec
6	4	0.0 sec
7	40	0.0 sec
8	92	0.0 sec
9	352	0.53 sec
10	742	1.01 sec
11	2680	1.4 sec
12	14200	5.2 sec
13	73712	31.6 sec
14	365596	220 sec
15	2279184	1605 sec

Genetic Algorithm Results Table [2]

Number of Iterations		Solution	
1017	3 5 7 1 6 0 2 4	2894	4 6 0 2 7 5 3 1
1812	6 3 1 7 5 0 2 4	4994	2 4 1 7 5 3 6 0
2417	3 6 0 7 4 1 5 2	455	6 3 1 4 7 0 2 5
8069	4 1 5 0 6 3 7 2	4700	3 5 7 2 0 6 4 1
7784	4 6 0 3 1 7 5 2	7016	5 0 4 1 7 2 6 3
406	3 1 6 2 5 7 0 4	2046	4 2 0 5 7 1 3 6
6325	0 6 4 7 1 3 5 2	373	2 0 6 4 7 1 3 5
1454	5 3 1 7 4 6 0 2	3437	2 5 7 1 3 0 6 4
1263	6 0 2 7 5 3 1 4	172	3 1 7 4 6 0 2 5
129	2 5 1 6 0 3 7 4	755	4 1 3 6 2 7 5 0
4676	3 6 4 2 0 5 7 1	400	3 0 4 7 1 6 2 5
2845	4 7 3 0 2 5 1 6	13	4 2 7 3 6 0 5 1
5188	4 6 1 3 7 0 2 5	1596	5 2 0 7 3 1 6 4
3056	3 1 6 4 0 7 5 2	419	5 1 6 0 3 7 4 2
1046	1 3 6 0 4 2 5 7	1821	4 6 1 5 2 0 7 3
2097	3 6 4 1 5 0 2 7	9710	5 2 6 1 3 7 0 4
4199	1 5 7 2 0 3 6 4	3210	0 6 3 5 7 1 4 2
909	6 2 0 5 7 4 1 3	1399	0 5 7 2 6 3 1 4
3685	5 3 6 0 2 4 1 7	2924	5 2 0 7 4 1 3 6
1548	6 4 2 0 5 7 1 3	128	1 5 0 6 3 7 2 4
861	3 7 0 2 5 1 6 4	8724	3 6 2 7 1 4 0 5
3811	2 5 1 6 4 0 7 3	66	1 7 5 0 2 4 6 3
4333	2 4 7 3 0 6 1 5	1692	7 3 0 2 5 1 6 4
4702	1 6 2 5 7 4 0 3	603	1 4 6 3 0 7 5 2
4082	4 0 7 3 1 6 2 5	9827	6 2 7 1 4 0 5 3
3524	4 6 3 0 2 7 5 1	1782	2 7 3 6 0 5 1 4
720	2 5 3 1 7 4 6 0	26584	5 2 0 6 4 7 1 3
3829	2 5 7 0 4 6 1 3	7314	6 1 3 0 7 4 2 5
12279	6 1 5 2 0 3 7 4	10854	0 4 7 5 2 6 1 3
167	4 1 7 0 3 6 2 5	3072	4 0 3 5 7 1 6 2
8402	4 2 0 6 1 7 5 3	3289	2 5 3 0 7 4 6 1
4958	7 2 0 5 1 4 6 3	23834	3 0 4 7 5 2 6 1
740	5 3 6 0 7 1 4 2	11526	3 7 4 2 0 6 1 5
4053	1 6 4 7 0 3 5 2	859	1 3 5 7 2 0 6 4
641	2 6 1 7 5 3 0 4	2738	5 7 1 3 0 6 4 2
3199	3 1 7 5 0 2 4 6	5181	3 5 0 4 1 7 2 6
6056	3 1 6 2 5 7 4 0	1341	4 1 3 5 7 2 0 6

13955	2 5 7 0 3 6 4 1	1944	5 2 4 6 0 3 1 7
8247	7 1 4 2 0 6 3 5	59389	4 0 7 5 2 6 1 3
2751	2 5 1 4 7 0 6 3	169369	7 1 3 0 6 4 2 5
2975	5 2 4 7 0 3 1 6	8087	4 6 1 5 2 0 3 7
24804	2 4 6 0 3 1 7 5	4817	3 1 4 7 5 0 2 6
37184	5 3 0 4 7 1 6 2	2499	1 4 6 0 2 7 5 3
56132	4 7 3 0 6 1 5 2	3355	3 7 0 4 6 1 5 2
8448	5 2 6 3 0 7 1 4	1502	2 4 1 7 0 6 3 5

VI. ANALYSIS AND DISCUSSION

This section provides a comparative analysis and evaluation of each algorithm mentioned before. These evaluations are conducted based on the key parameters such as time complexity, space complexity, accuracy, and scalability. By examining these factors it will become easier to highlight the advantages and disadvantages of each algorithm. The provided analysis is based on the experiments and results mentioned before.

VI.I Evaluation

1. Time Complexity

- **BFS & DFS:**
  - In BFS, the complexity is $O(b^{d+1})$, while DFS's complexity is $O(b^{m})$.
  - Where 'b' is the branching factor, 'd' is the depth of the search, and 'm' is the maximum path length.
  - As shown for N = 8, BFS expanded 23 nodes, while DFS expanded 4743 nodes. This highlights that DFS is very inefficient compared to BFS [6].
- **HCS & BeFS:**
  - In HCS, the complexity is $O(\infty)$, even though the results shown above were promising as HCS expanded for only 9 nodes. However, the worst case is that it could get stuck at local maxima [6].
  - In BeFS, the complexity is $O(b^{m})$
- **BF & BT & Improved BT:**
  - In BF, the complexity is $O(N^{N})$, while in BT and Improved BT share the same complexity of $O(N!)$.
  - BT appears to be faster than BF, as shown before in the results, at N = 12, BT took 32.25 seconds versus the BF which took 58.73 seconds [8].
  - Improved BT is generally faster than regular BT as it "looks ahead" which makes it very accurate [10].

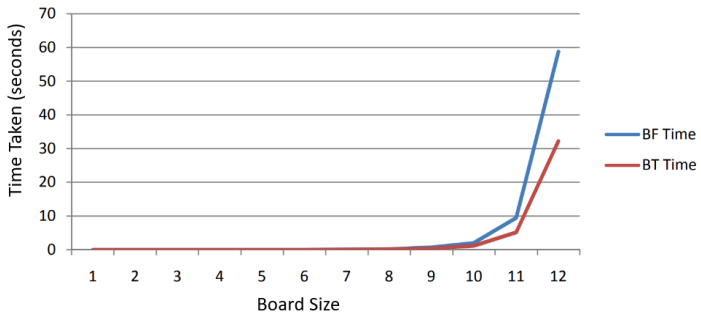

Figure 12. Comparison Between BF & BT - Time Complexity [8].

- **Genetic Algorithm:**
  - The complexity per candidate solution was $O(N)$, however the total complexity can be approximated as $O(G\ x\ P\ x\ n)$. Where 'G' is the number of generations and 'P' is the population size [2].
  - GA efficiently finds solutions for larger values of N, outperforming previous methods like BFS & DFS [2].

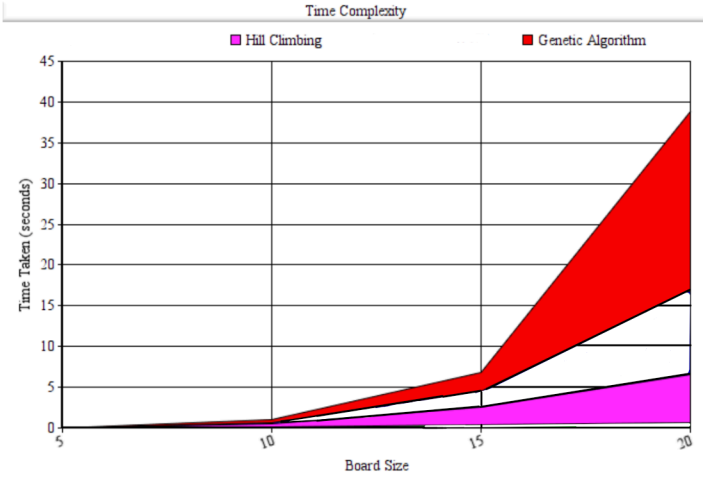

Figure 13. Comparison between HCS & GA - Time Complexity [4].

2. Space Complexity

- **BFS & DFS:**
  - In BFS, the complexity is $O(b^{d+1})$, while DFS's complexity is $O(b^{m})$.
  - BFS usually uses more memory compared to DFS [6].
- **HCS & BeFS:**
  - In HCS, the complexity is $O(b)$. While in BeFS the complexity is $O(b^{m})$.
  - HCS is very space-efficient, as it doesn't keep track of previously visited nodes, however this makes HCS more vulnerable to getting stuck.
  - BeFS requires additional memory for its queue, making its space usage higher than HCS [6].

- **BF & BT & Improved BT:**
  - In BF, the complexity is $O(N^N)$, as it stores every possible board configuration, making it very memory demanding [8].
  - In BT, the complexity $O(N)$, as solutions are built recursive and incrementally, which doesn't require as much memory as BF [8].
  - In Improved BT, the complexity remains $O(N)$, as the addition of the "looking ahead" feature does not significantly increase the memory usage [10].
- **Genetic Algorithm:**
  - In GA, the complexity could be approximated to $O(P\ x\ N)$, the memory usage here is significantly less than previous exhaustive methods.
  - It relies on a 1D array representation method which reduces memory usage by eliminating empty board positions [2].

3. Performance

- **BFS & DFS:**
  - BFS outperforms DFS in terms of performance as it finds a solution in fewer nodes, however it uses more memory.
  - DFS on the other hand takes longer time, but uses less memory.
  - Finally, both solutions are not very significantly different from each other, but they are both inefficient for larger values of N [6].
- **HCS & BeFS:**
  - HCS is very fast on problems with smaller values of N, and is very memory efficient but can get easily stuck in local maxima.
  - BeFS provides a more systematic search that doesn't get stuck, however it searches slightly more nodes than HCS and uses more memory.
  - BeFS outperforms BFS & DFS [6].
- **BF & BT & Improved BT:**
  - BF is extremely slow and uses large amounts of memory.
  - BT is significantly faster than BF, as for N = 12, it took 32 seconds instead of BF's 58 seconds.
  - Improved BT is much faster than BT as at N = 12, it only took 5.2 seconds.
  - BF is the worst algorithm so far, while BT offers better performance over BFS and DFS, BeFS still takes the lead and remains the better algorithm.
  - Lastly, the improved version of BT, outperformed BeFS [8], [10].
- **Genetic Algorithm**
  - GA is a very fast algorithm, offering a near optimal solution. While BeFS is faster for smaller values of N, GA is faster for larger values of N [2].

4. Scalability

- **BFS & DFS:**
  - Both algorithms are impractical for scaling, while DFS may offer better scaling over BFS due to its lower memory usage. They are both limited to N = 15 as the time required increases exponentially [6].
- **HCS & BeFS:**
  - HCS is highly scalable in terms of memory usage, but is also limited to a smaller value of N, because as the N increases the risk of getting stuck increases, making it a not so good algorithm for scaling.
  - BeFS isn't as scalable as HCs in terms of memory, it still remains not scalable in terms of time complexity as it's limited to around N = 16 [6].
- **BF & BT & Improved BT:**
  - BF is very bad for scalability, as both of it's time and memory complexity scale by an N^N factor. Making it limited to N = 8 or even less.
  - BT is significantly more scalable than BF, as it's time and memory complexity only scale by $N!$, however it is also limited to a value around N = 12.
  - The improved BT is further more scalable than regular BT, considering its reduced time complexity. However, it also remains limited to a value of N = 14 [8], [10].
- **Genetic Algorithm:**
  - GA is also very scalable, perhaps the most scalable algorithm so far, as its memory grows linearly and due to its low time complexity. GA can reach a range of N = 30 or higher. Making it very scalable and suitable for larger values of N [2], [3].

5. Ease of Implementation

- **BFS & DFS:**
  - BFS and DFS are simple to implement due to their uninformed search nature, requiring only a simple queue or stack data structure. However, DFS can be vulnerable to endlessly searching without depth constraints. Which may require more intermediate implementation skills [6].
- **HCS & BeFS:**
  - HCS and BeFS are of moderate effort, primarily in designing good heuristic functions to guide the search.
  - BeFS might be harder to implement than HCS as it uses a backtrack approach [6].
- **BF & BT & Improved BT:**
  - BF is very simple to apply.

- BT requires recursive backtracking which makes it significantly harder than BF.
- Improved BT contains heuristic improvements like the "look ahead" feature, which further increases the implementation complexity [8], [10].

- **Genetic Algorithm**
  - GA is the most complex algorithm compared to the previous algorithms, as it includes a fitness function design, crossover and mutation operators, and multiple parameters for tuning such as population size, mutation rates [2].

6. Strengths & Weaknesses

- **BFS:**
  - Strengths: Guarantees the shortest solution. Simple to implement and understand.
  - Weaknesses: High memory usage, poor scalability.
- **DFS:**
  - Strengths: Lower memory usage, Simple to implement and understand.
  - Weaknesses: Doesn't guarantee finding the shortest solution, poor scalability, and can get stuck exploring indefinitely.
- **HCS:**
  - Strengths: Very low memory usage, fast convergence.
  - Weaknesses: Can get stuck in local maxima, and has a worse case time complexity of infinity.
- **BeFS:**
  - Strengths: Uses heuristic evaluation to reduce the number of expanded nodes, very robust.
  - Weaknesses: High memory usage, and its solution outcome depends on the heuristic, if it's misleading it could output a bad solution.
- **BF:**
  - Strengths: Exhaustive search guarantees finding all possible solutions if time and memory are not taken into consideration.
  - Weaknesses: Very poor scalability, takes a very long time, only practical for very small values of N, and takes a lot of memory.
- **BT:**
  - Strengths: utilizes pruning to reduce the number of invalid configurations, which reduces the time and memory needed to find a solution.
  - Weaknesses: poor scalability, and only practical for small values of N.
- **Improved BT:**
  - Strengths: Uses a "look ahead" technique that improves speed, and it's more scalable than regular BT
  - Weaknesses: worst case complexity remains exponential, and practical for moderate values of N.
- **Genetic Algorithm**
  - Strengths: Very scalable, parallelizable, and very fast.
  - Weaknesses: does not guarantee finding an optimal solution, heavily dependent on parameter tuning making it very complex to implement and understand.

7. Real-World Applicability

- **BFS & DFS:**
  - not suitable for large scale or dynamic problems due to inefficiency. It's best used in small search spaces[6].
- **HCS & BeFS:**
  - Ideal for real-time applications for example games, and robotics. Where a near-optimal solution in a short time is preferable over exhaustive search [6].
- **BF & BT & Improved BT:**
  - Practical for small problem sizes ($N \leq 12$) with guaranteed solution. With the exception of the Improved BT as it improves the performance through heuristics for moderately larger problems [10].
- **Genetic Algorithm:**
  - Best suited for large or complex problems ($N > 25$) and using parallel search across candidate populations to find solutions for large boards quickly [2].

VI.II Discussion

- **BFS & DFS:**
  - **Key Findings & Trade-offs**

    Memory vs. Depth:

    1. BFS explores all the nodes level by level but is the disadvantage of using a lot of memory with space complexity $O(b^d)$, where b is the branching factor and d is the depth of the solution. This makes BFS inefficient in large or deep search spaces [6].
    2. DFS, however, has less memory requirements but could suffer from infinite recursion without bound on depth and hence potential stack overflow issues [6].
- **HCS & BeFS:**
  - **Key Observations & Trade-offs**

    Heuristic Quality:

    Both the performance of BeFS and HCS is heavily dependent on the quality and accuracy of the heuristic function. HCS can be trapped in local optimum or BeFS can be forced to follow worse routes if heuristics are poor. In contrast to the uninformed methods, these algorithms focus on goal-oriented search but can be penalized by partial searches if the heuristic is misleading [6].

- **Future Directions:**
  ML-Based Heuristics Future lines of research comprise the use of reinforcement learning (RL) for learning and revising heuristic functions dynamically, developing the effectiveness of search in dynamically changing environments [9].

- **BF & BT & Improved BT:**
  - **Key Observations & Trade-offs:**

  Completeness vs. Speed:
  These traditional methods are exhaustive (i.e., they will reach a solution if there is one) but time and memory intensive for big problems. Improved BT reduces search space overhead but remains beset by scalability problems for big N [8].

- **Future Directions:**
  Parallelized Backtracking:
  Distributed computing techniques would parallelize Backtracking, enhancing scalability and speeding up solution discovery for bigger problem sizes [10].

- **Genetic Algorithm**
  - **Key Observations & Trade-offs:**
    Completeness vs. Speed:

  Unlike traditional search algorithms, GAs prioritize speed and scalability over completeness. GAs employ stochastic (non-deterministic) search strategies, which may not always yield optimal solutions but can achieve good solutions in a short time [2].

  Future Directions:
  Hybrid Algorithms:
  Combining GA's global search with BT's local search (e.g., GA-seeded Backtracking) can be made to improve efficiency and solution quality [2, 10].

  Adaptive GA The use of self-tuning parameters, i.e., dynamically adjusted mutation rates and crossover probabilities, can minimize sensitivity to human parameter tuning and improve adaptability when operating in dynamic environments [9].

VII. CONCLUSION

The work demonstrates the compromises in algorithmic efficiency, scalability, and completeness in solving the N-Queens problem. Traditional methods like backtracking and brute force guarantee correctness for small boards ($N \leq 12$) but are impossible very rapidly with exponential time complexity. Better backtracking with its heuristics optimizations renders larger boards ($N \approx 15$) practical by discarding invalid states with efficiency. Heuristic methods like Hill-Climbing and Best-First Search offer better speed for medium sizes of values of N but might suffer from local optima and higher memory demands. On the other hand, Genetic Algorithms perform well in scaling to large ($N > 25$) with parallel search but occasionally fail to find the optimum solution. Deterministic methods are optimal for small-scale boards, with hybrid methods potentially being even better for medium values. For large problems, evolutionary methods excel in searching hard search spaces with effectiveness. Topics of research in the future include adaptive heuristics with the help of machine learning and hybrid methods like GA with backtracking to obtain better speed and accuracy

VIII. REFERENCES

[1] M. A. Khasawneh, *Hill-Climbing with Trees: A Novel Heuristic Approach for Discrete NP-Hard Combinatorial Optimization Problems*, Ph.D. dissertation, Binghamton Univ., NY, 2023.

[2] A. S. Farhan, W. Z. Tareq, and F. H. Awad, "Solving N Queen Problem using Genetic Algorithm," *Int. J. Comput. Appl.*, vol. 122, no. 12, pp. 11–12, Jul. 2015.

[3] V. Thada and S. Dhaka, "Performance Analysis of N-Queen Problem using Backtracking and Genetic Algorithm Techniques," *Int. J. Comput. Appl.*, vol. 102, no. 7, pp. 26–27, Sep. 2014.

[4] L. Andov, "Local Search Analysis N-Queens Problem," *Griffith University*, Mar. 2018.

[5] F. Soleimanian, B. Seyyedi, and G. Feyzipour, "A New Solution for N-Queens Problem using Blind Approaches: DFS and BFS Algorithms," *Int. J. Comput. Appl.*, vol. 53, no. 1, pp. 45–46, Sep. 2012.

[6] K. Mathew and M. Tabassum, "Experimental Comparison of Uninformed and Heuristic AI Algorithms for N Puzzle and 8 Queen Puzzle Solution," *Int. J. Digit. Inf. Wirel. Commun.*, vol. 4, no. 1, pp. 143–154, 2014.

[7] O. K. Majeed et al., "Performance comparison of genetic algorithms with traditional search techniques on the N-Queen Problem," in *Proc. 2023 Int. Conf. IT Ind. Technol.*, 2023, pp. 1–?.

[8] S. Mukherjee, S. Datta, P. B. Chanda, and P. Pathak, "Comparative Study of Different Algorithms to Solve N Queens Problem," *Int. J. Found. Computer Science Technol.*, vol. 5, no. 2, pp. 15–17, Mar. 2015.

[9] U. Sarkar and S. Nag, "An Adaptive Genetic Algorithm for Solving N-Queens Problem," arXiv preprint arXiv:1802.02006, 2017.

[10] A. Ahmed, A. Kamran, M. Ali, and A. W. Shaikh, "Heuristic Approaches for Solving N-Queens Problem," Journal of Applied and Emerging Sciences, vol. 2, no. 2, pp. 197–201, 2011.