COMPUTING PROPERTIES OF INDEPENDENT SETS BY GENERIC PROGRAMMING TENSOR NETWORKS *

XXX† AND YYY‡

Abstract. We introduce a new approach to compute various properties of independent sets by encoding the problems into tensor networks with different tensor element algebra. The type of graph problems that can be computed using this method includes: independence number, the number of independent sets, independence polynomial, maximal independence polynomial, and enumeration of the maximum independence set configurations. Although the computational complexity inevitably scales exponentially with the graph size, the performance of our method using tensor network contraction with generic programming is on par or outperforms state-of-the-art, sophisticated methods; on the other hand, our algorithms are very simple to implement and one can directly utilize recent advances in tensor network contraction techniques. To demonstrate the versatility of this tool, we apply it to a few examples including the calculations of the hard-square constant, Euler characteristics of the independence complex, partition functions, and finite-temperature phase transitions on square-lattice graphs.

Key words. independent set, tensor network, maximum independent set, independence polynomial

AMS subject classifications. 05C31, 14N07

1. Introduction.

[**JG: efficient counting maximal independent sets [24]**] In graph theory and combinatorics, there are many interesting but hard computational problems concerning properties of independent sets. An independent set is a set of vertices in a graph where no two vertices are adjacent to each other. More formally, given an undirected graph $G = (V, E)$, an independent set $I \subseteq V$ is a set that for any $u, v \in I$, there is no edge connecting u and v in G. The problem of finding an independent set of the largest possible size, the maximum independent set (MIS), is a paradigmatic NP-complete problem [18]. The size of the MIS of a graph G is called the independence number, denoted as $\alpha(G) \equiv \max_I |I|$.

Finding $\alpha(G)$ is a hard computational problem; moreover, it is NP-hard to even approximate $\alpha(G)$ within a factor $|V|^{1-\epsilon}$ for an arbitrarily small positive ϵ. Naive exhaustive search for the MIS requires time $O(2^{|V|})$. More efficient exact algorithms include the branching algorithms and dynamic programming. Sophisticated branching algorithms can reduce the base of the exponential time scaling to, e.g., $1.1893^n n^{O(1)}$ [33] [**ST: reading the abstract, I think the algorithm is $1.1996^{|V|}|V|^{O(1)}$. The $1.1893^{|V|}|V|^{O(1)}$ scaling is for graphs with maximum degree 6**]. Dynamic programming approaches [8, 13] works better for graphs with a small treewidth $\mathrm{tw}(G)$; these methods can produce algorithms of complexity $O(2^{\mathrm{tw}(G)}\mathrm{tw}(G)|V|)$.

The independent set problem is closely related to the clique problem and the vertex cover problem [26]; more concretely, an MIS of a graph G corresponds to a maximum clique in the complement graph of G, and the complement vertex set of an MIS corresponds to a minimum vertex cover of the graph G. Together, these problems find a wide range of applications in scheduling, logistics, social network analysis, bioinformatics, wireless networks and telecommunication, map labelling, computer vision, etc. [6, 32]. Besides the standard MIS problem, there are many other interesting problems pertaining to independent sets, such as the number of independent sets, independence polynomial, maximal independence polynomial, and enumeration of optimal and sub-optimal configurations. Some of these problems are of great interest in physics applications such as the hard-core lattice gas model [9, 11] in statistical mechanics and the Rydberg hamiltonian with neutral

---

*

Funding: ...

†XXX (email, website).

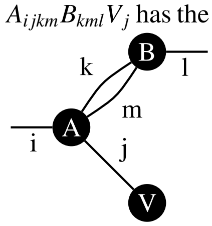
‡yyyyy (yyyy, email).

atoms [29] [**ST: cite experiment when it's ready**]; they can, for example, be used to understand phase transitions and to identify harder graphs in an ensemble of graphs [**ST: cite experiment**].

In this work, we introduce a tensor-network based framework to compute the various properties pertaining to independent sets. We map different problems into generic tensor network contraction with various tensor element algebra. Our approach does not necessarily provide a better time complexity compared to dynamic programming, but the tensor network approach is highly versatile so different problems can be generically solved with minimal implementation efforts. We benchmark our algorithms, the implementation of which benefits from recent advances in tensor-network contraction for the purposes of quantum circuit simulations [16, 28]. Lastly, we provide a few examples and show that the toolbox can be used to calculate the hard-square constant, Euler characteristics of the independence complex, partition functions, and finite-temperature phase transitions on square-lattice graphs.

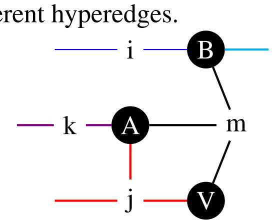
2. Tensor networks. [**ST: I removed the superscript and subscript, since I think it's not necessary to introduce those.**] A tensor network [7, 27] can be viewed as a generalization of matrix multiplication to multi-dimensional tensor contraction. Let A, B be two matrices; the matrix multiplication is defined as $C_{ik} = \sum_j A_{ij} B_{jk}$. Using the Einstein notation, we can write $C_{ik} = A_{ij} B_{jk}$, where j is implicitly summed over. In the standard notation, each index can appear at most twice. A tensor network consists of multiple tensors performing the sum-product operation, i.e., contraction, and the tensor network can be represented as a multigraph with open edges by viewing a tensor as a vertex, a label pairing two tensors as an edge, and the remaining unpaired labels as open edges.

Example 1. A tensor network $C_{il} = A_{ijkm} B_{kml} V_j$ has the following multigraph representation.



One can generalize the tensor network notation by removing the restriction that each index can appear at most twice. The notation can be considered as a generalized Einstein notation, which is sometimes called einsum, sum-product network or factor graph [4] in different contexts. The graphical representation of a tensor network we use in this paper is a hypergraph, where an edge (label) can be shared by an arbitrary number of vertices (tensors).

Example 2. $C_{ijkl} = A_{jkm} B_{mil} V_{jm}$ is a tensor network representing $C_{ijkl} = \sum_m A_{jkm} B_{mil} V_{jm}$ [**ST: I've changed this. please check. The original was $C_{ijk} = A_{jkm} B_{mil} V_{jm}$ and $C_{ijk} = \sum_{ml} A_{jkm} B_{mil} V_{jm}$**]. Its hypergraph representation is shown below, where we use different colors to annotate different hyperedges.



In the main text, we use the generalized tensor network notation. The generalized tensor network can be easily translated to the standard notation by adding identity tensors denoted as δ tensors. The example above can be written as $C_{ijkl} = A_{jkm}B_{mil}V_{jm} = A_{ukm}B_{pil}V_{vq}\delta_{mpq}\delta_{juv}$ in the standard notion. However, by adding the δ tensors, one may unnecessarily increase the contraction complexity of a graph, which we illustrate in Appendix B.

3. Generic programming. [**ST: I suggest we have a section on generic programming, putting both the semiring stuff and the summary table to this section, since generic programming is one of the highlights of this paper as well. I think it doesn't need to be long. Jinguo, could you write one or two paragraphs introducing generic programming?**]

[**ST: Generic programming is ...**] [**ST: dispatch to different types etc. ...**]

In the spirit of generic programming, the elements of the tensors in the tensor network do not always need to be floating point numbers or integers. In fact, for the tensor network contraction to make sense, one only needs the tensor elements to form a commutative semiring. A semiring is a ring without additive inverse, while a commutative semiring is a semiring where the multiplication operation is commutative. To define a commutative semiring with the addition operation $\oplus$ and the multiplication operation $\odot$ on a set R, the following relations must hold for arbitrary three elements $a, b, c \in R$.

$$
\begin{aligned}
(a \oplus b) \oplus c &= a \oplus (b \oplus c) && \triangleright \text{ commutative monoid } \oplus \text{ with identity } \mathbb{0}\\
a \oplus \mathbb{0} &= \mathbb{0} \oplus a = a &&\\
a \oplus b &= b \oplus a &&\\
\\
(a \odot b) \odot c &= a \odot (b \odot c) && \triangleright \text{ commutative monoid } \odot \text{ with identity } \mathbb{1}\\
a \odot \mathbb{1} &= \mathbb{1} \odot a = a &&\\
a \odot b &= b \odot a &&\\
\\
a \odot (b \oplus c) &= a \odot b \oplus a \odot c && \triangleright \text{ left and right distributive}\\
(a \oplus b) \odot c &= a \odot c \oplus b \odot c &&\\
\\
a \odot \mathbb{0} &= \mathbb{0} \odot a = \mathbb{0} &&
\end{aligned}
$$

The requirement of being commutative is for the tensor contraction result to be independent of the contraction order, which could be relaxed in certain cases. In the following sections, we show how to compute the independence polynomial, the maximal independence polynomial, independence number, the number of independent sets, and enumeration of the MIS configurations of a general graph G by designing tensor element types as commutative semirings while keeping the tensor network generic [31]. Table 1 summarizes the problems that can be solved by various tensor element types.

4. Independence polynomial and maximal independence polynomial.

4.1. Independence polynomial. The independence polynomial is an important graph polynomial that contains the counting information of independent sets. It is defined as

$$
I(G, x) = \sum_{k=0}^{\alpha(G)} a_k x^k, \tag{4.1}
$$

where a_k is the number of independent sets of size k in G, and $\alpha(G)$ is the independence number. The total number of independent sets is thus equal to $I(G, 1)$. The problem of

element type	**purpose**
real number	counting all independent sets
tropical number (Eq. (6.3))	finding the independence number
tropical number with counting (Eq. (6.3))	finding the independence number and the number of MISs
tropical number with configurations (Eq. (6.5))	finding the independence number and one MIS
tropical number with sets (Eq. (6.1))	finding the independence number and enumeration of all MISs
polynomial (Eq. (4.4))	computing the independence polynomial exactly
truncated polynomial (Eq. (5.4))	counting MISs and independent sets of size $\alpha(G) - 1$
complex number	fitting the independence polynomial with fast Fourier transform
finite-field algebra (Eq. (4.7))	fitting the independence polynomial exactly using number theory

Table 1: Tensor element types and their purposes in calculating various independent set properties.

computing the independence polynomial belongs to the complexity class #P-hard. One approach to exactly compute the independence polynomial requires a computational time of $O(1.442^{|V|})$ [12][**JG: I am not sure about this complexity, this is based on the naive analysis of theorem 2.2 in [12]**]. There are some interesting works on efficiently approximating the independence polynomial [17], but in this work, we focus on exact methods to compute the independence polynomial.

The independence polynomial is closely related to the matching polynomial, the clique polynomial, and the vertex cover polynomial. In fact, the independence polynomial can be viewed as a generalization of the matching polynomial, since the matching polynomial of a graph G is the same as the independence polynomial of the line graph of G [21]. Thus, our algorithm to compute the independence polynomial can also be used to compute the above graph polynomials. Some properties of the independence polynomial, such as the uni-modality, log-concavity, and roots of the polynomial, are well studied in the literature for special classes of graphs [21]. We hope our algorithms to calculate the independence polynomial can also help further research in these directions.

To compute the independence polynomial of a graph G, we encode it to a tensor network. On the vertex i of the graph G, we place a rank-one vertex tensor, $W(x_i)$, of size 2 parametrized by x_i and a rank-two edge tensor, B, of size 2×2 on an edge (i, j) [**ST: I removed the indices s_i etc, on the right of the equations, since that looks confusing to me. If I understand correctly, s_i, s_j just denotes the index of the vector/matrix.**]

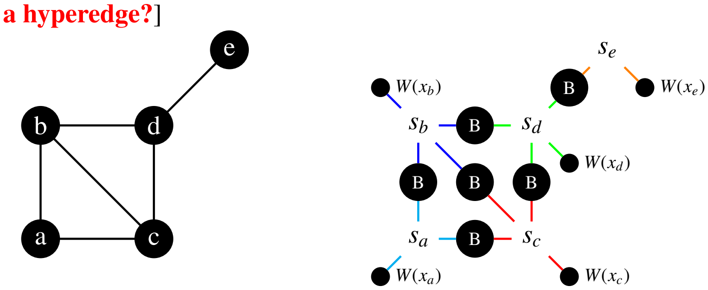
$$W(x_i)_{s_i} = \begin{pmatrix} 1 \\ x_i \end{pmatrix}, \qquad B_{s_i s_j} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \tag{4.2}$$

where the tensor index s_i is a boolean variable; $s_i = 1$ corresponds that the vertex i is in the independent set, and $s_i = 0$ means otherwise. Let us assume the tensor elements are real numbers for now. The edge tensor element $B_{s_i=1,s_j=1} = 0$ encodes the independent set constraint, meaning vertex i and j cannot be both in the independent set if they are connected by an edge (i, j). The contraction of the tensor network representing G gives

$$P(G, \{x_1, x_2, \ldots, x_{|V|}\}) = \sum_{s_1,s_2,\ldots,s_{|V|}=0}^{1} \prod_{i=1}^{|V|} W(x_i)_{s_i} \prod_{(i,j)\in E(G)} B_{s_i s_j}, \tag{4.3}$$

where the summation runs over all vertex configurations $\{s_1, s_2, \ldots, s_{|V|}\}$ and accumulates the product of tensor elements to the scalar output P (see Example 3 for a concrete example). In the special case of $x_i = x$, the contraction result directly corresponds to the independence polynomial. The connection can be understood as follows: the product over vertex tensor elements produces a factor x^k, where $k = \sum_i s_i$ counts the set size, and the product over edge tensor elements gives a factor 1 for a configuration being in an independent set and 0 otherwise. In the implementation of the algorithm, we benefit from recent advances in tensor network contraction, where researchers working on quantum circuit simulation evaluate the tensor network by pairwise contracting tensors in a good heuristic order [16, 28]. A good contraction order can reduce the time complexity significantly, at the cost of having a space overhead of $O(2^{\mathrm{tw}(G)})$ [25]. The pairwise tensor contraction also makes it possible to utilize basic linear algebra subprograms (BLAS) functions to speed up the computation for certain tensor element types.

Example 3. Mapping a graph (left) to a tensor network (right) that encodes the independence polynomial. In the generalized tensor network's graphical representation, a vertex is mapped to a hyperedge, and an edge is mapped to an edge tensor. [**ST: a vertex is mapped to a hyperedge?**]



The contraction of this tensor network can be done in a pairwise order:

$$\begin{aligned}
&\sum_{s_a,s_b,s_c,s_d,s_e} W(x_a)_{s_a} W(x_b)_{s_b} W(x_c)_{s_c} W(x_d)_{s_d} W(x_e)_{s_e} B_{s_a s_b} B_{s_b s_d} B_{s_c s_d} B_{s_a s_c} B_{s_b s_c} B_{s_d s_e} \\
=& \sum_{s_b,s_c} \Bigg(\sum_{s_d} \Bigg(\Bigg(\Bigg(\Bigg(\sum_{s_e} B_{s_d s_e} W(x_e)_{s_e} \Bigg) W(x_d)_{s_d} \Bigg) (B_{s_b s_d} W(x_b)_{s_b}) \Bigg) (B_{s_c s_d} W(x_c)_{s_c}) \Bigg) \\
&\quad \Bigg(B_{s_b s_c} \Bigg(\sum_{s_a} B_{s_a s_b} \left(B_{s_a s_c} W(x_a)_{s_a} \right) \Bigg) \Bigg) \Bigg) \\
=& 1 + x_a + x_b + x_c + x_d + x_e + x_a x_d + x_a x_e + x_c x_e + x_b x_e \\
=& 1 + 5x + 4x^2 \qquad (x_i = x)
\end{aligned}$$

Before contracting the tensor network and evaluating the independence polynomial numerically, let us first elevate the tensor elements 0s and 1s in tensors $W(x)$ and B from integers and floating point numbers to the additive identity, $\mathbb{0}$, and multiplicative identity, $\mathbb{1}$, of a commutative semiring as discussed in Sec. 3. The most natural approach is to treat the tensor elements as polynomials and evaluate the polynomial directly. Let us create a polynomial type, and represent a polynomial $a_0 + a_1x + \ldots + a_kx^k$ as a coefficient vector $(a_0, a_1, \ldots, a_k) \in \mathbb{R}^k$, so, e.g., x is represented as $(0, 1)$. We define the algebra between the polynomials a of order k_a and b of order k_b as

$$\begin{aligned}
a \oplus b &= (a_0 + b_0, a_1 + b_1, \ldots, a_{\max(k_a,k_b)} + b_{\max(k_a,k_b)}),\\
a \odot b &= (a_0 + b_0, a_1b_0 + a_0b_1, a_2b_0 + a_1b_1 + a_0b_2, \ldots, a_{k_a}b_{k_b}),\\
\mathbb{0} &= (), [\textbf{ST: should this be (0)?}]\\
\mathbb{1} &= (1).
\end{aligned} \tag{4.4}$$

We can see these operations are standard addition and multiplication operations of polynomials, and the polynomial type forms a commutative ring. The tensors W and B can thus be written as

$$W^{\text{poly}}_{s_i} = \begin{pmatrix} \mathbb{1} \\ (0,1) \end{pmatrix}, \qquad B^{\text{poly}}_{s_is_j} = \begin{pmatrix} \mathbb{1} & \mathbb{1} \\ \mathbb{1} & \mathbb{0} \end{pmatrix}. \tag{4.5}$$

By contracting the tensor network with the polynomial type, the final result is the exact representation of the independence polynomial. One way to efficiently evaluate the multiplication operation is to use the convolution theorem [30], but this approach suffers from a space overhead proportional to $\alpha(G)$ because each polynomial requires a vector of such size to store the coefficients.

Here, we propose to find the independence polynomial by fitting $\alpha(G)+1$ random values of x_i and $y_i = I(G, x_i)$. One can then compute the independence polynomial coefficients a_i by solving the linear equation:

$$\begin{pmatrix}
1 & x_0 & x_0^2 & \ldots & x_0^{\alpha(G)}\\
1 & x_1 & x_1^2 & \ldots & x_1^{\alpha(G)}\\
\vdots & \vdots & \vdots & \ddots & \vdots\\
1 & x_{\alpha(G)} & x_{\alpha(G)}^2 & \ldots & x_{\alpha(G)}^{\alpha(G)}
\end{pmatrix}
\begin{pmatrix} a_0\\ a_1\\ \vdots\\ a_{\alpha(G)} \end{pmatrix}
=
\begin{pmatrix} y_0\\ y_1\\ \vdots\\ y_{\alpha(G)} \end{pmatrix}. \tag{4.6}$$

With this approach, we do not incur the linear overhead in space. However, because the independence polynomial coefficients can have a huge order-of-magnitude range, if we use floating point numbers in the computation, the round-off errors can be significant for the counting of large independent set sizes. In addition, the number could easily overflow if we use fixed-width integer types. The big integer type is also not a good option because big integers with varying width can be very slow and is incompatible with graphics processing unit (GPU) devices. These problems can be solved by introducing a finite-field algebra GF(p):

$$\begin{aligned}
x \oplus y &= x + y \pmod{p},\\
x \odot y &= xy \pmod{p},\\
\mathbb{0} &= 0,\\
\mathbb{1} &= 1.
\end{aligned} \tag{4.7}$$

With a finite-field algebra, we have the following observations:

1. One can use Gaussian elimination [15] to solve the linear equation Eq. (4.6) since it is a generic algorithm that works for any elements with field algebra. The multiplicative inverse of a finite-field algebra can be computed with the extended Euclidean algorithm.

2. Given the remainders of a larger unknown integer x over a set of co-prime integers $\{p_1, p_2, \ldots, p_n\}$, $x \pmod{p_1 \times p_2 \times \ldots \times p_n}$ can be computed using the Chinese remainder theorem. With this, one can infer big integers from small integers.

With these observations, we develop Algorithm 4.1 to compute the independence polynomial exactly without introducing space overheads. In the algorithm, except the computation using the Chinese remainder theorem, all computations are done with integers of fixed width W. In Appendix D, we provide another method to solve the linear equation using discrete Fourier transform.

Algorithm 4.1 Exactly compute the independence polynomial of a graph G without integer overflow

---

Let $P = 1$, W be the integer width, vector $X = (0, 1, 2, \ldots, \alpha(G))$, matrix $\hat{X}_{ij} = X_i^j$, where $i, j = 0, 1, \ldots, \alpha(G)$

while *true* **do**

  compute the largest prime p that $\gcd(p, P) = 1$ and $p \leq 2^W$

  compute the tensor contraction on $\mathrm{GF}(p)$ and obtain $Y = (y_0, y_1, \ldots, y_{\alpha(G)}) \pmod p$ [**ST: tensor network contraction for each i**]

  $A_p = (a_0, a_1, \ldots, a_{\alpha(G)}) \pmod p = \text{gaussian\_elimination}(\hat{X}, Y \pmod p)$

  $A_{P\times p} = \text{chinese\_remainder}(A_P, A_p)$

  if $A_P = A_{P\times p}$ **then**

    return A_P ; `// converged`

  end

  $P = P \times p$

end

[**ST: discuss this algorithm table, X_{ij}, etc.**]

4.2. Maximal independence polynomial.

[**ST: It might be helpful to add a figure of the tensor network for computing the maximal independence polynomial.**] Instead of counting all independent sets, the maximal independence polynomial counts the number of maximal independent sets of various sizes [19]. Concretely, it is defined as

$$I_{\max}(G, x) = \sum_{k=0}^{\alpha(G)} b_k x^k, \tag{4.8}$$

where b_k is the number of maximal independent sets of size k in G. Obviously, $b_k \leq a_k$ and $b_{\alpha(G)} = a_{\alpha(G)}$. $I_{\max}(G, 1)$ counts the total number of maximal independent sets [14, 24], where the fastest algorithm currently has a runtime of $O(1.3642^{|V|})$ [14]. For the problem of finding the MIS, b_k counts the number of local optimum at size $k < \alpha(G)$, and can, in some cases, provide hints on the difficulty of finding the MIS using local algorithms [**ST: cite experiment**]. The uni-modality, log-concavity, and real-rootness properties of the maximal independence polynomial for special classes of graphs have also been studied [19].

Let us denote the neighborhood of a vertex v as $N(v)$ and denote $N[v] = N(v) \cup \{v\}$. A maximal independent set I_m is an independent set where there exists no vertex v such that $I_m \cap N[v] = \emptyset$. We can modify the tensor network for computing the independence polynomial to include this restriction. Instead of defining the restriction on vertices and edges, it is more natural to define it on $N[v]$:

$$T(x_v)_{s_1, s_2, \ldots, s_{|N(v)|}, s_v} = \begin{cases} s_v x_v & s_1 = s_2 = \ldots = s_{|N(v)|} = 0, \\ 1 - s_v & \textit{otherwise}. \end{cases} \tag{4.9}$$

Intuitively, it means if all the neighborhood vertices are not in I_m, i.e., $s_1 = s_2 = \ldots = s_{|N(v)|} = 0$, then v should be in I_m, counted as x_v, and if any of the neighborhood vertices is in I_m, i.e.,

$\exists i \in \{1, 2, \ldots, |N(v)|\}$ s.t. $s_i = 1$, then v cannot be in I_m. As an example, for a vertex of degree 2, the resulting rank-3 tensor is

$$T(x_v) = \begin{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \\ \begin{pmatrix} x_v & 0 \\ 0 & 0 \end{pmatrix} \end{pmatrix}. \tag{4.10}$$

With the tensors defined as $T(x_v)$, we can perform a similar computation of contracting the tensor network with the same tensor element types as described in the previous section 4.1, the result of which then produces the maximal independence polynomial. The computational complexity of this new tensor network contraction is often greater than the one for computing the independence polynomial. However, in most sparse graphs, this tensor network contraction approach is still significantly faster than enumerating all the maximal cliques on its complement graph using the Bron-Kerbosch algorithm [5], which is the standard algorithm that we are aware of to compute the maximal independence polynomial.

5. Maximum independent sets and its counting problem.

5.1. Tropical algebra for finding the independence number and counting MISs. In the previous section, we focused on computing the independence polynomial for a graph G of given independence number $\alpha(G)$, but we did not show how to compute this number. The method we use to compute this quantity is based on the following observations. Let $x = \infty$, the independence polynomial becomes

$$I(G, \infty) = a_{\alpha(G)} \infty^{\alpha(G)}, \tag{5.1}$$

where the lower-order terms vanish. We can thus replace the polynomial type $a = (a_0, a_1, \ldots, a_k)$ with $a(k) = (a_k, k)$, where the second element stores the largest exponent k and the first element stores the corresponding coefficient a_k. From this, we can define a new algebra as [**ST: I've changed this, but I don't have strong feelings about this. We can certainly consider changing back. I thought this would be more consistent with the earlier notations and show how it's stored in the program as well.**]

$$\begin{aligned} a(k) \oplus a(j) &= \begin{cases} \big(a_k + a_j, \max(k, j)\big), & k = j \\ \big(a_j, \max(k, j)\big), & k < j \\ \big(a_k, \max(k, j)\big), & k > j \end{cases}, \\ a(k) \odot a(j) &= (a_k a_j, k + j) \\ \mathbb{0} &= (0, -\infty) \\ \mathbb{1} &= (1, 0). \end{aligned} \tag{5.2}$$

The algebra of the exponents becomes the max-plus tropical algebra: $k \oplus j = \max(k, j)$ and $k \odot j = k + j$ [23, 26]. This algebra is the same as the one used in Liu et al. [22] to calculate and count spin glass ground states. For independent set calculations here, the vertex tensor and edge tensor becomes:

$$W_{s_i}^{\text{tropical}} = \begin{pmatrix} \mathbb{1} \\ (1, 1) \end{pmatrix}, \qquad B_{s_i s_j}^{\text{tropical}} = \begin{pmatrix} \mathbb{1} & \mathbb{1} \\ \mathbb{1} & \mathbb{0} \end{pmatrix}. \tag{5.3}$$

5.2. Truncated polynomial algebra for counting independent sets of large size. Instead of counting just the MISs, one may be interested in counting the independent sets of large sizes close to the MIS size. For example, if one is interested in counting only $a_{\alpha(G)}$ and $a_{\alpha(G)-1}$, we can define a truncated polynomial algebra by keeping only the largest two coefficients in the polynomial in Eq. (4.4):

$$
\begin{aligned}
a \oplus b &= (a_{\max(k_a,k_b)-1} + b_{\max(k_a,k_b)-1}, a_{\max(k_a,k_b)} + b_{\max(k_a,k_b)}),\\
a \odot b &= (a_{k_a-1}b_{k_b} + a_{k_a}b_{k_b-1}, a_{k_a}b_{k_b}),\\
\mathbb{0} &= (), [\textbf{ST: should this be (0)?}]\\
\mathbb{1} &= (1).
\end{aligned}
\tag{5.4}
$$

In the program, we thus need a data structure that contains three fields, the largest order k, and the coefficients for the two largest orders a_k and a_{k-1}. This approach can clearly be extended to calculate more independence polynomial coefficients and is more efficient than calculating the entire independence polynomial. [**ST: can be used to enumerate suboptimal solution as well?**]

6. Enumeration of configurations and enumeration bounds.

6.1. Enumeration of MIS configurations. The enumeration problems of independent sets are also interesting and has been studied extensively in the literature [5, 10, 20], including, for example, the enumeration of all independent sets, the enumeration of all maximal independent sets, or the enumeration of all MISs. The standard algorithm to enumerate all maximal independent sets is the Bron-Kerbosch algorithm [5], which, of course, includes the enumeration of all MISs and can be used to enumerate all independent sets as well. Here, we adapt the tensor element types for different enumeration problems. For example, our tensor-network based algorithm to enumerate all MISs is expectedly much faster and more space-efficient than the Bron-Kerbosch algorithm, which lists all maximal independent sets.

To enumerate all independent sets, we input a set of configurations into the tensor elements. More concretely, we design a new element type having the following algebra

$$
\begin{aligned}
s \oplus t &= s \cup t\\
s \odot t &= \{\sigma \vee^\circ \tau \,|\, \sigma \in s, \tau \in t\}\\
\mathbb{0} &= \{\}\\
\mathbb{1} &= \{0^{\otimes |V|}\}.
\end{aligned}
\tag{6.1}
$$

where s and t are each a set of $|V|$-bit strings and $\vee^\circ$ is the bitwise OR operation over two bit strings. [**ST: This is the bitwise OR operation right? I can't seem to find online what you call Hadamard. Is there a definition somewhere?**] [**ST: it would be useful to give an example of the operations above, $s = ..., t = ..., s \oplus t = ..., s \odot t = ...$**] [**ST: do we need to define $\{\} \vee^\circ \sigma = \sigma \vee^\circ \{\} = \{\}$?**] The variable x_i in the vertex tensor is initialized to $x_i = \{\boldsymbol{e}_i\}$, where $\boldsymbol{e}_i$ is the standard basis vector of size $|V|$ and having 1 at index i. The vertex and edge tensors are thus

$$
W_{s_i}^{\text{enum}} = \begin{pmatrix} \mathbb{1} \\ \{\boldsymbol{e}_i\} \end{pmatrix}, \qquad B_{s_i s_j}^{\text{enum}} = \begin{pmatrix} \mathbb{1} & \mathbb{1} \\ \mathbb{1} & \mathbb{0} \end{pmatrix}.
\tag{6.2}
$$

Contraction of the tensor network enumerates all the independent sets. To enumerate only the MISs, we can combine the above algebra Eq. (6.1) with the tropical algebra in Eq. (6.3) and define $s(k) = (s_k, k)$, where the first element follows the algebra in Eq. (6.1) and the second element follows the max-plus tropical algebra. The combined operations become:

$$
\begin{aligned}
s(k) \oplus s(j) &= \begin{cases} \left(s_k \cup s_j, \max(k, j)\right), & k = j \\ \left(s_j, \max(k, j)\right), & k < j \\ \left(s_k, \max(k, j)\right), & k > j \end{cases}, \\
s(k) \odot s(j) &= (\{\sigma \vee^{\circ} \tau \,|\, \sigma \in s_k, \tau \in s_j\}, k + j) \\
\mathbb{0} &= (\{\}, -\infty) \\
\mathbb{1} &= (\{0^{\otimes |V|}\}, 0).
\end{aligned} \tag{6.3}
$$

Clearly, the vertex tensor and edge tensor become

$$
W_{s_i}^{\text{MISenum}} = \begin{pmatrix} \mathbb{1} \\ (\{\boldsymbol{e}_i\}, 1) \end{pmatrix}, \qquad B_{s_i s_j}^{\text{MISenum}} = \begin{pmatrix} \mathbb{1} & \mathbb{1} \\ \mathbb{1} & \mathbb{0} \end{pmatrix}. \tag{6.4}
$$

The contraction of the corresponding tensor network yields an enumeration of all MIS configurations. If one is interested in obtaining only a single MIS configuration, one can just keep a single configuration in the intermediate computations to save the computational effort. Here is a new algebra defined on the bit strings, replacing the sets of bit strings in Eq. (6.1),

$$
\begin{aligned}
\sigma \oplus \tau &= \text{select}(\sigma, \tau), \\
\sigma \odot \tau &= (\sigma \vee^{\circ} \tau), \\
\mathbb{0} &= 1^{\otimes |V|}, [\textbf{ST} : \text{should be } \mathbb{0} = \{\}?] \\
\mathbb{1} &= 0^{\otimes |V|},
\end{aligned} \tag{6.5}
$$

where the `select` function picks one of σ and τ by some criteria to make the algebra commutative and associative, e.g. by picking one with a larger integer value. In practice, we can just pick randomly from them, in which case the program will output one of the MIS configurations randomly.

[**ST: The zero element doesn't seem right to me. Do we need to separately define**

$$
\begin{aligned}
a \oplus \mathbb{0} &= \mathbb{0} \oplus a = a \\
a \odot \mathbb{0} &= \mathbb{0} \odot a = \mathbb{0}
\end{aligned}
$$

]

[**ST: Can we also enumerate all maximal independent sets by combining with Eq. (4.9) or enumerate suboptimal solutions by combining with the truncated polynomial?**]

6.2. Bounding the MIS enumeration space. When we use the algebra in Eq. (6.3) to enumerate all MIS configurations, we find that the program stores significantly more intermediate configurations than necessary and thus incur significant overheads in space. To speed up the computation and reduce space overhead, we use $\alpha(G)$ to bound the searching space. First, we compute the value of $\alpha(G)$ with tropical algebra and cache all intermediate tensors. Then, we compute a boolean mask for each cached tensor, where we use a boolean true to represent a tensor element having a contribution to the MIS (i.e. with a non-zero gradient) and boolean false otherwise. Finally, we perform masked matrix multiplication using the new element type with the above algebra, Eq. (6.3), for obtaining all configurations. Note that these masks in fact correspond to tensor elements with non-zero gradients with respect to the MIS size; we compute these masks by back propagation of the gradients. To derive the back-propagation rule for tensor contraction, we first reduce the problem to finding the back-propagation rule of a tropical matrix multiplication $C = AB$. Since $C_{ik} = \bigoplus_j A_{ij} \odot B_{jk} = \max_j A_{ij} \odot B_{jk}$ with tropical algebra, we have the following inequality

$$A_{ij} \odot B_{jk} \leq C_{ik}. \tag{6.6}$$

Here $\leq$ on tropical numbers are the same as the real-number algebra. The equality holds for some j', which means $A_{ij'}$ and $B_{j'k}$ have contributions to C_{ik}. Since $A_{ij} \odot B_{jk} = A_{ij} + B_{jk}$, one can move B_{jk} to the right hand side of the inequality:

$$A_{ij} \leq C_{ik} \odot B_{jk}^{\circ -1} \tag{6.7}$$

where $^{\circ -1}$ is the element-wise multiplicative inverse on tropical algebra (which is the additive inverse on real numbers). The inequality still holds if we take the minimum over k:

$$A_{ij} \leq \min_k (C_{ik} \odot B_{jk}^{\circ -1}) = \left(\max_k \left(C_{ik}^{\circ -1} \odot B_{jk}\right)\right)^{\circ -1} = \left(\bigoplus_k \left(C_{ik}^{\circ -1} \odot B_{jk}\right)\right)^{\circ -1} = \left(C^{\circ -1} B^{\mathsf{T}}\right)_{ij}^{\circ -1}. \tag{6.8}$$

On the right hand side, we transform the operation into a tropical matrix multiplication so that we can utilize the fast tropical BLAS routines [**ST: do we need to cite here?**]. Again, the equality holds if and only if the element A_{ij} has a contribution to C (i.e. having a non-zero gradient). Let the gradient mask for C being $\overline{C}$, the back-propagation rule for gradient masks reads

$$\overline{A}_{ij} = \delta\left(A_{ij}, \left(\left(C^{\circ -1} \circ \overline{C}\right) B^{\mathsf{T}}\right)_{ij}^{\circ -1}\right), \tag{6.9}$$

where $\circ$ is the element-wise product, boolean false is treated as the tropical number $\mathbb{0}$, and boolean true is treated as the tropical number $\mathbb{1}$. This rule defined on matrix multiplication can be easily generalized to tensor contraction by replacing the matrix multiplication between $C^{\circ -1} \circ \overline{C}$ and B^{T} by a tensor contraction. With the above method, one can significantly reduce the space needed to store the intermediate configurations. [**JG: maybe add an appendix?**] [**ST: I've edited this section a bit, and I think it's okay now - probably don't need an appendix.**]

7. Benchmarks and case studies.

7.1. Performance benchmarks. We run a sequential program benchmark on CPU Intel(R) Core(TM) i5-10400 CPU @ 2.90GHz, and show the results in Figure 1. Tensor network contraction can be parallelized. When the element type is immutable, one can also run it on GPUs to speed up the computation.

7.2. Example case studies.

7.2.1. Hard-square constant.

7.2.2. Euler characteristics of independence complex.

7.2.3. Partition functions and finite-temperature phase transitions.

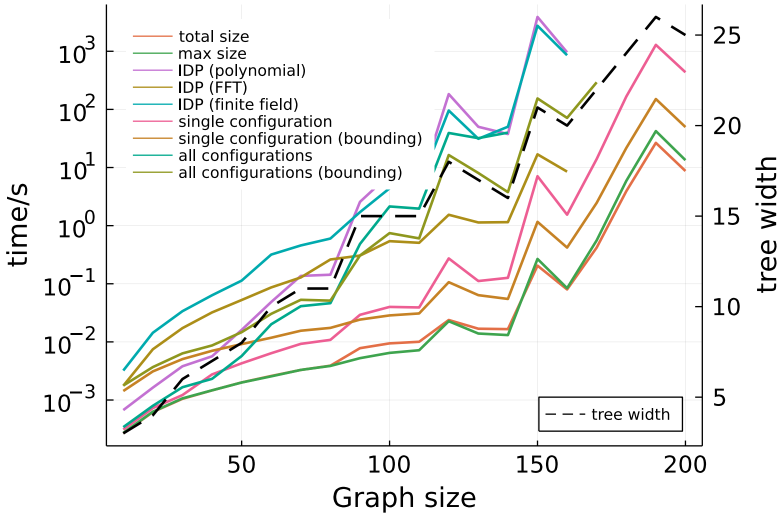


Figure 1: [**ST: label x-axis as "number of vertices, $|V|$"**] Benchmark results for computing different properties of independent sets with different tensor element types. The right axis is only for the dashed line.

[**ST: split into two graphs: one with time versus number of vertices, $|V|$, and one with time versus tree width.**] [**ST: We also need to show/say what types of graphs is used here for benchmarking.**]

8. Discussion and conclusion. We have introduced a new approach based on tensor network contraction to compute the independence number, independence polynomial, maximal independence polynomial, and enumeration of MIS configurations. We derived the backward rule for tropical tensor network to bound the search of solution space. Although many of these properties are global, we can encode them to different tensor element types as commutative semirings. The power of our tensor network approach is not only limited to the independent set problem, in Appendix C, we show how to map the matching problem and k-coloring problem to a tensor network. Here, we want to discuss more from the programming perspective. We show some of the Julia language [3] implementations in Appendix A and you will find it surprisingly short. What we need to do is just defining two operations $\oplus$ and $\odot$ and two special elements $\mathbb{0}$ and $\mathbb{1}$. The style that we program is called generic programming, meaning one can feed different data types into the same program, and the program will compute the result with a proper performance. In C++, one can use templates for such a purpose. We chose Julia because its just-in-time compiling is very powerful that it can generate fast code dynamically for users. Elements of fixed size, such as the finite-field algebra, truncated polynomial, tropical number and tropical number with counting or configuration field described in this paper can all be inlined in an array. Furthermore, these inlined arrays can be uploaded to GPU devices for faster computation with generic matrix multiplication implemented in CUDA.jl [2].

REFERENCES

[1] S. F. Barr, *Courcelle's Theorem: Overview and Applications*, PhD thesis, Oberlin College, 2020.
[2] T. Besard, C. Foket, and B. D. Sutter, *Effective extensible programming: Unleashing julia on gpus*, CoRR,

abs/1712.03112 (2017), http://arxiv.org/abs/1712.03112, https://arxiv.org/abs/1712.03112.
[3] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman, *Julia: A fast dynamic language for technical computing*, 2012, https://arxiv.org/abs/1209.5145, https://arxiv.org/abs/1209.5145.
[4] C. M. Bishop, *Pattern Recognition and Machine Learning*, Springer, 2006.
[5] C. Bron and J. Kerbosch, *Algorithm 457: finding all cliques of an undirected graph*, Communications of the ACM, 16 (1973), pp. 575–577.
[6] S. Butenko and P. M. Pardalos, *Maximum Independent Set and Related Problems, with Applications*, PhD thesis, USA, 2003. AAI3120100.
[7] I. Cirac, D. Perez-Garcia, N. Schuch, and F. Verstraete, *Matrix Product States and Projected Entangled Pair States: Concepts, Symmetries, and Theorems*, arXiv e-prints, (2020), arXiv:2011.12127, p. arXiv:2011.12127, https://arxiv.org/abs/2011.12127.
[8] B. Courcelle, *The monadic second-order logic of graphs. i. recognizable sets of finite graphs*, Information and computation, 85 (1990), pp. 12–75.
[9] J. C. Dyre, *Simple liquids' quasiuniversality and the hard-sphere paradigm*, Journal of Physics: Condensed Matter, 28 (2016), p. 323001.
[10] D. Eppstein, M. Löffler, and D. Strash, *Listing all maximal cliques in sparse graphs in near-optimal time*, in Algorithms and Computation, O. Cheong, K.-Y. Chwa, and K. Park, eds., Berlin, Heidelberg, 2010, Springer Berlin Heidelberg, pp. 403–414.
[11] H. C. M. Fernandes, J. J. Arenzon, and Y. Levin, *Monte carlo simulations of two-dimensional hard core lattice gases*, The Journal of Chemical Physics, 126 (2007), p. 114508, https://doi.org/10.1063/1.2539141, https://doi.org/10.1063/1.2539141, https://arxiv.org/abs/https://doi.org/10.1063/1.2539141.
[12] G. M. Ferrin, *Independence polynomials*, (2014).
[13] F. V. Fomin and P. Kaski, *Exact exponential algorithms*, Communications of the ACM, 56 (2013), pp. 80–88.
[14] S. Gaspers, D. Kratsch, and M. Liedloff, *On independent sets and bicliques in graphs*, Algorithmica, 62 (2012), pp. 637–658, https://doi.org/10.1007/s00453-010-9474-1, https://doi.org/10.1007/s00453-010-9474-1.
[15] G. H. Golub and C. F. Van Loan, *Matrix computations*, vol. 3, JHU press, 2013.
[16] J. Gray and S. Kourtis, *Hyper-optimized tensor network contraction*, Quantum, 5 (2021), p. 410, https://doi.org/10.22331/q-2021-03-15-410, http://dx.doi.org/10.22331/q-2021-03-15-410.
[17] N. J. Harvey, P. Srivastava, and J. Vondrák, *Computing the independence polynomial: from the tree threshold down to the roots*, in Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SIAM, 2018, pp. 1557–1576.
[18] J. Hastad, *Clique is hard to approximate within n/sup 1-/spl epsiv*, in Proceedings of 37th Conference on Foundations of Computer Science, IEEE, 1996, pp. 627–636.
[19] H. Hu, T. Mansour, and C. Song, *On the maximal independence polynomial of certain graph configurations*, Rocky Mountain Journal of Mathematics, 47 (2017), pp. 2219 – 2253, https://doi.org/10.1216/RMJ-2017-47-7-2219, https://doi.org/10.1216/RMJ-2017-47-7-2219.
[20] D. S. Johnson, M. Yannakakis, and C. H. Papadimitriou, *On generating all maximal independent sets*, Information Processing Letters, 27 (1988), pp. 119–123, https://doi.org/https://doi.org/10.1016/0020-0190(88)90065-8, https://www.sciencedirect.com/science/article/pii/0020019088900658.
[21] V. E. Levit and E. Mandrescu, *The independence polynomial of a graph-a survey*, in Proceedings of the 1st International Conference on Algebraic Informatics, vol. 233254, Aristotle Univ. Thessaloniki Thessaloniki, 2005, pp. 231–252.
[22] J.-G. Liu, L. Wang, and P. Zhang, *Tropical tensor network for ground states of spin glasses*, Physical Review Letters, 126 (2021), https://doi.org/10.1103/physrevlett.126.090506, http://dx.doi.org/10.1103/PhysRevLett.126.090506.
[23] D. Maclagan and B. Sturmfels, *Introduction to tropical geometry*, vol. 161, American Mathematical Soc., 2015, http://www.cs.technion.ac.il/~janos/COURSES/238900-13/Tropical/MaclaganSturmfels.pdf.
[24] F. Manne and S. Sharmin, *Efficient counting of maximal independent sets in sparse graphs*, in International Symposium on Experimental Algorithms, Springer, 2013, pp. 103–114.
[25] I. L. Markov and Y. Shi, *Simulating quantum computation by contracting tensor networks*, SIAM Journal on Computing, 38 (2008), p. 963–981, https://doi.org/10.1137/050644756, http://dx.doi.org/10.1137/050644756.
[26] C. Moore and S. Mertens, *The nature of computation*, OUP Oxford, 2011.
[27] R. Orús, *A practical introduction to tensor networks: Matrix product states and projected entangled pair states*, Annals of Physics, 349 (2014), pp. 117–158.
[28] F. Pan and P. Zhang, *Simulating the sycamore quantum supremacy circuits*, 2021, https://arxiv.org/abs/2103.03074.
[29] H. Pichler, S.-T. Wang, L. Zhou, S. Choi, and M. D. Lukin, *Computational complexity of the rydberg blockade in two dimensions*, arXiv preprint arXiv:1809.04954, (2018).
[30] A. Schönhage and V. Strassen, *Schnelle multiplikation grosser zahlen*, Computing, 7 (1971), pp. 281–292.
[31] A. A. Stepanov and D. E. Rose, *From mathematics to generic programming*, Pearson Education, 2014.

[32] Q. Wu and J.-K. Hao, *A review on algorithms for maximum clique problems*, European Journal of Operational Research, 242 (2015), pp. 693–709, https://doi.org/https://doi.org/10.1016/j.ejor.2014.09.064, https://www.sciencedirect.com/science/article/pii/S0377221714008030.
[33] M. Xiao and H. Nagamochi, *Exact algorithms for maximum independent set*, Information and Computation, 255 (2017), p. 126–146, https://doi.org/10.1016/j.ic.2017.06.001, http://dx.doi.org/10.1016/j.ic.2017.06.001.

Appendix A. Technical guide.

OMEinsum a package for the `einsum` function,

OMEinsumContractionOrders a package for finding the optimal contraction order for the `einsum` function https://github.com/Happy-Diode/OMEinsumContractionOrders.jl,

TropicalGEMM a package for efficient tropical matrix multiplication (compatible with OMEinsum),

TropicalNumbers a package providing tropical number types and tropical algebra, one o the dependency of TropicalGEMM,

LightGraphs a package providing graph utilities, like random regular graph generator,

Polynomials a package providing polynomial algebra and polynomial fitting,

Mods and Primes packages providing finite field algebra and prime number generators.

One can install these packages by opening a Julia REPL, type `]` to enter the `pkg>` mode and type, e.g.

```
pkg> add OMEinsum LightGraphs Mods Primes FFTW Polynomials TropicalNumbers
```

It may surprise you that the Julia implementation of algorithms introduced in the paper is so short that except the bounding and sparsity related parts, all are contained in this appendix. After installing required packages, one can open a Julia REPL and copy the following code into it.

```
using OMEinsum, OMEinsumContractionOrders
using OMEinsum: NestedEinsum, flatten, getixs
using LightGraphs
using Random

# generate a random regular graph of size 100, degree 3
graph = (Random.seed!(2); LightGraphs.random_regular_graph(100, 3))

# generate einsum code, i.e. the labels of tensors
code = EinCode(([minmax(e.src,e.dst) for e in LightGraphs.edges(graph)]..., # labels for edge
     tensors
               [(i,) for i in LightGraphs.vertices(graph)]...), ())        # labels for vertex
     tensors

# an einsum contraction without contraction order specified is called `EinCode`,
# an einsum contraction has contraction order (specified as a tree structure) is called `
     NestedEinsum`.
# assign each label a dimension-2, it will be used in contraction order optimization
# `symbols` function extracts tensor labels into a vector.
symbols(::EinCode{ixs}) where ixs = unique(Iterators.flatten(filter(x->length(x)==1,ixs)))
symbols(ne::OMEinsum.NestedEinsum) = symbols(flatten(ne))
size_dict = Dict([s=>2 for s in symbols(code)])
# optimize the contraction order using KaHyPar + Greedy, target space complexity is 2^17
optimized_code = optimize_kahypar(code, size_dict; sc_target=17, max_group_size=40)
println("time/space complexity is $(OMEinsum.timespace_complexity(optimized_code, size_dict))")

# a function for computing independence polynomial
function independence_polynomial(x::T, code) where {T}
  xs = map(getixs(flatten(code))) do ix
       # if the tensor rank is 1, create a vertex tensor.
       # otherwise the tensor rank must be 2, create a bond tensor.
       length(ix)==1 ? [one(T), x] : [one(T) one(T); one(T) zero(T)]
```

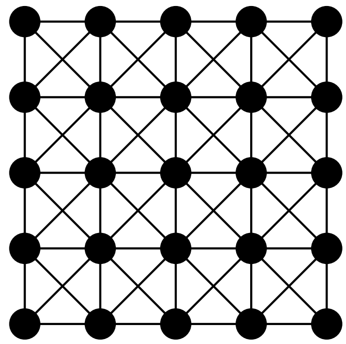
```julia
    end
    # both `EinCode` and `NestedEinsum` are callable, inputs are tensors.
   code(xs...)
end

########## COMPUTING MAXIMUM INDEPENDENT SET SIZE AND ITS DEGENERACY ###########

# using Tropical numbers to compute the MIS size and MIS degeneracy.
using TropicalNumbers
mis_size(code) = independence_polynomial(TropicalF64(1.0), code)[]
println("the maximum independent set size is $(mis_size(optimized_code).n)")
# A `CountingTropical` object has two fields, tropical field `n` and counting field `c`.
mis_count(code) = independence_polynomial(CountingTropical{Float64,Float64}(1.0, 1.0), code)[]
println("the degeneracy of maximum independent sets is $(mis_count(optimized_code).c)")

########## COMPUTING INDEPENDENCE POLYNOMIAL ###########

# using Polynomial numbers to compute the polynomial directly
using Polynomials
println("the independence polynomial is $(independence_polynomial(Polynomial([0.0, 1.0]),
     optimized_code)[])")

# using fast fourier transformation to compute the independence polynomial,
# here we chose r > 1 because we care more about configurations with large independent set sizes
     .
using FFTW
function independence_polynomial_fft(code; mis_size=Int(mis_size(code)[].n), r=3.0)
   ω = exp(-2im*π/(mis_size+1))
   xs = r .* collect(ω .^ (0:mis_size))
   ys = [independence_polynomial(x, code)[] for x in xs]
   Polynomial(ifft(ys) ./ (r .^ (0:mis_size)))
end
println("the independence polynomial (fft) is $(independence_polynomial_fft(optimized_code))")

# using finite field algebra to compute the independence polynomial
using Mods, Primes
# two patches to ensure gaussian elimination works
Base.abs(x::Mod) = x
Base.isless(x::Mod{N}, y::Mod{N}) where N = mod(x.val, N) < mod(y.val, N)

function independence_polynomial_finitefield(code; mis_size=Int(mis_size(code)[].n), max_order=1
     00)
    N = typemax(Int32) # Int32 is faster than Int.
    YS = []
    local res
    for k = 1:max_order
       N = Primes.prevprime(N-one(N))  # previous prime number
        # evaluate the polynomial on a finite field algebra of modulus `N`
        rk = _independance_polynomial(Mods.Mod{N,Int32}, code, mis_size)
        push!(YS, rk)
        if max_order==1
            return Polynomial(Mods.value.(YS[1]))
        elseif k != 1
            ra = improved_counting(YS[1:end-1])
            res = improved_counting(YS)
            ra == res && return Polynomial(res)
        end
    end
    @warn "result is potentially inconsistent."
    return Polynomial(res)
end
function _independance_polynomial(::Type{T}, code, mis_size::Int) where T
   xs = 0:mis_size
   ys = [independence_polynomial(T(x), code)[] for x in xs]
   A = zeros(T, mis_size+1, mis_size+1)
   for j=1:mis_size+1, i=1:mis_size+1
      A[j,i] = T(xs[j])^(i-1)
   end
   A \ T.(ys)  # gaussian elimination to compute ``A^{-1} y```
end
improved_counting(sequences) = map(yi->Mods.CRT(yi...), zip(sequences...))

println("the independence polynomial (finite field) is $(independence_polynomial_finitefield(
     optimized_code))")
```

```julia
########## FINDING OPTIMAL CONFIGURATIONS ###########

# define the config enumerator algebra
struct ConfigEnumerator{N,C}
    data::Vector{StaticBitVector{N,C}}
end
function Base.:+(x::ConfigEnumerator{N,C}, y::ConfigEnumerator{N,C}) where {N,C}
    res = ConfigEnumerator{N,C}(vcat(x.data, y.data))
    return res
end
function Base.:*(x::ConfigEnumerator{L,C}, y::ConfigEnumerator{L,C}) where {L,C}
    M, N = length(x.data), length(y.data)
    z = Vector{StaticBitVector{L,C}}(undef, M*N)
    for j=1:N, i=1:M
        z[(j-1)*M+i] = x.data[i] .| y.data[j]
    end
    return ConfigEnumerator{L,C}(z)
end
Base.zero(::Type{ConfigEnumerator{N,C}}) where {N,C} = ConfigEnumerator{N,C}(StaticBitVector{N,C
     }[])
Base.one(::Type{ConfigEnumerator{N,C}}) where {N,C} = ConfigEnumerator{N,C}([TropicalNumbers.
     staticfalses(StaticBitVector{N,C})])

# enumerate all configurations if `all` is true, compute one otherwise.
# a configuration is stored in the data type of `StaticBitVector`, it uses integers to represent
      bit strings.
# `ConfigTropical` is defined in `TropicalNumbers`. It has two fields, tropical number `n` and
     optimal configuration `config`.
# `CountingTropical{T,<:ConfigEnumerator}` is a simple stores configurations instead of simple
     counting.
function mis_config(code; all=false)
    # map a vertex label to an integer
    vertex_index = Dict([s=>i for (i, s) in enumerate(symbols(code))])
    N = length(vertex_index)  # number of vertices
    C = TropicalNumbers._nints(N)  # number of integers to store N bits
    xs = map(getixs(flatten(code))) do ix
        T = all ? CountingTropical{Float64, ConfigEnumerator{N,C}} : ConfigTropical{Float64, N,
     C}
        if length(ix) == 2
            return [one(T) one(T); one(T) zero(T)]
        else
            s = TropicalNumbers.onehot(StaticBitVector{N,C}, vertex_index[ix[1]])
            if all
                [one(T), T(1.0, ConfigEnumerator([s]))]
            else
                [one(T), T(1.0, s)]
            end
        end
    end
   return code(xs...)
end

println("one of the optimal configurations is $(mis_config(optimized_code; all=false)[].config)"
     )

# enumerating configurations directly can be very slow (~15min), please check the bounding
     version in our Github repo.
println("all optimal configurations are $(mis_config(optimized_code; all=true)[].c)")
```

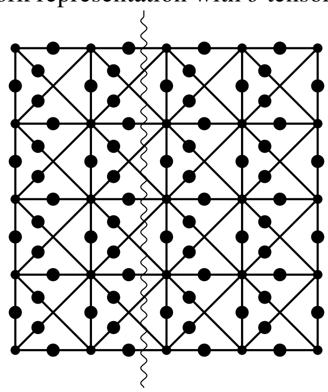
In the above examples, the configuration enumeration is very slow, one should use the optimal MIS size for bounding as described in the main text. We will not show any example about implementing the backward rule here because it has approximately 100 lines of code. Please checkout our GitHub repository https://github.com/Happy-Diode/NoteOnTropicalMIS.

Appendix B. Why not introducing δ tensors.

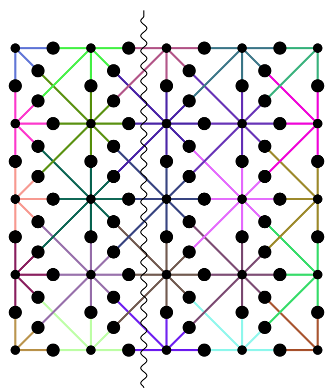
Given a graph



Its traditional tensor network representation with δ tensors is



where a small circle on an edge is a diagonal tensor. Its rank is 8 in the bulk. If we contract this tensor network in a naive column-wise order, the maximum intermediate tensor is approximately $3L$, giving a space complexity $\approx 2^{3L}$. If we treat it as the following generalized tensor network



where we use different colors to distinguish different hyperedges. Now, the vertex tensor is always rank 1. With the same naive contraction order, we can see the maximum intermediate tensor is approximately of size 2^L by counting the colors.

Appendix C. Generalizing to other graph problems. There are some other graph problems that can be encoded in a tensor network. To understand its representation power, it is a good starting point to connect it with dynamic programming because a tensor network can be viewed as a special type of dynamic programming where its update rule can be characterized by a linear operation. Courcelle's theorem [8, 1] states that a problem quantified by monadic second order logic (MSO) on a graph with bounded treewidth k can be solved in linear time with respect to the graph size. Dynamic programming is a traditional approach to attack a MSO problem, it can solve the maximum independent set problem in $O(2^k)n$, which is similar to the tensor network approach. We mentioned in the main text that tensor network has nice analytic property make it easier for generic programming. The cost is, the tensor network is less expressive than dynamic programming, However, that are still some other problems that can be expressed in the framework of generic tensor network.

C.1. Matching problem. A matching polynomial of a graph G is defined as

$$M(G, x) = \sum_{k=1}^{|V|/2} c_k x^k, \tag{C.1}$$

where k is the number of matches, and coefficients c_k are countings.

We define a tensor of rank $d(v) = |N(v)|$ on vertex v such that,

$$W_{v\to n_1, v\to n_2, \ldots, v\to n_{d(v)}} = \begin{cases} 1, & \sum_{i=1}^{d(v)} v \to n_i \leq 1, \\ 0, & \text{otherwise}, \end{cases} \tag{C.2}$$

and a tensor of rank 1 on the bond

$$B_{v\to w} = \begin{cases} 1, & v \to w = 0 \\ x, & v \to w = 1. \end{cases} \tag{C.3}$$

Here, we use bond index $v \to w$ to label tensors.

C.2. k-Coloring. Let us use 3-coloring on the vertex as an example. We can define a vertex tensor as

$$W = \begin{pmatrix} r_v \\ g_v \\ b_v \end{pmatrix}, \tag{C.4}$$

and an edge tensor as

$$B = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}. \tag{C.5}$$

The number of possible coloring can be obtained by contracting this tensor network by setting vertex tensor elements r_v, g_v and b_v to 1. By designing generic types as tensor elements, one should be able to get all possible colorings. It is straight forward to define the k-coloring problem on edges hence we will not discuss the detailed construction here.

Appendix D. Discrete Fourier transform for computing the independence polynomial.

In section 4.1, we show that the independence polynomial can be obtained by solving the linear equation Eq. (4.6). Since the coefficients of the independence polynomial can range many orders of magnitude, the round-off errors in fitting can be significant if we use random floating point numbers for x_i. In the main text, we propose to use a finite field GF(p) to circumvent overflow and round-off errors. Here, we give another method based on discrete Fourier transform. Instead of choosing x_i as random numbers, we can choose them such that they form a geometric sequence in the complex domain $x_j = r\omega^j$, where $r \in \mathbb{R}$ and $\omega = e^{-2\pi i/(\alpha(G)+1)}$. The linear equation thus becomes

$$\begin{pmatrix} 1 & r & r^2 & \dots & r^{\alpha(G)} \\ 1 & r\omega & r^2\omega^2 & \dots & r^{\alpha(G)}\omega^{\alpha(G)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & r\omega^{\alpha(G)} & r^2\omega^{2\alpha(G)} & \dots & r^{\alpha(G)}\omega^{\alpha(G)^2} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{\alpha(G)} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{\alpha(G)} \end{pmatrix}. \tag{D.1}$$

Let us rearrange the coefficients r^j to a_j, the matrix on the left side becomes the discrete Fourier transform matrix. Thus, we can obtain the coefficients by inverse Fourier transform $\vec{a}_r = \mathrm{FFT}^{-1}(\omega)\cdot\vec{y}$, where $(\vec{a}_r)_j = a_j r^j$. By choosing different r, one can obtain better precision in low independent set size region ($\omega < 1$) or high independent set size region ($\omega > 1$). [**ST: is that $\omega < 1$ or $r < 1$?**]