An investigation of SUDOKU-inspired non-linear codes with local constraints

Jossy Sayir and Joned Sarwar
University of Cambridge, U.K.

***Abstract*—Codes with local permutation constraints are described. Belief propagation decoding is shown to require the computation of permanents, and trellis-based methods for computing the permanents are introduced. New insights into the asymptotic performance of such codes are presented. A universal encoder for codes with local constraints is introduced, and simulation results for two code structures, SUDOKU and semi-pandiagonal Latin squares, are presented.**

I. Introduction

The codes we investigate are sets of codewords that satisfy a number of local non-linear constraints. As such, they are described by factor graphs just like low-density parity-check (LDPC) codes. Unlike LDPC codes for which the constraint nodes enforce linear constraints over fields, in the codes treated here the constraint nodes enforce non-linear constraints of the SUDOKU type. Each constraint requires that *all variables involved take on different values over the code alphabet.* For example, for a code alphabet $\{0,1,2,3\}$ and a constraint node of degree 4, the variables involved could take on the values $(2,1,0,3)$ or $(3,0,2,1)$ but not $(2,2,0,1)$ because 2 is repeated. Note that it is not imperative that the constraint node degree d_c equals the alphabet size q, but this will be the case in all the structures we will investigate, in which case the constraint can equivalently be described as enforcing that the variables involved take on values over the set of permutations of the code alphabet.

These codes are inspired by SUDOKU puzzles and their study was initially motivated as a tool for teaching belief propagation decoding for LDPC codes via its analogy with solving SUDOKU puzzles. Note that in general, there is no particular reason to arrange the variables in our codewords in a square $q \times q$ grid as they would be in a classic SUDOKU puzzle. The reason for arranging them in this manner is to visualise the constraints corresponding to rows and columns of the square. In this paper, we will look at some regular structures that can be visualised as squares with row-column constraints, but also cover some general structures that are best represented as a common one-dimenstional array of variables constrained by constraint nodes in a factor graph with random connections.

In Section II, we will discuss iterative decoding for codes with permutation constraints and show that the operation in the constraint node is equivalent to computing a set of permanents [1]. We will then show how to compute these permanents using a trellis-based approach, and specialise this approach for decoding over erasure channels. In Section III, we will discuss asymptotic performance analysis for codes with permutation constraints. In Section IV, we will present a universal approach to encoding codes with local constraints and discuss its limitations. In Section V, we will present simulation measurements of the performance of two specific code structures, SUDOKU and semi-pandiagonal Latin squares, over the erasure channel.

Funded in part by the European Research Council under ERC grant agreement 259663 and by the FP7 Network of Excellence NEWCOM# under grant agreement 318306.

II. Iterative decoding, permanents and trellises

A. General belief propagation decoding

Iterative decoding over factor graphs with non-linear constraints follows the same rules as iterative decoding for graphs with linear constraints. The belief propagation algorithm operates Bayesian estimation under the assumption that messages from the graph are independent observations, as described in [2] and references therein. We will express messages as q-ary probability mass functions, although it may be sensible in calculations to transfer them to the logarithmic or log-likelihood ratio domain. For a variable node, the operation is the same as that of variable nodes for linear codes. For a constraint node, the operation is

$$b_{ij} = \xi_i \sum_{(j_1,\ldots,j_i=j,\ldots,j_q)\in S_q} \prod_{k\neq i} a_{kj_k}. \tag{1}$$

where a_{ij} is the j-th component of the i-th incoming message to the constraint node, b_{ij} is the j-th component of the i-th outgoing message of the constaint node, ξ_i is a normalisation constant, and S_q is the symmetric group on $\{1,2,\ldots,q\}$. This can also be written as

$$b_{ij} = \frac{\mathrm{perm}(A_{ij})}{\mathrm{perm}(A)}, \tag{2}$$

where A is the $q \times q$ matrix of incoming messages. For any matrix M, we write $\mathrm{perm}(M)$ for the permanent of M, m_{ij} for the i,j-th element of M, and M_{ij} for the matrix obtained by removing the i-th row and j-th column from M.

Computing a permanent is a high complexity operation. A direct evaluation of

$$\mathrm{perm}(M) = \sum_{(i_1,\ldots,i_q)\in S_q} m_{1i_q} m_{2i_2} \ldots m_{qi_q}$$

requires $(q-1)q!$ multiplications and $q!-1$ additions. The best known efficient algorithm for computing an approximation of the permament of a matrix with positive entries has a probabilistic polynomial complexity, which polynomial is of degree 11 and does not provide any benefits for the alphabet sizes of interest to us, i.e., $q = 9$ or less, perhaps $q = 16$. Still, even for $q = 9$, the number of multiplications is about 3×10^6,

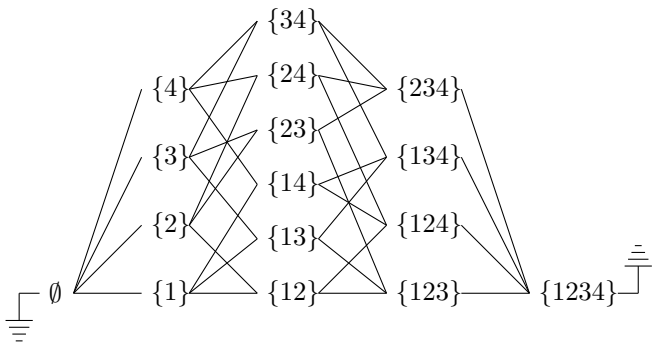

Fig. 1. Trellis-based permanent computation

which would need to be computed $q(q^2+1) = 801$ times at each constraint node at each iteration in order to evaluate (2), resulting in a total number of multiplication, say for 10 iterations for a SUDOKU codeword (81 variables, 27 constraints), of about 6.3×10^{11} per decoding operation. An efficiently programmed simulation of a transmission path for a sufficient number of codewords to measure error performance would hence be beyond the ability of a modern Gflop computer.

A solution to this is to compute the permanent using a trellis. This can be seen as a generalization of Laplace's co-factor expansion for computing the permanent or determinant. In Laplace's co-factor expansion, depending on where we start, we only compute the co-factors in the first row and the permanent of the matrix, whereas to evaluate (2), we need all the co-factors, corresponding to repeating Laplace's expansion starting with every row in turn. Figure 1 represents the model for the trellis-based computation of the permanent and all co-factors of a 4×4 matrix. We start with an empty set at the root of the trellis. Every trellis stage corresponds to a row of the matrix. At row 1, we can pick any of the four elements, resulting in four paths from the empty set to the atomic sets $\{1\}, \{2\}, \{3\}$ and $\{4\}$. At every further stage in the trellis, you can only advance using columns that have not been visited yet, so for example there a three paths leading forward from $\{1\}$ to $\{1,2\}, \{1,3\}$ and $\{1,4\}$. The termination (or "toor") of the trellis corresponds to the set $\{1,2,3,4\}$ where all the columns have been visited.

Multiplying matrix elements on every edge and adding them in the nodes of the trellis will compute the permanent of the matrix and the co-factors of its last row, and is completely equivalent to Laplace's co-factor expansion. This is also equivalent to the forward iteration of the BCJR [3] or forward-backward algorithm. Applying the full BCJR algorithm to the trellis yields the permanent and all the co-factors we need, which result as the sum of the products of the forward sums and the backward sums of all edges corresponding to an element in a row. For example to compute the co-factor M_{23} using the trellis in Figure 1, we need to multiply the forward-sums and the backward-sums in all the transitions corresponding to a 3, i.e., $\{1\}$ to $\{1,3\}$, $\{2\}$ to $\{2,3\}$, and $\{4\}$ to $\{3,4\}$, then sum those to obtain $\mathrm{perm}(M_{23})$.

The number of multiplications in the trellis-based permanent computation excluding the last stage for computing the co-factors is

$$2\sum_{i=1}^{q-1}\binom{q}{i}(q-i) = q(2^q-2),$$

which is still exponential in the alphabet size, but provides a significant reduction in complexity at the alphabet sizes of interest to us. For example, for $q = 9$, we now need to evaluate only 4590 multiplications per constraint node per iteration, or, repeating the evaluation above for decoding a SUDOKU codeword, 1.2×10^6 multiplications per decoding operation.

B. Decoding for the erasure channel

The trellis-based approach described can be further simplified when decoding for an erasure channel. When decoding for a q-ary erasure channel, messages start up as atomic distributions for non-erased positions, and uniform q-ary distributions for erased positions. Through the iterations, messages of varying supports will appear but all messages are always uniform distributions over their support. Hence, we can replace distribution-valued messages by 0/1 indicators or subsets of possible values.

The subset-valued rule for constraint nodes of an erasure decoder has been stated in [2] and we repeat it here. Letting $m_{v\to c}(i) \subseteq \{1,\ldots,q\}$ be the incoming subset message to a constraint node on its i-th edge, the resulting constraint node rule for generating the j-th outgoing message $m_{c\to v}(j) \subseteq \{1,\ldots,q\}$ is

$$m_{c\to v}(j) = \{1,\ldots,q\} - \bigcup_n A_n$$

where A_n is any set such that

$$\exists \mathcal{J} \subset \{1,\ldots,q\} \text{ such that } \begin{cases} j \notin \mathcal{J}, \\ A_n = \bigcup_{j'\in\mathcal{J}} m_{v\to c}(j'), \\ \text{and } \#\mathcal{J} = \#A_n, \end{cases}$$

where $\#S$ denotes the cardinality of the set S. In other words, the rule is that whenever the union of a number of incoming messages has cardinality that number, you can eliminate the members of that union from all remaining outgoing messages. For example, if incoming messages i and j are both $\{1,2\}$, then the values 1 and 2 can be reserved for the variables i and j and can be eliminated from all remaining variables in the constraint. This rule is familiar to those who like solving SUDOKU puzzles.

A direct evaluation of the rule above requires to examine the union of every combination of incoming messages, resulting in a loop over 2^q instances. Again, much can be gained by realizing the constraint node operation on a trellis. Let A be the matrix of incoming messages, where $a_{ij} = 1$ if element j belongs to the subset of incoming message i and $a_{ij} = 0$ otherwise. Using the same trellis structure as we did for the evaluation of the permanent, the following steps are applied:

1) Starting from the root node on a trellis with all edges blanked out, draw a forward path for every edge coming out of a node that satisfies the conditions
   (i) the edge exists in the full trellis; and
   (ii) there is a 1 in the matrix corresponding to the trellis stage and the symbol added by this edge.
2) Prune any paths that did not terminate in the "toor" node.
3) Insert a 1 in the outgoing message wherever there is an edge corresponding to that symbol in the trellis stage for that row.

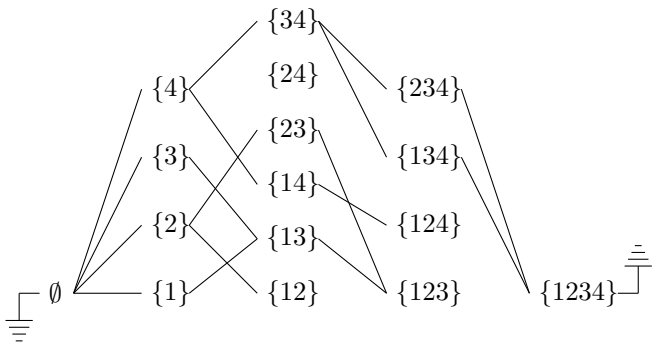

Fig. 2. Forward drawing step of the trellis based constraint node operation

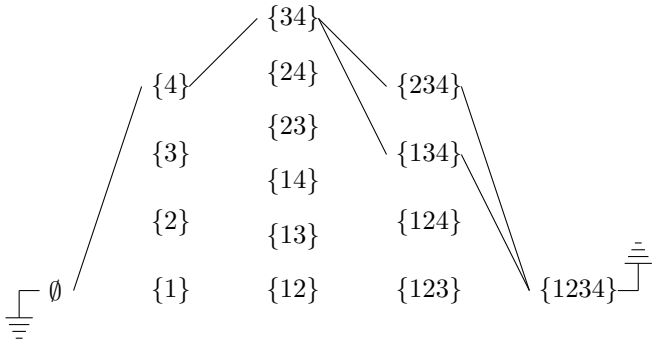

Fig. 3. Pruning step of the trellis based constraint node operation

The method is illustrated for the incoming message matrix

$$A = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{bmatrix}$$

in Figure 2 (forward drawing step) and Figure 3 (pruning step), resulting in the outgoing message matrix

$$B = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{bmatrix}$$

An analysis of the complexity of this method would be difficult as the choice of elementary operations to be counted was unclear to us. Furthermore, the number of operations is not a direct function of the number of ones in a row as it would vary according to the position of the ones. In our simulations, we observed an acceleration of several orders of magnitude by switching from the direct approach of iterating over the 2^q configurations of incoming subsets to the trellis-based approach, and this acceleration was more pronounced as the alphabet size increased. On the other hand, the trellis-based approach is not easy to implement and requires several pages of code, while the direct approach is trivial and only a few lines long in most programming languages.

III. Asymptotic performance analysis and density evolution

Density evolution can be applied to analyse the asymptotic performance of codes with non-linear constraints in a manner similar to LDPC codes. The elements of this analysis have been presented in [2] and we will not dwelve on them here. However, we take this opportunity to correct two inaccuracies in [2] and report on new insights obtained from [4] that cast a new light on the results in [2].

TABLE I. BP thresholds for regular $(3, q)$ graphs

q	3	4	5	6	7	8
Threshold θ	0.8836	0.7251	0.6209	0.5492	0.4965	0.4559
$1 - R_{cf}$	0.6845	0.5692	0.5063	0.4656	0.4365	0.4143

As explained in [2], finding the asymptotic rate of a code with non-linear constraints when the block length goes to infinity while maintaining fixed constraint and variable node degree distributions is not as easy as it is is for linear codes. For linear codes, the so-called "design rate" is a direct function of the degree distributions, and it is easy to show that the true rate for randomly chosen graphs is unlikely to deviate much from the design rate. In [2], we conjectured an expression for a rate estimate but were unable to give it a full justification. As it turns out, this estimate was not accurate at all. We can now make the following precise statement:

Lemma 1: If a factor graph with regular constraint node degree q equal to the alphabet size is a tree, then the rate of the corresponding code tends to $\log((q-1)!)/(q-1)$ as the number of nodes tends to infinity.

Proof: Start drawing the tree from its root with one constraint node and its associated q variables. The combination of variables can take on $q!$ values. Now add constraint nodes to any variable in the tree one at a time. With each addition, the combination of new $q-1$ variables added can take on $(q-1)!$ values. Hence the set of values for a tree with k constraint nodes is $q!((q-1)!)^{k-1}$ for a number of variables $q + (k-1)(q-1)$, which tends to the expression given for $k \longrightarrow \infty$. □

While this statement is correct, it is of little use for estimating the asymptotic rate of the code. We were mislead by the convergence theorem of density evolution, which states that, for a finite number of iterations, the horizon of a variable node converges to a cycle-free graph as the block length grows to infinity. While this is true for a finite number of iterations, it obviously does not remain true when the number of iterations grows to infinity, which would be necessary for the rate of the overall code to converge towards the rate of a tree. We now call this quantity the cycle-free rate R_{cf}.

Furthermore, the threshold calculations for variable degree $d_v = 3$ codes in [2, Table III] were erroneous, and we state the correct values in Table I along with $1 - R_{cf}$. Beyond $q = 8$, it becomes infeasible to compute thresholds using density evolution with our limited computational resources. We note that the thresholds are greater than $1 - R_{cf}$, which goes to show that R_{cf} is a bad estimate of the true rate since we would otherwise be violating Shannon's converse coding theorem.

In [4], Vontobel was able to obtain what he believes to be an accurate estimate of the rate of a regular (d_v, q) factor graph with permutation constraints using the Bethe approximation of the partition function of the factor graph. This approach yields

$$R_{est} = \max\left(0, (d_v/q) \cdot \log_2(q!) - (d_v - 1) \cdot \log_2(q)\right),$$

or using Stirling's approximation $q! \approx \sqrt{2\pi q}(q/e)^q$,

$$R_{est} \approx \max\left(0, \log_2(q \cdot (2\pi q)^{d_v/(2q)}/e^{d_v})\right).$$

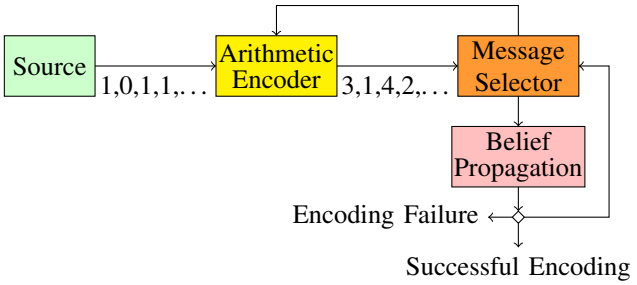

Fig. 4. A universal encoder structure for graphs with local constraints

This estimate is zero for all alphabet sizes q up to and including 11. As a result, the asymptotic analysis of codes with local permutation constraints is caught between a rock and a hard place: there are sub-exponentially many codewords for all alphabet sizes up to 11, but we are unable to perform density evolution for alphabet sizes larger than 8.

IV. Universal encoding and prefix reservation

We now return to finite length codes for which the considerations of the previous section do not apply. The rate of a finite length N code with non-linear constraints is $R = (\log M)/N$, where M is the number of sequences that fulfill the constraints. The number of valid classsical $q \times q$ SUDOKU grids for various q has been counted or estimated, as reported in [5], yielding, for $q = 9$, M_9=6,670,903,752,021,072,936,960 and for $q = 16$, $M_{16} \approx 5.9584 \times 10^{98}$, resulting in rates of $R_9 = (\log_9 M_9)/81 = 0.28$ and $R_{16} \approx 0.32$. The number M_9 was obtained through tedious counting of all valid SUDOKU grids after excluding as many symmetries as possible. This process yields a valid enumeration of SUDOKUs that could potentially be used for encoding purposes.

Nevertheless, we investigate a general structure illustrated in Figure 4 that can be used to encode any sequence with local constraints. It is inspired by Richardson and Urbanke's method to encode LDPC codes [6] using an erasure decoder. However, unlike in [6] it is not possible to start encoding by assigning information symbols to the systematic part of the codeword. There is no equivalent to the systematic property for general non-linear codes, and certainly for permutation constraints there is no part of a codeword beyond a single variable that can be assigned at will without taking constraints into account. Thus, our approach in Figure 4 starts with an empty grid. At every step, belief propagation is used to exclude incompatible values for the undetermined variables. Following that, the message selector takes the first variable whose set of possible values has cardinality $k > 1$, and asks an arithmetic encoder to convert source randomness into a uniform k-ary variable and uses this to determine the value of the free variable. The reason for using an arithmetic encoder is that the cardinality k may vary from one step to the next, while the source has a constant alphabet.

Let us for example encode a 4×4 classic SUDOKU grid. The first decoding operation on the empty grid makes no advance at all, and we begin by requesting a choice of cardinality 4 from the arithmetic encoder to determine the top left variable. Say the value assigned is 3, then the decoder will automatically exclude the value 3 from all remaining variables in the first row, first column, and first 2×2 subsquare. After this, the first available variable is the second element of the first row whose value can be 1, 2 or 4 and hence we request a uniform choice between 3 possibilities from the arithmetic encoder. Say this choice is 3, so we pick the third possible element 4 for our variable. This continues until the grid is filled. With this method, the encoding rate may vary from one codeword to the next and the true rate is the average of the individual codeword rates.

The problem with the method proposed is that it will not always succeed. If we had access to an optimal erasure decoder, the predictions for the possible values of a variable would always be accurate. Belief propagation on the other hand is a sub-optimal erasure decoder that treats each local constraint independently. The consequence of this is that the decoder may for example predict three possible values for a given variable, say 1, 4 and 7 in a 9-ary alphabet, but one of these possibilities, say 1, is in fact illegal as there exists no codeword that combines the determined values with a 1 in this position. An optimal decoder would never propose 1 as a possible value in this context. With a sub-optimal decoder, when a wrong prediction occurs, we are left in a precarious situation. Unlike SUDOKU puzzle solvers who can then simply backtrack and follow one of the other possibilities, we cannot do this because the decoder will never know that there was an illegal option among the 3. Decoding may fail because the decoder misinterprets the mapping of information sequence to codeword, even if the transmitted codeword was decoded correctly. Hence, the encoding process may terminate in an encoding failure, as indicated in Figure 4.

A way around this is to reserve a prefix of the tree of possible codewords for subsequent encoding attempts. For example in a 9×9 SUDOKU, we may exclude the value 9 for the first variable, effectively constraining the encoder to make a choice between 8 possibilities for this variable. Should the encoding process result in an encoding failure, all source symbols consumed in this encoding operation are returned to the source, and encoding begins again but is preceded by the reserved prefix 9. The decoder recognises that the reserved prefix has been used, hence knows that no source symbol has been consumed to produce the first code symbol. This techique can be applied recursively for example by reserving a prefix within the prefix for the case when the encoder fails again, and so forth until the probability of an encoding failure becomes negligible. Furthermore, although the prefix may appear wasted in terms of rate, it can in theory be balanced exactly against the probability of failure so that the resulting encoder is rate optimal. Another way of seeing this is that the sub-optimal erasure decoder overestimates the size of the code tree. We compensate for this by reducing the tree by the reserved prefix.

This prefix reservation approach is practical if the probability of encoding failure is quite small. We will see in the next section that the probability of an encoding failure for 9×9 SUDOKU is about 1.6%, making them very suited for the encoding technique we described in this section.

V. Simulation of SUDOKU and semi-pandiagonal squares

There are a number of regular graph structures for finite code design that we have considered in our work:

Latin squares: q^2 variables over an alphabet of size q. The constraints are best visualised by arranging the variables as a $q \times q$ square and correspond to all rows and columns.

SUDOKU: q^2 variables over an alphabet of size q, where $\sqrt{q}$ must be an integer. A Latin square with extra constraints corresponding to the q subsquares of dimension $\sqrt{q} \times \sqrt{q}$.

Pandiagonal Latin squares: a Latin square with further permutation constraints corresponding to the broken right *and left* diagonals [7] $(i, j+i)$ and $(i, j-i-1)$ for $i = 0, 1, \ldots, q-1$. These are also called Knut Vik designs.

Semi-pandiagonal Latin squares: a Latin square satisfying the broken right diagonal $(i, j+i)$ constraints. These have also been called Semi Knut Vik designs in [7].

As shown in [7], there are no semi-pandiagonal Latin squares for q even. We have counted the semi-pandiagonal Latin squares for various q and obtained 3! for $q = 3$, $3 \times 5!$ for $q = 5$, $635 \times 7!$ for $q = 7$, and $489{,}300 \times 9!$ for $q = 9$.

In our simulations, we focused on SUDOKU and on semi-pandiagonal Latin squares, mainly because for $q = 9$, both these correspond to regular (9,3) factor graphs, differing only on the last 9 of 27 constraints. We were surprised to find that, despite these apparent topological similarities, the properties of the code varies greatly. The table below shows the rate and probability of encoding failure for the two cases:

Factor Graph	SUDOKU	Semi-pandiagonal
True Rate	$R = 0.2824$	$R = 0.1455$
Probability of Encoding Failure	0.016	0.9995

We see that semi-pandiagonal Latin squares have a lower rate and an extremely high probability of encoding failure close to 1, making them unsuitable for the type of encoding suggested in the previous section. However, there is a rich literature on enumerating pandiagonal Latin squares [8], [9], [10], [11] and it may be possible to apply some of these techniques to semi-pandiagonal Latin square, thereby providing a method to encode them without using the factor graph or belief propagation on the erasure channel.

The simulated performance of both codes on the erasure channel is shown in Figure 5. The results show that the codes again have very different characteristics, with the semi-pandiagonal Latin square's error performance flattening out while the SUDOKU exhibits a steeper waterfall but overall further from its threshold $1 - R = 0.72$.

VI. Conclusion

We have described non-linear codes with local permutation constraints inspired by SUDOKU puzzles. For the decoder, we showed that permutation constraints require the evaluation of a permanent using a trellis. We specialised this decoder to erasure channels, where the operation becomes a trellis search. For the encoder, we described a universal approach to encoding a code with local constraints, and discussed its limitations when based on a sub-optimal decoder. We also gave some insight regarding asymptotic performance, and simulation results for select finite length codes.

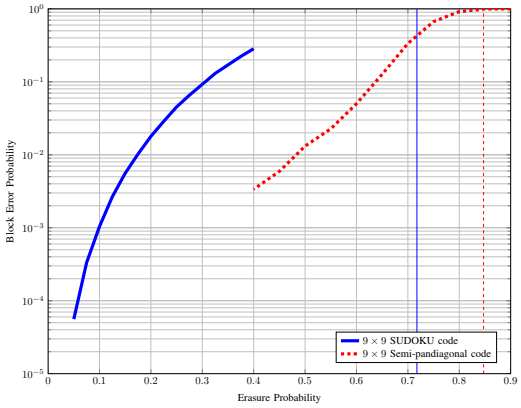

Fig. 5. Block error performance of a SUDOKU and a semi-pandiagonal code. Each data point was averaged over 100 codewords and each codeword was subjected to a sufficient number of erasure sequences to observe at least 100 block errors.

Having started as a tool for teaching belief propagation, the study of codes with non-linear constraints is having unexpected repercussions: it has brought up some interesting technical hurdles, taught us a few things about non-linear constraints and how different they behave from linear constraints, and in the process also highlighted a few properties that we take for granted in linear codes but are in fact quite surprising. In future work, it is worth paying more attention to structures such as pandiagonal Latin squares for which enumerations have been devised. Another line of enquiry are improved decoding rules that combine constraints in belief propagation.

Acknowledgment

The authors wish to thank Gottfried Lechner for suggesting to compute permanents with a trellis, as described in Section II.

References

[1] A.-L. Cauchy, "Mémoire sur les fonctions qui ne peuvent obtenir que deux valeurs égales et de signes contraires par suite des transpositions opérées entre les variables qu'elles renferment," *Journal de l'École Polytechnique*, pp. 91–169, 1815.

[2] C. Atkins and J. Sayir, "Density evolution for sudoku codes on the erasure channel," in *Proc. Int. Symp. on Turbo Codes & Rel. Topics*, Bremen, Germany, Aug. 2014.

[3] L. Bahl, J. Cocke, F. Jelinek, and J. Raviv, "Optimal decoding of linear codes for minimizing symbol error rate," *IEEE Trans. Inf. Theory*, pp. 284–287, Mar. 1974.

[4] P. Vontobel, "Comments on your paper," Sep. 2014, private communication.

[5] "The mathematics of Sudoku," article in Wikipedia. [Online]. Available: http://en.wikipedia.org/wiki/Mathematics_of_Sudoku

[6] T. Richardson and R. Urbanke, "Efficient encoding of low-density parity check codes," *IEEE Trans. Inf. Theory*, no. 2, pp. 638–656, 2001.

[7] A. Hedayat and W. T. Federer, "On the nonexistence of Knut Vik designs for all even orders," *The Annals of Statistics*, no. 2, pp. 445–447, 1975.

[8] A. Hedayat, "A complete solution to the existence and nonexistence of Knut Vik designs and orthogonal Knut Vik designs," *Journal of Combinatorial Theory (A)*, pp. 331–337, 1977.

[9] A. O. L. Atkin, L. Hay, and R. G. Larson, "Enumeration and constructon of pandiagonal Latin squares of prime order," *Comput. Math. Appl.*, no. 2, pp. 267–292, 1983.

[10] J. Bell and B. Stevens, "Constructing orthogonal pandiagonal Latin squares and panmagic squares from modular n-queens solutions," *Journal of Combinatorial Designs*, no. 2, pp. 221–234, May 2007.

[11] V. Dabbaghian and T. Wu, "Recursive construction of non-cyclic pandiagonal Latin squares," *Discrete Mathematics*, pp. 2835–2840, 2013.