




Final Project Report

Blanca Lozano Collado

In order to implement a Sudoku solver in Python, I have done some research into several algorithms & techniques and explored how to integrate them with data structures. But before I get into the code, I need to make sure I have a couple important ideas clear.

My project has the following structure once you unzip *sudoku_project*:

 levels	15/05/2025 12:47	Carpeta de archivos
 report	15/05/2025 12:44	Carpeta de archivos
 src	15/05/2025 17:14	Carpeta de archivos







In **levels** there are 4 .txt files with the following characteristics:

1. Easy: 30+ clues
2. Medium: 25-30 clues
3. Hard: 23-25 clues
4. Expert: 17-22 clues

(These files are going to be our boards using `load_puzzle()` in `utils.py` later on)

In the folder **report** is this file, in which I explain different algorithms & techniques and overall how the project was done.

In **src** (main source code) you will find all the coding I implemented to solve this problem in which you will find the following files:

 __init__	15/05/2025 12:54	Archivo de origen ...	1 KB
 main	15/05/2025 16:37	Archivo de origen ...	1 KB
 solver_backtracking	15/05/2025 17:25	Archivo de origen ...	1 KB
 solver_branchbound	15/05/2025 14:45	Archivo de origen ...	1 KB
 sudoku	15/05/2025 16:45	Archivo de origen ...	2 KB
 utils	15/05/2025 17:14	Archivo de origen ...	1 KB

The first algorithm I chose to implement was **Backtracking**, which recursively attempts to fill the sudoku board with 'guesses' while also obeying the rules of the sudoku. To do this, I created the next code:

```
def solve_backtracking(sudoku):
    # we guess numbers from 1-9 and:
    # - solve if it's a good guess
    # - go back if it's a bad guess (backtracking algorithm)

    cell = sudoku.find_empty()

    if cell is None:
        return True

    row, column = cell

    for guess in range(1,10): # 1-9
        if sudoku.valid_option(row, column, guess):
            sudoku.puzzle[row][column] = guess

            if solve_backtracking(sudoku):
                return True

            sudoku.puzzle[row][column] = 0

    return False
```

With line number 5 I find the next empty space in our board with the method `find_empty()` from class `Sudoku`.

Lines number 7 & 8 verify that if there are no empty spaces, the sudoku is completed and it returns `True`.

With lines from 10 to 17, I create a for loop in which I go through numbers from 1-9 (possible guesses). Then I need to verify if a guess is a valid option using the method `valid_option(row, column, guess)` from class `Sudoku` (in which the logic for this method is explained); and if it is, then I place the guess on the puzzle with line number 12.

Now I need to complete the rest of the puzzle so I recursively solve it with line 14 and if a guess is NOT a valid option (meaning it won't go into the if statement from line 11) then I need to reset it's value to 0 in our board.

And finally, line 19 indicates that if none of the guesses are right, then the sudoku is unsolvable and I return `False`.

Being our `find_empty()` method from `Sudoku` class:

```
def find_empty(self):
    # this method finds the first row,column (cell) that is not filled yet (represented with 0)

    for i in range(9):
        for j in range(9):
            if self.puzzle[i][j] == 0:
                return i,j

    return None # if we reach this point, there are no empty cells so we return None
```

The next algorithm I implemented was **Branch & Bound** which is pretty similar to backtracking as it also attempts to fill a space by making guesses and recursively attempts to solve the board. However, the main difference is that instead of focusing on the first empty space, I focus on the cell with the **fewest valid options**, which increases the chance that the bad guesses will be detected earlier. It may seem similar to backtracking but this technique reduces unnecessary work and is more efficient overall.

To solve it, I implemented the next code:

```
def solve_branchbound(sudoku):

    # we go for the space with fewest possible guesses
    # we 'remove' a path if we find that it's not a good path to follow

    cell = sudoku.find_best_cell()

    if cell is None:
        return True

    row, column = cell

    for guess in range(1,10):
        if sudoku.valid_option(row,column,guess):
            sudoku.puzzle[row][column] = guess

            if solve_branchbound(sudoku):
                return True

            sudoku.puzzle[row][column] = 0

    return False
```

This code implements the same logic as the previous one, being the only difference that I find the best possible empty cell instead of the first empty one.

Our method find_best_cell() being:

```
def find_best_cell(self):
    min_options = 10
    best_cell = None

    for i in range(9):
        for j in range(9):
            if self.puzzle[i][j] == 0:
                options = 0
                for guess in range(1,10):
                    if self.valid_option(i,j,guess):
                        options += 1
                if options < min_options:
                    min_options = options
                    best_cell = (i,j)
                if min_options == 1:
                    return best_cell

    return best_cell
```

It works by iterating over each empty cell. For each cell, I count the number of valid options by checking which numbers can be placed, then I increment a counter called options for every valid number I find. After calculating the valid options for each cell, I compare these counts and keep track of the cell with the fewest valid possibilities. Finally the method returns the location of the final best cell.

Comparing both algorithms, I find **backtracking** easier to understand and implement as it just keeps trying guesses until one is good in the next empty space. However, **branch&bound** tends to find the best empty space with the fewest possibilities, which was a bit confusing to understand at first.

Comparing the efficiency of each code, I find that branch&bound tends to perform better in many cases because it uses a more strategic approach to reduce the number of possibilities by 'bounding' or eliminating branches that are guaranteed not to lead to a valid solution, this can significantly reduce computing time (especially for larger problems than a sudoku).

The reference I used to create this project was the following:

GeeksforGeeks. (2023, 30 enero). *Difference between Backtracking and BranchNBound technique.* GeeksforGeeks.

<https://www.geeksforgeeks.org/difference-between-backtracking-and-branch-n-bound-technique/>