COMPUTING PROPERTIES OF INDEPENDENT SETS BY GENERIC PROGRAMMING TENSOR NETWORKS *

XXX† AND YYY‡

Abstract. We introduce a method of using generic programming tensor network to compute various properties of independent sets, which includes: maximum size of such sets, the number of sets of a given size, enumeration of sets of a given size. It can be generalized to compute properties of other problems like cut size, coloring, maximal cliques et al. At the same time, our algorithms are very simple to implement and one can directly utilize recent advances in tensor network contraction techniques like near optimal contraction order finding and slicing. The algorithmic complexity of this approach is $2^{\mathrm{tw}(G)}$, where $\mathrm{tw}(G)$ the tree width of the problem graph. To demonstrate the versatility of this tool, we apply it to a few examples including the calculations of the hard-square entropy constant, Euler characteristics of the independence complex, partition functions, and finite-temperature phase transitions on square-lattice graphs[**JG: remove some examples**].

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

AMS subject classifications. 05C31, 14N07

[**JG: independence polynomial at -1 [10, 35], two motivations of computing independence polynomial: 1. Lee-Yang zero [33, 51], 2. Lovász local lemma [46].**]

1. Introduction.

In graph theory and combinatorial optimization, there are many interesting hard computational problems concerning properties of independent sets. For an undirected graph $G = (V, E)$, an independent set $I \subseteq V$ is a set of vertices that for any vertex pair $u, v \in I$, $(u, v) \notin E$. The primary insterest of many computational complexity theorists is finding the maximum size of such vertex sets $\alpha(G) \equiv \max_I |I|$ and one of the sets of this size. 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 [27]. Naive exhaustive search for an MIS requires computing time $O\left(2^{|V|}\right)$. People developed more efficient exact algorithms for finding MISs, including the branching algorithms [48, 44] and dynamic programming. The branching algorithms can reduce the base of the exponential time scaling to, e.g., $1.1996^{|V|}|V|^{O(1)}$ [50], while dynamic programming approaches [14, 20] works better for graphs with a small treewidth $\mathrm{tw}(G)$; these methods can produce algorithms of complexity $O(2^{\mathrm{tw}(G)}\mathrm{tw}(G)|V|)$. People are eager to find better algorithms for solving the MIS problem; not only for it has a wide range of applications in scheduling, logistics, social network analysis, bioinformatics, wireless networks and telecommunication, map labelling, computer vision, etc. [12, 49], but also for it is a well know NP-complete problem that can be mapped from to many other important combinatorial optimization problems in polynomial time such as circuit satisfiability [], the maximum clique problem and the maximum vertex cover [40] problems.

In this paper, we do not limit our discussion to finding one of the maximum indepedent sets and its size. There are many other interesting problems pertaining to independent sets, such as the number of independent sets at a given size and enumerating independent sets at a give size or what we call computing *properties*. Some of these problems are of great interest in physics applications such as the hard-core lattice gas model [15, 17] in statistical mechanics and the Rydberg hamiltonian with neutral atoms [43] [**ST: cite experiment when it's ready**]; they can, for example, be used to understand phase transitions, to analyse the

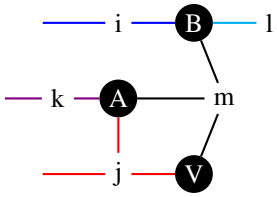
* **Funding:** ...

†XXX (email, website).
‡yyyyy (yyyy, email).

overlap gap properties [22, 21] and to identify harder graphs in an ensemble of graphs [**ST: cite experiment**]. The problem of computing the independence polynomial belongs to the complexity class #P-hard. One approach to exactly compute the independence polynomial requires a computational time of $O(1.442^{|V|})$ [18] [**JG: I am not sure about this complexity, this is based on the naive analysis of theorem 2.2 in [18]**]. There are some interesting works on efficiently approximating the independence polynomial [26], but in this work, we focus on computing the independence polynomial exactly. We introduce a tensor-network based framework to compute the various properties pertaining to independent sets. We map the independent set problem into generic tensor network contraction with specially designed tensor element algebra. Its algorithmic complexity is similar to that of dynamic programming, i.e. scales exponentially as tw(G). We benchmark our algorithms by computing various independent set properties on central processing units (CPUs) and graphics processing units (GPUs) and show its good performance on sparse graphs. Its high performance benefits from recent advances in random tensor-network contraction for the purposes of quantum circuit simulations [25, 42, 31]. People developed algorithms to optimize the contraction order for tensor networks with thousands of vertices, and the optimized contraction order gives us a near optimal space complexity $2^{\sim \mathrm{tw}(G)}$ to contract the tensor network. For cases when the tensors being too large to fit into a GPU, slicing technique can be used to further reduce the space consumption. Lastly, we provide a few examples and show that this toolbox can be used to calculate the hard-square entropy constant, Euler characteristics of the independence complex, partition functions, and finite-temperature phase transitions on square-lattice graphs. The implementation of our algorithm is publicly available online: [**?**].

2. Tensor networks. Tensor network [13, 41] is also known as einsum, factor graph or sum-product network [9] in different contexts, it can be viewed as a generalization of matrix multiplication to multiple tensors contraction. People often use Einstein's notation to represent a tensor network, e.g. the matrix multiplication between two matrices A and B can be represented in Einstein's notation as $C_{ik} = A_{ij}B_{jk}$, where we use a label in subscripts to represent a degree of freedom. We enumerate these degree of freedoms and accumulate the product of tensor elements to the output tensor. In the standard notation of Einstein's summation or tensor network in physics, each index appears precisely twice. Hence people can represent a tensor network as a simple graph, where a tensor is a vertex and two tensors sharing the same label are connected. In the main text, we do not have this restriction. An index can appear arbitrary times. The graphical representation of our generlized tensor network is hypergraph, in which an edge (label) can be shared by an arbitrary number of vertices (tensors). These two notations are equivalent in representation power because one can easily translate a generalized tensor network to the standard notation by adding δ tensors - a high dimensional equivalence of identity matrix. However, introducing δ tensors can potentially increase the contraction complexity a lot. We illustrate this point in Appendix B. [**JG: Is it better to introduce the contraction orders here or in example 2?**]

Example 1. $C_{ijk} = A_{jkm}B_{mil}V_{jm}$ is a tensor network that can be evaluated as $C_{ijk} = \sum_{ml} A_{jkm}B_{mil}V_{jm}$. Its hypergraph representation is shown below, where we use different colors to represent different hyperedges.



3. Generic programming. In previous works relating tensor networks and combinatoric problems [32, 8], the elements in the tensor networks are limited to standard number types such as floating point numbers and integers. Owning to the development of modern compiling technology, we no longer need to limit our imagination to standard number types. One of the key concepts that push the technology forward is called generic programming:

Definition 3.1 (Generic programming). *Generic programming is an approach to programming that focuses on designing algorithms and data structures so that they work in the most general setting without loss of efficiency. [47]*

This definition of generic programming contains two major aspects: a single program works in the most general setting and efficiency. To understand the first aspect, suppose we want to write a function that raises an element to a power, $f(x, n) := x^n$. One can easily write a function for standard number types that computes the power of x in $\log(n)$ steps using the multiply and square trick. Generic programming does not require x to be a standard number type, instead it treats x as an element with an associative multiplication operation $\odot$ and a multiplicative identity $\mathbb{1}$. In such a way, when the program takes a matrix as an input, it computes the matrix power without extra efforts. The second aspect is about performance. For dynamically typed languages such as Python, one can easily write very general codes, but the efficiency is not guaranteed; for example, the speed of computing the matrix multiplication between two numpy arrays with python objects as elements is much slower than statically typed languages such as C++ and Julia [7]. C++ uses templates for generic programming while Julia takes advantage of just-in-time compilation and multiple dispatch. When these languages "see" a new input type, the compiler can recompile the generic program for the new type. A myriad of optimizations can be done during the compilation, such as inlining immutable elements with fixed sizes in an array to decrease the cache miss rate when accessing data. In Julia, these inlined arrays can even be compiled to GPU devices for faster computation [6].

This motivates us to think about what is the most general element type that is allowed in a tensor network contraction program. We find that as long as the algebra of tensor elements forms a commutative semiring, the tensor network contraction result will be valid and be independent of the contraction order. A commutative semiring is a semiring with its multiplication operation being commutative, and a semiring is a ring without additive inverse. 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 any 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. In the following sections, we show how to compute the properties of independent sets by designing tensor element types as commutative semirings while keeping the tensor network generic [47]. Table 1 summarizes the properties that can be solved by various tensor element types.

4. 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. The total number of independent sets is thus equal to $I(G, 1)$. To compute the independence polynomial of a graph $G = (V, E)$, we reinterpret this problem as a tensor network contraction problem. We map a vertex $i \in V$ to a label $s_i \in \{0, 1\}$ of dimension 2 in a tensor network, where we use 0 (1) to denote a vertex absent (exists) in the set. For each label s_i, we defined a prametrized rank-one vertex tensor $W(x_i)$ indexed by it as

$$W(x_i) = \begin{pmatrix} 1 \\ x_i \end{pmatrix}. \tag{4.2}$$

On each edge (u, v), we define a matrix B indexed by (s_u, s_v) as

$$B = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \tag{4.3}$$

The corresponding tensor network contraction gives

$$P(G, \{x_1, x_2, \ldots, x_{|V|}\}) = \sum_{s_1, s_2, \ldots, s_{|V|}=0}^{1} \prod_{i=1}^{|V|} W(x_i)_{s_i} \prod_{(i,j)\in E(G)} B_{s_i s_j}, \tag{4.4}$$

where the summation runs over all vertex configurations $\{s_1, s_2, \ldots, s_{|V|}\}$ and accumulates the product of tensor elements to the output P (see Example 2 for a concrete example). The edge

element type	purpose
$\mathbb{R}$	real number for counting independent sets
$\mathbb{C}$	complex number for fitting the independence polynomial with fast Fourier transform
GF(p) (Eq. (4.8))	finite field algebra for fitting the independence polynomial exactly using number theory
T (Eq. (5.3: T))	tropical number for finding the independence number
P1 (Eq. (5.2: P1))	polynomial truncated to largest order, i.e. the tropical number with counting field for finding the independence number and the number of MISs
P2 (Eq. (5.5: P2))	polynomial truncated to largest two orders for counting MISs and independent sets of size $\alpha(G) - 1$
PN (Eq. (4.5: PN))	polynomial number for computing the independence polynomial directly
P1+S1	P1 with configuration sampler (Eq. (6.5: S1)) as its counting field for finding the independence number and one MIS
P1+SN (Eq. (6.3: P1+SN))	P1 with configuration enumerator (Eq. (6.1: SN)) as its counting field for finding the independence number and enumeration of all MISs

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

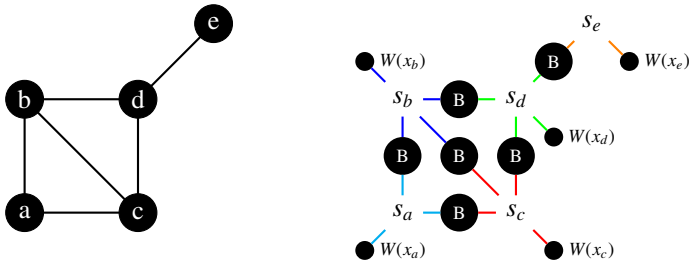
[ST: we should include maximal independence polynomial and counting, enumeration in this table as well.] [JG: I do not think so. Network structure and element types are two degree of freedoms, maximal IS is different network structure. We can use the above algebra in any network structure.]

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). 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.

To evalute Eq. (4.4), summing up the products directly is apparently computational inefficient. The standard approach to evaluate a tensor network is to contract two tensors a time with a certain order utilizing the associativity and commutativity of tensor elements. A good contraction order can reduce the time complexity significantly, at the cost of having a space overhead of $O(2^{\mathrm{tw}(G)})$ [39]. 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 2. 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. We attach a vertex tensor on each hyperedge and an edge tensor between two hyperedges if vertices of the two hyperedges are connected in the original graph.



The contraction of this tensor network can be done in a pairwise order utilizing the associativity, additive commutativity and multiplicative commutativity of tensor elements:

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

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. Then we can treat the tensor elements as polynomials and evaluate the polynomial directly. Let us create a polynomial type, and represent a polynomial $a_0 + a_1 x + \ldots + a_k x^k$ 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

$$
\begin{aligned}
a \oplus b &= (a_0 + b_0, a_1 + b_1, \ldots, a_{\max(k_a,k_b)} + b_{\max(k_a,k_b)}), \\
a \odot b &= (a_0 + b_0, a_1 b_0 + a_0 b_1, a_2 b_0 + a_1 b_1 + a_0 b_2, \ldots, a_{k_a} b_{k_b}), \\
\mathbb{0} &= (), \\
\mathbb{1} &= (1).
\end{aligned}
\tag{4.5: PN}
$$

Here, the multiplication operation can be evaluated efficiently using the convolution theorem [45]. 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

$$
W^{\text{poly}} = \begin{pmatrix} \mathbb{1} \\ (0, 1) \end{pmatrix}, \qquad B^{\text{poly}} = \begin{pmatrix} \mathbb{1} & \mathbb{1} \\ \mathbb{1} & \mathbb{0} \end{pmatrix}. \tag{4.6}
$$

By contracting the tensor network with the polynomial type, we have contraction result the exact representation of the independence polynomial. However, using polynomial numbers 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 pairs 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:

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \dots & x_0^{\alpha(G)} \\ 1 & x_1 & x_1^2 & \dots & x_1^{\alpha(G)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{\alpha(G)} & x_{\alpha(G)}^2 & \dots & x_{\alpha(G)}^{\alpha(G)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{\alpha(G)} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{\alpha(G)} \end{pmatrix}. \tag{4.7}$$

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 GPU devices. These problems can be solved by introducing a finite-field algebra GF(p):

$$\begin{aligned} x \oplus y &= x + y \pmod p, \\ x \odot y &= xy \pmod p, \\ \mathbb{0} &= 0, \\ \mathbb{1} &= 1. \end{aligned} \tag{4.8}$$

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

1. One can use Gaussian elimination [24] to solve the linear equation Eq. (4.7) 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 $\{p_1, p_2, \dots, p_n\}$, $x \pmod{p_1 \times p_2 \times \dots \times p_n}$ 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. This is an iterative algorithm that iterates over a sequence of large prime numbers until convergence. In each iteration, we choose a large prime number p, and contract the tensor networks to evaluate the polynomial for each variable $\chi = (x_0, x_1, \dots, x_{\alpha(G)})$ on GF(p) and denote the outputs as $(y_0, y_1, \dots, y_{\alpha(G)})$ (mod p). Then we solve Eq. (4.7) using the gaussian elimination on GF(p) to find the coefficient modulo p, $A_p \equiv (a_0, a_1, \dots, a_{\alpha(G)}) \pmod p$. As the last step of each iteration, we apply the Chinese remainder theorem to update $A \pmod P$ to $A \pmod{P \times p}$, where P is a product of all prime numbers chosen in previous iterations. If this number does not change compared with the previous iteration, it indicates the convergence of result and the program terminates. All computations are done with integers of fixed width W except the last step of applying the Chinese remainder theorem, where we use arbitrary precision integers to represent the counting. In Appendix D, we provide another method to solve the linear equation using discrete Fourier transformation.

5. Maximum independent sets and its counting.

Algorithm 4.1 Computing the independence polynomial exactly without integer overflow

Let $P = 1$, W be the integer width, vector $\chi = (0, 1, 2, \ldots, \alpha(G))$, matrix $X_{ij} = (\chi_i)^j$, where $i, j = 0, 1, \ldots, \alpha(G)$
while *true* **do**
  compute the largest prime p that $\gcd(p, P) = 1$ and $p < 2^W$
  for $i = 0 \ldots \alpha(G)$ **do**
    $y_i \pmod p$ = contract_tensor_network($\chi_i \pmod p$) ; // on GF(p)
  end
  $A_p = (a_0, a_1, \ldots, a_{\alpha(G)}) \pmod p$ = gaussian_elimination($X, (y_0, y_1, \ldots, y_{\alpha G}) \pmod p$)
  $A_{P\times p}$ = chinese_remainder(A_P, A_p)
  if $A_P = A_{P\times p}$ **then**
    return A_P ; // converged
  end
  $P = P \times p$
end

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 new type with two fields: the largest exponent k and its coefficient a_k. From this, we can define a new algebra as

$$\begin{aligned} a_x \infty^x \oplus a_y \infty^y &= \begin{cases} (a_x + a_y)\infty^{\max(x,y)}, & x = y \\ a_y \infty^{\max(x,y)}, & x < y \\ a_x \infty^{\max(x,y)}, & x > y \end{cases} \\ a_x \infty^x \odot a_y \infty^y &= a_x a_y \infty^{x+y} \\ \mathbb{0} &= 0\infty^{-\infty} \\ \mathbb{1} &= 1\infty^0. \end{aligned} \tag{5.2: P1}$$

When we are only interested in knowing the MIS size, we can drop the counting field. The algebra of the exponents becomes the max-plus tropical algebra [37, 40].:

$$\begin{aligned} x \oplus y &= \max(x, y) \\ x \odot y &= x + y \\ \mathbb{0} &= -\infty \\ \mathbb{1} &= 0. \end{aligned} \tag{5.3: T}$$

This algebra is the same as the one used in Liu et al. [36] to calculate and count spin glass ground states. For independent set calculations here, the vertex tensor and edge tensor becomes:

$$W^{\text{tropical}} = \begin{pmatrix} \mathbb{1} \\ \infty \end{pmatrix}, \qquad B^{\text{tropical}} = \begin{pmatrix} \mathbb{1} & \mathbb{1} \\ \mathbb{1} & \mathbb{0} \end{pmatrix}. \tag{5.4}$$

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. (4.5: PN):

$$\begin{aligned} a \oplus b &= (a_{\max(k_a,k_b)-1} + b_{\max(k_a,k_b)-1}, a_{\max(k_a,k_b)} + b_{\max(k_a,k_b)}), \\ a \odot b &= (a_{k_a-1}b_{k_b} + a_{k_a}b_{k_b-1}, a_{k_a}b_{k_b}), \\ \mathbb{0} &= (), \\ \mathbb{1} &= (1). \end{aligned} \tag{5.5: P2}$$

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. As will be shown below, this algebra can also be extended to enumerate those large-size independent sets.

6. Enumeration of independent sets.

6.1. Set algebra for configuration enumeration. The enumeration problems of independent sets are also interesting and has been studied extensively in the literature [11, 16, 30], including, for example, the enumeration of all independent sets, the enumeration of all maximal independent sets, or the enumeration of all MISs. Here, the term "maximal" has a different meaning from "maximum" which we discussed above; a maximal independent set is an independent set that is not a subset of any other independent set, but its size may not be the maximum. We can also compute the maximal independence polynomial and enumerate maximal independent sets with a different tensor network mapping, and this is described in Sec. C.1. To enumerate all independent sets, we designed an algebra defined on sets of bitstrings.

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

where s and t are each a set of $|V|$-bit strings and $\vee^\circ$ is the bitwise OR operation over two bit strings.

Example 3. For elements being bit vectors of length 5, we have the following set algebra

$$\{00001\} \oplus \{01110, 01000\} = \{01110, 01000\} \oplus \{00001\} = \{00001, 01110, 01000\}$$
$$\{00001\} \oplus \{\} = \{00001\}$$

$$\{00001\} \odot \{01110, 01000\} = \{01110, 01000\} \odot \{00001\} = \{01111, 01001\}$$
$$\{00001\} \odot \{\} = \{\}$$
$$\{00001\} \odot \{00000\} = \{00001\}$$

To enumerate all independent sets, we initialize variable x_i in the vertex tensor 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

$$W^{\text{enum}}(\{\boldsymbol{e}_i\}) = \begin{pmatrix} \mathbb{1} \\ \{\boldsymbol{e}_i\} \end{pmatrix}, \qquad B^{\text{enum}} = \begin{pmatrix} \mathbb{1} & \mathbb{1} \\ \mathbb{1} & \mathbb{0} \end{pmatrix}. \tag{6.2}$$

This set algebra can serve as the coefficients in Eq. (5.2: P1) to enumerate all MISs, Eq. (4.5: PN) to enumerate independent sets of different sizes, or Eq. (5.5: P2) to enumerate all independent sets of size $\alpha(G)$ and $\alpha(G) - 1$. For example, to enumerate only the MISs, with the tropical algebra, we define $s(k) = (s_k, k)$, where the coefficients follows the algebra in Eq. (6.1: SN) and the orders follows the max-plus tropical algebra. The combined operations become:

$$\begin{aligned} s_x \infty^x \oplus s_y \infty^y &= \begin{cases} (s_x \cup s_y)\infty^{\max(x,y)}, & x = y \\ s_y \infty^{\max(x,y)}, & x < y \\ s_x \infty^{\max(x,y)}, & x > y \end{cases} \\ s_x \infty^x \odot s_y \infty^y &= \{\sigma \vee^\circ \tau | \sigma \in s_x, \tau \in s_y\} \infty^{x+y}, \\ \mathbb{0} &= \{\}\infty^{-\infty}, \\ \mathbb{1} &= \{0^{\otimes |V|}\}\infty^0. \end{aligned} \tag{6.3: P1+SN}$$

Clearly, the vertex tensor and edge tensor become

$$W^{\text{MISenum}}(\{\boldsymbol{e}_i\}) = \begin{pmatrix} \mathbb{1} \\ \{\boldsymbol{e}_i\}\infty^1 \end{pmatrix}, \qquad B^{\text{MISenum}} = \begin{pmatrix} \mathbb{1} & \mathbb{1} \\ \mathbb{1} & \mathbb{0} \end{pmatrix}. \tag{6.4}$$

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: SN),

$$\begin{aligned} \sigma \oplus \tau &= \text{select}(\sigma, \tau), \\ \sigma \odot \tau &= (\sigma \vee^\circ \tau), \\ \mathbb{0} &= 1^{\otimes |V|}, \\ \mathbb{1} &= 0^{\otimes |V|}, \end{aligned} \tag{6.5: S1}$$

where the `select` function picks one of σ and τ by some criteria to make the algebra commutative and associative, e.g. by picking the one with a smaller integer value.

6.2. Bounding the MIS enumeration space. When we use the algebra in Eq. (6.3: P1+SN) 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 bound the searching space using the MIS size $\alpha(G)$. 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: P1+SN), 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 propagating the gradients. To derive the back-propagation rule for tropical 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}. Intuitively, one can use this relation to indentify elements with nonzero gradients in A and B directly, however, doing this might make us lost the advantage of using BLAS libraries [1]. 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 [1]. 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 be $\overline{C}$; 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 by setting the tensor elements masked false to zero during contraction.

7. Benchmarks and case studies.

7.1. Performance benchmarks. We run a single thread benchmark on CPU Intel(R) Xeon(R) CPU E5-2686 v4 @ 2.30GHz, and its CUDA version on a GPU Tesla V100. The results are summarized in Figure 1. The graphs that we use in benchmarks are random three regular graphs, a typical type of sparse graphs that has a small tree width asymptotically smaller than $|V|/6$ [19].

Figure (a) shows the time and space complexity obtained with tensor network optimization algorithm in Ref. [31], where the space complexity is the same as the tree width of the problem graph. One can see all the computing times in figure (b), (c) and (d) have a strong correlation with the treewidth. Among these benchmarks, computational tasks with data types `T (CPU)`, $\mathbb{R}$ `(CPU)`, $\mathbb{R}$ `(GPU)`, $\mathbb{C}$ `(CPU)`, $\mathbb{C}$ `(GPU)` and `T+bounding (CPU)` can utilize fast BLAS functions, hence are much faster comparing to non-BLAS methods in the same category. GPU computes much faster than CPU in all cases when the problem scale is large enough so that the actual computing time is comparable or larger than the launching time of CUDA kernels. Most algebras can be computed on GPU, except those requiring dynamic sized structures, i.e. `PN` and `SN`. In figure (c), one can see the Fourier transformation

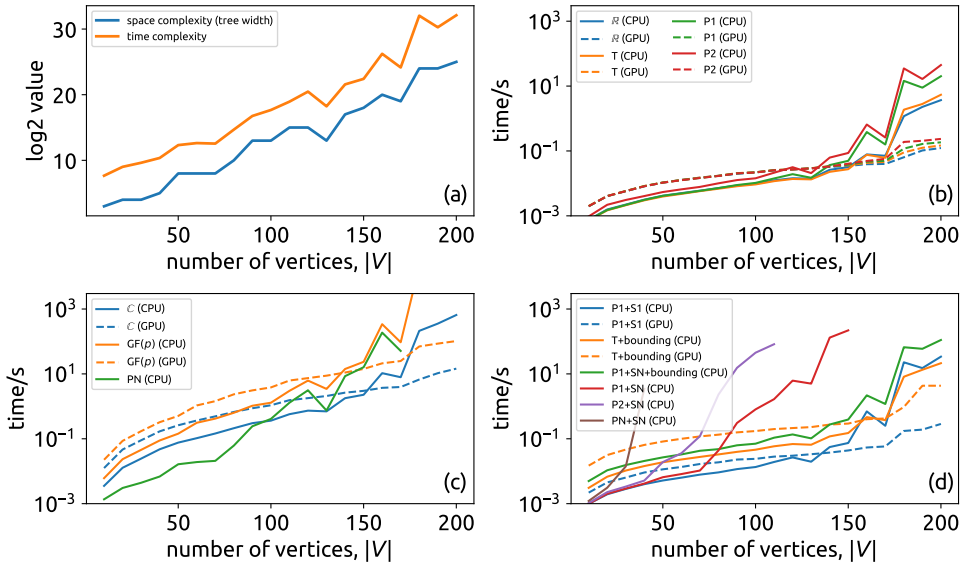


Figure 1: Benchmark results for computing different properties of independent sets of a random three regular graph with different tensor element types. The time in these plots only includes tensor network contraction, without taking the contraction order finding and just in time compiling time into account. [**ST: for each graph size, is there only one graph?**][**JG: yes**] (a) tree width versus the number of vertices for the benchmarked graphs. Legends are the data types that we used in the computation, one can find the corresponding computed property in Table 1. (b) The computing time for calculating the independence number and for counting of the number of independent sets, the number of MISs, and the number of independent sets having size $\alpha(G)$ and $\alpha(G) - 1$. (c) The computing time for calculating the independence polynomials with different approaches. (d) The computing time for configuration enumeration, including the enumeration of all independent set configurations, a single MIS configuration, all MIS configurations, all independent set configurations having size $\alpha(G)$ and $\alpha(G) - 1$, with or without bounding the enumeration space. [**ST: suggestions for the figure: 1. the legend seems to overlap the lines, especially the large subfigure. Not sure what's a better way to include the legend there. 2. I see you guys don't typically use markers. We typically use markers to distinguish the different lines, as well as using the color. Especially, for some journals, there is a physical non-color copy, so some requires the figures to be understandable without colors.**] [**JG: If we use markers, the data poitns are too dense. I prefer not adding markers, not much people read papers with kindle or on paper.**] [**JG: I used a smaller fontsize, there are still some overlaps. Does it look better (or worse) now?**]

based method is the fastest in computing the independence polynomial, however, it maybe suffer from the round off errors. The finite field (GF(p)) approach is the only method that does not incur round-off errors and can run on a GPU. In figure (d), one can see the technique to bound the enumeration space improves the performance for more than one order in enumerating the MISs. Bounding can also reduce the memory usage significantly, without which the largest computable graph size is only $\sim$ 150 on a 32G memory device.

[**JG: I am here!**]

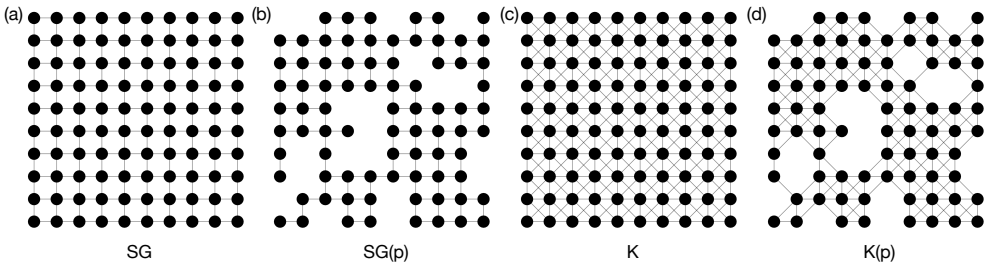


Figure 2: The types of graphs used in the benchmark and case studies. The lattice dimensions are $L \times L$. (a) Square grid graph denoted as SG. (b) Square grid graph with a filling factor $p = 0.8$, denoted as SG(0.8). (c) King's graph denoted as K. (d) King's graph with a filling factor $p = 0.8$, denoted as K(0.8).

7.2. Example case studies. In this section, we give a few examples where the different properties of independence sets are used. In all the examples, the types of graphs we used are shown in Figure 2, where vertices are all placed on square lattices with lattice dimensions $L \times L$. The graphs include: the square grid graphs, denoted as SG; the square grid graphs with a filling factor p, denoted as SG(p), where $\lfloor pL^2 \rfloor$ square grids are occupied with vertices; the King's graphs, denoted as K; the King's graphs with a filling factor p, denoted as K(p).

7.2.1. Number of independent sets and hard-square entropy constant. The number of independent sets for square grid graphs of size $L \times L$ form a well-known integer sequence (OEIS A006506), which is thought as a two-dimensional generalization of the Fibonacci numbers.

7.2.2. Euler characteristics of independence complex.

7.2.3. Partition functions and finite-temperature phase transitions.

7.2.4. Computing the Hamming distance statistics. With the ability to enumerate configurations, we can analyse the landscape of the target problem, like the overlap gap property [22, 21]. We compute all MIS configurations for two random 3 regular graph instances of size 200, and show the Hamming distance statistics in Fig. 3. Some figure has single peak structure while some other has multiple peaks. Multiple peak structure indicates disconnected clusters in the configuration space.

8. Discussion and conclusion. In this paper, we introduce an abstract algebra formalism to compute properties of independent sets. The properties include the independence number, number of independent set of a given size, and enumeration of independent sets of a give size. For each property, we design a algebra being a commutative semiring, and map the property computation to the contraction of a tensor network with such element type. We call this method generic programming tensor network, and its power is not limited to the independent set problem. In Appendix C, we show how to map the maximal independent set problem, matching problem, k-coloring problem, max cut problem and set packing problem to tensor networks. Moreover, since the independence polynomial is closely related to the matching polynomial [34], the clique polynomial [28], and the vertex cover polynomial [4], our algorithm to compute the independence polynomial can also be used to compute these graph polynomials. We show some of the Julia language implementations in Appendix A and you will find it surprisingly short. A complete implementation can be found in our Github repository [2].

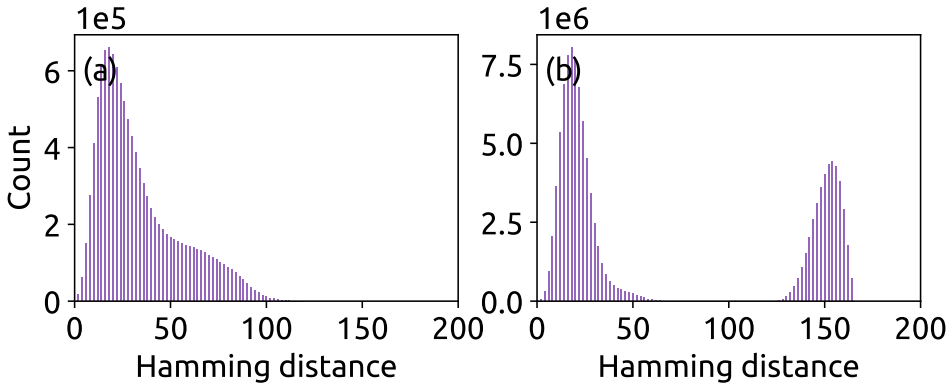


Figure 3: The statistics of Hamming distances between two MIS configurations for two specific three regular graph instance of size 200.

Acknowledgments. We would like to thank Pan Zhang for sharing his python code for optimizing contraction orders of a tensor network. We would like to acknowledge Sepehr Ebadi, Maddie Cain and Leo Zhou for popping up inspiring questions about independent sets, their questions are driving forces of this project. Thank Chris Elord for helping us writing the fastest matrix multiplication libary for tropical numbers, TropicalGEMM.jl, he is a man of speed! [**JG: funding information**]

REFERENCES

[1] https://github.com/TensorBFS/TropicalGEMM.jl.
[2] https://github.com/Happy-Diode/GraphTensorNetworks.jl.
[3] https://github.com/JuliaGraphs/Graphs.jl.
[4] S. Akbari and M. R. Oboudi, *On the edge cover polynomial of a graph*, European Journal of Combinatorics, 34 (2013), pp. 297–321.
[5] S. F. Barr, *Courcelle's Theorem: Overview and Applications*, PhD thesis, Oberlin College, 2020.
[6] 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.
[7] 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.
[8] J. Biamonte and V. Bergholm, *Tensor networks in a nutshell*, 2017, https://arxiv.org/abs/1708.00006.
[9] C. M. Bishop, *Pattern Recognition and Machine Learning*, Springer, 2006.
[10] M. Bousquet-Mélou, S. Linusson, and E. Nevo, *On the independence complex of square grids*, Journal of Algebraic combinatorics, 27 (2008), pp. 423–450.
[11] C. Bron and J. Kerbosch, *Algorithm 457: finding all cliques of an undirected graph*, Communications of the ACM, 16 (1973), pp. 575–577.
[12] S. Butenko and P. M. Pardalos, *Maximum Independent Set and Related Problems, with Applications*, PhD thesis, USA, 2003. AAI3120100.
[13] 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.
[14] B. Courcelle, *The monadic second-order logic of graphs. i. recognizable sets of finite graphs*, Information and computation, 85 (1990), pp. 12–75.
[15] J. C. Dyre, *Simple liquids' quasiuniversality and the hard-sphere paradigm*, Journal of Physics: Condensed Matter, 28 (2016), p. 323001.
[16] 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.

[17] 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.

[18] G. M. Ferrin, *Independence polynomials*, (2014).

[19] F. V. Fomin and K. Høie, *Pathwidth of cubic graphs and exact algorithms*, Information Processing Letters, 97 (2006), pp. 191–196.

[20] F. V. Fomin and P. Kaski, *Exact exponential algorithms*, Communications of the ACM, 56 (2013), pp. 80–88.

[21] D. Gamarnik and A. Jagannath, *The overlap gap property and approximate message passing algorithms for p-spin models*, 2019, https://arxiv.org/abs/1911.06943.

[22] D. Gamarnik and M. Sudan, *Limits of local algorithms over sparse random graphs*, 2013, https://arxiv.org/abs/1304.1831.

[23] 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.

[24] G. H. Golub and C. F. Van Loan, *Matrix computations*, vol. 3, JHU press, 2013.

[25] 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.

[26] 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.

[27] J. Hastad, *Clique is hard to approximate within $n^{1-\epsilon}$*, in Proceedings of 37th Conference on Foundations of Computer Science, IEEE, 1996, pp. 627–636.

[28] C. Hoede and X. Li, *Clique polynomials and independent set polynomials of graphs*, Discrete Mathematics, 125 (1994), pp. 219–228.

[29] 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.

[30] 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.

[31] G. Kalachev, P. Panteleev, and M.-H. Yung, *Recursive multi-tensor contraction for xeb verification of quantum circuits*, 2021, https://arxiv.org/abs/2108.05665.

[32] S. Kourtis, C. Chamon, E. Mucciolo, and A. Ruckenstein, *Fast counting with tensor networks*, SciPost Physics, 7 (2019), https://doi.org/10.21468/scipostphys.7.5.060, http://dx.doi.org/10.21468/SciPostPhys.7.5.060.

[33] T.-D. Lee and C.-N. Yang, *Statistical theory of equations of state and phase transitions. ii. lattice gas and ising model*, Physical Review, 87 (1952), p. 410.

[34] 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.

[35] V. E. Levit and E. Mandrescu, *The independence polynomial of a graph at -1*, 2009, https://arxiv.org/abs/0904.4819.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] C. Moore and S. Mertens, *The nature of computation*, OUP Oxford, 2011.

[41] R. Orús, *A practical introduction to tensor networks: Matrix product states and projected entangled pair states*, Annals of Physics, 349 (2014), pp. 117–158.

[42] F. Pan and P. Zhang, *Simulating the sycamore quantum supremacy circuits*, 2021, https://arxiv.org/abs/2103.03074.

[43] 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).

[44] J. M. Robson, *Algorithms for maximum independent sets*, Journal of Algorithms, 7 (1986), pp. 425–440.

[45] A. Schönhage and V. Strassen, *Schnelle multiplikation grosser zahlen*, Computing, 7 (1971), pp. 281–292.

[46] A. D. Scott and A. D. Sokal, *The repulsive lattice gas, the independent-set polynomial, and the*

lovász local lemma, Journal of Statistical Physics, 118 (2005), p. 1151–1261, https://doi.org/10.1007/s10955-004-2055-4, http://dx.doi.org/10.1007/s10955-004-2055-4.

[47] A. A. Stepanov and D. E. Rose, *From mathematics to generic programming*, Pearson Education, 2014.

[48] R. E. Tarjan and A. E. Trojanowski, *Finding a maximum independent set*, SIAM Journal on Computing, 6 (1977), pp. 537–546.

[49] 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.

[50] 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.

[51] C.-N. Yang and T.-D. Lee, *Statistical theory of equations of state and phase transitions. i. theory of condensation*, Physical Review, 87 (1952), p. 404.

Appendix A. Technical guide. Before showing the codes, we want to introduce some Julia packages that playing important roles in our code base.

OMEinsum and OMEinsumContractionOrders are packages providing the support for Einstein's (or tensor network) notation and contraction order optimizations. `OMEinsumContractionOrders` implements state of art algorithms for finding the optimal contraction order for the an tensor network, including the KaHypar+Greedy [25, 42] and local transformation based approaches [31],

TropicalNumbers and TropicalGEMM are packages providing tropical number and efficient tropical matrix multiplication,

Graphs is a package providing graph utilities, like random regular graph generator,

Polynomials is a package providing polynomial algebra and polynomial fitting,

Mods and Primes are packages providing finite field algebra and prime number generations.

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

```
pkg> add OMEinsum Graphs Mods Primes Polynomials TropicalNumbers OMEinsumContractionOrders
```

It may surprise you that the Julia implementation of algorithms introduced in the paper is so short that except the bounding algorithm, 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(50, 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)]
    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))")

########## FINDING OPTIMAL CONFIGURATIONS ###########

# define the set algebra
struct ConfigEnumerator{N}
    # NOTE: BitVector is dynamic, can be very slow, check our repo for the static version
    data::Vector{BitVector}
end
function Base.:+(x::ConfigEnumerator{N}, y::ConfigEnumerator{N}) where {N}
    res = ConfigEnumerator{N}(vcat(x.data, y.data))
    return res
end
function Base.:*(x::ConfigEnumerator{L}, y::ConfigEnumerator{L}) where {L}
    M, N = length(x.data), length(y.data)
    z = Vector{BitVector}(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}(z)
end
Base.zero(::Type{ConfigEnumerator{N}}) where {N} = ConfigEnumerator{N}(BitVector[])
Base.one(::Type{ConfigEnumerator{N}}) where {N} = ConfigEnumerator{N}([falses(N)])

# the algebra sampling one of the configurations
struct ConfigSampler{N}
    data::BitVector
end

function Base.:+(x::ConfigSampler{N}, y::ConfigSampler{N}) where {N}  # biased sampling: return
      `x`, maybe using random sampler is better.
    return x  # randomly pick one
end
function Base.:*(x::ConfigSampler{L}, y::ConfigSampler{L}) where {L}
    ConfigSampler{L}(x.data .| y.data)
end

Base.zero(::Type{ConfigSampler{N}}) where {N} = ConfigSampler{N}(trues(N))
Base.one(::Type{ConfigSampler{N}}) where {N} = ConfigSampler{N}(falses(N))

# 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
    xs = map(getixs(OMEinsum.flatten(code))) do ix
        T = all ? CountingTropical{Float64, ConfigEnumerator{N}} : CountingTropical{Float64,
     ConfigSampler{N}}
        if length(ix) == 2
            return [one(T) one(T); one(T) zero(T)]
        else
            s = falses(N)
            s[vertex_index[ix[1]]] = true  # one hot vector
            if all
                [one(T), T(1.0, ConfigEnumerator{N}([s]))]
            else
                [one(T), T(1.0, ConfigSampler{N}(s))]
            end
        end
    end
   return code(xs...)
end
```

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

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

Please checkout our GitHub repository for the full featured version:
https://github.com/Happy-Diode/GraphTensorNetworks.jl
Here is a short introduction to the functionalities in this libary.

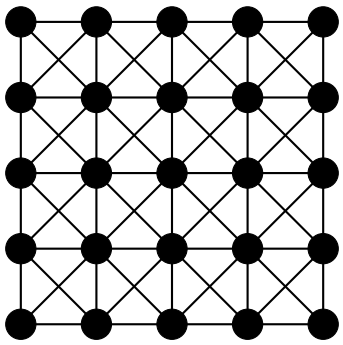
```
julia> using GraphTensorNetworks, Random, Graphs

julia> graph = (Random.seed!(2); Graphs.smallgraph(:petersen))
{10, 15} undirected simple Int64 graph

julia> problem = Independence(graph; optimizer=TreeSA(sc_target=0, sc_weight=1.0, ntrials=10, βs=
    0.01:0.1:15.0, niters=20, rw_weight=0.2));
┌ Warning: target space complexity not found, got: 4.0, with time complexity 7.965784284662087,
    read-right complexity 8.661778097771988.
└ @ OMEinsumContractionOrders ~/.julia/dev/OMEinsumContractionOrders/src/treesa.jl:71
time/space complexity is (7.965784284662086, 4.0)

# maximum independent set size
julia> solve(problem, "size max")
0-dimensional Array{TropicalNumbers.TropicalF64, 0}:
4.0ₜ

# all independent sets
julia> solve(problem, "counting sum")
0-dimensional Array{Float64, 0}:
76.0

# counting maximum independent sets
julia> solve(problem, "counting max")
0-dimensional Array{TropicalNumbers.CountingTropicalF64, 0}:
(4.0, 5.0)ₜ

# counting independent sets of max two sizes
julia> solve(problem, "counting max2")
0-dimensional Array{Max2Poly{Float64, Float64}, 0}:
30.0*x^3 + 5.0*x^4

# using `Polynomial` type
julia> solve(problem, "counting all")
0-dimensional Array{Polynomial{Float64, :x}, 0}:
Polynomial(1.0 + 10.0*x + 30.0*x^2 + 30.0*x^3 + 5.0*x^4)

# using the finitefield approach
julia> solve(problem, "counting all (finitefield)")
0-dimensional Array{Polynomial{BigInt, :x}, 0}:
Polynomial(1 + 10*x + 30*x^2 + 30*x^3 + 5*x^4)

# using the fourier approach
julia> solve(problem, "counting all (fft)", r=1.0)
0-dimensional Array{Polynomial{ComplexF64, :x}, 0}:
Polynomial(1.0000000000000029 + 2.664535259100376e-16im + (10.000000000000004 - 1.95124353988574
    92e-16im)x + (30.0 - 1.9622216671393801e-16im)x^2 + (30.0 + 1.1553104311877194e-15im)x^3 +
    (5.0 - 1.030417436395244e-15im)x^4)

# one of MISs
julia> solve(problem, "config max")
0-dimensional Array{CountingTropical{Float64, ConfigSampler{10, 1, 1}}, 0}:
(4.0, ConfigSampler{10, 1, 1}(1010000011))ₜ

julia> solve(problem, "config max (bounded)")
0-dimensional Array{CountingTropical{Float64, ConfigSampler{10, 1, 1}}, 0}:
(4.0, ConfigSampler{10, 1, 1}(1010000011))ₜ
```

```
# enumerate all MISs
julia> solve(problem, "configs max")  # not recommended
0-dimensional Array{CountingTropical{Float64, ConfigEnumerator{10, 1, 1}}, 0}:
(4.0, {1010000011, 0100100110, 1001001100, 0010111000, 0101010001}),

julia> solve(problem, "configs max (bounded)")
0-dimensional Array{CountingTropical{Int64, ConfigEnumerator{10, 1, 1}}, 0}:
(4, {1010000011, 0100100110, 1001001100, 0010111000, 0101010001}),

# enumerate all MIS and MIS-1 configurations
julia> solve(problem, "configs max2")
0-dimensional Array{Max2Poly{ConfigEnumerator{10, 1, 1}, Float64}, 0}:
{0010101000, 0101000001, 0100100010, 0010100010, 0100000011, 0010000011, 1001001000, 1010001000,
       1001000001, 1010000001, 1010000010, 1000000011, 0100100100, 0000101100, 0101000100, 000100
      1100, 0000100110, 0100000110, 1001000100, 1000001100, 1000000110, 0100110000, 0000111000, 0
      101010000, 0001011000, 0010110000, 0010011000, 0001010001, 0100010001, 0010010001}*x^3 + {1
      010000011, 0100100110, 1001001100, 0010111000, 0101010001}*x^4

# enumerate all IS configurations
julia> solve(problem, "configs all")
0-dimensional Array{Polynomial{ConfigEnumerator{10, 1, 1}, :x}, 0}:
Polynomial({0000000000} + {0010000000, 0000100000, 0001000000, 0100000000, 0000001000, 000000000
      1, 0000000010, 1000000000, 0000000100, 0000010000}*x + {1000000010, 0010100000, 0010001000,
       0100100000, 0000101000, 0101000000, 0001001000, 0001000001, 0100000001, 0010000001, 000010
      0010, 0100000010, 0010000010, 0000000011, 1001000000, 1000001000, 1010000000, 1000000001, 0
      000000110, 0000100100, 0001000100, 0100000100, 0000001100, 1000000100, 0010010000, 00001100
      00, 0001010000, 0100010000, 0000011000, 0000010001}*x^2 + {1010000010, 1000000011, 00101010
      00, 0101000001, 0100100010, 0010100010, 0100000011, 0010000011, 1001001000, 1010001000, 100
      1000001, 1010000001, 0000100110, 0100000110, 0100100100, 0000101100, 0101000100, 0001001100
      , 1001000100, 1000001100, 1000000110, 0010110000, 0010011000, 0100110000, 0000111000, 01010
      10000, 0001011000, 0001010001, 0100010001, 0010010001}*x^3 + {1010000011, 0100100110, 10010
      01100, 0010111000, 0101010001}*x^4)
```

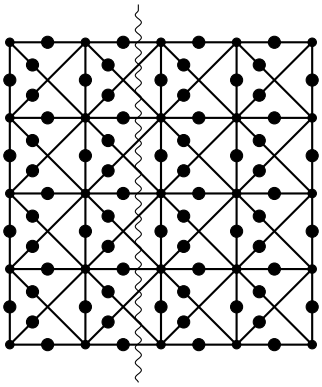
Appendix B. Why not using the standard tensor network notations. As we have mentioned in the main text, a standard tensor network notation inevitably introduces δ tensors to encode some graph problems. A δ tensor of rank d is defined as

$$\delta_{i_1,i_2,\ldots,i_d} = \begin{cases} 1, & i_1 = i_2 = \ldots = i_d, \\ 0, & \text{otherwise.} \end{cases} \tag{B.1}$$

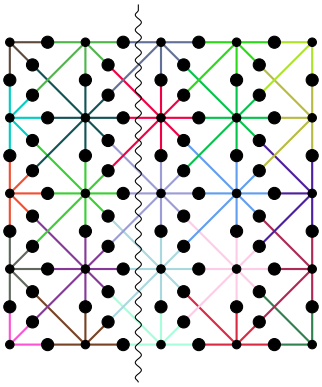
Given a graph



Its standard tensor network representation includes a δ tensors on each vertex, so its graphical representation is as bellow.



Here, we use a small circle on a vertex to represent a diagonal δ tensor. Its rank is 8 in the bulk. If we contract this tensor network in a naive column-wise order, the maximum intermediate tensor has a rank $\sim 3L$, requiring a storage of size $\approx 2^{3L}$. If we treat it as the following generalized tensor network



Here, we use different colors to distinguish different hyperedges. The vertex tensor is always rank 1 in this representation. Again, if we contract this tensor network in the column-wise contraction order, we can see the maximum intermediate tensor has rank $\sim L$ by counting the colors.

Appendix C. Generalizing to other graph problems. There are some other graph problems that can be encoded to 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. Dynamic programming can solve monadic second order logic (MSO) definable problem on a graph with small tree width efficiently [14, 5]. It can solve the maximum independent set problem in $O(2^k)n$, which is similar to the tensor network approach. The graph problem can be expressed in the tensor network language is clearly much less than that can be expressed by dynamic programming. For example, it is hard to determine whether a collection of vertices forms a path in a graph using the tensor network language, while this problem is MSO definable. As a gain, a tensor network has a nicer analytic property, which makes contraction order optimization, generic programming and utilizing BLAS libaries very convenient. In the following, we introduce some other problems that can be expressed by a tensor network.

C.1. Maximal independent sets and maximal cliques. Sometimes, one may be interested in knowing maximal solutions to understand why his or her program is trapped in

a local minimum. Then instead of counting all independent sets, the maximal independence polynomial counts the number of maximal independent sets of various sizes [29] would be more helpful. Concretely, it is defined as

$$I_{\max}(G, x) = \sum_{k=0}^{\alpha(G)} b_k x^k, \tag{C.1}$$

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 [23, 38], where the fastest algorithm currently has a runtime of $O(1.3642^{|V|})$ [23]. If we want to find an 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 [29].

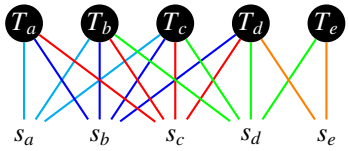
Let us denote the neighborhood of a vertex v as $N(v)$ and denote $N[v] = N(v) \cup \{v\}$. A maximal independent set I_m is an independent set where there exists no vertex v such that $I_m \cap N[v] = \emptyset$. We can modify the tensor network for computing the independence polynomial to include this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

$$T(x_v)_{s_1, s_2, \ldots, s_{|N(v)|}, s_v} = \begin{cases} s_v x_v & s_1 = s_2 = \ldots = s_{|N(v)|} = 0, \\ 1 - s_v & \text{otherwise.} \end{cases} \tag{C.2}$$

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

$$T(x_v) = \begin{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \\ \begin{pmatrix} x_v & 0 \\ 0 & 0 \end{pmatrix} \end{pmatrix}. \tag{C.3}$$

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. Let us consider the example in Sec. 2: its corresponding tensor network structure for computing the maximal independent polynomial becomes



One can see that the average degree of a tensor is increased. 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 [11], which is the standard algorithm that we are aware of to compute the maximal independence polynomial.

We show the benchmark of computing the maximal independent set properties in Fig. 4, including a comparison to the Bron-Kerbosch algorithm from the package Graphs [3]. Since

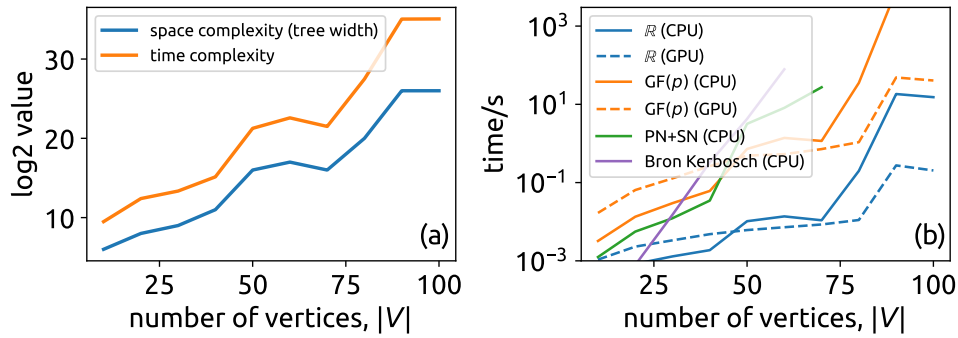


Figure 4: Benchmark results for computing different properties of maximal independent sets on a random three regular graph with different tensor element types. (a) treewidth versus the number of vertices for the benchmarked graphs. (b) The computing time for calculating the number of independent sets and enumerate all MISs.

the network for computing maximal independent set properties is different from the one for computing the independence polynomial, the tree width of this tensor network is significantly larger, hence only small graphs can be benchmarked. The time for the tensor netwrok approach and the Bron-Kerbosch approach to enumerate all maximal independent sets are comparable, while the tensor network does counting much more efficiently. Due to the memory limit, this Bron-Kerbosch algorithm stops working at size 70 and above.

C.2. Matching problem. A matching polynomial of a graph G is defined as

$$M(G,x) = \sum_{k=1}^{|V|/2} c_k x^k, \tag{C.4}$$

where k is the number of matches, and coefficients c_k are the corresponding counting. We define a tensor of rank $d(v) = |N(v)|$ on vertex v such that,

$$W_{\langle v,n_1\rangle,\langle v,n_2\rangle,\ldots,\langle v,n_{d(v)}\rangle} = \begin{cases} 1, & \sum_{i=1}^{d(v)} \langle v,n_i\rangle \leq 1, \\ 0, & \text{otherwise}, \end{cases} \tag{C.5}$$

and a tensor of rank 1 on the bond

$$B_{\langle v,w\rangle} = \begin{cases} 1, & \langle v,w\rangle = 0 \\ x, & \langle v,w\rangle = 1. \end{cases} \tag{C.6}$$

Here, we use bond index $\langle v,w\rangle \in \{0,1\}$ ($\langle v,w\rangle$ and $\langle w,v\rangle$ are equivalent) to label tensors, where 1 means two vertices of an edge are matched, 0 means otherwise.

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

$$W = \begin{pmatrix} r_v \\ g_v \\ b_v \end{pmatrix}, \tag{C.7}$$

and an edge tensor as

$$B = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}. \tag{C.8}$$

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 can get all possible colorings. Similarly, one can define the k-coloring problem on edges too.

C.4. Max cut problem. Max cut problem is also known as the boolean spinglass problem. Its tensor network representation does not contain vertex tensors. The edge tensor can be defined as

$$B(x_{\langle i,j\rangle}) = \begin{pmatrix} 1 & x_{\langle i,j\rangle} \\ x_{\langle i,j\rangle} & 1 \end{pmatrix}. \tag{C.9}$$

We can also define a polynomial about edges variables by setting $x_{\langle i,j\rangle} = x$, where its kth coefficient is two times the number of configurations of cut size k.

C.5. Set packing. Set packing is the hypergraph version of the maximum independent set problem, where each set corresponds to a vertex, each element corresponds to a hyperedge. To solve the set packing problem, we just remove the rank 2 restriction of an edge tensor

$$B_{v,w,\ldots,z} = \begin{cases} 1, & v + w + \ldots + z \leq 1, \\ 0, & \text{otherwise.} \end{cases} \tag{C.10}$$

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. (4.7). 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 integer overflow and round-off errors. One drawback of using finite field algebra is its matrix multiplication is less computational efficient compared with floating point matrix multiplication. Here, we give an alternative method based on discrete Fourier transform with controllable round off errors. 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

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

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 $\vec{a}_r = \text{FFT}^{-1}(\omega)\cdot\vec{y}$, where $(\vec{a}_r)_j = a_j r^j$. By choosing different r, one can obtain better precision in low independent set size region by choosing $r < 1$ or high independent set size region by choosing $r > 1$.

Appendix E. Integer sequences formed by the number of independent sets.

We also computed independence polynomial rigorously for these lattices in our Github repo[**JG: add link to the repo**].

Table 2: The number of independent sets for square grid graphs of size $L \times L$. This forms the integer sequence OEIS A006506.

L	square grid graphs
1	2
2	7
3	63
4	1 234
5	55 447
6	5 598 861
7	1 280 128 950
8	660 647 962 955
9	770 548 397 261 707
10	2 030 049 051 145 980 050
11	12 083 401 651 433 651 945 979
12	162 481 813 349 792 588 536 582 997
13	4 935 961 285 224 791 538 367 780 371 090
14	338 752 110 195 939 290 445 247 645 371 206 783
15	52 521 741 712 869 136 440 040 654 451 875 316 861 275
16	18 396 766 424 410 124 752 958 806 046 933 947 217 821 482 942
17	14 557 601 701 834 111 295 974 187 104 248 827 765 798 599 152 358 303
18	26 024 585 612 650 837 861 658 126 921 792 857 026 992 497 268 285 945 167 621
19	105 105 055 066 577 962 012 604 229 608 317 915 229 737 651 637 019 975 757 755 051 314
20	958 979 036 662 929 619 406 859 624 886 958 746 851 620 546 557 485 230 898 539 651 354 907 499
21	19 766 982 580 727 525 559 447 459 703 066 410 506 378 295 841 376 040 664 015 562 851 325 369 819 076 327
22	920 486 427 724 231 647 653 646 535 134 255 190 048 558 085 587 696 509 535 085 179 014 710 020 739 303 637 413 062
23	96 836 737 362 332 577 228 126 233 146 266 027 907 220 602 904 916 569 863 928 397 932 638 147 313 462 817 836 132 159 477 643
24	23 014 876 830 102 973 080 295 375 510 481 185 265 791 418 130 718 634 637 816 237 986 620 978 856 344 595 418 898 906 840 911 239 867 869
25	12 357 280 628 458 621 610 261 003 683 798 282 889 567 362 022 871 449 310 490 079 744 330 288 088 293 888 381 458 751 196 205 785 114 467 063 479 978
26	14 989 353 830 657 887 045 253 535 577 369 170 604 561 059 504 403 111 230 207 533 142 521 161 755 543 754 839 412 836 401 335 298 693 619 461 437 872 472 165 615
27	41 076 048 814 756 904 170 038 431 958 637 474 710 328 322 517 168 428 037 124 975 797 939 221 729 929 250 098 889 601 666 346 169 376 778 430 801 983 574 787 538 529 309 191
28	254 296 368 874 677 123 746 928 211 898 730 495 474 495 246 876 690 760 159 507 126 754 738 599 026 016 292 991 448 162 576 157 292 681 301 528 315 863 006 160 203 433 527 765 235 617 478
29	3 556 619 491 488 838 337 298 644 336 171 591 528 960 687 731 348 622 394 588 805 214 244 031 766 315 339 709 627 465 589 460 212 852 431 000 227 410 783 701 037 177 097 019 793 172 369 009 198 594 591
30	112 377 766 778 527 126 203 646 648 402 050 958 984 330 252 199 994 469 198 044 986 979 798 657 215 652 532 005 142 877 926 040 203 499 014 641 860 333 039 219 325 350 361 465 468 647 055 420 204 598 509 312 491 705

Table 3: The number of independent sets for King's graphs of size $L \times L$.

L	King's graphs
1	2
2	5
3	35
4	314
5	6 427
6	202 841
7	12 727 570
8	1 355 115 601
9	269 718 819 131
10	94 707 789 944 544
11	60 711 713 670 028 729
12	69 645 620 389 200 894 313
13	144 633 664 064 386 054 815 370
14	540 156 683 236 043 677 756 331 721
15	3 641 548 665 525 780 178 990 584 908 643
16	44 222 017 282 082 621 251 230 960 522 832 336
17	968 503 939 616 343 947 563 582 929 715 005 880 647
18	38 227 887 218 717 761 202 510 261 178 854 062 185 464 315
19	2 720 444 488 584 821 384 410 936 779 813 343 554 469 758 172 682
20	348 970 226 122 589 397 373 342 369 495 005 120 745 703 462 667 115 175
21	80 700 603 403 721 730 646 640 814 391 653 008 712 705 595 500 769 624 448 529
22	33 641 616 174 796 469 294 898 513 022 199 100 689 671 634 779 118 656 571 910 751 320
23	25 281 578 706 433 684 460 290 055 263 926 749 952 595 755 044 481 112 956 327 672 312 862 611
24	34 249 181 078 331 384 968 700 380 345 306 575 903 108 280 266 841 066 358 396 857 518 201 026 192 547
25	83 641 072 313 734 275 009 578 098 702 552 656 178 287 685 025 530 905 558 603 555 359 601 823 180 929 638 318
26	368 222 048 967 797 645 785 624 418 568 072 838 671 415 214 857 177 141 161 824 615 541 365 640 076 045 923 328 316 979
27	2 922 282 601 123 898 422 508 409 690 495 824 015 239 663 506 989 438 099 254 205 806 998 557 618 858 143 959 252 475 337 777 029
28	41 807 680 908 633 213 277 041 952 346 680 116 482 996 387 928 973 684 599 097 559 098 879 721 953 006 036 791 335 134 446 016 561 667 772