The Implementation of Backtracking Search on N-Queens Problem - Artificial Intelligence Final Project

Jen-Yin Chao

BU MET Collage Computer Science Department,

CS664 Fall 2017 – Alexander Belyaev

Table of Contents

Abstract

N-queen problem is one of the typical algorithm problems. By using depth first algorithm to solve it, it takes 15 minutes to find a solution for 40-queens, which is the limitation to this method, while using backtracking search can speed it up to 74 secs and even breaks the limitation to 50-queens, though it takes more memory space to store other candidate nodes during processing.

Introduction

In one of the previous assignment of n-queen problem, by using depth first algorithm with choosing the least constraining branch, I could only solve it up to 40 dimensions with 879 seconds running time, because I had to back to the very beginning to spread a brand new tree when I end up with an unsuccessful solution. And then, I learned the backtracking search in the following lecture, so I want to implement the method in this problem with tree data structure, which I've never tried before, to hopefully improve the performance.

N-queen Problem

The n-queen problem is the problem of placing N queens on an N*N chessboard, and none of the queens can attack the others, which means they can't be in the same row, column, slash line, and backslash line, shown as figure 1.

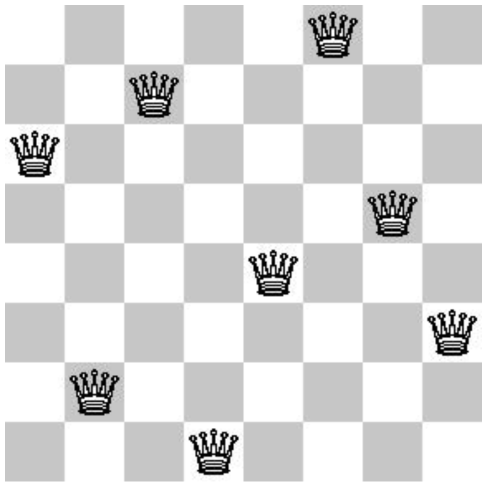


Figure 1. One solution of 8-queen problem

Backtracking Search

Backtracking search is used for a depth-first search that chooses values for one variable at a time and backtracks when a variable has no legal values left to assign, shown as figure 2. It is theoretically more efficient than pure depth-first search, though it's more complicated to implement.

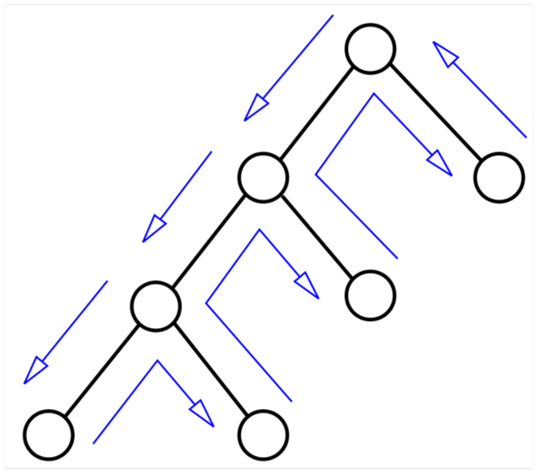


Figure 2. Backtracking Search

Objective

The objectives are the following:

1. Implement the backtracking search on n-queens problem.
2. Compare the efficiency between backtracking search and the pure depth-first algorithm.

Approach

To approach the problem, there are several steps as the follows:

1. Create an n*n array with all 0 to mark all blocks as available, named intersection table.
2. Randomly choose a block in first column to put the first queen on it, create a tree beginning with it, and update those intersectional blocks on the table as unavailable, marked as 1.
3. Get the available blocks in column 2, depending on the table, add them as children nodes in the tree, and calculate the number of available block left after choosing each candidate block.
4. Choose and put a queen on the block in column 2 with the most available blocks left after choosing it and update the block table.
5. In the following columns, it does the step 3 and 4 iteratively until the tree spreads to the depth of n and then ends the code running.
6. Anytime, if it ends with a deadlock, which means no block is available in the next column, it will pop the current node out, backtrack to the second-best candidate block from the data stored in the tree, and keep searching.

7. If it tries all the nodes in the tree and still cannot find a solution, it will drop the whole tree, go back to step 1, and choose another block in the first column without previous choice.

Coding

Python is the software used in this project, and the structure is shown as figure 3.

```
import numpy for easier data manipulation
import random for random function

class Node() for tree construction
def intersection_table_update()
def child_adding() which will automatically find hierarchical child and return the answer set
        intersection_table_update()
        find all available block in the next column
        calculate the number of available blocks for each candidate block
        for each candidate block depending on the decreasing order from above
                child_adding()
def Node_printing() for outcome printing


Main
parameter setting
loop while answer is not found {
        initiate intersection table
        choose the first node and construct the tree
        child_adding()
}
Node_printing()
```

Figure 3. Pseudo Code

Result

The result, shown as table 1 and figure 4, the average processing time of backtracking search algorithms are all lower than depth-first's, and it can even deal with the problem in 50 dimensions within 10 minutes, which is almost 40 times faster in high dimension, though it still can't reach 100 dimensions.

Table 1. Processing Time of Depth-first and Backtracking Search

N (dimension)		**5**	**10**	**15**	**20**	**25**	**30**	**35**	**40**	**45**	**50**
Depth-first		0.27s	0.30s	0.33s	7.37s	2.63s	8.67s	121.4s	879.1s	*	
Backtrack	**1**	0.3s	0.3s	0.3s	0.5s	0.7s	1.1s	25.6s	11.7s	335.8s	62.2s
	2	0.3s	0.3s	0.3s	0.4s	0.3s	2.8s	6.9s	22.7s	18.2s	378.8s
	3	0.3s	0.3s	0.3s	0.3s	0.4s	3.1s	2.7s	217.1s	496.8	319.2s
	4	0.3s	0.3s	0.3s	0.3s	1.5s	0.9s	0.9s	101.7s	99.9s	*
	5	0.3s	0.3s	0.3s	0.3s	0.8s	0.5s	3.8s	19.3s	42.8s	*
	Ave	0.3s	0.3s	0.3s	0.36s	0.74s	1.68s	7.89s	74.3s	198.7s	512.0s

* more than 15 minutes

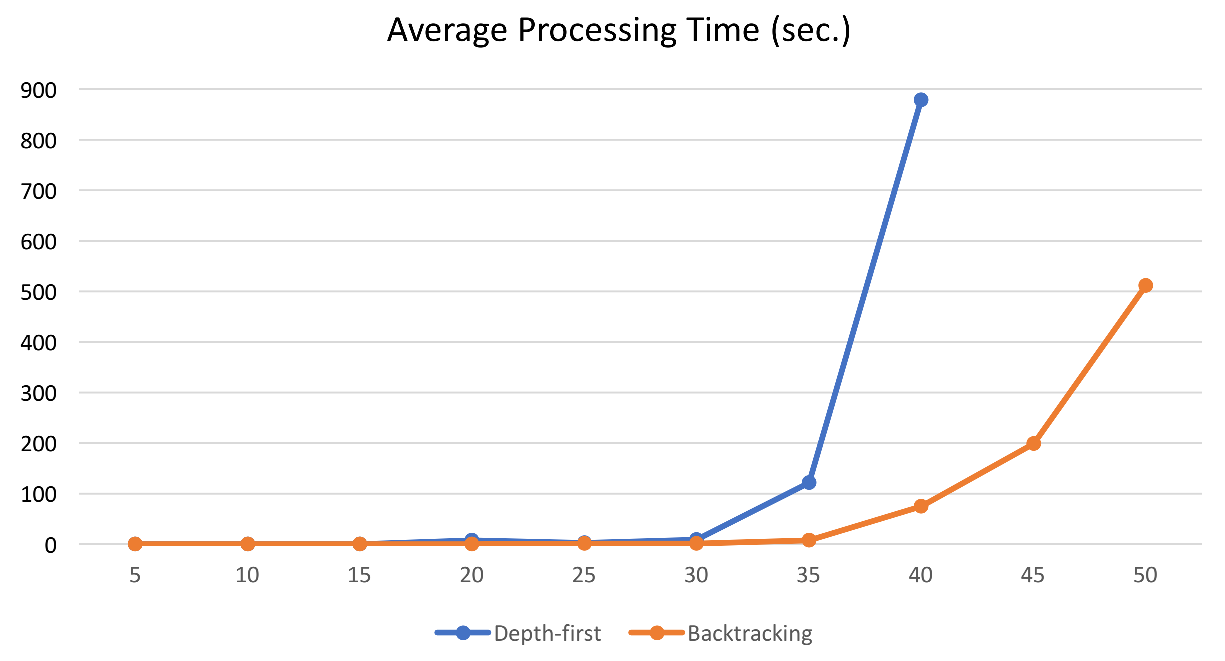


Figure 4. Average Processing Time Chart

By changing the scale of processing time into log(T) as figure 5, it seems that the curve of backtracking might be linear, which means this algorithm runs as P problem. As the result, I do it in regression analysis and the result of depth-first is shown as table 1 and figure 6, backtracking as table 2 and figure 7.

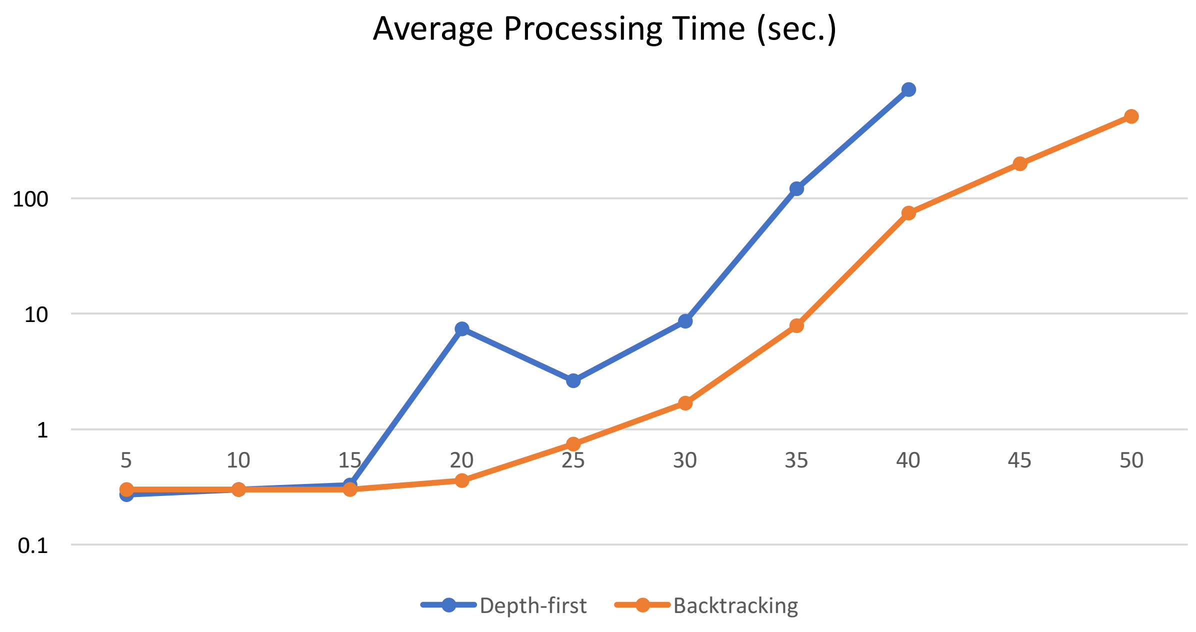


Figure 5. Average Processing Time Chart in log scale

Table 1. Regression coefficient table of depth-first

Source	Value	Standard error	t	Pr > |t|	Lower bound (95%)	Upper bound (95%)
Intercept	-1.510	0.376	-4.018	**0.007**	-2.429	-0.590
N	0.099	0.015	6.629	**0.001**	0.062	0.135

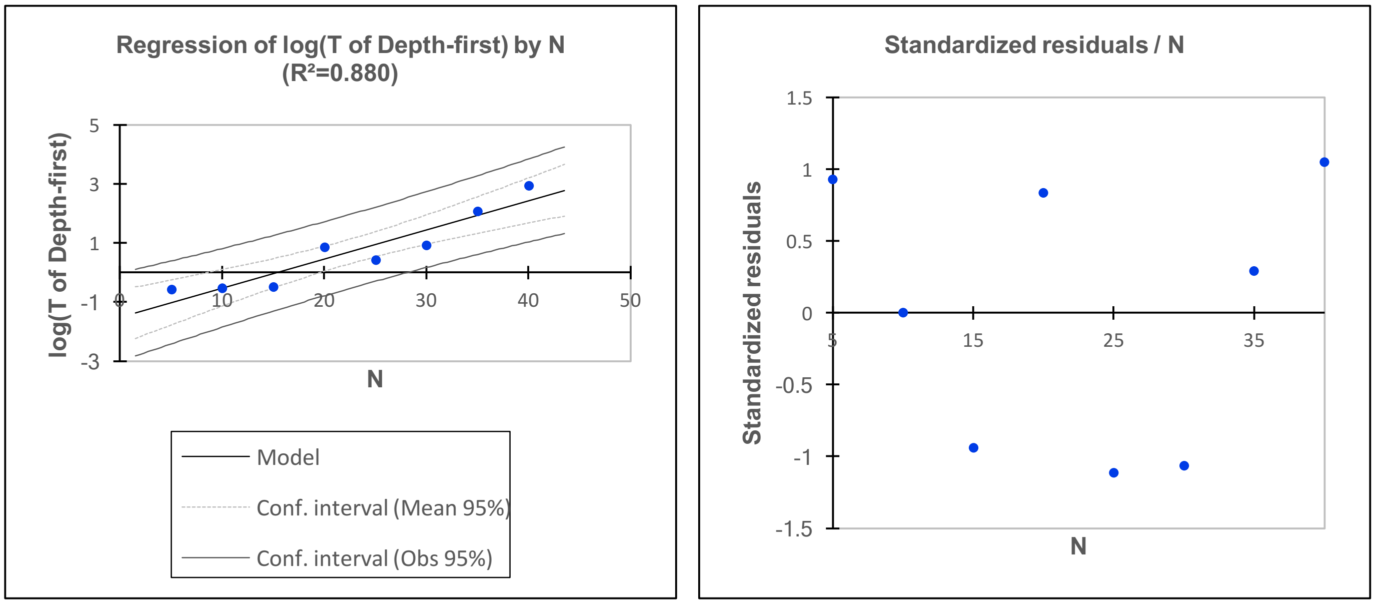
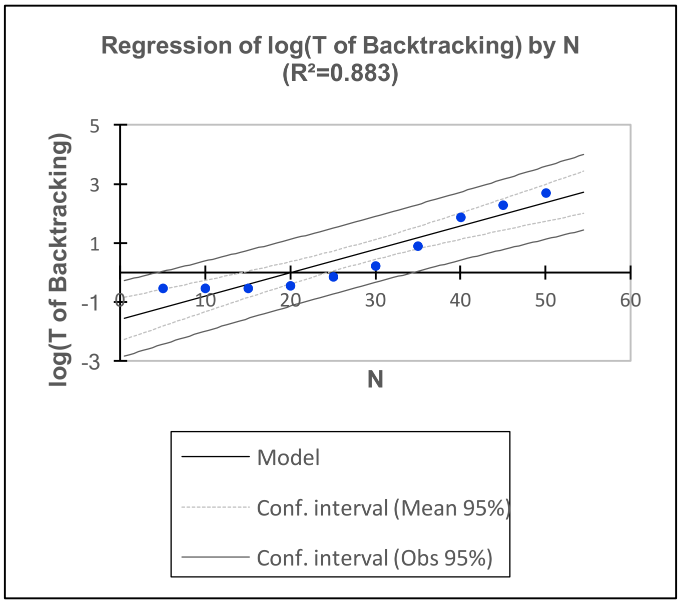


Figure 6. Regression Analysis of depth-first

Table 2. Regression coefficient table of backtracking

Source	Value	Standard error	t	Pr > |t|	Lower bound (95%)	Upper bound (95%)
Intercept	-1.587	0.315	-5.032	**0.001**	-2.314	-0.860
N	0.079	0.010	7.773	**< 0.0001**	0.056	0.102



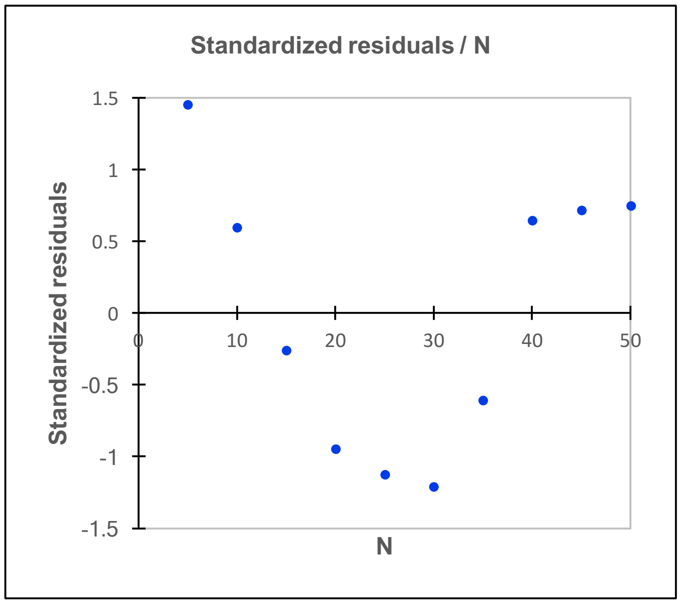


Figure 7. Regression Analysis of backtracking

Depending on the result of regression analysis, although the log plot of performance looks like linear lines, the residual plot of both algorithms does not distribute randomly, shaping in a specific trend, so that they are obviously not a linear function, representing they are still dealing it as NP problem.

Conclusion

Backtracking search is indeed a better algorithm than depth-first, especially in this kind of constraint satisfaction problems. Furthermore, tree data structure is quite useful to store unlimited processing data, which is unreplaceable in this project.

References

[1] Stuart Russell, Peter Norvig, *Artificial Intelligence*, 3rd Ed., 2013