Lab 1: Sudoku

Alex Wenstrup, Hwei-Shin Harriman, Lilo Heinrich

Due March 15, 2021 by 12 pm EST

1 Description

A description of the game and, if necessary, the description of the specific formulation of the game that is NP complete. (Remember that pictures are worth 1000 words and also that we are functionally illiterate, so while long descriptions will be helpful, images and figures will be too.)
For this lab, we will prove that Sudoku is an NP-Complete problem. In this lab, we consider the Sudoku Decision Problem (SDC) to be the question: given a partially filled in Sudoku grid, is it possible to fill in the rest of the grid while obeying the rules of Sudoku?

The rules of Sudoku are as follows:

- Sudoku is played on a square grid of size n by n, where n is a perfect square.
- The goal is to fill in every square in the grid with a number in $\{1, 2, ...n\}$
- Placements are restricted as follows:
  - No row may contain the same element more than once
  - No column may contain the same element more than once
  - When the Sudoku "board" is broken into subgrids of size $\sqrt{n}$ by $\sqrt{n}$, no subgrid may contain the same element more than once.

An example sudoku puzzle is shown below.

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

Figure 1: An example sudoku puzzle. Numbers in black were given, and numbers in red were filled in to complete the puzzle.

2 Proving Sudoku is NP-Complete

You need to prove that your problem formulation is NP-Complete. To do this, you should include a proof of reduction from one of of the Karp's 21 NP-Complete Problems or the Traveling Salesman Problem:

2.1 Sudoku is NP

To prove that Sudoku is in NP, we must verify that a solution to a given instance of a Sudoku puzzle can be guessed and checked in polynomial time. Given a Sudoku puzzle of dimensions nxn, it is possible for an to guess the values of the remaining empty squares in at most n^4 steps. Checking the assigned values of the rows, columns, and subsections would be at most n^2 steps each, making the overall runtime of a guess and check algorithm $O(n^4)$, which is in polynomial time. Therefore, Sudoku is in NP.

2.2 Sudoku is NP-hard

After spending many hours reviewing the following papers in detail:

- Latin Square to Sudoku
- Sudoku to Hamiltonian path
- Failed 3SAT to Sudoku, Latin Square to Sudoku
- Sudoku to SAT
- Latin Squares, Triangle Partitions, and 3SAT

we have found little intuitive literature documenting the NP-Completeness of Sudoku. Most literature documents methods for solving Sudoku puzzles with other NP-Complete problems (proving that Sudoku is in NP), but seemingly the only proof for the NP-Completeness of Sudoku is a reduction from the Latin Square problem, a problem closely related to Sudoku, and not one of Karps' 21 NP-Complete problems. The reduction from 3SAT to Sudoku can be done in the following multi-step reduction:

1. First, reduce 3SAT to the the TTP (tripartite triangle partition) problem. The TTP problem asks whether a given tripartite graph's edges can be partitioned into triangles. The proof of this reduction is shown here: Holyer. See definitions of important terms below.
    - **tripartite graph:** a graph whose edges that can be divided into 3 sets in such a way that each set contains no edges internal to that set
    - **partition:** divide into exhaustive, non-overlapping groups
2. Next, prove that TTP reduces to the Latin Square problem This involves representing a partial (incomplete) Latin Square with a defect graph. The proof of this is shown here: Colbourn. See definitions of important terms below, and see our proof of this step in Appendix A (section 4.1).
    - **latin square:** an n by n grid whose boxes all are all filled with an element of $\{1, 2, ...n\}$, and whose rows and columns never contain more than one of any number
    - **partial latin square:** a latin square with one or more empty boxes
    - **latin square completeness:** the decision problem of whether a partial latin square can be filled to become a latin square
    - **defect graph:** a tripartite graph representing a partial latin square. It is defined in more detail below, in Appendix A (section 4.1).
3. The final step is to reduce the Latin Square Completion problem to Sudoku Puzzle Completion. This involves expressing a Sudoku for any given Latin Square, such that the Sudoku is only solvable if the Latin Square is. The full paper can be found here: Hoexum. Important terms are listed below, and the full reduction can be found in Appendix B (section 4.2).

- **Sudoku Puzzle Completion (SPC)**: the decision problem that determines if a given Sudoku puzzle is solvable.
- **order**: With regard to Latin Squares, the order refers to the dimension of the square, generally denoted by n. With regard to Sudoku, the order refers to the length of a sub-block. Note that this means that the total side-length of a given Sudoku puzzle is n^2, so a traditional Sudoku puzzle of order 3, will have a total side length of 9, and 9x9 individual squares. See Figure 3 for a visual representation.
- **stack**: A column of nxn subsections within a Sudoku puzzle with a total side-length equal to n^2, which must contain one of each number from 1 to n^2. See Figure 3 for a visual representation, where the blue area represents a stack.
- **band**: A band refers to a row of nxn subsections within a Sudoku puzzle. For a traditional Sudoku puzzle of order 3, there are 3 possible bands, the first contains rows [1,3], the second contains rows [4, 6], and the last contains rows [7, 9]. See Figure 3 for a visual representation. The red area represents a band

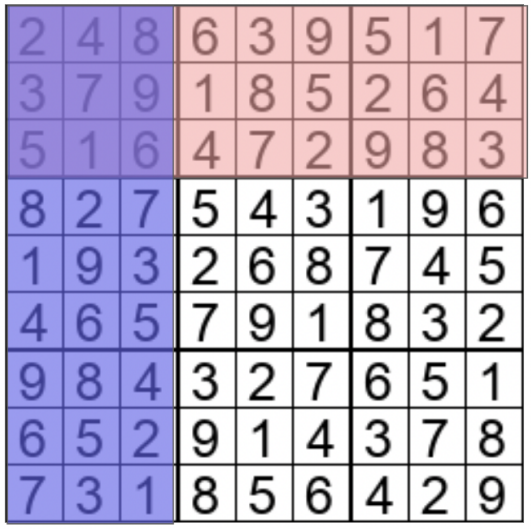

Figure 2: A order $n = 3$ Sudoku puzzle. The blue area shows one of three stacks, and the red area shows one of three bands.

3 Implementation

So here you have two options (depending on how much work you did for the other parts). You could either a) try to code your own custom solver for your problem based on the reduction (feel free to also outsource any necessary starter code if you want), or b) run a CNF-SAT configuration (which you will have to figure out and explain to us) of your problem through an existing SAT Solver and analyze the results of that (in terms of game 'accuracy,' time, ease of use, etc.)

Edit: We anticipate you testing your implementation via test cases. Please upload any files related to the tests to your Github repository, and include at least one sample test case and output in your submission write-up.

Our Sudoku solver converts the given Sudoku problem into CNF form and runs it through a SAT solver, then converts back to a Sudoku solution.

3.1 Convert Sudoku to CNF

Lynce provides a SAT encoding of an $n = 3$ Sudoku puzzle, so we familiarized ourselves with this encoding scheme and generalized it to work for any n value.

To convert a Sudoku problem into CNF form, first consider how to represent a Sudoku puzzle using only booleans. Let n be the size of the Sudoku puzzle, defined as the side length of each sub-grid (for example, normal 9 by 9 Sudoku has a size of $n = 3$). The puzzle has n^2 rows, n^2 columns, and n^2 possible values for each entry (row and column combination). Create a boolean literal for each row, column, and value combination and refer to it by the notation s_{xyz} where x is the row number, y is the column number, and z is the value. This adds up to n^6 total literals.

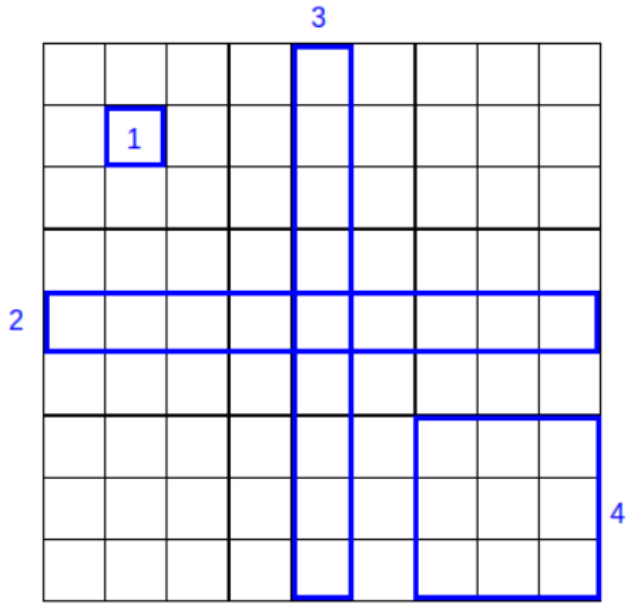

Figure 3: Constraints on Sudoku

Now define clauses that enforce the constraints of the game. First of all, there must be at least one value in each entry for the puzzle to be fully filled out. For any given (x, y) combination that specifies one entry's location, at least one of the n^2 literals must be assigned true:

$$\bigwedge_{x=1}^{n^2} \bigwedge_{y=1}^{n^2} \bigvee_{z=1}^{n^2} s_{xyz} \tag{1}$$

Each value must appear at most once in each row. To accomplish this: for each row, create clauses from each possible pair of literals, and make the literals such that they can only be satisfied if at least one of the two values is false. For example, if the same value z showed up twice some row x, there needs to be a clause $(\neg s_{xy_1z} \vee \neg s_{xy_2z})$ to invalidate this assignment as a possible solution. A more generalized representation:

$$\bigwedge_{x=1}^{n^2} \bigwedge_{y=1}^{n^2-1} \bigwedge_{z=1}^{n^2} \bigwedge_{i=y+1}^{n^2} (\neg s_{xyz} \vee \neg s_{xiz}) \tag{2}$$

Similarly, each value must appear at most once in each column:

$$\bigwedge_{x=1}^{n^2-1} \bigwedge_{y=1}^{n^2} \bigwedge_{z=1}^{n^2} \bigwedge_{i=x+1}^{n^2} (\neg s_{xyz} \vee \neg s_{iyz}) \tag{3}$$

And lastly, create clauses which enforce that each value appears at most once in each sub-grid. First compare each pair of literals that are in the same sub-grid and row:

$$\bigwedge_{z=1}^{n^2}\bigwedge_{i=0}^{n-1}\bigwedge_{j=0}^{n-1}\bigwedge_{x=1}^{n}\bigwedge_{y=1}^{n}\bigwedge_{k=y+1}^{n}\left(\neg s_{(3i+x)(3j+y)(z)} \vee \neg s_{(3i+x)(3j+k)(z)}\right) \tag{4}$$

Then compare each pair of literals in the same sub-grid and column, as well as in the same sub-grid but in a different row and column:

$$\bigwedge_{z=1}^{n^2}\bigwedge_{i=0}^{n-1}\bigwedge_{j=0}^{n-1}\bigwedge_{x=1}^{n}\bigwedge_{y=1}^{n}\bigwedge_{k=x+1}^{n}\bigwedge_{l=1}^{n}\left(\neg s_{(3i+x)(3j+y)(z)} \vee \neg s_{(3i+k)(3j+l)(z)}\right) \tag{5}$$

The reason we don't need to specify that there must be at most one value in each entry is that the row, column, and sub-grid encoding clauses inherently already do this. There cannot be more than one value in an entry without it causing unsatisfiability in that row and column where the doubled value occurs.

Similarly, we don't need to specify that each value must appear at least once in each row, column, and sub-grid because we have already ensured that the whole puzzle must be filled out with exactly one value in each entry. If there is exactly one value in each entry and the puzzle does not contain any duplicate values within each row, column, and sub-grid, then it has already implicitly encoded that each possible value must appear exactly once in each row, column, and sub-grid, fulfilling the requirements for a valid solution.

3.2 Implementation

GitHub Repo: https://github.com/liloheinrich/Sudoku

```
python sudoku.py puzzles/puzzle8.txt
8     | 1     |   7
  2   |   4   | 8
  6   | 7     |
--------------------
      | 4 7   | 9   8
2 4   |   8   |
  3 8 |       |     5
--------------------
  8   | 6   4 | 1
9     |     7 | 2   4
    5 | 8 1   |     6

Solvable? True

8 9 4 | 1 3 5 | 6 7 2
5 2 7 | 9 4 6 | 8 3 1
1 6 3 | 7 2 8 | 5 4 9
--------------------
6 5 1 | 4 7 3 | 9 2 8
2 4 9 | 5 8 1 | 7 6 3
7 3 8 | 2 6 9 | 4 1 5
--------------------
3 8 2 | 6 9 4 | 1 5 7
9 1 6 | 3 5 7 | 2 8 4
4 7 5 | 8 1 2 | 3 9 6
```

Figure 4: Screenshot of Sudoku Solver in action

Our Sudoku Solver takes in a text file containing the size of the puzzle and a dictionary of the Sudoku clues. From that, it uses the encoding scheme above to construct a set of clauses in Conjunctive Normal Form by mapping each literal s_{xyz} to a corresponding unique integer identifier. It then runs the clauses through the DPLL SAT solver, and converts it back into a readable Sudoku solution.

The solver finds valid solutions for the given puzzle, but the more difficult the puzzle is, the longer it takes. On 6 out of 8 of our test puzzles (including three "medium" rated 9-by-9 puzzles), it is able to find the correct solution in less than ten seconds. Due to inefficiency in the implementation of the SAT solver, it is unable to solve the two "hard" puzzles in a timely manner.

To improve our implementation, we could work on reducing the SAT solver runtime, or use some other existing SAT Solver from the internet since it is likely possible to switch out the solver program with minimal modifications. Overall, we are satisfied with this implementation, as it is easy to use, accurate, and applicable to many of standard Sudoku puzzles.

4 Appendices

4.1 Appendix A: Reduction from TTP to LSC

This proof assumes familiarity with the latin square terminology and graph terminology defined above.

Part 1: LSC is in NP

Any potential solution to an n by n latin square is verifiable in polynomial time by the following algorithm:

```
s = potential_solution()
for row in s:
    if has_duplicates(row): return false  # O(n)
for column in s:
    if has_duplicates(column): return false # O(n)
return true

def has_duplicates(l):
    seen = set()
    for e in l:
        if e in seen: return true
        seen.add(e)
    return false
```

Part 2: LSC is NP-hard
We begin our proof by showing an equivalence between TTP and LSC (Latin Square Completion).

We show this by defining a construction from any partial latin square of size n to a tripartite graph D, called a defect graph. The vertices in this graph include:

- All rows that contain at least one empty square
- All columns that contain at least one empty square
- All numbers that appear less than n times in the partial latin square

The edges of this graph include:

- $\{r, c\}$, connecting the row r and column c of any empty square
- $\{e, c\}$, connecting the any column c to any of its missing elements e

- $\{e, r\}$, connecting the any row r to any of its missing elements e

Clearly, D is tripartite; D can be divided into rows, columns, and elements, and there will be no edges within each set. Additionally, the edges of D can be partitioned into triangles if and only if the latin square it represents is solvable. Each triangle $\{e, r, c\}$ represents the assignment of an element e to the box $\{r, c\}$. If any edges in D remain after all assignments are made, then one of the three conditions that define edges above is left unsatisfied, and the latin square is incomplete. An example of this transformation is shown below.

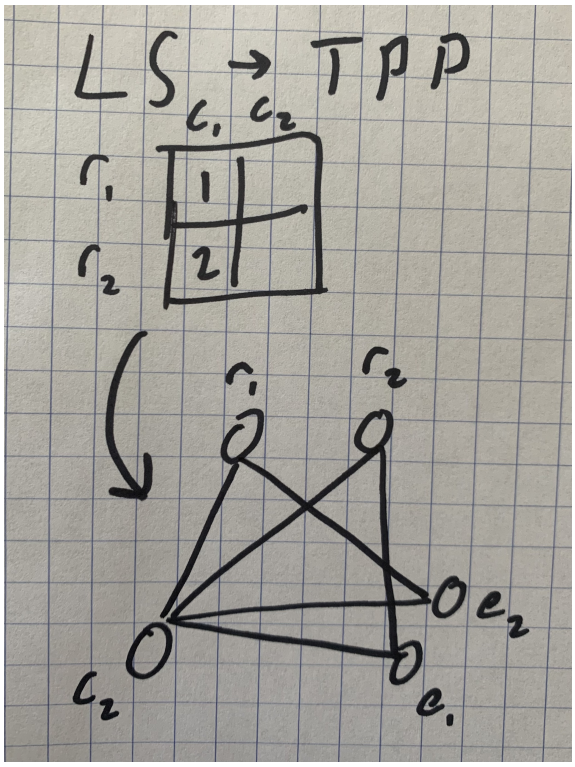

Figure 5: Transforming a latin square to a defect graph.

We must now show that any TPP can be represented as a latin square. Let us divide all tripartite graphs into two categories:

- **non-uniform tripartite graphs:** Consider a tripartite graph divided into sets V_1, V_2, and V_3 such that each contains no internal edges. A tripartite graph is uniform if every vertex (say, v in V_1) contains an equal number of neighbors in both remaining sections of the graph (say, V_2 and V_3). Equivalently, because tripartite graphs are 3-colorable, each red vertex must have an equal number of blue and green neighbors (and blue vertices must have equal numbers of red and green neighbors, and same for green).

  If a tripartite graph is non-uniform, then its edges cannot be partitioned into triangles. Consider vertex v_1 in V_1 with n neighbors in V_2 and m neighbors in V_3 ($n \geq m$). When drawing triangles containing v_1, edges drawn to V_3 will run out before edges drawn to V_2, and at least one edge between V_1 and V_2 will be left without a triangle. This means that the graph cannot be partitioned into triangles. Without loss of generality, we can say that no non-uniform tripartite graph has edges that can be partitioned into triangles. Because checking for uniformity takes at most $O(v * e)$ time, this case is in P. An example is shown below.

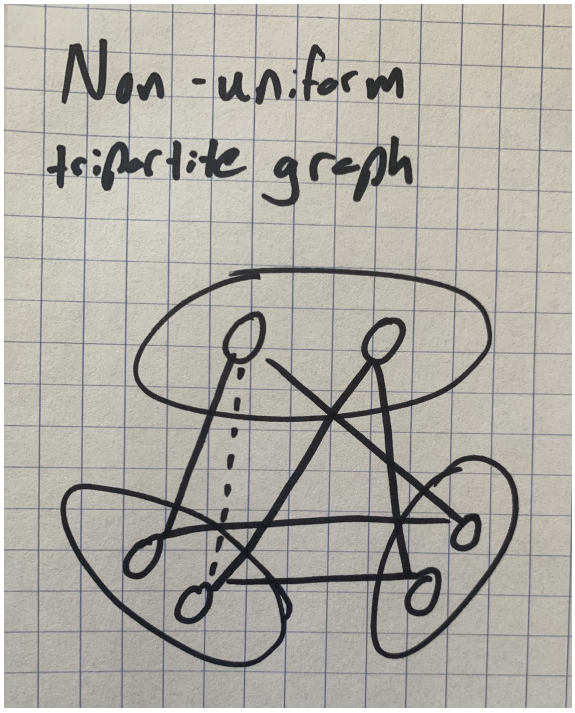

Figure 6: Example of a nonuniform tripartite graph. The dotted edge cannot be included in any triangle; it is a leftover.

- **uniform tripartite graphs:** Colbourn documents a procedure for this reduction in detail, but I will provide an overview of the steps here for completeness. Let's begin by defining our gadget, the latin framework. A latin framework LF(G, r, s, t) represents a tripartite graph G, and is an r by s array, where each entry is either empty or in $\{1, 2, ...t\}$. Note that if r=s=t, then the framework is a partial latin square. We must therefore describe a procedure by which any (uniform) tripartite graph can be transformed into a latin framework with r=s=t.

  1. Given a graph G with n vertices, construct a latin framework LF(G, n, n, 2n). To do this, we first label all vertices of G arbitrarily. Vertices in V_1 are named $\{r_1, r_2, ...r_{|V_1|}\}$. Vertices in V_2 and V_3 are named similarly, with c and e, respectively.

     Next, we draw our n by n latin framework, and fill it according to this algorithm

```
for i in range(n):
    for j in range(n):
        if G.has_edge(r[i], c[j]):
            leave LF[i][j] empty
        else:
            LF[i][j] = 1 + n + ((i + j) mod n)
```

     Notice that LF is a latin square with twice as many symbols as is allowed (symbols $1 - n$ are not used yet). An example is shown below.

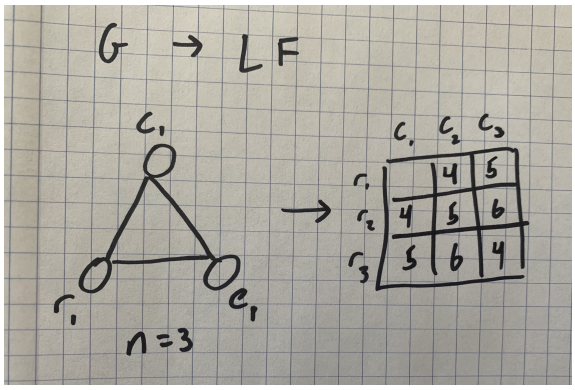

Figure 7: Transforming a tripartite graph G with n vertices to latin framework $LF(G, n, n, 2n)$

2. Now, we have a latin square with n rows, n columns, and $2n$ possible numbers. In order to grow this square, we follow this procedure (by Colbourn):

   "Let L be an LF(G, r, s, t) for a uniform tripartite graph G. Denote by R(k) the number of times element k appears in L plus half the degree of e_k in G. Then, whenever $R(K) \geq r + s - t$ for all $1 \leq k \leq t$, L can be extended to an LF(G, r, s+1, t) L' in which $R'(k) \geq r + (s + 1) - t$ for all $1 \leq k \leq t$."

   He goes onto describe the process for doing this; reference his proof for details. For the purposes of this appendix, we will note that this procedure allows us to grow our latin framework to size LF(G, n, 2n, 2n). Note that this is now a rectangle, not a square.

3. The same procedure as above can be used to grow the rectangle in the other direction. Instead of incrementing s by 1, we increment r by 1. This allows us to find a latin framework LF(G, 2n, 2n, 2n) that represents our initial graph G. Solving this latin square gives us a triangular partition for G, and if the latin square is not solvable, then G's edges cannot be partitioned in to triangles.

4.2 Appendix B: Reduction from LSC to SPC

To begin, we must formally define the SPC and LSC problems. As we have already proven that LSC is NP-Complete, and we have proven that it is possible to solve SPC problem in NP time, the only step that remains is to prove that LSC reduces to SPC. In other words, we must prove that given a Latin Square that is of order n, a Sudoku puzzle can be constructed such that the Sudoku puzzle is only solvable if the Latin Square is.

- **Sudoku Puzzle Completion**: Given an order, n, a matrix $S \in \mathbb{R}^{n^2 x n^2}$ and a set H of coordinates (i, j) such that $S(i, j)$ is within $[1, n^2]$ and the elements H^C are left blank, is there a way to assign the labels 1 to n^2 to the elements H^C such that the same label appears exactly once in each sub-line?
- **Latin Square Completion**: Given an order, n, a matrix $L \in \mathbb{R}^{nxn}$ and a set H of coordinates (i, j) such that $L(i, j)$ is within $[1, n]$ and the elements H^C are left blank, is there a way to assign the labels 1 to n to the elements H^C such that the same label appears exactly once in each row and column?

Next, we define our LSP and corresponding SCP for the reduction.

1. We define an instance of the LSP, $I = \langle n, L, H \rangle$ where L is an nxn matrix and H as the set of coordinates $\{(i, j)|i, j, L(i, j) \in [1, n]\}$. All the squares that are not in H are blanks, therefore H^C represents the set of all the remaining empty squares, $\{(i, j)|i, j \in [1, n], L(i, j) = \omega$, where ω represents an empty square.

2. We define a solution to the Sudoku puzzle, S of order n as described at the beginning of this appendix. We can formally define this solution as follows:

$$S(i,j) = ((j-1+((i-1) \mod n)\cdot n + \lfloor\frac{i-1}{n}\rfloor) \mod n^2) + 1$$

where $j-1$ enforces the rule that the labels are in order for each row, $((i-1) \mod n)\cdot n$ ensures that the rows within a band are shifted n squares to the left. Note that the fraction $\frac{i-1}{n}$ is floored (rounded down to the nearest integer), ensuring that the first row in each band is equal to the first row of the previous band shifted one square to the left. Figure 3 is a visual representation of the Sudoku puzzle that we just defined. You can see that it is indeed a valid Sudoku puzzle.

3. Define a set B that contains the squares that make up a Latin Square:

$$B = \{(i,j) | i \in [1,n], (j-1) \mod n = 0\}$$

which contains the intersection between the first band ($i \in [1,n]$) and the first column of each stack ($(j-1) \mod n = 0$). These intersections are illustrated in the figure below.

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

Figure 8: Intersection between the first band and the first column of each stack in a valid 3x3 Sudoku puzzle. source

4. To complete the transformation, we define an instance of SPC, $I' = \langle n, S, H' \rangle$, such that:
   - The order n of I', the SPC problem, is equal to the order n for I, the LSC.
   - $H' = B^C \cup \{(i, 1+(j-1)n) | (i,j) \in H\}$. This ensures that the set of filled squares within the Sudoku puzzle is equal to the union between the empty squares within the LSC and the set of coordinates $(i, 1+(j-1)n)$ for every coordinate in the LSC.
   - The full definition of our SPC, with two additional constraints to make it suitable for solving the LSC:

$$S(i,j) = \begin{cases} ((j-1+((i-1) \mod n)\cdot n + \lfloor\frac{i-1}{n}\rfloor) \mod n^2) + 1 & \text{if } (i,j) \notin B \\ 1 + (L(i, \frac{j-1}{n} + 1) - 1)n & \text{if } (i,j) \in B \cup H' \\ \omega & \text{if } (i,j) \in (H')^C \end{cases}$$

The second case within $S(i,j)$ adjusts the labels L to fit inside B within our Sudoku, S, while the third case ensures that squares which are left blank within L and are left blank is well (source). Values not in B are left unchanged from the original solution.

Now we are finally ready to move on to the reduction. As a reminder, we are proving that the instance I of the LSC problem is solvable if and only if the instance I' of the SPC is also solvable. Due to the verbosity of this proof, some intermediate steps are skipped or summarized. If you wish to view the proof in its entirety, you may reference here. We will prove that the configuration of the SPC that we defined such that it solves the given LSC problem, upholds each of the three rules of the Sudoku game. Validity of the configuration will be proven from left to right, and from right to left.

4.2.1 Proof: Given I prove I' is a valid solution

1. **Proof of the row rule**: We start by assuming I to be solvable, so we must only show that the instance I' of the SCP is solvable. We then define L' to be an nxn-dimensional matrix to be a valid solution to I. We can then define our proposed solution S' to be equivalent to S, with no empty squares remaining (i.e. $(H')^C$ is empty).

$$S'(i,j) = \begin{cases} ((j + ((i-1) \mod n) \cdot n + \lfloor \frac{i-1}{n} \rfloor) \mod n^2) + 1 & \text{if } (i,j) \notin B \\ 1 + (L'(i, \frac{j-1}{n} + 1) - 1)n & \text{if } (i,j) \in B \end{cases}$$

   For this to be a valid solution, labels must be within $[1, n^2]$ and each label should only occur once within any given subline. This can be proven by noting that $L'(i,j)$ must be within 1 and n, as L' is a solution of order n, and therefore $1 \leq (L'(i,j) - 1)n \leq 1 + (n-1)n \leq n^2$, so $S'(i,j) = S(i,j) \in [1, n^2]$ for all $i, j \in B$. These observations also prove that sub-lines not containing elements of B do not need to be considered in this proof.

   We can then assume that we are only concerned with the top band. If we hold i fixed, we then take $j, k \in [1, n^2]$ where $j \neq k$. We then walk through all 3 possible cases:

   - $(i,j), (i,k) \in B$. By definition all values not in B provide a valid solution, so this case is easy.
   - $(i,j), (i,k) \notin B$. Note that $S(i,j) \neq S(i,k)$, due to the fact that if they were indeed equal, then a number from $[1, n^2]$ would have to appear twice in a given column, which is against the rules of the game. Therefore, we can state that:

$$S(i,j) = 1 + L'(i, \frac{j-1}{n} + 1) - 1)n \neq 1 + L'(i, \frac{k-1}{n} + 1) - 1)n = S'(i,k)$$

   - $(i,j) \in B$ and $(i,k) \notin B$. In this case, we can show that:

$$S'(i,j) = 1 + L'(i, \frac{j-1}{n} + 1) - 1)n \text{ and } S'(i,k) = ((j + ((i-1) \mod n) \cdot n + \lfloor \frac{i-1}{n} \rfloor) \mod n^2) + 1$$

   Lastly, we prove the row rule by contradicting the case where $S(i,j) = S(i,k)$. Based on the above equations, we note that $S'(i,j) = 1$, and then imply (for the sake of our contradiction), that $S(i,k) = 1$. For the actual contradiction, please view page 13 of this reference. In these steps, shows that if we try setting $S(i,k) = 1$ and simplifying, we find that $k - 1 \mod n = 0$, so a contradiction is raised, therefore proving that the row rule is satisfied.

2. **Proof of the column rule** For this part, only the first columns of each stack are considered, therefore $(j-1) \mod n = 0$. Similarly to the row rule we just discussed, the column rule states that given $(i,k) \in [1, n^2]$ s.t. $i \neq k$, then $S(i,j) \neq S(k,j)$. We again go through the three possible cases:

   - $(i,j), (k,j) \in B$. Again, all values not in B provide a valid solution, so this case is easy.
   - $(i,j), (k,j) \notin B$, then the corresponding squares in L' are $(i, \frac{j-1}{n} + 1), (i, \frac{k-1}{n} + 1)$. Therefore, we show that:

$$S(i,j) = 1 + L'(i, \frac{j-1}{n} + 1) - 1)n \neq 1 + L'(k, \frac{j-1}{n} + 1) - 1)n = S'(k,j)$$

   - $(i,j) \in B$ and $(i,k) \notin B$. In this case, $i \in [1, n]$ and $k \in [n+1, n^2]$.

Again, we prove by contradiction that it is not possible for $S(i,j) = S(k,j)$ by noticing that, according to the above formulations, $S'(i,j) \mod n = 1$. If we try setting $S'(k,j) \mod n = 1$ and simplifying, we find that there exists an assignment that leads to $\lfloor \frac{k-1}{n} \rfloor \mod n = 0$. However, since $k \in [n+1, n^2], 1 \leq \lfloor \frac{k-1}{n} \rfloor \leq n-1$, this is a contradiction since $\lfloor \frac{k-1}{n} \rfloor \mod n \neq 0$. This proves that the column rule is satisfied.

3. **Proof of the subgrid rule** We need only consider the top band of the Sudoku puzzle for this proof, so $i, k \in [1, n]$. In any given subgrid, $S'(i,j) \neq S'(k,l)$ for all $(i,j), (k,l)$ within a subgrid, where $i \neq k, j \neq l$. We then examine the possible cases, as follows:

   - $(i,j), (k,l) \notin B$. Since $j \neq l$, $(i,j), (k,l)$ must be in different columns, but since they are within the same subgrid, only one of them can be in B.
   - $(i,j) \notin B$ and $(k,l) \in B$. This means that $S'(k,l) \mod n = 1$

4.2.2 Proof: Given I' prove I is a valid solution

If S' is an n^2xn^2-dimensional matrix solution to the instance I' of the SPC, then a solution L' to the LSC problem by rewriting the formulation of S' in the previous section of the proof. Squares of Sudoku that are not in B are not considered, since they do not affect the solution L'. Therefore, $S(i,j) = 1 + (L'(i, \frac{j-1}{n} + 1) - 1)n$, rewritten as:

$$L'(i,j) = 1 + \frac{S'(i,j') - 1}{n}$$

where $j' = 1 + (j-1)n - 1$. We then assume that $i \in [1, n]$, so only the squares in B are considered. The coordinates in (i, j') are in B such that

$$(j' - 1) \mod n = ((1 + (j-1)n - 1) \mod n = ((j-1)n \mod n) = 0$$

In order for L' to be a valid solution, the labels can only take on values from 1 to n and each value can only appear once in any given row or column. This can be proven by noting that $(i, j') \in B$ and therefore $S'(i, j') \mod n = 1$. This information can be used to then determine that $L'(i,j) = 1 + \frac{S'(i,j')-1}{n}$ is restricted to only integer values. These two observations along with the fact that $S(i, j') \in [1, n^2]$ (since S' is a solution to I can be used to show that the labels in L' are in $[1, n]$:

$$S'(i,j') \in [1, 1 + (n-1)n] \Rightarrow L'(i,j) = 1 + \frac{S'(i,j') - 1}{n} \in [1 + \frac{1-1}{n}, 1 + \frac{1 + (n-1)n - 1}{n}] = [1, n]$$

Lastly, we prove the same row/column rules in S':

$$j \neq k \Rightarrow j' \neq k' \Rightarrow S'(i,j') \neq S'(i,k') \Rightarrow 1 + \frac{S(i,j') - 1}{n} \neq 1 + \frac{S(i,k') - 1}{n} \Rightarrow L'(i,j) \neq L'(i,k)$$

$$i \neq k \Rightarrow S'(i,j') \neq S'(k,j') \Rightarrow 1 + \frac{S(i,j') - 1}{n} \neq 1 + \frac{S(k,j') - 1}{n} \Rightarrow L'(i,j) \neq L'(k,j)$$

Thus, L is a valid solution to I, which proves our original claim. This reduction can be performed in polynomial time. Given an instance of the input size is polynomially related to the order, n. Creating an n^2xn^2 matrix is $O(n^4)$, and labels are assigned according to $S(i,j)O(n^4)$, so the overall runtime of the reduction is $O(n^4)$ so we have proved that a reduction exists from a known NP-Complete problem to Sudoku in polynomial time. Combining this information with our proof that Sudoku is NP proves that Sudoku is indeed NP-Complete.

Resources

We chose to provide inline citations throughout the document as necessary.