This document is a description of the two Sudoku solvers built for this project. The first part details the Z3 theorem prover based solution, whilst the second details the implementation of a chronological backtracking based algorithm. For a detailed description about how to run the tools, please read the `README` file.

The input

The solvers take as input a file containing:

```
n
X X X ... X X
X X X ... X X
X X X ... X X
⋮ ⋮ ⋮ ⋱ ⋮ ⋮
X X X ... X X
X X X ... X X
```

- Where `n` is the size of the grid (`n`×`n`). `n` can be 4, 9, 16, or 25
- Whitespace and blank lines are ignored
- `X` can be _ or - or . for a square without value given initially, or a hexadecimal digit (`123456789ABCDEF...P`) up to `n`. It can be uppercase or lowercase

1 Z3 based solver

Our goal here is to use the Z3 API in Python to solve a given Sudoku problem.

1.1 Source files

- `src/z3_solver.py`: the main program
- `src/util/tools.py`: contains parsing utilities to read the input grid read from a file, as well as printing functions to print the Sudoku grid in a specific format

1.2 Data structures

- **Arrays**: used to initially store the Sudoku grid lines (to be parsed) that are read as a file given in the first argument to the program. Parsing consists in preparing a 2-dimension array representing the grid

  Arrays are also used to describe the Sudoku solution as a vector of Z3 variables, as follows:

  ```
  # Initialize solution vector
  sol = [ Int(f"s{int_to_hex(i)}_{int_to_hex(j)}") for i in range(n) for j in range(n) ]
  ```

  So that it takes the following format:

  ```
  [s0_0, s0_1, s0_2 ... s1_0, s1_1, s1_2 ... s{n-1}_{n-2}, s{n-1}_{n-1}]
  ```

1.3 SAT Solving

We want to reduce the problem to SAT. So we build a **Solver Object** to which we add the following constraints:

- The solution must be initialized with the initial Sudoku grid digits
- Digits of the solution are natural numbers belonging to the interval [1,n]
- For each line, digits are mutually distinct

- For each column, digits are mutually distinct
- For each box ($\sqrt{n} \times \sqrt{n}$), digits are mutually distinct

All these constraints are added to the solver object. As to the last constraint (distinct box values), it was divided into multiple constraints, each representing mutual exclusion of digits hold by two squares in the same box.

Then, we check the satisfiablity of the resulting formula. If satisfiable,we print the solution.

2 Backtracking based search engine

2.1 Source files

- `src/backtrack_solver.py`: the main program
- `src/util/algo.py`: contains the recursion algorithm for backtracking
- `src/util/config.py`: contains parameters used to configure the program

2.2 Data structures

- We use **arrays** to track the evolution of the following parameters:
  - `grid`: a 2-dimension array containing the initial grid
  - `sol`: a 2-dimension array representing the updated state of solution search
  - `pre`: used to store the initially blank squares of the Sudoku grid. It is updated as the searching advances
  - `pos`: used to keep track of the digits that have been tried for every blank square. It is indexed identically as the `pre` array. Each value of this array denotes the index of the digit that is being picked at a given moment for the corresponding square. (the following subsection describes how digits are picked)

2.3 Algorithm: General overview

Input: Sudoku problem
Result: Sudoku solution
`pos` := $\varnothing$;
`pre` := $\varnothing$;
`i` := 0;
`j` := 0;
`SOLVED` :=true;
return FIND_SOLUTION(`grid`, `n`, `i`, `j`, `pos`, `pre`);

```
function FIND_SOLUTION(grid, n, i, j, pos, pre)
if grid[i][j] == 0 then
    if encountered blank square for the first time then
        pre.add(i , j)
        get a valid digit for this square;
        if a valid digit exists then
            # put digit in the grid:
            grid[i][j] = validDigit;
        else
            # backtrack, or stop if impossible to find a solution:
            if there exists other possibilities to try then
                # compute previous square to backtrack to:
                return FIND_SOLUTION(grid, n, previous_i, previous_j, pos, pre);
            else
                # unsatisfiable:
                return grid, not(SOLVED);
            end
        end
    end
end
if achieved the end of the grid then
    return grid, SOLVED;
else
    # move to the following blank square:
    compute next (i , j);
    return FIND_SOLUTION(grid, n, next_i, next_j, pos, pre);
end
end function
```

2.4 Algorithm - Implementation choices

- Picking the next digit to assign: we use a simple method that consists in finding all the possible digits for a square assuming the hypothesis that the previously picked digits are correct. At each time, we pick the smallest digit among the possibilities; the array `pos` contains the next position of the digit to be picked for every square of the grid
- Pure backtracking: we intentionally choose to only keep a pure back-tacking algorithm. Otherwise, to speed up search we could've started by filling the trivial squares (squares with an initially single possible digit) with the appropriate digits before running the algorithm
- Properties: this program to implements the *Soundness* property; it either finds a solution or reports unsatisfiability (when given a grid that is not a Sudoku problem)
- Recursion limit: the recursion depth limit is easily achieved with this program. As a solution, we break the recursion when we are 10 steps behind the recursion limit, we save the state of parameters used to track the search, then we re-launch the algorithm. And so on until a solution is found (or if the problem in unsatisfiable). We specify the recursion limit in the beginning as:

  ```
  RECURSION_LIMIT = sys.getrecursionlimit()-10
  ```

- Keeping a trace of the search: we output the state of search to a log file (`trace.log`); we print the evolution of parameters denoting the picked digits for a given square, and the depth of backtracking. Compared to the pseudo-code above, the FIND_SOLUTION function in the source code contains an additional parameter to indicate the backtracking depth

2.5 Algorithm - Customization

The following parameters can be customized, by modifying them in the file `src/util/config.py`:

- `RECURSION_LIMIT`
- `LOGGING_STEP`: the program notifies the state of the grid every {`LOGGING_STEP`×`RECURSION_LIMIT`} steps

2.6 Possible enhancements

This search engine can be improved further by:

- implementing some good heuristics for picking the next digit to assign
- checking the uniqueness of a solution

Comparison

The following table synthesizes a comparison between the latency of each of the two solvers, when they were given some samples (available in folder `samples/`) to solve:

Samples	time for: `z3_solver.py`	time for: `backtrack_solver.py`
`grid_9.txt`	0.411 s	0.042 s
`grid_16.txt`	1.667 s	0.040 s
`grid_25.txt`	5.753 s	26.841 s