Revisiting the proof of the complexity of the sudoku puzzle

Eline Sophie Hoexum

June 2020

First supervisor:
Prof. dr. L.C. Verbrugge

Second supervisor:
Prof. dr. J. Top

Abstract

In 2003 it was shown by Takayuki Yato that the sudoku puzzle is an NP-complete and ASP-complete problem. The proof provided in this paper involved a reduction from the Latin Square Completion problem, which is quite similar to a sudoku puzzle. However, as the paper was not solely focused on the sudoku puzzle, the formulation of the proof was short and not very intuitive. This paper will describe the method of proof in more detail and give more context, as to make it more understandable. Additionally, because there has not been much research done into other ways to prove the NP- and ASP-completeness of the sudoku puzzle, a different approach to the proof is explored. After this, the solving procedure is explained and the performances of different solving algorithms are compared to determine which is the most efficient method for solving a sudoku puzzle.

Keywords: sudoku puzzle, complexity, NP-completeness, ASP-completeness, solving algorithm

Science & Engineering
Rijksuniversiteit Groningen
The Netherlands

Contents

1 Introduction

The sudoku puzzle is a logic puzzle popular all around the world, with hundreds of puzzles being printed in newspaper puzzle pages every day. The goal of the puzzle is to complete a square grid, usually 9 by 9, with the integers 1 to 9 such that each integer only appears once in each row, column, and 3 by 3 sub-block. Even though no actual mathematics is required to solve a sudoku puzzle, it can be seen as a mathematical problem. Because of this, its complexity has been studied by many mathematicians.

The field of complexity is a subject within computing science, which concerns itself with how fast or efficiently a problem can be solved with an algorithm. In this field, the set of P-problems is defined such that it contains all the problems that can be solved with a deterministic algorithm in polynomial time. This set is a subset of the set of NP-problems, which consists of all the problems that can be solved by guessing a candidate solution and then checking if it is valid in polynomial time. In 1971, Stephen Cook proved in [3] that determining if a given Boolean formula is satisfiable is at least as hard as every other problem in NP. This discovery created a new category, the set of NP-complete problems. These are the problems in NP that every other problem in NP can be reduced to. This implies that, given a problem Π in NP, if there exists an NP-complete problem that can be reduced to Π in polynomial time, then Π is NP-complete. For this reason, NP-complete problems are seen as the hardest problems in NP.

For any problem Π, there is also an associated Another Solution Problem, often referred to as ASP-Π. This problem is concerned with finding another solution S' to Π when a solution S is already given. Besides being NP-complete, a problem can also be ASP-complete, meaning its ASP is an NP-complete problem.

The complexity of the sudoku puzzle has already been studied a lot and the most significant result until now is that finding a solution to a puzzle is both NP-complete and ASP-complete. This was shown in 2003 by Yato in [11] and since then the result has been used in many research papers. However, the proof provided in his paper is very theory-heavy and hard to understand on the first few readthroughs. Additionally, it seems like there has not been much further research done into the way the NP-completeness of the sudoku puzzle can be proven.

The goal of this paper will be to dive deeper into the proof of the NP-completeness of the sudoku puzzle. In Chapter 2, the sudoku puzzle is defined, along with the set of all sudoku solutions. Then some transformations are defined that can be performed on these solutions in order to show that certain solutions are transformations of each other and thus have a very similar solving procedure.

In Chapter 3, the subject of complexity is described in detail, specifically the categories of NP and P. Then, the Boolean satisfiability problem (SAT) and a similar problem, called 3SAT, are defined and it is shown that 3SAT is an NP-complete problem. Then, an attempt is made to reduce 3SAT to the sudoku puzzle in order to show it is NP-complete as well. After this, a more detailed description of the original proof by Yato from [11] will be given, such that it may be easier to understand how it works.

Next, in Chapter 4, the uniqueness of a sudoku puzzle solution is studied. This is done using the Another Solution Problem (ASP) of the sudoku puzzle. It is shown how the original proof shows ASP-completeness as well as NP-completeness, as the reduction used is parsimonious. After this, the attempted proof from the previous chapter is expanded to also become parsimonious in order to show ASP-completeness.

Finally, in Chapter 5, the solving procedure of the sudoku puzzle is explained step by step. After this, the programming methods of backtracking, stochastic search, and constraint programming, which can be used to solve a sudoku puzzle algorithmically are explained. These methods are then each tested on a set of test puzzles and their results are compared.

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

Figure 1: A sudoku puzzle

2 The Sudoku Puzzle

A *sudoku puzzle of order* n is an $n^2 \times n^2$ grid, with some grid squares (the *hints*) filled in with one of the *labels*, the integers from 1 to n^2. A *solution* to such a puzzle is the same grid, but with each empty square filled in with one of the labels, such that each label only occurs exactly once in each row, column, and $n \times n$ sub-block. These groups, which contain only one of each label, are referred to as *sub-lines*. A row of n sub-blocks next to each other is called a *band* and a column of n sub-blocks on top of each other is a *stack*.

Usually, a sudoku puzzle will have one unique solution. If this weren't the case, it wouldn't be possible for a human to solve the puzzle. However, in the definition of a sudoku puzzle this paper uses, this is not a requirement. Most sudoku puzzles are of order 3, meaning they are a 9×9 grid. An example of such a sudoku puzzle is given in Figure 1.

2.1 The Mathematics of the Sudoku Puzzle

A sudoku puzzle of order 3 can be represented mathematically by equating the grid to a matrix in $\mathbb{R}^{9\times 9}$. By setting the following restrictions on this set, the set $\mathcal{S}$ of all the possible solutions S is obtained. In the rest of this paper, the set of integers between n and m, $\{n, \dots, m\}$, will be indicated by $[n, m]$. This set usually contains the non-integers in this range as well, but in this paper these values will usually not be necessary, so $[n, m]$ will only contain the integers in this range.

- Each element of the matrix is one of the labels 1 to 9, so $S(i, j) \in [1, 9]$ for all $i, j \in [1, 9]$.
- Each label may occur only once in each column, so $S(i, j) \neq S(k, j)$ for all $i, j, k \in [1, 9]$ s.t. $i \neq k$.
- Each label may occur only once in each row, so $S(i, j) \neq S(i, l)$ for all $i, j, l \in [1, 9]$ s.t. $j \neq l$.
- Each label may occur only once in each of the 3×3 sub-blocks. To make the notation somewhat easier, the sets D_1, D_2 and D_3 are defined to be equal to $[1, 3], [4, 6]$ and $[7, 9]$ respectively. This means that each sub-block can be written as $D_n \times D_m$ for some $m, n \in [1, 3]$. If there is a bijection between a sub-block $D_n \times D_m$ and the set $[1, 9]$, then these sets have the same number of elements, so therefore each element of $D_n \times D_m$ will be distinct. If such a bijection exists for each sub-block, then the sub-block rule is satisfied.

These restrictions are combined into the following set.

$$\begin{aligned}\mathcal{S} = \{S \in \mathbb{R}^{9\times 9} \mid\ & S(i, j) \in [1, 9],\ S(i, j) \neq S(k, j)\ \forall i, j, k \in [1, 9] \text{ s.t. } i \neq k,\ S(i, j) \neq S(i, l)\ \forall i, j, l \in [1, 9] \text{ s.t. } j \neq l, \\ & \text{there exists a bijection from } D_n \times D_m \text{ to } [1, 9]\ \forall m, n \in [1, 3]\}\end{aligned}$$

This set contains all possible solutions and must be a finite set, as there are not infinitely many possible ways to assign digits to the grid squares and adding restrictions will only make the set smaller.

2.2 Equivalent sudoku puzzles

In [5], it was shown that there are 6.670.903.752.021.072.936.960 possible sudoku solutions. However, many of these solutions have no significant differences. For example, if one were to change the order

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

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

Figure 2: The same puzzle with the order of the labels 1-9 changed

of the labels 1 to 9, the result would be a different solution, but the order of the labels doesn't affect the solving process, so any sudoku puzzle with such a solution would be solved in the same way. In Figure 2, an example of this is given. The grid on the right is the same as the one on the left, but each label has been interchanged for the next one and 9 has been replaced by 1. Similarly, if two rows or columns are swapped within a band or stack or two bands or stacks are swapped altogether, the method of solving any puzzle with those solutions would essentially not become any different. Solutions like these are called *equivalent.* If two solutions are not equivalent they are *essentially different.* The complete list of transformations that preserve equivalence is as follows.

1. Changing the order of (permuting) the labels 1 through 9
2. Swapping rows in a band.
3. Swapping columns in a stack.
4. Swapping bands.
5. Swapping stacks.
6. Mirroring, both horizontally and vertically. These transformations can also be accomplished by swapping rows/columns and bands/stacks.
7. Transposition. This transformation mirrors the sudoku through the diagonal from the upper left to the lower right, the *downward diagonal.* A transformation that could also be considered is a transposition through the *upward diagonal*, but this can also be achieved by horizontally and vertically mirroring the transposition through the downward diagonal.
8. Rotations. There are four orientations for each sudoku, the original and rotations of 90, 180, and 270 degrees. It turns out that these transformations can also be accomplished by a combination of mirroring and transposing.

These transformations can be described as functions. First, there are the swap functions, which correspond to items 1 through 5 in the list.

<u>Definition</u>: For $u, v \in [1,9]$, the *label swap function* $l_{uv} : \mathcal{S} \to l_{uv}(\mathcal{S})$ is defined such that

$$(l_{uv}(S))(i,j) = \begin{cases} u & \text{if } S(i,j) = v \\ v & \text{if } S(i,j) = u \\ S(i,j) & \text{otherwise} \end{cases}$$

<u>Definition</u>: For $u, v \in D_{n'}$, for some $n' \in [1,3]$,

- ...the *column swap function* $c_{uv} : \mathcal{S} \to c_{uv}(\mathcal{S})$ is defined such that

$$(c_{uv}(S))(i,j) = \begin{cases} S(i,v) & \text{if } j = u \\ S(i,u) & \text{if } j = v \\ S(i,j) & \text{otherwise} \end{cases}$$

- …the *row swap function* $r_{uv} : \mathcal{S} \to r_{uv}(\mathcal{S})$ is defined such that

$$(r_{uv}(S))(i,j) = \begin{cases} S(v,j) & \text{if } i = u \\ S(u,j) & \text{if } i = v \\ S(i,j) & \text{otherwise} \end{cases}$$

Definition: For $u, v \in D_1$ such that $u < v$,

- …the *band swap function* $b_{uv} : \mathcal{S} \to b_{uv}(\mathcal{S})$ is defined such that

$$(b_{uv}(S))(i,j) = \begin{cases} S(i,j-3(v-u)) & \text{if } j \in D_v \\ S(i,j+3(v-u)) & \text{if } i \in D_u \\ S(i,j) & \text{otherwise} \end{cases}$$

- …the *stack swap function* $s_{uv} : \mathcal{S} \to s_{uv}(\mathcal{S})$ is defined such that

$$(s_{uv}(S))(i,j) = \begin{cases} S(i-3(v-u),j) & \text{if } i \in D_{\max\{u,v\}} \\ S(i+3(v-u),j) & \text{if } i \in D_{\min\{u,v\}} \\ S(i,j) & \text{otherwise} \end{cases}$$

In all of the above definitions, it is assumed that $u \neq v$, otherwise the functions would simply be the identity map I. Next, maps for mirroring, transposition, and rotation are defined.

Definition: The *horizontal mirror function* $H : \mathcal{S} \to H(\mathcal{S})$ is defined s.t. $(H(S))(i,j) = S(10-i,j)$.

Definition: The *vertical mirror function* $V : \mathcal{S} \to V(\mathcal{S})$ is defined s.t. $(V(S))(i,j) = S(i,10-j)$.

Definition: The *downward transposition function* $T_d : \mathcal{S} \to T_d(\mathcal{S})$ is defined s.t. $(T_d(S))(i,j) = S(j,i)$.

Definition: The *upward transposition function* $T_u : \mathcal{S} \to T_u(\mathcal{S})$ is defined such that

$$(T_u(S))(i,j) = S(10-i,10-j)$$

Definition: The *quarter-rotation function* $R : \mathcal{S} \to R(\mathcal{S})$ is defined s.t. $(R(S))(i,j) = S(j,10-i)$.

Except for quarter- rotation, all the defined maps are their own inverses. For quarter-rotation, an inverse can easily be created, namely R^{-1} such that $(R^{-1}(S))(i,j) = S(10-j,i)$. As the functions have inverses, they are bijective, which means that their range is equal to $\mathcal{S}$. Therefore, any matrix these functions produce is a valid sudoku solution.

Four of the last five functions can also be performed by using combinations of the first five transformations and downward transposition. Firstly, the mirror functions can be performed by swapping rows/columns and bands/stacks. This is intuitive and is not proven here, but it means that

$$H(S) = r_{89}(r_{78}(r_{12}(r_{23}(r_{46}(b_{13}(S)))))) \quad \text{and} \quad V(S) = c_{89}(c_{78}(c_{12}(c_{23}(c_{46}(s_{13}(S))))))$$

Similarly, the upward transposition can be performed by mirroring the downward transposition horizontally and vertically. In other words, $T_u(S) = V(H(T_d(S)))$. This holds, as

$$(V(H(T_d(S))))(i,j) = (H(T_d(S)))(i,10-j) = (T_d(S))(10-i,10-j) = S(10-j,10-i) = (T_u(S))(i,j) \quad \forall i,j \in [1,9]$$

Lastly, the quarter-rotation function can be performed by first downward transposing and then mirroring horizontally. This means that $R(S) = H(T_d(S))$, which holds, because

$$(H(T_d(S)))(i,j) = (T_d(S))(10-i,j) = S(j,10-i) = (R(S))(i,j) \quad \forall i,j \in [1,9]$$

As each rotation can be achieved by applying the quarter-rotation function multiple times, it has now been shown that each rotation can be achieved by applying the other rules. By proving these three claims, it becomes clear that there are only six actual transformations to consider when trying to discover how many equivalent solutions there are.

1. *Relabelling.* Rearranging the order of the 9 labels creates groups of 9! equivalent solutions.
2. *Row/column swaps.* As there are three rows/columns in each band/stack, there are 3! ways to rearrange the rows in each band. Therefore, both of these transformations create groups of $(3!)^3$ equivalent solutions.
3. *Swapping bands/stacks.* As there are 3 bands/stacks, there are 3! ways to rearrange them, so both

of these transformations create groups of 3! equivalent solutions.

4. *Transposition.* This only creates one other solution, so it makes pairs of 2 equivalent solutions.

This reduces the number of essentially different solutions to 1.218.998.108.160. The actual number of essentially different solutions is lower, namely 5.472.730.538. This result was found in [10] by using group theory.

3 The existence problem

Proving that a given sudoku puzzle has a solution by simply looking at the given hints might seem like a useful and simple problem to tackle. However, from the definition of a puzzle used in this paper it is not given that the puzzle must be solvable, so to prove that there exists a solution, one would have to show that there are no squares where no label would fit according to the rules. The process of doing this essentially replicates the solving procedure a human solver would go through. This does not add much, as in the process of trying to prove that a solution exists, a solution would have already been found. Furthermore, multiple solutions may exist, which means that attempting to solve the sudoku will result in certain squares still having multiple possible values, so then this method delivers no result. This gives the impression that trying to prove that a sudoku puzzle has a solution is not a very interesting venture and would certainly not result in an easily checkable list of requirements. Instead, this chapter seeks to show the complexity of the Sudoku Puzzle Completion problem (SPC), which is defined as follows.

Given an order n, a matrix $S \in \mathbb{R}^{n^2 \times n^2}$ and a set H of coordinates (i,j) s.t. $S(i,j) \in [1,n^2]$ and the elements in H^C are left blank, is there a way to assign the labels 1 to n^2 to the elements in H^C such that the same label appears exactly once in each sub-line?

To find the complexity of this problem, the necessary theory will first be explained.

3.1 NP and P and the satisfiability problem

In the theory of computational complexity, decision problems are studied in order to classify their difficulty into different categories. These problems are questions that can be answered by "yes" or "no". The SPC as defined above is one of these decision problems. The relevant categories for the topic of the sudoku puzzle will be P and NP. The class P contains decision problems that can be solved in polynomial time by a deterministic Turing machine. A Turing Machine is a theoretical computer that can run algorithms to solve problems. It works with a string of tape squares, where a symbol could be written in each square. An algorithm runs reads the symbol a in the square it is currently on and, depending on the state q the Turing machine is in, determines which next state and tape square to move to. The machine has two special states, the accept state q_Y and the reject state q_N, which determine whether the input is accepted and therefore what the answer to the decision problem is. In a *deterministic Turing machine*, for every pair $(q$,$a)$, there is only one next step possible. This means that the algorithm simply follows the instructions it is given and therefore it could happen that it doesn't reach q_Y or q_N, but instead gets stuck in an infinite loop. Because of this, it seems logical that not all problems can be solved by a deterministic Turing machine. This means that the set of P-problems contains problems that could be called "easy". On the other hand, there is the class NP, which contains problems that can be solved in polynomial time by a *non-deterministic Turing machine*. In contrast to the deterministic machine, machines like these have a set of steps they can take for each pair (q,a). This means that it accepts an input if any of the possible consecutive step choices lead the program to q_Y. From this definition, it can be inferred that every problem in P is also in NP, as any program the deterministic Turing machine runs could also be run by a non-deterministic machine.

There is a subset of NP, which contains the "hardest" problems in NP. These problems are called NP-complete and their defining property is that each problem in NP can be reduced to them. The concept of reduction will be explained further on in this section. The first problem that was proven to be NP-complete was satisfiability (SAT). This problem begins with the set $U = \{u_1, \dots, u_m\}$, which contains m Boolean variables. A truth assignment over this set U is a function $t : U \to \{T, F\}$ that assigns a truth value to each variable. If $t(u) = T$, then the variable is true, if $t(u) = F$, then it's false. A literal is either a variable out of the set U, or the negation $\overline{u}$ for some $u \in U$, where $t(\overline{u}) = T$ iff $t(u) = F$. A clause is a set of literals from U, which represents a disjunction and is *satisfied* by a truth assignment t iff at least one of the literals in the set is true under that assignment. A collection C of clauses is satisfied by a truth assignment t if all the clauses in C are satisfied by t. This collection is a different way to represent a formula in CNF, as it is a conjunction of disjunctions. Using these definitions, the SAT problem is defined as follows.

Given a set of Boolean variables U and a collection C of clauses with literals from U, does there exist a truth assignment t such that C is satisfied?

In 1971, Cook proved that this problem is NP-complete, by showing that any problem in NP could be reduced to it. A problem Π_1 can be reduced to another problem Π_2 if there exists a transformation that transforms any input I for Π_1 into an input I' for Π_2, such that the answer that I' gives as an input for Π_2 is the same answer that I would give as an input for Π_1. It should be possible to perform this transformation in polynomial time. If Π_1 can be reduced to Π_2, then Π_2 is *at least as hard* as Π_1, so if it can be shown that an NP-complete problem reduces to another problem in NP, then that second problem must also be NP-complete. This will be the strategy to prove the complexity of the Sudoku Puzzle Completion problem.

3.2 A proposed reduction from 3SAT to an adjusted version of the SPC problem

It is generally known that the sudoku problem is NP-complete [11]. To prove this, the first step is to show that the problem is NP. This can be shown by proving that a solution to an instance of the problem can be guessed and checked in polynomial time. Given a sudoku puzzle of order n, an algorithm could guess the labels for the squares that have not been given, which would be at most n^4 steps. Then, to check whether it satisfies the sudoku rules, it would do n^2 checks for the rows, n^2 checks for the columns, and n^2 checks for the sub-blocks. This means that the algorithm would be at most $O(n^4)$, so a solution for an instance of the Sudoku Puzzle Completion problem can be checked in polynomial time. Therefore, the SPC problem is NP.

Next, to show that the SPC problem is NP-complete, SAT will be reduced to SPC. In [11] the SAT problem is reduced to the 3SAT problem, which is defined similarly to the SAT problem, except that the clauses may contain at most 3 literals each. As SAT is NP-complete, this would imply that 3SAT is also NP-complete. Because this paper uses different notations and definitions, the proof is replicated here.

Claim: 3SAT is NP-complete.

Proof. Given an instance of SAT, a collection C of clauses with literals from the set U, reducing it to 3SAT would mean breaking every clause with 4 literals up into clauses with no more than 3 literals. In [11], this is done two literals at a time. Given a clause with the literals $u_1, u_2, \dots, u_m$, a new variable d is introduced and the clause is split up into

$$u_1 \lor u_2 \lor \overline{d}, \qquad \overline{u_1} \lor d, \qquad \overline{u_2} \lor d \quad \text{and} \quad d \lor u_3 \lor \cdots \lor u_m.$$

The conjunction of the first three clauses is equivalent to $(u_1 \lor u_2) \equiv d$, ensuring the last clause has the same truth value as the starting clause. This step can be repeated as many times as needed until the last clause is reduced to 3 or fewer literals, which results in a new collection C' of clauses and a new set of variables U'. It is easily observable that this can be done in polynomial time, as for any clause with m literals, the step needs to be repeated no more than $\frac{1}{2}m$ times. Therefore, SAT can be reduced to 3SAT with a polynomial-time reduction and, because SAT is NP-complete, this proves that 3SAT is NP-complete. □

Now that this claim has been proven, reducing 3SAT to the Sudoku Puzzle Completion problem will be enough to prove the latter is NP-complete. To reduce 3SAT to the SPC problem, the possible clauses will be studied to find structures in a sudoku puzzle that could represent them. In 3SAT, the clauses could contain at most 3 literals, so there are three cases to consider, a clause of 1, 2, or 3 literals. Next, it is illustrated how these cases can be transformed into constructs placed in a sudoku puzzle solution of order 2 and, through these examples, it shall be made clear how they can be expanded to a sudoku puzzle of any size.

- A clause with 1 element, $\{u_1\}$. To deal with this case, a variable s will be defined as follows.

$$s = \begin{cases} 1 & \text{if } t(u_1) = T \\ 5 & \text{if } t(u_1) = F \end{cases}$$

  This way the variable will produce an invalid label for a sudoku of order 2 if the literal u_1 is assigned false by t. This means that the sudoku will not have a valid solution whenever the clause is not satisfied by t. This variable is placed in a solution of order 2 in a grid square which originally contained a 1, shown on the left in Figure 3, to make sure the rest of the sudoku is valid. If the literal is assigned true, then the sudoku will already be its own solution.

- A clause with 2 literals, $\{u_1, u_2\}$. Two new variables a and b are defined as follows.

$$a = \begin{cases} 1 & \text{if } t(u_1) = T \\ 3 & \text{if } t(u_1) = F \end{cases} \qquad b = \begin{cases} 2 & \text{if } t(u_2) = T \\ 3 & \text{if } t(u_2) = F \end{cases}$$

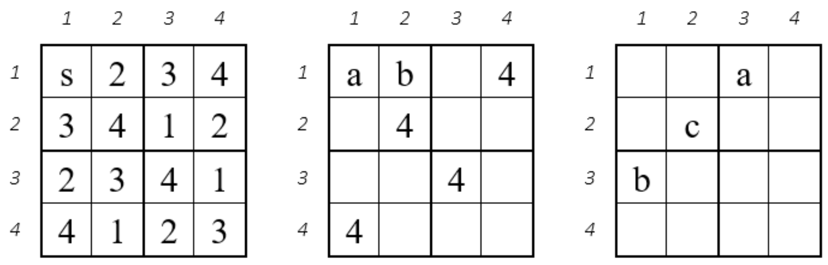


Figure 3: Constructs to represent clauses with one, two or three literals

This means that they will only be the same if u_1 and u_2 are both assigned false. To place them in a solution of order 2, every instance of the labels they can produce, namely $1, 2$ and 3, will need to be deleted. When the variables are then placed in adjacent squares as shown in the middle in Figure 3, they will create a contradiction with the sudoku rules if both are assigned false, which means that the sudoku won't be solvable. As each instance of the used labels was removed, the sudoku will be solvable as long as the two values are different and this is exactly the case whenever at least one of the literals is assigned true under t, so when the clause is satisfied by t.

- A clause with 3 literals, $\{u_1, u_2, u_3\}$. For this case, a new variable c is defined as:

$$c = \begin{cases} 3 & \text{if } t(u_3) = T \\ 4 & \text{if } t(u_3) = F \end{cases}$$

To place this variable along with a and b into a solution of order 2, all instances of the labels they use have to be removed. So, in this case, all the labels have to be removed, resulting in an empty grid. Then the variables are placed in the grid as shown on the right in Figure 3. This way, the sudoku has no contradictions as long as at least one of the variables is assigned true. If all are assigned false, there is no place for a 3 in the first sub-block. Therefore, this sudoku has a solution whenever the clause is satisfied.

Now, these methods work for the clauses individually, but for the transformation to work for a whole collection of clauses, it should be possible to combine the constructs into one sudoku grid. In Appendix A, pseudo-code is given for the full transformation from 3SAT to SPC. In short, the algorithm creates a solution of appropriate size and then applies the constructs described above for each clause in given input collection C. By emptying and changing some grid squares according to these steps, this algorithm creates a sudoku puzzle. By way of illustrating this process, an example follows.

Example: Take the instance of the 3SAT with Boolean variables $U = \{u_1, u_2, u_3\}$ and the collection $C = \{\{u_1, u_2\}, \{\overline{u_1}, u_2, u_3\}, \{\overline{u_3}\}\}$. This can be written in CNF as $(u_1 \vee u_2) \wedge (\neg u_1 \vee u_2 \vee u_3) \wedge (\neg u_3)$. The size of this input, the sum of the number of elements of each clause, is 6, so the sudoku must be at least of order 4. The algorithm generates a pre-existing solution of order 4, shown in Figure 4, in the 16×16 matrix S. This solution is created by filling the first row with the labels 1 to 16 in order and then shifting the row 4 squares to the left to create the next row until the first band is filled. Then for the first row of the next band, the first row of the previous band is shifted 1 to the left and the process is repeated until the sudoku is filled.

The first clause $\{u_1, u_2\}$ has size 2. The squares $S(3, 1), S(3, 2)$ and $S(4, 1)$ are chosen, as they are three grid squares in the first sub-block that are not in any of the 2×2 blocks along the diagonal. The labels of these are 9, 10, and 13, so every instance of those labels is removed and then $S(3, 1)$ and $S(3, 2)$ are replaced by a and b defined as follows.

$$a = \begin{cases} 9 & \text{if } t(u_1) = T \\ 13 & \text{if } t(u_1) = F \end{cases} \qquad b = \begin{cases} 10 & \text{if } t(u_2) = T \\ 13 & \text{if } t(u_2) = F \end{cases}$$

The next clause $\{\overline{u_1}, u_2, u_3\}$ has 3 elements and it is the first of this size. Therefore, the first 2×2 block along the diagonal, the one containing $S(1, 1), S(1, 2), S(2, 1)$ and $S(2, 2)$, will be chosen. The labels in these squares are $1, 2, 5$ and 6, so each instance of those labels is removed, and then $S(1, 1)$ is replaced by the variable c defined as follows.

$$c = \begin{cases} 1 & \text{if } t(\overline{u_1}) = T \\ 2 & \text{if } t(\overline{u_1}) = F \end{cases}$$

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
2	5	6	7	8	9	10	11	12	13	14	15	16	1	2	3	4
3	9	10	11	12	13	14	15	16	1	2	3	4	5	6	7	8
4	13	14	15	16	1	2	3	4	5	6	7	8	9	10	11	12
5	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	1
6	6	7	8	9	10	11	12	13	14	15	16	1	2	3	4	5
7	10	11	12	13	14	15	16	1	2	3	4	5	6	7	8	9
8	14	15	16	1	2	3	4	5	6	7	8	9	10	11	12	13
9	3	4	5	6	7	8	9	10	11	12	13	14	15	16	1	2
10	7	8	9	10	11	12	13	14	15	16	1	2	3	4	5	6
11	11	12	13	14	15	16	1	2	3	4	5	6	7	8	9	10
12	15	16	1	2	3	4	5	6	7	8	9	10	11	12	13	14
13	4	5	6	7	8	9	10	11	12	13	14	15	16	1	2	3
14	8	9	10	11	12	13	14	15	16	1	2	3	4	5	6	7
15	12	13	14	15	16	1	2	3	4	5	6	7	8	9	10	11
16	16	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	c		f	4			7	8			11	12		14	15	16
2			7	8			11	12		14	15	16	e		3	4
3	a	b	11	12		14	15	16			3	4			7	8
4		14	15	16			3	4			7	8			11	12
5		3	4			7	8			11	12		14	15	16	
6		7	8			11	12		14	15	16			3	4	
7		11	12		14	15	16			3	4			7	8	
8	14	15	16			3	4			7	8			11	12	
9	3	4			7	8			11	12		14	15	16		
10	7	8			11	12		14	15	16			3	4		
11	11	12		14	15	16			3	4			7	8		
12	15	16			3	4			7	8			11	12		14
13	4	d		7	8			11	12		14	15	16			3
14	8			11	12		14	15	16			3	4			7
15	12		14	15	16			3	4			7	8			11
16	16			3	4			7	8			11	12		14	15

Figure 4: The generated solution of order 4 and the result of the transformation

The second column intersects with the 2×2 block, but not with $S(1,1)$. The first grid square in this column, that previously contained $1, 2, 5$ or 6, is $S(13,2)$, so this square will contain the variable d defined as follows.

$$d = \begin{cases} 5 & \text{if } t(u_2) = T \\ 1 & \text{if } t(u_2) = F \end{cases}$$

Similarly, the second row intersects with the 2×2 block, but not with $S(1,1)$. The first grid square in this column, that previously contained $1, 2, 5$ or 6, is $S(2,13)$, so this square will contain the variable e defined as follows.

$$e = \begin{cases} 6 & \text{if } t(u_3) = T \\ 1 & \text{if } t(u_3) = F \end{cases}$$

The last clause $\{\overline{u_3}\}$ has 1 element, so all that is needed is a non-blank square in the first sub-block. Take $S(1,3)$ and replace it with f defined as follows.

$$f = \begin{cases} 3 & \text{if } t(\overline{u_3}) = T \\ 17 & \text{if } t(\bar{u_3}) = F \end{cases}$$

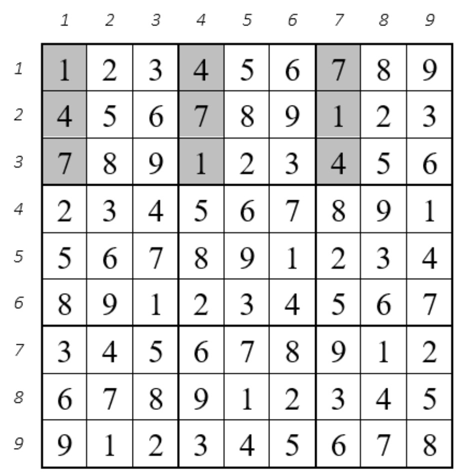
In this way, the generated solution shown on the left in Figure 4, is transformed into a sudoku puzzle. However, the resulting puzzle contains some squares filled with the variables above, which are in essence functions of the truth assignment t and a Boolean variable u_k. This creates a problem, as these are not a part of the problem definition for the SPC problem. What has now been created is a transformation from 3SAT into the following problem.

Given an order n, a matrix $S \in \mathbb{R}^{n^2 \times n^2}$ and a set H of coordinates (i,j) s.t. $S(i,j) \in [1, n^2]$ and the elements in H^C are left blank, does there exist a truth assignment t such that there is a way to assign the labels 1 to n^2 to the elements in H^C such that the same label appears exactly once in each sub-line?

This adapted version of the SPC problem is at first sight more complex than the original SPC problem and it does not seem possible to reduce it to the SPC problem, so this transformation is not successful in proving the NP-completeness of the sudoku puzzle. As the goal of the 3SAT problem is to find a truth assignment, this truth assignment should not be necessary to solve the instance of SPC, but this is the case. It seems too difficult to adapt this transformation into one that doesn't use the truth assignment. As 3SAT is NP-complete, it is not possible to determine a truth assignment that satisfies C in polynomial time and as the Boolean variables in U do not have any value without a truth assignment, it is difficult to incorporate them into defining the sudoku solution. Therefore, this transformation is left here as proof that the adapted version of the SPC problem is NP-complete.

3.3 Reducing the Latin Square Completion problem to the SPC problem

Although the proof in the previous chapter failed, it is already known that the problem of finding a solution to a sudoku puzzle is NP-complete. This was first proven in [11] using the Latin square. However, the proof is quite theory-heavy and though it is sound, the line of reasoning is hard to follow



	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 5: A Latin square of order 3 can be found in a sudoku puzzle of order 3

at times. This section will follow Chapter 3.2 from [11] and attempt to make it more accessible to people just starting research into complexity.

The proof in [11] is based on the *Latin square*. A Latin square of order n is an $n \times n$ grid in which each column and row contains each integer from 1 to n exactly once. From this definition it is easy to see that it is quite similar to a sudoku puzzle, only missing the rule for sub-blocks, so making a connection between them seems very logical. The Latin square also has a complexity problem similar to the SPC problem, namely the Latin Square Completion (LSC) problem, which is defined as follows.

Given an order n, a matrix $L \in \mathbb{R}^{n \times n}$ and a set H of coordinates (i, j) s.t. $L(i, j) \in [1, n]$ and the elements in H^C are left blank, is there a way to assign the labels from 1 to n to the elements in H^C such that the same label appears exactly once in each row and column?

In [1] it was proven that the LSC is NP-complete with a reduction from the tripartite graph triangle-partition problem. This means that giving a reduction, that can be computed in polynomial time, from LSC to SPC will be enough to prove that the SPC problem is NP-complete.

In order to give a reduction, given an instance of the LSC problem, a sudoku puzzle should be constructed that is solvable if and only if the Latin square is. Taking n as the order of the Latin square, the starting point for this transformation is a sudoku puzzle solution of order n, which is constructed in the same way as in the example in Section 3.2. In this solution, a Latin square can be found with non-consecutive labels. In the top band, the leftmost columns of each sub-block contain the same integers, all in different positions. Taking these squares out and pasting them next to each other creates a Latin square. An example for order 3 is shown in Figure 5. If the marked grid squares are made blank, the resulting puzzle has a solution. If the labels of the given LSC problem are then adjusted to fit those from the sudoku puzzle and these labels are placed into the blank squares, the result is a sudoku puzzle which is solvable only if the Latin square can be completed. This is the outline of how the transformation will work. Now, it will be described mathematically and the proof of the complexity of SPC will follow.

What is given at the start is an instance $I = \langle n, L, H \rangle$ of the Latin Square Completion problem, consisting of an $n \times n$ matrix L and a set $H = \{(i, j) \mid i, j, L(i, j) \in [1, n]\}$. All the squares that are not in H are blanks, so $H^C = \{(i, j) \mid i, j \in [1, n], L(i, j) = \perp\}$, where $\perp$ indicates a blank square. The first step is to generate the sudoku solution S of order n as described in the previous paragraph, which can be defined by the following formula.

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

In this formula, $j - 1$ ensures that the labels are in order in each row, $((i - 1) \bmod n) \cdot n$ ensures that the rows within one band are each shifted n squares to the left. In the expression $\lfloor \frac{i-1}{n} \rfloor$, the brackets mean that the value is rounded down to the closest integer. This ensures that the first row of each band is the first row of the previous band shifted one square to the left. The formulation is similar to the one in [11], however in this version the +1 at the end and using $j - 1$ and $i - 1$ instead of just i and j keeps the labels the same as in the original definition (1 to n^2), where Yato used 0 to $n^2 - 1$. The fact that this is a valid sudoku puzzle is left to the reader, as it is easy to observe from the example in Figure 5. Next, a set B is defined which contains the grid squares that make up the Latin square.

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

This set contains the intersection between the first band ($i \in [1, n]$) and the first column of each stack ($(j-1) \bmod n = 0$). To complete the transformation, an instance $I' = \langle n, S, H' \rangle$ of SPC is defined with the same order n as the instance I of LSC. The other parts are defined as follows.

$$H' = B^C \cup \{(i, 1 + (j-1)n) \mid (i, j) \in H\}$$

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

In the second case in the definition of S, the labels from L are adjusted to fit into B inside the sudoku S and, by the third case, the squares that are blank in L are left blank in S as well. The values that are not in B are left unchanged from the original solution.

Claim: Instance I of the LSC problem is solvable if and only if instance I' of the SPC is solvable.

Proof. First, the proof from left to right. It is assumed that the instance I is solvable and it must be shown that the instance I' is solvable. Say an $n \times n$ matrix L' is a solution to I, then the goal is to prove that there exists a solution to the instance I' of the SCP problem. The following suggested solution S' is essentially defined as S above in the case that there are no empty squares, so when $(H')^C = \emptyset$.

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

For this to be a valid solution, the labels should only take values from 1 to n^2 and each label should only occur once in each sub-line. The first fact can be easily shown. For $(i, j) \in B$, as L' is a solution of order n, $1 \leq L'(i, j) \leq n$ and $L'(i, j) \in \mathbb{Z}$. This means that $S(i, j) = 1 + (L'(i, j) - 1)n \in \mathbb{Z}$ $1 \leq 1 + (L'(i, j) - 1)n \leq 1 + (n - 1)n \leq n^2$, so $S'(i, j) \in [1, n^2]$. As the values for $(i, j) \notin B$ were defined to already be a valid solution, $S'(i, j) = S(i, j) \in [1, n^2]$ for all $i, j \notin B$. This also means that the sub-lines that contain no elements of B do not need to be considered in the proof.

Assume $i \in [1, n]$, which means that the coordinates are in the top band. With i fixed, take $j, k \in [1, n^2]$ such that $j \neq k$. The row rule states that $S'(i, j) \neq S'(i, k)$ should hold for all such i, j, k. If $(i, j), (i, k) \notin B$, then this fact follows from the fact that all labels outside B are defined to give a valid solution. Instead, say $(i, j), (i, k) \in B$. These squares in S correspond with $((i, \frac{j-1}{n} + 1)), (i, \frac{k-1}{n} + 1)$ in L'. As $j \neq k$, it holds that $\frac{j-1}{n} + 1 \neq \frac{k-1}{n} + 1$, so, as L' is a solution to I, $L'(i, \frac{j-1}{n} + 1) \neq L'(i, \frac{k-1}{n} + 1)$, so

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

For the last case, without loss of generality, say that $(i, j) \in B$ and $(i, k) \notin B$. This means

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

It is easy to see that $S'(i, j) \bmod n = 1$. Say $S'(i, j) = S'(i, k)$, implying $S'(i, k) \bmod n = 1$, and then work towards a contradiction. Note that $i \in [1, n]$ means that $\lfloor \frac{i-1}{n} \rfloor = 0$, so this term can be left out.

$$\begin{aligned} S'(i,k) \bmod n = 1 &\Rightarrow \bigg(\Big(\big(k + ((i-1) \bmod n) \cdot n - 1\big) \bmod n^2\Big) + 1\bigg) \bmod n = 1 \\ &\Rightarrow \Big(\big(k + ((i-1) \bmod n) \cdot n - 1\big) \bmod n^2\Big) \bmod n = 0 \\ &\Rightarrow \exists s \in \mathbb{Z} \text{ s.t. } \big(k + ((i-1) \bmod n) \cdot n - 1\big) \bmod n^2 = sn \\ &\Rightarrow \exists t \in \mathbb{Z} \text{ s.t. } k + ((i-1) \bmod n) \cdot n - 1 = sn + tn^2 \\ &\Rightarrow k - 1 = sn + tn^2 - ((i-1) \bmod n) \cdot n = n(s + tn + (i-1) \bmod n) \end{aligned}$$

This last equation shows that there exists a $u \in \mathbb{Z}$ s.t. $k-1 = un$, so $(k-1) \bmod n = 0$. But, as $(i, k) \in B$, $(k - 1) \bmod n = 0$, so a contradiction is reached. Therefore, $S'(i, j) \neq S'(i, k)$ for all $i, j, k \in [1, n^2]$ s.t. $j \neq k$, so the row rule is satisfied.

For the column rule, only the first columns of each stack are considered, so $(j-1) \bmod n = 0$. Given $i,k \in [1,n^2]$ such that $i \neq k$, the column rule states that $S'(i,j) \neq S'(k,j)$. Just like the row rule, if $(i,j),(k,j) \notin B$ the column rule already holds. If $(i,j),(k,j) \in B$, the corresponding squares in L' are $(i, \frac{j-1}{n}+1), (k, \frac{j-1}{n}+1)$. As $i \neq k$ and L' is a solution to I, $L'(i, \frac{j-1}{n}+1) \neq L'(k, \frac{j-1}{n}+1)$, so

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

For the last case, say that $(i,j) \in B$ and $(k,j) \notin B$, then $i \in [1,n]$ and $k \in [n+1,n^2]$. This implies that $S'(i,j) \bmod n = 1$, so if it is assumed that $S'(i,j) = S'(k,j)$, then $S'(k,j) \bmod n = 1$. Again, this leads to a contradiction.

$$\begin{aligned}
S'(k,j) \bmod n = 1 &\Rightarrow \left(\left(\left(j + ((k-1) \bmod n)\cdot n + \left\lfloor \frac{k-1}{n} \right\rfloor - 1\right) \bmod n^2\right) + 1\right) \bmod n = 1\\
&\Rightarrow \left(\left(j + ((k-1) \bmod n)\cdot n + \lfloor \tfrac{k-1}{n} \rfloor - 1\right) \bmod n^2\right) \bmod n = 0\\
&\Rightarrow \exists s \in \mathbb{Z} \text{ s.t. } \left(j + ((k-1) \bmod n)\cdot n + \lfloor \tfrac{k-1}{n} \rfloor - 1\right) \bmod n^2 = sn\\
&\Rightarrow \exists t \in \mathbb{Z} \text{ s.t. } j + ((k-1) \bmod n)\cdot n + \lfloor \tfrac{k-1}{n} \rfloor - 1 = sn + tn^2\\
(j-1) \bmod n = 0 &\Rightarrow \exists u \in \mathbb{Z} \text{ s.t. } ((k-1) \bmod n)\cdot n + \lfloor \tfrac{k-1}{n} \rfloor + un = sn + tn^2\\
&\Rightarrow \lfloor \tfrac{k-1}{n} \rfloor = sn + tn^2 - un + ((k-1) \bmod n)\cdot n = n(s + tn - u + (k-1) \bmod n)
\end{aligned}$$

This last equation shows that there exists a $v \in \mathbb{Z}$ s.t. $\lfloor \frac{k-1}{n} \rfloor = vn$, so $\lfloor \frac{k-1}{n} \rfloor \bmod n = 0$. However, as $k \in [n+1,n^2]$, $1 \leq \lfloor \frac{k-1}{n} \rfloor \leq n-1$. This is a contradiction, as it would mean that $\lfloor \frac{k-1}{n} \rfloor \bmod n \neq 0$. Therefore, $S'(i,j) \neq S'(k,j)$ for all $i,j,k \in [1,n^2]$ s.t. $i \neq k$, so the column rule is satisfied.

For the sub-block rule, only the top band needs to be considered, so $i,k \in [1,n]$. The sub-block rule states that $S'(i,j) \neq S'(k,l)$ should hold for all $(i,j),(k,l)$ in the same sub-block s.t. $i \neq k$ and $j \neq l$. Again, if $(i,j),(k,l) \notin B$, then this rule already holds. As, $j \neq l$, (i,j) and (k,l) must be in different columns, but they are in the same sub-block, so only one of them can be in B. This leaves the case where $(i,j) \notin B$ and $(k,l) \in B$. This means $S'(k,l) \bmod n = 1$, but $(i,j) \notin B$ means that $(j-1) \bmod n \neq 0$. For the row rule, it was shown that

$$\left.\begin{array}{l} i \in [1,n] \\ (j-1) \bmod n \neq 0 \end{array}\right\} \Rightarrow S'(i,j) \bmod n \neq 1$$

This means that $S'(i,j) \neq S'(k,l)$ for all $(i,j),(k,l)$ in the same sub-block such that $i \neq k$ and $j \neq l$. All the sudoku rules have been shown to hold, so S' is a valid solution to I.

Next, the proof from right to left. Say the $n^2 \times n^2$ matrix S' is a solution to the instance I' of SPC, then a solution L' to the LSC problem can be formulated by rewriting the definition of S' in the first part of the proof. The squares of the sudoku that are not in B are not considered, as they will not affect the solution L'. This means $S'(i,j) = 1 + (L'(i, \frac{j-1}{n}+1)-1)n$, which can be rewritten as

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

For ease of notation j' denotes $1+(j-1)n-1$. It will be assumed that $i \in [1,n]$, so that only the squares in B are considered. The coordinates (i,j') are in B as

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

For L' to be a valid solution, the labels should only take values from 1 to n and each label should only occur once in each row and column. To show the first fact, consider that $(i,j') \in B$ means that $S'(i,j') \in \mathbb{Z}$ and $S'(i,j') \bmod n = 1$. This means that $S'(i,j')-1 \bmod n = 0$, so $L'(i,j) = 1 + \frac{S'(i,j')-1}{n} \in \mathbb{Z}$. The first two facts combined with the fact that $S'(i,j') \in [1,n^2]$, as S' is a solution to I, show that

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

So the labels of L' are all in $[1,n]$. The row/column rules follow from the row/column rules in S'.

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

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

So L is a valid solution to I, which means the claim has been proven. □

It has now been shown that the instance $I' = \langle n, S, H' \rangle$ of the SPC problem as defined at the start of this section is solvable *if and only if* the instance $I = \langle n, L, H \rangle$ of the LSC problem is solvable. This means that this transformation from I to I' is a reduction from LSC to SPC.

Claim: The SPC problem is NP-complete

Proof. As stated earlier, the LSC problem has been proven to be NP-complete. At the start of section 3.2, it was shown that the SPC problem is NP, by showing a solution could be checked in polynomial time. In this section, it has been shown that there is a reduction from LSC to SPC. This reduction can be easily performed in polynomial time. Given an instance of the LSC problem, the input size is polynomially related to the order n. An $n^2 \times n^2$ matrix is created $(O(n^4))$ and the appropriate labels are assigned according to the definition of $S(i, j)$ $(O(n^4))$. This means the process is of time complexity $O(n^4)$. As the reduction can be performed in polynomial time, this shows that the SPC problem is NP-complete. $\square$

4 The uniqueness problem

Now the problem of finding a solution to a sudoku puzzle has been discussed. However, there is an important difference between *the existence of a solution* and *solvability.* For a puzzle to be solvable, there can't be more than one solution. If a puzzle were to have multiple solutions, a person solving it would eventually get stuck, as some squares would have multiple possible labels. This brings up the question of uniqueness, in other words, given a solution S_1 to an instance I of a problem, does there exist another solution S_2 such that $S_1 \neq S_2$? This problem is called the Another Solution Problem and for any problem, there is an associated ASP version of that problem. So, to study the uniqueness of a solution to the SPC problem, the complexity of ASP-SPC will be investigated.

The ASP version of a problem can be shown to be NP-complete in a similar way to the problem itself. The main distinction is that a normal reduction from problem A to problem B is not enough, but a special case of reduction is needed, namely a *parsimonious reduction*, which has the special property that the number of solutions to instance I of problem A is the same as the number of solutions to the transformed instance I' of problem B. In other words, a parsimonious reduction is a bijection between the sets of solutions of both problems. If A can be reduced to B with such a reduction and ASP-A is NP-complete, then ASP-B is also NP-complete. So, to construct a proof that ASP-SPC is NP-complete, or in other words, SPC is ASP-complete, the strategy is to find a problem whose ASP is NP-complete and that can be reduced to SPC parsimoniously.

4.1 The original proof of NP-completeness as a proof of ASP-completeness

In the original paper [11], Yato's goal was not to simply prove the NP-completeness of the sudoku puzzle. The paper was focused on the Another Solution Problem and considered many different logical puzzles and their ASPs. The reduction, used in section 3.2 of this paper, was used in [11] to prove the ASP-completeness of SPC. When observing this reduction again, it seems reasonable to say that it is parsimonious, as any solution to the instance of LSC can be transformed into a solution to the instance of SPC, implying the number of solutions each problem has would be the same.

So, the reduction is parsimonious, but it should also be shown that the LSC problem itself is ASP-complete. In [11], Yato points to [2], a paper similar to [1]. In this paper, Colbourn proves not only that the Latin Square Completion problem is NP-complete, but that its Another Solution Problem is NP-complete as well. He reduces the satisfiability problem UNIQUE 1-IN-4 SAT, which is NP-complete, to the problem of unique partition into triangles of tripartite graphs, which is then in turn reduced to the Latin Square Completion uniqueness problem. The problems in this paper are all in the form of finding out if there are multiple solutions, in other words, they are ASPs. Every reduction preserves the number of solutions each problem has, which effectively proves that ASP-LSC is NP-complete. So, as ASP-LSC was reduced with a parsimonious reduction in section 3.3, the fact that ASP-SPC is NP-complete follows.

It seems as if, after [11] was published, not much further research has been done on how to prove the SPC problem and its ASP are NP-complete. This does make sense, as when something is proven, researchers can simply point to who proved it and assume the claim holds from there. It is however quite important that a proof can be easily understood. If it is difficult to understand the way something is proven, a researcher may use the claim, without truly understanding why it holds. Furthermore, continuing research into these proofs may reveal problems with the original proof, easier alternative proofs, or completely separate conclusions. With this in mind, the proof from the previous chapter will not be abandoned and it will be attempted to adapt it into a parsimonious reduction to use it to show the ASP-completeness of the adapted SPC problem.

4.2 Expanding the suggested transformation to prove the adapted SPC is ASP-complete

To be able to use the transformation from the previous chapter, it should be shown that the ASP for 3SAT is NP-complete. In the first proof that SAT is NP-complete by Cook, the reduction used is a parsimonious one, as it preserves the number of solutions. This means that the ASP for SAT is NP-complete. In section 3.2, it was shown that 3SAT is NP-complete by reducing SAT to it, and the reduction from that proof turns out to be parsimonious. A truth assignment t over U can only be extended to U' in one way, as all the new variables are equivalent to a disjunction of literals from U, so their truth assignment is already fixed. This means that the number of truth assignments that satisfy C is the same as the number that satisfy C', so therefore the transformation retains the number of solutions and is a parsimonious reduction. So, as SAT can be reduced to 3SAT with a parsimonious reduction, ASP-3SAT is NP-complete. With this proven, it is only necessary to find a parsimonious reduction from 3SAT to the

	1	2	3	4
1	a	b	d	4
2	b	4	a	b
3	b	d	4	a
4	4	a	b	d

	1	2	3	4
1	e	2	3	4
2	3	4	1	2
3	2	3	4	1
4	4	1	2	3

Figure 6: New constructs for clauses of 2 and 3 literals

adapted version of the SPC problem to show that the ASP for the SPC problem would be NP-complete.

If it has to be shown that a transformation conserves the number of solutions, it must be clear what constitutes a solution. For 3SAT, the solution would be a truth assignment t which satisfied the collection. The solution to the adapted version of the SPC problem from Chapter 3 is a pair (t, S) of a truth assignment t and a sudoku puzzle solution S. Because of the way the transformation is defined, the truth assignment is the same in both problems, so for the transformation to be parsimonious, there should be only one possible sudoku solution for each truth assignment t. This means the transformation from Chapter 3 turns out not to be parsimonious. This can be easily seen from the construct for a clause of 2 literals in Figure 3, which has multiple possible solutions. Only row and column 1 can be filled in with numbers that certainly belong there, but the remaining squares leave different possibilities. To be able to use this transformation, it will need to be adapted.

To ensure that there is only one possible sudoku solution for each truth assignment, the constructs for clauses of 2 and 3 literals will need to be changed. The construct for 1 literal already works as needed, as it leaves no empty squares, thus creates no extra solutions.

- The construct for a clause with 2 literals will be changed by not removing all instances of each label that is used. Instead, all squares that contained the same label, will now contain the same variable. As a and b are placed in the squares where 1 and 2 were, every instance of 1 and 2 will be replaced by a and b respectively. For the remaining squares, a new variable will be introduced.

  $$d = 6 - a - b$$

  Each instance of the label 3 will be replaced with this variable. This results in the construct in Figure 6. As there are no empty squares, this construct doesn't allow for multiple solutions.
- The construct for a clause with 3 literals will have to be changed more. As a sudoku puzzle of order 2 only has 4 different labels, placing more than two of the variables defined in section 3.2 in one block will create contradictions even if one of the variables is true. A possible construct is similar to the construct for a clause of 1 literal. A new variable e is defined as follows.

  $$e = \begin{cases} 1 & \text{if } t(u_1) = T,\ t(u_2) = T \text{ or } t(u_3) = T \\ 5 & \text{otherwise} \end{cases}$$

  This is not the most interesting solution, but it does assure that every time one of the variables is true, the sudoku is unsolvable. One might note that this strategy could also be used for any other clause, but this would make the transformation less efficient computationally.

With these new constructs, a new reduction has been defined, which will ensure that each sudoku has only one solution if the clause is satisfiable. This parsimonious reduction shows that the adapted SPC problem defined in Chapter 3 is also an ASP-complete problem. However, it is of course still not enough to show that the original SPC problem is NP-complete.

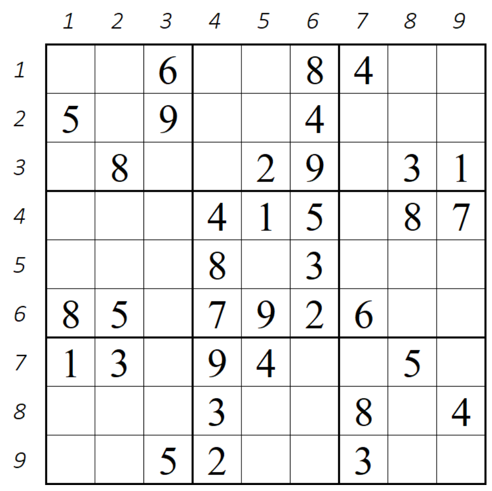


Figure 7: A sudoku puzzle

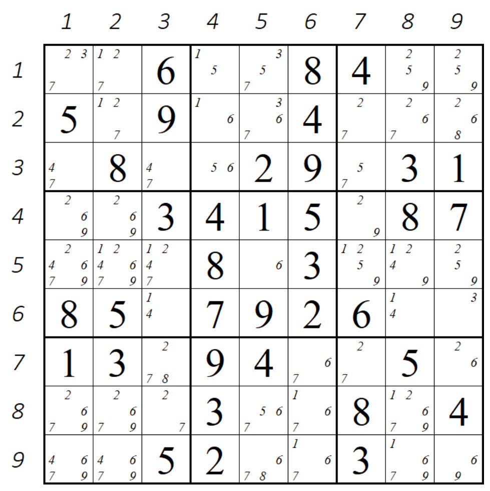


Figure 8: A sudoku puzzle with candidates

5 The solving procedure

Now that the complexities of the existence and uniqueness questions have been found, the solving procedure can be studied next. In this chapter, the steps of this procedure will be explained from those that are easy to observe, like one missing element in a sub-line, to the more complex steps, like colorings. These steps are taken from [4], in some cases slightly redefined, and some examples from that paper are also used here. The more complex steps will usually not be necessary for solving an average sudoku puzzle, but to make sure that for any sudoku with a unique solution this solution can be found, all the steps will be explained. After the solving procedure has been explained, several solvers will be compared on their efficiency in solving a set of test puzzles.

5.1 Solving steps

5.1.1 Unique Missing Labels

The most obvious step when solving a sudoku puzzle is to see if there are any sub-lines which already contain eight of the nine labels. Remember, the term sub-line not only refers to a row or column, but also a sub-block. If this is the case, the ninth empty square can be filled in automatically with the one label that is missing. In Figure 7, this is the case in the middle sub-block, where the only missing label is 6, so it can be filled in immediately.

5.1.2 Naked or Hidden Singles

A big part of the strategy for solving a sudoku puzzle is noting in each square which labels could be filled in with the information given thus far, also called *candidates*. An example of how this is done is shown in Figure 8. When this has been done, there may be a square that contains only one candidate, meaning that it can only contain that label and the square can be filled in. This is called a *naked single*. In Figure 8, a naked single appears in $S(6,9)$, so 3 can be filled in. Whenever a square is filled in, it affects the candidates of other squares, which may result in more naked singles.

Similarly, there might be a sub-line in which a certain candidate appears only once. This means that there is no other square this label could occur in that sub-line, so therefore it must be in this square. This is called a *hidden single*. In Figure 8, there are a few hidden singles, for example in $S(1,1)$. This square contains the only instance of a 3 in the upper left sub-block. As the sub-block must contain a 3, it must be in $S(1,1)$, so it can be filled in.

5.1.3 Locked Candidates

If a sub-block contains multiple instances of a certain candidate, it is still possible to gain information from it. If a candidate only appears in one row or column within the sub-block, then the instance of that label in that sub-block must be in that row or column. The position of this candidate is *locked* in that sub-block. This means that in the other sub-blocks this row or column intersects with, that label can't be in the same row or column. So the candidate can be removed from all squares in the row or column that are not in the original sub-block.

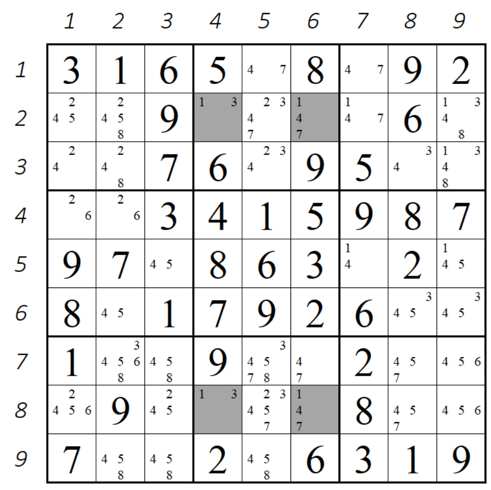


Figure 9: An X-Wing

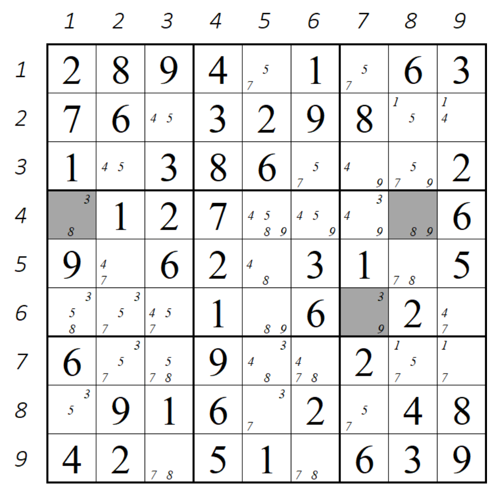


Figure 10: An XY-Wing

This can also occur with a candidate that appears only in one sub-block in a certain row/column. In this case, all the instances of that label in the rest of the sub-block can be removed, as otherwise there would be no place for the candidate in that row/column.

In Figure 8, the first kind of locked candidate occurs in the lower right sub-block. The candidate 1 only appears in the middle column in this block, meaning that it can not occur in that column in the sub-blocks above it. So the candidate 1 can be removed from $S(5,8)$ and $S(6,8)$.

5.1.4 Naked and Hidden Groups

Similarly to the naked and hidden singles, there may be i squares in a sub-line that together have only i distinct candidates. In this case, all these labels must be contained in the i squares, so they can not occur in another square in the same sub-line. This means the candidates can be removed from all other squares in the sub-line. Depending on the value of i, these groups of squares are called *naked pairs*, *triplets* or *quads*. The most easily spotted are naked pairs, as they consist of two squares that contain the same two candidates. For triplets, it is sufficient for the squares together to contain only three (or in the case of quads, four) candidates. In Figure 8, some naked pairs can be spotted fairly quickly, for example $S(1,5)$ and $S(2,5)$, which both only contain only the candidates 3 and 7. There are also some naked triples, like in column 6. Because the squares $S(7,6)$, $S(8,6)$ and $S(9,6)$ only contain the candidates 1, 6 and 7, those candidates can be removed from the other squares in the sub-block, $S(8,5)$ and $S(9,5)$.

It can also be the case that there are i candidates that only appear in i squares in one sub-line. The difference here is that other candidates may also appear in the squares, as long as the i candidates do not appear in any other squares in the sub-line. These are called *hidden pairs*, *triplets*, or *quads*. In the case of a hidden pair, there would be two candidates that only appear in two squares. This again means that those labels must be in those squares, so the squares can have no other candidates. From these definitions, it might be easy to see that if a naked group occurs, the other squares in that sub-line are automatically a hidden group, as the remaining labels only occur in those squares. As the squares $S(1,8)$ and $S(1,9)$ in Figure 8 contain a naked pair with 2 and 9, the other blank squares in the first row contain a hidden triple with 1, 3, and 7. Those three labels only appear in $S(1,1)$, $S(1,2)$, and $S(1,5)$ in that row, so no other labels could be filled in in those squares.

5.1.5 X-Wings

Another situation is where one candidate appears exactly i times in i rows, in the same columns in each row. In this case, the instance of that label in each column must be in one of the i rows. So, in the squares in those columns that are not in one of the i rows, the candidate should not occur. The same holds if "row" and "column" are interchanged. If i is equal to 2, the squares form a rectangle and two opposing corners must contain the label. This pattern is where the name *X-Wing* comes from. In Figure 9, an X-Wing is formed by $S(2,4)$, $S(2,6)$, $S(8,4)$, and $S(8,6)$. As the candidate 1 only appears twice in the columns 4 and 6, it can be removed from any other squares in the rows 2 and 8, particularly $S(2,7)$ and $S(2,9)$.

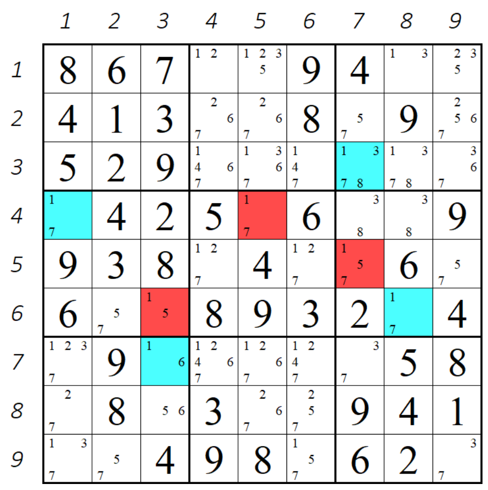


Figure 11: A single coloring

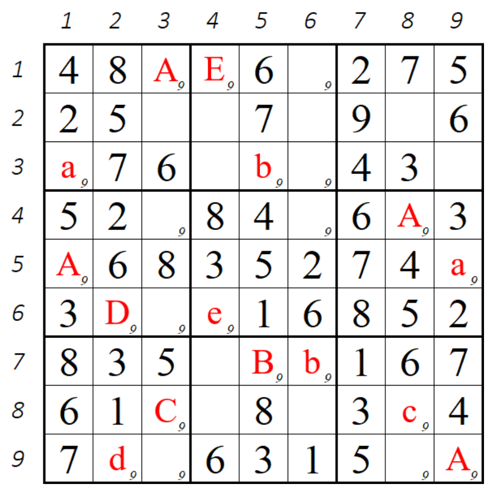


Figure 12: Multi-coloring

5.1.6 XY-Wings

For the next few steps, the procedure becomes a bit more like guessing, where a square is filled in and the consequences are observed. However, these steps are more structured than simply guessing and seeing what happens. For *XY-wings*, the general idea is that, given a square which has only two candidates, if the first candidate is filled in and this leads to a conclusion, which also holds if the second candidate is filled in, then that conclusion holds in either case, so it can be applied.

In an XY-wing, a square XY with two candidates, x and y, influences two other squares, XZ and YZ, who contain two candidates: x and z, respectively y and z. If x is filled in, then in the first square z will be filled in. If y is filled in, z will be filled in the second square. This means that no matter which label is filled in into square XY, an instance of the label z will follow in either XZ or YZ. Then, in the intersection of any of the sub-lines that contain XZ and YZ, there can not exist an instance of z, as this would mean z can't occur in XZ and YZ. So, the candidate z should be removed from the intersection of the sub-lines XZ and YZ are in. This intersection could be between a row/column and a sub-block, or between a row and a column. If XZ and YZ were in the same sub-block, this would simply be a case of a naked triple with labels x, y, and z.

In Figure 10, an XY-wing occurs in the middle band. The square $S(4,8)$ has two candidates, 8 and 9, and if one of them is filled in, then a 3 will be filled in in either $S(4,1)$ or $S(6,7)$. This means that the squares where their sub-lines overlap could not contain a 3. These squares are $S(6,1)$, $S(6,2)$, and $S(6,3)$.

5.1.7 Coloring

If there is a sub-line where one candidate appears only two times, then filling in one of them with that label, means the other will not contain that label. This may lead to another square becoming the only one containing an instance of that candidate in a certain sub-line, so then that one can be filled in as well. In this way, a chain can be constructed of cells that alternate between being filled in and not being filled in. If instead of filling in the first square, the candidate was removed from this square, then the chain would be the same, but with the filled in and not filled in squares reversed. This means that some squares are always the same "color" (filled in or not) as each other. This can lead to contradictions, as the intersection of the row and column of two differently colored squares can not contain the label, as that would mean neither of the colored squares can contain the label. In Figure 11, a coloring is shown starting with $S(4,1)$. If a 1 is filled in in this square, then all the blue squares will contain a 1. If the candidate 1 is removed from the square, then all the red squares will contain a 1. This means that $S(3,5)$ and $S(7,5)$ can't contain a 1, as they are on the intersection of differently colored squares.

If one chain does not lead to a conclusion, multiple chains can be constructed for the same candidate. Then, if any two differently colored squares are in the same sub-line, they can not both contain the label, so connections can be made between the different chains and conclusions may be drawn from it. An example is shown in 12 for the candidate 9, where three different chains are shown, indicated with interchanging capital and lowercase letters. There are many places where different letters appear in the same sub-line, for example A and E in the first row. This means that if the squares with an E contain a

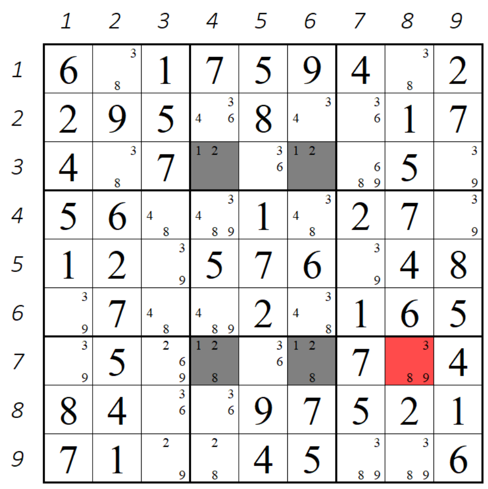


Figure 13: Avoiding multiple solutions

9, then the squares with an A can't contain a 9. This will be written as $A!E$ (or $E!A$, interchangeably). By definition $X!x$ for any of the letters a through e. By looking at other places where different letters overlap, the following connections can be made.

$$\begin{array}{ccccccccc} & & A!a & B!b & C!c & D!d & E!e & & \\ A!b & A!C & A!c & A!D & A!E & a!b & C!D & D!e & E!b \end{array}$$

Now, this set of connections can be expanded, using the fact that the connection ! is transitive. For example, take $A!a$, $A!C$, and $a!b$. If square C contains a 9, then squares A don't contain a 9, which means that squares a contain a 9, so squares b can't contain a 9. So it can be concluded that $C!b$. Applying this method wherever possible yields the following full set of connections.

$$\begin{array}{ccccccccc} & & A!a & B!b & C!c & D!d & E!e & & \\ A!b & A!C & A!c & A!D & A!E & a!b & C!D & D!e & E!b \\ & & b!b & b!C & b!c & b!D & C!e & & \\ & & A!A & A!b & b!e & b!b & A!e & & \end{array}$$

With this full set of connections, contradictions can be found. For example, A!A and b!b imply that squares with A or b can't contain 7, so then it must be filled in in the squares with a and B. Besides these cases, for each connection, the intersection of the sub-lines with their opposites can't contain the label. For example, as A!e, any intersection between a square with a and E can not contain the candidate 7. This means that 7 can't be filled in in $S(3,6)$.

5.1.8 Uniqueness of the solution

Though this assumption is not made in this paper, it is usually given when one is solving a sudoku puzzle, that there is only one solution. As was stated in Chapter 4, if a sudoku puzzle has multiple solutions, it is not solvable. So, it should not be possible that there are squares that could be filled in in two ways, no matter the rest of the puzzle. Such a situation happens when there is a rectangle of four squares which all contain the same two candidates. In this case, these squares could be filled in in two ways, which contradicts the assumption that there is only one solution.

If there is such a rectangle, with the exception that in one of the rows or columns, both squares contain one other candidate, then that candidate must be in one of the two squares. If this is not the case, the result is a rectangle as described above. This means that all instances of that candidate in the same row or column must be removed. In Figure 13, this is the case with the rectangle containing $S(3,4)$, $S(3,6)$, $S(7,4)$, and $S(7,6)$. This square contains the candidates 1, 2, and 8. If the instances of 8 were removed, the rectangle would have two possible solutions. This means that an 8 can't be placed in another square in row 7, so the candidate 8 is removed from $S(7,8)$.

5.1.9 Guessing

If applying all the steps above does not yet yield a solution, then the last possibility is simply guessing. In this case, a label is assigned to a square at random and then the solving procedure continues until a contradiction is reached or until the entire grid is filled in. This is similar to the way some solvers approach a sudoku puzzle, known in computing science as backtracking or brute-forcing.

5.2 The approach of different solvers

There are a few different approaches to solve a sudoku puzzle with an algorithm. The three methods that will be studied here are *backtracking*, *stochastic search*, and *constraint programming*. In the next sections, examples of such solvers will be shown and tested, by solving six test puzzles with each code.

5.2.1 Backtracking

On `techwithtim.net/tutorials/python-programming/sudoku-solver-backtracking/`, a code by a programmer known online as Tech With Tim can be found. This code solves a sudoku puzzle using backtracking. This method finds the first empty square and fills The first candidate. If no contradiction is created, it moves on to the next square and fills in the first candidate there. If a contradiction is created, it fills in the next candidate, until there is no more contradiction. Whenever every candidate has been tried for a certain square, but a contradiction is created every time, the algorithm returns to the previous square and fills in the next candidate there. In this way, the algorithm runs through the entire grid, until it reaches the bottom right square and fills in a successful value there. If this happens, a solution has been found and the algorithm ends and shows the solution it has found. This method will from now on be referred to as BAC.

From the way this algorithm works, some observations/predictions can be made. Firstly, as the program runs until it finds one solution, given a puzzle that has multiple solutions it would give only one. However, its run-time would likely not be affected by the fact that there are multiple solutions, as the procedure is not affected by one cell having multiple possible labels. As long as one label works, it can continue. Secondly, as the program has to try values in each empty square, the number of empty squares would likely affect the run-time. In each empty square, each possible candidate is checked. Therefore, each extra empty square adds a factor between 1 and 9 to the run-time. Additionally, if there are more empty squares, there are fewer hints, so the other empty squares will also have more candidates. The exact increase depends on if the square appears early in the puzzle (in a higher row and/or in a column more towards the left) and on the number of candidates that are smaller than the (smallest) value the square takes in (one of) the solution(s). A backtracking algorithm would always find a solution if there is one, but as it may spend a long time assuming a value which is wrong and it does not apply much logic in deciding the value in each square, it would likely spend more time on finding a solution than other methods.

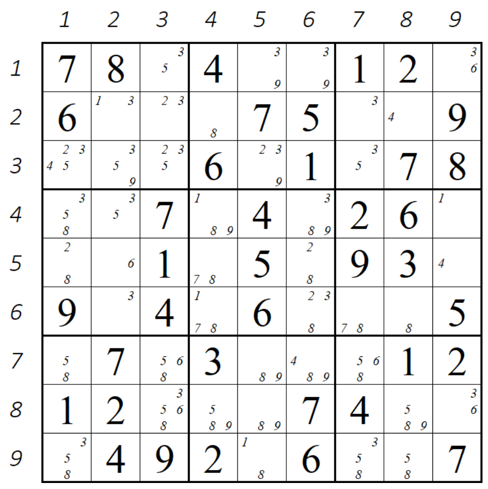
5.2.2 Stochastic Search

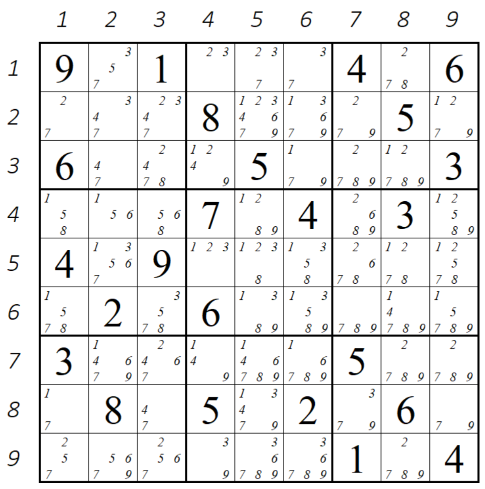
On `github.com/ananthamapod/Sudoku`, a code written by Ananth Rao can be found. This code can solve a sudoku puzzle with a stochastic search, more specifically a genetics algorithm. This algorithm starts off by creating a series of n candidate boards by randomly filling in the empty squares in the puzzle. It then checks how many mistakes each board has and if this number is zero, the puzzle is solved. If a board does have mistakes, a set of m successors is generated from it by taking two squares in the same row that were not given in the original puzzle and switching their labels. These successors are added to the set of candidate boards, which is then again sorted by the number of mistakes and the n boards with the least mistakes are taken. With this new set of n boards, the process is then repeated, until a solution is found. This method will from now on be referred to as STO.

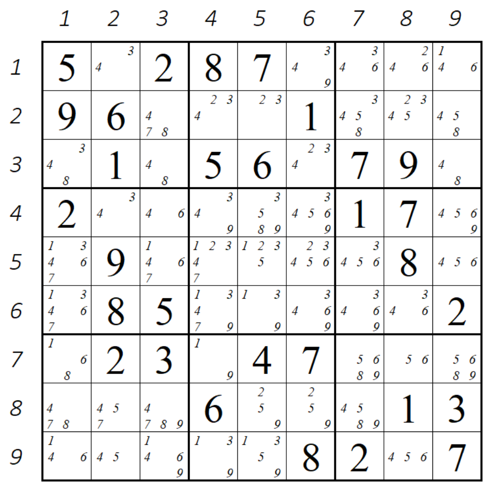
Due to the relative randomness in choosing the successors, it does not seem this method would be as reliable as BAC. It is dependent on chance how fast the algorithm would be able to solve a sudoku puzzle, as it could repeatedly happen that the successors are less accurate than the original boards, which means the algorithm would not get closer to a solution. Also, just like with BAC, the more empty squares a puzzle has, the more time the algorithm would likely take, as for each solution there are more possible successors and thus the chance of picking the right one is smaller. Additionally, the sudoku puzzle having more than one solution seems like it could pose a problem for the algorithm. The set of boards may become essentially divided into two groups, the ones that are close to the one solution and the ones that are close to the other. This means that each group gets less successors, so the chance of finding a correct solution is divided across the two groups. This would not actively stop the algorithm from finding a valid solution, although it seems it would decrease the chance of the solution being found in any next generation.

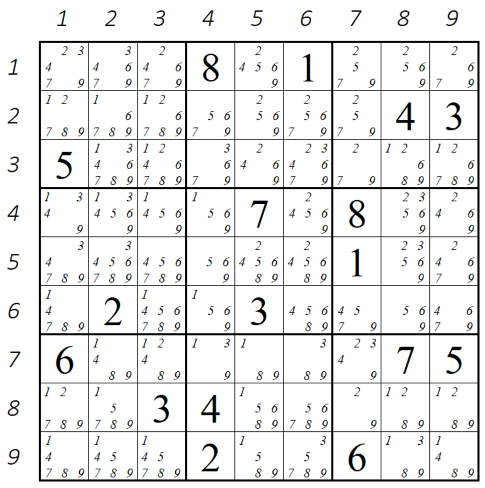
5.2.3 Constraint Programming

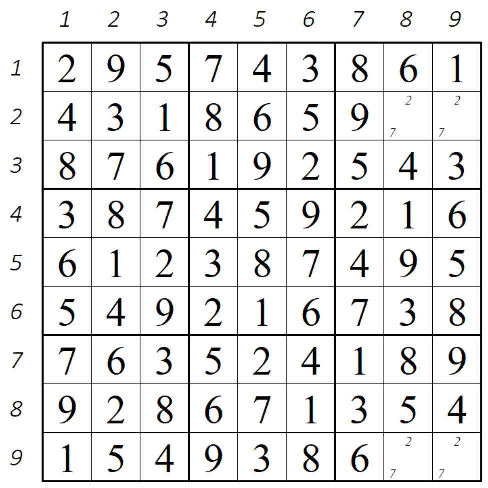
On `gist.github.com/ksurya/3940679`, a code written by Surya Kasturi can be found, which can solve a sudoku puzzle using constraint programming. In this algorithm, the sudoku rules are first written as constraints, rules that a possible solution must adhere to. Then, the algorithm looks for any possible solutions using the `GetSolutions` function in python. In this algorithm, the specific method used to


Test Puzzle 1


Test Puzzle 2


Test Puzzle 3


Test Puzzle 4


Test Puzzle 5

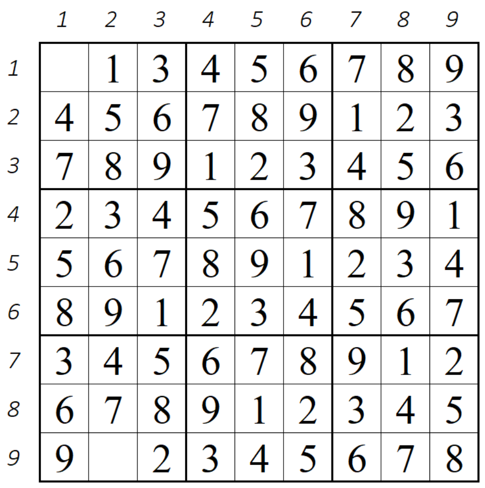

Test Puzzle 6

Figure 14: The given test puzzles

find solutions is the `RecursiveBacktrackingSolver`. This means that this program essentially combines constraints with backtracking. This method will from now on be referred to as CON.

Because of the use of the backtracking solver, this algorithm does not operate very differently from the backtracking algorithm. One important difference to note is that `GetSolutions` function gives *every* solution to a problem, so given a sudoku puzzle with multiple solutions, this algorithm would find each of them. In this case, the algorithm likely has a higher run-time than the backtracking algorithm, as the latter would simply stop after one solution. Outside of this difference, CON will likely perform about as well as the backtracking algorithm.

5.2.4 Test puzzles and predictions

To test these three solving algorithms, the test puzzles shown in Figure 14 will be run through each of them and the resulting solutions and run-times will be compared. These puzzles have been chosen to test the range of applicability of the given codes and to see what would happen if the algorithms get an input they don't expect.

The first three puzzles were chosen to have increasing difficulty. While the first can be solved by applying the Naked/Hidden Singles rule, the second has an instance of an X-Wing, which needs to be found to solve the puzzle. The third puzzle is such that the solving procedure would have to include guessing. The fourth test puzzle contains as little hints as possible, which is 17, as shown in [7]. The fifth and sixth puzzles are chosen not to have a unique solution. For the fifth puzzle, two solutions are possible and in the sixth there is a contradiction in the hints, meaning there is no solution. The relevant data for each test puzzle is given in Table 1. These puzzles will be run through each of the algorithms

Sudoku	# of hints	Hardest rule	# of solutions
Test Puzzle 1	38	Hidden single	1
Test Puzzle 2	24	X-Wing	1
Test Puzzle 3	30	Guessing	1
Test Puzzle 4	17	Hidden Group	1
Test Puzzle 5	77	-	2
Test Puzzle 6	79	-	0

Table 1: Qualities of the test puzzles

ten times, while being timed and the average of these times will be used to compare the algorithms. With the observations made about each algorithm, some predictions can be made about the results. For BAC, the main determining factor for the run-time seems to be the number of empty squares, so TP4 will likely take the most time out of the solvable puzzles. As TP5 has multiple solutions, it will find one of them quite fast, but not both. For TP6, BAC will likely give some sort of error. STO may perform slightly worse with fewer hints, but will probably have similar times for the first four puzzles. For TP5 a solution will likely be found quickly, maybe even in the first generation, and for TP6 STO will run infinitely, as it can never find a candidate board with no errors. CON will probably perform similarly to BAC on the first four puzzles. For TP5 CON will by design give both solutions and for TP6 it will likely return an empty set, as no solutions exist.

5.2.5 Results

The three algorithms given above have been altered slightly before using them in this paper. As STO and CON were written for Python 2.7, they were updated to Python 3 to run them. Additionally, as STO turned out to often get stuck on a certain board when all similar candidates would have more mistakes, it was rewritten slightly to restart when it got stuck. Then, the codes were combined into one program, that can run all methods, where the desired method can be specified in the input. Some lines were also added to allow an input file with the sudoku puzzle. This code is given in Appendix B. Using this code, each method was used ten times to solve the test puzzles. The average times and results can be found in Table 2.

	Average run-time (s)			Result		
	BAC	STO	CON	BAC	STO	CON
Test Puzzle 1	0.372	3.971	0.377	Solved	Solved	Solved
Test Puzzle 2	0.767	-	0.447	Solved	Local Minimum	Solved
Test Puzzle 3	0.339	-	0.321	Solved	Local Minimum	Solved
Test Puzzle 4	224.541	-	30.619	Solved	Local Minimum	Solved
Test Puzzle 5	0.366	0.374	0.328	Solved	Solved	Solved
Test Puzzle 6	0.364	-	-	Not Solved	Error	Error

Table 2: Test results

Firstly, the times for BAC mostly confirm the intuition that the number of empty squares greatly influences the run-time. The sudoku with the least hints, TP4, has the longest run-time out of all the puzzles by a large margin. For the other puzzles, the times are quite similar, where TP2, which also doesn't have too many hints, takes about twice as long as the others. This gives the impression that the number of empty squares and the run-time are exponentially related. This seems reasonable, as each new empty square multiplies the number of states the algorithm has to run through by a factor between 1 and 9. Furthermore, the algorithm finds a solution whenever there is one. In the case of TP6, the algorithm returns the same puzzle that was entered. This happens because the program checks every label for the first empty square, but because no value is valid in this square. After the solver runs through the loop, it returns false and prints the original board.

For STO, the results are quite surprising. It seems that the stochastic algorithm is very unreliable, as most of the puzzles did not return a solution even after running the algorithm for a long time. This happens because the algorithm gets stuck at a board that has some mistakes, but each board that it can be transformed into for the next generation has the same number of mistakes or more. This means that the number of mistakes has reached a *local minimum*. This problem seems to be worsened by the fact that there is no check if the boards are different. Whenever the algorithm gets stuck, all the boards that are generated are very similar and many of them are the same. To try and make the algorithm reach a solution eventually, it was rewritten slightly to restart when it got stuck. However, it still gets stuck on a local minimum almost every time, so it simply keeps restarting. The first and fifth puzzle can be solved somewhat reliably on the first try, although there are cases where the program gets stuck on a local minimum in the first run. The fact that these puzzles can be solved, unlike others, could be because of the high number of hints, which causes less possible different boards and so a lower chance of getting stuck on a local minimum. In the case of TP6, which has no solution, the program results in an error when it has to pick a random square to switch around, as there are no two empty squares in any row. This is not necessarily tied to there not being a solution, so it might behave differently given a sudoku puzzle without a solution that has more empty squares.

The times for CON are quite similar to those for BAC. This is reasonable, as the constraint algorithm uses a backtracking solver. However, for TP2 and TP4 the times are lower than those for BAC. This likely means that the backtracking algorithm CON uses is more efficient than BAC. Also, running SP5 doesn't give multiple solutions. The hypothesis about this in the previous section was likely based on a misunderstanding of the way the `RecursiveBacktrackingSolver` and the constraint library in Python work. However, the solution that is returned is a valid one, the same as the one that is returned by BAC. Lastly, the program returns an error when running SP6. This happens because `getSolutions` returns an empty set, as there are no solutions, and then the algorithm tries to normalize this set to make sure it has the right layout. As the set is empty, it fails at this step and returns an error. This could be fixed by entering an if-clause that returns the empty set if the input to the function is the empty set.

From these results, it seems that in this case, the constraint algorithm is the most reliable program. It returns a solution whenever there is one, although it can take a little longer if there are too few hints, but not nearly as long as the backtracking algorithm takes with the same number of hints. The stochastic search algorithm is completely unreliable, takes longer than both others (if it even returns a solution), and therefore works the least efficiently.

6 Conclusion

The complexity of the sudoku puzzle, as shown in [11] by Yato, is indeed NP-complete and ASP-complete. This original proof used the fact that the Latin Square Completion problem is NP- and ASP-complete to show the same for the Sudoku Puzzle Completion problem, by reducing the first to the second parsimoniously. However, the original proof is short and the context is not very clear. This paper makes is clear, after explaining the proof in more detail, that it is indeed valid and thus the resulting NP- and ASP-completeness of the sudoku puzzle are true.

A different approach to the proof was tried, involving the 3SAT problem, which is known to be NP- and ASP-complete, and reducing it to the Sudoku Puzzle Completion problem. However, in the attempt to construct a different proof, it was found that this approach did not result in a sound proof of the NP-completeness of the sudoku puzzle. The main problem with the attempted proof was the fact that the Boolean variables from 3SAT needed a truth assignment t in order to have an assigned value, but this truth assignment is the result of the 3SAT problem and is therefore unknown during the solving procedure. Nevertheless, the constructs described in Chapter 3 and 4 may be found to be useful to use in a different approach to the proof, or possible for a different goal entirely.

Though the sudoku puzzle is NP-complete, there exist algorithms that can solve a puzzle in quite a short time. Usually, a sudoku puzzle is not very big, ensuring that the computation time stays small. There are quite a few different algorithms that can find solutions to a sudoku puzzle. Among the ones considered in this paper, the constraint solver was the most time-efficient. The stochastic search is unreliable, as it relies on randomness and it can easily get stuck on a result with few mistakes, which is still quite far off from the actual solution. The backtracking algorithm is more reliable, but as it uses brute force, it has a lot of situations to consider, especially when there are a lot of empty squares. Though the constraint solver also uses backtracking to find solutions, it goes about finding these solutions a bit smarter and thus takes less time.

Though the attempt made in this paper at a new proof did not give the desired result, the investigation into different ways to prove the NP- or ASP-completeness of the sudoku puzzle should not be abandoned. Finding new ways to prove certain claims, though seemingly unnecessary, is an important goal in research, if not to make the proof more understandable to those who use the result for their own research, then to simply increase the knowledge about certain subjects. For this particular proof, combining this reduction with some other NP-complete problem may make it successful at proving the NP-completeness of the Sudoku Puzzle Completion problem.

References

[1] C.J. Colbourn. The complexity of completing partial Latin squares. *Discrete Applied Mathematics*, 8(1):25–30, 1984.

[2] C.J. Colbourn, M.J. Colbourn, and D.R. Stinson. The computational complexity of recognizing critical sets. In *Graph Theory Singapore 1983*, pages 248–253. Springer, 1984.

[3] S.A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, 1971.

[4] T. Davis. The mathematics of sudoku. In G.L. Alexanderson, T. Shubin, and D.F. Hayes, editors, *Expeditions in Mathematics.* Mathematical Association of America, 2011.

[5] B. Felgenhauer and F. Jarvis. Mathematics of sudoku I. *Mathematical Spectrum*, 39(1):15–22, 2006.

[6] M.R. Garey and D.S. Johnson. *Computers and intractability.* W.H. Freeman and Company, 1979.

[7] G. McGuire, B. Tugemann, and G. Civario. There is no 16-clue sudoku: Solving the sudoku minimum number of clues problem via hitting set enumeration. *Experimental Mathematics*, 23(2):190–217, 2014.

[8] R.A. Oosterman. Complexity and solvability of nonogram puzzles. Master's thesis, Faculty of Science and Engineering, 2017.

[9] C.H. Papadimitriou. *Computational complexity.* Addison-Wesley, Reading, MA, 1994.

[10] E. Russell and F. Jarvis. Mathematics of sudoku II. *Mathematical Spectrum*, 39(2):54–58, 2006.

[11] T. Yato and T. Seta. Complexity and completeness of finding another solution and its application to puzzles. *Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E86-A(5):1052–1060, 2003.

A Pseudo-code for the proposed transformation

```
function TRANSFORMATION
    INPUT: $C = \{C_1, \ldots, C_m\}$, with all $C_i$ in the form of a set of no more than 3 literals
    set the input length $I$ to $\sum_{i=1}^{m} |C_1|$
    set the sudoku order $n$ to $RoundUp(2 * (I/3))$
    create an empty matrix $S$ of size $n^2 \times n^2$
    for $i$ from 1 to $n^2$ do
        for $j$ from 1 to $n^2$ do
            Assign $S_{ij} = \big(j + ((i-1) \mod n) \cdot n + RoundDown(\frac{i-1}{4})\big) \mod 16$
            if $S_{ij} = 0$ then
                Assign $S_{ij} = n^2$
            else
                Don't change $S_{ij}$
            end if
        end for
    end for
    define $a(s_1, s_2, u, t)$, which returns $s_1$ if $t(u) = T$ and $s_2$ if $t(u) = F$
    define $D$ to contain the $2 \times 2$ blocks in the first sub-block whose first element is on the diagonal
    for $i$ from 1 to $m$ do
        if $|C_i| = 1$ then
            read the one element of the set $C_i$, called $u$
            find a non-blank element $s$ in the first sub-block (if $I \leq 3$, not in $D$)
            replace this element with $a(s, n^2+1, u, t)$ (doesn't have an outcome, so will stay a function)
        else
            if $|C_i| = 2$ then
                read the two elements of the set $C_i$, called $u_1$ and $u_2$
                find a $2 \times 2$ block containing $s_1, s_2, s_3$ non-blank in the first sub-block and not in $D$
                find every instance of $s_1, s_2, s_3$ and make those blank elements
                replace $s_1$ and $s_2$ by $a(s_1, s_3, u_1, t)$ and $a(s_2, s_3, u_2, t)$ respectively
            else[$|C_i| = 3$]
                read the three elements of the set $C_i$, called $u_1, u_2$ and $u_3$
                find a $2 \times 2$ block in $D$ in the first sub-block with non-blank elements $s_1, s_2, s_3, s_4$
                find every instance of the values $s_1, s_2, s_3, s_4$ and make those blank elements
                replace $s_1$ with $a(s_3, s_4, u_1, t)$
                find the first blank element in the same row as $s_2$ and replace it with $a(s_1, s_3, u_2, t)$
                find the first blank element in the same column as $s_3$ and replace it with $a(s_2, s_3, u_3, t)$
            end if
        end if
    end for
```

B Combined code for solvers

```
from random import randint
from math import sqrt
import sys
from functools import reduce
from constraint import *
import numpy

GENERATION_SIZE = 50
BRANCHING_FACTOR = 4

def heuristic_s(board, gridsize=9, blocksize=3):
    collisions = 0
    for i in range(gridsize):
        for j in range(gridsize):
            val = board[i][j]
            for n in range(gridsize):
                if n != i and board[n][j] == val:
                    collisions += 1
            for m in range(gridsize):
                if m != j and board[i][m] == val:
                    collisions += 1
            squareX = j // blocksize
            squareY = i // blocksize
            for n in range(blocksize):
                for m in range(blocksize):
                    if not (blocksize * squareX + m == j or blocksize *
                        squareY + n == i) and board[blocksize * squareY + n
                        ][blocksize * squareX + m] == val:
                         collisions += 1
    return collisions

def deepcopy_board_s(board):
    ret = []
    for row in board:
        ret_row = []
        for elem in row:
            ret_row.append(elem)
        ret.append(ret_row)
    return ret

def generate_successor_s(board, size, fixed):
    choices = [[y for y in x[1] if (x[0], y) not in fixed] for x in
        enumerate([list(range(size)) for x in range(size)])]
    row = randint(0,size-1)
    index1 = randint(0, len(choices[row])-1)
    choice1 = choices[row][index1]
    del choices[row][index1]
    index2 = randint(0, len(choices[row])-1)
    choice2 = choices[row][index2]
    del choices[row][index2]
    ret = deepcopy_board_s(board)
    ret[row][choice2], ret[row][choice1] = ret[row][choice1], ret[row][
        choice2]
    return ret

def generate_board_s(original_board, size, fixed):
    board = deepcopy_board_s(original_board)
    choices = [[y for y in range(1, size+1) if y not in x]
        for x in original_board
    ]
    for i in range(size):
        for j in range(size):
            if (i,j) not in fixed:
                index = randint(0, len(choices[i])-1)
                board[i][j] = choices[i][index]
                del choices[i][index]
    return board

def solver_s(size=9):
    if len(sys.argv)>1:
        fileName = sys.argv[1].upper()
    else:
        print("No␣input␣puzzle␣given")
        exit()
    board = open(fileName).read()
    board = [ int(i) for i in board.split() ]
    board = [board[i * 9:(i + 1) * 9] for i in range((len(board) + 9 - 1)
       // 9 )]
    original_board = board
    fixed_values = set([])
    for i in range(size):
        for j in range(size):
            if original_board[i][j] != 0:
                fixed_values.add((i, j))
    solved = False
    solution = None
    boards = []
    for i in range(GENERATION_SIZE):
        board = generate_board_s(original_board, size, fixed_values)
        boards.append(board)
    boards = [(heuristic_s(board, gridsize=size, blocksize=int(sqrt(size)))
       , board) for board in boards]
    lowest = 0
    m = 1
    while not solved:
        boards.sort(key=lambda x: x[0])
        lowest_ = boards[0][0]
        if lowest_ == lowest:
            m = m + 1
        else:
            m = 1
        lowest = lowest_
        if m > 200:
            print("Local␣minimum")
            solver_s()
            exit()
        boards = boards[:GENERATION_SIZE]
        if boards[0][0] == 0:
            solved = True
            solution = boards[0][1]
            print("")
            print("")
            print("Solution")
            print_board_b(solution)
        else:
            successors = []
            for board in boards:
                for i in range(BRANCHING_FACTOR):
                    successors.append(generate_successor_s(board[1], size,
                        fixed_values))
            for s in successors:
                boards.append((heuristic_s(s, gridsize=size, blocksize=int(
                    sqrt(size))), s))
    return solution

def solve_b(bo):
    find = find_empty_b(bo)
    if not find:
        return True
    else:
        row, col = find
    for i in range(1,10):
        if valid_b(bo, i, (row, col)):
            bo[row][col] = i
            if solve_b(bo):
                return True
            bo[row][col] = 0
    return False

def valid_b(bo, num, pos):
    for i in range(len(bo[0])):
        if bo[pos[0]][i] == num and pos[1] != i:
            return False
    for i in range(len(bo)):
        if bo[i][pos[1]] == num and pos[0] != i:
            return False
    box_x = pos[1] // 3
    box_y = pos[0] // 3
    for i in range(box_y*3, box_y*3 + 3):
        for j in range(box_x * 3, box_x*3 + 3):
            if bo[i][j] == num and (i,j) != pos:
                return False

    return True

def print_board_b(bo):
    for i in range(len(bo)):
        if i % 3 == 0 and i != 0:
            print("-␣-␣-␣-␣-␣-␣-␣-␣-␣-␣-␣-␣-␣")
        for j in range(len(bo[0])):
            if j % 3 == 0 and j != 0:
                print("␣|␣", end="")
            if j == 8:
                print(bo[i][j])
            else:
                print(str(bo[i][j]) + "␣", end="")

def find_empty_b(bo):
    for i in range(len(bo)):
        for j in range(len(bo[0])):
            if bo[i][j] == 0:
                return (i, j)
    return None

def dataNormalize_c (data):
    sudoku_nums = [ eachPos[1] for eachPos in sorted( data[0].items() ) ]
    sudoku = []
    for step in range(0,81,9):
        sudoku.append(sudoku_nums[step:step+9])
    print_board_b(sudoku)

def sudoku_solve_c ():
    if len(sys.argv)>1:
            fileName = sys.argv[1].upper()
    puzzleNums = open(fileName).read()
    puzzleNums = [ int(eachNum) for eachNum in puzzleNums.split() ]
    sudoku = Problem( RecursiveBacktrackingSolver() )
    sudokuIndex = [ (row, col) for row in range(9) for col in range(9) ]
    for eachIndex,eachNum in zip(sudokuIndex, puzzleNums):
        if eachNum == 0:
            sudoku.addVariable(eachIndex, range(1,10) )
        else:
            sudoku.addVariable(eachIndex, [eachNum] )
    var = 0
    for aCount in range(9):
        rowIndices = [ (var, col) for col in range(9) ]
        sudoku.addConstraint( AllDifferentConstraint(), rowIndices )
        colIndices = [ (row, var) for row in range(9) ]
        sudoku.addConstraint( AllDifferentConstraint(), colIndices )
        var+=1
    rowStep = 0
    colStep = 0
    while rowStep < 9:
        colStep = 0
        while colStep < 9:
            boxIndices = [ (row, col) for row in range(rowStep, rowStep+3)
                \
                                for col in range(colStep, colStep+3)]
            sudoku.addConstraint( AllDifferentConstraint(), boxIndices )
            colStep+=3
        rowStep+=3
    return sudoku.getSolutions()

if __name__ == "__main__":
    if len(sys.argv)>1:
        if len(sys.argv)>2:
            method = sys.argv[2].upper()
            if method == "BAC":
                if len(sys.argv)>1:
                    fileName = sys.argv[1].upper()
                else:
                    print("No␣input␣puzzle␣given")
                    exit()
                board = open(fileName).read()
                board = [ int(i) for i in board.split() ]
                board = [board[i * 9:(i + 1) * 9] for i in range((len(board
                    ) + 9 - 1) // 9 )]
                print("Backtracking")
                if solve_b(board):
                    print_board_b(board)
                else:
                    print("No solution found")
                    exit()
            elif method == "STO":
                print("Stochastic Search")
                solver_s()
            elif method == "CON":
                print("Constraint Programming")
                dataNormalize_c( sudoku_solve_c() )
            else:
                print("Invalid method. Valid inputs are BAC, CON or STO")
        else:
            print("No method specified")
            exit()
    else:
        print("No input puzzle given")
```