The algorithm chose for the task was the constraint satisfaction. The main reason I chose this algorithm is because we are not trying to find a valid path of possible actions, but simply trying to find a goal state. In our case, all sudokus have one solution only, so no matter how we get to this solution, it only matters to get there. Also, it was the algorithm you guys recommended. I think the constraint satisfaction is well suited for this kind of problem because we are dealing with a constraint satisfaction problem: we need to select a number respecting some constraints. It's like the eight queens' problem, where we have some constraints and an initial board partially filled. The algorithm works by recursively trying to fill in the empty cells of a Sudoku board with numbers from 1 to 9, ensuring that each number appears exactly once in each row, column, and 3x3 unit. In more detail:

- We find an empty cell on the board using the *emptyCell* function. This function returns a coordinate [row, column]. If or when no empty cell is found, we found a solution for the Sudoku.
- We find a number and input it on the coordinates we found using the function *inputNumberOnEmptyCell.* This function calls a function called *checkMove*, that checks which number we can input on the coordinates.
- *checkMove* calls three other functions: *checkRows*, *checkColumns* and *checkUnit*. The first one check if the number selected repeats in the row, the other one if it repeats in the column and the last one if it repeats in the box 3x3. If so, we change the number.
- After inputting the number, we move to the next cell calling *emptyCell* again. If there are none available, we have filled the board and the function returns True, if not, we recursively start over. If no number could be assigned to the coordinate, we backtrack and return False. Backtracking means it's going to erase the previous input number and try another one. If the board doesn't have a solution, we return False.

The most important idea for this algorithm is that if a number can be placed, it updates the board with that number, appends the move to the tracker list, and recursively calls itself for the next empty cell. If a solution is found (i.e., all cells are filled), it returns True. Otherwise, it backtracks, deletes from the track list the number, row, and column, and returns False. Below there is an example of the algorithm backtracking:

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

```
1st backtrack interaction = [[5, 9, 1, 4, 3, 2, 6, 7, 0],
2nd backtrack interaction = [[5, 9, 1, 4, 3, 2, 6, 0, 0],
3rd backtrack interaction = [[5, 9, 1, 4, 3, 2, 0, 0, 0],
4th backtrack interaction = [[5, 9, 1, 4, 3, 2, 7, 0, 0],
5th backtrack interaction = [[5, 9, 1, 4, 3, 2, 0, 0, 0],
6th backtrack interaction = [[5, 9, 1, 4, 3, 0, 0, 0, 0],
```

In the first backtrack interaction, no number can be assigned, so the function returns False and deletes the previous input. Now, the loop (that is currently at 6), tries to assign a value to new empty cell and can't again, so it deletes the previous one. This happens one more time, and then the algorithm can place a 7, but no number after that, so it erases again. The algorithm places the first available number going from 1-9 and backtracks when it fails.

The "tracker" I added is for future use, where we can see how the algorithm inputted the numbers and how it backtracked. This list holds the value of the input number, the row, and the column. It pops out of the list when the algorithm backtracks. Maybe we can find some use for this in the future (I left this in the code because doesn't affect the performance). Performance is increased by using separate functions to check rows, columns, and 3x3 units, that way the algorithm avoids unnecessary iterations and improves efficiency. Also, the algorithm terminates early if it encounters a board with no empty cells, indicating that the puzzle is already solved and backtracking is employed to explore different possibilities efficiently, and pruning is performed by reverting invalid moves to avoid exploring unpromising branches of the search tree. The time complexity of the backtracking algorithm is exponential, but the actual runtime depends on factors such as the number of empty cells and the complexity of the puzzle.

The algorithm's performance can be evaluated using metrics such as solving time, number of recursive calls, and the complexity of the puzzles solved. While the backtracking algorithm efficiently solves medium to hard Sudoku puzzles, it may struggle with extremely difficult or irregularly shaped puzzles due to the exponential search space. Further work could focus on improving performance, such as implementing more sophisticated techniques such as constraint propagation or heuristics like Minimum Remaining Values (MRV), that could enhance the solver's performance. Also, parallelization: utilizing parallel processing techniques to explore multiple branches of the search tree concurrently could speed up the solving process.