SOLVING THE INDEPENDENT SET PROBLEM BY GENERIC PROGRAMMING TENSOR NETWORKS [*]

XXX[†] AND YYY[‡]

Abstract. This paper is about solving the independent set problem by eincoding this problem to a tensor network. We show how to obtain the maximum independent set size, the independence polynomial and optimal configurations of a graph by engineering the tensor element algebra.

Key words. maximum independent set, tensor network

AMS subject classifications. 05C31, 14N07

1. Introduction.

[**JG: some literature does not distinguish maximal and maximum, efficient counting maximal independent sets [22], overlap gap property [12, 11], independence polynomial at -1 [5, 19], treewidth of 3-regular graphs [9]**] In this work, we introduce a tensor based framework to study the famous graph problem of finding independent sets. Given an undirected graph $G = (V, E)$, an independent set $I \subseteq V$ is a set that for any $u, v \in I$, there is no edge connecting u and v in G. The problem of finding the maximum independent set (MIS) size $\alpha(G) \equiv \max_I |I|$ belongs to the complexity class NP-complete [17], which is unlikely to be decided in polynomial time. It is hard to even approximate this size in polynomial time within a factor $|V|^{1-\epsilon}$ for an arbitrarily small positive ϵ. The exhaustive search for a solution costs time $2^{|V|}$. More efficient algorithms to compute the MIS size exactly includes the branching algorithm [31, 28] and dynamic programming. Without changing the fact of exponential scaling in computing time, the branching algorithm gives a smaller base. For example, in [32], a sophisticated branching algorithm has a time complexity $1.1996^n n^{O(1)}$. The dynamic programming approach [6, 10] works better for sparse graphs with small tree width $tw(G)$ [9], it gives an algorithms of complexity $O(2^{tw(G)}tw(G)n)$. People are interested in solving the independent set problem better not only because it is an NP-complete problem that directly related to other NP-complete problems like maximal cliques and vertex cover [24], but also for its close relation with physical applications like hard spheres lattice gas model [7], and Rydberg hamiltonian [27]. However, in these applications, knowing the MIS size and one of the optimum solutions is not the only goal. People often ask different questions about independent sets in order to understand the landscape of their models better. These questions includes but not limited to, counting all independent sets, obtaining all independent sets of size $\alpha(G)$ and $\alpha(G) - 1$, counting independent sets of different sizes, and understanding the effect of a local gadget. In this work, we attack this problem by mapping it to an generic tensor network. It does not give a better time complexity comparing to dynamic programming, but is versatile enough to answer the above questions by engineering the tensor elements with minimum effort.

2. Tensor networks.

A tensor network can be viewed as a generalization to of binary matrix multiplication to n-ary tensor contraction. Let A, B be two matrices, the matrix multiplication is defined as $C_{ik} = \sum_j A_{ij}B_{jk}$. A traditional tensor network refers to the Einstein's notation. In this notation, the matrix multiplication is denoted as $C_i^k = A_i^j B_j^k$, where the paired subscript and superscript j is a dummy index summed over, hence each

[*]

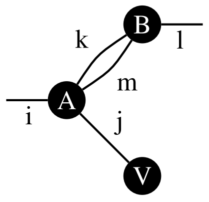
Funding: ...

[†]XXX (email, website).

[‡]yyyyy (yyyy, email).

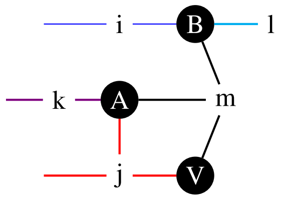
index appears precisely twice. When we have multiple tensors doing the above sum-product operation, we get a traditional tensor network [25]. A traditional tensor network can be represented as a mutigraph with open edges by viewing a tensor as a vertex, a label pairing two tensors as an edge, and the remaining unpaired labels as open edges.

Example 1. A traditional tensor network $C^l_i = A^{km}_{ij} B^l_{km} V^j$ has the following multigraph representation.



Here, we want to use a generalized tensor network notation by not restricting the number of times a label appears in the notation, hence whether an index is a superscript or a subscript makes no sense now. It is also called einsum, sum-product network or factor graph [4] in some contexts. The graphical representation of a tensor network in this paper is a hypergraph, where an edge (label) can be shared by an arbitrary number of vertices (tensors).

Example 2. $C_{ijk} = A_{jkm} B_{mil} V_{jm}$ is a tensor network, it represents $C_{ijk} = \sum_{ml} A_{jkm} B_{mia} V_{jm}$. Its hypergraph representation is as the following, where we use different color to annotate different hyperedges.



In the main text, we stick to the our generalized tensor network notation rather than the traditional notation. As a note to those who are more familiar with the traditional tensor network representation, although one can easily translate a generalized tensor network to the equivalent traditional tensor network by adding δ tensors (a generalization of identity matrix to higher order). It can sometime increase the contraction complexity of a graph. We have an example demonstrating this point in Appendix B.

3. Independence polynomial. One can encode the independence polynomial [18, 8, 16] of G to a tensor network. Independence polynomial is an important graph polynomial that contains the counting information of an independent set problem. It is defined as

$$I(G, x) = \sum_{k=1}^{\alpha(G)} a_k x^k, \tag{3.1}$$

where a_k is the number of independent sets of size k in G, and $\alpha(G)$ is the maximum independent set size. The problem of computing independence polynomial belongs to the complexity class #P-hard. A traditional approach to compute independence polynomial rigorusly requires a computing time $O(1.442^n)$ [8][**JG: I am not sure about this complexity, this is baed on the naive analysis of theorem 2.2 in [8]**]. There are some interests in approximating this polynomial efficiently [15], but here, we focus on the rigorous approaches. We encode this polynomial to a tensor network by placing a rank one tensor of size 2 parametrized by x_i on a vertex i

$$W(x_i)_{s_i} = \begin{pmatrix} 1 \\ x_i \end{pmatrix}_{s_i}, \tag{3.2}$$

and a rank two tensor of size 2×2 on an edge (i, j)

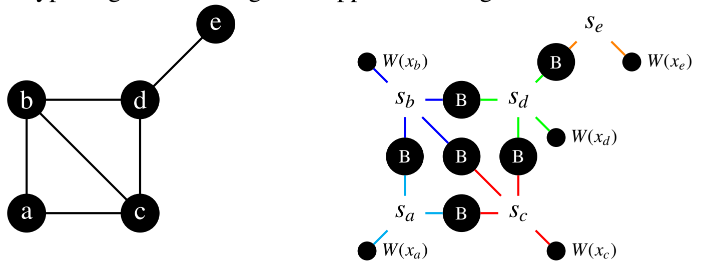
$$B_{s_i s_j} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}_{s_i s_j}, \tag{3.3}$$

where a tensor index s_i is a boolean variable that having the meaning of being 1 if vertex i is in the independent set, 0 otherwise. It corresponds to a hyperedge in the hypergraph. The contraction of such a tensor network gives

$$P(G, \{x_1, \ldots, x_n\}) = \sum_{s_1, s_2, \ldots, s_n = 0}^{1} \prod_{i=1}^{n} W(x_i)_{s_i} \prod_{(i,j) \in E(G)} B_{s_i s_j}, \tag{3.4}$$

where the summation runs over all vertex configurations $\{s_1, \ldots, s_n\}$ and accumulates the product of tensor elements to the scalar output P. We can see an edge tensor represents the restriction on an edge that if both vertices connected by it are included in the set, then such configuration has no contribution to the output. When we set $x_i = x$, the contraction result corresponds to the independence polynomial. One can see the connection from the fact that the product over vertex tensor elements gives a factor x^k, where $k = \sum_i s_i$ counts the set size, and the product over edge tensor elements gives a factor 1 for a configuration being in an independent set, 0 otherwise. One directly benefit of mapping the independent set problem to a tensor network is one can take the advantage of recently developed techniques in tensor network based quantum circuit simulations [14, 26], where people evaluate a tensor network by pairwise contracting tensors in a heuristic order. A good contraction order can reduce the time complexity significantly, at the cost of having a space overhead of $O(2^{tw(G)})$. Here $tw(G)$ is the tree width of the line graph of a tensor network hypergraph, while the line graph of a tensor network hypergraph corresponds to the original graph G that we mapped from. [23] The pairwise tensor contraction also makes it possible to utilize basic linear algebra subprograms (BLAS) functions to speed up our computation for certain tensor element types.

Example 3. Mapping a graph (left) to a tensor network, the resulting tensor network is shown in the right panel. In the generalize tensor network's graphical representation, a vertex is mapped to a hyperedge, and an edge is mapped to an edge tensor.



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

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

Before contracting the tensor network and evaluating the independence polynomial numerically, let us first give up thinking 0s and 1s in tensors $W(x)$ and B as regular computer numbers such as integers and floating point numbers. Instead, we treat them as the additive identity and multiplicative identity of a commutative semiring. A semiring is a ring without additive inverse, while a commutative semiring is a semiring that multiplication commutative. To define a commutative semiring with addition algebra $\oplus$ and multiplication algebra $\odot$ on a set R, the following relations must hold for arbitrary three elements $a, b, c \in R$.

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

The property of being commutative is required here because we want the contraction result independent of the contraction order. In the following, we show how to obtain the independence polynomial, the maximum independent set size and optimal configurations of a general graph G by designing tensor element types as commutative semirings, i.e. making the tensor network generic [30]. A straight forward approach to evaluate the independence polynomial is treating the tensor elements as polynomials, and evaluate the polynomial directly. Let us create a polynomial type, and represent a polynomial $a_0 + a_1 x + \ldots + a_k x^k$ as a vector $(a_0, a_1, \ldots, a_k) \in R^k$, e.g. x is represented as $(0, 1)$. We define the algebra between the polynomials a of order k_a and b of order k_b as

$$\begin{aligned} a \oplus b &= (a_0 + b_0, a_1 + b_1, \ldots, a_{\max(k_a,k_b)} + b_{\max(k_a,k_b)}),\\ a \odot b &= (a_0 + b_0, a_1 b_0 + a_0 b_1, \ldots, a_{k_a} b_{k_b}),\\ \mathbb{0} &= (),\\ \mathbb{1} &= (1). \end{aligned} \tag{3.5}$$

By contracting the tensor network with polynomial type, the final result is the exact representation of the independence polynomial. In the program, the multiplication can be evaluated efficiently with the convolution theorem [29]. However, this approach suffers from a space overhead that proportional to the maximum independent set size because each polynomial requires a vector of such size to store the factors. In Appendix D, we provide a fitting based approach to compute the independence polynomial. One just set x to $\alpha(G) + 1$ random values, compute the result and use the polynomial fitting to get the factors. In this way, we do not have linear overheads in space, however, due to fact that countings of different MIS sizes can be different in many orders, the round off error dominates the high MIS size region that we are most interested about if we use floating point numbers in computation, meanwhile the number easily overflows if we use fixed width integer types. The big integer type is not an option because big integers with varying width can be very slow and incompatible with graphic processing units (GPU) devices. Then we want to resort to integer numbers, however, fixed width integer types are often too small to store the counting, This problem can be solved by introducing finite field algebra $GF(p)$

$$\begin{aligned} x \oplus y &= x + y \pmod p,\\ x \odot y &= xy \pmod p,\\ \mathbb{0} &= 0,\\ \mathbb{1} &= 1. \end{aligned} \tag{3.6}$$

In a finite field algebra, we have the following observations

1. One can use Gaussian elimination [13] to solve a linear equation Eq. (D.2) because it is a generic function 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 unkown 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 developed Algorithm 3.1 to compute independence polynomial exactly without introducing space overheads. In the algorithm, except the computation of Chinese remainder theorem, all computations are done with integers of fixed width W.

3.1. Maximal independence polynomial. Sometimes people are interested in knowing maximal solutions to understand why their programs are trapped in a local minimal. In this paper, the word "maximal" is different with the "maximum" discussed the previous discussion in such a way that it is not necessarily refer to the gloal optimum. Let us denote the neighbour of a vertex v as $N(v)$ and $N[v] = N(v) \cup \{v\}$. A maximal independent set I_m is an independent sets that there does not exist a vertex v that $N[v] \cap I_m = \emptyset$. Let us modify the tensor network for computing independence polynomial by adding this restriction. Instead of defining the restriction on vertices and edges, we define it on $N[v]$

$$T(x_v)_{s_1,s_2,\ldots,s_{|N(v)|},s_v} = \begin{cases} s_v x_v & s_1 = s_2 = \ldots = s_{|N(v)|} = 0,\\ 1 - s_v & \textit{otherwise}. \end{cases} \tag{3.7}$$

Algorithm 3.1 Compute independence polynomial exactly without integer overflow

```
Let $P = 1$, vector $X = (0, 1, 2, \ldots, m)$, matrix $\hat{X}_{ij} = X_i^j$, where $i, j = 0, 1, \ldots m$
while true do
    compute the largest prime $p$ that $\gcd(p, P) = 1$ and $p \leq 2^W$
    compute the tensor contraction on $GF(p)$ and obtain $Y = (y_0, y_1, \ldots, y_m) \pmod p$
    $A_p = (a_0, a_1, \ldots, a_m) \pmod p$ = gaussian_elimination($\hat{X}, Y \pmod p$)
    $A_{P\times p}$ = chinese_remainder($A_P, A_p$)
    if $A_P = A_{P\times p}$ then
        return $A_P$ ; // converged
    end
    $P = P \times p$
end
```

As an example, for a vertex of degree 2, the resulting rank 3 tensor is

$$T(x_v) = \begin{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \\ \begin{pmatrix} x_v & 0 \\ 0 & 0 \end{pmatrix} \end{pmatrix}. \tag{3.8}$$

We do the same computation as independence polynomial, the coefficients of resulting polynomial gives the counting of maximal independent sets, or the maximal independence polynomial. The computational complexity of this new tensor network is often larger than the one for computing independence polynomial. However, in many sparse graphs, this tensor network contraction approach is still much faster than computing the maximal cliques on its complement by applying the Bron-Kerbosch algorithm.

4. Maximum independent sets and its counting problem. In the previous section, we focused on computing independence polynomial for a given maximum independent set size $\alpha(G)$, but we didn't mention how to compute this number. The method we use to compute this quantity is based on the following observations. Let $x = \infty$, the independence polynomial becomes

$$I(G, \infty) = a_k \infty^{\alpha(G)}, \tag{4.1}$$

where the lower orders terms disappear automatically. We can define a new algebra as

$$\begin{aligned} a_x\infty^x \oplus a_y\infty^y &= \begin{cases} (a_x + a_y)\infty^{\max(x,y)}, & x = y \\ a_y\infty^{\max(x,y)}, & x < y \\ a_x\infty^{\max(x,y)}, & x > y \end{cases} \\ a_x\infty^x \odot a_y\infty^y &= a_x a_y \infty^{x+y} \\ \mathbb{0} &= 0\infty^{-\infty} \\ \mathbb{1} &= 1\infty^0 \end{aligned} \tag{4.2}$$

In the program, we only store the power x and the corresponding factor a_x that initialized to 1. This algebra is the same as the one in [20] for counting spin glass ground states. If one is only interested in obtaining $\alpha(G)$, he can drop the factor parts, then the new algebra becomes the max-plus tropical algebra [21, 24].

4.1. Sub-optimal solutions. Some times people are interested in finding sub-optimal solutions efficiently. We define a truncated polynomial algebra by keeping only largest two factors in the polynomial in Eq. (3.5).

$$\begin{aligned} a \oplus b &= (a_{\max(k_a,k_b)-1} + b_{\max(k_a,k_b)-1}, a_{\max(k_a,k_b)} + b_{\max(k_a,k_b)}),\\ a \odot b &= (a_{k_a-1}b_{k_b} + a_{k_a}b_{k_b-1}, a_{k_a}b_{k_b}),\\ \mathbb{0} &= (),\\ \mathbb{1} &= (1). \end{aligned} \tag{4.3}$$

In the program, we need a data structure that contains three fields, the largest order k and factors for two largest orders a_k and a_{k-1}.

5. Enumerating configurations. One may also want to obtain all solutions, it can be achieved replacing the factors with a set of solutions, We design a new element type having the following algebra

$$\begin{aligned} s \oplus t &= s \cup t\\ s \odot t &= \{\sigma \vee^\circ \tau | \sigma \in s, \tau \in t\}\\ \mathbb{0} &= \{\}\\ \mathbb{1} &= \{0^{\otimes n}\} \end{aligned} \tag{5.1}$$

where $\vee^\circ$ is the Hadamard logic or operation over two bit strings, which means joining of two local configurations. The variable x in the vertex tensor is initialized to $x_i = \{\boldsymbol{e}_i\}$, where $\boldsymbol{e}_i$ is a one hot vector of size $|G|$. As an example, if we want to enumerate all maximum independent sets, we can use the above set as the factors in Eq. (4.2). We get the following new algebra.

$$\begin{aligned} s_x\infty^x \oplus s_y\infty^y &= \begin{cases} (s_x \cup s_y)\infty^{\max(x,y)}, & x = y\\ s_y\infty^{\max(x,y)}, & x < y\\ s_x\infty^{\max(x,y)}, & x > y \end{cases},\\ s_x\infty^x \odot s_y\infty^y &= \{\sigma \vee^\circ \tau | \sigma \in s_x, \tau \in s_y\}\infty^{x+y},\\ \mathbb{0} &= \{\}\infty^{-\infty},\\ \mathbb{1} &= \{0^{\otimes n}\}\infty^0, \end{aligned} \tag{5.2}$$

One can easily check if the factor algebra is a commutative semiring, when we use the above algebra as factors of independence polynomials, the resulting algebra is also a commutative semiring. If one is only interested in obtaining a single configuration, one can also just keep a single configuration to save the computational effort. We arrive at a new algebra defined on bit strings.

$$\begin{aligned} \sigma \oplus \tau &= \text{select}(\sigma, \tau)\\ \sigma \odot \tau &= (\sigma \vee^\circ \tau),\\ \mathbb{0} &= 1^{\otimes n},\\ \mathbb{1} &= 0^{\otimes n}, \end{aligned} \tag{5.3}$$

where the `select` function picks one of σ_x and σ_y by some criteria to make the algebra commutative and associative, e.g. by their integer values. In practise, one can just pick randomly from them, then the program will output one of the configurations randomly.

5.1. Bounding the enumeration space. When one uses the set algebra in Eq. (5.1) to represent the factors in Eq. (4.2) for enumerating all optimum configurations, he will find the program stores more than necessary intermediate configurations and cause significant overheads in space. To speed up the computation, we use $\alpha(G)$ to bound the searching space. We first compute the value of $\alpha(G)$ with tropical numbers and cache all intermediate tensors. Then we compute a boolean masks for each cached tensor, where we use a boolean true to represent a tensor element having contribution to the maximum independent set (i.e. with a non-zero gradient) and boolean false otherwise. Finally, we perform masked matrix multiplication using the new element type with the above algebra for obtaining all configurations. Notice that these masks are in fact tensor elements with non-zero gradients with respect to MIS size, we compute these masks by back propagating gradients. To derive the backward rule for tensor contraction, we first reduce the problem to finding the backward rule of a tropical matrix multiplication $C = AB$, where we have the following inequality

$$A_{ij} \odot B_{jk} \leq C_{ik}. \tag{5.4}$$

Here is $\leq$ on tropical numbers are same as regular algebra. The tropical multiplication $\odot$ is the same as the regular $+$, then one can move B_{ik} to the right hand side and get

$$A_{ij} \leq C_{ik} \odot B_{jk}^{\circ -1} \tag{5.5}$$

where the tropical multiplicative inverse is defined as the additive inverse of the regular algebra. The inequality still holds if we take the minimum over k

$$A_{ij} \leq \min_k (C_{ik} \odot B_{jk}^{\circ -1}) = (\oplus_k (C_{ik}^{-1} \odot B_{jk}))^{\circ -1}. \tag{5.6}$$

On the right hand side, we transformed the operation into a tropical matrix multiplication so that we can utilize the fast tropical BLAS routines. The equality holds if and only if element A_{ij} has contribution to C (i.e. has non-zero gradient). Let the gradient mask for C being $\overline{C}$, the backward rule for gradient masks reads

$$\overline{A}_{ij} = \delta(A_{ij}, ((C^{\circ -1} \circ \overline{C})B^T)_{ij}^{\circ -1}), \tag{5.7}$$

where $^{\circ -1}$ is the Hadamard inverse, $\circ$ is the Hadamard product, boolean false is treated as tropical zero and boolean true is treated as tropical one. 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. [] [**JG: maybe add an appendix?**]

6. Benchmarks and case study. We run a sequetial program benchmark on CPU Intel(R) Core(TM) i5-10400 CPU @ 2.90GHz, and show the results bellow. Tensor network contraction is parallelizable. When the element type is immutable, one can just upload the data to GPU to enjoy the speed up.

7. Discussion. We introduced in the main text how to compute the independence polynomial, maximum independent set and optimal configurations, derived the backward rule for tropical tensor network to bound the search of solution space. Although many of these properties are global, we can encode it to different tensor element types as commutative semirings. The power of tensor network's is not limited to the indenepent set problem, in Appendix C we show how to map matching problem and k-coloring to a tensor network. Here, we want to discuss more from the programming perspective. We show some of the Julia language [3] implementations in Appendix A, you will find it being surprisingly short. What we need to do is just defining two operations $\oplus$ and $\odot$ and two special elements $\mathbb{0}$ and $\mathbb{1}$. The style that we program is called generic programming, meaning one can feed

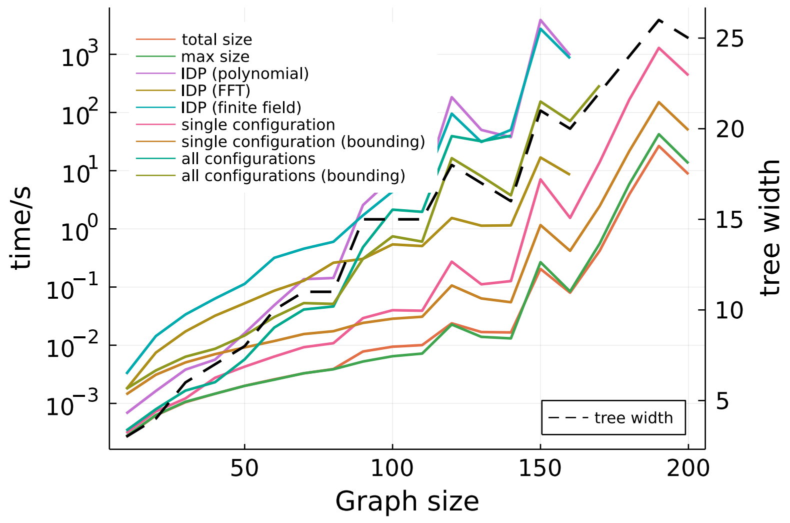


Figure 1: Benchmark results for computing different properties with different element types. The right axis is only for the dashed line.

different data types into a same program, and the program will compute the result with a proper performance. In C++, users can use templates for such a purpose. We chose Julia because its just in time compiling is very powerful that it can generate fast code dynamically for users. Elements of fixed size, such as the finite field algebra, truncated polynomial, tropical number and tropical number with counting or configuration field used in the main text can be inlined in an array. Furthermore, these inlined arrays can be upload to GPU devices for faster generic matrix multiplication implemented in CUDA.jl [2].

Acknowledgments. We would like to acknowledge Sepehr Ebadi, Maddie Cain and Leo Zhou for popping up inspiring questions, their questions are the driving force of this project. Thank Chris Elord for helping us writting the fastest matrix multiplication libary for GEMM, TropicalGEMM.jl, he is a remarkable guy! [**JG: funding information**]

REFERENCES

[1] S. F. Barr, *Courcelle's Theorem: Overview and Applications*, PhD thesis, Oberlin College, 2020.

[2] T. Besard, C. Foket, and B. D. Sutter, *Effective extensible programming: Unleashing julia on gpus*, CoRR, abs/1712.03112 (2017), http://arxiv.org/abs/1712.03112, https://arxiv.org/abs/1712.03112.

[3] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman, *Julia: A fast dynamic language for technical computing*, 2012, https://arxiv.org/abs/1209.5145, https://arxiv.org/abs/1209.5145.

[4] C. M. Bishop, *Pattern Recognition and Machine Learning*, Springer, 2006.

[5] M. Bousquet-Mélou, S. Linusson, and E. Nevo, *On the independence complex of square grids*, Journal of Algebraic combinatorics, 27 (2008), pp. 423–450.

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

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

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

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

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

[11] D. Gamarnik and A. Jagannath, *The overlap gap property and approximate message passing algorithms for*

element type	**purpose**
regular number	counting all indenepent sets
tropical number (Eq. (4.2))	finding the maximum independent set size
tropical number with counting (Eq. (4.2))	finding both the maximum independent set size and its degeneracy
tropical number with configurations (Eq. (5.3))	finding the maximum independent set size and one of the optimal configurations
tropical number with sets (Eq. (5.1))	finding the maximum independent set size and all optimal configurations
polynomial (Eq. (3.5))	computing the indenpendence polynomials exactly
truncated polynomial (Eq. (4.3))	counting the suboptimal independent sets
complex number	fitting the indenpendence polynomials with fast fourier transformation
finite field algebra Eq. (3.6)	fitting the indenpendence polynomials exactly using number theory

Table 1: Tensor element types used in the main text and their purposes.

p-spin models, 2019, https://arxiv.org/abs/1911.06943.

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

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

[14] 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.

[15] 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.

[16] N. J. A. Harvey, P. Srivastava, and J. Vondrák, *Computing the independence polynomial: from the tree threshold down to the roots*, 2017, https://arxiv.org/abs/1608.02282.

[17] J. Hastad, *Clique is hard to approximate within n/sup 1-/spl epsiv*, in Proceedings of 37th Conference on Foundations of Computer Science, IEEE, 1996, pp. 627–636.

[18] V. E. Levit and E. Mandrescu, *The independence polynomial of a graph-a survey*, in Proceedings of the 1st International Conference on Algebraic Informatics, vol. 233254, Aristotle Univ. Thessaloniki Thessaloniki, 2005, pp. 231–252.

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

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] C. Moore and S. Mertens, *The nature of computation*, OUP Oxford, 2011.
[25] R. Orús, *A practical introduction to tensor networks: Matrix product states and projected entangled pair states*, Annals of Physics, 349 (2014), pp. 117–158.
[26] F. Pan and P. Zhang, *Simulating the sycamore quantum supremacy circuits*, 2021, https://arxiv.org/abs/2103.03074.
[27] 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).
[28] J. M. Robson, *Algorithms for maximum independent sets*, Journal of Algorithms, 7 (1986), pp. 425–440.
[29] A. Schönhage and V. Strassen, *Schnelle multiplikation grosser zahlen*, Computing, 7 (1971), pp. 281–292.
[30] A. A. Stepanov and D. E. Rose, *From mathematics to generic programming*, Pearson Education, 2014.
[31] R. E. Tarjan and A. E. Trojanowski, *Finding a maximum independent set*, SIAM Journal on Computing, 6 (1977), pp. 537–546.
[32] M. Xiao and H. Nagamochi, *Exact algorithms for maximum independent set*, Information and Computation, 255 (2017), p. 126–146, https://doi.org/10.1016/j.ic.2017.06.001, http://dx.doi.org/10.1016/j.ic.2017.06.001.

Appendix A. Technical guide.

OMEinsum a package for the `einsum` function,
OMEinsumContractionOrders a package for finding the optimal contraction order for the `einsum` function
https://github.com/Happy-Diode/OMEinsumContractionOrders.jl,
TropicalGEMM a package for efficient tropical matrix multiplication (compatible with OMEinsum),
TropicalNumbers a package providing tropical number types and tropical algebra, one o the dependency of TropicalGEMM,
LightGraphs a package providing graph utilities, like random regular graph generator,
Polynomials a package providing polynomial algebra and polynomial fitting,
Mods and Primes packages providing finite field algebra and prime number generators.

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

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

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

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

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

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

# an einsum contraction without contraction order specified is called `EinCode`,
# an einsum contraction has contraction order (specified as a tree structure) is called `
     NestedEinsum`.
# assign each label a dimension-2, it will be used in contraction order optimization
# `symbols` function extracts tensor labels into a vector.
symbols(::EinCode{ixs}) where ixs = unique(Iterators.flatten(filter(x->length(x)==1,ixs)))
symbols(ne::OMEinsum.NestedEinsum) = symbols(flatten(ne))
```

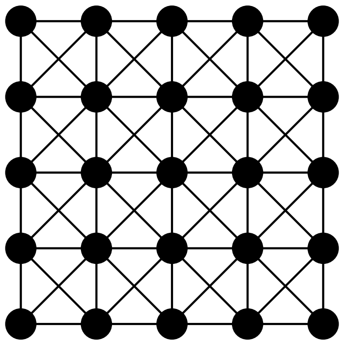
```julia
size_dict = Dict([s=>2 for s in symbols(code)])
# optimize the contraction order using KaHyPar + Greedy, target space complexity is 2^17
optimized_code = optimize_kahypar(code, size_dict; sc_target=17, max_group_size=40)
println("time/space complexity is $(OMEinsum.timespace_complexity(optimized_code, size_dict))")

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

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

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

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

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

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

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

function independence_polynomial_finitefield(code; mis_size=Int(mis_size(code)[].n), max_order=1
      00)
    N = typemax(Int32) # Int32 is faster than Int.
    YS = []
    local res
    for k = 1:max_order
       N = Primes.prevprime(N-one(N))  # previous prime number
        # evaluate the polynomial on a finite field algebra of modulus `N`
        rk = _independance_polynomial(Mods.Mod{N,Int32}, code, mis_size)
        push!(YS, rk)
        if max_order==1
            return Polynomial(Mods.value.(YS[1]))
        elseif k != 1
            ra = improved_counting(YS[1:end-1])
            res = improved_counting(YS)
            ra == res && return Polynomial(res)
        end
    end
    @warn "result is potentially inconsistent."
    return Polynomial(res)
end
function _independance_polynomial(::Type{T}, code, mis_size::Int) where T
   xs = 0:mis_size
```

```julia
    ys = [independence_polynomial(T(x), code)[] for x in xs]
    A = zeros(T, mis_size+1, mis_size+1)
    for j=1:mis_size+1, i=1:mis_size+1
        A[j,i] = T(xs[j])^(i-1)
    end
    A \ T.(ys)  # gaussian elimination to compute ``A^{-1} y```
end
improved_counting(sequences) = map(yi->Mods.CRT(yi...), zip(sequences...))

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

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

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

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

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

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

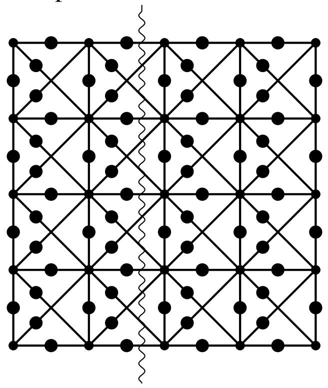
In the above examples, the configuration enumeration is very slow, one should use the optimal MIS size for bounding as decribed in the main text. We will not show any example about implementing the backward rule here because it has approximately 100 lines of code. Please checkout our GitHub repository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

Appendix B. Why not introducing δ tensors.

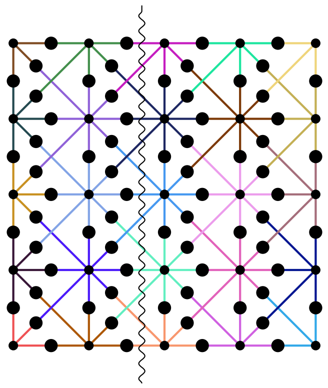
Given a graph



Its traditional tensor network representation with δ tensors is



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



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

Appendix C. Generalizing to other graph problems. There are some other graph problems that can be encoded in a tensor network. To understand its representation power, it is a good starting point to connect it with dynamic programming because a tensor network can be viewed as a special type of dynamic programming where its update rule can be characterized by a linear operation. Courcelle's theorem [6, 1] states that a problem quantified by monadic second order logic (MSO) on a graph with bounded tree width k can be solved in linear time with respect to the graph size. Dynamic programming is a traditional approach to attack a MSO problem, it can solve the maximum independent set problem in $O(2^k)n$, which is similar to the tensor network approach. We mentioned in the main text that tensor network has nice analytic property make it easier for generic programming. The cost is, the tensor network is less expressive than dynamic programming, However, that are still some other problems that can be expressed in the framework of generic tensor network.

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

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

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

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

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

and a tensor of rank 1 on the bond

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

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

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

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

and an edge tensor as

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

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

Appendix D. The fitting and Fourier transformation approaches to computing independence polynomial. Let $m = \alpha(G)$ be the maximum independent set size and X be a set of real numbers of cardinality $m + 1$. We compute the tensor network contraction for each $x_i \in X$ and obtain the following relations

$$\begin{aligned} a_0 + a_1 x_1 + a_1 x_1^2 + \ldots + a_m x_1^m &= y_0 \\ a_0 + a_1 x_2 + a_2 x_2^2 + \ldots + a_m x_2^m &= y_1 \\ &\ldots \\ a_0 + a_1 x_m + a_2 x_m^2 + \ldots + a_m x_m^m &= y_m \end{aligned} \tag{D.1}$$

The polynomial fitting between X and $Y = \{y_0, y_1, \ldots, y_m\}$ gives us the factors. The polynomial fitting is essentially about solving the following linear equation

$$\begin{pmatrix} 1 & x_1 & x_1^2 & \ldots & x_1^m \\ 1 & x_2 & x_2^2 & \ldots & x_2^m \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_m & x_m^2 & \ldots & x_m^m \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_m \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_m \end{pmatrix}. \tag{D.2}$$

In practise, the fitting can suffer from the non-negligible round off errors of floating point operations and produce unreliable results. This is because the factors of independence polynomial can be different in magnitude by many orders. Instead of choosing X as a set of random real numbers, we make it form a geometric sequence in the complex domain $x_j = r\omega^j$, where $r \in \mathbb{R}$ and $\omega = e^{-2\pi i/(m+1)}$. The above linear equation becomes

$$\begin{pmatrix} 1 & r\omega & r^2\omega^2 & \ldots & r^m\omega^m \\ 1 & r\omega^2 & r^2\omega^4 & \ldots & r^m\omega^{2m} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & r\omega^m & r^2\omega^{2m} & \ldots & r^m\omega^{m^2} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_m \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_m \end{pmatrix}. \tag{D.3}$$

Let us rearrange the factors r^j to a_j, the matrix on left side is exactly the a discrete Fourier transformation (DFT) matrix. Then we can obtain the factors using the inverse Fourier transformation $\vec{a}_r = \text{FFT}^{-1}(\omega) \cdot \vec{y}$, where $(\vec{a}_r)_j = a_j r^j$. By choosing different r, one can obtain better precision in low independent set size region ($\omega < 1$) and high independent set size region ($\omega > 1$).