# Iterative Decoding for Sudoku and Latin Square Codes

Edwin Soedarmadji and Robert J. McEliece

California Institute of Technology, MC 136-93
Pasadena CA 91125, USA
*edwin@systems.caltech.edu*

**Abstract**

This paper presents a new class of $q$-ary erasure-correcting codes based on Latin and Sudoku squares of order $q$, and an iterative decoding algorithm similar to the one used for the Low Density Parity Check code. The algorithm works by assigning binary variables to the $q$-ary values, and by generalizing the definition of parity check operation to represent the constraints that define the Latin and Sudoku squares.

## 1 Introduction

Latin square is a $q \times q$ array of numbers that meets two conditions: the numbers 1 to $q$ appear only (1) once on each row and (2) once on each column. A Latin square is called a Sudoku square if $q = r^2$ for some integer $r$ and it meets an additional condition: (3) if the square is divided into $r^2$ square blocks of $r \times r$ numbers, each block also contains the numbers 1 to $q$ [1]. Figure 1a shows an example of the $4 \times 4$ Sudoku square.

| | | | |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 3 | 4 | 1 | 2 |
| 2 | 1 | 4 | 3 |
| 4 | 3 | 2 | 1 |

| | | | |
|---|---|---|---|
| 1 | | | |
| | | | 2 |
| | | 4 | |
| | 3 | | |

Fig. 1. A $4 \times 4$ Sudoku square with (a) and without (b) erasures.

One of the interesting properties of a Latin square (and by extension, a Sudoku square) is that the two (or three, for Sudoku) conditions in its definition introduce a large number of interlocking constraints that eliminate many degrees of freedom in assigning the $q^2$ numbers into the array. In fact, these constraints are so restrictive that even when some numbers are removed from an otherwise valid Latin *or* Sudoku (LS, for short) square, often these numbers can be recovered. For example, in figure 1b, the twelve numbers that are removed from figure 1a can be recovered uniquely using the constraints.

This property is the basis for the Sudoku puzzles. A Sudoku (Latin) puzzle is created from a valid Sudoku (Latin) square by removing numbers from the square such that these numbers can be *uniquely* restored (without this restriction, an empty square would qualify as a "puzzle" with many answers).

Each LS puzzle can also be viewed as a $q$-ary codeword of length $q^2$ with erasures. To decode such codewords, in this paper we present a new algorithm similar to the iterative algorithm used to decode the Low Density Parity Check (LDPC) codes [2]. The algorithm first converts the $q$-ary symbols into binary symbols, and the missing numbers into erasures, which are then decoded and corrected. We prove that our algorithm returns the same answer(s) consistently, even under a randomized mode of operation, invariant to the decoding path used. The algorithm is a list decoder: depending on a "decoding radius", it can recursively return from one up to all codewords within the radius.

## 2 Codes, Combinatorial Designs, and Graphs

The $q \times q$ entries of a $q$-ary Latin square $l$ can be mapped into the symbols of a $q$-ary codeword $c$ of length $q^2$. Denote this mapping by $\mathsf{C}$, which maps the set $\mathcal{L}$ of all $q \times q$ Latin squares is mapped into a "$q$-L" code, and the set $\mathcal{S}$ of all $q \times q$ Sudoku squares (where $q = r^2$) into a "$q$-S" code. For brevity, we refer to both codes as the $q$-LS code, or simply the LS code. Under this mapping, missing numbers in the puzzle map to codeword erasures. Solving a puzzle amounts to correcting these erasures and finding the causal codeword.

The entries of $l$ can also be mapped into the labels of the $q^2$ vertices (nodes) of a graph $g$, whose adjacent vertices have different labels. Denote this mapping by $\mathsf{G}$. Two vertices in $g$ are adjacent iff their corresponding entries in $l$ share a row or column. Since both $\mathsf{C}$ and $\mathsf{G}$ are 1-to-1 mappings, their inverse mappings exist. In fact, $\mathsf{GC}^{-1}$ maps a $q$-LS code into its corresponding *graph code*.

Graph codes are among the hottest current research topics in coding theory [2]. Some of them – including many linear ones, and the LDPC codes – perform very close to the Shannon Capacity, an impressive feat considering their conceptual simplicity. The sparsity of LDPC matrices makes graphs the natural choice of representation for analysis and data structure implementation. Formulated as a graph code, the LS codes can be decoded with a variant of the standard iterative decoding algorithms used by other (LDPC) graph codes.

We now give an example for the standard iterative erasure-correction algorithm using the simple (7,4) Hamming code – formulated as an LDPC graph code – whose graph is shown in figure 2. The graph uses Tanner's [3] convention. The seven circular nodes are variable nodes, each containing one of the seven codeword symbols. The three square nodes are check nodes, each representing one of the three parity check equations. Each equation operates on four symbols, as shown by the four outgoing edges.

The iterative erasure correction algorithm is simple: (1) each check node can correct a single erasure, (2) each correction is propagated to other check nodes, which in turn correct their own single erasures, (3) the process continues until all erasures are corrected (unfortunately, the algorithm may reach a "stopping set" when there is no more *single* erasures for the check nodes to correct).

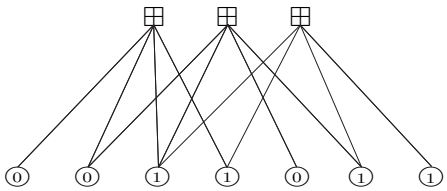


Fig. 2. The Tanner graph for the (7,4) Hamming code

Suppose symbols 4, 5, and 6 in figure 2 are erased. The algorithm recovers these symbols as follows. First, check node 1 determines that symbol 4 must be a one and make a correction, which is propagated to check node 3, which then determines that symbol 6 must be a one, which is propagated to check node 2, which determines that symbol 5 must be a zero. However, the readers can verify that if symbols 2, 4, and 6 are erased, our algorithm encounters a stopping set. This algorithm can correct any double erasures and many triple erasures (exceeding simple decoders that only corrects double erasures).

To use this algorithm for decoding LS codes, we have to make several changes. Why? First, LDPC codes are binary, whereas LS codes are $q$-ary. Second, the check nodes in LDPC codes require the variable nodes to contain an even (or odd) number of ones. In LS codes, only one variable node can contain a value of one. The next section describes these changes through a simple example.

## 3 Decoding Example

For economy of presentation, the example is drawn from the smallest non-trivial $4 \times 4$ Sudoku code. Figure 3a shows the Sudoku square $S$ with $q = 4$ (and $r = 2$) that we showed earlier in figure 1. The entries $S_{ij}$ of $S$ contain the values from 1 to $q$ that satisfy the Sudoku constraints. For example, $S_{14} = 4$ and $S_{44} = 1$. To the right of $S$ is an $r \times r$ binary subarray $S'_{14}$ corresponding to $S_{14}$. Each $S_{ij}$ has a corresponding subarray $S'_{ij}$ that contains $q$ cells $S'_{ijk}$, with $k = 1 \ldots q$. Cell $k = 1$ is on the upper left corner of the subarray and cell $k = q$ is on the lower right corner. The cells $S'_{ijk}$ contain all zeros except where $k = S_{ij}$. Replacing the values $S_{ij}$ in $S$ with their $r \times r$ binary subarrays $S'_{ij}$ produces a new $qr \times qr$ square $S'$ with $(qr)^2 - q^2$ zeros and $q^2$ ones. Performing this operation on the $S$ shown in figure 3a produces $S'$ shown in figure 3b.

Figure 4a is derived from figure 3a, showing only $S_{11}$ and the other entries of $S$ related through the Sudoku constraints. Each $S_{ij}$ obeys the three constraints that prevent duplicates (1) across any row (2) down any column (3) in the

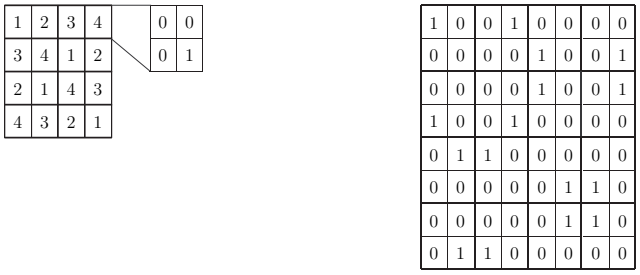

Fig. 3. The (a) $q$-ary and (b) binary version of the $4 \times 4$ Sudoku square

same $r \times r$ block (Latin squares use the first two constraints). Since $S_{11}$ in $S$ is related to $S'_{11}$ in $S'$, instead of the $q$-ary values, figure 4b shows $S'_{111}$ and the other binary values $S'_{ijk}$ related through the constraints.

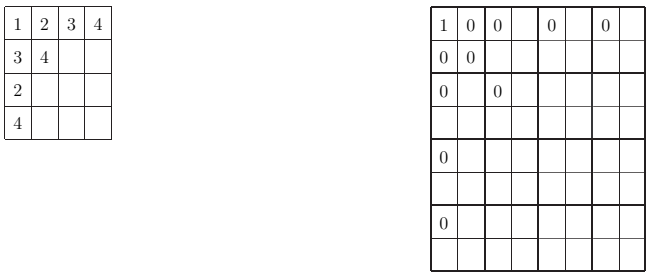

Fig. 4. The entries related to $S'_{111}$ through the constraints

Figure 4b gives away the algorithm: the entry $S'_{111}$ is controlled by four constraints: $h_h$, $h_v$, $h_b$, and $h_c$. The subscripts $_{\{h,v,b,c\}}$ indicate that the constraint is operating horizontally, vertically, in a block, and in a cell. Each constraint $h_{\{h,v,b,c\}}$ operates on a set of cells $V(h_{\{h,v,b,c\}})$. For example, in figure 4

$$V(h_h) = \{S'_{111}, S'_{121}, S'_{131}, S'_{141}\}, \qquad V(h_v) = \{S'_{111}, S'_{211}, S'_{311}, S'_{411}\},$$
$$V(h_b) = \{S'_{111}, S'_{121}, S'_{211}, S'_{221}\}, \qquad V(h_c) = \{S'_{111}, S'_{112}, S'_{113}, S'_{114}\}.$$

Each of the other $S'_{ijk}$'s also has its own constraints $h_h$, $h_v$, $h_b$, and $h_c$ (possibly different from $S'_{111}$'s). Denote by $H_h(v)$, $H_v(v)$, $H_b(v)$, and $H_c(v)$ the horizontal, vertical, block, and cell constraints controlling $v$, and $H(v) = H_h(v) \cup H_v(v) \cup H_b(v) \cup H_c(v)$. How many such constraints are there? Let us count the horizontal constraints first. For a fixed $i$ and $k$, $H_h(S'_{ijk}) = H_h(S'_{ij'k})$, therefore for all $i$ and $k$, we have $q^2$ horizontal constraints. The same is true for the vertical constraints. There are $q$ blocks, each with $q$ constraints for $k = 1 \ldots q$, for a total of $q^2$ constraints. Finally, each of the $q^2$ cells has its own constraint. Thus, $4q^2$ constraints control the $q^3$ entries of $S'$.

Figure 5 shows the four check nodes $h_h$, $h_v$, $h_b$, and $h_c$ for $S'_{111}$. The eleven variable nodes are shown underneath the circles labeled with their $i, j$, and $k$'s. From figure 4, $S'_{111} = 1$ and 0 for other values of $i, j$, and $k$. Each variable node $S'_{ijk}$ is entangled in the same constraining structure which enforces the integrity of the codeword: i.e., erasing one symbol affects several check nodes.

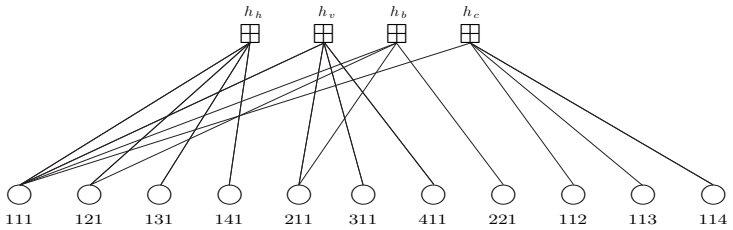


Fig. 5. The check equations $h_h$, $h_v$, $h_b$, and $h_c$ acting on $S'_{111}$

We now provide a our own definition for the check nodes, different from the standard LDPC's. Suppose that $h$ operates on a set of $q$ variable nodes $V(h)$, or $V$ for short. Denote by $V_1(h)$ the set of variable nodes with ones, and by $V_0(h)$ those with zeros. Likewise, denote by $V_e(h)$ the set of variable nodes with erasures. For any $h$, $|V_e \cup V_0 \cup V_1| = |V| = q$. Define $\mathsf{H}$ as a function of $h$ that returns four possible values: *error*, *valid*, *pause*, or *solve* (or $e$, $v$, $p$, and $s$). The function $\mathsf{H}(h)$ reports the *state* of $h$. When $\mathsf{H}(h)$ returns an *error*, we say that $h$ is in the *error* state, or $h$ is an *error* node. Whenever $V(h)$ is updated, $h$ is updated unless it is an *error* node.

$$\mathsf{H}(h) = \begin{cases} \textit{error} & \text{if } |V_1| > 1 \text{ or } |V_0| = q \\ \textit{valid} & \text{if } |V_1| = 1 \text{ and } |V_0| = q - 1 \\ \textit{pause} & \text{if } |V_e| > 1 \text{ and } |V_1| = 0 \\ \textit{solve} & \text{if } |V_e| = 1 \text{ or } |V_1| = 1. \end{cases} \tag{1}$$

If $S$ is a valid Sudoku square (and thus a valid codeword), then all its $4q^2$ check nodes are *valid* nodes. As entries are erased, some affected check nodes become *solve* nodes. As more entries are erased, the number of *pause* nodes increase. A decodeable codeword (or a solvable puzzle) leaves enough unerased entries to recover the numbers uniquely. Erasure correction starts from the *solve* nodes, iteratively attempting to convert all check nodes to *valid* nodes.

Figures 6a and 6b show a codeword $S$ and its binary array $S'$ with $qr \times qr$ variable nodes. Surrounding $S'$ are four groups of $4q^2$ check nodes. Each block has $q$ copies of $r \times r$ subarrays, each indexed by $k = 1 \ldots q$. Directly above $S'$ is the $H_v$ group. The group has four subarrays $j = 1 \ldots 4$, from left to right. The $k$th cell in subarray $j$ contains $H_v(S'_{1jk})$. To the right of $S'$ is the $H_h$ group with its four subarrays $i = 1 \ldots 4$, from top to bottom. The $k$th cell in subarray $i$ contains $H_h(S'_{i1k})$. To the right is the square $H_b$ group with its subarrays arranged from top left corner to the bottom right corner. The $k$th node in subarray $i$ controls all the $k$th cells in group $i$ of $S'$. Finally, we have the $H_c$ group, whose $(i, j)$th *cell* contains $H_c(S'_{ijk})$. Having established these definitions, the decoding process can now start.

Figure 6b shows the initial states of all the check nodes. In this figure, there is no *valid* node, only *pause* nodes and *solve* nodes. For example, $H_h(S'_{411})$ and $H_h(S'_{413})$ are both *solve* nodes because $S'_{423}$ and $S'_{441}$ contain ones.

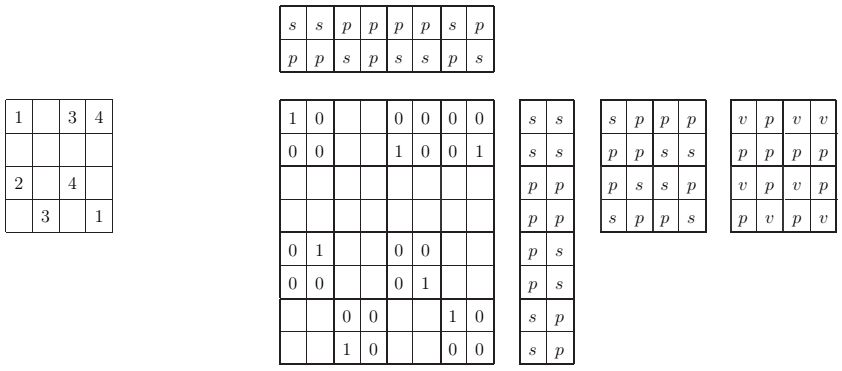


Fig. 6. The puzzle (a) and its variable nodes and check nodes (b)

In the first iteration, all *solve* nodes correct their single erasures, update their states to *valid*, and propagate the corrections to other check nodes. The result is shown in figure 7. The set of *solve* and *valid* nodes grows, while the set of *pause* nodes shrinks. All erasures are removed in three iterations.

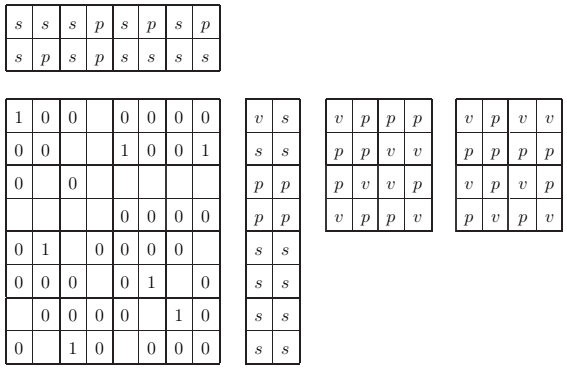


Fig. 7. The puzzle after the first iteration

The iterative algorithm we just described inherits the same limitations as the one used for decoding erased LDPC codewords: some codeword (and thus puzzle) contains a stopping set that stops the algorithm prematurely.

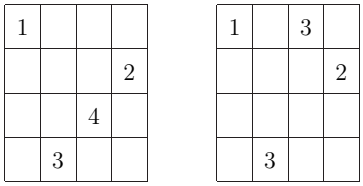


Fig. 8. A square without and with stopping set

For example, the erasure pattern in figure 8a poses no problem, but the one in figure 8b has a stopping set. Once stopped, the algorithm can continue by selecting from the ($q$-ary) $S$ the block, row, or column with the fewest number of erasures $e$ and try all the $e!$ possible completion configurations. For each configuration, the algorithm runs until it encounters an error.

## 4 Algorithm

In this section we provide a detailed description of the two procedures that form our iterative decoding. The procedure in listing 1 iteratively corrects as many erasures as possible given a starting *context* denoted by $X = (S, V, E, P, C)$, defined as a collection of four lists $S$, $V$, $E$ and $P$ which contain *pointers* to the *solve*, *valid*, *error*, and *pause* nodes, respectively, along with the list $C$ that contains the values (zeros, ones, and erasures) of the $q^3$ variable nodes. In listing 1, the notation $V \leftarrow h$ indicates that a check node $h$ is moved from its current list into $V$. The notation $H \leftarrow S$ means the whole content of the list $S$ is moved into the list $H$.

The procedure SOLVE simply calculates a modified context and a return value. It does not handle a stopping set. A very important question is: can we prove that SOLVE returns a "unique" answer regardless of the order in which the elements $h \in H$ are processed on line 4 of listing 1?

**Listing 1** Procedure for iteratively removing erasures

```
1: procedure SOLVE (S, V, E, P, C)
2:     while S ≠ ∅ do                          ▷ Iterate over all solve nodes
3:         H ← S                               ▷ Move the solve nodes to a local list
4:         for all h ∈ H do                    ▷ For all solve nodes
5:             CORRECT( V_e(h) ⊂ C )           ▷ Unerase V_e(h)
6:             V ← h                           ▷ and move to the valid list
7:             H' = {h' | V(h') ∩ V_e(h) ≠ ∅}
8:             for all h' ∈ H' do              ▷ update their states
9:                 {V | S | E | P} ← h'
10:            end for                         ▷ While solve node exists
11:        end for
12:    end while
13:    if E ≠ ∅ then rv ← −1                   ▷ We have an error
14:    else if P = ∅ then rv ← 1               ▷ Solution C is found
15:    else rv ← 0                             ▷ We have run into a stopping set
16:    return ( S, V, E, P, C, rv )
17: end procedure
```

Before proving the uniqueness of the result of SOLVE, let us establish some notations and definitions. Let $V_1$ and $V_2$ denote two sets of variable nodes, whose binary contents are represented by the vectors $\vec{V_1}$ and $\vec{V_2}$, respectively. Let $H_1$ and $H_2$ denote two sets of check nodes whose four possible states are stored in the vectors $\vec{H_1}$ and $\vec{H_2}$, respectively.

**Definition 1** The sets $V_1$ and $V_2$ ($H_1$ and $H_2$) are *equal* if and only if they contain the same variable (check) nodes. The two vectors $\vec{V_1}$ and $\vec{V_2}$ ($\vec{H_1}$ and $\vec{H_2}$) are *equal* if and only if their elements are pairwise identical. □

| | | | *Configuration* 1 | | | | | | | *Configuration* 2 | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | 0 | 12 | | | 21 | | | 0 | 12 | | | 21 | | |
| $h_1$ | $h_2$ | $V_h^-$ | $h$ | $v_1$ | $v_2$ | $h$ | $v_1$ | $v_2$ | $h$ | $h$ | $v$ | $h_2$ | $h$ | $v$ | $h_1$ | $h$ |
| $s_0$ | $s_0$ | $0_+$ | $p$ | 0 | 0 | $e$ | 0 | 0 | $e$ | $s_1$ | 0 | $v$ | $e$ | 0 | $v$ | $e$ |
| $s_0$ | $s_0$ | $0_+e$ | $p$ | 0 | 0 | $s_1$ | 0 | 0 | $s_1$ | $p$ | 0 | $v$ | $s_1$ | 0 | $v$ | $s_1$ |
| $s_0$ | $s_0$ | $0_+e_+$ | $p$ | 0 | 0 | $p$ | 0 | 0 | $p$ | $p$ | 0 | $v$ | $p$ | 0 | $v$ | $p$ |
| $s_0$ | $s_0$ | $0_+1$ | $s_0$ | 0 | 0 | $v$ | 0 | 0 | $v$ | $s_0$ | 0 | $v$ | $v$ | 0 | $v$ | $v$ |
| $s_0$ | $s_0$ | $0_+1e$ | $s_0$ | 0 | 0 | $s_0$ | 0 | 0 | $s_0$ | $s_0$ | 0 | $v$ | $s_0$ | 0 | $v$ | $s_0$ |
| $s_0$ | $s_0$ | $0_+1e_+$ | $s_0$ | 0 | 0 | $s_0$ | 0 | 0 | $s_0$ | $s_0$ | 0 | $v$ | $s_0$ | 0 | $v$ | $s_0$ |
| $s_0$ | $s_1$ | $0_+$ | $p$ | 0 | 1 | $v$ | 0 | 1 | $v$ | $s_1$ | 0 | $e$ | $v$ | 1 | $e$ | $v$ |
| $s_0$ | $s_1$ | $0_+e$ | $p$ | 0 | 1 | $s_0$ | 0 | 1 | $s_0$ | $p$ | 0 | $e$ | $s_0$ | 1 | $e$ | $s_0$ |
| $s_0$ | $s_1$ | $0_+e_+$ | $p$ | 0 | 1 | $s_0$ | 0 | 1 | $s_0$ | $p$ | 0 | $e$ | $s_0$ | 1 | $e$ | $s_0$ |
| $s_0$ | $s_1$ | $0_+1$ | $s_0$ | 0 | 1 | $e$ | 0 | 1 | $e$ | $s_0$ | 0 | $e$ | $e$ | 1 | $e$ | $e$ |
| $s_0$ | $s_1$ | $0_+1e$ | $s_0$ | 0 | 1 | $e$ | 0 | 1 | $e$ | $s_0$ | 0 | $e$ | $e$ | 1 | $e$ | $e$ |
| $s_0$ | $s_1$ | $0_+1e_+$ | $s_0$ | 0 | 1 | $e$ | 0 | 1 | $e$ | $s_0$ | 0 | $e$ | $e$ | 1 | $e$ | $e$ |
| $s_1$ | $s_0$ | $0_+$ | $p$ | 1 | 0 | $v$ | 1 | 0 | $v$ | $s_1$ | 1 | $e$ | $e$ | 0 | $e$ | $e$ |
| $s_1$ | $s_0$ | $0_+e$ | $p$ | 1 | 0 | $s_0$ | 1 | 0 | $s_0$ | $p$ | 1 | $e$ | $s_1$ | 0 | $e$ | $s_1$ |
| $s_1$ | $s_0$ | $0_+e_+$ | $p$ | 1 | 0 | $s_0$ | 1 | 0 | $s_0$ | $p$ | 1 | $e$ | $p$ | 0 | $e$ | $p$ |
| $s_1$ | $s_0$ | $0_+1$ | $s_0$ | 1 | 0 | $e$ | 1 | 0 | $e$ | $s_0$ | 1 | $e$ | $v$ | 0 | $e$ | $v$ |
| $s_1$ | $s_0$ | $0_+1e$ | $s_0$ | 1 | 0 | $e$ | 1 | 0 | $e$ | $s_0$ | 1 | $e$ | $s_0$ | 0 | $e$ | $s_0$ |
| $s_1$ | $s_0$ | $0_+1e_+$ | $s_0$ | 1 | 0 | $e$ | 1 | 0 | $e$ | $s_0$ | 1 | $e$ | $s_0$ | 0 | $e$ | $s_0$ |
| $s_1$ | $s_1$ | $0_+$ | $p$ | 1 | 1 | $e$ | 1 | 1 | $e$ | $s_1$ | 1 | $v$ | $v$ | 1 | $v$ | $v$ |
| $s_1$ | $s_1$ | $0_+e$ | $p$ | 1 | 1 | $e$ | 1 | 1 | $e$ | $p$ | 1 | $v$ | $s_0$ | 1 | $v$ | $s_0$ |
| $s_1$ | $s_1$ | $0_+e_+$ | $p$ | 1 | 1 | $e$ | 1 | 1 | $e$ | $p$ | 1 | $v$ | $s_0$ | 1 | $v$ | $s_0$ |
| $s_1$ | $s_1$ | $0_+1$ | $s_0$ | 1 | 1 | $e$ | 1 | 1 | $e$ | $s_0$ | 1 | $v$ | $e$ | 1 | $v$ | $e$ |
| $s_1$ | $s_1$ | $0_+1e$ | $s_0$ | 1 | 1 | $e$ | 1 | 1 | $e$ | $s_0$ | 1 | $v$ | $e$ | 1 | $v$ | $e$ |
| $s_1$ | $s_1$ | $0_+1e_+$ | $s_0$ | 1 | 1 | $e$ | 1 | 1 | $e$ | $s_0$ | 1 | $v$ | $e$ | 1 | $v$ | $e$ |

**Table 1:** The first two configurations

Suppose $h_1, h_2 \in H$ are two *solve* nodes waiting to be processed on line 4. Denote by $V_1 = V_e(h_1)$ and $V_2 = V_e(h_2)$ the sets of erasures connected to (and eventually corrected by) $h_1$ and $h_2$. Denote by $H_1 = H(V_1)$ and $H_2 = H(V_2)$ the check nodes connected to $V_1$ and $V_2$; by $\vec{V}_{1^0}$ and $\vec{V}_{2^0}$ the contents of $\vec{V}_1$ and $\vec{V}_2$ *before* $h_1$ and $h_2$ are processed on line 4; by $\vec{V}_{1^{12}}$ and $\vec{V}_{2^{12}}$ the contents *after* $h_1$ and $h_2$ are processed (in that order), and by $\vec{V}_{1^{21}}$ and $\vec{V}_{2^{21}}$ the contents *after* $h_2$ and $h_1$ are processed. Denote by $\vec{H}_{1^0}$, $\vec{H}_{2^0}$, $\vec{H}_{1^{12}}$, $\vec{H}_{2^{12}}$, $\vec{H}_{1^{21}}$ and $\vec{H}_{2^{21}}$ the counterparts for $\vec{H}_1$ and $\vec{H}_2$. Finally, define $V_\cap = V_1 \cap V_2$ and $H_\cap = H_1 \cap H_2$, along with $\vec{V}_\cap$, $\vec{H}_\cap$, $\vec{V}_{\cap^0}$, $\vec{H}_{\cap^0}$, $\vec{V}_{\cap^{12}}$, $\vec{H}_{\cap^{12}}$, $\vec{V}_{\cap^{21}}$ and $\vec{H}_{\cap^{21}}$.

**Definition 2** The procedure SOLVE returns a *unique* answer with respect to $h_1$ and $h_2 \in H$ on line 4 in SOLVE if and only if on line 11, these conditions are met: (a) both $\vec{H}_{\cap^{12}}$ and $\vec{H}_{\cap^{21}}$ contain at least one *error* node, or (b) $\vec{H}_{\cap^{12}} = \vec{H}_{\cap^{21}}$ and $\vec{V}_{\cap^{12}} = \vec{V}_{\cap^{21}}$. The procedure SOLVE returns a *unique* answer if it returns a unique answer with respect to any pair $h_1$ and $h_2 \in H$. □

**Theorem 3** The procedure SOLVE returns a unique answer.

PROOF: We can prove the theorem for any pair $h_1$ and $h_2$ by considering every possible cardinality and interconnectivity of $V_\cap$ and $H_\cap$ and combinations of their corresponding vector values. Fortunately, we only need to consider three canonical configurations shown in figure 9. Our proof can be extended to other configurations that are unions of these canonical configurations. The middle nodes are in $V_\cap$, and the bottom nodes are in $H_\cap$.

Due to its topology, the third configuration automatically meets the conditions (a) and (b). Table 1 summarizes how the first and second configurations meet conditions (a) and (b). The first two columns in the table are the possible *solve* states for $h_1$ and $h_2$. The notation $s_0$ means that the check node infers that the erased variable node(s) should be zero(s), and vice versa with $s_1$. Other states are abbreviated with the first letter of their names in equation 1.

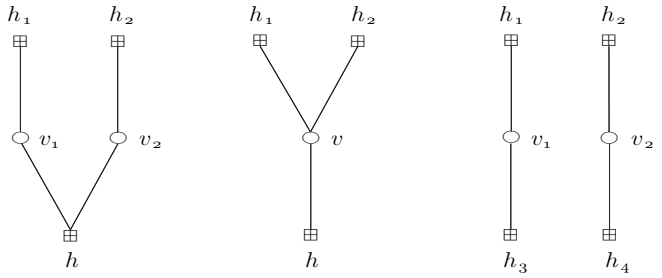


Fig. 9. The three canonical configurations

In figure 9a, $h$ is connected to $v_1$ and $v_2$ and $q-2$ other variable nodes not shown on the diagram. In figure 9b, $h$ is connected to $v$ and $q-1$ other variable nodes. Initially, $v_1$, $v_2$, and $v$ all contain erasures (otherwise $h_1$ and $h_2$ are not *solve* nodes). Denote by $V_h^-$ the $q-2$ (or $q-1$) variable nodes connected to $h$. The vector $\vec{V}_h^-$ is the initial state of $h$ (columns 4 and 11). Column 3 of table 1 describes $\vec{V}_h^-$ in shorthand notation. The + sign after a 0 means "one or more" and a + after an $e$ (for erasure) means "two or more". For example, $0_+1$ means $h$ is connected to $\vec{V}_h^-$ with one or more variable nodes with zeros, one node with a one, plus $v_1$ and $v_2$ in figure 9a (or plus $v$ in figure 9b).

The column groups labeled 12 (or 21) shows the final contents of $\vec{H}_\cap$ and $\vec{V}_\cap$ after $h_1$ and $h_2$ (or $h_2$ and $h_1$) are processed, in that order. For the second configuration, the intermediate state of $h_2$ (or $h_1$) right after $h_1$ (or $h_2$) is processed is also shown because in the second configuration, any change by $h_1$ (or $h_2$) on $v$ directly affects $h_2$ (or $h_1$), raising the possibility of an error at this step. Using definition 2, the procedure SOLVE returns a unique answer if for each configuration: (a) both the 12 and 21 groups contain at least one *error* state, or (b) the contents of the 12 and 21 groups are identical. Comparing columns 5-7 to 8-10 (or 12-14 to 15-17) in table 1 confirms that configurations 1 and 2 meet conditions (a) and (b). Thus SOLVE returns a unique answer.□

Having proven that SOLVE returns a unique answer regardless of the order in which the *solve* nodes are processed, we address the issue of handling the stopping sets. SOLVE is called (recursively) by another procedure called DECODE, shown in listing 2 below. The recursion is initiated by executing DECODE($X$,1). The parameter *maxsol* controls the maximum number of results returned by DECODE, effectively making it a variable *list decoder*.

**Listing 2** Procedure for recursively removing stopping sets

```
 1: procedure DECODE (X, level)
 2:     if level = 1 then
 3:         Store the received codeword into Y
 4:     else
 5:         x ← row / col / block in X with minimum number of e erasures
 6:         Generate all e! configurations of x and store in Y
 7:     end if
 8:     for all y ∈ Y do
 9:         ( X', rv ) = SOLVE( y )
10:         if rv = 1 then SOL ← SOL ∪ C
11:         if |SOL| > maxsol then exit
12:         if rv = 0 then DECODE( X', level + 1 )
13:     end for
14: end procedure
```

Line 5 requires explanation: the row, column, block, and erasures mentioned there refers to the $q$-ary $S$, not the binary $S'$. For instance, if the first row has only three erasures, while the other rows, columns, and blocks have more erasures, then $x$ refers to the first row. If $q = 9$ and the missing numbers in $x$ are 3, 4, and 7, then there are 3! possible ways to place these numbers into $x$, each potentially leading to a solution, which is stored in $Y$. Each $y \in Y$ is then passed as a context to SOLVE, which tries to find a solution within $y$.

In conclusion, together SOLVE and DECODE form the algorithm to decode (and solve) the Latin and Sudoku codes (and puzzles). We proved that the algorithm returns consistent solutions regardless of the path taken to compute them. The interesting question for future research is, can this decoding algorithm be adapted to solve Sudoku puzzles with symbol errors (instead of erasures)?

## References


[1] Contributors, "Sudoku", *Wikipedia, The Free Encyclopedia*, [January 2006] `http://en.wikipedia.org/w/index.php?title=Sudoku`

[2] D.J.C. MacKay, *Information Theory, Inference, and Learning Algorithms*, Cambridge University Press, 2003.

[3] R.M. Tanner, "A Recursive Approach to Low Complexity Codes", *IEEE Transactions on Information Theory*, **IT** vol. 27, no. 5, pp. 533–547, Sept 1981.