Comparison of Algorithms for the N-Queen Problem

Abstract

This paper addresses the experiment which will take a number of algorithms to solve Constraint Satisfaction Problem to find which algorithm is doing the best performance in terms of memory perspective, and execution time perspective. The Constraint Satisfaction Problem for this experiment, the one of the simplest Constraint Satisfaction Problem, but well-known classic Constraint Satisfaction Problem called N-Queens[4] will be used. A solution for N-Queen problem asks to avoid sharing the same row, column or diagonal by multiple queens. The efficiencies of algorithms will be compared by time to solve N-Queens problem and their memory usage comparison also will be measured.

1 INTRODUCTION

N-Queens problem is the problem of placing n queens on a NxN chessboard such that no two queens attack each other.[3] The N-Queens problem was originated from 8-Queens problem which is published by Max Bezzel who was the Chess Composer in 1848. After the first solution published in 1850, N-Queens problem was generalized from 8-Queens problem, introduced in 1850 by Carl Gauss, and became one of the well-known Constraint Satisfaction Problem. While it has been one of the well-known problems in the artificial intelligence area, numerous solutions have been found, and various search strategies and algorithms have been found since the original problem was proposed by Carl Gauss.

Since each queen must not be in the same row, column or diagonal, if one simply tries to search through every possible combination successfully, the number of possibilities one needs to examine often grows exponentially as the number of N grows. Table 1 shows how the number of solutions for N-Queens problem grows as the number of N increases. First five N amounts will be too small to see its growth, but after sixth value, it will show an increase in the number of the solution as the search load grows exponentially. Figure 1 demonstrates how N-Queens problem can be solved by showing one of the solutions for 8-Queens problem.

N	the number of solutions
1	1
2	0
3	0
4	2
5	10
6	4
7	40
8	92
9	352
10	724
11	2680
12	14200

Table 1: The number of solutions for the N-Queens problem

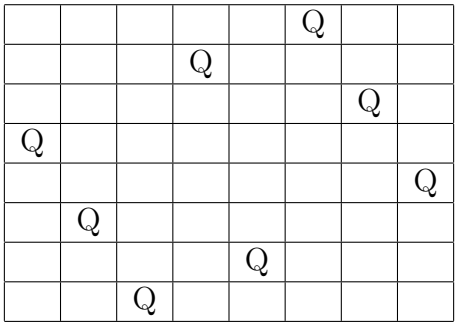

Figure 1: Solution to 8-Queens problem

Since simplicity of the N-Queens problem, it has widely been chosen as a test model to develop and benchmark new AI search problem-solving strategies in the AI area. This paper will compare various search algorithms in case of the N-Queens Problem by their times to solve and memory usages to see which approaches are efficient.

The remainder of the paper is organized as follows. Section 2 will review the related work done in the literature. Section 3 will present the concise description of the approach to compare each algorithms, including each algorithm details. Section 4 will present the description of the experiment design and results. Section 5 will show the analysis of the results. Finally, the paper ends with conclusions and summary.

2 THE LITERATURE REVIEW/RELATED WORK

Various researchers try to show how to overcome N-Queens problem. Various solutions to N-Queens problem have been published since the original N-Queen problem was proposed by Carl Gauss. This section will present numerous approaches to solve the N-Queens problem.

One of the approaches was search through every search space and collects all of the possible solutions. Numerous search algorithms have been developed to find every possible combination for N-Queens problem solution set. Backtracking search algorithm is the one of the well-known search algorithms, which is generating all possible combinations for the Constraint Satisfaction Problem. Usually, backtracking search sets the partial solutions to arrange n queens in the first n rows on board, all in different rows and columns. Any solution that has two mutually attacking queens on the board can be abandoned. To collect every possible solution for N-Queens problem, backtracking starts by putting one queen on the first column of the first row and keep placing next queens on the other rows and columns continually, while not violating the constraints hold by a previously placed queen. Once backtracking reaches the situation that the next queen cannot be placed due to the constraints by previously placed queens, it simply backtracks to the previously placed queen and tries to find another solution. However, when the search space gets really larger, it often suffers from the exponential growth of computing time.[8]

The authors of "Efficient local search with conflict minimization: A case study of the n-queens problem"[8] presented an alternative to the backtracking search by proposing a novel local search algorithm with conflict minimization to solve the n-queens problem. They implemented an efficient local search algorithm for the n-queens problem, and their results show that this efficient local search is capable of solving the N-Queens in linear time without backtracking. Also, they suggested that it is able to find the solution for really large size n-queens problems. It is capable of finding a solution for extremely large N-Queens problems without suffering from the exponential growth of computing time. [8]

Many other researchers proposed other efficient search algorithms to solve the N-Queens problem. These algorithms techniques include common traversal of graphs and digraphs algorithms such as Breadth First Search and Depth First Search[7], forward checking algorithm which is another general CSP algorithm[2], search heuristic methods such as genetic algorithms[1], and heuristic repairing method[5], and dynamic CSPs[6].

The authors of "A New Solution for N-Queens Problem using Blind Approaches: DFS and BFS Algorithms."[7] used depth-first-search and breadth-first-search to solve the N-Queens problem. The authors proposed a new blind algorithm for solving the N-Queens using a combination of DFS and BFS searches by the recursive approach. The proposed algorithm act based on placing queens on chess board directly. They ran multiple experiments to show the result that performance and run time with the combination of DFS and BFS were much better than backtracking methods and hill climbing methods. Also, they compared the performance and run time between DFS and BFS algorithm to show that the DFS algorithms are quicker than BFS algorithm and they also show that DFS uses less required memory by showing the number of extended node in DFS algorithm is less than BFS algorithm.[7]

In the other reference, "Constraint satisfaction problems: Algorithms and applications" [2], the authors start defining CSPs and describing the basic techniques for solving them by using three different algorithms, and compare the performance of these algorithms. Authors introduce the forward checking algorithm by comparing with the backtracking algorithm to show that it checks the constraints between the current and past variables and the future

variables. In this way, it leads the branch with failure to be pruned earlier than the backtracking algorithm. Even though it does more work than simple backtracking when each assignment is added to the current partial solution, in order to reduce the size of the search tree and thereby reduce the overall amount of work done.[2]

Additionally, in the reference "Solving Large-Scale Constraint Satisfaction and Scheduling Problems Using a Heuristic Repair Method,"[5] the authors describes a simple heuristic method for solving large-scale constraint satisfaction by using heuristic repair method. Given an initial assignment for the variables in a problem, the method operates by searching through the space of possible repairs. The search is guided by an ordering heuristic, the min-conflicts heuristic, that attempts to minimize the number of constraint violations after each step. They explain how the method works by searching the space and guided by an ordering heuristic which tries to minimize the constraint, and shows that the performance of this approach is better than backtracking techniques.[5]

Also, the authors of "Solving n-Queen problem using global parallel genetic algorithm"[1] show that the genetic algorithm can be used to solve N-Queens problem. They presented custom chromosome representation, fitness function to evaluate, and the results with several n values. They proved that genetic algorithms are able to solve combinatory problems with simple "yes" and "no" answers. Furthermore, tests showed that genetic algorithm is able to find different solutions for a given number of queens. Additionally, they used parallelization to show it significantly improve genetic algorithm's performance since genetic algorithm performs a large number of computations.[1]

Lastly, the authors of "A dynamic programming solution to the n-queens problem"[6] shows that even dynamic programming can solve the n-queens problem. They started their approach by describing the simple algorithm based on dynamic programming which can solve the n-queens problem in time $O(f(n)8^n)$. They implemented the algorithm consists of performing dynamic programming with the breadth first search. Since dynamic programming uses memoization, it will have space requirements, and they specified it depends on the maximum size of a queue at any point in the algorithm. At the end of the algorithm, the number of solutions will be residing in the queue. [6]

3 EXPERIMENT DESCRIPTION/ALGORITHMS

The goal of this paper is to compare the performance differences of each algorithm to see which approach is efficient in time and memory usage. To do this experiment, I am going to use the publicly available code called AIMA-lisp. AIMA-lisp already implemented the environment to make N-Queens problem, methods for this experiments such as algorithms, and the function to measure execution time and memory usage. Run time and memory usage will be tested for each algorithm and compared to see efficiency of each algorithm by the results of each experiment.

3.1 N-Queens in AIMA

Here is the AIMA's N-Queens problem code. I am going to make the test environment with this code in lisp environment by simply calling make-nqueens-problem with passing appropriate N values as the arguments. This will take N values and pass it to nqueens-initial-state function to create N-Queens problem's initial environment.

Listing 1: AIMA's N-Queens problem code

```
(defun make-nqueens-problem (&rest args &key (n 8) (explicit? nil))
  (apply #'create-nqueens-problem
         :initial-state (nqueens-initial-state n explicit?)
         args))$

(defun nqueens-initial-state (n &optional (explicit? nil) (complete? nil))
  (let ((s (make-CSP-state
            :unassigned (mapcar #'(lambda (var)
                                    (make-CSP-var :name var
                                                  :domain (iota n)))
                                (iota n))
            :assigned nil
            :constraint-fn (if explicit?
                               (let ((constraints (nqueens-constraints n)))
                                 #'(lambda (var1 val1 var2 val2)
                                     (CSP-explicit-check
                                      var1 val1 var2 val2 constraints)))
                               #'nqueens-constraint-fn))))
    (if complete? (CSP-random-completion s) s)))
```

Here are some information about some candidates algorithms.

3.2 Brute-Force Search

Here is the code for brute-force search in AIMA. It will expand nodes according to the specification of PROBLEM until it finds a solution or runs out of nodes to expand. The QUEUING-FN decides which nodes to look at first. This QUEUING-FN will be used to handle the order for differentiating DFS and BFS search algorithms.

Listing 2: AIMA's Brute-Force search code

```
(defun general-search (problem queuing-fn)
  (let ((nodes (make-initial-queue problem queuing-fn))
        node)
```

```
(loop (if (empty-queue? nodes) (RETURN nil))
      (setq node (remove-front nodes))
      (if (goal-test problem (node-state node)) (RETURN node))
      (funcall queuing-fn nodes (expand node problem)))))
```

3.3 Depth-First-Search and Breath-First-Search

Here is the code for Depth-First-Search which is using brute-force search's QUEUING-FN. It will search the deepest nodes in the search tree first since it takes enqueue-at-front as the argument for general-search.

Listing 3: AIMA's Depth-First-Search code

```
(defun depth-first-search (problem)
  (general-search problem #'enqueue-at-front))
```

Here is the code for Breadth-First-Search which is using brute-force search's QUEUING-FN. It will search the shallowest nodes in the search tree first since it takes enqueue-at-front as the argument for general-search.

Listing 4: AIMA's Breadth-First-Search code

```
(defun breadth-first-search (problem)
  (general-search problem #'enqueue-at-end))
```

3.4 Search Algorithms That Use Heuristic Information

I also picked some candidates from the heuristic search algorithms such as greedy search, tree A* search, and uniform-cost-search. Every heuristic algorithm uses best-first-search as a foundation code with eval-fn to differentiate each algorithm. Greedy search uses heuristic as distance to the goal, tree A* search uses estimated total cost as F = G + H (G: traveled distance, H: heuristic cost), and uniform-cost-search uses the node's depth as its cost. Here are codes for each algorithm.[9]

Listing 5: AIMA's Heuristic Algorithms

```
(defun best-first-search (problem eval-fn)
  (general-search problem #'(lambda (old-q nodes)
              (enqueue-by-priority old-q nodes eval-fn))))

(defun greedy-search (problem)
  (best-first-search problem #'node-h-cost))
```

```
(defun tree-a*-search (problem)
    (best-first-search problem #'node-f-cost))

(defun uniform-cost-search (problem)
  (best-first-search problem #'node-depth))
```

3.5 Backtracking Search and Forward Checking Algorithm

Lastly, it is definitely needed to pick Backtracking search and forward checking search as candidates since these two algorithms are most common algorithms to handle the constraint satisfaction problem including N-Queens problem. AIMA's backtracking search algorithm uses CSP-LEGAL-STATEP to check a consistency before the goal check, and avoid expanding inconsistent states. Forward checking search added a test to make sure the assignments so far have not eliminated all the possible values for one of the unassigned variables. Assumes that the problem definition uses CSP-forward-checking-successors, which removes conflicting values from the domains of the unassigned variables each time a variable is assigned.[9]

Listing 6: AIMA's Backtracking search and Forward Checking search

```
(defun csp-backtracking-search (problem &optional
                    (queuing-fn #'enqueue-at-front))
  (let ((nodes (make-initial-queue problem queuing-fn))
        node)
    (loop (if (empty-queue? nodes) (RETURN nil))
          (setq node (remove-front nodes))
          (when (CSP-legal-statep (node-state node))
            (if (goal-test problem node) (RETURN node))
            (funcall queuing-fn nodes (expand node problem))))))

(defun csp-forward-checking-search (problem &optional
                    (queuing-fn #'enqueue-at-front))
  (setf (csp-problem-forward-checking? problem) t)
  (let ((nodes (make-initial-queue problem queuing-fn))
        node)
    (loop (if (empty-queue? nodes) (RETURN nil))
          (setq node (remove-front nodes))
          (when (and (CSP-legal-statep (node-state node))
                     (not (CSP-empty-domainp (node-state node))))
            (if (goal-test problem node) (RETURN node))
            (funcall queuing-fn nodes (expand node problem))))))
```

4 EXPERIMENTS/RESULTS

For the first experiment, I will measure time to solve N-Queens problem for each search algorithm to decide which algorithm is fastest and efficient for N-Queens problem, and to see how each search algorithm makes the result differently. For the second experiment, I will measure memory usage for each search algorithm to see how each search algorithm takes the space to solve N-Queens problem and to decide which algorithm is space efficient.

For the search algorithm candidates, I decided to compare three different groups of search. A first group is a group of simple algorithms such as depth-first-search, breadth-first-search, and iterative deepening search. A second group is a group of the heuristic search algorithms such as tree-A*-search, greedy-search, and uniform-cost search. The last group is the algorithms which are notably used for the Constraint Satisfaction Problems such as backtracking search and forward checking search.

Before running the experiment, I needed to find reference point value for N to compare each algorithm within the same environment. To decide the reference point value for N, I will use brute-force search (Naive search), one of the candidate algorithms, to find the possible maximum number of N, which brute-force search can capable of finding a solution since brute-force search will take the longest time to find the solution. Simple brute-force algorithm would be a very poor for solving the N-Queen problem since placing a single queen in each row will have N^N combinations. In this way, I can get the number of queens to get results from every search algorithms to be compared. Once I get the possible maximum number of N, I will fix the maximum number N to create the environment for this experiment.

After performing brute-force search to find appropriate value N, I come up with a the values of N: 6. A Larger value of N than 6 would not work on the machine I tested since its search space was too large to take a long time and seems to stuck because we have general search algorithms as candidates.

With fixed N value as 6, I ran the first experiment to measure run time to solve 6-Queens problem. In the Table 2, I ran every candidate algorithms for 6-Queens on the same machine and measured the time to solve 6-Queens problem.

Algorithm	Run Time(seconds)
BFS	4.39682
DFS	0.05819
Iterative Deepening	0.226043
Greedy	0.434645
Tree A*	0.594049
Uniform Cost	0.600404
Backtracking	0.003507
Forward Checking	0.001294

Table 2: Time to solve N-Queens N: 6

With fixed N value as 6, I ran the second experiment to measure memory usage 6-Queens problem. In the Table 3, I ran every candidate algorithms for 6-Queens on the same machine and measured the memory usage of 6-Queens problem.

Algorithm	Memory Usage(bytes)
BFS	23983720
DFS	9130728
Iterative Deepening	9177704
Greedy	13124152
Tree A*	24702312
Uniform Cost	24702312
Backtracking	204416
Forward Checking	118048

Table 3: Memory Usage of N-Queens N: 6

5 ANALYSIS OF THE RESULTS

According to up results, we can see the run time and memory usages for each candidate algorithm. Comparison result for run time can be seen in the Figure 2. Comparison result for memory usage can be seen in the Figure 3.

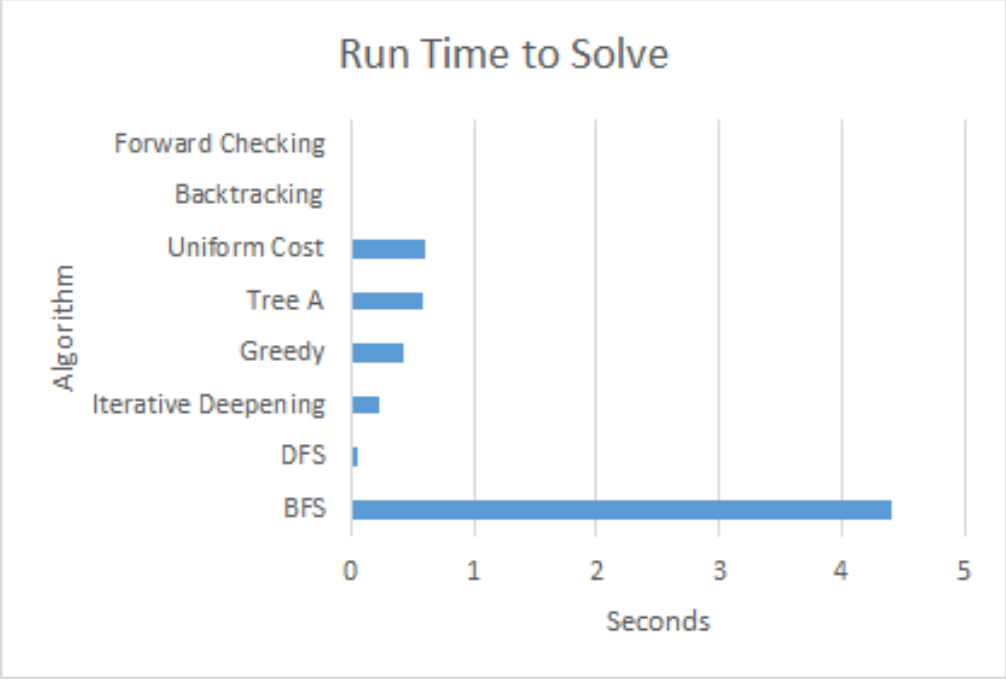

Figure 2: Time to solve 6-Queens

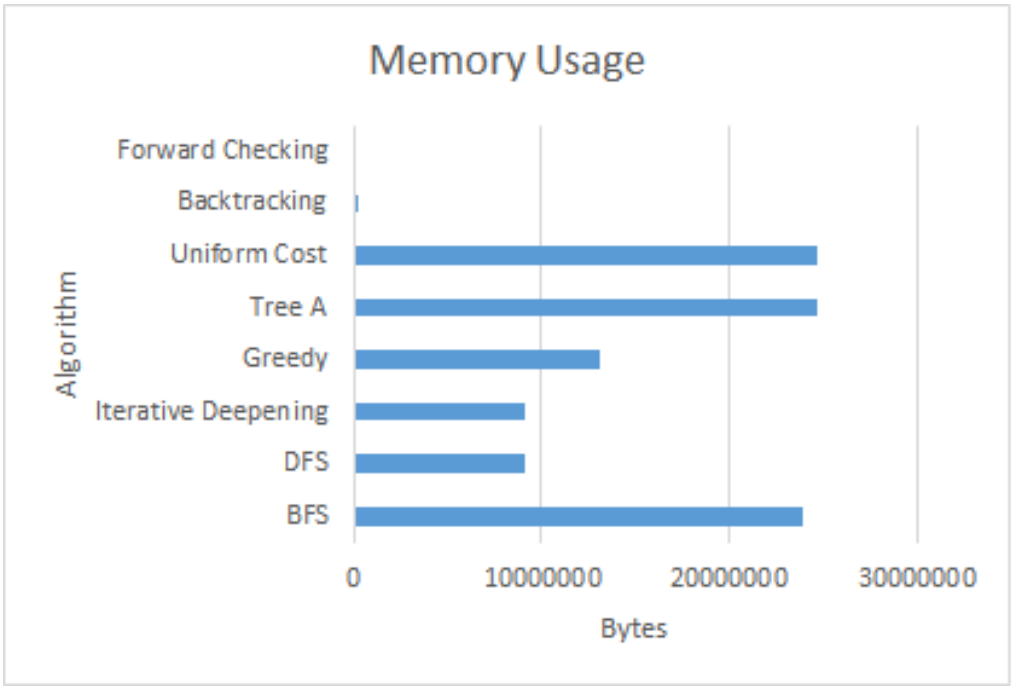

Figure 3: Memory Usage 6-Queens

For the first group, general search algorithms, we can see the run time to reach the solution in DFS algorithm was 0.05819 seconds, the time for BFS algorithm was 4.39682 seconds while the time for Iterative deepening DFS was 0.226043 seconds. Therefore, the DFS algorithm is much quicker than BFS algorithm, and Iterative deepening DFS is obviously slower than DFS, since it has to repeatedly visit top rows of tree nodes, unlike DFS. A number of extended node in DFS algorithm is less than BFS algorithm, therefore, the needed memory for BFS is more than DFS.

For the second group, heuristic search algorithms, we can see the run time to reach the solution in Greedy search was 0.434645 seconds, the time for Tree A* search was 0.594049 seconds while the time for Uniform cost search was 0.600404 seconds. Therefore, the greedy search is quicker than Tree A* search and Uniform. Because greedy search only expands the node that appears to be closest to the goal, in this case, it will only explore the first spot available until it reaches the goal while tree A* search avoids expanding paths that are already expensive with calculated heuristic values. A number of extended node in greedy search is obviously less than tree-A* search and uniform cost search, therefore, the needed memory for greedy is less than tree-A* and uniform cost search.

For the last group, search algorithms which are most widely used for CSP, we can see the run time to reach the solution in backtracking was 0.003507 seconds, while the time for forward checking search was 0.001294 seconds. Therefore, the forward checking search is quicker than backtracking search. It was expected since backtracking search tries to find every possible combination to the goal while forward checking search doesn't explore the node with constraint and prune it earlier. A number of extended node in forward checking

search is obviously less than backtracking search, therefore, the needed memory for forward checking is less than backtracking search.

6 CONCLUSION

This paper experimented the efficiency of various algorithms from different types of search that work by placing the queens on N-Queens problem with given value N as 6. Simple search algorithms and heuristic algorithms were not good enough to surpass traditional CSP methods like backtracking search and forward checking search. Traditional CSP methods were really better than other search methods, and it shows why these methods acknowledged becoming traditional CSP methods. For the future works, we can make heuristic functions more elaborately to see if heuristic searches improved and surpass these traditional CSP methods. Also, we can also add more efficient search algorithms to the candidate list to solving the N-Queens problem and compare to find a better solution.

References

[1] M. Božikovic, M. Golub, and L. Budin. Solving n-queen problem using global parallel genetic algorithm. In *International Conference on Computer as a tool EUROCON 2003*, 2003.

[2] S. C. Brailsford, C. N. Potts, and B. M. Smith. Constraint satisfaction problems: Algorithms and applications. *European Journal of Operational Research*, 119(3):557–581, 1999.

[3] M. G. Kaosar, M. Shorfuzzaman, and S. Ahmed. A novel approach to solving n-queens problem. In *Proceedings of the 8th World Multi conference on Systemic, Cybernetics and Informatics (SCI 2004), Orlando, Florida, USA*, pages 1–5, 2004.

[4] C. Letavec and J. Ruggiero. The n-queens problem. *INFORMS Transactions on Education*, 2(3):101–103, 2002.

[5] S. Minton, M. D. Johnston, A. B. Philips, and P. Laird. Solving large-scale constraint-satisfaction and scheduling problems using a heuristic repair method. In *AAAI*, volume 90, pages 17–24, 1990.

[6] I. Rivin and R. Zabih. A dynamic programming solution to the n-queens problem. *Information Processing Letters*, 41(5):253–256, 1992.

[7] F. Soleimanian, B. Seyyedi, and G. Feyzipour. A new solution for n-queens problem using blind approaches: Dfs and bfs algorithms. *International Journal of Computer Applications*, 2012.

[8] R. Sosič and J. Gu. Efficient local search with conflict minimization: A case study of the n-queens problem. *Knowledge and Data Engineering, IEEE Transactions on*, 6(5):661–668, 1994.

[9] P. N. Stuart Russell. Aima lisp code.