COMPUTING SOLUTION SPACE PROPERTIES BY GENERIC PROGRAMMING TENSOR NETWORKS *

XXX† AND YYY‡

Abstract. We introduce a tensor network algorithm to compute various solution space properties of a class of combinatorial optimization problems on graphs that can be rephrased as satisfiability problems quantified over local sets of vertices, including the independent set problem, the maximum cut problem, the vertex coloring problem, the maximal clique problem, the dominating set problem, and the satisfiability problem, among others.[**MC: I still think there is a more concise way to word the class of combinatorial optimization problems this works on...**][**JG: changed a bit, better?**] We look at the independent set problem as an example, and show how to compute the size of the maximum independent set, count the number of independent sets of a given size, and enumerate/sample the independent sets of a given size. By using generic programming techniques, the same simple-to-implement framework can be used to compute all of these properties. Our algorithm utilizes recent advances in tensor network contraction techniques to achieve high performance, including methods to quickly find a near-optimal contraction order and slicing. To demonstrate how our versatile tool helps to understand these hard problems, we apply it to a few examples, including computing the entropy constant for several hardcore gases on disordered lattices, studying the Overlap Gap Property on unit disk graphs and regular graphs, and visualizing the output of quantum optimization algorithms for the independent set problem. [**MC: I might not directly mention the classes of graphs we look at for Overlap Gap - it might be too much information. I'm not sure though, what do you think?**] [**JG: I agree "King's graphs at 0.8 filling and 3-regular graphs" is a bit too detailed, so I rephrased it as "unit disk graphs and regular graphs" that only conveys the most important information.**]

Key words. solution space property, tensor network, maximum independent set, independence polynomial, generic programming

AMS subject classifications. 05C31, 14N07

[**ML: a comment**] and [~~**ML: a resolved comment**~~]

1. Introduction.

In graph theory and combinatorial optimization, there is an important class of problems that can rephrased as satisfiability problems quantified over local sets that typically consists of a vertex and its neighborhood, including the independent set problem, the cutting problem, the dominating set problem, the set packing problem, the vertex coloring problem, the K-SAT problem, the maximal clique problem, and the vertex cover problem [45]. Many decision and counting properties of these problems are in general intractable at large sizes (exponential run time in problem size in the worst case), but it is nevertheless desirable to have high-performance algorithms because of their ubiquitous applications [12, 58].

The most studied aspect of the above problems is finding an optimal solution, which is typically NP-hard [33], i.e. it is unlikely to be solved in a polynomial time. In this paper, we focus on the less studied problem of computing *solution space properties* which can be much harder, e.g. counting independent sets of a general graph is #P-complete [57], but is crucial towards understanding a hard combinatorial optimization problem. Here, the *solution space property* refers to a class of quantities that not only include the maximum/minimum set size, but also include the number of sets at a give size, enumeration of all sets at a give size or direct sampling of such sets when they are not enumeratable. For the weighted version of the above problems, it also includes finding largest/smallest k sets and their sizes. In the past, people use the counting of configurations at different sizes to tell how likely the simulated annealing will be trapped in certain size in the strong defensive alliance problem [59], and use pair-

* **Funding:** ...

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

wise Hamming distance distribution of configurations at a given size to tell the presence or absence of the Overlap Gap Property [26, 25], a no go theorem about when the solution space geometry is formidably hard for any local search algorithm. In a recent experiment of using Rydberg atom arrays to optimize the independent set problem [18], the authors use the counting information and the configuration space connectivity to find instances hard for classical algorithms and design better quantum algorithms to solve these instances faster, and this experiment gives birth to the paper we are presenting.

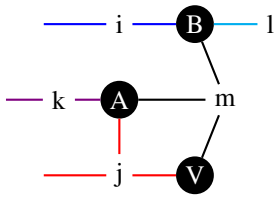
We give an algebraic interpretation of the computation of all these properties and use the unified framework of generic tensor-network to solve these seemingly unrelated properties all. Tensor network is a computational model widely used in condensed matter physics [46], quantum computing [44], big data [14] and mathematics [47], which is also known as the sum product network in probabilistic modeling [9] or einsum in linear algebra libraries such as numpy [31]. The tensor element types are typically restricted to standard number types such as real numbers and complex numbers. We extend this concept to generic tensor network by generalizing the tensor element type to any type being a commutative semiring. Given a good contraction order, the space required to contract a tensor network is two to the power of the treewidth of its line graph [44]. Recent progress in simulating quantum circuits with random tensor networks [29, 48, 36] enables us to find such a good contraction order for a tensor network with thousands of tensors in a reasonable time using heuristic (inexact) methods. Tensor network is very related to a well known computer programming method, the dynamic programming [16, 23], like the Viterbi algorithm for finding most probable configuration in the hidden Markov model has a nice matrix product state interpretation [30], and the tropical tensor network is very likely equivalent to dynamic programming in solving hard combinatorial optimization problems in certain way [41] and they have the same time complexity related to the treewidth of a graph. However, while being more restrictive regarding the problem set it can solve, the tensor network has a richer algebraic structure, which crucially enables the computation of solution space properties beyond finding a single solution, which is not possible with dynamic programming.

To avoid the discussion being unnecessarily diverged, we mainly focus on the independent set problem, and discuss other combinatorial optimization problems, including the cutting problem, the matching problem, the vertex coloring problem, the satisfiability problem, the dominating set problem, the set packing problem and the maximal clique problem, in Appendix C. The paper is organized as follows. We first introduce the basic concepts of tensor networks and generic programming in Sec. 2 and Sec. 3. Then we show how to reduce the independent set problem to a tensor network contraction problem in Sec. 4 and show how to engineer the element types in Sec. 6, Sec. 7 and Sec. 8 to count independent sets at a given size, enumerate/sample independent sets at a given size and find largest set sizes in weighted graphs. Lastly, we benchmark our algorithms in Sec. 9 and provide three examples in Sec. 10 to demonstrate the versatility of our tool.

2. Tensor networks. A tensor network [15, 46] is composed of a collection of tensors, a collection of labels associated with tensor dimensions to index tensor elements and an indicator of which tensor is the output, while the contraction of a tensor network is defined as a summation of products of tensor elements over the labels not appearing in the output tensor. e.g. the matrix multiplication is a special tensor network that can be represented as $C_{ik} = A_{ij}B_{jk}$, where A, B and C are tensors, ik, ij and ik are labels associated with them, and "=" is the indicator of C being the output, while its contraction is defined as $C_{ik} = \sum_j A_{ij}B_{jk}$. The graphical representation of a generalized tensor network is an open hypergraph, where an input tensor is mapped to a vertex and a label is mapped to a hyperedge that can connect an arbitrary number of vertices, while the labels appearing in the output tensor are open. As

a reminder, our notation slightly generalizes the standard tensor network used in physics by not restricting how many times a label can appear in tensors, they are equivalent in representation power thought some transformation, but this generalized tensor network might produce smaller contraction complexity, which will be illustrated in Appendix B.

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 tensor contractions. [**MC: On a second read of this section, I think that the clarity could be improved. My understanding is this is the flow: We would like to use generic programming, where we write code that works regardless of input type, and is optimized on all input types without sacrificing efficiency. In general, we can write code such that a function runs correctly regardless of the input data type. This is easy to do for dynamically typed languages like Python, which only verify and enforce the rules for the input data type at run time (called type-checking). However, when the input data type is only checked at run time (as opposed to when the code is compiled), type-specific optimization cannot be used. High efficiency is only achieved in statically typed languages, where modern compiling technology allows the function to be individually optimized for each input data type (then give C++/Julia examples).**][**JG: how about now?**] In previous works relating tensor networks and combinatoric problems [38, 8], the elements in the tensor networks are limited to standard number types such as floating point numbers and integers. In fact, the elements in a tensor does not have to be a regular number type, or even a field (since division is not used in tensor contraction). One can use the same tensor contraction program for different purposes with different tensor element types, and this idea of using the same program for different purposes is also called the generic programming in computer science:

Definition 3.1 (Generic programming [55]). *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.*

This definition of generic programming covers two major aspects: "work in the most general setting" and "with out loss of efficiency". By in the most general setting, we mean that a single program should work correctly for different input data types. For example, 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 $O(\log(n))$ 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 the efficiency. 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 optimize the memory layout of immutable elements with fixed sizes in an array to speed up the array indexing. In Julia, arrays with immutable elements with fixed sizes can even be compiled to graphics processing units (GPU) for faster computation [6].

This motivates us to think about what is the most general tensor element type 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 will be doable and the result will be independent of the contraction order. Here, a commutative semiring is a field without additive inverse and multiplicative inverse. Giving up these nice properties of a field means a lot to tensor computation: tensor network compression algorithms might be not applicable because matrix factorization is NP-hard for commutative semirings [54], and matrix multiplication faster than $O(n^3)$ does not exist for an algebra without additive inverse [37]. But here, we only use the commutative properties of an algebra for the purpose of tensor network contraction orders optimization. To define a commutative semiring with the addition operation $\oplus$ and the multiplication operation $\odot$ on a set S, the following relations must hold for any arbitrary three elements $a, b, c \in S$.

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

In the following sections, we show how to compute a number of properties of independent sets by designing tensor element types as specific commutative semirings without changing the tensor network contraction program [55]. The Venn diagram in Fig. 1 shows the algebras we will introduce in the main text and their relation, and Table 1 summarizes those properties that can be computed by various tensor element types.

4. Tensor network representation of independent sets. This section is about reducing the independent set problem to a tensor network contraction problem. For people with physics background, we suggest using an alternative approach described in Appendix A to understand this reduction from an energy model. Let $G = (V, E)$ be a weighted undirected graph with each vertex $v \in V$ being associated with a weight w_v. An independent set $I \subseteq V$ is a set of vertices that for any vertex pair $u, v \in I$, $(u, v) \notin E$, and we denote the maximum independent set (MIS) size $\alpha(G) \equiv \max_I \sum_{v \in I} w_v$. We map a vertex $v \in V$ to a label $s_v \in \{0, 1\}$ of dimension 2 in a tensor network, where we use 0 (1) to denote a vertex is absent (present) in the set. For

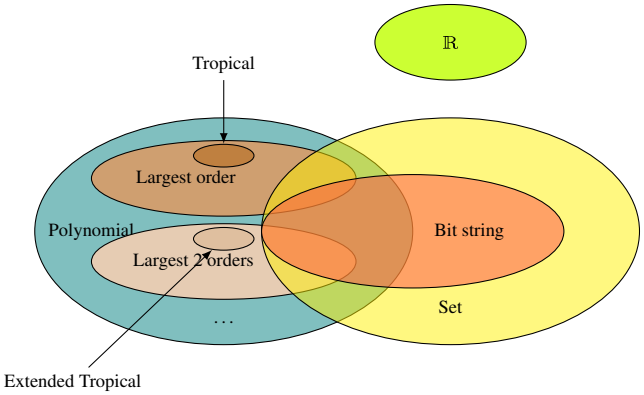


Figure 1: The the tensor network element types used in this work, while the purpose of these element types can be found in Table 1. The overlap between two algebra indicates that a new algebra can be created by combining those two types of algebra. "Largest order" and "Largest 2 orders" mean truncating the polynomial by only keeping its largest or largest two orders.

Element type	**Solution space property**
$\mathbb{R}$	Counting of all independent sets
Polynomial (Eq. (5.2: PN))	Independence polynomial
Tropical (Eq. (6.3: T))	Maximum independent set size
Extended tropical of order k (Eq. (8.1: Tk))	Largest k independent set sizes
Polynomial truncated to k-th order (Eq. (6.2: P1) and Eq. (6.5: P2))	k largest independent sizes and their degeneracy
Set (Eq. (7.1: SN))	Enumeration of independent sets
Sum-Product expression tree (Eq. (7.6: EXPR))	Sampling of independent sets
Polynomial truncated to largest order combined with bit string (Eq. (7.5: S1))	Maximum independent set size and one of such configurations
Polynomial truncated to k-th order combined with set (Eq. (7.3: P1+SN))	k largest independent set sizes and their enumeration

Table 1: Tensor element types and the independent set properties that can be computed using them.

each label s_v, we defined a parametrized rank-one vertex tensor $W(x_v, w_v)$ as

$$W(x_v, w_v) = \begin{pmatrix} 1 \\ x_v^{w_v} \end{pmatrix}. \tag{4.1}$$

where subscripts are used to annotate different vertices in configuration enumeration and sampling. One can index tensor elements by subscripting tensors, e.g. $W(x_v, w_v)_0 = 1_v$ is the first element associated with $s_v = 0$ and $W(x_v, w_v)_1 = x_v^{w_v}$ is the second element associated with $s_v = 1$. Similarly, on each edge (u, v), we define a matrix B indexed by s_u and s_v as

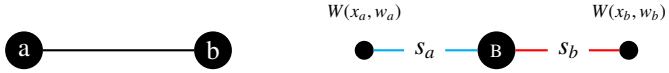
$$B = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}. \tag{4.2}$$

The corresponding tensor network contraction gives

$$P(G, \{x_1^{w_1}, x_2^{w_2}, \ldots, x_{|V|}^{w_{|V|}}\}) = \sum_{s_1, s_2, \ldots, s_{|V|}=0}^{1} \prod_{v \in V} W(x_v, w_v)_{s_v} \prod_{(w,v) \in E} B_{s_w s_v}, \tag{4.3}$$

where the summation runs over all vertex configurations $\{s_1, s_2, \ldots, s_{|V|}\}$ and accumulates the product of tensor elements to the output P. 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).

Example 2. In the following, we show a minimum example of mapping the independent problem of a 2 vertex complete graph K2 (left) to a tensor network (right).



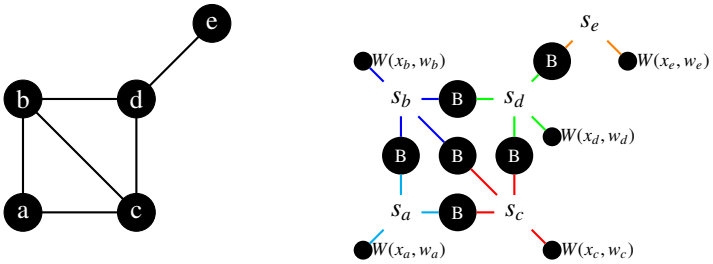
In the graphical representation of tensor network on the right panel, we use a circle to represent a tensor, a hyperedge in cyan color to represent the degree of freedom s_a, and a hyperedge in red color to represent a degree of freedom s_b. Tensors sharing a same degree of freedom are connected by that hyperedge. The contraction of this tensor network can be evaluated manually:

$$\begin{aligned} P(\mathrm{K2}, \{x_a^{w_a}, x_b^{w_b}\}) &= \begin{pmatrix} 1 & x_a^{w_a} \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ x_b^{w_b} \end{pmatrix} \\ &= 1 + x_a^{w_a} + x_b^{w_b} \end{aligned} \tag{4.4}$$

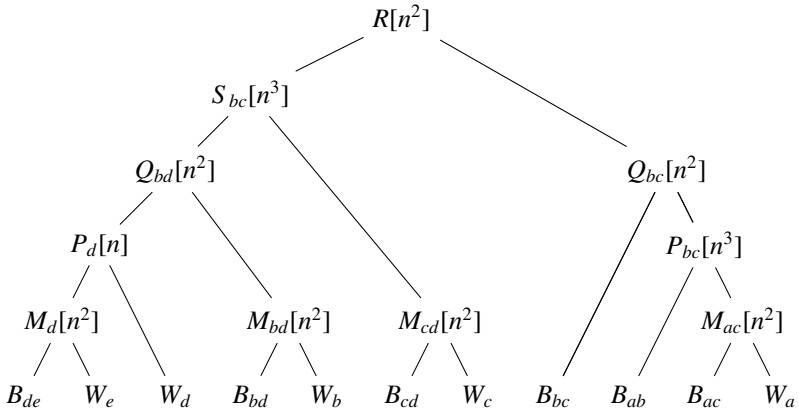
The resulting polynomial represents 3 different independent sets $\{\}$, $\{a\}$ and $\{b\}$, with weights being 0, w_a and w_b respectively.

In general, it is not computationally efficient to evaluate Eq. (4.3) by directly summing up the $2^{|V|}$ products. The standard approach to evaluate a random tensor network is: first find a good pair-wise tensor contraction order as a binary tree, and then contract two tensors a time by this order. An optimal tensor network contraction order corresponds to an optimal tree decomposition of its line graph and the largest intermediate tensor has a rank equal to the tree width of this optimal tree decomposition [44]. In practice, it is difficult to find an optimal contraction order for large tensor networks because finding the treewidth itself is NP-hard. However, it is easy to find a close to optimal contraction order in typically a few minutes with a heuristic contraction order finding algorithm [38, 36] utilizing the associativity and commutativity of tensor contraction. For large scale applications, it is also possible to slice over certain degrees of freedom to reduce the space complexity, i.e. loop and accumulate over certain degrees of freedom so that one can have a smaller tensor network inside the loop due to the removal of these degrees of freedom, which might have a smaller space complexity.

Example 3. In the following, we map a 5-vertex graph (left) to a tensor network (right) and show how the contraction order reduces the time and space complexities.



One can represent a possible pair-wise contraction of tensors as a binary tree structure:



The contraction process goes from bottom to top, where the root stores the contraction result that we want, the leaves are tensors in our tensor network while the rest nodes are all intermediate contraction results. Tensor subscripts are indices so that the number of subscripts indicates the space complexity to store this tensor. The contraction complexity to generate a tensor is annotated in the square brackets, where n is the dimension of degrees of freedom, which in our case is 2. One can easily check the largest tensor during computing has space complexity $O(n^2)$ and this is optimal, i.e. the treewidth of the original 5-vertex graph is 2, and the time complexity is $O(n^3)$, which is much smaller than that of direct enumeration $O(n^5)$.

5. Independence polynomial. Let $x_i = x$ and $w_i = 1$, the evaluation of Eq. (4.3) corresponds to the independence polynomial [32, 21]. which is a useful graph characteristic related to, for example, the partition functions [39, 60] and Euler characteristics of the independence complex [10, 40]. The independence polynomial is an important graph polynomial that contains the counting information of independent sets. It is defined as

$$I(G, x) = \sum_{k=0}^{\alpha(G)} a_k x^k, \tag{5.1}$$

where a_k is the number of independent sets of size k in graph $G = (V, E)$. The total number of independent sets is thus equal to $I(G, 1)$. Its connection to Eq. (4.3) 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. The summation counts the number of independent sets of size k. By assigning a real number to x, one can evaluate this independence polynomial for this specific value directly by contracting this tensor network. However, instead of evaluating this polynomial for a certain value, we are more interested in knowing the coefficients of this polynomial, because this quantity tells us the counting of

independent sets at each size. To achieve this, let us first elevate the tensor elements 0s and 1s in tensors $W(x, 1)$ 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. [**MC: I'd say that you're about to describe this semiring, otherwise it seems like you're going to use the general definition of the semiring rather than a specific type of semiring**][**JG: from 1 to $\mathbb{1}$ and 0 to $\mathbb{0}$ applies for a general semiring, I haven't change this part, let's discuss to make sure I understand your point.**] Let us create a polynomial type and represent a polynomial $a_0 + a_1 x + \ldots + a_k x^k$ as a coefficient vector $a = (a_0, a_1, \ldots, a_k) \in \mathbb{R}^k$, 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

$$
\begin{aligned}
a \oplus b &= (a_0 + b_0, a_1 + b_1, \ldots, a_{\max(k_a,k_b)} + b_{\max(k_a,k_b)}),\\
a \odot b &= (a_0 + b_0, a_1 b_0 + a_0 b_1, a_2 b_0 + a_1 b_1 + a_0 b_2, \ldots, a_{k_a} b_{k_b}),\\
\mathbb{0} &= (),\\
\mathbb{1} &= (1).
\end{aligned}
\tag{5.2: PN}
$$

Here, the multiplication operation can be evaluated in time $O(k \log(k))$, where $k = \max k_a, k_b$, using the convolution theorem [53]. 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

$$
W^{\mathrm{PN}} = \begin{pmatrix} \mathbb{1} \\ (0, 1) \end{pmatrix}, \qquad B^{\mathrm{PN}} = \begin{pmatrix} \mathbb{1} & \mathbb{1} \\ \mathbb{1} & \mathbb{0} \end{pmatrix}. \tag{5.3}
$$

By contracting the tensor network with the polynomial type, we have the exact representation of the independence polynomial. In practise, using the polynomial type suffers a space overhead proportional to $\alpha(G)$ because each polynomial requires a vector of such size to store the coefficients. One may argue that one can first evaluate this polynomial at different x being a real number, and then apply the Gaussian elimination to fit the coefficients of this polynomial. This seemingly more time and space efficient approach suffers from severe issues in practise. The data ranges of standard integer types are too small to cover many practical using cases, while the floating point numbers may have round off error that much larger than the value itself due to the fact that the number of sets at different sizes may vary tens or even hundreds of orders and this makes the small coefficients can not be retrieved correctly by fitting. For the purpose of evaluating these coefficients, we refer readers to Appendix E, where we provide an accurate and memory efficient method to find the polynomial by fitting it in a finite field. Alternatively, one can use the method in Appendix F to fit this polynomial more time efficiently with complex numbers with controllable round off errors. For the purpose of discussion in the main text, let us stick to this less efficient polynomial algebra.

6. Maximum independent sets and its counting.

6.1. Tropical algebra for finding the MIS size and counting MISs. The MIS size $\alpha(G)$, or the independence number is an important graph characteristics. The method we use to compute this quantity is based on the following observations. Let $x = \infty$, the independence polynomial in the previous section becomes

$$
I(G, \infty) = \lim_{x \to \infty} \sum_{k=0}^{\alpha(G)} a_k x^k = a_{\alpha(G)} \infty^{\alpha(G)}, \tag{6.1}
$$

where all terms except the largest order vanish. We can thus replace the polynomial type $a = (a_0, a_1, \ldots, a_k)$ with a new type that has two fields: the largest exponent k and its coefficient a_k. From this, we can define a new algebra as

$$
\begin{aligned}
a_x\infty^x \oplus a_y\infty^y &= \begin{cases} (a_x + a_y)\infty^{\max(x,y)}, & x = y \\ a_y\infty^{\max(x,y)}, & x < y \\ a_x\infty^{\max(x,y)}, & x > y \end{cases} , \\
a_x\infty^x \odot a_y\infty^y &= a_x a_y \infty^{x+y} \\
\mathbb{0} &= 0\infty^{-\infty} \\
\mathbb{1} &= 1\infty^0.
\end{aligned}
\tag{6.2: P1}
$$

Note here, in order to define the zero element, we have generalized the previous polynomial to the Laurent polynomial. To implement this algebra programmatically, we create a data type with two fields (x, a_x) to store the MIS size and its counting, and define the above operations and constants correspondingly. If one is only interested in finding the MIS size, one can drop the counting field. The algebra of the exponents becomes the max-plus tropical algebra [42, 45]:

$$
\begin{aligned}
x \oplus y &= \max(x, y) \\
x \odot y &= x + y \\
\mathbb{0} &= -\infty \\
\mathbb{1} &= 0.
\end{aligned}
\tag{6.3: T}
$$

Algebra Eq. (6.3: T) and Eq. (6.2: P1) are the same as those used in Liu et al. [41] to calculate and count spin glass ground states. For independent set calculations here, the vertex tensor and edge tensor becomes:

$$
W^{\mathrm{T}} = \begin{pmatrix} \mathbb{1} \\ \infty \end{pmatrix}, \qquad B^{\mathrm{T}} = \begin{pmatrix} \mathbb{1} & \mathbb{1} \\ \mathbb{1} & \mathbb{0} \end{pmatrix}.
\tag{6.4}
$$

6.2. Truncated polynomial algebra for counting independent sets of large size. Instead of counting just the MISs, one may also be interested in counting the independent sets with largest several sizes. 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. (5.2: PN) as:

$$
\begin{aligned}
a \oplus b &= (a_{\max(k_a,k_b)-1} + b_{\max(k_a,k_b)-1}, a_{\max(k_a,k_b)} + b_{\max(k_a,k_b)}), \\
a \odot b &= (a_{k_a-1}b_{k_b} + a_{k_a}b_{k_b-1}, a_{k_a}b_{k_b}), \\
\mathbb{0} &= (), \\
\mathbb{1} &= (1).
\end{aligned}
\tag{6.5: P2}
$$

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. Similarly, one can also truncate the polynomial and keep only its smallest several orders, which can be used for e.g. counting the maximal independent sets with smallest cardinality, where a maximal independent set is an independent set that can not be made larger by adding a new vertex into the it without violating the independence constraint. As will be shown below, this algebra can also be extended to enumerate those large-size independent sets.

7. Enumerating and sampling independent sets.

7.1. Set algebra for configuration enumeration. The configuration enumeration of independent sets includes, for example, the enumeration of all independent sets, the enumeration of all MISs and the enumeration of independent sets with maximum several sizes. Recall that in the definition of a vertex tensor in Eq. (4.1), variables can carry labels, so that in the output polynomials, one can read out all independent sets directly. The multiplication between labelled variables are commutative while the summation of labelled variables forms a set. Intuitively, one can use a bit string as the representation of a labelled variable and use bit-wise or $\vee^\circ$ as the multiplication operation, e.g. in a 5-vertex graph, x_2 can be represented as 01000, x_5 can be represented as 00001 and their multiplication x_2x_5 can be represented as $01000 \vee^\circ 00001 = 01001$. To enumerate all independent sets, we designed an algebra defined on sets of bit strings:

$$\begin{aligned} s \oplus t &= s \cup t \\ s \odot t &= \{\sigma \vee^\circ \tau \,|\, \sigma \in s, \tau \in t\} \\ \mathbb{0} &= \{\} \\ \mathbb{1} &= \{0^{\otimes |V|}\}. \end{aligned} \tag{7.1: SN}$$

where s and t are each a set of $|V|$-bit strings.

Example 4. For elements being bit strings 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 a basis bit string of size $|V|$ that has only one non-zero value at location i. The vertex and edge tensors are thus

$$W^{\rm SN}(\{\boldsymbol{e}_i\}) = \begin{pmatrix} \mathbb{1} \\ \{\boldsymbol{e}_i\} \end{pmatrix}, \qquad B^{\rm SN} = \begin{pmatrix} \mathbb{1} & \mathbb{1} \\ \mathbb{1} & \mathbb{0} \end{pmatrix}. \tag{7.2}$$

This set algebra can serve as the coefficients in Eq. (5.2: PN) to enumerate independent sets of all different sizes, Eq. (6.2: P1) to enumerate all MISs, or Eq. (6.5: P2) to enumerate all independent sets of size $\alpha(G)$ and $\alpha(G) - 1$. As long as the coefficients in a truncated polynomial forms a commutative semiring, the polynomial itself is still a commutative semiring. For example, to enumerate only the MISs, with the tropical algebra, we can define $s_k \infty^k$, where the coefficients follow the algebra in Eq. (7.1: SN) and the orders (exponents) follow the max-plus tropical algebra. The combined operations become:

$$\begin{aligned} s_x \infty^x \oplus s_y \infty^y &= \begin{cases} (s_x \cup s_y)\infty^{\max(x,y)}, & x = y \\ s_y \infty^{\max(x,y)}, & x < y \\ s_x \infty^{\max(x,y)}, & x > y \end{cases}, \\ s_x \infty^x \odot s_y \infty^y &= \{\sigma \vee^\circ \tau | \sigma \in s_x, \tau \in s_y\} \infty^{x+y}, \\ \mathbb{0} &= \{\} \infty^{-\infty}, \\ \mathbb{1} &= \{0^{\otimes |V|}\} \infty^0. \end{aligned} \tag{7.3: P1+SN}$$

Clearly, the vertex tensor and edge tensor become

$$W^{\mathrm{P1+SN}}(\{\boldsymbol{e}_i\infty^1\}) = \begin{pmatrix} \mathbb{1} \\ \{\boldsymbol{e}_i\}\infty^1 \end{pmatrix}, \qquad B^{\mathrm{P1+SN}} = \begin{pmatrix} \mathbb{1} & \mathbb{1} \\ \mathbb{1} & \mathbb{0} \end{pmatrix}. \tag{7.4}$$

The contraction of the corresponding tensor network yields an enumeration of all MIS configurations. Direct implementing this algebra for configuration MISs might incur significant space overheads for keeping too many intermediate states not contributing to the final maximum independent sets, one can avoid this issue by using the bounding technique in Appendix D. One may also be interested in a more studied enumeration of maximal independent sets [11, 19, 35]. It will be discussed in Appendix C.1 since it requires using a different tensor network structure for the maximal independent set problem.

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. (7.1: SN),

$$\begin{aligned} \sigma \oplus \tau &= \mathtt{select}(\sigma, \tau), \\ \sigma \odot \tau &= (\sigma \vee^{\circ} \tau), \\ \mathbb{0} &= 1^{\otimes |V|}, \\ \mathbb{1} &= 0^{\otimes |V|}, \end{aligned} \tag{7.5: S1}$$

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. In practise, it is completely fine to pick a random one (not commutative anymore) to generate a random MIS.

7.2. Sampling extremely large configuration space. When the graph size goes large, a set of configurations might be impossible to fit into any type of storage. To get something meaningful out of configuration space, we use a binary sum-product expression tree as a compact representation of a set of configurations, i.e. instead of directly computing a set using the algebra in Eq. (7.1: SN), we store the process of computing it. A node in this tree is a quadruple $(type, data, left, right)$, where $type$ is one of `LEAF`, `ZERO`, `SUM` and `PROD`, $data$ is a bit string as the content in a `LEAF` node, $left$ and $right$ are left and right operands of a `SUM` or `PROD` node.

$$\begin{aligned} s \oplus t &= (\mathtt{SUM}, /, s, t) \\ s \odot t &= (\mathtt{PROD}, /, s, t) \\ \mathbb{0} &= (\mathtt{ZERO}, /, /, /) \\ \mathbb{1} &= (\mathtt{LEAF}, 0^{\otimes |V|}, /, /). \end{aligned} \tag{7.6: EXPR}$$

The vertex tensor and edge tensor become

$$W^{\mathrm{EXPR}}((\mathtt{LEAF}, \boldsymbol{e}_i, /, /)) = \begin{pmatrix} \mathbb{1} \\ (\mathtt{LEAF}, \boldsymbol{e}_i, /, /) \end{pmatrix}, \qquad B^{\mathrm{EXPR}} = \begin{pmatrix} \mathbb{1} & \mathbb{1} \\ \mathbb{1} & \mathbb{0} \end{pmatrix}. \tag{7.7}$$

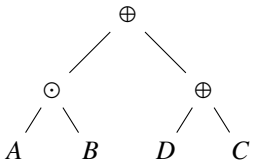
This algebra is a commutative semiring because we define the equivalence of two sum-product expression trees by comparing their expanded (using Eq. (7.1: SN)) forms, rather than their storages. Except using the sum-product expression tree directly as tensor elements, one can also let it be the coefficients of a (truncated) polynomial so that one can have such trees

for independent sets with largest several sizes. Although one probably can not enumerate all configurations represented by a sum-product expression tree due to its extremely large size, he can draw unbiased samples from it efficiently. One starts from the root node, and descends this tree recursively to its left or right sibling with probability decided by the size of each sub-tree while doing the computation specified by the first field. The recursion stops at a `LEAF` node having size 1 or a `ZERO` node having size 0. In an sum-product expression tree, the number of configurations of a sub-trees can be determined easily as we will show in the following example.

Example 5. Let us consider the following sum-product expression tree

$$(\texttt{SUM}, /, (\texttt{PROD}, /, A, B), (\texttt{SUM}, /, C, D)),$$

where siblings A, B, C and D can be any four types of nodes. It can be represented diagrammatically as the following.



The left and right sibling of root node have size $|A| \times |B|$ and $|C| + |D|$ respectively. Since the root node has type `SUM`, its size can be computed as $|A| \times |B| + |C| + |D|$. One can compute the sizes of A, B, C and D recursively until the program meets either a the `LEAF` node or a `ZERO` node, which has a known size of 1 or 0 respectively.

8. Weighted graphs. All the previous discussion for unweighted graphs still holds for integer weighted graphs, while for general weighted graphs, the independence polynomial is not well defined anymore. For general weighted graphs, we are more interested to know the k most weighted sets and their sizes. Hence we define the extended tropical numbers as a natural generalization of tropical numbers:

(8.1: Tk)
$$\begin{aligned} s \oplus t &= \texttt{largest}(s \cup t, k) \\ s \odot t &= \texttt{largest}(\{a + b | a \in s, b \in t\}, k) \\ \mathbb{0} &= -\infty^{\otimes k} \\ \mathbb{1} &= -\infty^{\otimes k-1} \otimes 0 \end{aligned}$$

where $\texttt{largest}(s, k)$ means truncating the set by only keeping k largest values in the set s. The computation of $s \odot t$ is a maximum sum combination problem which can be done in time $\sim k \log(k)$. One can find an efficient algorithm in Appendix H. The vertex tensor and edge tensor become

(8.2)
$$W^{\mathrm{T}k}(-\infty^{\otimes k-1} \otimes 1, w_v) = \begin{pmatrix} \mathbb{1} \\ -\infty^{\otimes k-1} \otimes w_v \end{pmatrix}, \qquad B^{\mathrm{T}k} = \begin{pmatrix} \mathbb{1} & \mathbb{1} \\ \mathbb{1} & \mathbb{0} \end{pmatrix},$$

where we have used the relation $(-\infty^{\otimes k-1} \otimes 1)^{w_v} = -\infty^{\otimes k-1} \otimes w_v$. Similar to the tropical numbers, we can combine the extended tropical algebra with a bit string sampler (Eq. (7.5: S1)) for finding the configuration of each size. Since the $\odot$ operation of configuration sampler is not used, the resulting configurations are deterministic and complete.

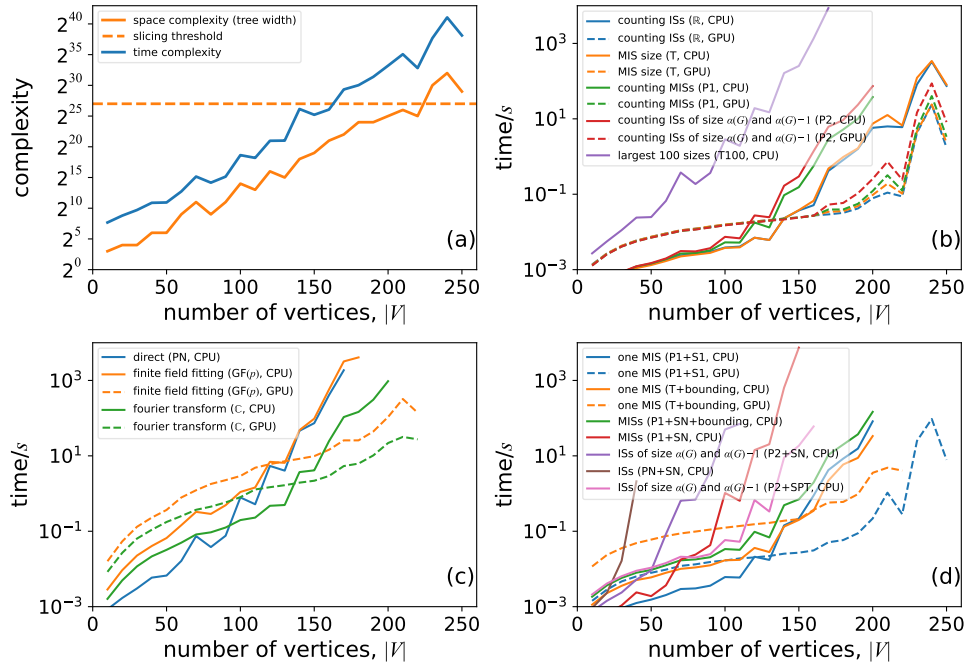


Figure 2: Benchmark results for computing different properties of independent sets of a random three regular graph with different tensor element types. The time in these plots only includes tensor network contraction, without taking into account the contraction order finding and just-in-time compilation time. Legends are properties, algebra and devices that we used in the computation; one can find the corresponding computed solution space property in Table 1. (a) time and space complexity versus the number of vertices for the benchmarked graphs. (b) The computing time for calculating the MIS size and for counting of the number of all independent sets (ISs), the number of MISs, the number of independent sets having size $\alpha(G)$ and $\alpha(G)-1$, and finding 100 largest sets sizes. (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 $\alpha(G)-1$, and all independent set configurations having size $\alpha(G)$ and $\alpha(G)-1$ in the sum-product expression tree format, with or without bounding the enumeration space.

9. Performance benchmarks. We run a single thread benchmark on central processing units (CPU) Intel(R) Xeon(R) CPU E5-2686 v4 @ 2.30GHz, and its CUDA version on a GPU Tesla V100-SXM2 16G. The results are summarized in Figure 2. The graphs that we use in benchmarks are random three regular graphs, a typical type of sparse graphs that have a small treewidth that asymptotically smaller than $|V|/6$ [22]. In this benchmark, we do not include traditional algorithms for finding the MIS size such as the branching algorithms [56, 52] or dynamic programming based algorithms [16, 23], because to the best of our knowledge, most of the properties mentioned in this paper are not computable within these frameworks. The main goal of this section is to show the relative timing for computing different properties.

Figure 2(a) shows the time and space complexity of tensor network contraction for different graph sizes, where the space complexity is the same as the treewidth of the problem

graph. Here, we assume our contraction order finding program has been converged to the optimal treewidth. The contraction order is obtained using the local search algorithm in Ref. [36]. In practice, slicing technique [36] is used for graphs with treewidth greater than 27 to fit the computation into a 16GB memory. One can see that all the computing times in figure 2(b), (c) and (d) have a strong correlation with the treewidth, while in panel (d), the timing of certain computing tasks also strongly correlates with other factors like the configuration space size. Among these benchmarks, computational tasks with data types tropical number of on CPU and real and complex numbers on both CPU and GPU can utilize fast basic linear algebra subprograms (BLAS) functions to evaluate tensor contractions, hence are much faster comparing to non-BLAS element types in the same category. GPU computes much faster than CPU in all cases when the problem scale is large enough such that the actual computing time is comparable or larger than the launching overhead of CUDA kernels. Most algebras can be computed on a GPU, except those requiring dynamic sized structures, i.e. types with algebra defined in Eq. (5.2: PN), Eq. (7.1: SN), Eq. (8.1: Tk) and Eq. (7.6: EXPR). In figure 2(c), one can see the Fourier transformation based method is the fastest in computing the independence polynomial, but it may suffer from round-off errors (Appendix F). The finite field (GF(p)) approach is the only method that does not have round-off errors and can be run on a GPU. In figure 2(d), one can see the technique to bound the enumeration space improves the performance for more than one order of magnitude in enumerating the MISs. Bounding can also reduce the memory usage significantly, without which the largest computable graph size is only ~ 150 on a device with 32GB main memory.

10. Example case studies. In this section, we give a few examples where the different properties of independence sets are used.

10.1. Number of independent sets and entropy constant for some hardcore lattice gases. We compute the counting of all independent sets for graphs shown in Figure 3, where vertices are all placed on square lattices with lattice dimensions $L \times L$. The types of graphs include: the square lattice graphs (Figure 3(a)); the square lattice graphs with a filling factor $p = 0.8$, which means $\lfloor pL^2 \rceil$ square lattices are occupied with vertices (Figure 3(b)); the King's graphs (Figure 3(c)); the King's graphs with a filling factor $p = 0.8$ (Figure 3(d)). The number of independent sets for square lattice 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. We computed the integer sequence for $L = 38$ and $L = 39$, which, to the best of our knowledge, is not known before. In the computation, we used the finite-field algebra for contracting arbitrarily high precision integer tensor networks.

A theoretically interesting number that can be computed using the number of independent sets is the entropy constant for the hardcore lattice gases on these graphs, which can tell us thermal dynamic properties of these lattices at the high temperature limit. For the square lattice graphs, it is called the *hard square entropy constant* (OEIS A085850), which is defined as $\lim_{L\to\infty} F(L, L)^{1/L^2}$, where $F(L, L)$ is the number of independent sets of a given lattice dimensions $L \times L$. This quantity arises in statistical mechanics of hard-square lattice gases [5, 49] and is used to understand phase transitions for these systems. This entropy constant is not known to have an exact representation, but it is accurately known in many digits. Similarly, we can define entropy constants for other lattice gases. In Fig. 4, we look at how $F(L, L)^{1/\lfloor pL^2 \rceil}$ scales as a function of the grid size L for all types of graphs shown in Figure 3. Our results match the known results for the non-disordered square lattice and King's graphs. For disordered square lattice and King's graphs with a filling factor $p = 0.8$, we randomly sample 1000 graph instances. To our knowledge, the entropy constants for these disordered graphs have not been studied before. Interestingly, the variations due to different random instances are negligible for this entropy quantity.

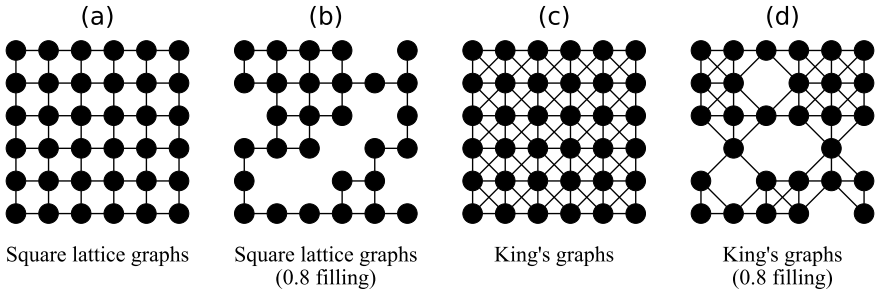


Figure 3: The types of graphs used in the case study in section 10.1. The lattice dimensions are $L \times L$. (a) square lattice graphs. (b) square lattice graphs with a filling factor $p = 0.8$. (c) King's graphs. (d) King's graphs with a filling factor $p = 0.8$.

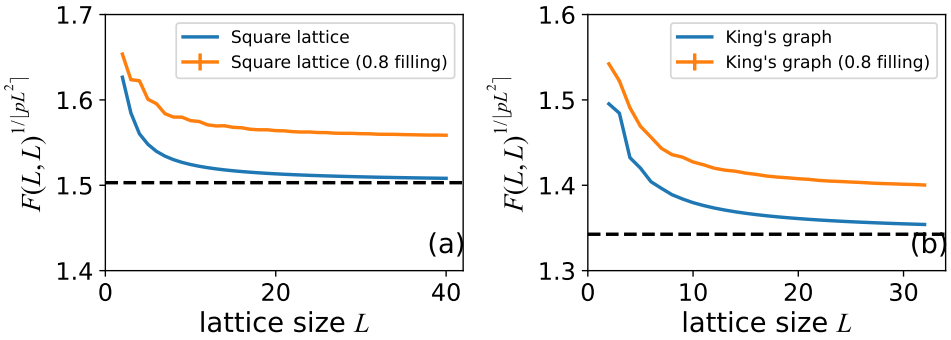
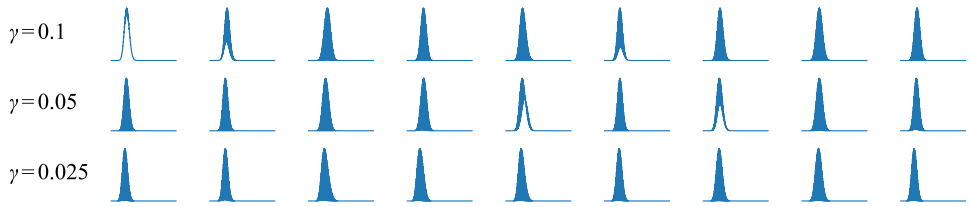
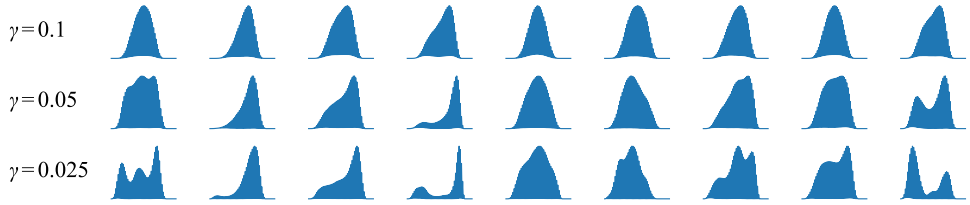


Figure 4: Mean entropy for lattice gases on graphs defined in Fig. 3. We sampled 1000 instances for $p = 0.8$ lattices and the error bar is too small to be visible. The horizontal black dashed lines are for $\lim_{L\to\infty} F(L,L)^{1/L^2}$ for the corresponding non-disordered square lattice and King's graphs.

10.2. The Overlap Gap Property. With the tool to enumerate configurations, one can understand the structure of the independent set configuration space, such as the optimization landscape for finding the MISs. One of the known barriers for finding the MIS is the so-called Overlap Gap Property [26, 25]. Intuitively, if the Overlap Gap Property is present, it means every two large independent sets either have a significant intersection or very small intersection; it implies that large independent sets are clustered together. [**MC: Do you want me to write a more formal definition as well or do you think it's not necessary?**][**JG: I suppose you want to justify the origin of "ratio", then it would great if you can explain it more clearly. However, explaining the concept overlap gap property well seems to be impossible in a this paper (you need at least some plots).**] This clustering property has been used to rigorously prove upper bounds on the performance of local algorithms [26, 25]. It was shown that for finding MISs of d-regular graphs with d and graph sizes both being large, there exists Overlap Gap Property [51, 24]. However, it is not known whether for small d, e.g. 3- regular graphs, this statement is still true. Here, we direct sample from the computed configurations of large independent sets to investigate the presence or absence of the Overlap



(a) Nine King's graphs of size 20 ×20, 0.8 filling.



(b) Nine random three regular graphs of size 110.

Figure 5: The plot of pairwise Hamming distances distribution for configurations sampled from independent sets with sizes $\geq \lceil \gamma \times \alpha(G) \rceil$. In each plot, the x-axis is the Hamming distance normalized by the total number of vertices and the y-axis is the probability. The "filled" region is caused by the fast oscillation of configuration populations of even and odd Hamming distances.

Gap Property for some given graphs. Because the number of configurations is too large to fit into storage, we use the compact sum-product expression tree representation to sample from independent sets of a given size. We compare nine randomly generated instances from two categories of graphs: King's graphs at $\rho = 0.8$ filling and 3-regular graphs.Our method can sample from independent sets of King's graph up to 20×20 (320 vertices) and the size of 3-regular graphs are 110. In particular, in Fig. 5, we sample two sets of 10^4 configurations from the ISs at sizes $\geq \lceil \gamma \times \alpha(G) \rceil$ for each instance and show the pairwise Hamming distance distribution for the enumerated configurations. In Fig. 5(a), we observed a clear single peak structure at a fixed distance normalized by the MIS size for the King's graphs. Which is a clear evidence of the absence of the Overlap Gap Property in 0.8 filling King's graph. Given the fact that the MIS problem on an arbitrary graph can be mapped to a King's graph at a certain filling [18], this result is highly nontrivial. While in Fig. 5(b), we observed the multiple peak structure in 3-regular graphs as the independent set sizes become large. This indicates the presence of the Overlap Gap Property [**JG: @Maddie, I am not sure if we can say "indicates the presence of the Overlap Gap Property" here because it is so different from the official definition of OGP, I want to change it to "disconnected clusters exist in the configuration space for large independent sets, however, it requires further study to show whether 3-regular graphs have the overlap gap property or not.", which one do you think is more correct?**] and disconnected clusters exist in the configuration space for large independent sets. We expect our numerical tool can be used to understand this phenomenon better and to further investigate the graph properties and the geometry of the configuration spaces for a variety of graph instances.

10.3. Visualizing experimental quantum algorithm outputs for solving Maximum Independent Set. [**MC: There are some other ideas I have to add to this...**] The ability

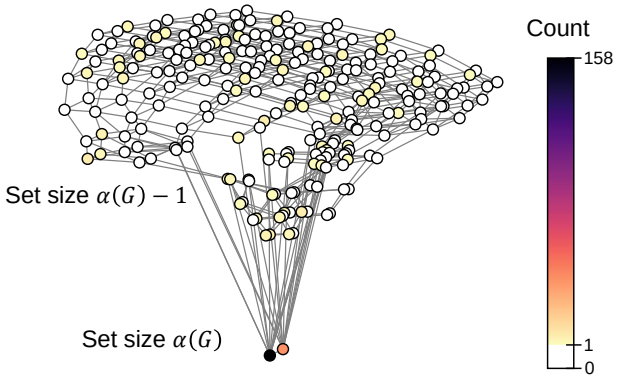


Figure 6: A visualization of experimental outputs of a quantum algorithm for solving the maximum independent set problem. Each vertex represents an independent set, and each edge represents a pair of independent sets that differ by a swap operation or a vertex addition.

to enumerate configurations can also be used to understand the distribution of outputs from algorithms for solving NP-hard combinatorial optimization problems. This can give detailed insights to performance of the algorithm, such as if the algorithm is likely to return independent sets from certain clusters of local or global minima in the solution space. As an example, we show in Figure 6 how our algorithm can be used to visualize experimental outputs of a quantum algorithm for solving the Maximum Independent Set problem using a programmable Rydberg atom array quantum computer [18]. The structure of the solution space for a King's graph with 39 vertices and filling factor $\rho = 0.8$ can be visualized on a graph where each vertex represents a large independent set, and each edge represents a pair of independent sets that differ by a swap operation or a vertex addition to the set. The algorithm either finds the MIS with size $\alpha(G)$ with high probability, or returns a suboptimal independent set, usually with size $\alpha(G) - 1$. The algorithm does not appear to return solely local minima with large Hamming distance from the MISs as suggested by [**?**][**JG: Fix ref.**], but samples from local minima with a wide range of Hamming distances from the MISs. Consistent with Sec. 10.2, there are no disconnected clusters in the solution space for this King's graph.

11. Discussion and conclusion. In this paper, we introduce a method to use generic programming and tensor networks to compute a number of different solution space properties of certain classes of NP-hard combinatorial optimization problems. For each solution space property, we design a commutative semiring algebra for the tensor element type. By using the same tensor network but replacing the tensor network elements with elements of different algebras, we can compute different solution space properties. This technique can further be generalized to computing solution space properties beyond the examples given in this paper. The different data types introduced in the main text to compute these properties are summarized in the diagram in Fig. 1. Other than the case with real numbers, we have polynomials and truncated polynomials. We combine them with the bit string algebra, the set algebra, and sum-product expression tree algebra for finding, enumerating, and sampling of independent sets at a given size. Three examples are shown to demonstrate how our method can be used to understand the solution space better. In the first example, we show how our method can help computing the entropy constant for some

hardcore lattice gases on 2D square lattices. We compute the 2D generalization of Fibonacci integer sequence up to size 39, which is larger than the previous record 37. In the second example, we analyze the pairwise Hamming distances from configurations directly sampled from large independent sets of King's graphs at 0.8 filling and three regular graphs. We show the strongest numeric evidence so far of the absence of Overlap Gap Property in King's graph and the existence of Overlap Gap properties in three regular graphs. In the last example, we show how our algorithm helps people to understand the performance of a quantum variational algorithms on Rydberg atom arrays to find maximum independent sets. This is a research frontier that understanding which can help people design better quantum algorithms to beat its classical counterpart. These algebras are of generic use: they can be utilized to compute properties of maximal independent sets and a variety of other combinatorial problems such as the matching problem, the k-coloring problem, the max-cut problem, and the set packing problem, as detailed in Appendix C. By adapting to the style of generic programming, we can implement all the algorithms without much effort. We show some of the Julia language implementations in Appendix I. A complete implementation can be found in our Github repository [1]. We expect our tool can be used to understand and study many interesting applications of independent sets and beyond. We also hope this approach of generic tensor networks can inspire future works on tensor network computations.

Acknowledgments. We would like to thank Pan Zhang for sharing his python code for optimizing contraction orders of a tensor network. We acknowledge Sepehr Ebadi and Leo Zhou for coming up with many interesting questions about independent sets and their questions strongly motivated the development of this project. We thank Benjamin Schiffer for providing helpful feedback on the manuscript. We thank Chris Elord for helping us write the fastest matrix multiplication library for tropical numbers, TropicalGEMM.jl. We would also like to thank a number of open-source software developers, including Roger Luo, Time Besard, Edward Scheinerman and Katharine Hyatt for actively maintaining their packages and resolving related issues voluntarily. [**JG: funding information**]

REFERENCES

[1] https://github.com/Happy-Diode/GraphTensorNetworks.jl.
[2] https://github.com/JuliaGraphs/Graphs.jl.
[3] https://github.com/TensorBFS/TropicalGEMM.jl.
[4] S. Alikhani and Y. hock Peng, *Introduction to domination polynomial of a graph*, 2009, https://arxiv.org/abs/0905.2251.
[5] R. J. Baxter, I. G. Enting, and S. K. Tsang, *Hard-square lattice gas*, Journal of Statistical Physics, 22 (1980), pp. 465–489, https://doi.org/10.1007/BF01012867, https://doi.org/10.1007/BF01012867.
[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] P. Butera and M. Pernici, *Sums of permanental minors using grassmann algebra*, 2014, https://arxiv.org/abs/1406.5337.
[14] A. Cichocki, *Era of big data processing: A new approach via tensor networks and tensor decompositions*, arXiv preprint arXiv:1403.2048, (2014).

[15] 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.

[16] B. Courcelle, *The monadic second-order logic of graphs. i. recognizable sets of finite graphs*, Information and computation, 85 (1990), pp. 12–75.

[17] J. C. Dyre, *Simple liquids' quasiuniversality and the hard-sphere paradigm*, Journal of Physics: Condensed Matter, 28 (2016), p. 323001.

[18] S. Ebadi, A. Keesling, M. Cain, T. T. Wang, H. Levine, D. Bluvstein, G. Semeghini, A. Omran, J. Liu, R. Samajdar, X.-Z. Luo, B. Nash, X. Gao, B. Barak, E. Farhi, S. Sachdev, N. Gemelke, L. Zhou, S. Choi, H. Pichler, S. Wang, M. Greiner, V. Vuletic, and M. D. Lukin, *Quantum optimization of maximum independent set using rydberg atom arrays*, 2022, https://arxiv.org/abs/2202.09372.

[19] 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.

[20] 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.

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

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

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

[24] D. Gamarnik, *The overlap gap property: A topological barrier to optimizing over random structures*, Proceedings of the National Academy of Sciences, 118 (2021).

[25] 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.

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

[27] 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.

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

[29] 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.

[30] Z.-Y. Han, J. Wang, H. Fan, L. Wang, and P. Zhang, *Unsupervised generative modeling using matrix product states*, Physical Review X, 8 (2018), p. 031012.

[31] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. Fernández del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, *Array programming with NumPy*, Nature, 585 (2020), p. 357–362, https://doi.org/10.1038/s41586-020-2649-2.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] L. R. Kerr, *The effect of algebraic structure on the computational complexity of matrix multiplication*, tech. report, Cornell University, 1970.

[38] 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.

[39] 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.

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

[41] 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.
[42] 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.
[43] 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.
[44] 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.
[45] C. Moore and S. Mertens, *The nature of computation*, OUP Oxford, 2011.
[46] R. Orús, *A practical introduction to tensor networks: Matrix product states and projected entangled pair states*, Annals of Physics, 349 (2014), pp. 117–158.
[47] I. V. Oseledets, *Tensor-train decomposition*, SIAM Journal on Scientific Computing, 33 (2011), pp. 2295–2317.
[48] F. Pan and P. Zhang, *Simulating the sycamore quantum supremacy circuits*, 2021, https://arxiv.org/abs/2103.03074.
[49] P. A. Pearce and K. A. Seaton, *A classical theory of hard squares*, Journal of Statistical Physics, 53 (1988), pp. 1061–1072, https://doi.org/10.1007/BF01023857, https://doi.org/10.1007/BF01023857.
[50] 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).
[51] M. Rahman and B. Virág, *Local algorithms for independent sets are half-optimal*, The Annals of Probability, 45 (2017), https://doi.org/10.1214/16-aop1094, http://dx.doi.org/10.1214/16-AOP1094.
[52] J. M. Robson, *Algorithms for maximum independent sets*, Journal of Algorithms, 7 (1986), pp. 425–440.
[53] A. Schönhage and V. Strassen, *Schnelle multiplikation grosser zahlen*, Computing, 7 (1971), pp. 281–292.
[54] Y. Shitov, *The complexity of tropical matrix factorization*, Advances in Mathematics, 254 (2014), pp. 138–156.
[55] A. A. Stepanov and D. E. Rose, *From mathematics to generic programming*, Pearson Education, 2014.
[56] R. E. Tarjan and A. E. Trojanowski, *Finding a maximum independent set*, SIAM Journal on Computing, 6 (1977), pp. 537–546.
[57] S. P. Vadhan, *The complexity of counting in sparse, regular, and planar graphs*, SIAM Journal on Computing, 31 (2001), pp. 398–427.
[58] 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.
[59] Y.-Z. Xu, C. H. Yeung, H.-J. Zhou, and D. Saad, *Entropy inflection and invisible low-energy states: Defensive alliance example*, Physical Review Letters, 121 (2018), https://doi.org/10.1103/physrevlett.121.210602, http://dx.doi.org/10.1103/PhysRevLett.121.210602.
[60] 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. An alternative way to construct the tensor network.

Let us characterize the independent set problem on graph $G = (V, E)$ as an energy model

$$\mathcal{E}(G, s) = -\sum_{i\in V(G)} w_i s_i + \infty \sum_{\langle i,j\rangle\in E(G)} s_i s_j \tag{A.1}$$

where s_i is a spin on vertex $i \in V$ and w_i is an onsite energy term associated with it. The first term is the energy that corresponds to the inverse of the independent set size and the second term describes the independence contraint. In physical systems, this contraint term usually corresponds to the Rydberg blockade [50, 18] in cold atom arrays or the repulsive force in hard core lattice model [17, 20]. Its partition function is defined as

$$\begin{aligned} Z(G,\beta) &= \sum_{s} e^{-\beta\mathcal{E}(G,s)} = \sum_{s\in\mathcal{I}(G)} e^{\beta\sum w_i s_i} \\ &= \sum_{k=0}^{\alpha(G)} a(k) e^{\beta k} \qquad (k = \sum w_i s_i) \end{aligned} \tag{A.2}$$

where $\mathcal{I}(G)$ is the set of independent sets of graph G. $\alpha(G)$ is the absolute value of the minimum energy (maximum independent set size). $a(k)$ is the number of spin configurations

with energy $-k$ (independent sets of size k). The partition function can be evaluated as a tensor network contraction, where the tensor network is constructed by placing a vertex tensor on each spin i

$$W(\beta, w_i) = \begin{pmatrix} 1 \\ e^{\beta w_i} \end{pmatrix}, \tag{A.3}$$

and an edge tensor on each bond

$$B = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \tag{A.4}$$

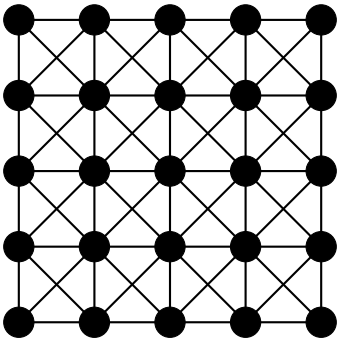
where the 0 in the bond tensor coming from $e^{-\beta\infty}$ in the second term of Eq. (A.1). By letting $x = e^{\beta}$, we get the tensor network for computing the independence polynomial as described by Eq. (4.1) and Eq. (4.2), and by letting $w_i = 1$, the second line of Eq. (A.2) is the independence polynomial.

Appendix B. An example of contraction complexity increase due to using the standard tensor network notation.

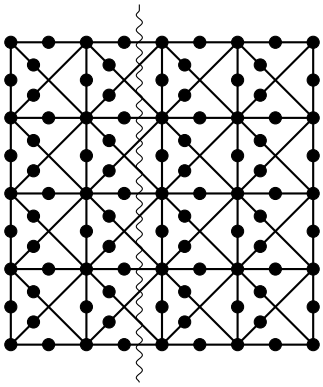
In the standard Einstein's notation for tensor networks in physics, each index appears precisely twice, either both in input tensors (which will be summed over) or one in a input tensor and another in the output tensor. Hence a tensor network can be represented as a open simple graph, where a tensor is mapped to a vertex, a label shared by two input tensors is mapped to a normal edge and a label appears in the output tensor is mapped to an open edge. A standard tensor network notation is equivalent to the generalized tensor network by introducing δ tensors, where a δ tensor of rank d is defined as

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

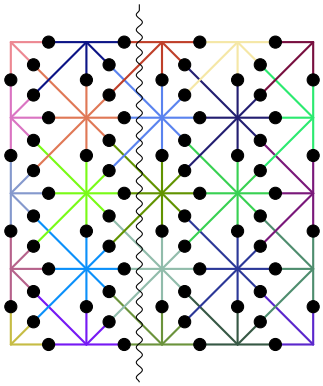
In the following, we are going to show introducing δ tensors might increase the contraction complexity of a tensor network. Let us consider the following King's graph.



By mapping the independent set problem to a standard tensor network with δ tensors, we have the following graphical representation.



In this diagram, the circle on each vertex in the original graph is a δ tensor of rank 8 or less. If we contract this tensor network in a naive column-wise order, the maximum intermediate tensor has rank $\sim 3L$, requiring a storage of size $\approx 2^{3L}$. If we relax the restriction that each label must appears exactly twice, we have the following hypergraph representation of a generalized tensor network.



Here, we use different colors to distinguish different hyperedges. A vertex tensor always has rank 1 and is not shown here since it does not change the contraction complexity. If we contract this tensor network in the column-wise order, the maximum intermediate tensor rank is $\sim L$, which can be seen by counting the number of colors at the cut.

Appendix C. Hard problems and tensor networks.

C.1. Maximal independent sets and maximal cliques. Since finding maximal cliques of a graph is equivalent to finding the maximal independent sets of its complement graph, in the following, we limit our discussion to finding maximal independent sets. Let us denote the neighborhood of a vertex v as $N(v)$ and the closed neighborhood of v as $N[v] = N(v) \cup \{v\}$. A maximal independent set I_m is an independent set where there does exist a vertex $v \in V$ such that $I_m \cap N[v] = \emptyset$, i.e. an independent set that can not become a larger one by adding a new vertex. To characterize the maximal independence restriction, one can quantify over a local set $N[v]$ with a local tensor:

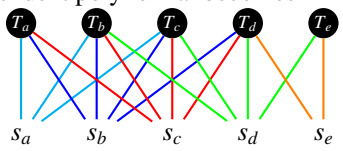
$$T(x_v, w_v)_{s_1,s_2,\ldots,s_{|N(v)|},s_v} = \begin{cases} s_v x_v^{w_v} & s_1 = s_2 = \ldots = s_{|N(v)|} = 0, \\ 1 - s_v & \text{otherwise.} \end{cases} \tag{C.1}$$

Intuitively, it means if all the neighbourhood vertices are not in I_m, i.e., $s_1 = s_2 = \ldots = s_{|N(v)|} = 0$, then v should be in I_m and contribute a factor x_v, otherwise, if any of the neighbourhood vertices is in I_m, then v cannot be in I_m. As an example, for a vertex of

degree 2, the resulting rank-3 tensor is

$$T(x_v, w_v) = \begin{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \\ \begin{pmatrix} x_v^{w_v} & 0 \\ 0 & 0 \end{pmatrix} \end{pmatrix}. \tag{C.2}$$

Let us consider the example in Sec. 3: its corresponding tensor network structure for computing the maximal independent polynomial becomes



One can see that the average rank 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.

By contracting this tensor network with generic element types, we can compute the maximal independent set properties such as the maximal independence polynomial and the enumeration of maximal independent sets. Similar to the independence polynomial, the maximal independence polynomial counts the number of maximal independent sets of various sizes [34]. Let $G = (V, E)$ be a graph, its maximal independence polynomial is defined as

$$D_m(G, x) = \sum_{k=0}^{\alpha(G)} b_k x^k, \tag{C.3}$$

where b_k is the number of maximal independent sets of size k. Comparing with the independence polynomial in Eq. (5.1), we have $b_k \leq a_k$ and $b_{\alpha(G)} = a_{\alpha(G)}$. $D_m(G, 1)$ counts the total number of maximal independent sets [27, 43], which to our knowledge, the best algorithm gives a time complexity $O(1.3642^{|V|})$ [27]. If we want to find an MIS, b_k can, in some cases, provide hints on the difficulty of finding the MIS using local algorithms [18].

We show the benchmark of computing the maximal independent set properties in Fig. 7, including a comparison to the Bron-Kerbosch algorithm from Julia package Graphs [2]. The treewidth of this tensor network is significantly larger, so benchmark with only a smaller graph size is feasible. The time for the tensor network 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 is only feasible up to a graph size around 70.

C.2. Matching problem. A k-matching in a graph $G = (V, E)$ is a set of k edges, no two of which have a vertex in common. We map an edge $(u, v) \in E$ to a degree of freedom $\langle u, v\rangle \in \{0, 1\}$ in a tensor network, where 1 means this edge is in the set and 0 otherwise. To characterize the matching, one can define a tensor of rank $d(v) = |N(v)|$ on vertex v

$$W_{\langle v,n_1\rangle,\langle v,n_2\rangle,\ldots,\langle v,n_{d(v)}\rangle} = \begin{cases} 1, & \sum_{i=1}^{d(v)} \langle v, n_i\rangle \leq 1, \\ 0, & \text{otherwise}, \end{cases} \tag{C.4}$$

and a tensor of rank 1 on the bond

$$B(w_{\langle u,v\rangle})_{\langle u,v\rangle} = \begin{cases} 1, & \langle u, v\rangle = 0 \\ x_{\langle u,v\rangle}^{w_{\langle u,v\rangle}}, & \langle u, v\rangle = 1, \end{cases} \tag{C.5}$$

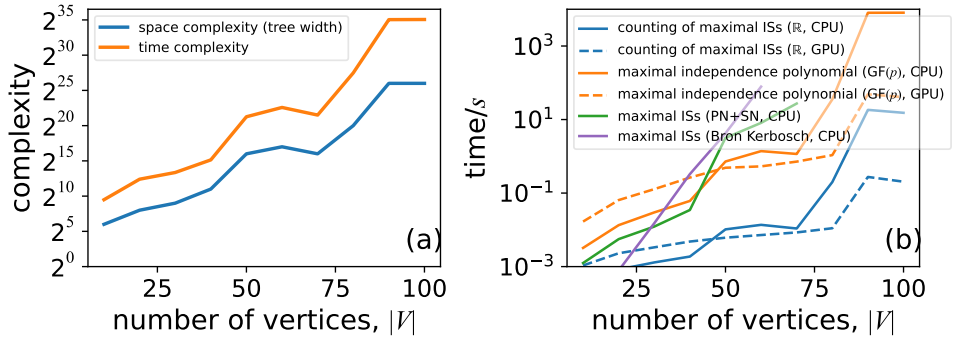


Figure 7: Benchmark results for computing different properties of maximal independent sets on a random three regular graph with different tensor element types. (a) treewidth versus the number of vertices for the benchmarked graphs. (b) The computing time for calculating the number of independent sets and enumerate all MISs.

where label $\langle v, u\rangle$ is equivalent to $\langle u, v\rangle$. The vertex tensor specifies the constraint that a vertex cannot be shared by two edges in the matched set, while an edge tensor carries the weights as the target to optimize. Let $x^{w_{\langle u,v\rangle}}_{\langle u,v\rangle} = x$, the tensor network contraction gives us the matching polynomial

$$M(G, x) = \sum_{k=1}^{|V|/2} c_k x^k, \tag{C.6}$$

where k is the size of a matched edges set, and a coefficients c_k is the number of k-matchings.

C.3. Vertex coloring. Let $G = (V, E)$ be a graph, a vertex coloring is an assignment of colors to each vertex $v \in V$ such that no edge connects two identically colored vertices. In a k-coloring problem, the number of different colors can be used is limited to less or equal to k. Let us use the 3-coloring problem as an example to show how to characterize with local tensors. For each vertex $v \in V$, we associate a degree of freedom $c_v \in \{0, 1, 2\}$ to it. Then we define a vertex tensor labelled by it as

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

where r_v, g_v and b_v are colors for labeling the configurations. For each edge (u, v), we specify the constraint carrying a weight w_{uv} on it by defining an edge tensor labelled by (c_u, c_v) as

$$B(x, w_{uv}) = \begin{pmatrix} 0 & x^{w_{uv}} & x^{w_{uv}} \\ x^{w_{uv}} & 0 & x^{w_{uv}} \\ x^{w_{uv}} & x^{w_{uv}} & 0 \end{pmatrix}. \tag{C.8}$$

The number of possible colorings can be obtained by contracting this tensor network by setting r_v, g_v and b_v to 1. Let $x^{w_{uv}} = x$, we have a graph polynomial, in which the k-th coefficient is the number of coloring with k bonds satisfied. If a graph is colorable, the maximum order of this polynomial should be equal to the number of edges in this graph. Similarly, one can define the k-coloring problem on edges by defining the tensor network on the line graph of G.

C.4. Cutting problem. In graph theory, a cut is a partition of the vertices of a graph into two disjoint subsets, which is also known as the boolean spin glass problem in statistic physics. Let $G = (V, E)$ be a graph, to reduce the cutting problem on G to the contraction of a tensor network, we first define a boolean degree of freedom $s_v \in \{0, 1\}$ for each vertex $v \in V$. Then for each edge $(u, v) \in E$, we define an edge matrix labelled by $s_u s_v$ as

$$B(x, w_{uv}) = \begin{pmatrix} 1 & x_v^{w_{uv}} \\ x_u^{w_{uv}} & 1 \end{pmatrix}, \tag{C.9}$$

where variables x_u and x_v represents a cut on this edge or a domain wall in an spin glass. Let $x_u^{w_{uv}} = x_v^{w_{uv}} = x$, we have a graph polynomial similar to previous ones, where its kth coefficient is two times the number of cuts with size k (i.e. cutting k edges).

C.5. Dominating Set. In graph theory, a dominating set for a graph $G = (V, E)$ is a subset $D \subseteq V$ such that every vertex not in D is adjacent to at least one member of D. To reduce this problem to the contraction of a tensor network, we first map a vertex $v \in V$ to a boolean degree of freedom $s_v \in \{0, 1\}$. Then for each vertex $v \in V$ we define a tensor on its closed neighborhood $\{v\} \cup N(v)$ as

$$T(x, w_v)_{s_1, s_2, \ldots, s_{|N(v)|}, s_v} = \begin{cases} 0_v & s_1 = s_2 = \ldots = s_{|N(v)|} = s_v = 0, \\ 1_v & s_v = 0, \\ x^{w_v} & \text{otherwise}, \end{cases} \tag{C.10}$$

where w_v is the weight of vertex v. This tensor means if both v and its neighbouring vertices are not in D, i.e., $s_1 = s_2 = \ldots = s_{|N(v)|} = s_v = 0$, this configuration is forbidden because v is not adjacent to any member in the set. Otherwise, if v is in D, this tensor contributes a multiplicative factor $x_v^{w_v}$ to the output. The graph polynomial for the dominating set problem is known as the domination polynomial [4], which is defined as

$$D(G, x) = \sum_{k=0}^{\gamma(G)} d_k x^k, \tag{C.11}$$

where d_k is the number of dominating sets of size k.

C.6. Boolean satisfiability Problem. The boolean satisfiability problem is the problem of determining if there exists an interpretation that satisfies a given boolean formula. One can specify a satisfiable problem in the conjuctive normal form (CNF): a conjuction of disjunctions of boolean literals, where a disjunction of boolean literals is also named a clause. To reduce the problem of solving CNF to a tensor network contraction, we first map a boolean literal a and its negation $\neg a$ to a same degree of freedom (label) $s_a \in \{0, 1\}$, where 0 stands for variable a having value `false` while 1 stands for having value `true`. Then we map a clause to a tensor. For example, the k-th clause $\neg a \vee b \vee \neg c$ can be mapped to a tensor labeled by (s_a, s_b, s_c).

$$C_k = \begin{pmatrix} \begin{pmatrix} x^{w_k} & x_b^{w_k} \\ x_a^{w_k} & x_{ab}^{w_k} \end{pmatrix} \\ \begin{pmatrix} x_c^{w_k} & x_{bc}^{w_k} \\ 1_{ac} & x_{abc}^{w_k} \end{pmatrix} \end{pmatrix}, \tag{C.12}$$

where a weight w_k is associated with a clause. There is only one entry $(s_a, s_b, s_c) = (1, 0, 1)$ that makes this clause unsatisfied. If we contract this tensor network, we will get a multiplicative factor x whenever there is a clause satisfied. Let $x_{\ldots}^{w_k} = x$, one can get a polynomial, in which the k-th coefficient gives the number of assignments with k clauses satisfied.

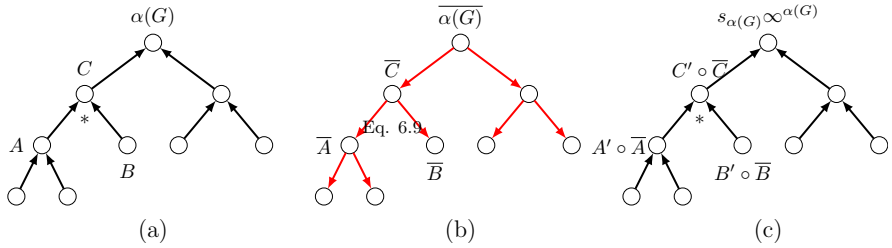


Figure 8: Bounded enumeration of maximum independent sets. In these graphs, a circle is a tensor, an arrow specifies execution direction of a function and $\circ$ is the Hadamard (element-wise) multiplication. $\overline{A}$ means the boolean mask of A. (a) is the forward pass with algebra Eq. (6.3: T) for computing $\alpha(G)$. (b) is the backward pass for computing boolean gradients as masks. (c) is the masked forward pass with algebra Eq. (7.3: P1+SN) for enumerating configurations.

C.7. Set packing. Set packing is the hypergraph generalization of the maximum independent set problem, where a set corresponds to a vertex and an element corresponds to a hyperedge. To solve the set packing problem, we can just remove the rank-2 restriction of the edge tensor in Eq. (4.2)

$$B_{s_u,s_v,\ldots,s_w} = \begin{cases} 1, & s_u + s_v + \ldots + s_w \leq 1, \\ 0, & \text{otherwise.} \end{cases} \tag{C.13}$$

Appendix D. Bounding the MIS enumeration space.

When we use the algebra in Eq. (7.3: 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 information from the computation of the MIS size $\alpha(G)$. As shown in Fig. 8, (a) we first compute the value of $\alpha(G)$ with tropical algebra and cache all intermediate tensors. (b) 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. (c) Finally, we perform masked tensor network contraction (i.e. discarding the unnecessary intermediate configurations) using the element type with the algebra in Eq. (7.3: P1+SN) to obtain all MIS 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

$$A_{ij} \odot B_{jk} \leq C_{ik}. \tag{D.1}$$

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 identify elements with nonzero gradients in A and B, but if doing this directly, one loses the advantage of using BLAS libraries [3] for high performance. Since $A_{ij} \odot B_{jk} = A_{ij} + B_{jk}$, one can move B_{jk} to the right hand side of the inequality:

$$A_{ij} \leq C_{ik} \odot B_{jk}^{\circ -1} \tag{D.2}$$

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:

(D.3)
$$A_{ij} \le \min_k (C_{ik} \odot B_{jk}^{\circ-1}) = \left(\max_k \left(C_{ik}^{\circ-1} \odot B_{jk}\right)\right)^{\circ-1} = \left(\bigoplus_k \left(C_{ik}^{\circ-1} \odot B_{jk}\right)\right)^{\circ-1} = \left(C^{\circ-1} B^{\mathsf{T}}\right)_{ij}^{\circ-1}.$$

On the right hand side, we transform the operation into a tropical matrix multiplication so that we can utilize the fast tropical BLAS routines [3]. 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

(D.4)
$$\overline{A}_{ij} = \delta\left(A_{ij}, \left(\left(C^{\circ-1} \circ \overline{C}\right) B^{\mathsf{T}}\right)_{ij}^{\circ-1}\right),$$

where δ is the Dirac delta function that returns one if two arguments have the same value and zero otherwise, $\circ$ 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.

Appendix E. The fitting approach to computing the independence polynomial. In this section, 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:

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

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-size independent sets. 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):

(E.2: GF(p))
$$\begin{aligned} x \oplus y &= x + y \pmod p, \\ x \odot y &= xy \pmod p, \\ \mathbb{0} &= 0, \\ \mathbb{1} &= 1. \end{aligned}$$

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

1. One can use Gaussian elimination [28] to solve the linear equation Eq. (E.1) 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 E.1 to compute the independence polynomial exactly without introducing space overheads. The algorithm 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 $\chi = (x_0, x_1, \ldots, x_{\alpha(G)})$ on GF(p) and denote the outputs as $(y_0, y_1, \ldots, y_{\alpha(G)}) \pmod{p}$. Then we solve Eq. (E.1) using Gaussian elimination on GF(p) to find the coefficient modulo p, $A_p \equiv (a_0, a_1, \ldots, a_{\alpha(G)}) \pmod{p}$. 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 the 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.

Algorithm E.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

Appendix F. The discrete Fourier transform approach to computing the independence polynomial.

In section 5, we show that the independence polynomial can be obtained by solving the linear equation Eq. (E.1). 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

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

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 G. Integer sequence formed by the number of independent sets. We computed the number of independent sets on square lattices and King's graphs with our generic tensor network contraction algorithm on GPUs. The tensor element type is finite-field algebra so that we can reach arbitrary precision. We also computed the independence polynomial exactly for these lattices which can be found in our Github repo.

Table 2: The number of independent sets for square lattice graphs of size $L\times L$. This forms the integer sequence OEIS A006506. Here we only include two updated entries for $L = 38, 39$, which, to our knowledge, has not been computed before [13].

L	square lattice graphs
38	616 412 251 028 728 207 385 738 562 656 236 093 713 609 747 387 533 907 560 081 990 229 746 115 948 572 583 817 557 035 128 726 922 565 913 748 716 778 414 190 432 479 964 245 067 083 441 583 742 870 993 696 157 129 887 194 203 643 048 435 362 875 885 498 554 979 326 352 127 528 330 481 118 313 702 375 541 902 300 956 879 563 063 343 972 979
39	29 855 612 447 544 274 159 031 389 813 027 239 335 497 014 990 491 494 036 487 199 167 155 042 005 286 230 480 609 472 592 158 583 920 411 213 748 368 073 011 775 053 878 033 685 239 323 444 700 725 664 632 236 525 923 258 394 737 964 155 747 730 125 966 370 906 864 022 395 459 136 352 378 231 301 643 917 282 836 792 261 715 266 731 741 625 623 207 330 411 607

Appendix H. Computing maximum sum combination. Given two sets A and B of the same size n. It is known that the maximum n sum combination of A and B can be computed in time $n\log(n)$. The standard approach to solve the sum combination problem requires storing the variables in a heap - a highly dynamic binary tree structure which can can be much slower to manipulate than arrays. In the following, we show an algorithm that has roughly the same complexity (given the data range is not exponentially large) but does not need a heap. This algorithm first sorts both A and B, then use the bisection to find the n-th largest value in the sum combination. The key point is we can count the number of entries greater than a specific value in the sum combination of A and B in linear time. We summarize the algorithm as in Algorithm H.1.

Function `collect_geq` is similar the `count_geq` except the counting is replace by the item collection. Note in function `count_geq`, variable k monotoneously increase while q monotoneously decrease in each iteration, the total number of iterations must be smaller or equal to $2n$. Here for simplicity, we do not handle the $-\infty$s in A and B and the degeneracy in the sums. They need to be taken seriously in practical implementations.

Appendix I. Technical guide.

This appendix covers some technical aspects of this paper, including an open source package and a gist about how to implement this package. If you are only interested in using, please checking our GitHub repository for an efficient and full featured implementation aimed for productively: https://github.com/Happy-Diode/GraphTensorNetworks.jl. One can install these packages by opening a Julia REPL, type `]` to enter the `pkg>` mode and type, e.g.

Algorithm H.1 Fast sum combination without using heap

Let A and B be two sets of size n
`// sort` A `and` B `in ascending order`
$A \leftarrow \text{sort}(A)$
$B \leftarrow \text{sort}(B)$
`// use bisection to find the` n`-th largest value in sum combination`
high $\leftarrow A_n + B_n$
low $\leftarrow A_1 + B_n$
while *true* **do**
    mid $\leftarrow$ (high + low)/2
    $c \leftarrow$ `count_geq`(n, A, B, mid)
    if $c > n$ **then**
        low $\leftarrow$ mid
    else if $c = n$ **then**
        return `collect_geq`(n, A, B, mid)
    else
        high $\leftarrow$ mid
    end
end
function `count_geq`*(n, A, B, v)*
    $k \leftarrow 1$; `// number of entries in` A `s.t.` $a + b \geq v$
    $a \leftarrow A_n$; `// the smallest entry in` A `s.t.` $a + b \geq v$
    $c \leftarrow 0$; `// the counting of sum combinations s.t.` $a + b \geq v$
    for $q = n, n-1 \ldots 1$ **do**
        $b \leftarrow B_{n-q+1}$
        while $k < n$ ***and*** $a + b \geq v$ **do**
            $k \leftarrow k + 1$
            $a \leftarrow A_{n-k+1}$
        end
        if $a + b \geq v$ **then**
            $c \leftarrow c + k$
        else
            $c \leftarrow c + k - 1$
        end
    end
    return c
end

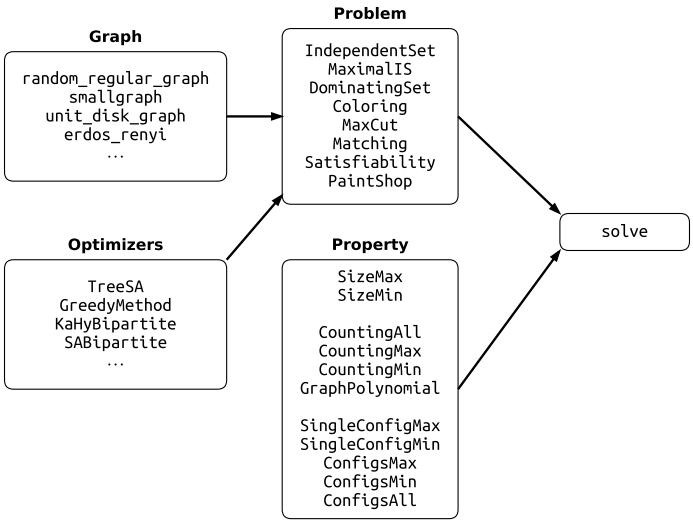
```
pkg> add GraphTensorNetworks
```

To use it for solving solution space properties, just open a Julia REPL and type

```
julia> using GraphTensorNetworks, Graphs

julia> solve(
           IndependentSet(
               Graphs.random_regular_graph(20, 3);
               optimizer = TreeSA(),
               weights = NoWeight(),
               openvertices = ()
           ),
           GraphPolynomial();
           usecuda=false
       )
0-dimensional Array{Polynomial{BigInt, :x}, 0}:
Polynomial(1 + 20*x + 160*x^2 + 659*x^3 + 1500*x^4 + 1883*x^5 + 1223*x^6 + 347*x^7 + 25*x^8)
```

Here the main function `solve` takes three inputs, the problem instance of type `IndependentSet`, the property instance of type `GraphPolynomial` and an optional key word argument `usecuda` to decide use GPU or not. If one wants to use GPU to accelerate the computation, statement "`using CUDA`" must be executed before calling the solve function. The problem instance takes four arguments to initialize, the only positional argument is the graph instance one wants to solve, the key word argument `optimizer` is for tensor network optimization, the key word argument `weights` specifies the weights of vertices as either a vector or `NoWeight()` and the keyword argument `openvertices` allows one to specify the degrees of freedom not summed over, i.e. returning a tensor rather than its sum. Here, we use `TreeSA` method in Ref. [36]. The first execution of the code will trigger Julia's just in time compiling, which is a bit slow, but the following runs will be fast. The following diagram lists possible combinations of input arguments, where graphs are mainly defined in package `Graphs`, and the rest can be found in `GraphTensorNetworks`.



The code we will show below is a gist of how the above package is implemented, which is mainly for the pedagogical purpose. It covers most of the topics in the paper without caring much about the performance. This project depends on multiple open source packages in the Julia ecosystem:

OMEinsum and OMEinsumContractionOrders are packages providing the support for Einstein's (or tensor network) notation and contraction order optimizations. `OMEinsumContractionOrders` implements state-of-the-art algorithms for finding the optimal contraction order for a tensor network, including the KaHypar+Greedy [29, 48] and local transformation based approaches [36],

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.

They can be installed in a similar way to `GraphTensorNetworks`. After installing the required packages, one can open a Julia REPL, then copy and paste the following code into it.

```
using OMEinsum, OMEinsumContractionOrders
using Graphs
using Random

# generate a random regular graph of size 100, degree 3
graph = (Random.seed!(2); Graphs.random_regular_graph(50, 3))

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

# an einsum contraction without a contraction order specified is called `EinCode`,
# an einsum contraction having a contraction order (specified as a tree structure) is called `
     NestedEinsum`.
# assign each label a dimension-2, it will be used in the contraction order optimization
# `uniquelabels` function extracts the tensor labels into a vector.
size_dict = Dict([s=>2 for s in uniquelabels(code)])
# optimize the contraction order using the `TreeSA` method; the target space complexity is 2^17
optimized_code = optimize_code(code, size_dict, TreeSA())
println("time/space complexity is $(OMEinsum.timespace_complexity(optimized_code, size_dict))")

# a function for computing the independence polynomial
function independence_polynomial(x::T, code) where {T}
   xs = map(getixsv(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 THE MAXIMUM INDEPENDENT SET SIZE AND ITS COUNTING/DEGENERACY ###########

# using Tropical numbers to compute the MIS size and the 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 THE 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)[])")

########## FINDING MIS CONFIGURATIONS ###########

# define the set algebra
struct ConfigEnumerator{N}
    # NOTE: BitVector is dynamic and it 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`
    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}` 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(uniquelabels(code))])
    N = length(vertex_index)  # number of vertices
    xs = map(getixsv(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)"
     )

# direct enumeration of configurations 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)")
```