CS202A: Mathematics For Computer Science-II

Assignment 1 Report

Aditya Tanwar
200057

Akhil Agrawal
200076

February 2, 2022

Part 1

Statement: Given a sudoku puzzle pair S_1, S_2 (both of dimension k) as input, your job is to write a program to fill the empty cells of both sudokus such that it satisfies the following constraints

- Individual sudoku properties should hold.
- For each cell $S_1^{ij} \neq S_2^{ij}$, where i is row and j is column.

Notation: We use *box* to mean a $k \times k$ sub-square in the overall grid, and use (n, m) to identify it, where n is its number vertically, and m is its number horizontally. Naturally, their range is $1 \leq n, m \leq k$. Further, we define $v' \coloneqq v - 1$ for any variable v in order to switch to *0-based indexing*.

Encoding: Let $a_{(s,x,i,j)}$be a boolean variable which is T if the x^{th} number is written in the cell at i^{th} row and j^{th} column in the s^{th} sudoku, and F otherwise. Clearly then, for 2 sudokus of dimensions (of smaller boxes) k, the variables will be in the range $1 \leq x, i, j \leq k^2$, and $1 \leq s \leq 2$.

Constraints: The problem was encoded in the form of these expressions-

- Cell: Each cell must have exactly one number (in the range [1, k^2])-
  - Each cell [i_0, j_0] in the s_0^{th} has at least one number (in the range [1, k^2]), written in it-

$$\bigvee_{1 \leq x \leq k^2} a_{(s_0,x,i_0,j_0)}$$

  - Each cell [i_0, j_0] in the s_0^{th} has at most one number (in the range [1, k^2]), written in it-

$$\neg a_{(s_0,x_1,i_0,j_0)} \vee \neg a_{(s_0,x_2,i_0,j_0)} \qquad \forall\, 1 \leq x_1 < x_2 \leq k^2$$

- Row: Each row i_0 in the s_0^{th} must have exactly one occurrence of each number in the range [1, k^2]-
  - At least one occurrence of each number-

$$\bigvee_{1 \leq j \leq k^2} a_{(s_0,x,i_0,j)} \qquad \forall\, 1 \leq x \leq k^2$$

  - At most one occurrence of each number-

$$\neg a_{(s_0,x,i_0,j_1)} \vee \neg a_{(s_0,x,i_0,j_2)} \quad \forall\, 1 \leq j_1 < j_2 \leq k^2 \qquad \forall\, 1 \leq x \leq k^2$$

- Column: Each column j_0 in the s_0^{th} must have exactly one occurrence of each number in the range [1, k^2]-
  - At least one occurrence of each number-

$$\bigvee_{1 \leq i \leq k^2} a_{(s_0,x,i,j_0)} \qquad \forall\, 1 \leq x \leq k^2$$

  - At most one occurrence of each number-

$$\neg a_{(s_0,x,i_1,j_0)} \vee \neg a_{(s_0,x,i_2,j_0)} \quad \forall\, 1 \leq i_1 < i_2 \leq k^2 \qquad \forall\, 1 \leq x \leq k^2$$

- Box: Each box (n_0, m_0) in the s_0^{th} sudoku must have exactly one occurrence of each number in the range [1, k^2]-

- At least one occurrence of each number-

$$\bigvee_{\substack{(n-1)\cdot k<i\leq n\cdot k\\(m-1)\cdot k<j\leq m\cdot k}} a_{(s_0,x,i,j)} \qquad \forall\, 1\leq x\leq k^2$$

- At most one occurrence of each number-

$$\neg a_{(s_0,x,i_1,j_1)} \vee \neg a_{(s_0,x,i_2,j_2)} \quad \forall\,(n-1)\cdot k<i_1<i_2\leq n\cdot k,\ \ (m-1)\cdot k<j_1<j_2\leq m\cdot k \quad \forall\, 1\leq x\leq k^2$$

- Different values: For each cell $[i_0,\, j_0]$ in both the sudokus, the value written in them should be different-

Hashing: Since the SAT solvers in *PySAT* use integers, we have hashed the boolean variable(s) $a_{(s,x,i,j)}$to integers using the following method-

$$a_{(s,x,i,j)} = s'\cdot k^6 + x'\cdot k^4 + i'\cdot k^2 + j' + 1$$

We elaborate the reasons behind the choice of the function below-

- $+\mathbf{1}$ has been added because the literals should start from 1 in the solver, and not 0.
- $\mathbf{v'}$ variables have been used in place of v variables because they allow us to work in a "continuous" space.
- This hash function leaves no *unused* variables in the range $[1, 2k^6]$, hence the "continuous" use. This is significant because it reduces the number of redundant solutions, and reduces the number of *unused* variables to 0, thus, improving run-time by helping the SAT solver.
- Given a hash, it is easy to reverse engineer it and find (s', x', i', j') and hence find (s, x, i, j), since it is essentially just a number in base k^2.

Redundancy: These three constraints (*1*, *2*, *3*), are redundant, if the rest of the constraints are applied, i.e., the the constraints excluding these three are completely sufficient to make a *Pair Sudoku Solver*, and can be removed altogether.
Nonetheless, they have been used in the code because a speed-up in run-time was observed in some cases. Further, *this paper* reinforced our hypothesis that redundancy helps-

> "Indeed, it is well-known that redundant information can actually help SAT solvers"

Analysis:

- Variables: It is easy to see that the number of variables $(a_{(s,x,i,j)})$ is $2\cdot k^2\cdot k^2\cdot k^2 = 2k^6 = \underline{\mathcal{O}(k^6)}$.
- Clauses:
  - Cell: Each cell contributes to 1 clause from "at least" constraint and $\binom{k^2}{2}$ clauses from "at most" constraint.
  - Row: Each row contributes to $k^2\cdot k^2$ clauses from "at least" constraint and $k^2\cdot\binom{k^2}{2}$ from "at most" constraint.
  - Column: Each column contributes to $k^2\cdot k^2$ clauses from "at least" constraint and $k^2\cdot\binom{k^2}{2}$ from "at most" constraint.
  - Box: Each box contributes to $k\cdot k\cdot k^2$ clauses from "at least" constraint and $k^2\cdot\binom{k^2}{2}$ from "at most" constraint.

Summing these up, for 2 sudokus, we get-

$$2\cdot\left((k^4)\cdot\left(1+\binom{k^2}{2}\right)+3\cdot(k^2)\cdot\left(k^4+k^2\binom{k^2}{2}\right)\right) = \underline{\mathcal{O}(k^8)}$$

Part 2

Statement: In the second part, you have to write a *k-sudoku* puzzle pair generator. The puzzle pair must be maximal (have the largest number of holes possible) and must have a unique solution.

Approach: We took an empty $k \times k$ sudoku, and filled either a row, or a column, or a box (exclusively), with a random permutation of the range $[1, k^2]$. *(This was done to introduce a sense of randomization, and consequently uniqueness in each sudoku pair).*
Then, we used a single-sudoku solver to complete the rest of this first sudoku, and initialized the second sudoku to be completely empty (filled with 0's throughout), and sent this pair to the solver in the first part to be filled completely with any solution.
Then, we iterated through the cells in the sudokus randomly and checked for each cell, whether removing it increased the number of possible solutions. If removing the number written in the cell, doesn't give rise to a new solution, then we simply empty the cell, else, we let the cell retain its value.

Proof: If emptying any cell yields another solution, then revisiting it after emptying another cell is futile, because it will yield at least just as many solutions as the last time. This is because emptying a cell cannot decrease the number of solutions.
The decision to delete the current cell is greedy, and each such greedy choice would be included in a *"maximal"* solution, because all the remaining cells yielded multiple solutions on emptying them, thus, ensuring maximality.

Runtime Statistics

These are some of the total run-times obtained for solving an empty sudoku pair with the *solver* implemented in *Part 1*. The choice of empty sudokus was to standardise testing, and avoid arbitrariness/ luck.

	k	Time (in s.)
1.	2	0.0025
2.	3	0.03
3.	4	0.32
4.	5	2.55
5.	6	10.81

These are some of the total run-times obtained for generating a single sudoku-pair by the code written for *Part 2*.

	k	Time (in s.)
1.	2	0.05
2.	3	4.47
3.	4	163.56

We hypothesise that this drastic increase in run-times in *Part 2* as compared to *Part 1* is a result of $2 \cdot k^4$ many calls to the *solver* implemented in *Part 1*. Consequently, we were not able to obtain time taken for $k = 5$. Summarising, we can write-

$$T(P_2) = O(k^4) \cdot T(P_1)$$