SOLVING THE MAXIMUM INDEPENDANT SET PROBLEM BY GENERIC PROGRAMMING EINSUM NETWORKS [*]

XXX[†] AND YYY[‡]

Abstract. Solving the maximum independent set size problem by mapping the graph to an einsum 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. We also show how to analyse the local properties of a graph by contracting an open einsum network.

Key words. maximum independent set, einsum network

AMS subject classifications. 05C31, 14N07

1. Introduction.

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. Finding the maximum independent set (MIS) size $\alpha(G) \equiv \max_I |I|$ belongs to the complexity class NP-complete [10], 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 arbituarily small possitive ϵ. The naive algorithm of enumerating all configuration space gives a $2^{|V|}$ time solution. More efficient algorithms to compute the MIS size exactly includes the branching algorithm and dynamic programming. Without changing the fact of exponential scaling in computing time, the branching algorithm gives a smaller base. For example, in [20], a sophisticated branching algorithm gives a time complexity $1.1893^n n^{O(1)}$. The dynamic programming approach [3, 6] works better for graphs with small tree width $tw(G)$, 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 a NP-complete problem that directly related to other NP-complete prolems like maximal cliques and vertex cover [15], but also for its close relation with physical applications like hard spheres lattice gas model [4], and Rydberg hamiltonian [18]. However, in these applications, knowing the MIS size and one of the optimal solution 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 indenepent sets of size $\alpha(G)$ and $\alpha(G) - 1$, counting the number of (maximal) 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 "einsum" 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. Einsum network.

Einstein's notation is originally proposed 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 $C_{ik} = \sum_j A_{ij} B_{jk}$. In Einstein's notation, it 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 index appears precisely twice. When we have multiple tensors doing the above sum-product operation, we get a tensor network [16]. A tensor network has a nice a muti-graph with open edges. We view a tensor on the right hand side as a vertex in a graph, a label

[*]

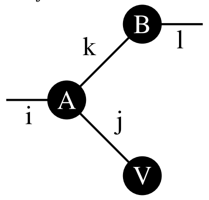
Funding: ...

[†]XXX (email, website).

[‡]yyyyy (yyyy, email).

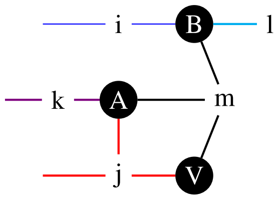
pairing two tensors as an edge, and the remaining unpaired labels as open edges.

Example 1. A tensor networks $C^l_i = A^k_{ij} B^l_k V^j$ has the following graphical representation.



Einsum network is a generalization of tensor network by not restricting the number of times a label appears, hence whether an index is a superscript or a subscript makes no sense now. It is also called sum-product network or factor graph [2] in some contexts. The graphical representation of an einsum is a hypergraph, where an edge (label) can be shared by an arbituary number of vertices (tensors).

Example 2. $C_{ijk} = A_{jkm} B_{mil} V_{jm}$ is an einsum 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 einsum notation rather than the tensor network notation. As a note to those who are more familiar with tensor network representation, although one can easily translate an einsum network to the equivalent 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 in Appendix B.

3. Independence polynomial. One can encode the independent set problem on graph G to an einsum network by placing a rank one tensor of size 2 on vertex i

$$W(x_i)_{s_i} = \begin{pmatrix} 1 \\ x_i \end{pmatrix}_{s_i}, \tag{3.1}$$

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

$$B_{s_i s_j} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}_{s_i s_j}, \tag{3.2}$$

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

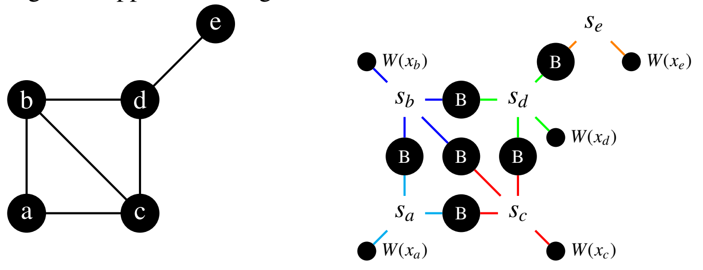
$$A(G, \{x_1, \ldots, x_n\}) = \sum_{s_1, s_2, \ldots, s_n = 0}^{1} \prod_{i=1}^{n} W(x_i)_{s_i} \prod_{(i,j) \in E(G)} B_{s_i s_j}. \tag{3.3}$$

Here, the einsum runs over all vertex configurations $\{s_1, \ldots, s_n\}$ and accumulates the product of tensor elements to the scalar output. Let $x_i = x$, then the product over vertex tensors gives a factor x^k, where $k = \sum_i s_i$ is the vertex set size, and the product over edge tensors gives a factor 0 for configurations not being an independent set. The contraction of this einsum network gives the independence polynomial [5, 9] of G

$$I(G, x) = \sum_{k=1}^{\alpha(G)} a_k x^k, \tag{3.4}$$

where a_k is the number of independent sets of size k in G, and $\alpha(G)$ is the maximum independent set size. The benefit of mapping the independent set problem to the einsum network is one can take the advantage of recently developed techiniques in tensor network based quantum circuit simulations [8, 17], 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 $O(2^{tw(G)})$, where $tw(G)$ is the treewidth of the line graph of a tensor network, here it corresponds to the original graph G that we mapped from. [14] The pairwise tensor contraction also makes it possible to utilize fast basic linear algebra subprograms (BLAS) functions for certain tensor element types.

Example 3. Mapping a graph (left) to an einsum network, the resulting einsum network is shown in the right panel. A vertex is mapped to a hyperedge in the einsum's graphical notation. An edge is mapped to an edge tensor.



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

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

Before contracting the einsum 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 arbituary three elements

$a, b, c \in R$.

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

In the following, 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 einsum network generic [19].

3.1. The polynomial approach. 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 einsum 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. The only problem of this method is it suffers from a space overhead that propotional to the maximum independant set size because each polynomial requires a vector of such size to store the factors. In the following subsections, we managed to solve this problem.

3.2. The fitting and Fourier transformation approaches. 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 einsum contraction for each $x_i \in X$ and obtain the following relations

$$\begin{aligned}
a_0 + a_1 x_1 + a_1 x_1^2 + \ldots + a_m x_1^m &= y_0\\
a_0 + a_1 x_2 + a_2 x_2^2 + \ldots + a_m x_2^m &= y_1\\
&\ldots\\
a_0 + a_1 x_m + a_2 x_m^2 + \ldots + a_m x_m^m &= y_m
\end{aligned} \tag{3.6}$$

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

$$\begin{pmatrix} 1 & x_1 & x_1^2 & \ldots & x_1^m \\ 1 & x_2 & x_2^2 & \ldots & x_2^m \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_m & x_m^2 & \ldots & x_m^m \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_m \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_m \end{pmatrix}. \tag{3.7}$$

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

$$\begin{pmatrix} 1 & r\omega & r^2\omega^2 & \ldots & r^m\omega^m \\ 1 & r\omega^2 & r^2\omega^4 & \ldots & r^m\omega^{2m} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & r\omega^m & r^2\omega^{2m} & \ldots & r^m\omega^{m^2} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_m \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_m \end{pmatrix}. \tag{3.8}$$

Let us rearange the factors r^j to a_j, the matrix on left side is exactly the a descrete 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 diferent r, one can obtain better precision in low independant set size region ($\omega < 1$) and high independant set size region ($\omega > 1$).

3.3. The finite field algebra approach. It sounds a bit over ambitious to compute the independence polynomial regorously using integer number types only, because the fixed width integer types are often too small to store the countings, while big integer with varying width can be very slow and imcompatible with graphic processing units (GPU) devices. This problem will be solved if we computate on a finite field algebra $GF(p)$

$$\begin{aligned} x \oplus y &= x + y \pmod{p}, \\ x \odot y &= xy \pmod{p}, \\ \mathbb{0} &= 0, \\ \mathbb{1} &= 1. \end{aligned} \tag{3.9}$$

In a finite field algebra, we have the following observations

1. One can still use Gaussian elimination [7] to solve a linear equation Eq. (3.7). This is because a field has the property that the multiplicative inverse exists for any non-zero value. The multiplicative inverse here can be computed with the extended Euclidean algorithm.
2. Given the remainders of a larger integer x over a set of coprime 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 even though its bit width is larger than the register size.

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.

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 $\gcd(p, P) = 1 \wedge p \leq 2^W$
    compute the tensor network 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
```

3.4. Maximal independence polynomial. Let us denote the neighbor 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 einsum network for computing independence polynomial to count maximal independent sets. We define a tensor on $N[v]$ to capture this property

$$T(x)_{s_1,s_2,\ldots,s_{|N(v)|},s_v} = \begin{cases} s_v x & s_1 = s_2 = \ldots = s_{|N(v)|} = 0, \\ 1 - s_v & otherwise. \end{cases} \tag{3.10}$$

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

$$T(x) = \begin{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \\ \begin{pmatrix} x & 0 \\ 0 & 0 \end{pmatrix} \end{pmatrix}. \tag{3.11}$$

We do the same computation as independence polynomial, the coefficients of resulting polynomial gives the counting of maximal independent sets, or the maximal independence polynoimal. The treewidth 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 mentioned how to compute 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 [12] for counting spin glass ground states. If one is only interested in obtaining $\alpha(G)$, he can drop the factor parts, then the algebra of x becomes the max-plus tropical algebra [13, 15].

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

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

By changing the factors to sets, and plus and multiplication operations on factors to set union and product, one can get all suboptimal solutions too.

5. Enumerating configurations. One may also want to obtain all solutions, it can be achieved replacing the factors a_x with a set of bit strings s_x, We design a new element type that having 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 e_i is a one hot vector of size $|G|$. One can easily check this algebra is a commutative semiring. When we use the above algebra as factors in Eq. (4.2), the resulting algebra is also a commutative semiring. With this new element type, the einsum network contraction will give all solutions rather than just a number for counting. By slightly modifying the above algebra, it can also be used to obtain just a single configuration to save the computational effort.

$$
\begin{aligned}
\sigma \oplus \tau &= \mathrm{select}(\sigma, \tau)\\
\sigma \odot \tau &= (\sigma \vee^\circ \tau),\\
\mathbb{0} &= 1^{\otimes n},\\
\mathbb{1} &= 0^{\otimes n},
\end{aligned} \tag{5.2}
$$

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 optimal configurations randomly.

5.1. Bounding the enumeration space. If one implements the above algebra for enumerating configurations naively, he will find the program stores more than nessesary intermedite configurations and cause significant overheads in space. To speed up the computation, we use $\alpha(G)$ to bound the search 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 nonzero 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 nonzero gradients with respect to MIS size, we compute these masks by back propagating gradients. To derive the backward rule, we consider a tropical matrix multiplication $C = AB$, we have the following inequality

$$A_{ij} \odot B_{jk} \leq C_{ik}. \tag{5.3}$$

Moving B_{ik} to the right hand side, we have

$$A_{ij} \leq (\oplus_k (C_{ik}^{-1} \odot B_{jk}))^{-1} \tag{5.4}$$

where the tropical multiplicative inverse is defined as the additive inverse of the regular algebra. The equality holds if and only if element A_{ij} contributions to C (i.e. has nonzero gradient). Let the mask for C being $\overline{C}$, the backward rule for "gradient" masks reads

$$\overline{A}_{ij} = \delta(A_{ij}, ((C^{\circ -1} \circ \overline{C})B^T)_{ij}^{\circ -1}), \tag{5.5}$$

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 the einsum of two tensors by replacing the matrix multiplication between $C^{\circ -1} \circ \overline{C}$ and B^T by an einsum.

6. Tropical tensors for automated branching. Branching rules can be automatically discovered by contracting the tropical einsum network for a subgraph $R \subseteq G$. Let us denote the resulting tropical tensor of rank $|C|$ as A, where C is the set of boundary vertices defined as $C := \{c | c \in R \wedge c \in G \backslash R\}$ and $|C|$ the size of C. Each tensor entry A_σ is a local maximum independant set size with a fixed boundary configuration $\sigma \in \{0, 1\}^{|C|}$ by marginalizing the inner degrees of freedom. If we are only interested in finding a single maximum independent set rather than enumerating all possible solutions, this tensor can be further "compresed" by setting some entries to tropical zero. Let us define a relation of *less restrictive* as

$$(\sigma_a \prec \sigma_b) := (\sigma_a \neq \sigma_b) \wedge (\sigma_a \leq^\circ \sigma_b) \tag{6.1}$$

where $\leq^\circ$ is the Hadamard less or equal to operation.

Definition 6.1. *A tensors A is MIS-compact if are no two nonzero entries of it that one is "better" than another, where an entry A_{σ_a} is "better" than A_{σ_b} if*

$$(\sigma_a \prec \sigma_b) \wedge (A_{\sigma_a} \geq A_{\sigma_b}). \tag{6.2}$$

If we remove such A_{σ_b}, the contraction over the whole graph is guaranted to give the same maximum independant set size. It can be seen by considering two entries with the same local maximum independent set sizes and different boundary configurations as shown in Fig. 1 (a) and (b). If we have $\sigma_b \cup \overline{\sigma_b}$ being one of the solutions for maximum independant sets in G, then $\sigma_a \cup \overline{\sigma_b}$ is another solution giving the same $\alpha(G)$. Hence, we can set A_{σ_b} to tropical zero safely.

Theorem 6.2. *A MIS-compact tropical tensor is optimal, i.e. any of its nonzero entries can produce the only global optimal solution given a proper environment.*

Proof. Let use prove it by showing $\forall \sigma$ in a MIS-compact tropical tensor for a subgraph R, there exists a graph G that $R \subseteq G$ and σ is the only boundary configuration that produces

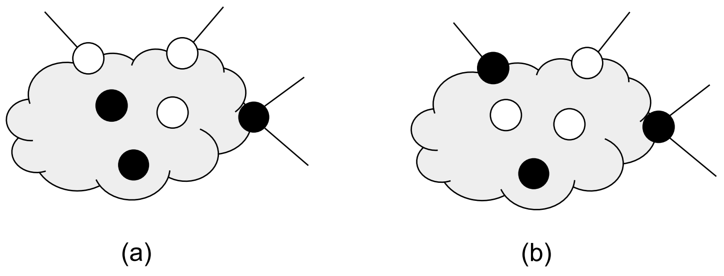
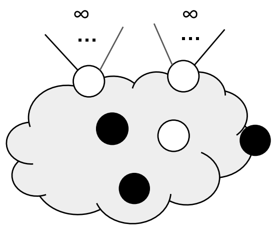


Figure 1: Two configurations with the same local independent size $A_{\sigma_a} = A_{\sigma_b} = 3$ and different boundary configurations (a) $\sigma_a = \{001\}$ and (b) $\sigma_b = \{101\}$, where black nodes are 1s (in the independent set) and white nodes are 0s (not in the independent set).

the maximum independent set. i.e. no tensor entry can be removed without knowledge about $G\backslash R$. Let A be a tropical tensor, and an entry of it being A_σ, where σ is the bounary configuration. Let us construct a graph G such that for a vertex $v \in C$, if $\sigma_v = 1$, $\alpha(N[v] \cap (G\backslash R)) = 0$, otherwise, $\alpha(N[v] \cap (G\backslash R)) = \infty$, meanwhile, for any $v, w \in C$, $N[v] \cap N[w] = \emptyset$. The simplest construction is connecting vertices that $\sigma_v = 0$ with infinite many mutually disconnected vertices as illustrated in the following graph.

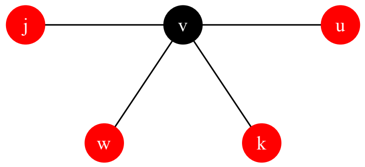


Then we have the maximum independent set size with boundary configuration σ being $\alpha(G, \sigma) = \infty(|C|-|\sigma|)+A_\sigma$, where $|\sigma|$ is defined as the number of 1s in σ. Let us assume there exists another configuration τ that generating the same or even better maximum independent set size $\alpha(G, \tau) \geq \alpha(G, \sigma)$. Then we have $\tau \prec \sigma$, otherwise it will suffer from infinite punishment from $G\backslash R$. For such a τ, we have $A_\tau < A_\sigma$, otherwise $A_\sigma \prec A_\tau$ contradicts with A being MIS-compact. Finally, we have $\alpha(G, \tau) = \infty(|C|-|\sigma|)+A_\tau < \alpha(G, \sigma)$, which contradicts with our preassumtion. Such τ does not exist and σ is the only boundary configuration that $\alpha(G) = \alpha(G, \sigma)$. ∎

6.1. The tensor network compactifying detects branching rules automatically. Almost all branching rules are based on the same idea of analysing a local subgraph induced by a vertex v by including its neighborhoods, and keep only the configurations that has the potential to produce the only maximum independent sets. Since an MIS-compact tensor is optimal, by analysing the correlation of vertex configurations on the resulting tensor for $N^3[v]$, one can discover the optimal branching vector automatically.

Corollary 6.3. *If a vertex v is in an independent set I, then none of its neighbors can be in I. On the other hand, if I is a maximum (and thus maximal) independent set, and thus if v is not in I then at least one of its neighbors is in I.*

Contract $N[v]$ and the resulting tensor A has a rank $|N(v)|$. Each tensor entry A_σ corresponds to a locally maximized independant set size with fixed boundary configuration $\sigma \in \{0,1\}^{|N(v)|}$. If the boundary configuration is a bit string of 0s, σ_v will takes value 1 to maximize the local independant set size.



After contracting $N[v]$, v becomes an internal degree of freedom. Applying tensor compactifying rule Eq. (6.2), the resulting rank 4 tropical tensor is

$$T_{juwk} = \begin{pmatrix} \begin{pmatrix} 1 & -\infty \\ -\infty & 2 \end{pmatrix}_{ju} & \begin{pmatrix} -\infty & 2 \\ 2 & 3 \end{pmatrix}_{ju} \\ \begin{pmatrix} -\infty & 2 \\ 2 & 3 \end{pmatrix}_{ju} & \begin{pmatrix} 2 & 3 \\ 3 & 4 \end{pmatrix}_{ju} \end{pmatrix}_{wk}. \tag{6.3}$$

If we use sets for counting, one can check all configurations too. By studying the correlation between vertex variables, one can easily see x_v does not co-exist with other vertex variables. These anti-correlation determines possible branching vectors in the maximum independent set problem.

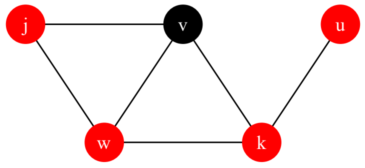
$$S_{juwk} = \begin{pmatrix} \begin{pmatrix} 1+x_v & - \\ - & x_jx_u \end{pmatrix}_{ju} & \begin{pmatrix} - & x_ux_k \\ x_jx_k & x_ux_jx_k \end{pmatrix}_{ju} \\ \begin{pmatrix} - & x_wx_u \\ x_wx_j & x_wx_jx_u \end{pmatrix}_{ju} & \begin{pmatrix} x_wx_k & x_wx_kx_u \\ x_wx_kx_j & x_jx_ux_wx_k \end{pmatrix}_{ju} \end{pmatrix}_{wk}, \tag{6.4}$$

where we use "-" to denote an entry is forbiden.

Corollary 6.4 (mirror rule). *For some $v \in V$, a node $u \in N^2(v)$ is called mirror of v, if $N(v)\backslash N(u)$ is a clique. We denote the set of of a node v mirrors [6] by $M(v)$. Let $G = (V, E)$ be a graph and v a vertex of G. Then*

$$\alpha(G) = \max(1 + \alpha(G\backslash N[v]), \alpha(G\backslash(M(v) \cup \{v\})). \tag{6.5}$$

This rule states that if v is not in M, there exists an MIS I that $M(v) \notin I$. otherwise, there must be one of $N(v)$ in the MIS (*local maximum rule*). Although this statement involves $N(u)$, however, deriving this rule only requires information upto second neighborhood of v. If w is in I, then none of $N(v) \cap N(w)$ is in I, then there must be one of node in the clique $N(v)\backslash N(w)$ in I (*local maximum rule*), since clique has at most one node in the MIS, by moving the occuppied node to the interior, we obtain a "better" solution. In the following example, since $u \in N^2(v)$ and $N(v)\backslash N(u)$ is a clique, u is a mirror of v.



After contracting $N[v] \cup u$, v becomes an internal degree of freedom. Applying tensor compactifying rule Eq. (6.2), the resulting rank 4 tropical tensor is

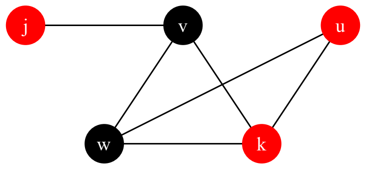
$$T_{juwk} = \begin{pmatrix} \begin{pmatrix} 1 & 2 \\ \cancel{1} & \cancel{2} \end{pmatrix}_{ju} & \begin{pmatrix} \cancel{1} & -\infty \\ 2 & -\infty \end{pmatrix}_{ju} \\ \begin{pmatrix} \cancel{1} & \cancel{2} \\ -\infty & -\infty \end{pmatrix}_{ju} & \begin{pmatrix} -\infty & -\infty \\ -\infty & -\infty \end{pmatrix}_{ju} \end{pmatrix}_{wk}, \tag{6.6}$$

where entries striked through are removed by compactifying.

Corollary 6.5 (satellite rule). *Let G be a graph, $v \in V$. A node $u \in N^2(v)$ is called satellite [11] of v, if there is some $u' \in N(v)$ such that $N[u'] \backslash N[v] = \{u\}$. The set of satellites of a node v is denotedby $S(v)$, and we also use the notation $S[v] := S(v) \cup v$. Then*

$$\alpha(G) = \max\{\alpha(G\backslash\{v\}), \alpha(G\backslash N[S[v]]) + |S(v)| + 1\}. \tag{6.7}$$

This rule can be capture by contracting $N[v] \cup S(v)$. In the following example, since $u \in N^2(v)$ and $w \in N(v)$ satisfies $N[w] \backslash N[v] = \{u\}$, u is a satellite of v.

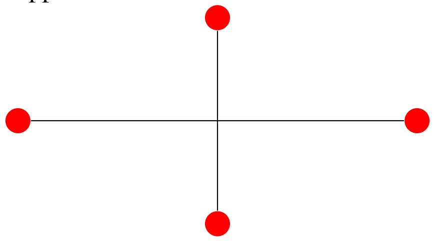


After contracting $N[v] \cup u$, both v and w become internal degrees of freedoms. Applying tensor compactifying rule Eq. (6.2), the resulting rank 3 tropical tensor is

$$T_{juk} = \begin{pmatrix} \begin{pmatrix} 1 & 2 \\ 2 & \cancel{2} \end{pmatrix}_{ju} \\ \begin{pmatrix} \cancel{1} & -\infty \\ \cancel{2} & -\infty \end{pmatrix}_{ju} \end{pmatrix}_{k}. \tag{6.8}$$

There are 3 nonzero entries. The internal configurations of entry $T(j = 1, u = 0, k = 0) = 2$ is $(v = 0, w = 1)$, that of entry $T(j = 0, u = 1, k = 0) = 2$ is $(v = 1, w = 0)$, and that of entry $T(j = 0, u = 0, k = 0) = 1$ is $(v = 1, w = 0)$ or $(v = 0, w = 1)$. For entry $T(j = 0, u = 0, k = 0) = 1$, we post-select the internal degree of freedom as $(v = 0, w = 1)$. Then we can see the satellite rule either $v, u \in I$ or $v \notin I$ is satisfied. In this case, the effective branching number is $3^{1/5} \approx 1.2457$.

6.2. gadget design. Suppose we have a local structure as the following.



Contract this local structure gives the tropical tensor

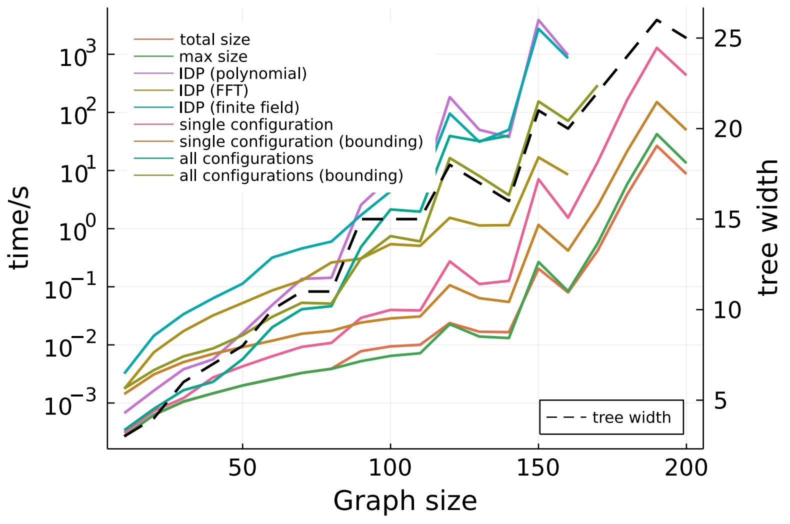
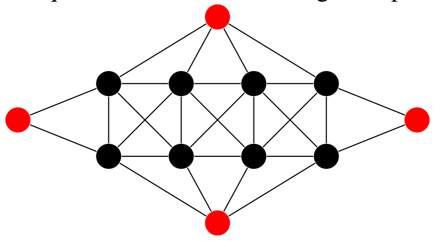


Figure 2: Benchmark results for computing different properties with different element types. The right axis is only for the dashed line.

$$\begin{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 2 \end{pmatrix} & \begin{pmatrix} 1 & -\infty \\ 2 & -\infty \end{pmatrix} \\ \begin{pmatrix} 1 & 2 \\ -\infty & -\infty \end{pmatrix} & \begin{pmatrix} 2 & -\infty \\ -\infty & -\infty \end{pmatrix} \end{pmatrix}. \tag{6.9}$$

The following gadget is equivalent to the above diagram up to a constant 2.



$$\begin{pmatrix} \begin{pmatrix} 2 & 3 \\ 3 & 4 \end{pmatrix} & \begin{pmatrix} 3 & 3 \\ 4 & 4 \end{pmatrix} \\ \begin{pmatrix} 3 & 4 \\ 2 & 3 \end{pmatrix} & \begin{pmatrix} 4 & 4 \\ 3 & 4 \end{pmatrix} \end{pmatrix} \xrightarrow[\text{compactify, -2}]{} \begin{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 2 \end{pmatrix} & \begin{pmatrix} 1 & \cancel{1} \\ 2 & \cancel{2} \end{pmatrix} \\ \begin{pmatrix} 1 & 2 \\ \cancel{0} & \cancel{1} \end{pmatrix} & \begin{pmatrix} 2 & \cancel{2} \\ \cancel{1} & \cancel{2} \end{pmatrix} \end{pmatrix} \tag{6.10}$$

We can see these two subgraphs produce exactly the same compact tensor. When we replace the original tensor with this gadget, the solution.

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

8. discussion.

We introduced in the main text how to compute the indenpendence polynomial, maximum independent set and optimal configurations. It is interesting that although these properties are global, they can be solved by designing different element types that having two operations $\oplus$ and $\odot$ and two special elements $\mathbb{0}$ and $\mathbb{1}$. One thing in common is that they all defines a commutative semiring. Here, we want the $\oplus$ and $\odot$ operations being commutative because we do not want the contraction result of an einsum network to be sensitive to the contraction order. We show most of the implementation in Appendix A. It is supprisingly short. The style that we program is called generic programming, it is about writing a single copy of code, feeding different types into it, and the program computing the result with a proper performance. It is language dependent feature. If someone want to implement this algorithm in python, one has to rewrite the matrix multiplication for different element types in C and then export the interface to python. In C++, users can use templates for such a purpose. In our work, 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, tropical number, tropical number with counting/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.

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. (5.2))	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. (3.9)	fitting the indenpendence polynomials exactly using number theory

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

REFERENCES

[1] S. F. Barr, *Courcelle's Theorem: Overview and Applications*, PhD thesis, Oberlin College, 2020.
[2] C. M. Bishop, *Pattern Recognition and Machine Learning*, Springer, 2006.

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

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

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

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

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

[8] 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.

[9] 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.

[10] 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.

[11] J. Kneis, A. Langer, and P. Rossmanith, *A fine-grained analysis of a simple independent set algorithm*, in IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2009.

[12] 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.

[13] 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.

[14] 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.

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

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

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

[18] 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).

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

[20] 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 einsum,

OMEinsumContractionOrders a package for finding the optimal contraction order for einsum 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 supprise 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.

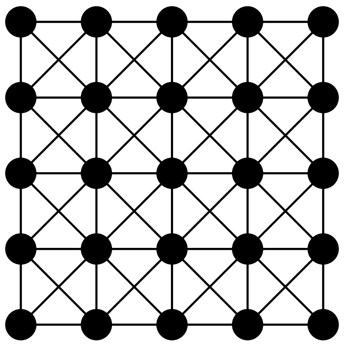
```julia
using OMEinsum, OMEinsumContractionOrders
using OMEinsum: NestedEinsum, flatten, getixs
using LightGraphs
using Random

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

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

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

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

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

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

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

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

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

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

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

# define the 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)")
```

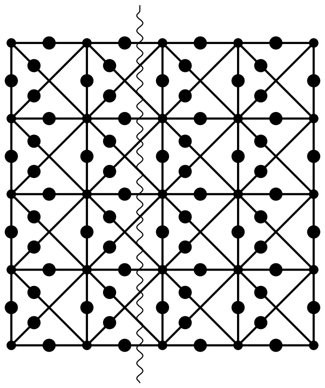
In the above examples, the configuration enumeration is very slow, one should use the optimal MIS size for bounding as decribed in the main text. We will not show any example about implementing the backward rule here because it has approximately 100 lines of code. Please checkout our Github repository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

Appendix B. When a tensor network is worse than an einsum network.

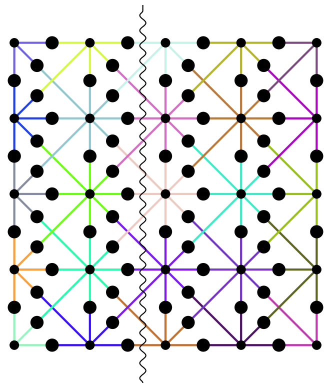
Given a graph



Its tensor network representation 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 columwise order, the maximum intermediate tensor is approximately $3L$, giving a space complexity $\approx 2^{3L}$. If we treat it as the following einsum 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 an einsum network. To understand its representation power, it is a good starting point to connect it with dynamic programming because an einsum 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 [3, 1] states that a problem quantified by monadic second order logic (MSO) on a graph with bounded treewidth k can be solved in linear time with respect to the graph size. Dynamic programming is a traditional approach to attack a MSO problem, it can solve the maximum independent set problem in $O(2^k)n$, which is similar to the einsum network approach. We mentioned in the main text that einsum network has nice mathematic property make it easier for generic programming. The cost is, the einsum network is less expressive than dynamic programming, However, that are still some other problems that can be expressed in the framework of generic einsum 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-Coloring. Let us use 3-coloring on the vertex as an example. We can define a vertex tensor as

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

and an edge tensor as

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

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