

# CMSC 441 Fall 2005 Project

## Put Them in Their Place: Sudoku Solver Analysis

Natalie Podrazik  
12/12/2005  
[natalie2@umbc.edu](mailto:natalie2@umbc.edu)

## Overview

My solution to the Sudoku puzzle involves the use of one C program called within a Perl script for multiple executions. The C program iteratively rules out all of the possibilities of each cell, given an input file from which to start. The program exits normally when a solution to the given input is found; otherwise, it exits with one error code to determine that a solution may exist but it could not be determined from that input, and exits with another code if it is impossible to generate a solution from the given input file.

## I. C Implementation

### A. Algorithm

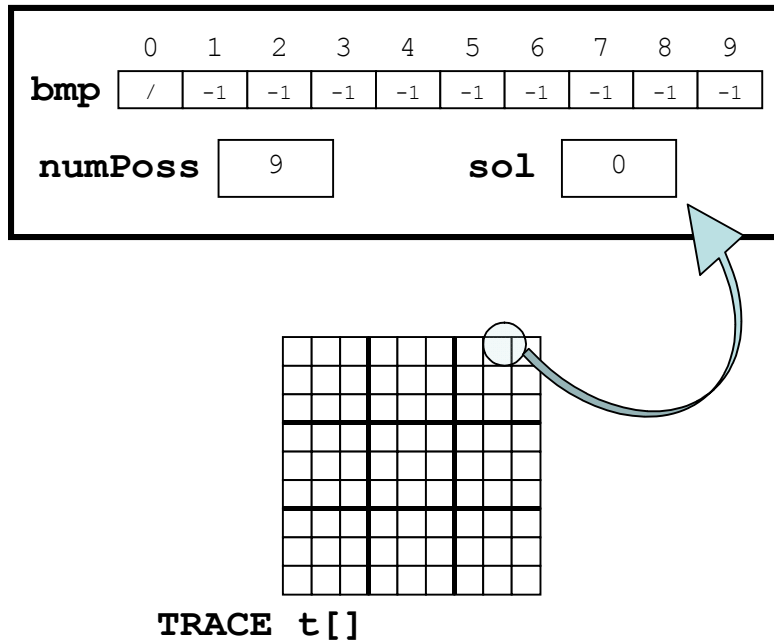
First, the program reads in the puzzle from an input file, and stores it into an integer array `a`, as shown in Figure 1.

1	0	0	8	0	0	0	6	7
0	6	0	0	4	0	0	3	0
0	0	0	3	0	0	5	0	0
0	0	0	5	0	2	1	0	9
5	0	0	0	7	0	0	0	2
9	0	6	4	0	1	0	0	0
0	0	1	0	0	4	0	0	0
0	5	0	0	9	0	0	7	0
7	3	0	0	0	5	0	0	6

`int a[]`

**Figure 1:** The conceptual array `a` after reading in the contents of the `n=3` file for medium difficulty.

It then initializes the values of an array `t` of structs of type `TRACE`, defined in detail in Figure 2. Each `t[i][j]` begins with a `bmp` (short for bitmap) array of all UNKNOWN values (`#defined` to be `-1`), `numPoss` to be the  $n^2$ , and a `sol` of 0. The `bmp` array is later indexed to record the possibilities of pip values for a particular cell; as possibilities are ruled out or affirmed, the values change from UNKNOWN (`-1`) to NO (0) or YES (1). The `numPoss` variable for each cell speeds up the search by keeping track of how many digits could be placed in a particular cell. Starting at the value of  $n^2$ , it is decremented at each change in the `bmp` array from UNKNOWN to NO. Cell `t[0][7]` of the `TRACE` array is shown in Figure 2.



**Figure 2:** The TRACE structure (top) that contains an integer array called  **bmp** , as well as an integer  **numPoss**  for a cell's remaining possibilities, and the cell's  **sol**  (solution). The bottom shows a conceptual 2-D array  $t$  of type TRACE, where  $n=3$ .

Each cell in the integer array  $a$  that contains a non-zero digit must contain a solution in the corresponding cell of  $t$ . Say  $a[i][j]$  contains the integer  $q$ . This would result in a call to the function  **Found()**  at row  $i$ , column  $j$ , pip value  $v$ .

The  **Found()**  function performs a few bookkeeping tasks for the TRACE array. It sets  $t[i][j].\text{bmp}[q] = \text{YES}$  (1), and all other values in the  **bmp**  array at  $t[i][j] = \text{NO}$  (0). The  **sol**  for that cell becomes  $q$ , and the  **numPoss**  = 1. It decrements the global value for  **REMAINING** , as one less cell remains to be solved. It also calls three functions that ensure the exclusiveness of that  $q$  for the row, cell and neighborhood:  **RuleOutRow()** ,  **RuleOutCol()** , and  **RuleOutNeighborhood()** .

**RuleOutRow()**  sets the  **bmp[ $q$ ]**  to be  **NO**  for every  $t[i][k]$  (if it was previously  **UNKNOWN** ), where  $0 \leq k < n^2$  and  $k \neq j$ , decrementing the  **numPoss**  values.  **RuleOutCol()**  is similar, yet it works with every  $t[m][j]$ , where  $0 \leq m < n^2$  and  $m \neq i$ .  **RuleOutNeighborhood()**  works with the cells in the array of  $n$  by  $n$  size. Given a found cell at  $t[i][j]$ ,  **RuleOutNeighborhood()**  first calculates where to start the ruling out of  **bmp**  values at  **startRow** , where  **startRow**  =  $\text{row} / n$ , then  **startRow**   $\times n$ . C's integer division truncates the decimal point, so that helped to readily find the start of that 'neighborhood' square. For instance, if the found cell was at  $t[4][0]$  with value 5 as the array  $a$  shows in Figure 1, where  **row**  = 4,  **col**  = 0, and  **pip**  = 5, the  **startRow**  value would first be  $(\text{row} / n) = 4 / 3 = 1$  by integer division. It then multiplies  **startRow**  by  $n$  (3) to have the starting row for neighborhood traversal at  $t[3][0]$ , where the  **startCol**  is calculated the same way to have the value 0. Once the function has the proper starting place, it can easily traverse through a neighborhood in nested for loops of  $n$  length each to assign  **UNKNOWN**   **bmp[ $q$ ]**  to be  **NO** , just like the  **RuleOutRow()**  and  **RuleOutCol()**  functions.

A caveat must be noted regarding the  **RuleOut\*()**  functions. If, at any time, the number of possibilities ( **numPoss** ) in a given cell is decremented to be equal to

0, then the puzzle for that input file is impossible to solve. For example, the test file I used for this condition was when two 7 values were placed in the same row as the initial input. When Found()ing one of the 7's, it changes the bmp[7] to be NO for the other 7 location in t. At the Found() of the next 7, it changes that bmp for all values but 7 to be YES, continually decrementing the numPoss to eventually be 0, thus catching a devious input file. Should this condition arise, the program frees allocated memory and exits with a certain code that the outer Perl script recognizes and handles adequately.

The TRACE array is initialized using this iterative Found()ing of nonzero digits found in a. Once the t array and all of its contents are properly, the ruling out of possibilities begins. The program keeps a global count of the remaining possibilities, REMAINING, as a condition to stop looping because when REMAINING == 0, all cells have been solved, and a solution has been found. Solutions found are printed to the output filename designated at the command line in a readable format.

The loop in main controls the continual ruling out of possibilities in the C program. It iterates while there are remaining cells to be solved and it can solve them. First, the loop attempts to find the "obvious" solution, where the bmp array of a particular cell contains all NO except for one UNKNOWN. After reading in the medium difficulty input file shown in Figure 1, the first "obvious" solution appears at t[0][5], where the cell contains what is shown in Figure 3.

t[0][5]

	0	1	2	3	4	5	6	7	8	9
<b>bmp</b>	/	N	N	N	N	N	N	N	N	?
<b>numPoss</b>	1					<b>sol</b>	0			

Figure 3: This shows an "obvious" solution, where the cell at t[0][5] can only hold the solution 9.

FindObvious() calls Found() on this cell and pip value of 9, calling RuleOut\*() to continue to rule out this value for other cells.

If no obvious solutions exist, the function FindRowAns() is called to check if there exists one possible valid pip value left for a particular row. For each row, and for each pip value, it counts the number of UNKNOWN values. If the number of UNKNOWNs for that value in that row is exactly 1, then a solution for that cell can be Found(). The situation shown in Figure 4 would result in a success from FindRowAns(), as the column for pip value 3 contains only one UNKNOWN value.

	1	2	3	4	5	6	7	8	9	sol
t[0][0]	Y	N	N	N	N	N	N	N	N	1
t[0][1]	N	?	N	?	N	N	N	N	?	
t[0][2]	N	?	?	?	?	N	N	N	?	
t[0][3]	N	N	N	N	N	N	N	Y	N	8
t[0][4]	N	?	N	N	?	N	N	N	N	
t[0][5]	N	N	N	N	N	N	N	N	?	
t[0][6]	N	?	N	?	N	N	N	N	?	
t[0][7]	N	N	N	N	N	Y	N	N	N	6
t[0][8]	N	N	N	N	N	N	Y	N	N	7

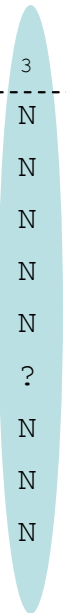
**Figure 4:** The result of a call to DumpTrace(), which prints out the bmp array for each cell in t, as well as the solution for that cell if it has one. The first row's bmp array is shown here. Note that a "row answer" exists for pip value = 3 at t[0][2], shown in blue.

If FindRowAns() returns -1, meaning no "row answers" could be determined, the function FindColAns() is called to try to find a "column answer". It works similarly to the FindRowAns() function, but works with the same column in all rows. The scenario shown in Figure 5 would lead to a Found() column answer at t[4][3] with pip value 5.

	1	2	3	4	5	6	7	8	9	sol
t[0][3]	Y	N	N	N	N	N	N	N	N	1
t[1][3]	N	?	N	?	N	N	N	N	?	
t[2][3]	N	N	Y	N	N	N	N	N	N	3
t[3][3]	N	N	N	N	N	N	N	Y	N	8
t[4][3]	N	?	N	N	?	N	N	N	N	
t[5][3]	N	N	N	N	N	N	N	N	?	
t[6][3]	N	?	N	?	N	N	N	N	?	
t[7][3]	N	N	N	N	N	Y	N	N	N	6
t[8][3]	N	N	N	N	N	N	Y	N	N	7

**Figure 5:** A trace through FindColAnswer(). A column solution is shown in blue.

FindNeighborhoodAns() works just the same, but works with certain ranges of indices. It would find the solution indicated in Figure 6.



	1	2	3	4	5	6	7	8	9	sol
t[3][3]	Y	N	N	N	N	N	N	N	N	1
t[3][4]	N	N	N	Y	N	N	N	N	N	4
t[3][5]	N	?	N	N	N	N	N	N	?	
t[4][3]	N	N	N	N	N	N	N	Y	N	8
t[4][4]	N	N	N	N	Y	N	N	N	N	5
t[4][5]	N	N	?	N	N	N	N	N	?	
t[5][3]	N	?	N	N	N	N	N	N	?	
t[5][4]	N	N	N	N	N	Y	N	N	N	6
t[5][5]	N	N	N	N	N	N	Y	N	N	7

**Figure 6:** A trace through FindNeighborhoodAnswer(), with a neighborhood solution shown in blue.

The continuous finding of solutions happens until there are no more unknown cells' solutions or FindNeighborhoodAns() fails. This causes the program to exit on a special code, freeing allocated memory, and dumping the contents of the TRACE array for easy parsing in the Perl program. The raw data dumped from the TRACE array looks very similar to the data shown in Figures 4-6, with the addition of the length of one side of the puzzle board ( $n^2$ ) printed at the very beginning of the file.

## B. Correctness

The C program finds solutions to the "easy", "medium", and "hard" problems for  $n=3$ , as well as the  $n=4$  problem posted on the project website. It cannot solve 50 of the 81 cells for the "evil" problem.

## C. Complexity

The bulk of the complexity in the C program relies upon the continual ruling out of possibilities and determining the remaining ones. The setup of the TRACE array takes  $O(n^6)$  time, as there are  $n^4$  elements in the  $n^2 \times n^2$  grid, each with a bmp portion of size  $n^2$ . The copying of the integer array *a* to the TRACE array *t* results in a call to Found() for each non-zero value, which is a given solution as input. Found() sets all the values of the bmp array for a particular  $t[i][j]$  to be NO, as well as initializing the sol and numPoss portions in constant time, overall taking  $O(n^2)$  time. It also calls RuleOutRow(), RuleOutCol(), and RuleOutNeighborhood(), each of which take  $O(n^2)$  time to set the bmp array for that found value to be NO for each element in that row, column, and neighborhood.

If  $w$  = number of solutions given in the input file and  $x$  = number of 0's in the input file, the complexity is as follows:

Initialization of TRACE array to default values:

$$O(n^6)$$

Initialization of TRACE array at non-zero values:

$$O(x + w(O(n^2) + O(n^2) + O(n^2) + O(n^2)))$$

$$= O(n^6) + O(x + w(4O(n^2)))$$

As  $w$  approaches  $n^4$ , these quantities become approximately equal, but our sample input files contain solutions for between 32% and 44% of the total cells, which is about half of  $n^4$ . Therefore, the costs of setting up is  $O(n^6)$ , which is the cost of initializing the TRACE array to default values.

After the TRACE array is initialized, additional solutions can be determined using the  $t$  array, as described in section IA. Its ability to find these solutions relies mostly upon its ability to solve cells with an "obvious" answer. The table below shows the breakdown of how each cell of the puzzle was solved, as well as a row for the number of cells not solved, in the case for the Evil puzzle.

Types of Solutions Found for Each Cell (Row Answers checked first)

	Easy (n=3)	Medium (n=3)	Hard (n=3)	N=4	Evil (n=3)
# Solved Cells at Start	36	29	27	110	26
# cells with Obvious answer	29	39	40	84	0
# cells with Row answer	16	13	14	60	4
# cells with Column answer	0	0	0	2	0
# cells with Neighborhood Answer	0	0	0	0	1
# cells unsolved	0	0	0	0	50
total	$= 3^2 = 81$	$= 3^2 = 81$	$= 3^2 = 81$	$= 4^2 = 256$	$= 3^2 = 81$

Table 1: This shows the first implementation of the program's solutions found. The number of solved cells at the start equals the quantity of digits given as input. The totals add up to the exact number of cells, which is  $n^2$  for each.

I hypothesized that these results relied heavily upon the Row Answers found because FindRowAns() is called as the first resort to a failure from FindObvious(). After swapping the placement of FindColAns() and FindRowAns(), the results shown in Table 2 were generated.

Types of Solutions Found for Each Cell (Column Answers checked first)

	Easy (n=3)	Medium (n=3)	Hard (n=3)	N=4	Evil (n=3)
# Solved Cells at Start	36	29	27	110	26
# cells with Obvious answer	33	37	40	91	0
# cells with Row answer	0	2	6	12	0
# cells with Column answer	12	13	8	43	4
# cells with Neighborhood Answer	0	0	0	0	1
# cells unsolved	0	0	0	0	50
total	$= 3^2 = 81$	$= 3^2 = 81$	$= 3^2 = 81$	$= 4^2 = 256$	$= 3^2 = 81$

Table 2: This shows the second implementation of the program's solutions found. It attempts to find the column answers, then upon failure, tries to find row answers.

Similarly, Table 3 shows the results for the ordering of FindNeighborhoodAns(), FindColAns(), then FindRowAns().

Types of Solutions Found for Each Cell (Neighborhood Answers checked first)

	Easy (n=3)	Medium (n=3)	Hard (n=3)	N=4	Evil (n=3)
# Solved Cells at Start	36	29	27	110	26
# cells with Obvious answer	30	37	40	98	0
# cells with Row answer	0	0	4	0	0
# cells with Column answer	0	1	0	0	5
# cells with Neighborhood Answer	15	14	10	48	0
# cells unsolved	0	0	0	0	50
total	$= 3^2 = 81$	$= 3^2 = 81$	$= 3^2 = 81$	$= 4^2 = 256$	$= 3^2 = 81$

Table 3: This shows the third implementation for finding solutions, where the ordering of function calls is FindNeighborhoodAns(), FindColAns(), then FindRowAns().

The differences produced by the ordering of function calls does not change the final solution created, even for the unsolvable evil problem. The number of obvious answers is affected slightly in each case, as each problem may lend itself to be favorable to a certain function, say if it had many row solutions, then FindRowAns() would be the most efficient to execute first. However interesting that method may be, I could not readily determine if a given input was more row, column, or neighborhood favorable, so I tried all three along a cascading failure.

The functions FindRowAns(), FindColAns(), and FindNeighborhoodAns() all traversed similarly through the same amount of cells. The best case would result in  $n^2$  verifications, where the first pip value (1) is examined across an entire row,



column, or neighborhood of  $n^2$  cells, which results in  $O(n^2)$  time. This case is extremely rare, as there are  $n^2$  pip values for each of the  $n^2$  cells within each function, so the chances of the best case are  $1/n^4$ . The worst case, on the other hand, results in the examination of all  $n^2$  cells for all  $n^2$  pip values and no solutions exist. This results in  $O(n^4)$  time. But because FindRow() is called after a call to FindObvious(), a FindRow() result comes after a failed attempt to FindObvious() of cost  $O(n^4)$ , making its solution actually cost  $2O(n^4)$ . Because FindColAns() is called after FindRowAns(), the actual time is  $3O(n^4)$ , as the entire array is searched twice. Similarly, FindNeighborhoodAns() searches again, making it  $4O(n^4)$ .

If all three of the Find\*Ans() functions fail, the contents of the TRACE array are dumped to standard output for the PERL program to use by calling DumpTrace() and exits. This call takes  $O(n^6)$  time, as it takes  $O(n^4)$  to visit each cell in the grid, which has  $O(n^2)$  elements in each cell's bmp array to print. This only happens in the evil input problem, where a solution could not be found.

Two cases exist for the complexity analysis here: solutions that can be found and solutions that cannot be found (due to more possibilities than can be easily ruled out, not to an impossible solution). For solutions that can be found, Tables 1-3 show that the majority of solutions are found by FindObvious(). FindObvious() returns an obvious solution if the numPoss for a particular cell in the TRACE array is exactly equal to 1, taking  $O(n^4)$  time to potentially look at every cell in the TRACE array.

For the easy, medium, hard, and  $n=4$  input problems, the complexity is as follows. Suppose  $w$  is the number of given solutions in the input file,  $y$  is the number of solutions found by FindObvious(),  $z$  is the number of solutions found by FindRowAns(), and  $a$  is the number of solutions found by FindColAns() and FindNeighborhoodAns(). The overall complexity, given these values, is approximately equal to:

$$(yO(n^4)) + (z2O(n^4)) + (a3O(n^4))$$

I lumped the found column and neighborhood answers in this complexity because there were so few of them. Looking at the data files and the quantity of solutions produced in Table 1, the following complexities can be produced. The dominating costs are underlined.

Easy ( $n=3$ ):  $w = 36$ ,  $y = 29$ ,  $z = 16$ ,  $a = 0$ .  
 $= 29(O(n^4)) + \underline{16(2O(n^4))} + 0(3O(n^4)) = 32(O(n^4)) = \theta(n^7)$

Medium ( $n=3$ ):  $w = 29$ ,  $y = 39$ ,  $z = 13$ ,  $a = 0$ .  
 $= \underline{39(O(n^4))} + 13(2O(n^4)) + 0(3O(n^4)) = 39(O(n^4)) = \theta(n^7)$

Hard ( $n=3$ ):  $w = 27$ ,  $y = 40$ ,  $z = 14$ ,  $a = 0$ .  
 $= \underline{40(O(n^4))} + 14(2O(n^4)) + 0(3O(n^4)) = 40(O(n^4)) = \theta(n^7)$

N=4:  $w = 110$ ,  $y = 84$ ,  $z = 60$ ,  $a = 2$ .  
 $= 84(O(n^4)) + \underline{60(2O(n^4))} + 2(3O(n^4)) = 120(O(n^4)) = \theta(n^7)$

The complexity in the searching loop when a solution found is approximately  $n^7$  for all of these cases; therefore the complexity of the meat of this program is  $\theta(n^7)$ . This is greater than the initialization cost of  $O(n^6)$ .

When a solution is not found, the complexity of one run of the program decreases because the program ends prematurely. The only costly thing it does before exiting is exhausting all possibilities of answers by searching FindObvious() and the three Find\*(), then finally calling DumpTrace() of time  $O(n^6)$ . However, this only happens one time. The quantity of solved cells is greatly outnumbered by the quantity of unsolved cells, so the cost is still dominated by the  $O(n^6)$  time for DumpTrace().

## II. Perl Script

### A. Algorithm

The outer script, implemented in Perl, runs the C program. If the C program ends via a found solution or an "impossible" ruling (where no solution can be found), the script ends. If this is not the case, the script parses the data dump generated by the C program. By maintaining a stack of 2-d Sudoku grids, it implements a Depth-First Search that branches upon an unknown cell, generating one grid of size  $n^4$  for each possibility at that unknown cell. This is shown in Figure 7, and takes  $O(n^4)$  time for each possibility at that cell.

9	8	0	0	4	1	0	0	0
2	7	4	0	0	0	0	0	1
0	5	0	0	0	7	4	0	0
4	0	5	9	0	2	0	0	0
0	0	0	0	0	0	0	4	0
0	0	0	8	0	4	9	0	5
0	0	8	6	2	0	0	1	4
6	4	0	0	0	0	0	2	9
0	0	0	4	3	0	0	0	8
9	6	0	0	4	1	0	0	0
2	7	4	0	0	0	0	0	1
0	5	0	0	0	7	4	0	0
4	0	5	9	0	2	0	0	0
0	0	0	0	0	0	0	4	0
0	0	0	8	0	4	9	0	5
0	0	8	6	2	0	0	1	4
6	4	0	0	0	0	0	2	9
0	0	0	4	3	0	0	0	8
9	3	0	0	4	1	0	0	0
2	7	4	0	0	0	0	0	1
0	5	0	0	0	7	4	0	0
4	0	5	9	0	2	0	0	0
0	0	0	0	0	0	0	4	0
0	0	0	8	0	4	9	0	5
0	0	8	6	2	0	0	1	4
6	4	0	0	0	0	0	2	9
0	0	0	4	3	0	0	0	8

Top of the stack

Figure 7: This is the stack of possible grids of solutions. They are used in the Perl program to search, depth-first, on the remaining possibilities of a puzzle. At each pop() of the stack, the grid is printed to a file and fed into the C program to be solved. The blue shaded boxes indicate the estimate for that "push", meaning there were three possible pip values for t[0][1]: 3, 6, and 8, so the program attempts to solve the puzzle on those values.

Each grid is pushed onto the stack, and at every pop of the stack, the grid popped is printed to a file which is then fed into the execution of the C program to find possible solutions. This continues until a valid solution is found.

## B. Correctness

The script finds a valid solution for the evil problem. It also finds a valid solution given an array of all 0's where  $n=3$  after approximately 10 seconds, but searches for a near infinite amount of time on the UMBC server for  $n=4$ .

## C. Complexity

If the C program can solve a particular input on the first try, the Perl program's complexity is the same as the C program plus some minor costs to run the script itself, which is  $\theta(n^7)$ .

If the puzzle cannot be easily solved, the Perl script runs a DFS to trace the possible solutions in a particular cell. Depth-first search has the complexity of  $O(V + E)$  on a given graph of  $V$  vertices and  $E$  edges<sup>1</sup>. For the Perl program's DFS, the vertices are the possible Sudoku grids that can be produced by parsing through the dump from the C program's failed execution. The number of edges that come from each is the number of possibilities (numPoss) for a given cell.

These  $V$  and  $E$  values are highly dependent on the number of unknown cells in the data dump. For the evil  $n=3$  example, even though there are 50 unsolved cells, the depth-first search is able to create just enough grids to create one that is solvable by the C program. The search for the evil example pushes the three grids shown in Figure 5 in the array at  $[0][1]$ , one for each possible value 3, 6, and 8. After the C program fails for the search on that popped value, it pushes the two grids for the values 3 and 6 generated for the unsolved cell  $[0][2]$ , and upon failure, it pushes three more grids for the values 2, 3, and 5 for the unsolved cell  $[0][3]$ . The C program then finds the solution after a search of 8 nodes and 3 pops of the stack.

The only conclusion I can make about the complexity of the Perl script is that it must generate and print a grid of size  $n^4$  for each vertex in the depth first search graph, plus the work that must be done for the popping off of each vertex in the stack, which indicates the cost of an edge. Therefore, the bulk of the Perl script happens as it generates numerous grids and pushes them on the stack, costing  $|V|O(n^4)$  time.

# III. Entire Solution

## A. Complexity

The C program performs more work than the Perl script alone, but the combined effort of the entire program for  $V$  vertices in the depth-first search in the Perl program costs

---

<sup>1</sup> Cormen, Leiserson, Rivest, and Stein, Introduction to Algorithms, second edition, McGraw-Hill, 2001.

$$\theta(n^7) + \theta(n^7) (|V|O(n^4))$$

when it searches for a solution. The running time is simply  $\theta(n^7)$  if a solution can be determined by the C program.

It is not easy to evaluate the complexity of this solution in the theoretical  $O()$  notation because  $n$  is very small. In class, we generally consider the overall complexity of an algorithm for very large values of  $n$ , or at least the amortized (average over many values) complexity. Calculating the exact complexity for my solution is difficult because  $n = 3$ . The number of vertices in the depth-first solution greatly impacts the growth of the search, so I cannot easily extract  $|V|$  to be some constant  $O(1)$  outside of the  $O(n^4)$  complexity, but cannot assign  $|V|$  to be an approximation in terms of  $n$  either. Because  $n$  is so small, every operation that we usually regard in some constant time may amount to more work than operations performed  $n$  time.

The most precise conclusion I can make is that my C program discovers solutions for a given input in approximately  $O(n^7)$  time, and the outer Perl script takes less time to generate more input files for the C program.

## B. Future Work

The complexity of the C program could have improved if I kept track of the remaining rows, columns, and neighborhoods to be solved to cut out some of the  $O(n^4)$  searching within the entire Sudoku/TRACE grid. If I could access the next cell to be solved in constant time instead of searching for it, performance would improve. Perhaps a modified min-heap implementation could give quicker access to unsolved cells, forcing cells with valid solutions to trickle down to the bottom, never to be searched again.

I am also questioning the use of the structure for known solutions in the puzzle, or even their presence in the array of puzzle cells in which to search. Once a cell has been solved, it should be removed from the grid of cells to cut down on the time to search for the next solvable cell. But then this implementation would lead more bookkeeping so that the cells would keep their indices, and to print the solved and unsolved cells to a file would require sorting, taking  $O(n^2 \log n^2)$  more time. The cost of additional bookkeeping may outweigh the benefits of a faster search, but only more research can determine for sure.

If I had  $n^2$  parallel processors to work with, the complexity would decrease as the ruling out of a single row, column, or neighborhood could happen in (approximately) constant time. This might be another application for parallel processing research, although I hope they have better things to study than newspaper games.