Don Taylor
CS325
Portfolio Project
3/3/2021

Sudoku

Sudoku is played on a grid of n^2 by n^2 squares. The grid is further divided into n^2 segments of size n by n. In each square the player must put a number from 1 to n^2 such that each row, column, and segment contains each number from 1 to n^2 with no repeated numbers. Typically, the game is played on 9x9 grid, and each square must be filled with the numbers 1 through 9.

Unsolved:

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

Solved:

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

Verification Algorithm:

My algorithm determines the correctness of a solution by individually checking each row, column, and segment. If a row, column, or segment doesn't contain the numbers 1-9 without repetition, the algorithm returns false.

The puzzle is stored as a matrix, or a list of 9 lists. The main work is being done in the check_list() function. Once we are able to determine if a single list contains the numbers 1-9, with no repetitions, we can simply check each row, each column, and each segment.

```
def verify_puzzle():
    check_lists(rows)
    check_lists(columns)
    check_lists(segments)

    if all three are true:
        return True
    else:
        return False.

def check_lists(lst):
    “””Takes a list of 9 lists and verifies each list contains the numbers 1 through 9 with no
    repetitions.

    for each list:
        check_list()

    if check_list returns true for every list:
        return True
    else:
        return False

def check_list(list):
    “””Takes a list as a parameter. Returns true if the the list contains the numbers 1 through
    9 with no repetitions. “””

    for i = 1 —> 9:
        if i is in the list:
            remove it
        else:
            return False

    if list.length == 0:
        return True
    else:
        return False
```

Proof of Correctness:

check_list looks for each number 1 through 9 in the list. If it's not there, then we return False. If it is there we remove it. Once he have removed 1 through 9, we check that the list is empty. If we have successfully removed each number 1 through 9, and the list is empty, then each number exists in the list and there are no duplicates. check_list() will return True.

Now that we can successfully verify the correctness of a single list, we simply check each row, each column, and each segment. In doing so, we can verify the correctness of the entire puzzle. If check_list returns True for each row, column, and segment, verify_puzzle() will return True.

If any row, column, or segment contains a duplicate, or does not contain each number 1-9, then check_list(), will return False, check_lists() will return False, and verify_puzzle() will return false.

Time Complexity:

Even though by the puzzle definition the grid size is n^2 by n^2, (in our case 3^2 by 3^2), for the purposes of discussing time complexity we will describe the grid as n by n, and let n = 9 for our standard size.

check_list() runs in O(n) time, as it looks at each number once.

check_lists() runs in O(n^2) time. It takes a list of size n and runs check_list() on each list.

verify_puzzle() runs in O(n^2) + O(n^2) + O(n^2), as it performs check_lists() 3 times, once for rows, once for columns, and once for segments. We can simplify this to O(n^2).

Therefore, the runtime of the verification algorithm is O(n^2).

Solving Algorithm:

My algorithm solves a sudoku puzzle by moving through the puzzle sequentially, starting in the top left and moving towards the bottom right one cell at a time. In each cell, numbers 1 through 9 are tried, if it a number is found to be valid we move to the next cell and try numbers there. If a number is found to be invalid, the next number is tried. If all 9 numbers have been tried in a cell and none of them are found to be valid, we move back to the previous cell and increment the number there.

Time Complexity:

The time complexity for this algorithm is O(n^(m+2)), where n is the size of the grid (9 for a typical sudoku puzzle) and m is the number of empty spaces in the puzzle.

For each empty cell in the puzzle, there are n possible entries. In the worst case, the algorithm must consider every possible combination of entries. If there are 3 empty spaces, we have n * n * n possible solutions. If there are m empty spaces, we have n^m possible solutions. The algorithm could ultimately try every possible solution until it finds one that works. For each possible solution we must use the verification algorithm, which has a time complexity of O(n^2).

(n^m) * (n^2) = n^(m+2)

In the case of a completely empty puzzle, the time complexity would be (n^(n^2+2)). If the puzzle is completely empty, there are n^2 empty spaces, and thus n^(n^2) possible solutions. Again, we must use the O(n^2) verification algorithm for each possibility.

n^(n^2) * n^2 = n^(n^2 + 2)