COMPUTING SOLUTION SPACE PROPERTIES OF COMBINATORIAL OPTIMIZATION PROBLEMS VIA GENERIC TENSOR NETWORKS

*JIN-GUO LIU[†], XUN GAO[‡], MADELYN CAIN[†], MIKHAIL D. LUKIN[†], AND SHENG-TAO WANG[§]

Abstract. We introduce a unified framework to compute the solution space properties of a broad class of combinatorial optimization problems. These properties include finding one of the optimum solutions, counting the number of solutions of a given size, and enumeration and sampling of solutions of a given size. Using the independent set problem as an example, we show how all these solution space properties can be computed in the unified approach of generic tensor networks. We demonstrate the versatility of this computational tool by applying it to several examples, including computing the entropy constant for hardcore lattice gases, studying the overlap gap properties, and analyzing the performance of quantum and classical algorithms for finding maximum independent sets.

Key words. solution space property, tensor networks, maximum independent set, independence polynomial, generic programming, combinatorial optimization

AMS subject classifications. 05C31, 14N07

1. Introduction.

An important class of problems in graph theory and combinatorial optimization can be formulated as satisfiability problems involving constraints specified over a vertex and its neighborhood. These include, for example, the independent set, the cutting problem, the dominating set, the set packing and the set covering, the vertex coloring problem, the K-SAT, the clique problem, and the vertex cover problem [45]. These problems have a wide range of applications in scheduling, logistics, wireless networks and telecommunication, and computer vision, among others [12, 57]. Finding an optimum solution for these problems is typically NP-hard [34].

In this Article, we introduce a unified framework to compute a broad class of properties associated with the solutions of these problems, beyond just finding an optimum solution. We call them *solution space properties*. In practice, these can be much harder to compute (corresponding e.g. to #P-complete class [45]). However, these properties can be crucial for understanding detailed properties of hard combinatorial optimization problems. These *solution space properties* can include not only the maximum or minimum set size but also the number of sets at a given size, enumeration of all sets at a given size, and direct sampling of such sets when they are too large to be fit into memory. They can be used to understand the hardness of finding an optimum solution for a given problem instance and the performance of a specific solver. For example, the number of configurations at different sizes can inform how likely a simulated annealing algorithm will be trapped in local minima at certain sizes [58]. The pair-wise Hamming distance distribution of configurations at a given size can indicate the presence or absence of the overlap gap property [27, 26], which can be used to bound the performance of local optimization algorithms. In a recent experiment based on a Rydberg atom array quantum computer, the counting and the configuration space connectivity information was used to find maximum independent set (MIS) problem instances that are hard for simulated annealing and to evaluate the corresponding quantum algorithm performance [18]. The need for understanding these important aspects of combinatorial optimization motivates us to find methodologies to compute these solution space properties.

*JIN-GUO LIU AND XUN GAO CONTRIBUTED EQUALLY TO THIS WORK.

[†]Department of Physics, Harvard University, Cambridge, Massachusetts 02138, USA; QuEra Computing Inc., 1284 Soldiers Field Road, Boston, MA, 02135, USA (jinguoliu@g.harvard.edu).

[‡]Department of Physics, Harvard University, Cambridge, Massachusetts 02138, USA (xungao@g.harvard.edu).

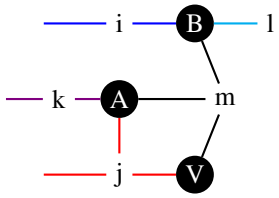
[§]QuEra Computing Inc., 1284 Soldiers Field Road, Boston, MA, 02135, USA

To this end, we show how to obtain all of these seemingly unrelated properties in a unified approach using *generic tensor networks*. Tensor networks are a computational model widely used in condensed matter physics [46], quantum computing [44], big data [14] and mathematics [47]. They are also known as the sum-product networks in probabilistic modeling [9] or `einsum` in linear algebra libraries such as NumPy [32]. Recent progress in simulating quantum circuits with tensor networks [31, 48, 36] makes it possible to contract a randomly structured sparse tensor network with up to thousands of tensors in a reasonable time. In previous studies, the data types of the tensor elements are typically restricted to standard number types such as real numbers and complex numbers. Here, we extend to *generic tensor networks* by generalizing the tensor element data types to any type that has the algebraic structure of a commutative semiring. In what follows, for clarity of presentation, we focus on the independent set problem in the main text, and show how to compute the solution space properties for other combinatorial optimization problems in Appendix C. The latter include cutting, matching, vertex coloring, satisfiability, dominating set, set packing, set covering, and the clique problem.

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. Subsequently, we explain how to engineer the element types to compute various solution space properties in Sec. 6, Sec. 7, and Sec. 8. Lastly, we benchmark our algorithms in Sec. 9 and provide three example applications 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, and an indicator of which tensor is the output. The operation to evaluate a tensor network is called *contraction*, which is defined as a summation of tensor element products over the labels not appearing in the output tensor. For example, 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 matrices (two-dimensional 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 tensor network is an open (meaning an edge can connect to external vertices) 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. Our notation is a minor generalization of the standard tensor network notation used in physics as we do not restrict the number of times a label can appear in the tensors. While this generalized form is equivalent in representation power, it can have smaller contraction complexity as 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. In previous works relating tensor networks and combinatoric problems [38, 8], the element types in the tensor networks are

limited to standard number types such as floating-point numbers and integers. We propose to use more general element types with a certain algebraic property. With different data types, we can solve different problems within the same unified framework. This idea of using the same program for different purposes is also called 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 "without loss of efficiency". By the most general setting, we mean that a single program should work correctly for the most general 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 $\mathbb{1}$. Then, when the program takes a matrix as an input instead of a standard number type, it computes the matrix power correctly without rewriting the program. The second aspect is about efficiency. For dynamically typed languages such as Python, the type information is not available for type-specific optimizations at the compilation stage. Therefore, one can easily write code that works for general input types, 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 (i.e. the type information can be accessed at the compilation stage) languages such as C++ and Julia [7]. C++ uses templates for generic programming while Julia takes advantage of just-in-time compilation and multiple dispatches. When these languages "see" a new input type, the compiler recompiles the generic program for the new type to generate an efficient binary. A myriad of optimizations can be done during the compilation. For example, the compiler can optimize the memory layout of immutable elements with fixed sizes in an array to speed up array indexing. In Julia, if a type is immutable and contains no references to other values, an array of that type can even be compiled to graphics processing units (GPU) for faster computation [6].

This motivates us to identify the most general tensor element type allowed in a tensor network contraction program. We find that as long as the tensor elements are members of a commutative semiring, the tensor network contraction will be well defined and the result will be independent of the contraction order. Here, a commutative semiring is a field that needs not to have an additive inverse and a multiplicative inverse. Giving up these nice properties of fields has significant implications for tensor computation: tensor network compression algorithms might not be 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 an additive inverse [37]. Here, we only use the commutative properties of an algebra for optimizing the tensor network contraction order. 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 $a, b, c \in S$.

$$
\begin{aligned}
(a \oplus b) \oplus c &= a \oplus (b \oplus c) \\
a \oplus \mathbb{0} &= \mathbb{0} \oplus a = a \\
a \oplus b &= b \oplus a
\end{aligned}
\qquad \triangleright \text{ commutative monoid } \oplus \text{ with identity } \mathbb{0}
$$

$$
\begin{aligned}
(a \odot b) \odot c &= a \odot (b \odot c) \\
a \odot \mathbb{1} &= \mathbb{1} \odot a = a \\
a \odot b &= b \odot a
\end{aligned}
\qquad \triangleright \text{ commutative monoid } \odot \text{ with identity } \mathbb{1}
$$

$$
\begin{aligned}
a \odot (b \oplus c) &= a \odot b \oplus a \odot c \\
(a \oplus b) \odot c &= a \odot c \oplus b \odot c
\end{aligned}
\qquad \triangleright \text{ left and right distributive}
$$

$$
a \odot \mathbb{0} = \mathbb{0} \odot a = \mathbb{0}
$$

In the following sections, we will show how to compute solution space properties of independent sets using the same tensor network contraction algorithm by engineering tensor element algebra. The Venn diagram in Fig. 1 shows the different types of algebra we will introduce in the main text and their relation, and Table 1 summarizes which solution space properties can be computed by which tensor element types.

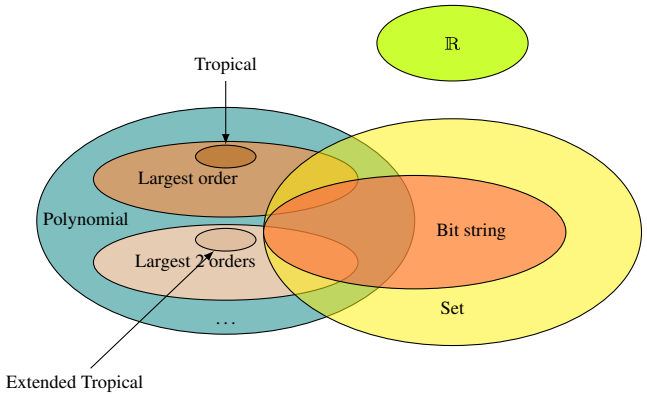


Figure 1: The tensor network element types used in this work and their relations. The overlap between two ellipses indicates that a new algebra can be created by combining those two types of algebra. "Largest order" and "Largest 2 orders" mean truncating the polynomial by only keeping its largest or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

4. Tensor network representation of independent sets. This section describes the reduction of the independent set problem to a tensor network contraction problem. An alternative interpretation, perhaps more accessible to physicists, can be found in Appendix A, where we introduce the reduction from the energy model of hardcore lattice gases [17, 21]. Let $G = (V, E)$ be an 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

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.

pair $u, v \in I$, we have $(u, v) \notin E$. To reduce the independent set problem on G to a tensor network contraction, we first map a vertex $v \in V$ to a label $s_v \in \{0, 1\}$ of dimension 2, where we use 0 (1) to denote a vertex is absent (present) in the set. For each vertex v, we define a parameterized rank-one tensor indexed by s_v as

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

where x_v is a variable associated with v. Tensor elements can be indexed by subscripts, e.g. $W(x_v^{w_v})_0 = 1$ 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, for each edge $(u, v) \in E$, we define a matrix B indexed by s_u and s_v as

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

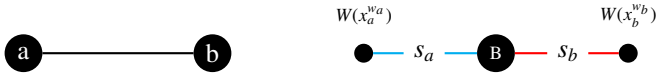
which encodes the independent set constraint since $B_{11} = 0$, meaning the two vertices cannot be both in the independent set.

The corresponding tensor network contraction can be defined as

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

where the summation runs over all $2^{|V|}$ vertex configurations $\{s_1, s_2, \ldots, s_{|V|}\}$ and accumulates the product of tensor elements to the output P. A vertex tensor element $W(x_v^{w_v})_{s_v}$ contributes a multiplicative factor $x_v^{w_v}$ whenever v is in the set.

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



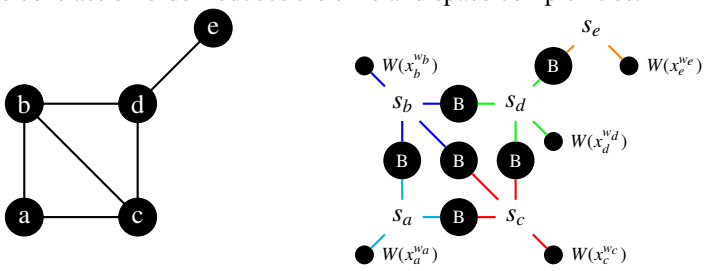
In the graphical representation of the tensor network on the right panel, we use a circle to represent a tensor, a hyperedge in cyan color to represent the degree of freedom s_a, and a hyperedge in red color to represent the degree of freedom s_b. Tensors sharing the same degree of freedom are connected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

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

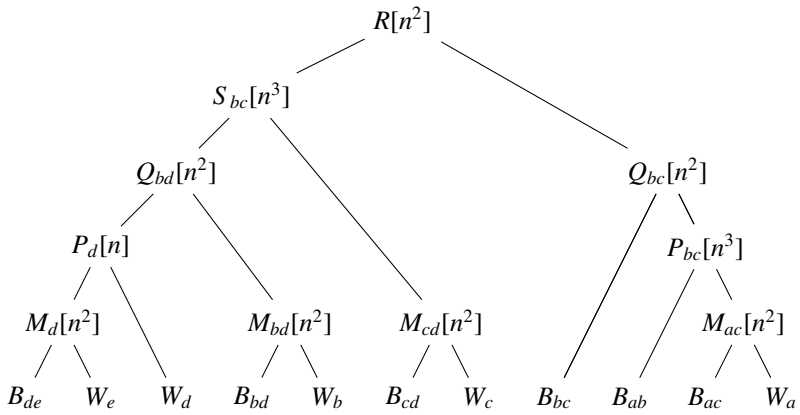
The resulting polynomial represents 3 different independent sets {}, $\{a\}$, and $\{b\}$, with weights being 0, w_a, and w_b respectively.

For a general graph, it is computationally inefficient to evaluate Eq. (4.3) by directly summing up the $2^{|V|}$ products. A better approach to evaluate a tensor network is: to find a good pair-wise tensor contraction order as a binary tree and then contract two tensors at a time by this order. Let us denote the line graph of the hypergraph representation of a tensor network as G'. A contraction order of this tensor network corresponds to a tree decomposition of G' and the largest intermediate tensor has a rank equal to the width of this tree decomposition [44]. Therefore, an optimal (in terms of space complexity) contraction order corresponds to the tree decomposition of G' with the smallest width (or the treewidth $\mathrm{tw}(G')$). In the independent set problem, the treewidth of G' is the same as the treewidth of the graph this tensor network is mapped from. In practice, it is difficult to find an optimal contraction order for large tensor networks because finding the treewidth is a well-known NP-hard problem. However, it is easy to find a close-to-optimal contraction order within typically a few minutes with a heuristic algorithm [38, 36]. 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.

Example 3. In this example, 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 node stores the contraction result; the leaves are input tensors and the rest of the 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 the degree of freedoms, which is 2 in a tensor network mapped from an independent set problem. One can easily check the largest tensor in contraction has a space complexity $O(n^2)$ and this is the smallest among all possible contraction trees, i.e. the treewidth of the original 5-vertex graph is 2. The time complexity is $O(n^3)$, which is much smaller than that of direct evaluation $O(n^5)$.

5. Independence polynomial. Let $x_i = x$ and $w_i = 1$, Eq. (4.3) corresponds to the independence polynomial [33, 22]:

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

where a_k is the number of independent sets of size k, $\alpha(G) \equiv \max_I \sum_{v\in I} w_v$ is the size of a maximum independent set and it is also called the independence number. An independence polynomial is a useful graph characteristic related to, for example, the partition functions [39, 59] and Euler characteristics of the independence complex [10, 40]. By assigning a real number to x, one can evaluate this independence polynomial for this specific value directly using tensor network contraction. For example, the total number of independent sets can be evaluated as $I(G, 1)$. 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 different sizes. To this end, let us create a polynomial number data type by representing a polynomial $a_0 + a_1 x + \ldots + a_k x^k$ as a coefficient vector $a = (a_0, a_1, \ldots, a_k) \in \mathbb{R}^k$, e.g. x is represented as $(0, 1)$. Then we can define an algebra among coefficient vectors, including a redefinition of additive identity and multiplicative identity. To avoid potential confusion, let us denote the additive identity as $\mathbb{0}$, and the multiplicative identity is $\mathbb{1}$. The algebra between the polynomials number a of order k_a and b of order k_b is defined as

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

where $\oplus$ and $\odot$ are the standard polynomial addition and multiplication operations. The multiplication can be evaluated in time $O(k\log(k))$ using the convolution theorem [53], where $k = \max(k_a, k_b)$. The tensors W and B, which are introduced in the previous section (Sec. 4), 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 this polynomial type, we have the exact representation of the independence polynomial. In practice, 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 procedure to fit the coefficients of this polynomial. This seemingly more time and space-efficient approach suffer from precision issues in practice. The data ranges of standard integer types are too small to cover many practical use cases, while the floating-point numbers may have round-off errors that are much larger than the value itself. These are because the number of independent sets at different sizes may vary by tens or even hundreds of orders of magnitude. For practical methods to evaluate these coefficients, we refer readers to Appendix E, where we provide an accurate and memory-efficient method to find the polynomial by contracting and fitting on finite field algebra. Alternatively, one can use the method in Appendix F to fit this polynomial faster with complex numbers with controllable round-off errors. For simplicity, we use this less efficient polynomial algebra for the discussion in the main text.

6. Maximum independent sets and its counting.

6.1. Tropical algebra for finding the MIS size and counting MISs. 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 one with 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

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

Here, we have generalized the previous polynomial to the Laurent polynomial to define the zero-element properly. 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 compute the spin glass ground state energy and its degeneracy. For independent set calculations here, the vertex tensor and edge tensor becomes:

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

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 the 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. It can be used, for example, to count the maximal independent sets with the smallest cardinality, where a maximal independent set is an independent set that cannot be made larger by adding a new vertex into 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 include, for example, the enumeration of all independent sets, the enumeration of all MISs, and the enumeration of independent sets with the largest several sizes. Recall that in the definition of a vertex tensor in Eq. (4.1), variables carry labels, so that one can read out all independent sets directly from the output polynomials. The multiplication between labeled variables is commutative while the summation of labeled variables forms a set. Intuitively, one can use a bit string as the representation of a labeled variable and use the bit-wise or $\vee^\circ$ as the multiplication operation. For example, in a 5-vertex graph, x_2 and x_5 can be represented as 01000 and 00001 respectively and their multiplication $x_2 x_5$ can be represented as $01000 \vee^\circ 00001 = 01001$. To enumerate all independent sets, we designed an algebra 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 sizes $\alpha(G)$ and $\alpha(G) - 1$. As long as the coefficients in a truncated polynomial are members of a commutative semiring, the polynomial itself is a commutative semiring. For example, to enumerate only the MISs, we can define a combined element type $s_k \infty^k$, where the coefficients follow the algebra in Eq. (7.1: SN) and the 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

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

The enumeration of all MIS configurations corresponds to the contraction of this tensor network. However, direct contraction might have significant space overheads for keeping too many intermediate states irrelevant to the final maximum independent sets. We introduce the bounding technique in Appendix D to avoid this issue. One may also be interested in the more studied maximal independent sets [11, 19, 35] enumeration. It will be discussed in Appendix C.1 since it requires using a different tensor network structure.

If one is interested in obtaining only one MIS configuration, they can just keep one configuration in each tensor element to save the computational effort. By replacing the sets of bit strings in Eq. (7.1: SN) with a single bit string, we have the following algebra

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

The `select` function picks one of σ and τ by some criteria. It can be picking the one smaller in the lexicographical order such that the addition operation is commutative and associative. In most cases, it is completely fine for the `select` function to pick a random one (not commutative and associative anymore) to generate a random MIS.

7.2. Sampling extremely large configuration space. When the problem scale becomes larger, a set of all bitstrings might be impossible to fit into any type of storage. To get something meaningful out of the 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. Each 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, and $left$ and $right$ are left and right operands of a `SUM` or `PROD` node.

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

The vertex tensor and edge tensor become

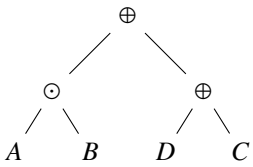
(7.7)
$$W^{\mathrm{EXPR}}((\texttt{LEAF}, \boldsymbol{e}_i, /, /)) = \begin{pmatrix} \mathbb{1} \\ (\texttt{LEAF}, \boldsymbol{e}_i, /, /) \end{pmatrix}, \qquad B^{\mathrm{EXPR}} = \begin{pmatrix} \mathbb{1} & \mathbb{1} \\ \mathbb{1} & \mathbb{0} \end{pmatrix}.$$

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 storage. Except using the sum-product expression tree directly as tensor elements, one can also let it be the coefficients of a (truncated) polynomial to compute such trees for independent sets with the largest several sizes. Although likely, one cannot collect all configurations represented by a sum-product expression tree into a vector due to its extremely large size, they can draw unbiased samples from it by direct sampling. The sampling program starts from the root node and descends this tree recursively to the left and right siblings. If a node has type `SUM`, the program draws samples from the left and right siblings with a probability decided by the size of each sub-tree and returns the union of samples. Otherwise, if a node has type `PROD`, the program draws two sets of samples of equal sizes from its left and right siblings and returns the element-wise multiplication ($\vee^{\circ}$) of them. The recursion stops at a `LEAF` node having size 1 or a `ZERO` node having size 0. In a sum-product expression tree, the number of configurations of 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 sub-trees A, B, C and D can be any of the four types of nodes. This sum-product expression tree can be represented diagrammatically as the following:



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

8. Weighted graphs. All the solution space properties and the corresponding algebra on unweighted graphs still hold for integer-weighted graphs, while for general weighted graphs, the independence polynomial is not well defined anymore. For general weighted graphs, it is more useful to know the k maximum weighted sets and their sizes. They can be computed by the extended tropical algebra, which is a natural generalization of the max-plus tropical algebra:

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

where $\texttt{largest}(s, k)$ means truncating the set s by only keeping its k largest values. The computation of $s \odot t$ is a maximum sum combination problem that can be done in time $O(k \log(k))$. An efficient algorithm for computing the maximum sum combination problem can be found in Appendix H. The vertex tensor and edge tensor become

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

where we have used the equation $(-\infty^{\otimes k-1} \otimes 1)^{w_v} = -\infty^{\otimes k-1} \otimes w_v$. To find solutions corresponding to the largest k sizes, one can combine the extended tropical algebra with the bit string algebra (Eq. (7.5: S1)). Since the $\oplus$ operation of the configuration sampler will not be used in the combined algebra, the resulting configurations are deterministic and complete.

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 in all benchmarks are random three-regular graphs, which have treewidth that is asymptotically smaller than $|V|/6$ [23]. In this benchmark, we do not include traditional algorithms for finding the MIS sizes such as branching [56, 52] or dynamic programming [16, 24]. To the best of our knowledge, these algorithms are not suitable for computing most of the solution space properties mentioned in this paper. The main goal of this section is to show the relative computation time for calculating different solution space properties.

Figure 2(a) shows the time and space complexity of tensor network contraction for different graph sizes. The contraction order is obtained using the local search algorithm in Ref. [36]. If we assume our contraction-order finding program has found the optimal treewidth, which is very likely to be true, the space complexity is the same as the treewidth

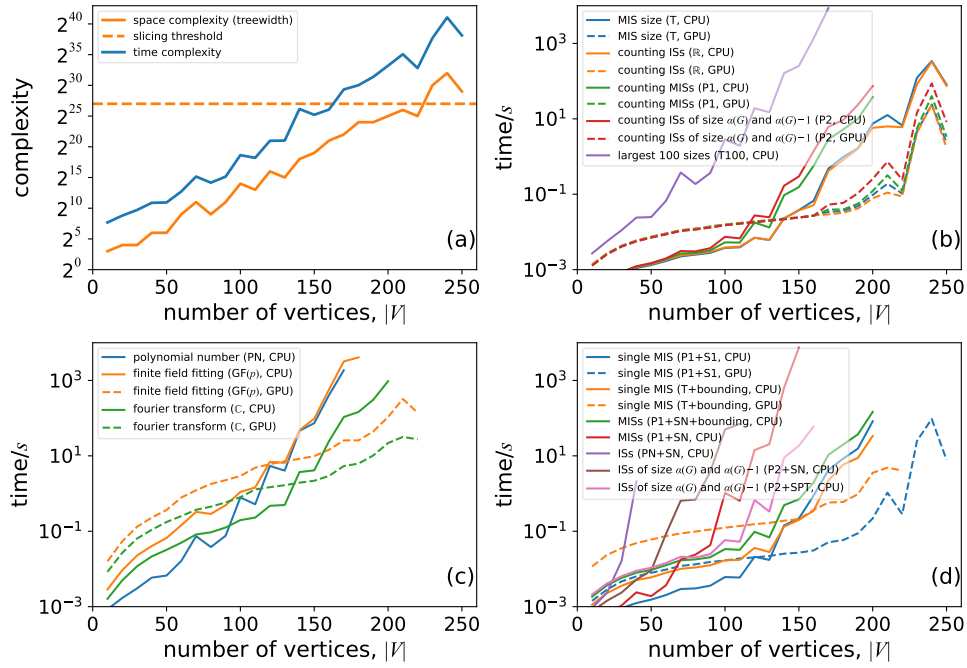


Figure 2: Benchmark results for computing different solution space properties of independent sets of random three-regular graphs with different tensor element types. The time in these plots only includes tensor network contraction, without taking into account the contraction order finding and just-in-time 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 computation time for calculating the MIS size and for counting 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 set sizes. (c) The computation time for calculating the independence polynomials with different approaches. (d) The computation time for configuration enumeration, including single MIS configuration, the enumeration of all independent set configurations, all MIS configurations, all independent sets, and all independent set configurations having size $\alpha(G)$ and $\alpha(G)-1$.

of the problem graph. Slicing technique [36] has been used for graphs with space complexity greater than 2^{27} (above the yellow dashed line) to fit the computation into a 16GB memory. One can see that all the computation times in figure 2 (b), (c), and (d) have a strong correlation with the predicted time and space complexity. While in panel (d), the computation time of configuration enumeration and sum-product expression tree generation also strongly correlates with other factors such as the configuration space size. Among these benchmarks, computational tasks with data types real numbers, complex numbers, or tropical numbers (CPU only) can utilize fast basic linear algebra subprograms (BLAS) functions. These tasks usually compute much faster than ones with other element types in the same category. Immutable data types with no reference to other values can be compiled to GPU devices that run much faster than CPUs in all cases when the problem scale is big enough. These data types do not include those defined in Eq. (5.2: PN), Eq. (7.1: SN),

Eq. (8.1: Tk) and Eq. (7.6: EXPR) or a data type containing them as a part. 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 in Appendix D improves the performance for more than one order of magnitude in enumerating the MISs. The bounding technique 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 applications.

10.1. Number of independent sets and entropy constant for 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 of 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 $\lfloor pL^2 \rceil$ sites 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)), which is the ensemble of graphs used in Ref. [18] to benchmark quantum algorithms on a Rydberg atom array quantum computer.

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 is, to the best of our knowledge, not known before. In the computation, we used finite-field algebra for contracting integer tensor networks with arbitrarily high precision.

A theoretically interesting number that can be computed using the number of independent sets is the entropy constant, which can describe the thermodynamic properties of hard-core lattice gases at the high-temperature limit. For the square lattice graphs, this number 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. They may be used to study phase transitions for disordered lattices, which are typically much harder to understand. Interestingly, the variations due to different random instances are negligible for this quantity.

10.2. The overlap gap property.

With this tool to enumerate or sample 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 to finding the MIS is the so-called overlap gap property [27, 26]. If the overlap gap property is present, it means every two large independent sets either have a significant intersection or a very small intersection; it implies that large independent sets are clustered together. This clustering property has been used to rigorously prove upper bounds on the performance of local search algorithms [27, 26]. To investigate the overlap gap property, we compute pair-wise Hamming distance distributions of large independent sets as they are good indicators of the presence or absence of overlap gap properties. We inspect two types of

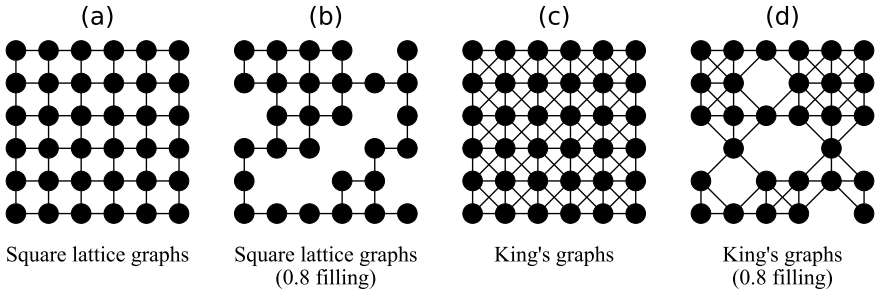


Figure 3: The types of graphs used in the case study in section 10.1. The lattice dimensions are $L \times L$. (a) square lattice graphs. (b) square lattice graphs with a filling factor $p = 0.8$. (c) King's graphs. (d) King's graphs with a filling factor $p = 0.8$.

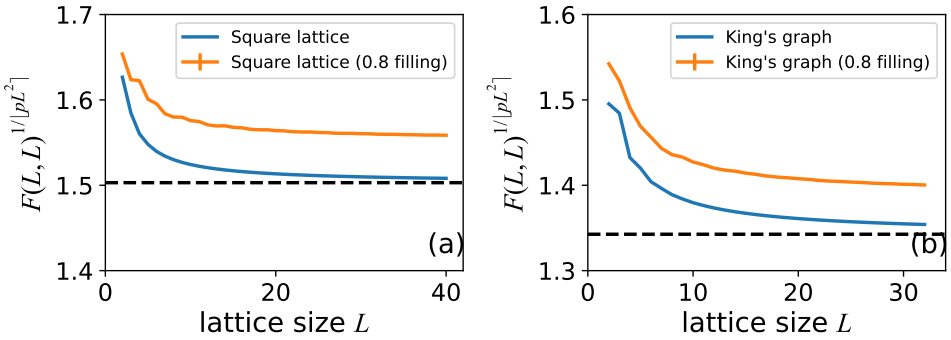
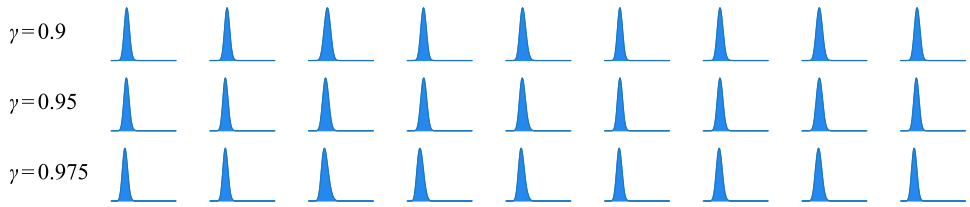
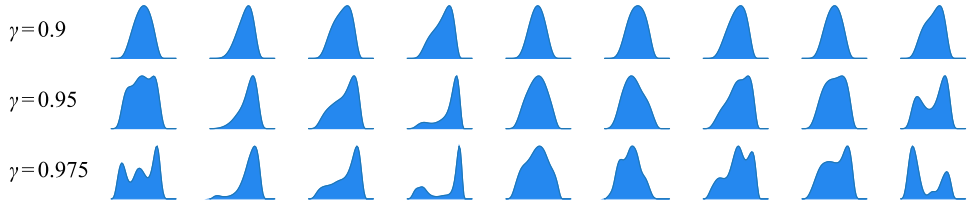


Figure 4: Mean entropy for lattice gases on graphs defined in Fig. 3. We sampled 1000 instances for $p = 0.8$ lattices and the error bar is too small to be visible. The horizontal black dashed lines are for $\lim_{L\to\infty} F(L,L)^{1/L^2}$ for the corresponding non-disordered square lattice and King's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

graphs that are particularly interesting, the King's graphs with defects and 3-regular graphs. It is known that the MIS problem on a general graph can be mapped to the King's graph with defects [28, 18]. However, it is not clear whether the MIS problem defined on a randomly generated King's graph with defects can have the overlap gap property. It is known that finding MISs of a d-regular graphs has the overlap gap property [51, 25] when both d and the graph sizes are large, but, it is not known whether, for small d, e.g. for 3-regular graphs, this statement remains true. We randomly generated 9 instances for each category of King's graph at 0.8 filling with size 20×20 (320 vertices) and 3-regular graphs with size 110. At this problem size, independent sets are too many to fit into any storage, hence we combine the truncated polynomial and sum-product expression tree to directly sample from the target configuration space. For each instance G, we sample 10^4 pairs of configurations from the independent sets of sizes $\geq \lceil \gamma \times \alpha(G) \rceil$ and show the pair-wise Hamming distance distribution in Fig. 5. We observe a clear single peak structure at a fixed distance normalized by the MIS size for the King's graphs, indicating the absence of the overlap gap property in a random King's graph at 0.8 filling. Since the MIS problem on an arbitrary graph can be



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



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

Figure 5: 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.

mapped to a King's graph at a certain filling, this result is highly nontrivial. It likely implies that the King's graphs with defects mapped from hard MIS instances have a very small measure in the total defected King's graph space. In contrast, very different pair-wise Hamming distributions are obtained in Fig. 5(b), where we observed the multiple peak structure when the control parameter γ is big enough. It indicates the existence of disconnected clusters in the configuration space of the MIS problem on 3-regular graphs. We expect this 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. Analyzing quantum and classical algorithms for Maximum Independent Set. In a recent work, the ability to enumerate configurations and compute independence polynomials was critical in understanding the performance of quantum optimization algorithms for the MIS problem on a Rydberg atom quantum computer [18]. This work focused on exploring King's graphs with 0.8 filling. The hardest instances for classical simulated annealing could be accurately predicted from the independence polynomial, which gave information about the density of local minima at different independent set sizes. On the hardest graph instances for simulated annealing studied in the experiment, a high density of local minima were found at independent set sizes of $\alpha(G) - 1$, which the algorithm became trapped in instead of finding the optimal solution of size $\alpha(G)$. By enumerating the configurations using techniques described in the present work, we found that simulated annealing randomly explores the independent sets of size $\alpha(G) - 1$ until an optimum solution is found. Therefore, the large ratio of local to global minima prevents simulated annealing from efficiently finding an MIS.

Although the performance of the quantum algorithm is more challenging to understand due to the inherent difficulty in studying quantum systems, the present methods allow one to gain significant insights by visualizing the experimental outputs of a quantum algorithm

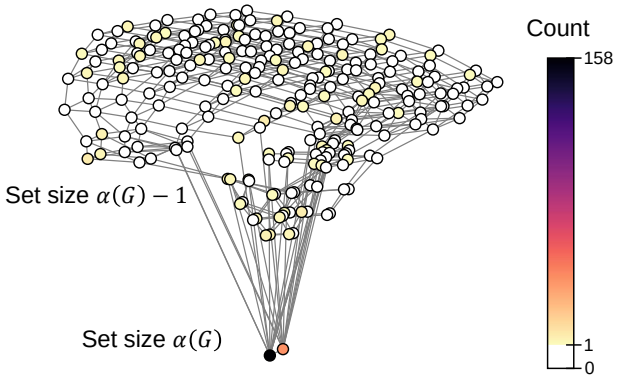


Figure 6: Visualization of experimental outputs of a quantum algorithm for solving the maximum independent set problem [18]. Each vertex represents an independent set, and each edge represents a pair of independent sets that differ by a swap operation or a vertex addition.

over the solution space, as shown in Figure 6 for instances with 39 nodes [18]. Here, the structure of the solution space is shown on a graph where each vertex represents a large independent set. Each edge represents a pair of independent sets that differ by a swap operation or a vertex addition to the set, which are the operations naturally present in the effective dynamics at the end of the quantum algorithm. The solution space graph is well-connected by local changes to the spin configurations, has a small diameter, and the degree of each node appears to concentrate. This visualization makes it clear that the quantum algorithm does not appear to return solely local minima with a large Hamming distance from the MISs as suggested for adiabatic algorithms e.g. by Ref. [4] which would appear as a long path on the solution space graph from the sampled local minima to the MIS. Instead, the quantum algorithm samples from local minima across the solution space graph with a wide range of Hamming distances from the MISs. In the case where a large superposition state of local minima is created during the coherent evolution, the quantum algorithm achieves a quadratic speedup over simulated annealing. Looking forward, we expect these tools can be applied to understanding the performance of quantum and classical algorithms on a wide class of NP-hard combinatorial optimization problems.

11. Discussion and conclusion. In this work, we introduced a framework that uses generic tensor networks to compute different solution space properties of a certain class of NP-hard combinatorial optimization problems. Each solution space property is computed using the same tensor network with different tensor element algebra. The different data types introduced in the main text to compute these properties are summarized in the diagram in Fig. 1. The class of problems solvable by a tensor network includes but is not limited to maximum independent sets and a variety of other combinatorial problems such as the matching problem, the k-coloring problem, the max-cut problem, the set packing problem, and the set covering problem, as detailed in Appendix C.

Looking ahead, it could be possible to generalize the idea of generic programming to other algorithms that have certain algebraic structures such as those using the inclusion-exclusion principle or subset convolution [24] and explore what new properties can be computed. To this end, dynamic programming [16, 24] approaches could be

considered. Dynamic programming is closely related to a tropical tensor network [41]; for example, the Viterbi algorithm for finding the most probable configuration in a hidden Markov model can be interpreted as a matrix product state featured with tropical algebra, and the tropical tensor network in the main text is potentially equivalent to dynamic programming in finding an optimum solution. Since dynamic programming has much broader applications, it would be interesting to extend the ideas from this paper to provide an algebraic interpretation for dynamic programming so that it can be used to compute other solution space properties beyond just finding an optimum solution.

The source code in Julia language for this paper can be found in the Github repository [1]. There is a short introduction to this repository as well a gist to show how it works in Appendix I. We expect our tool can be used to understand and study many interesting applications of independent sets and beyond. We also hope the toolkit we built, including tensor network contraction order optimization and efficient tropical matrix multiplication, can be helpful to the development of other scientific software.

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 writing of this 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. We acknowledge financial support from the DARPA ONISQ program (grant no. W911NF2010021), the Center for Ultracold Atoms, the National Science Foundation, the Vannevar Bush Faculty Fellowship, the U.S. Department of Energy (DE-SC0021013 and DOE Quantum Systems Accelerator Center (contract no. 7568717), the Army Research Office MURI. We acknowledge the computation credits provided by Amazon Web Services for running the benchmarks and case studies. Jinguo Liu acknowledges funding support provided by QuEra Computing Inc. through a sponsored research program.

REFERENCES

[1] https://github.com/QuEraComputing/GenericTensorNetworks.jl.
[2] https://github.com/TensorBFS/TropicalGEMM.jl.
[3] S. Alikhani and Y. hock Peng, *Introduction to domination polynomial of a graph*, 2009, https://arxiv.org/abs/0905.2251.
[4] B. Altshuler, H. Krovi, and J. Roland, *Anderson localization makes adiabatic quantum optimization fail*, Proceedings of the National Academy of Sciences, 107 (2010), pp. 12446–12450, https://doi.org/10.1073/pnas.1002116107, https://www.pnas.org/doi/abs/10.1073/pnas.1002116107, https://arxiv.org/abs/https://www.pnas.org/doi/pdf/10.1073/pnas.1002116107.
[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] J. Fairbanks, M. Besançon, S. Simon, J. Hoffiman, N. Eubank, and S. Karpinski, *Juliagraphs/graphs.jl: an optimized graphs package for the julia programming language*, 2021, https://github.com/JuliaGraphs/Graphs.jl/.

[21] 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.

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

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

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

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

[26] 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.

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

[28] M. R. Garey and D. S. Johnson, *The Rectilinear Steiner Tree Problem is $ NP$ -Complete*, SIAM Journal on Applied Mathematics, 32 (1977), pp. 826–834, https://doi.org/10.1137/0132071, https://doi.org/10.1137/0132071.

[29] 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.

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

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[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] 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.
[58] 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.
[59] 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 with two parts

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

where s_i is a spin on vertex $i \in V$ and w_i is an onsite energy term associated with it. The first part corresponds to the negative independent set size and the second part describes the independence constraint, which corresponds to the Rydberg blockade [50, 18] in cold atom arrays or the repulsive force in hardcore lattice models [17, 21]. The partition function is

defined as

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

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 expressed as a tensor network by placing a vertex tensor on each spin i

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

and an edge tensor on each bond

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

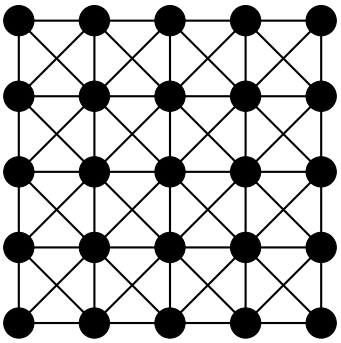
where the 0 in the edge tensor comes from $e^{-\beta\infty}$ in the second term of Eq. (A.1), which is the independence constraint. By letting $x = e^{\beta}$, we get the tensor network for computing the independence polynomial as described by Eq. (4.1) and Eq. (4.2). If we further let $w_i = 1$, the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

Appendix B. An example of increased contraction complexity for the standard tensor network notation.

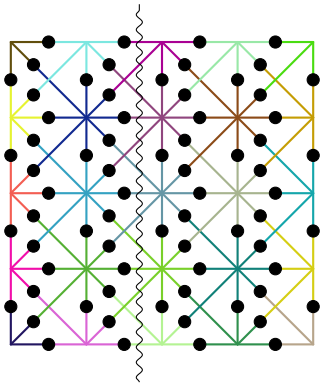
In the standard Einstein's notation for tensor networks in physics, each index appears precisely twice: either both are in input tensors (which will be summed over) or one is in an input tensor and another in the output tensor. Hence a tensor network can be represented as an open simple graph, where an input tensor is mapped to a vertex, a label shared by two input tensors is mapped to an edge and a label that appears in the output tensor is mapped to an open edge. A standard tensor network notation is equivalent to the generalized tensor network in representation power. A generalized tensor network can be converted to a standard one by adding a δ tensors at each hyperedge, 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 example, we will show this conversion might increase the contraction complexity of a tensor network. Let us consider the following King's graph.

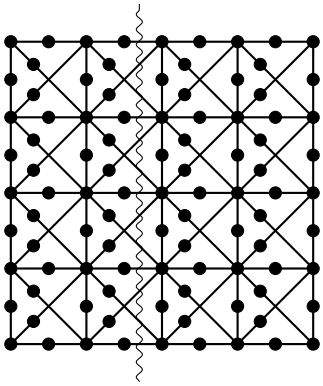


The generalized tensor network for solving the MIS problem on this graph has the following hypergraph representation, where we use different colors to distinguish different hyperedges.



Vertex tensors are not shown here because they can be absorbed into an edge tensor and hence do not change the contraction complexity. If we contract this tensor network in the column-wise order, the maximum intermediate tensor has rank $\sim L$, which can be seen by counting the number of colors at the cut.

By adding δ tensors to hyperedges, we have the standard tensor network represented as the following simple graph.



In this diagram, the additional δ tensors can have ranks up to 8. If we still contract this tensor network in a column-wise order, the maximum intermediate tensor has rank $\sim 3L$, i.e. the space complexity is $\approx 2^{3L}$, which has a larger complexity than using the generalized tensor network notation.

Appendix C. Hard problems and tensor networks.

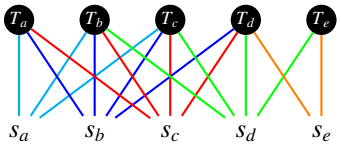
C.1. Maximal independent sets and maximal cliques. In this section, we focus the discussion on the maximal independent sets problem since finding maximal cliques of a graph is equivalent to finding the maximal independent sets of the complement of the graph. Let $G = (V, E)$ be a graph, we denote the neighborhood of a vertex $v \in V$ as $N(v)$. A maximal independent set I_m is an independent set such that no $v \in V$ satisfies $I_m \cap (\{v\} \cup N[v]) = \emptyset$, i.e. an independent set that cannot become a larger one by adding a new vertex. To characterize the maximal independence restriction, we defined a tensor on each $\{v\} \cup N(v)$ as

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

It means if none of the neighbours of v are in I_m ($s_1 = s_2 = \ldots = s_{|N(v)|} = 0$), then v must 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. For a vertex v that has a degree 2, the tensor has the following form

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

Let us consider the graph in Example 3. The corresponding tensor network structure for computing the maximal independent polynomial has the following hypergraph representation.



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. The 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 [29, 43]; to our knowledge, the best algorithm has a time complexity $O(1.3642^{|V|})$ [29].

We show the benchmark of computing the maximal independent set properties on 3-regular graphs in Fig. 7, including a comparison to the Bron-Kerbosch algorithm from Julia package Graphs [20]. Fig. 7(a) shows the space and time complexities of tensor contraction, which are typically larger than those for the independent set problem. In Fig. 7(b), one can see counting maximal independent sets are much more efficient than enumerating them, while our generic tensor network approach runs slightly faster than the Bron-Kerbosch approach in enumerating all maximal independent sets.

C.2. Matching problem. A k-matching in a graph $G = (V, E)$ is a set of k edges that 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 an edge is in the set and 0 otherwise. To characterize the matching constraint, we define a tensor for each $N(v) = \{n_1, n_2, \ldots, n_{d(v)}\}$ as

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

and a tensor for each bond as

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

where a label $\langle v, u\rangle$ is equivalent to $\langle u, v\rangle$. W tensor specifies the constraint that a vertex cannot be shared by two edges in the edge set, and an edge tensor carries the weights. Let

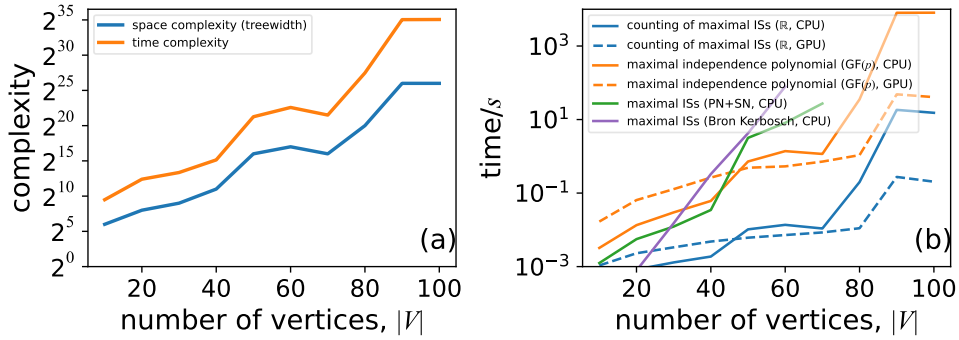


Figure 7: Benchmarks of computing different solution space properties of the maximal independent sets (ISs) problem on random three regular graphs at different sizes. (a) time and space complexity of tensor network contraction. (b) The wall clock time for counting and enumeration of maximal ISs.

$x^{w_{\langle u,v\rangle}}_{\langle u,v\rangle} = x$, the tensor network contraction corresponds to the matching polynomial

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

where k is the size of an edge set, and a coefficient 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 colors is limited to less or equal to k. Let us use the 3-coloring problem as an example to show how to reduce it to tensor contractions. We first map a vertex $v \in V$ to a degree of freedom $c_v \in \{0, 1, 2\}$. Then we define a tensor labeled by c_v for each vertex v as

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

and a bond tensor labelled by (c_u, c_v) for each edge $(u, v) \in E$ as

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

where r_v, g_v and b_v are colors for labeling the configurations. B tensors are for specifying the coloring constraints and W tensors are for labeling the solutions. Let $x^{w_{uv}} = x$ and $r_v = g_v = b_v = 1$; we then have a graph polynomial, in which the k-th coefficient is the number of coloring that k bonds satisfy constraints. 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 an edge coloring problem 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 spin-glass problem in statistical physics.

Let $G = (V, E)$ be a graph. We associate a weight w_v to each $v \in V$. 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 labeled by (s_u, s_v) as

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

where variables $x_u^{w_{uv}}$ and $x_v^{w_{uv}}$ are for a cut on this edge or a domain wall in a spin glass problem. Let $x_u^{w_{uv}} = x_v^{w_{uv}} = x$; we have a graph polynomial similar to the previous ones, in which the kth coefficient is two times the number of cut configurations that have 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, we define a tensor on its closed neighborhood $\{v\} \cup N(v)$ as

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

where w_v is the weight associated with the vertex v. This tensor implies a configuration having a closed neighborhood of v not in D ($s_1 = s_2 = \ldots = s_{|N(v)|} = s_v = 0$) cannot be a dominating 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 [3]

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

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 assignment that satisfies a given Boolean formula. One can specify a satisfiable problem in the conjunctive normal form (CNF), i.e. a conjunction of clauses (or disjunctions of Boolean literals). To reduce the problem of solving a CNF to a tensor network contraction, we first map a Boolean literal a and its negation $\neg a$ to the same degree of freedom $s_a \in \{0, 1\}$. $s_a = 0$ stands for variable a having value `false` while $s_a = 1$ stands for having value `true`. Then we map a clause to a tensor. For example, a k-th clause $\neg a \vee b \vee \neg c$ can be mapped to a tensor labeled by (s_a, s_b, s_c).

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

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

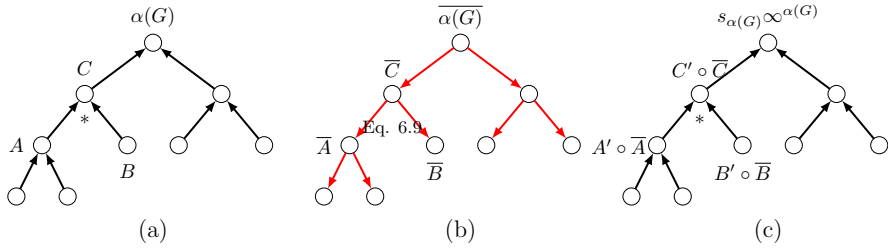


Figure 8: Bounded enumeration of maximum independent sets. Here, a circle is a tensor, an arrow specifies the execution direction of a function, $\overline{A}$ is the Boolean mask for A and $\circ$ is the Hadamard (element-wise) multiplication. (a) is the forward pass with tropical algebra (Eq. (6.3: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

C.7. Set packing. Suppose one has a finite set S and a list of subsets of S. Then, the set packing problem asks if some k subsets in the list are pairwise disjoint. It is the hypergraph generalization of the independent set problem, where a set corresponds to a vertex and an element corresponds to a hyperedge. The generic tensor network for the set packing problem has a similar form as that for the independent set problem, where the vertex tensor is same as Eq. (4.1), while the edge tensor generalizes Eq. (4.2) to higher dimensions

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

C.8. Set covering. Given a collection of elements, the set covering problem aims to find the minimum number of sets that incorporate (cover) all of these elements. In the set covering problem, two sets are given: a set U of elements and a set S of subsets of the set U. The union of all the subsets covers the set U. For each set $s \in S$, we associate it with weight w_s to it. To get the generic tensor network representation, we first map a set $s \in S$ to a Boolean degree of freedom $l_s \in \{0, 1\}$. For each set s, we define a parameterized rank-one tensor indexed by l_s as

$$W(x_s^{w_s}) = \begin{pmatrix} 1 \\ x_s^{w_s} \end{pmatrix} \tag{C.14}$$

where x_s is a variable associated with s. For each unique element $a \in U$, we can define a constraint over all $s \in S$ containing this element, i.e. $N(a) = \{s | s \in S \wedge a \in s\}$, as

$$B_{l_1,l_2,\ldots,l_{|N(a)|}} = \begin{cases} 0 & l_1 = l_2 = \ldots = l_{|N(a)|} = 0, \\ 1 & \text{otherwise.} \end{cases} \tag{C.15}$$

If a subset of S does not include any sets containing element a, then this configuration has zero contribution to the contraction result.

Appendix D. Bounding the MIS enumeration space.

When using the algebra in Eq. (7.3: P1+SN) to enumerate all MISs, the program often stores significantly more intermediate configurations than necessary. To reduce the space overhead, we will show how to bound the searching space using the MIS size $\alpha(G)$. The

bounded contraction consists of three stages 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 and Boolean `false` otherwise. (c) Finally, we perform masked tensor network contraction (i.e. discarding elements masked `false`) using the element type with the algebra in Eq. (7.3: P1+SN) to obtain all MIS configurations. The crucial part is computing the masks in step (b). Note that these masks correspond to tensor elements with non-zero gradients to the MIS size; we can 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 [2] 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)

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

On the right-hand side, we transform the operation into a tropical matrix multiplication so that we can utilize the fast tropical BLAS routines [2]. Again, the equality holds if and only if the element A_{ij} has a contribution to C (i.e. having a non-zero gradient). Let the gradient mask for C be $\overline{C}$; the back-propagation rule for gradient masks reads

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

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:

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

Unlike using the polynomial numbers in Eq. (5.2: PN), the fitting approach does not have the linear overhead in space. However, since the independence polynomial coefficients can have a huge order-of-magnitude range, the round-off errors can be larger than the value itself when using floating-point numbers in computation. To avoid using the arbitrary precision number that can be very slow and is incompatible with GPU devices, we introduce the following finite-field algebra GF(p) approach:

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

Regarding the finite-field algebra, we have the following observations:

1. One can use Gaussian elimination [30] 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 Appendix E, we show that the independence polynomial can be obtained by solving the linear equation Eq. (E.1) using the finite field algebra. One drawback of using finite field algebra is that its matrix multiplication is less computationally efficient compared with floating-point matrix multiplication. Here, we show an alternative method with standard number types but 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

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

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 $\vec{a}_r = \mathrm{FFT}^{-1}(\omega)\cdot\vec{y}$, where $(\vec{a}_r)_j = a_j r^j$. By choosing different r, one can obtain better precision for small j by choosing $r < 1$ or large j 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 the finite-field algebra so that we can reach an arbitrary precision. We also computed the independence polynomial for these lattices up to size 30×30 in our Github repository.

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 $O(n\log(n))$. The standard approach to solve the sum combination problem requires storing the variables in a heap — a highly dynamic binary tree structure that can be much slower to manipulate than arrays. In the following, we show an algorithm with roughly the same complexity but does not need a heap. This algorithm first sorts both A and B and then uses 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. As long as the data range is not exponentially large, the bisection can be

done in $O(\log(n))$ steps, giving the time complexity $O(n\log(n))$. We summarize the algorithm as in Algorithm H.1.

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

In this algorithm, function `collect_geq` is similar the `count_geq` except the counting is replace by collecting the items to a set. Inside the function `count_geq`, variable k monotoneously increase while q monotoneously decrease in each iteration and the total number of iterations is upper bounded by $2n$. Here for simplicity, we do not handle the special element $-\infty$ in A and B and the potential degeneracy in the sums. It is nevertheless important to handle them properly in a practical implementation.

Appendix I. Technical guides.

This appendix covers some technical guides for efficiency, including an introduction to an open-source package GenericTensorNetworks [1] implementing the algorithms in this paper and the gist about how this package is implemented. One can install `GenericTensorNetworks` in a Julia REPL, by first typing `]` to enter the `pkg>` mode and then typing

```
pkg> add GenericTensorNetworks
```

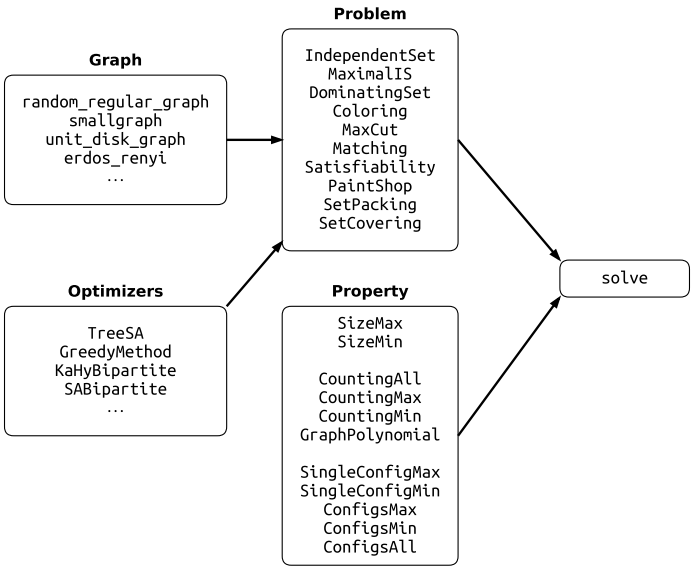
followed by an <ENTER> key. To use it for solving solution space properties, just go back to the normal mode (type <BACKSPACE>) and type

```
julia> using GenericTensorNetworks, Graphs

julia> # using CUDA

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 to use GPU or not. If one wants to use GPU to accelerate the computation, "`using CUDA`" must be uncommented. The problem instance takes four arguments to initialize: the only positional argument is the graph instance one wants to solve, the keyword argument `optimizer` is for specifying the tensor network optimization algorithm, the keyword argument `weights` is for specifying the weights of vertices as either a vector or `NoWeight()`, and the keyword argument `openvertices` is for specifying the degrees of freedom not summed over. Here, we use the `TreeSA` method as the tensor network optimizer, and leave `weights` and `openvertices` as default values. The `TreeSA` algorithm, which was invented in Ref. [36], performs the best in most of our applications. The first execution of this function will be a bit slow due to Julia's just-in-time compilation. After that, the subsequent runs will be faster. The following diagram lists possible combinations of input arguments, where functions in the `Graph` are mainly defined in the package `Graphs`, and the rest can be found in `GenericTensorNetworks`.



The code we will show below is a gist of how the above package was implemented, which is mainly for pedagogical purpose. It covers most of the topics in the paper without caring much about performance. It is worth mentioning that 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 state-of-the-art algorithms for contraction order optimization, which includes the one based on KaHypar+Greedy [31, 48] and the one based on local search [36].

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

Graphs is a foundational package for graph manipulation in the Julia community.

Polynomials is a package providing polynomial algebra and polynomial fitting.

Mods and the Primes package providing finite field algebra and prime number manipulation.

They can be installed in a similar way to `GenericTensorNetworks`. After installing the required packages, one can open a Julia REPL, and copy-paste the following code snippet 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)")
```