Different Algorithms for Solving N-QUEENS Problem

Mengxiao Wang

December 14, 2020

Abstract

This paper provides a description for N-queens problem and literal reviews related to this questions. And measure six common algorithms on how to solve it. Clarify in which situation, what algorithms perform better with data result. Also we will compare graph search and tree search, to see which one fits our problem. Through the research we will find the conclusion and future plan.

1 Introduction

My research topic is the to compare different algorithms performance on famous "N-queens puzzle". Its common version is 8-queens puzzle, which was published in 1848 by Chess composer Max Bezzel.Find all solutions for 8-queens problem can be computationally expensive, and there are 4426165368 possible different arrangements of eight queens on board. Find all solutions can be difficult. Also, for human being, it is hard to find all possible solutions when n is larger than 8. So, my research is to compare all the common algorithms and novel algorithms which can solve N-queens puzzle, and what's the pros and cons for them. And we will discuss when to use which algorithms specifically.

First of all, let me introduce what is n-queen puzzle. The n-queens puzzle is the problem of placing n queens on an n×n chessboard such that no two queens attack each other. So each queen can not in the same row and column, and they can not in the same diagonal. It can be complete-state problem, which means we can put n queens on the board, and find all the solutions. Or, we can place one queens in a place, and find others position. This problem is fascinating because it has many views to solve. Consider N-queens as CSP, Dynamic programming, or even local search problem. It is interesting because they are many algorithms to solve them, but not a lot of people know which one is better. And our research topic is to find the differences between common algorithms and novel algorithms.

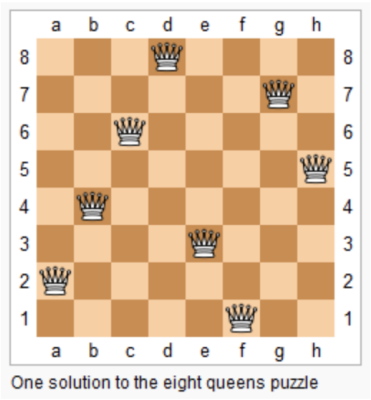

Figure 1: one of the 8-queens puzzle fundamental solutions

n	fundamental	all
1	1	1
2	0	0
3	0	0
4	1	2
5	2	10
6	1	4
7	6	40
8	12	92
9	46	352
10	92	724
11	341	2,680
12	1,787	14,200
13	9,233	73,712
14	45,752	365,596

Figure 2: number of fundamental and all solutions for the 8-queens puzzle

The eight queens puzzle is an example of the more general n queens problem of placing n non-attacking queens on an n×n chessboard. As we just says, 8-queens problem is computationally expensive, it has 4426165368 possible arrangements. The eight queens puzzle has 92 distinct solutions. If solutions that differ only by the symmetry operations of rotation and reflection of the board are counted as one, the puzzle has 12 fundamental solutions.. A fundamental solution means we can find other solutions simply by rotating these fundamental solutions. And below is one example of fundamental solutions for N-queens puzzle.

This kind of puzzle question is a fully observable, and we can use complete state formulation to solve the 8-queen on board, which has 2057 states, while when we use incremental model, it will have 1.8e14 states. Some common al-

gorithm to solve n-queens puzzle including nontraditional approaches such as constraint programming, logic programming or genetic algorithms, and also it is used as an example of a problem that can be solved with a recursive algorithm, by phrasing the n queens problem inductively in terms of adding a single queen to any solution to the problem of placing n1 queens on an n×n chessboard. By showing how many fundamental and all solutions for n-queens puzzle when n is less than 15, we can see how complicated the N-queen problem is. The figure is in Figure 2. As you can see, n=2 and 3 has no solutions. And when n=6, there are only 4 solutions. We can conclude that there is no connect between n and the number of solutions for N-queens problem.

2 The Literal review of N-queens problem

First of all, we want to see how others compare the different algorithms to solve N queens problem through the article "Comparative study of different algorithms to solve N queens problem"[6]. It mainly shows how 4 algorithm works, and how well these 4 algorithm performs in n-queens problem. The 4 algorithm is the Genetic Algorithm (GA), the Simulated Annealing (SA) Algorithm, the Backtracking (BT) Algorithm and the Brute Force (BF) Search Algorithm. Through its introduction, we know that N Queens problem is a classical intractable problem[6], brief overview the 4 algorithms and how to use these 4 algorithm to solve the N queens problem. Next, is the important part, the results of these four algorithm. When numbers of queens are less than 20, SA use little less time than GA, while numbers of queens are greater than 20, SA performs much better than GA on runtime area. And there fitness are quite same. For BF and BT, when numbers of queens are less than 8, BF and BT runtime has no difference, while numbers of queens are greater or equal than 8, BT use less runtime than BF. And they have nearly same fitness of this problem. Overall, the article concludes that SA performs better than GA, GA performs better than BT and BF. Lastly, the article shows some future wish if they can find more algorithm to solve Queens problem also[6].

Then we compare the different perspectives of this problems by "Different perspectives of the N-Queens problem"[2]. It first explained n-queens problem in six ways, algebraic, modular arithmetic, constraint satisfaction, maximum stable set, maximum clique and integer programming. And then explains by rotating the solution, we can get three more solutions. And they analyze three algorithms and do comparison among them. Just as we said, N-queens problem can be consider in many different views. And three algorithms to solve this problem is very common, the first one is brute force, which takes exponential time, then backtracking and permutation generation algorithms[2]. After find one solutions, it uses other 4 algorithms to find more. They are nondeterministic algorithms, backtracking algorithms, probabilistic algorithms and divide-and-conquer algorithms[2]. After introduce all these algorithm, they don't provide every results for each of them, so we don't know how well they perform, we only

know how they works,but not their performance.

Other than general view of N-queen problem, we also compare some novel algorithms with traditional backtracking or brute force algorithm. The first one is use series rules by "A new approach to solve n-queens problem based on series"[4]. First of all, it introduces the n-queens problem, and it can be applied in many areas, like parallel memory storage schemes, VLSI testing, traffic control and deadlock prevention[4]. Then it shows all the notation will be used for the algorithm. The algorithm is a combination of two rules and eight distinct series. Series are different from each others. There are two rules that are used with a particular series to a achieve solution for a given chess board[4]. The first rule is to find the solution lies in the series S_2 from the solution S_1. The second rule is to find the solution lies in the series S_6 from the solution S_5. Then use move function to show the pattern of placing queen on chess board. After nine steps, the proposed algorithm will find the solution of N-queen. Last but not least, this algorithm will only provides one unique solution when $n > 7$. Another novel one is use DNA computing to find solutions."A polynomial-time DNA computing solution for the N-Queens problem"[5] introduces DeoxyriboNucleic Acid (DNA)[5], which is a novel method to compute that uses DNA molecules for computation. And then explain other algorithms to find solutions for N-queens problem, including Brute-force algorithms, backtracking search, local search and conflict minimization techniques, neural networks, search heuristic methods, probabilistic local search algorithms and integer programming[5]. After that, it shows how DNA computing works. Just like DNA, it use 4 important operations, extract, copy, merge and append. Then it explains what is N-queens problem[5]. Furthermore, it uses the DNA algorithm to solve the N-queens problem, and shows the pseudo code for the procedure. Use the case when n=4, and find the solutions for that case. Finally, evaluation and conclude that the DNA computing use $O(N^2)$ time, which means it gives polynomial time solutions for the N-queens problem[5].

We also use two common algorithms for optimal problem and local search problem, which is dynamic programming and genetic algorithms to solve N-queen problem. "A dynamic programming solution to the n-queens problem"[7] introduces how to use dynamic programming in application. First of all, it shows the n-queens problem and change regular n-queens problem to toroidal n-queens problem. And they can solve all solutions for n-queens problem in polynomial multiplication time. Since this paper is kind old, so the algorithm is not sufficient as modern algorithms. They used dynamic programming to solve n-queens problems. They compare toroidal n-queens problem with regular one[7]. They used tuples and queue to store the value, and when terminated, the number of solutions is in queue. It is actually quite complicated, and not so fast, use more than polynomial time and space. So, they changed their algorithm, using line exhaustion and proceeding in row-major order[7], it do decrease the runtime, but not so much. Since there are no results come out, and no comparisons between algorithms, maybe put some results would help more. But dynamic pro-

gramming solution is a new insight of n-queens problem. Then, "The N-queens problem and genetic algorithms"[3] explains n-queens problem and genetic algorithms, and specifically explains how reproduction and crossover work[3]. Used a lot of works to explain genetic algorithms. Actually, it doesn't have to, since genetic algorithm is not that hard. And perform the results to us, using tables and figures, and we can see some of the GA can't find solutions. But genetic algorithms perform better than backtracking,[3] and when N is large, GA can still solve n-queens problem. And they also conclude that hybrid genetic algorithm can also solve constraint satisfaction problems[3]. There are many comparison between GA and other algorithms, so we don't know whether GA is the best for n-queens problems or not.

Finally, "A polynomial time algorithm for the N-Queens problem"[8] shows how to solve N-queens problem in polynomial time. It first introduces n-queens problem as classic AI search area. And then introduces the easiest n-queens problem, 4-queens problem[8], and backtracking is not able to solve large n-queens problem, so they used a new probabilistic local search algorithm based on a gradient-based heuristic[8]. It only uses polynomial time. It used random permutation and gradient-based heuristic[8], after each swap, the number of collisions decrease. Though the results, we can see, it really uses polynomial time to find a solution. The withdraw for this algorithm is that it can only find one solution as fast as possible, so when we want to know all the solutions, it may perform not so well.

At the end, I see some negative view about N-queens problem[1].While this paper is less than one pages, it first introduces n-queens problem, and shows that find one solution is trivial, because in the future we can save a lot of CPU-hours[1]. And use a mathematical way to solve n-queens problem.It doesn't have conclusion and results, and not a lot of explanations for how he solved the n-queens problem. Author thinks there is no need to find the better algorithm, because when time goes by, hardware will always be better than now.

3 Approach Description

I have solve N-queens using backtracking search, and in this research we use aimacode that textbook provided. We want to compare how different algorithms perform in runtime area from n=5 to n=10. And also we want to mainly focus on how bfs and dfs search performs, tree search or graph search is better for this problem. We will see the memory usage between these two, and use these algorithms code to explain why.

3.1 NQueensProblem setup

1.First initialize a tuple with n.
2.Define actions to put the queens on the chessboard.
3.Define conflicted rules. Check if all columns filled and no conflicts.
4.check any queens in row and col and ensure there are no conflicts.
5. Define a function return number of conflicting queens for a given node.
6.Check the result whether it follows the rule or not.
Input: Since we use python code, our input is a setup for N-queens problem, and which algorithms to use to solve, we also need to import package that we need.
Output: Give us the result of N-queens problem, and use time command to see runtime of the algorithms.

3.2 Algorithms of BFS tree search

```
def breadth_first_tree_search(problem):
    frontier = deque([Node(problem.initial)])# FIFO queue
    while frontier:#use like a queue
        node = frontier.popleft()
        if problem.goal_test(node.state):#test the goal
            return node
        frontier.extend(node.expand(problem))#List nodes
    return None
```

The BFS tree search will search the shallowest nodes in the search tree first.And search through the successors of a problem to find a goal. Also, the argument frontier should be an empty queue. Algorithms just repeats infinitely in case of loops.

3.3 Algorithms of DFS tree search

```
def depth_first_tree_search(problem):
    frontier = [Node(problem.initial)]#Initialize the problem
    while frontier:#use like a queue
        node = frontier.pop()
        if problem.goal_test(node.state):
            return node
        frontier.extend(node.expand(problem))#List nodes
    return None
```

DFS will not like BFS, instead it will search the deepest nodes in the search tree first. And search through the successors of a problem to find a goal. The argument frontier should be an empty queue. And repeats infinitely in case of loops.

3.4 Algorithms of BFS graph search

```
def breadth_first_graph_search(problem):
    node = Node(problem.initial) #Initialize the problem
    if problem.goal_test(node.state):
        return node
    frontier = deque([node])
    explored = set()
    while frontier:
        node = frontier.popleft()
        explored.add(node.state)#explored list
        for child in node.expand(problem):
        #check if in the list or not
            if child.state not in explored and child
            not in frontier:
                if problem.goal_test(child.state):
                    return child
                frontier.append(child)
    return None
```

BFS graph search we use priority queue, and keep track of what we have already explored to save memory usage and time. One we find the goal, return the child.

3.5 Algorithms of DFS graph search

```
def depth_first_graph_search(problem):
    frontier = [(Node(problem.initial))]#Initialize the problem
    explored = set()
    while frontier:
        node = frontier.pop()
        if problem.goal_test(node.state):#test the goal
            return node
        explored.add(node.state)#explored list
        frontier.extend(child for child in node.expand(problem)
            if child.state not in explored and child not
            in frontier)
    return None
```

Algorithms will search the deepest nodes in the search tree first.Then search through the successors of a problem to find a goal. Since we have if statement, it will not get trapped by loops.And if two paths reach a state, only use the first one.

3.6 Algorithms of iterative_deepening_search

```
def iterative_deepening_search(problem):
    for depth in range(sys.maxsize):
        result = depth_limited_search(problem, depth)
        if result != 'cutoff':
            return result
```

Use dls as helper function, and dls is just like dfs, but dfs will loop infinately, by not dls. And ids is just an improvement of dls.

3.7 Algorithms of A* search

```
def astar_search(problem, h=None, display=False):
    h = memoize(h or problem.h, 'h')
    return best_first_graph_search(problem, lambda n:
    n.path_cost + h(n), display)
```

A* search is best-first graph search with f(n) = g(n)+h(n). We need to specify the h function when you call astar_search. Best-first search will search the nodes with the lowest f scores first. And you need to specify the function f(node) that you want to minimize; for example, if f is a heuristic estimate to the goal, then we have greedy best first search; if f is node.depth then we have breadth-first search. And after doing a best first search you can examine the f values of the path returned.

4 Experiments and Results

In this research we use aimacode that textbook provided. We want to compare how different algorithms perform in runtime area from n=5 to n=10. And also we want to mainly focus on how bfs and dfs search performs, whether tree search or graph search is better for this problem.

We have six algorithms to compare, BFS tree search, DFS tree search, BFS graph search, DFS graph search, iterative_deepening_search and A* search. We will use python code and a laptop with 32GB memory MACBOOK pro to make our experiments.

We choose n=5 to n=10 because it won't run too long and too short time period, and we can compare the differences between each algorithm. We will take 10 results for each algorithm and use the average in case there are errors occur. Also if some value are extremely large or low, we will dismiss that value. We will mainly focus on BFS and DFS, since it is our common algorithms, and want to see tree or graph search is fitted for N-queens problem. We will focus on runtime on each algorithms, and not memory usage, because we just want to find the faster algorithms and memory usage can be different via different hardware, and since in the literal review, the hardware is improving, so memory usage is not a significant problem nowadays.

Table 1: Six algorithms runtime from n=5 to 10(in seconds)

Algorithms	BFST	DFST	BFSG	DFSG	IDS	A*
n=5	0.63	0.63	0.65	0.63	0.64	0.62
n=6	0.65	0.64	0.66	0.63	0.66	0.63
n=7	0.67	0.65	0.74	0.62	0.71	0.64
n=8	0.84	0.65	1.12	0.63	0.95	0.65
n=9	1.10	0.66	3.19	0.64	1.23	0.70
n=10	1.63	0.66	36.69	0.66	2.45	0.73

BFST is breadth_first_tree_search, DFST is depth_first_tree_search, BFSG is breadth_first_graph_search, DFSG is depth_first_graph_search, IDS is iterative_deepening_search, A* is A* search.

We want to find the fastest algorithm between these six algorithms. So we choose top two fastest algorithms and see what they performs when n is larger than 15.

Table 2: A* and DFST runtime from n=15 to 20(in seconds)

Algorithms	A*	DFST
n=15	1.40	0.94
n=16	3.12	1.35
n=17	2.18	1.19
n=18	6.40	2.61
n=19	5.14	1.07
n=20	11.18	10.59

5 Analysis

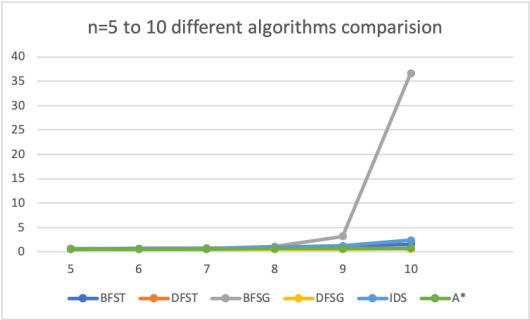

Figure 3: one of the 8-queens puzzle fundamental solutions

Through the data and graph, when can see when n is from 5 to 7, there

are not many differences between these six algorithms. But when n is larger or equal than 8, DFST, DFSG perform good, and also A*, then BFST and IDS are not so bad, but BFSG is significantly bad when n is large. And in order to see which algorithms is better, A* or DFS, we use another data set and graph.

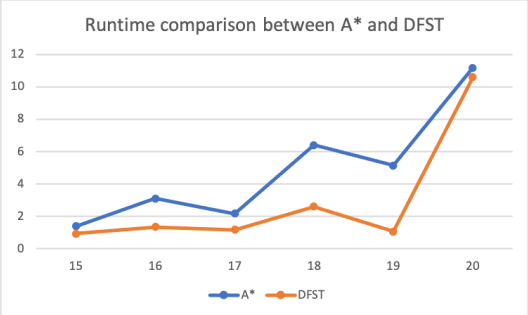

Figure 4: one of the 8-queens puzzle fundamental solutions

Because both DFS and A* perform well when n is small, so we want to see what happened we n is larger. When n is from 15 to 19, DFS performs better than A*, but when n=20, they perform quite same, and not very well.

And BFS, and IDS is not a good choice to solve N-queens problem. In code ways, it is true. Because BFS need to pass through all the nodes to find the solutions, and it we want to find all solutions for N-queens, BFS will perform good. Since we only want to find one solutions, and DFS will get an answers and return out of the loop, so it will be faster. Also, IDS is a combine of BFS and DFS, so same logic as BFS, it will perform good when we want to find all solutions, but in one solution situation, it will not perform so good.

As for graph or tree search, use the data and graph, we can see tree search is better than graph search in some way. Because graph search will need to memory the explored set, and it will cost more memory, but in N-queens problem, it is not necessary to keep track of the explored set, since we just want to find one solutions, not all of them.

6 Conclusion

In this paper, all six know algorithms has compared. BFS tree search, DFS tree search, BFS graph search, DFS graph search, iterative_deepening_search and A* search. In these six algorithms, DFS performs well, and secondly the A* algorithm. But when n is larger, both of them will not perform that well. And tree search is better to use for N-queens problem instead of graph search, because we only want to find one solution. For future work, we want to find some new novel algorithms to solve N-queens problem, and maybe some algorithms

can perform better than DFS when n is larger.

References

[1] Bo Bernhardsson. Explicit solutions to the n-queens problem for all n. *ACM SIGART Bulletin*, 2(2):7, 1991.

[2] Cengiz Erbas, Seyed Sarkeshik, and Murat M Tanik. Different perspectives of the n-queens problem. In *Proceedings of the 1992 ACM annual conference on Communications*, pages 99–108, 1992.

[3] Abdollah Homaifar, Joseph Turner, and Samia Ali. The n-queens problem and genetic algorithms. In *Proceedings IEEE Southeastcon'92*, pages 262–267. IEEE, 1992.

[4] Vishal Kesri and Manoj Kumar Mishra. A new approach to solve n-queens problem based on series. *International Journal of Engineering, Research and Applications*, 3(3):1349–1349, 2013.

[5] Ramin Maazallahi, Aliakbar Niknafs, and Paria Arabkhedri. A polynomial-time dna computing solution for the n-queens problem. *Procedia-Social and Behavioral Sciences*, 83:622–628, 2013.

[6] Soham Mukherjee, Santanu Datta, Pramit Brata Chanda, and Pratik Pathak. Comparative study of different algorithms to solve n queens problem. *Int. J. Found. Comput. Sci. Technol*, 5(2):15–27, 2015.

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

[8] Rok Sosic and Jun Gu. A polynomial time algorithm for the n-queens problem. *ACM SIGART Bulletin*, 1(3):7–11, 1990.