

SOLVING THE INDEPENDENT SET PROBLEM BY GENERIC PROGRAMMING TENSOR NETWORKS *

XXX[†] AND YYY[‡]

Abstract. This paper is about solving the independent set problem by encoding 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. We also show how to analyse the local properties of a graph by contracting an open tensor network.

Key words. maximum independent set, tensor network

AMS subject classifications. 05C31, 14N07

1. Introduction. [JG: efficient counting maximal independent sets [17]] 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 [13], 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 and dynamic programming. Without changing the fact of exponential scaling in computing time, the branching algorithm gives a smaller base. For example, in [25], a sophisticated branching algorithm has a time complexity $1.1893^n n^{O(1)}$. The dynamic programming approach [5, 8] 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 an NP-complete problem that directly related to other NP-complete problems like maximal cliques and vertex cover [19], but also for its close relation with physical applications like hard spheres lattice gas model [6], and Rydberg hamiltonian [22]. 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 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 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 index appears precisely twice. When we have multiple tensors doing the above sum-product

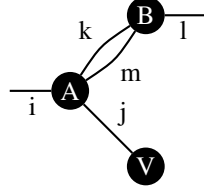
Funding: ...

[†]XXX (email, website).

[‡]yyyyy (yyyy, email).

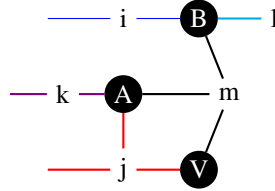
operation, we get a traditional tensor network [20]. A traditional tensor network can be represented as a multigraph with open edges by viewing a tensor as a vertex, a label pairing two tensors as an edge, and the remaining unpaired labels as open edges.

Example 1. A traditional tensor network $C_i^l = A_{ij}^{km} B_{km}^l V^j$ 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 $C_{ijk} = \sum_{ml} A_{jkm} B_{mil} V_{jm}$. 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 [7, 12] 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

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

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 rigorously requires a computing time $O(1.442^n)$ [7][JG: I am not sure about this complexity, this is baed on the naive analysis of theorem 2.2 in [7]]. There are some interests in approximating this polynomial efficiently [11], 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

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

2

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

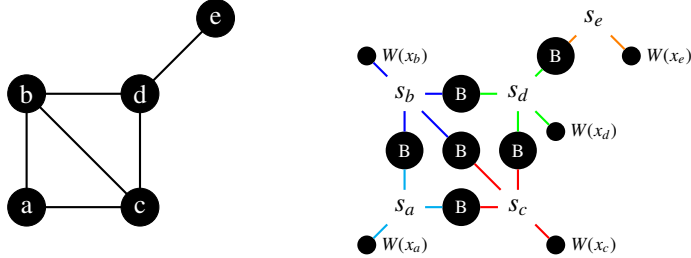
75 (3.3)
$$B_{s_i s_j} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}_{s_i s_j},$$

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

79 (3.4)
$$P(G, \{x_1, \dots, x_n\}) = \sum_{s_1, s_2, \dots, s_n=0}^1 \prod_{i=1}^n W(x_i)_{s_i} \prod_{(i,j) \in E(G)} B_{s_i s_j},$$

80 where the summation runs over all vertex configurations $\{s_1, \dots, s_n\}$ and accumulates the
 81 product of tensor elements to the scalar output P . We can see an edge tensor represents
 82 the restriction on an edge that if both vertices connected by it are included in the set, then
 83 such configuration has no contribution to the output. When we set $x_i = x$, the contraction
 84 result corresponds to the independence polynomial. One can see the connection from the
 85 fact that the product over vertex tensor elements gives a factor x^k , where $k = \sum_i s_i$ counts
 86 the set size, and the product over edge tensor elements gives a factor 1 for a configuration
 87 being in an independent set, 0 otherwise. One directly benefit of mapping the independent set
 88 problem to a tensor network is one can take the advantage of recently developed techniques
 89 in tensor network based quantum circuit simulations [10, 21], where people evaluate a tensor
 90 network by pairwise contracting tensors in a heuristic order. A good contraction order can
 91 reduce the time complexity significantly, at the cost of having a space overhead of $O(2^{tw(G)})$.
 92 Here $tw(G)$ is the tree width of the line graph of a tensor network hypergraph, while the line
 93 graph of a tensor network hypergraph corresponds to the original graph G that we mapped
 94 from. [18] The pairwise tensor contraction also makes it possible to utilize basic linear algebra
 95 subprograms (BLAS) functions to speed up our computation for certain tensor element types.

96 **Example 3.** Mapping a graph (left) to a tensor network, the resulting tensor network is
 97 shown in the right panel. In the generalize tensor network's graphical representation, a vertex
 98 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.

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

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 is 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 $a, b, c \in R$.

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

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 [24]. 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 $a_0 + a_1x + \dots + a_kx^k$ as a vector $(a_0, a_1, \dots, a_k) \in R^k$, 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

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

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 [23]. 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 [9] 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 unknown integer x over a set of co-prime integers $\{p_1, p_2, \dots, p_n\}$, $x \pmod{p_1 \times p_2 \times \dots \times p_n}$ can be computed using the Chinese remainder theorem. With this, one can infer big integers from small integers.

With these observations, we 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. Some times people are interested in knowing maximal solutions to understand why their programs are trapped in a local minimal. Then they might want to compute the maximal independence polynomial. 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 $N[v]$

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

Algorithm 3.1 Compute independence polynomial exactly without integer overflow

Let $P = 1$, vector $X = (0, 1, 2, \dots, m)$, matrix $\hat{X}_{ij} = X_i^j$, where $i, j = 0, 1, \dots, m$

```

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  $Y = (y_0, y_1, \dots, y_m) \pmod{p}$ 
   $A_p = (a_0, a_1, \dots, a_m) \pmod{p} = \text{gaussian\_elimination}(\hat{X}, Y \pmod{p})$ 
   $A_{P \times p} = \text{chinese\_remainder}(A_p, A_p)$ 
  if  $A_p = A_{P \times p}$  then
    return  $A_p$  ; // converged
  end
   $P = P \times p$ 
end
  
```

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

177 (3.8)
$$T(x_v) = \begin{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 1 \\ x_v & 0 \\ 0 & 0 \end{pmatrix} \end{pmatrix}.$$

178

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

185 **4. Maximum independent sets and its counting problem.** In the previous section, we
 186 focused on computing independence polynomial for a given maximum independent set size
 187 $\alpha(G)$, but we didn't mention how to compute this number. The method we use to compute
 188 this quantity is based on the following observations. Let $x = \infty$, the independence polynomial
 189 becomes

190 (4.1)
$$I(G, \infty) = a_k \infty^{\alpha(G)},$$

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

192 (4.2)
$$a_x \infty^x \oplus a_y \infty^y = \begin{cases} (a_x + a_y) \infty^{\max(x,y)}, & x = y \\ a_y \infty^{\max(x,y)}, & x < y \\ a_x \infty^{\max(x,y)}, & x > y \end{cases}$$

$$a_x \infty^x \odot a_y \infty^y = a_x a_y \infty^{x+y}$$

$$\mathbb{0} = 0 \infty^{-\infty}$$

$$\mathbb{1} = 1 \infty^0$$

193

194 In the program, we only store the power x and the corresponding factor a_x that initialized to
 195 1. This algebra is the same as the one in [15] for counting spin glass ground states. If one is
 196 only interested in obtaining $\alpha(G)$, he can drop the factor parts, then the new algebra becomes
 197 the max-plus tropical algebra [16, 19].

198 **4.1. Sub-optimal solutions.** Some times people are interested in finding sub-optimal
 199 solutions efficiently. We define a truncated polynomial algebra by keeping only largest two

200 factors in the polynomial in Eq. (3.5).

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

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

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

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

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

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

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

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

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

226 **5.1. Bounding the enumeration space.** When one uses the set algebra in Eq. (5.1) to
227 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

$$(5.4) \quad A_{ij} \odot B_{jk} \leq C_{ik}.$$

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

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

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

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

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 \bar{C} , the backward rule for gradient masks reads

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

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 $C^{\circ-1} \circ \bar{C}$ and B^T by a tensor contraction. [] [JG: maybe add an appendix?]

6. Tropical tensors for automated branching. [JG: ?]

Branching rules can be automatically discovered by contracting the tensor network on a subgraph $R \subseteq G$ with tropical numbers as its element type. Let C be the set of boundary vertices defined as $C := \{u | u \in R \wedge (\exists v \in (G \setminus R) \wedge \text{adj}(u, v))\}$, then the rank of the resulting tensor A is $|C|$. Here, we use $\text{adj}(u, v)$ to denote two vertices u and v are adjacent to each other. Each tensor entry A_σ is a local maximum independent set size for the fixed boundary configuration $\sigma \in \{0, 1\}^{|C|}$. Suppose our goal is to find the maximum independent set size, then this tensor can be further “compactified” by removing some entries. To determine which entry can be removed, let us define a relation of *less restrictive* as

$$(6.1) \quad (\sigma_a < \sigma_b) := (\sigma_a \neq \sigma_b) \wedge (\sigma_a \leq^\circ \sigma_b)$$

where \leq° is the Hadamard less or equal to operation.

DEFINITION 6.1. A tensors A is MIS-compact if

$$(6.2) \quad \forall \sigma_b \neg \exists \sigma_a (\sigma_a < \sigma_b) \wedge (A_{\sigma_a} \geq A_{\sigma_b}).$$

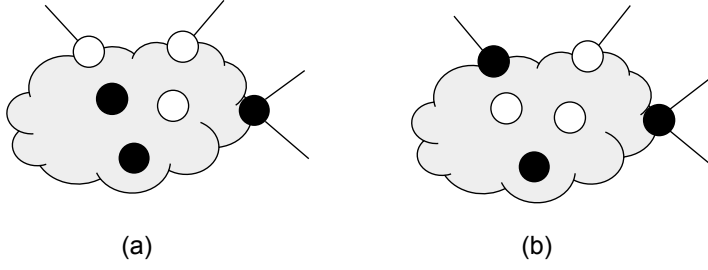
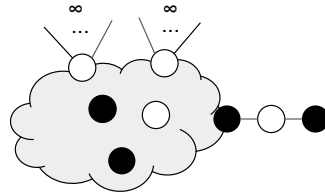


Figure 1: Two configurations with the same local independent size $A_{\sigma_a} = A_{\sigma_b} = 3$ and different boundary configurations (a) $\sigma_a = \{001\}$ and (b) $\sigma_b = \{101\}$, where black nodes are 1s (in the independent set) and white nodes are 0s (not in the independent set).

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

281 **THEOREM 6.2.** *A MIS-compact tropical tensor can not be further reduce without gloal*
 282 *information, i.e. any of its non-zero entries can produce the only global optimal solution*
 283 *given a proper environment.*

284 *Proof.* Let us prove it by showing for any σ in a MIS-compact tropical tensor of a
 285 subgraph R , there exists a parent graph G that $R \subseteq G$ and σ is the boundary configuration
 286 that gives the only maximum independent set. Let A be a tropical tensor, and an entry of it
 287 being A_σ , where σ is the boundary configuration. Let us construct a graph G such that for a
 288 vertex $v \in C$, if $\sigma_v = 1$, we connect it with two vertices $u, w \in G \setminus R$ that
 289 $\text{adj}(v, u) \wedge \text{adj}(u, w) \wedge \neg \text{adj}(v, w)$. Otherwise, we attach infinite many disconnected neighbors
 290 to v .



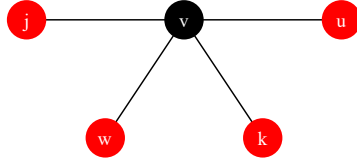
291 Then we have the maximum independent set size $\alpha(G, \sigma) = A_\sigma + \infty(|C| - \sum_{v=1}^{|C|} \sigma_v) +$
 292 $\sum_{v=1}^{|C|} 1 - \sigma_v$. Let us assume there exists another configuration τ that generating the same or
 293 better maximum independent set size $\alpha(G, \tau) \geq \alpha(G, \sigma)$. Then we have $\tau < \sigma$, otherwise
 294 it will loss infinite contribution from the environment. For such a τ , we have $A_\tau < A_\sigma$,
 295 otherwise $A_\sigma < A_\tau$ contradicts with A being MIS-compact. Finally, we have $\alpha(G, \tau) =$
 296 $\infty(|C| - |\sigma|) + A_\tau + \sum_{v=1}^{|C|} 1 - \sigma_v < \alpha(G, \sigma)$, hence σ is the only boundary configuration that
 297 gives the maximum independent set for this graph. \square

6.1. The tensor network compactification 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 neighbourhoods, and keep only the configurations that has the potential to produce the only maximum independent set. Since an MIS-compact tensor is optimal, by analysing the correlation of vertex configurations on the resulting tensor for the k -th neighbourhood $N^k[v]$, one can discover the optimal branching vector automatically. In the following, we are going to introduce several important rules for branching in the literature and show how it is connected to our tensor formulation.

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 independent 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 independent set size.



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

$$(6.3) \quad T_{juwk} = \left(\left(\begin{pmatrix} 1 & -\infty \\ -\infty & 2 \end{pmatrix}_{ju} \begin{pmatrix} -\infty & 2 \\ 2 & 3 \end{pmatrix}_{ju} \right)_{ju} \right)_{wk},$$

where we use “ $-\infty$ ” to denote an entry is forbidden. If we use sets for counting, one can check all configurations too. The resulting polynomial tensor is

$$(6.4) \quad P_{juwk} = \left(\left(\begin{pmatrix} 1 + x_v & - \\ - & x_j x_u \end{pmatrix}_{ju} \begin{pmatrix} - & x_u x_k \\ x_j x_k & x_u x_j x_k \end{pmatrix}_{ju} \right)_{ju} \right)_{wk}.$$

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. It is easier to see if we list the set of optimal solutions as

$$(6.5) \quad S_{juwk} = \{00001, 10001, 01010, 10010, 11010, 10100, 01100, 11100, 00110, 01110, 10110, 11110\}.$$

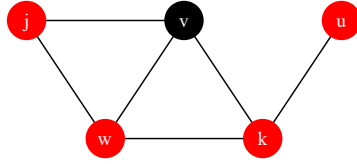
The branching vector $(1, 5)$ gives a branching number $\tau(1, 5) \approx 1.3247$

COROLLARY 6.4 (mirror rule). *For some $v \in V$, a node $u \in N^2(v)$ is called mirror of v , if $N(v) \setminus N(u)$ is a clique. We denote the set of of a node v mirrors [8] by $M(v)$. Let $G = (V, E)$*

330 be a graph and v a vertex of G . Then

$$331 \quad (6.6) \quad \alpha(G) = \max(1 + \alpha(G \setminus N[v]), \alpha(G \setminus (M(v) \cup \{v\}))).$$

332 This rule states that if v is not in M , there exists an MIS I that $M(v) \notin I$. otherwise, there
 333 must be one of $N(v)$ in the MIS (*local maximum rule*). Although this statement involves $N(u)$,
 334 however, deriving this rule only requires information upto second neighbourhood of v . If w is
 335 in I , then none of $N(v) \cap N(w)$ is in I , then there must be one of node in the clique $N(v) \setminus N(w)$
 336 in I (*local maximum rule*), since clique has at most one node in the MIS, by moving the
 337 occupied node to the interior, we obtain a “better” solution. In the following example, since
 338 $u \in N^2(v)$ and $N(v) \setminus N(u)$ is a clique, u is a mirror of v .



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

$$341 \quad (6.7) \quad T_{juwk} = \left(\left(\begin{pmatrix} 1 & 2 \\ \cancel{x} & \cancel{z} \end{pmatrix}_{ju} \begin{pmatrix} \cancel{x} & -\infty \\ 2 & -\infty \end{pmatrix}_{ju} \right) \begin{pmatrix} -\infty & -\infty \\ -\infty & -\infty \end{pmatrix}_{ju} \right)_{wk},$$

343 where entries stroked through are removed by compactification. The corresponding polyno-
 344 mial tensor is

$$345 \quad (6.8) \quad P_{juwk} = \left(\begin{pmatrix} 1 + x_v & x_u + x_u x_v \\ / & / \end{pmatrix}_{ju} \begin{pmatrix} x_j x_k & - \\ - & - \end{pmatrix}_{ju} \right)_{wk}.$$

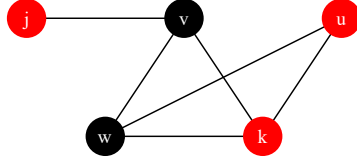
347 One can see w , as a mirror of v does not appear in the maximum independent set after com-
 348 pactification.

$$349 \quad (6.9) \quad S_{juwkv} = \{00001, 01001, 10010\}.$$

351 **COROLLARY 6.5** (satellite rule). Let G be a graph, $v \in V$. A node $u \in N^2(v)$ is called
 352 satellite [14] of v , if there is some $u' \in N(v)$ such that $N[u'] \setminus N[v] = \{u\}$. The set of satellites
 353 of a node v is denoted by $S(v)$, and we also use the notation $S[v] := S(v) \cup v$. Then

$$354 \quad (6.10) \quad \alpha(G) = \max\{\alpha(G \setminus \{v\}), \alpha(G \setminus N[S[v]]) + |S(v)| + 1\}.$$

355 This rule can be capture by contracting $N[v] \cup S(v)$. In the following example, since
 356 $u \in N^2(v)$ and $w \in N(v)$ satisfies $N[w] \setminus N[v] = \{u\}$, u is a satellite of v .



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

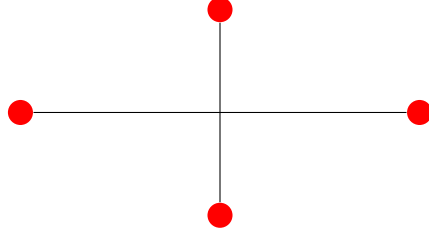
$$(6.11) \quad P_{juk} = \left(\begin{array}{cc|c} 1 + x_w + x_v & x_u + x_u x_v & \\ x_j + x_w x_j & & /_{ju} \\ & / & - \\ & / & - \end{array} \right)_{ju}_k.$$

360
 361 By choosing one of the optimal configurations in each entry, we can see the satellite rule
 362 of either $v, u \in I$ or $v \notin I$ is satisfied.

$$(6.12) \quad S_{juwkv} = \{\{00100, 00001\}, 10100, 01001\}.$$

365 6.2. gadget design. [JG: ×]

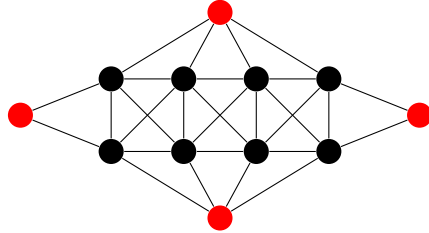
366 Suppose we have a local structure as the following.



367 Contract this local structure gives the tropical tensor

$$(6.13) \quad \left(\begin{array}{cc} 0 & 1 \\ 1 & 2 \\ 1 & 2 \\ -\infty & -\infty \end{array} \right) \left(\begin{array}{cc} 1 & -\infty \\ 2 & -\infty \\ 2 & -\infty \\ -\infty & -\infty \end{array} \right).$$

369
 370 The following gadget is equivalent to the above diagram up to a constant 2.



$$(6.14) \quad \left(\begin{array}{cc} 2 & 3 \\ 3 & 4 \\ 3 & 4 \\ 2 & 3 \end{array} \right) \left(\begin{array}{cc} 3 & 3 \\ 4 & 4 \\ 4 & 4 \\ 3 & 4 \end{array} \right) \xrightarrow{\text{compactify, } -2} \left(\begin{array}{cc} 0 & 1 \\ 1 & 2 \\ 1 & 2 \\ \emptyset & \lambda' \end{array} \right) \left(\begin{array}{cc} 1 & \lambda' \\ 2 & \lambda' \\ 2 & \lambda' \\ \lambda' & \lambda' \end{array} \right)$$

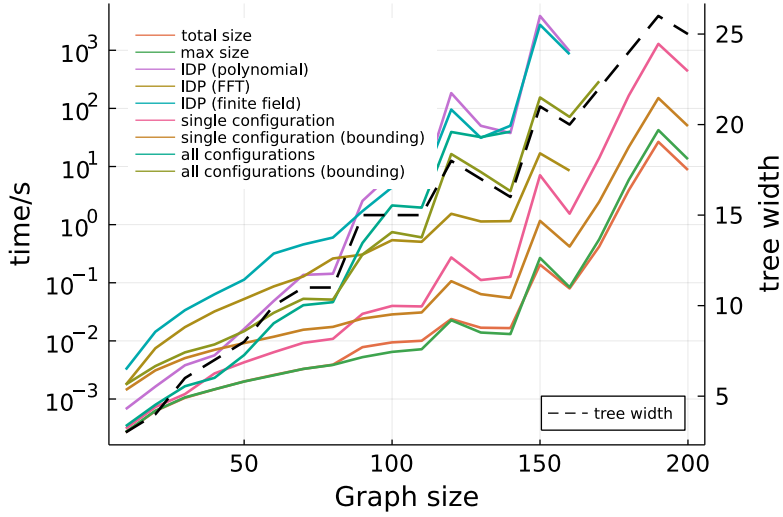


Figure 2: Benchmark results for computing different properties with different element types. The right axis is only for the dashed line.

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 and case study. We run a sequential 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.

8. 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 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].

REFERENCES

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

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.

- [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] B. COURCELLE, *The monadic second-order logic of graphs. i. recognizable sets of finite graphs*, Information and computation, 85 (1990), pp. 12–75.
- [6] J. C. DYRE, *Simple liquids' quasiuniversality and the hard-sphere paradigm*, Journal of Physics: Condensed Matter, 28 (2016), p. 323001.
- [7] G. M. FERRIN, *Independence polynomials*, (2014).
- [8] F. V. FOMIN AND P. KASKI, *Exact exponential algorithms*, Communications of the ACM, 56 (2013), pp. 80–88.
- [9] G. H. GOLUB AND C. F. VAN LOAN, *Matrix computations*, vol. 3, JHU press, 2013.
- [10] 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>.
- [11] 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.
- [12] 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>.
- [13] J. HASTAD, *Clique is hard to approximate within n^{ϵ}* , in Proceedings of 37th Conference on Foundations of Computer Science, IEEE, 1996, pp. 627–636.
- [14] 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.
- [15] 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>.
- [16] 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>.
- [17] 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.
- [18] 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>.
- [19] C. MOORE AND S. MERTENS, *The nature of computation*, OUP Oxford, 2011.
- [20] R. ORÚS, *A practical introduction to tensor networks: Matrix product states and projected entangled pair states*, Annals of Physics, 349 (2014), pp. 117–158.
- [21] F. PAN AND P. ZHANG, *Simulating the sycamore quantum supremacy circuits*, 2021, <https://arxiv.org/abs/2103.03074>.
- [22] 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).
- [23] A. SCHÖNHAGE AND V. STRASSEN, *Schnelle multiplikation grosser zahlen*, Computing, 7 (1971), pp. 281–292.
- [24] A. A. STEPANOV AND D. E. ROSE, *From mathematics to generic programming*, Pearson Education, 2014.
- [25] M. XIAO AND H. NAGAMUCHI, *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 of 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.

```
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`.
```

```

483 # assign each label a dimension-2, it will be used in contraction order optimization
484 # `symbols` function extracts tensor labels into a vector.
485 symbols(:EinCode{ixs}) where ixs = unique(Iterators.flatten(filter(x->length(x)==1,ixs)))
486 symbols(ne::OMEinsum.NestedEinsum) = symbols(flatten(ne))
487 size_dict = Dict{<math>s \geq 2</math> for s in symbols(code)}
488 # optimize the contraction order using KaHyPar + Greedy, target space complexity is  $2^{17}$ 
489 optimized_code = optimize_kahypar(code, size_dict; sc_target=17, max_group_size=40)
490 println("time/space complexity is $(OMEinsum.timespace_complexity(optimized_code, size_dict))")
491
492 # a function for computing independence polynomial
493 function independence_polynomial(x::T, code) where {T}
494     xs = map(getixs(flatten(code))) do ix
495         # if the tensor rank is 1, create a vertex tensor.
496         # otherwise the tensor rank must be 2, create a bond tensor.
497         length(ix)==1 ? [one(T), x] : [one(T) one(T); one(T) zero(T)]
498     end
499     # both `EinCode` and `NestedEinsum` are callable, inputs are tensors.
500     code(xs...)
501 end
502
503 ##### COMPUTING MAXIMUM INDEPENDENT SET SIZE AND ITS DEGENERACY #####
504
505 # using Tropical numbers to compute the MIS size and MIS degeneracy.
506 using TropicalNumbers
507 mis_size(code) = independence_polynomial(TropicalF64(1.0), code)[]
508 println("the maximum independent set size is $(mis_size(optimized_code).n)")
509 # A `CountingTropical` object has two fields, tropical field `n` and counting field `c`.
510 mis_count(code) = independence_polynomial(CountingTropical{Float64,Float64}(1.0, 1.0), code)[]
511 println("the degeneracy of maximum independent sets is $(mis_count(optimized_code).c)")
512
513 ##### COMPUTING INDEPENDENCE POLYNOMIAL #####
514
515 # using Polynomial numbers to compute the polynomial directly
516 using Polynomials
517 println("the independence polynomial is $(independence_polynomial(Polynomial([0.0, 1.0]),
518     optimized_code)[])")
519
520 # using fast fourier transformation to compute the independence polynomial,
521 # here we chose  $r > 1$  because we care more about configurations with large independent set sizes
522 .
523 using FFTW
524 function independence_polynomial_fft(code; mis_size=Int(mis_size(code)[].n), r=3.0)
525      $\omega = \exp(-2im\pi/(mis\_size+1))$ 
526     xs = r .* collect( $\omega$  .^ (0:mis_size))
527     ys = [independence_polynomial(x, code)[] for x in xs]
528     Polynomial(ifft(ys) ./ (r .^ (0:mis_size)))
529 end
530 println("the independence polynomial (fft) is $(independence_polynomial_fft(optimized_code))")
531
532 # using finite field algebra to compute the independence polynomial
533 using Mods, Primes
534 # two patches to ensure gaussian elimination works
535 Base.abs(x::Mod) = x
536 Base.isless(x::Mod{N}, y::Mod{N}) where N = mod(x.val, N) < mod(y.val, N)
537
538 function independence_polynomial_finitefield(code; mis_size=Int(mis_size(code)[].n), max_order=1
539     00)
540     N = typemax(Int32) # Int32 is faster than Int.
541     YS = []
542     local res
543     for k = 1:max_order
544         N = Primes.prevprime(N-one(N)) # previous prime number
545         # evaluate the polynomial on a finite field algebra of modulus `N`
546         rk = _independence_polynomial(Mods.Mod{N,Int32}, code, mis_size)
547         push!(YS, rk)
548         if max_order==1
549             return Polynomial(Mods.value.(YS[1]))
550         elseif k != 1
551             ra = improved_counting(YS[1:end-1])
552             res = improved_counting(YS)
553             ra == res && return Polynomial(res)
554         end
555     end
556     @warn "result is potentially inconsistent."

```



```

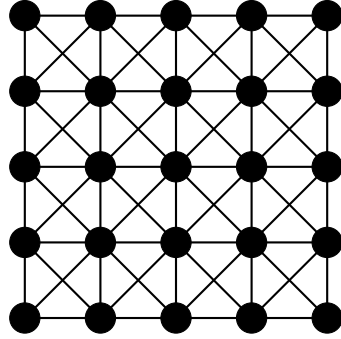
557     return Polynomial(res)
558 end
559 function _independence_polynomial(::Type{T}, code, mis_size::Int) where T
560     xs = 0:mis_size
561     ys = [independence_polynomial(T(x), code)[] for x in xs]
562     A = zeros{T, mis_size+1, mis_size+1}
563     for j=1:mis_size+1, i=1:mis_size+1
564         A[j,i] = T(xs[j])^(i-1)
565     end
566     A \ T.(ys) # gaussian elimination to compute ``A^{-1} y``
567 end
568 improved_counting(sequences) = map(yi->Mods.CRT(yi...), zip(sequences...))
569
570 println("the independence polynomial (finite field) is $(independence_polynomial_finitefield(
571     optimized_code))")
572
573 ##### FINDING OPTIMAL CONFIGURATIONS #####
574
575 # define the config enumerator algebra
576 struct ConfigEnumerator{N,C}
577     data::Vector{StaticBitVector{N,C}}
578 end
579 function Base.+(x::ConfigEnumerator{N,C}, y::ConfigEnumerator{N,C}) where {N,C}
580     res = ConfigEnumerator{N,C}(vcat(x.data, y.data))
581     return res
582 end
583 function Base.*(x::ConfigEnumerator{L,C}, y::ConfigEnumerator{L,C}) where {L,C}
584     M, N = length(x.data), length(y.data)
585     z = Vector{StaticBitVector{L,C}}(undef, M*N)
586     for j=1:N, i=1:M
587         z[(j-1)*M+i] = x.data[i] .| y.data[j]
588     end
589     return ConfigEnumerator{L,C}(z)
590 end
591 Base.zero(::Type{ConfigEnumerator{N,C}}) where {N,C} = ConfigEnumerator{N,C}(StaticBitVector{N,C}[])
592 Base.one(::Type{ConfigEnumerator{N,C}}) where {N,C} = ConfigEnumerator{N,C}([TropicalNumbers.staticfalses(StaticBitVector{N,C})])
593
594 # enumerate all configurations if `all` is true, compute one otherwise.
595 # a configuration is stored in the data type of `StaticBitVector`, it uses integers to represent
596 # bit strings.
597 # `ConfigTropical` is defined in `TropicalNumbers`. It has two fields, tropical number `n` and
598 # optimal configuration `config`.
599 # `CountingTropical{T,<:ConfigEnumerator}` is a simple stores configurations instead of simple
600 # counting.
601 function mis_config(code; all=false)
602     # map a vertex label to an integer
603     vertex_index = Dict{String{Char}, Int}()
604     for (i, s) in enumerate(symbols(code))
605         vertex_index[s] = i
606     end
607     N = length(vertex_index) # number of vertices
608     C = TropicalNumbers.nints(N) # number of integers to store N bits
609     xs = map(getixs(flatten(code))) do ix
610         T = all ? CountingTropical{Float64, ConfigEnumerator{N,C}} : ConfigTropical{Float64, N, C}
611         if length(ix) == 2
612             return [one(T) one(T); one(T) zero(T)]
613         else
614             s = TropicalNumbers.onehot(StaticBitVector{N,C}, vertex_index[ix[1]])
615             if all
616                 [one(T), T(1.0, ConfigEnumerator{N,C}(s))]
617             else
618                 [one(T), T(1.0, s)]
619             end
620         end
621     end
622     return code(xs...)
623 end
624
625 println("one of the optimal configurations is $(mis_config(optimized_code; all=false)[].config)")
626 )
627
628 # enumerating configurations directly can be very slow (~15min), please check the bounding
629 # version in our Github repo.
630 println("all optimal configurations are $(mis_config(optimized_code; all=true)[].c)")

```

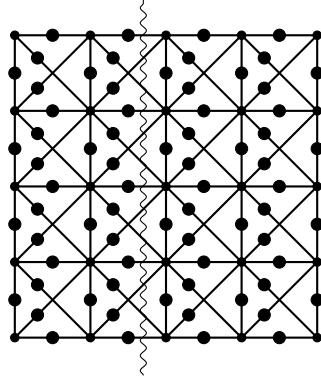
In the above examples, the configuration enumeration is very slow, one should use the optimal MIS size for bounding as described in the main text. We will not show any example about implementing the backward rule here because it has approximately 100 lines of code. Please checkout our GitHub repository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

Appendix B. Why not introducing δ tensors.

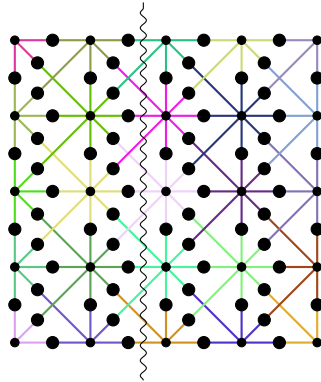
Given a graph



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 [5, 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

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

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,

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

and a tensor of rank 1 on the bond

$$(C.3) \quad B_{v \rightarrow w} = \begin{cases} 1, & v \rightarrow w = 0 \\ x, & v \rightarrow w = 1. \end{cases}$$

Here, we use bond index $v \rightarrow w$ 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

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

and an edge tensor as

$$(C.5) \quad B = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}.$$

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

683 set of real numbers of cardinality $m + 1$. We compute the tensor network contraction for
 684 each $x_i \in X$ and obtain the following relations

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

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

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

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

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

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