Solving the maximum independant set problem with einsum networks

Jin-Guo Liu
Harvard University
jinguoliu@g.harvard.edu

Xun Gao
Harvard University
xungao@g.harvard.edu

Abstract

Solving the maximum independent set size problem by mapping the problem to an einsum network. We obtain the maximum independent set size, the independence polynomial and the optimal configuration by using different tensor elements to a commutative semiring.

1 Three approaches to computing independence polynomial

The independence polynomial (Ferrin, 2014; Harvey et al., 2017) of a graph G is defined as

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

where a_k is the number of independent sets of size k in G, and $\alpha(G)$ is the maximum independent set size. [**JG: here, I preassumed a user knows tensor network, add more details later.**] If we map a graph into an einsum network, as shown in Fig. 1, by placing a rank one tensor of size 2 at each vertex

$$W(x)_{s_i} = \begin{pmatrix} 1 \\ x \end{pmatrix}_{s_i}, \tag{2}$$

and a rank two tensor of size 2×2 at each edge

$$B_{s_i s_j} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}_{s_i s_j}, \tag{3}$$

the contraction of this einsum network generates the independence polynomial

$$I(G, x) = \sum_{s_1, s_2, \ldots, s_n = 0}^{1} \prod_{i=1}^{n} W(x)_{s_i} \prod_{(i,j) \in E(G)} B_{s_i s_j}. \tag{4}$$

Here, the einsum enumerates all possible configurations and accumulates the product of tensor elements to the output. The product over vertex tensors provides a factor x^k, where $k = \sum_i s_i$ is the set size. While the product over edge tensors provides a factor 0 if the configuration is not an independent set, otherwise 1. By mapping the problem to einsum, one can use some existing algorithms to decrease the complexity of the problem. Especially those techiniques developed in recent quantum circuit simulations for finding the optimal exactly contraction order of a tensor network (Gray & Kourtis, 2021; Pan & Zhang, 2021). With these algorithms, one can easily find a good contraction order that reduces the complexity to a quantity related to the minimum treewidth of the graph. (Markov & Shi, 2008) Such mapping also makes it possible to utilize faster BLAS functions. Before contracting the tensor network and evaluating this 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

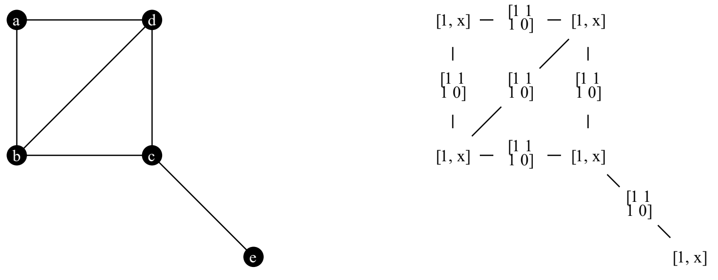

Figure 1: Mapping a graph to an einsum network.

identity of a commutative semiring. A ring without additive inverse is called a semiring, and 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 relation must hold for arbituary three elements $a, b, c \in R$.

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

1.1 Symbolic computing

To evaluate this polynomial, one can use the symbolic computing directly, that is, storing a polynomial of x as a vector of factors. We define the algebra between the polynomials a of order k_a and b of order k_b as

$$\begin{aligned}
a \oplus b &= \{a_0 + b_0, a_1 + b_1, \ldots, a_{\max(k_a,k_b)} + b_{\max(k_a,k_b)}\},\\
a \odot b &= \{a_0 + b_0, a_1 b_0 + a_0 b_1, \ldots, a_{k_a} b_{k_b}\},\\
\mathbb{0} &= \{\},\\
\mathbb{1} &= \{1\}.
\end{aligned} \tag{5}$$

The variable x now is represented as a vector $\{0, 1\}$. Then we insert them to the original tensor network contraction algorithm, the final result is the exact representation of the independence polynomial. In the program, the multiplication can be evaluated efficiently with the convolution theorem. However, this method suffers from a space overhead that propotional to the maximum independant set size because each polynomial requires a vector of such size to store.

1.2 Polynomial fitting and Fourier transformation

Let $m = \alpha(G)$ be the maximum independent set size and X be a set of $m + 1$ random real numbers, e.g. $\{0, 1, 2, \ldots, m\}$. We compute the einsum contraction for each $x \in X$ and obtain the following relations

$$\begin{aligned} a_0 + a_1 x_1 + a_1 x_1^2 + \ldots + a_m x_1^m &= y_0 \\ a_0 + a_1 x_2 + a_2 x_2^2 + \ldots + a_m x_2^m &= y_1 \\ &\ldots \\ a_0 + a_1 x_m + a_2 x_m^2 + \ldots + a_m x_m^m &= y_m \end{aligned} \tag{6}$$

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

$$\begin{pmatrix} 1 & x_1 & x_1^2 & \ldots & x_1^m \\ 1 & x_2 & x_2^2 & \ldots & x_2^m \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_m & x_m^2 & \ldots & x_m^m \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_m \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_m \end{pmatrix}. \tag{7}$$

However, the result is very unreliable in practise due to the non-negligible round off error of floating point numbers. Instead of choosing X as a set of random real numbers, we let $x_i = r\omega^i$ and the above linear equation becomes

$$\begin{pmatrix} 1 & r\omega & r^2\omega^2 & \ldots & r^m\omega^m \\ 1 & r\omega^2 & r^2\omega^4 & \ldots & r^m\omega^{2m} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & r\omega^m & r^2\omega^{2m} & \ldots & r^m\omega^{m^2} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_m \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_m \end{pmatrix}. \tag{8}$$

When $r = 1$, the left side is a DFT matrix. We can obtain the factors using the relation $\vec{a} = \mathrm{FFT}^{-1}(\omega) \cdot \vec{y}$. In the special case that $\omega = e^{-2\pi i/(m+1)}$, it is directly solvable with inverse fast fourier transformation algorithm in package `FFTW`.

$$\mathrm{FFT}(\omega) \cdot \vec{a}_r = \vec{y} \tag{9}$$

where $(\vec{a}_r)_k = a_k r^k$, by choosing diferent r, we can obtain better precision in low independant set size region ($\omega < 1$) and high independant set size region ($\omega > 1$).

1.3 Finite field algebra

It is possible to compute the independence polynomials exactly using basic integer types only, even when the degeneracy is bigger than that can be represented by 64 bit integers. What we need is computing with the finite field algebra $GF(p)$

$$\begin{aligned} x \pmod{p} \oplus y \pmod{p} &= x + y \pmod{p} \\ x \pmod{p} \odot y \pmod{p} &= xy \pmod{p} \\ \mathbb{0} &= 0 \\ \mathbb{1} &= 1 \end{aligned} \tag{10}$$

It a field because the multiplicative inverse can defined on non-zero values, which can be computed with the extended Euclidean algorithm. In a finite field algebra, we have the following observations

1. One can still use Gaussian elimination (Golub & Van Loan, 2013) to solve the linear equations in a finite field, since the multiplicative inverse is properly defined in a finite field,
2. Let remainders of an integer x over a set of coprime integers $\{p_1, p_2, \ldots, p_n\}$ being $\{x \pmod{p_1}, x \pmod{p_2}, \ldots, x \pmod{p_n}\}$, then $\{x \pmod{p_1 \times p_2 \times \ldots \times p_n}\}$ can be obtained using the chinese remainder theorem.

With these observations, we use Algorithm 1 to compute degeneracies. In the algorithm, except the computation of chinese remainder theorem, all computations are done with integers with fixed width.

Algorithm 1: Compute independence polynomial exactly without integer overflow

```
Let $P = 1, X = 0, 1, 2, \ldots, m$ and $\hat{X}_{ij} = X_i^j$, where $i, j = \{0, 1, \ldots m\}$;
while true do
    compute the largest prime 64 bit integer $p$ that $\gcd(p, P) = 1$;
    compute the tensor network contraction on $GF(p)$ and obtain $Y \pmod p = \{y_0 \pmod p, y_1 \pmod p, \ldots, y_m \pmod p\}$;
    $A \pmod p = \{a_0 \pmod p, a_1 \pmod p, \ldots, a_m \pmod p\}$ = gaussian_elimination($\hat{X}, Y \pmod p$);
    $A \pmod{P \times p}$ = chinese_remainder($A \pmod P, A \pmod p$);
    if $A \pmod P = A \pmod{P \times p}$ then
        return $A \pmod P$;
    end
    $P = P \times p$;
end
```

2 Computing maximum independent set size and its corresponding degeneracy and configurations

Obtaining the maximum independent set size and its degeneracy is computational more efficient. Let $x = \infty$, then the independence polynomial becomes

$$I(G, \infty) = a_k \infty^{\alpha(G)}, \tag{11}$$

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

$$\begin{aligned} a_x\infty^x \oplus a_y\infty^y &= \begin{cases} (a_x + a_y)\infty^{\max(x,y)}, & x = y \\ a_y\infty^{\max(x,y)}, & x < y \\ a_x\infty^{\max(x,y)}, & x > y \end{cases} \\ a_x\infty^x \odot a_y\infty^y &= a_x a_y \infty^{x+y} \\ \mathbb{0} &= 0\infty^{-\infty} \\ \mathbb{1} &= 1\infty^0 \end{aligned} \tag{12}$$

In the program, we only store the power x and the corresponding factor a_x that initialized to 1. This algebra is consistent with the one we derived in (Liu et al., 2021) that uses the tropical tensor network solving spin glass ground states. Here if we are only interested in obtaining $\alpha(G)$ and omit the factors, the algebra of x corresponds to the tropical algebra (Maclagan & Sturmfels, 2015; Moore & Mertens, 2011).

One may want to obtain all ground state configurations, it can be achieved replacing the factors a_x with a set of bit strings s_x initialized to a vertex index i dependent onehot vector $s_i = \{\boldsymbol{\delta}_{ij=1,\ldots,n}\}$.

$$\begin{aligned} s_x\infty^x \oplus s_y\infty^y &= \begin{cases} (s_x \cup s_y)\infty^{\max(x,y)}, & x = y \\ s_y\infty^{\max(x,y)}, & x < y \\ s_x\infty^{\max(x,y)}, & x > y \end{cases} \\ s_x\infty^x \odot s_y\infty^y &= \{\sigma \vee \tau | \sigma \in s_x, \tau \in s_y\}\infty^{x+y} \\ \mathbb{0} &= \{\}\infty^{-\infty} \\ \mathbb{1} &= \{\mathbf{0}\}\infty^0 \end{aligned} \tag{13}$$

The $\vee$ is the bit-wise `or`. One can easily check that this replacement does not change the fact that the algebra is a commutative semiring. By slightly modifying the above algebra, it can also be used to obtain just a single configuration to save computational effort. We leave this as an exercise for readers.

3 Utilizing sparsity

So far, we have used the language of einsum networks for contraction. When using sparse tropical tensor networks to find the maximum independent set, we can introduce a new rule to compress the

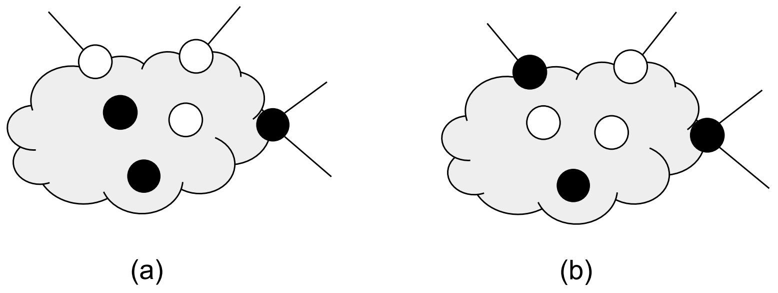

Figure 2: Two configurations with same local independent size $v_a = v_b = 3$ and different boundary configurations (a) 001 and (b) 101, where black nodes are 1s (in the independent set) and white nodes are 0s (not in the independent set).

tensor by removing elements that are not helpful. As show in Fig. 2, after we contract the tensors in a subregion $R \subseteq G$ of a graph G, and obtain a resulting tensor A of rank $|C|$, where C is the set of vertice tensors at the cut. Each tensor entry A_σ defines a local maximum independant set size with a fixed boundary configuration $\sigma \in \{0,1\}^{|C|}$ by marginalizing the inner degrees of freedom. We call this properly the *local maximum rule*.

In addition, suppose we have two entries with the same local maximum independent set size corresponding to local configurations shown in (a) and (b), it is safe to claim configuration (a) being better than configuration (b) and remove the entry A_{σ_b}. This is because the boundary of (a) is less restrictive to the rest of the graph while having the equally good local maximum independent set size, i.e. any exterior configuration $\overline{\sigma}$ making $\overline{\sigma} \cup \sigma_b$ a global maximum independent set also makes $\overline{\sigma} \cup \sigma_a$ a maximum independent set. Hence an entry A_{σ_a} is "better" than A_{σ_b} can be defined as

$$(\sigma_a \wedge \sigma_b = \sigma_a) \wedge (A_{\sigma_a} \geq A_{\sigma_b}), \tag{14}$$

where $\wedge$ is a bitwise and operations. The first term means that whenever a bit in σ_a has boolean value 1, the corresponding bit in σ_b is also 1. While the second term means the maximum independant set size with boundary configuration fixed to σ_a is not less than that fixed to σ_b. The word "better" means the best solution with boundary configuration σ_a is never worse than that with σ_b. When Eq. 14 holds, It is easy to see that if $\sigma_b \cup \overline{\sigma_b}$ is one of the solutions for maximum independant sets in G, $\sigma_a \cup \overline{\sigma_b}$ is also a solution. With this observation, it is safe to set A_{σ_b} to tropical zero. We call this property the *least restrictive rule*, that is, among boundary configurations with equal local maximum independent sizes, we only retain those least restrictive (less ones at the boundary) to exterial configurations.

3.1 The tensor network compression detects branching rules automatically

In the following, we are going to show *local maximum rule* and *least restrictive rule* can automatically discover branching rules and use it for compressing tensor elements, i.e. sparse tropical tensor network can not be worse than branching in truncating the search space.

We are going the verify the Lemmas used for branching in book (Fomin & Kaski, 2013).

***Branching Rule* 1.** *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.

***Branching Rule* 2.** *Let $G = (V, E)$ be a graph, let v and w be adjacent vertices of G such that $N[v] \subseteq N[w]$. Then*

$$\alpha(G) = \alpha(G \backslash w). \tag{15}$$

Contract $N[w]$, both $\{v, w\}$ disapear from the tensor indices because they are inner degrees of freedom. If w is one, then $N[v]$ are all zeros, the resulting tensor element can not be larger than setting $v = 1$ and $w = 0$. By the maximization rule, the local tensor does not change if we remove w.

***Branching Rule* 3.** *Let $G = (V, E)$ be a graph and let v be a vertex of G. If no maximum independent set of G contains v then every maximum independent set of G contains at least two vertices of $N(v)$.*

Contract $N[v]$, the minimum tensor element is 2, otherwise letting v be in the independent set is one of the solutions. This is again captured by the *local maximum rule*.

***Branching Rule* 4.** *Let $G = (V, E)$ be a graph and v a vertex of G. Then*

$$\alpha(G) = \max(1 + \alpha(G\backslash N[v]), \alpha(G\backslash(M(v) \cup \{v\})). \tag{16}$$

Here, $M(v)$ is the set of mirrors of v in G. A vertex $w \in N^2(v)$ is called a mirror of v if $N(v)\backslash N(w)$ is a clique. 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*). 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, the tensor compression will eliminate this solution by moving the occuppied node to the interior. Hence, the *least restrictive rule* captures the mirror rule.

***Branching Rule* 5.** *Let $G = (V, E)$ be a graph and v be a vertex of G such that $N[v]$ is a clique. Then*

$$\alpha(G) = 1 + \alpha(G\backslash N[v]). \tag{17}$$

Contract $N[v]$, and this rule can be captured by the *local maximum rule*.

***Branching Rule* 6.** *Let G be a graph, let S be a separator of G and let $I(S)$ be the set of all subsets of S being an independent set of G. Then*

$$\alpha(G) = \max_{A\in I(S)} |A| + \alpha(G\backslash(S \cup N[A])). \tag{18}$$

This rule corresponds to first contract tensors in subgraph S, then $N[S]$, and then contract the rest parts. These branching rule can be captured by the *local maximum rule*.

***Branching Rule* 7.** *Let $G = (V, E)$ be a disconnected graph and $C \subseteq V$ a connected component of G. Then*

$$\alpha(G) = \alpha(G[C]) + \alpha(G\backslash C)). \tag{19}$$

Contract by the disconnected parts, and this rule can be captured by the *local maximum rule*.

References

Gregory Matthew Ferrin. Independence polynomials. 2014.

Fedor V Fomin and Petteri Kaski. Exact exponential algorithms. *Communications of the ACM*, 56 (3):80–88, 2013.

Gene H Golub and Charles F Van Loan. *Matrix computations*, volume 3. JHU press, 2013.

Johnnie Gray and Stefanos Kourtis. Hyper-optimized tensor network contraction. *Quantum*, 5:410, Mar 2021. ISSN 2521-327X. doi: 10.22331/q-2021-03-15-410. URL http://dx.doi.org/10.22331/q-2021-03-15-410.

Nicholas J. A. Harvey, Piyush Srivastava, and Jan Vondrák. Computing the independence polynomial: from the tree threshold down to the roots, 2017.

Stefanos Kourtis, Claudio Chamon, Eduardo Mucciolo, and Andrei Ruckenstein. Fast counting with tensor networks. *SciPost Physics*, 7(5), Nov 2019. ISSN 2542-4653. doi: 10.21468/scipostphys.7.5.060. URL http://dx.doi.org/10.21468/SciPostPhys.7.5.060.

Jin-Guo Liu, Lei Wang, and Pan Zhang. Tropical tensor network for ground states of spin glasses. *Physical Review Letters*, 126(9), Mar 2021. ISSN 1079-7114. doi: 10.1103/physrevlett.126.090506. URL http://dx.doi.org/10.1103/PhysRevLett.126.090506.

Diane Maclagan and Bernd Sturmfels. *Introduction to tropical geometry*, volume 161. American Mathematical Soc., 2015. URL http://www.cs.technion.ac.il/~janos/COURSES/238900-13/Tropical/MaclaganSturmfels.pdf.

Igor L. Markov and Yaoyun Shi. Simulating quantum computation by contracting tensor networks. *SIAM Journal on Computing*, 38(3):963–981, Jan 2008. ISSN 1095-7111. doi: 10.1137/050644756. URL http://dx.doi.org/10.1137/050644756.

Cristopher Moore and Stephan Mertens. *The nature of computation*. OUP Oxford, 2011.

Feng Pan and Pan Zhang. Simulating the sycamore quantum supremacy circuits, 2021.

Sebastian Schlag. *High-Quality Hypergraph Partitioning*. PhD thesis, Karlsruhe Institute of Technology, Germany, 2020.

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
```

An example of computing the maximum independant set size

```
julia> using OMEinsum, LightGraphs, TropicalNumbers

julia> n, k = 6, 3
(6, 3)

julia> g = LightGraphs.random_regular_graph(n, k)
{6, 9} undirected simple Int64 graph

julia> ixs = [minmax(e.src,e.dst) for e in LightGraphs.edges(g)]
9-element Vector{Tuple{Int64, Int64}}:
 (1, 2)
 (1, 4)
 (1, 5)
 (2, 3)
 (2, 6)
 (3, 5)
 (3, 6)
 (4, 5)
 (4, 6)

julia> code = EinCode((ixs..., [(i,) for i in LightGraphs.vertices(g)]...), ())
1∘2, 1∘4, 1∘5, 2∘3, 2∘6, 3∘5, 3∘6, 4∘5, 4∘6, 1, 2, 3, 4, 5, 6 ->

julia> optimized_code = optimize_greedy(code, Dict([i=>2 for i=1:6]))
4∘6, 4∘6 ->
├─ 4∘6, 6 -> 4∘6
│  ├─ 6
│  └─ 4∘6
│
```

```
└─ 2∘5∘4, 5∘2∘6 -> 4∘6
   ├─ 2∘5∘6, 6∘2 -> 5∘2∘6
   │  ├─ 2∘6, 2 -> 6∘2
   │  │  ├─ 2
   │  │  └─ 2∘6
   │  └─ 2∘3, 5∘6∘3 -> 2∘5∘6
   │     ├─ 3∘5, 6∘3 -> 5∘6∘3
   │     │  ├─ 3∘6, 3 -> 6∘3
   │     │  │  ⋮
   │     │  │
   │     │  └─ 3∘5, 5 -> 3∘5
   │     │     ⋮
   │     │
   │     └─ 2∘3
   └─ 2∘4∘1, 1∘4∘5 -> 2∘5∘4
      ├─ 1∘5, 5∘4 -> 1∘4∘5
      │  ├─ 4∘5, 4 -> 5∘4
      │  │  ├─ 4
      │  │  └─ 4∘5
      │  └─ 1∘5
      └─ 2∘1, 1∘4 -> 2∘4∘1
         ├─ 1∘4
         └─ 1∘2, 1 -> 2∘1
            ├─ 1
            └─ 1∘2


julia> function mis_contract(code::OMEinsum.NestedEinsum, x::T) where T
           tensors = map(OMEinsum.getixs(Iterators.flatten(code))) do ix
               @assert length(ix) == 1 || length(ix) == 2
                           length(ix) == 1 ? [one(T), x] : [one(T) one(T); one(T) zero(T)]
           end
           code(tensors...)
       end
mis_contract (generic function with 1 method)

julia> result = mis_contract(optimized_code, Tropical(1.0))
0-dimensional Array{TropicalF64, 0}:
2.0ₜ
```

For larger graph, we hight recommend using two additional packages, `TropicalGEMM` for BLAS speed tropical matrix multiplication and `OMEinsumContractionOrders` for hyper optimized contraction order (the KaHyPar (Schlag, 2020) approach) (Kourtis et al., 2019; Gray & Kourtis, 2021; Pan & Zhang, 2021).

B Indenpendence polynomials

B.1 Common

```
using Polynomials

_auto_mispolytensor(x::T, ix::NTuple{2}) where T = T[1 1; 1 0]
_auto_mispolytensor(x::T, ix::NTuple{1}) where T = T[1, x]
function generate_polyxs!(x::T, code::OMEinsum.NestedEinsum, xs=Vector{Any}(undef, ninput(code))) where
     {T}
    for (ix, arg) in zip(OMEinsum.getixs(code.eins), code.args)
      if arg isa Integer
         xs[arg] = _auto_mispolytensor(x, ix)
      else
          generate_polyxs!(x, arg, xs)
      end
    end
   return xs
end

function generate_polyxs!(x::T, code::EinCode, xs=Vector{Any}(undef, ninput(code))) where {T}
    for (i,ix) in enumerate(OMEinsum.getixs(code))
      xs[i] = _auto_mispolytensor(x, ix)
    end
   return xs
end

function mis_polysolve(code, x::T) where {T}
```

```
   xs = generate_polyxs!(x, code, Vector{Any}(undef, NoteOnTropicalMIS.ninput(code)))
   code(xs...)
end
```

B.2 Symbolic

```
function independence_polynomial(::Val{:polynomial}, code)
    mis_polysolve(code, Polynomial([0, 1.0]))[]
end
```

B.3 Fitting and FFT

```
using FFTW
function independence_polynomial(::Val{:fft}, code; mis_size=Int(mis_solve(code)[].n), r=1.0)
   ω = exp(-2im*π/(mis_size+1))
   xs = r .* collect(ω .^ (0:mis_size))
   ys = [mis_polysolve(code, x)[] for x in xs]
   Polynomial(real.(ifft(ys) ./ (r .^ (0:mis_size))))
end

function independence_polynomial(::Val{:fitting}, code; mis_size=Int(mis_solve(code)[].n))
   xs = (0:mis_size)
   ys = [mis_polysolve(code, x)[] for x in xs]
   fit(xs, ys, mis_size)
end
```

B.4 Finite field algebra

```
using Mods, Primes

# pirate
Base.abs(x::Mod) = x
Base.isless(x::Mod{N}, y::Mod{N}) where N = mod(x.val, N) < mod(y.val, N)

function _independance_polynomial(::Type{T}, code, mis_size::Int) where T
   xs = 0:mis_size
   ys = [mis_polysolve(code, T(x))[] 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)
end

function independence_polynomial(::Val{:finitefield}, code; mis_size=Int(mis_solve(code)[].n), max_order
       =100)
    N = typemax(Int)
    YS = []
    local res
    for k = 1:max_order
       N = prevprime(N-1)
        T = Mods.Mod{N,Int}
        rk = _independance_polynomial(T, 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 improved_counting(sequences)
```

```julia
    map(yi->Mods.CRT(yi...), zip(sequences...))
end
```

C Maximum independent set

C.1 Tropical numbers

```julia
function independence_polynomial(::Val{:polynomial}, code)
    mis_polysolve(code, Polynomial([0, 1.0]))[]
end
```

C.2 Enumerating configurations

```julia
using OMEinsum, TropicalNumbers

struct ConfigEnumerator{N}
    data::Vector{BitVector}
end

Base.length(x::ConfigEnumerator{N}) where N = length(x.data)
Base.:(==)(x::ConfigEnumerator, y::ConfigEnumerator) = x.data == y.data

function Base.:+(x::ConfigEnumerator{N}, y::ConfigEnumerator{N}) where N
    res = ConfigEnumerator{N}(vcat(x.data, y.data))
    @show length(x), length(y), length(res)
    @assert length(res) == length(x) + length(y)
    return res
end

function Base.:*(x::ConfigEnumerator{L}, y::ConfigEnumerator{L}) where L
    M, N = length(x), length(y)
    z = Vector{BitVector}(undef, M*N)
    for j=1:N, i=1:M
        z[(j-1)*M+i] = x.data[i] .| y.data[j]
    end
    #@show M, N, length(z)
    @assert length(z) == length(x) * length(y)
    return ConfigEnumerator{L}(z)
end

Base.zero(::Type{ConfigEnumerator{N}}) where N = ConfigEnumerator{N}(BitVector[])
Base.one(::Type{ConfigEnumerator{N}}) where N = ConfigEnumerator{N}([falses(N)])

function onehot(::Type{ConfigEnumerator{N}}, i::Int) where N
    res = one(ConfigEnumerator{N})
    res.data[1][i] |= true
    return res
end

function enumerator_t(::Type{T}, ix::NTuple{2}, vertex_index) where {T1, N, T<:CountingTropical{T1,
      ConfigEnumerator{N}}}
    [one(T) one(T); one(T) zero(T)]
end
function enumerator_t(::Type{T}, ix::NTuple{1}, vertex_index) where {T1, N, T<:CountingTropical{T1,
      ConfigEnumerator{N}}}
    [one(T), CountingTropical(one(T1), onehot(ConfigEnumerator{N}, vertex_index[ix[1]]))]
end

symbols(::EinCode{ixs}) where ixs = unique(Iterators.flatten(ixs))
symbols(ne::NestedEinsum) = symbols(Iterators.flatten(ne))
function mis_enumerate(code)
    vertex_index = Dict([s=>i for (i, s) in enumerate(symbols(code))])
   xs = generate_xs!((T, ix)->enumerator_t(T, ix, vertex_index), CountingTropical{Float64,
        ConfigEnumerator{length(vertex_index)}}, code, Vector{Any}(undef, ninput(code)))
   code(xs...)
end
```