SOLVING THE MAXIMUM INDEPENDANT SET PROBLEM BY GENERIC PROGRAMMING 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 graph to an einsum network. By contracting the einsum network with generic element types, we show how to obtain the maximum independent set size, the independence polynomial and optimal configurations.

1 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.**] Let us map the graph G into an einsum network, as shown in Fig. 1, by placing a rank one tensor of size 2 on vertex i

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

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

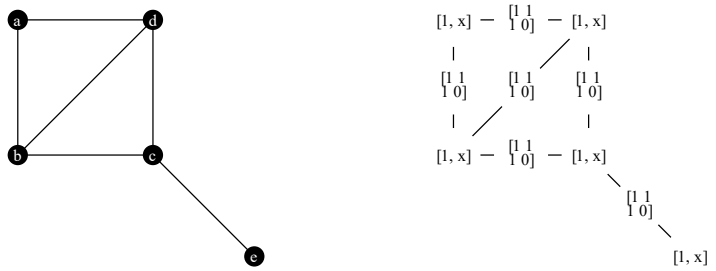
$$B_{s_i s_j} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}_{s_i s_j}, \tag{3}$$

where a tensor index s_i is a boolean variable that being 1 if vertex i is in the independent set, 0 otherwise. The contraction of this einsum network gives 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 runs over all possible vertex 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 vertex set size, while the product over edge tensors provides a factor 0 for configurations not being an independent set. With this einsum network representation, one can take the advantage of recently developed techiniques in tensor network based quantum circuit simulations (Gray & Kourtis, 2021; Pan & Zhang, 2021), 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 G. (Markov & Shi, 2008) The pairwise tensor contraction also makes it possible to utilize fast basic linear algebra subprograms (BLAS) functions for certain tensor element types.

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



(a) (b)

Figure 1: Mapping a graph to an einsum network.

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 relation must hold for arbitrary 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 rest of this paper, 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 programming generic (Stepanov & Rose, 2014).

1.1 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

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

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.

1.2 The fitting and Fourier transformation approaches

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 $x_i \in X$ and obtain the following relations

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

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

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

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

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

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 ($\omega > 1$).

1.3 The finite field algebra approach

It is possible to compute the independence polynomials exactly using 64 bit integers types only, even when the factors are larger than that can be represented by 64 bit integers. We achieve this by designing a finite field algebra $GF(p)$

$$\begin{aligned} x \oplus y &= x + y \pmod p, \\ x \odot y &= xy \pmod p, \\ \mathbb{0} &= 0, \\ \mathbb{1} &= 1. \end{aligned} \tag{9}$$

In a finite field algebra, we have the following observations

1. One can still use Gaussian elimination (Golub & Van Loan, 2013) to solve a linear equation. 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 1 to compute independent polynomial exactly without introducing space overheads. In the algorithm, except the computation of chinese remainder theorem, all computations are done with integers with fixed width W.

Algorithm 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

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

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

$$I(G, \infty) = a_k \infty^{\alpha(G)}, \tag{10}$$

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

$$\begin{aligned} a_x\infty^x \oplus a_y\infty^y &= \begin{cases} (a_x + a_y)\infty^{\max(x,y)}, & x = y \\ a_y\infty^{\max(x,y)}, & x < y \\ a_x\infty^{\max(x,y)}, & x > y \end{cases} \\ a_x\infty^x \odot a_y\infty^y &= a_x a_y \infty^{x+y} \\ \mathbb{0} &= 0\infty^{-\infty} \\ \mathbb{1} &= 1\infty^0 \end{aligned} \tag{11}$$

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 for solving 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 (Maclagan & Sturmfels, 2015; Moore & Mertens, 2011).

One may also want to obtain all ground state configurations, 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

$$\begin{aligned} s_x\infty^x \oplus s_y\infty^y &= \begin{cases} (s_x \cup s_y)\infty^{\max(x,y)}, & x = y \\ s_y\infty^{\max(x,y)}, & x < y \\ s_x\infty^{\max(x,y)}, & x > y \end{cases}, \\ s_x\infty^x \odot s_y\infty^y &= \{\sigma \vee \tau | \sigma \in s_x, \tau \in s_y\}\infty^{x+y}, \\ \mathbb{0} &= \{\}\infty^{-\infty}, \\ \mathbb{1} &= \{\mathbf{0}\}\infty^0, \end{aligned} \tag{12}$$

where $\vee$ is the bit-wise `or`. One can easily check that this replacement does not change the fact that the algebra is a commutative semiring. We first initialize the bit strings of the variable x in the vertex tensor to a vertex index i dependent onehot vector $x_i = \boldsymbol{e}_i$, then we contract the tensor network. The resulting object will give us the set of all optimal configurations. 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.

2.1 Bounding the enumeration space

When we try to implement the above algebra for enumerating configurations, we find the space overhead is larger than than we have expected. It stores more than nessesary intermedite

configurations. To speed up the computation, we use $\alpha(G)$ that much easier to compute for bounding. 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. To compute the masks, we "back propagate" the masks step by step through contraction process using the cached intermediate tensors. Consider a tropical matrix multiplication $C = AB$, we have the following inequality

$$A_{ij} \odot B_{jk} \leq C_{ik}. \tag{13}$$

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

$$A_{ij} \leq (\oplus_k (C_{ik}^{-1} \odot B_{jk}))^{-1} \tag{14}$$

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

$$\overline{A}_{ij} = \delta(A_{ij}, ((C^{\circ -1} \circ \overline{C}) B^T)_{ij}^{\circ -1}), \tag{15}$$

where $^{\circ -1}$ is the Hadamard inverse, $\circ$ is the Hadamard multiplication, 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.

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 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

$$(\sigma_a \wedge \sigma_b = \sigma_a) \wedge (A_{\sigma_a} \geq A_{\sigma_b}), \tag{16}$$

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. 16 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).

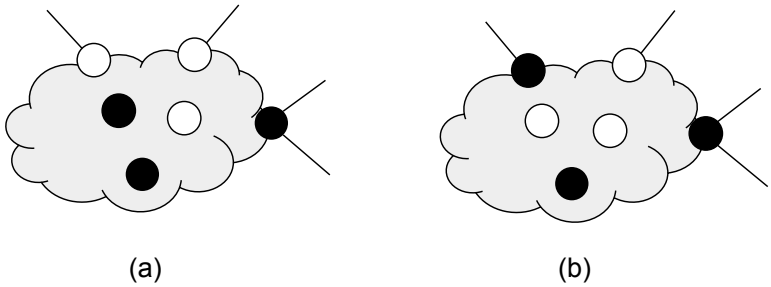


Figure 2: Two configurations with same local independent size $v_a = v_b = 3$ and different boundary configurations (a) 001 and (b) 101, where black nodes are 1s (in the independent set) and white nodes are 0s (not in the independent set).

***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*

$$\alpha(G) = \alpha(G \backslash w). \tag{17}$$

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*

$$\alpha(G) = \max(1 + \alpha(G \backslash N[v]), \alpha(G \backslash (M(v) \cup \{v\})). \tag{18}$$

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*

$$\alpha(G) = 1 + \alpha(G \backslash N[v]). \tag{19}$$

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*

$$\alpha(G) = \max_{A \in I(S)} |A| + \alpha(G \backslash (S \cup N[A])). \tag{20}$$

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*

$$\alpha(G) = \alpha(G[C]) + \alpha(G \backslash C)). \tag{21}$$

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

4 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.

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.

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.

Alexander A Stepanov and Daniel E Rose. *From mathematics to generic programming*. Pearson Education, 2014.

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.

```
using OMEinsumContractionOrders: OMEinsum
using OMEinsum, OMEinsumContractionOrders
using OMEinsum: NestedEinsum
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(ixs)) # `ixs` is input tensor labels
symbols(ne::NestedEinsum) = symbols(Iterators.flatten(ne)) # `Iterators.flatten` converts a `
     NestedEinsum` to an `EinCode`
size_dict = Dict([s=>2 for s in symbols(code)])
# optimize the contraction order using KaHyPar + Greedy, target space complexity is 2^20
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(OMEinsum.getixs(Iterators.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=100)
    N = typemax(Int)
    YS = []
    local res
    for k = 1:max_order
       N = Primes.prevprime(N-1)  # previous prime number
        # evaluate the polynomial on a finite field algebra of modulus `N`
        rk = _independance_polynomial(Mods.Mod{N,Int}, 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}` 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(OMEinsum.getixs(Iterators.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.