The supplementary material of "Computing solution space properties of combinatorial optimization problems via generic tensor networks"

Jin-Guo Liu[*] Xun Gao[†] Madelyn Cain[†]
Mikhail D. Lukin[†] Sheng-Tao Wang[‡]

June 6, 2022

Contents

[*]Department of Physics, Harvard University, Cambridge, Massachusetts 02138, USA; QuEra Computing Inc., 1284 Soldiers Field Road, Boston, MA, 02135, USA.

[†]Department of Physics, Harvard University, Cambridge, Massachusetts 02138, USA (contributed equally with Jin-Guo Liu to this work.).

[‡]QuEra Computing Inc., 1284 Soldiers Field Road, Boston, MA, 02135, USA

Performance benchmarks

We run a single thread benchmark on central processing units (CPU) Intel(R) Xeon(R) CPU E5-2686 v4 @ 2.30GHz, and its CUDA version on a GPU Tesla V100-SXM2 16G. The results are summarized in Figure 1. The graphs in all benchmarks are random three-regular graphs, which have treewidth that is asymptotically smaller than $|V|/6$ [4]. In this benchmark, we do not include traditional algorithms for finding the MIS sizes such as branching [10, 9] or dynamic programming [3, 5]. To the best of our knowledge, these algorithms are not suitable for computing most of the solution space properties mentioned in this paper. The main goal of this section is to show the relative computation time for calculating different solution space properties.

Figure 1(a) shows the time and space complexity of tensor network contraction for different graph sizes. The contraction order is obtained using the local search algorithm in Ref. [7]. If we assume our contraction-order finding program has found the optimal treewidth, which is very likely to be true, the space complexity is the same as the treewidth of the problem graph. Slicing technique [7] has been used for graphs with space complexity greater than 2^{27} (above the yellow dashed line) to fit the computation into a 16GB memory. One can see that all the computation times in figure 1 (b), (c), and (d) have a strong correlation with the predicted time and space complexity. While in panel (d), the computation time of configuration enumeration and sum-product expression tree generation also strongly correlates with other factors such as the configuration space size. Among these benchmarks, computational tasks with data types real numbers, complex numbers, or tropical numbers (CPU only) can utilize fast basic linear algebra subprograms (BLAS) functions. These tasks usually compute much faster than ones with other element types in the same category. Immutable data types with no reference to other values can be compiled to GPU devices that run much faster than CPUs in all cases when the problem scale is big enough. These data types do not include those defined in Eq. (**??**), Eq. (**??**), Eq. (**??**) and Eq. (**??**) or a data type containing them as a part. In figure 1(c), one can see the Fourier transformation-based method is the fastest in computing the independence polynomial, but it may suffer from round-off errors (Appendix). The finite field (GF(p)) approach is the only method that does not have round-off errors and can be run on a GPU. In figure 1(d), one can see the technique to bound the enumeration space in Appendix improves the performance for more than one order of magnitude in enumerating the MISs. The bounding technique can also reduce the memory usage significantly, without which the largest computable graph size is only ~ 150 on a device with 32GB main memory.

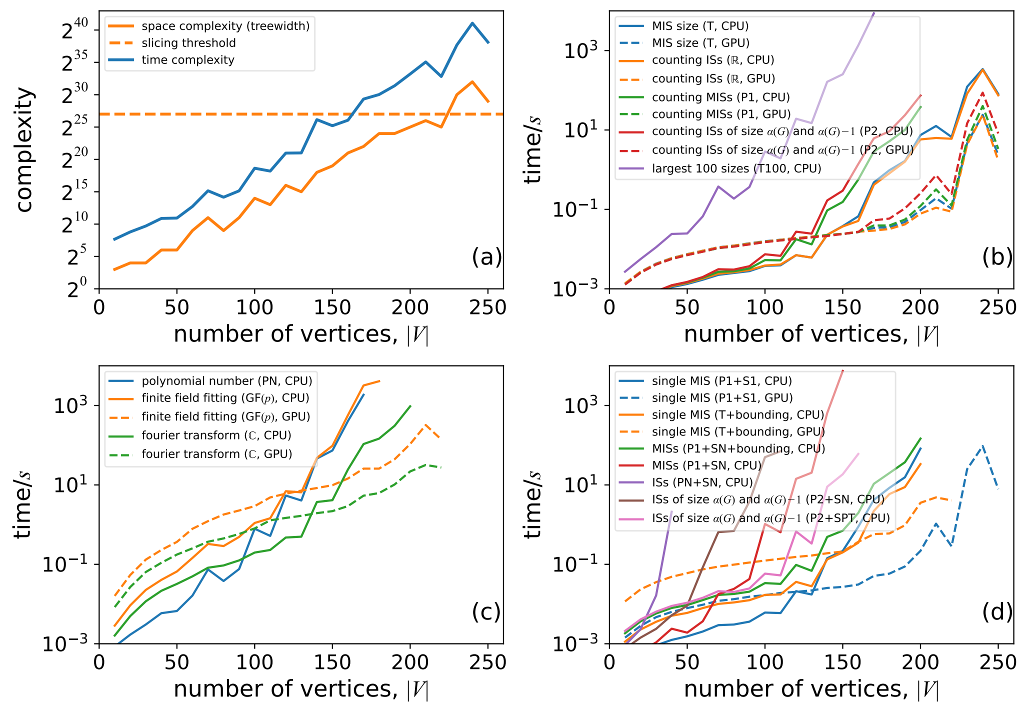


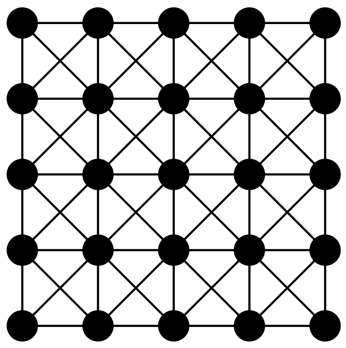
Figure 1: Benchmark results for computing different solution space properties of independent sets of random three-regular graphs with different tensor element types. The time in these plots only includes tensor network contraction, without taking into account the contraction order finding and just-in-time compilation time. Legends are properties, algebra, and devices that we used in the computation; one can find the corresponding computed solution space property in Table 1 in the main text. (a) time and space complexity versus the number of vertices for the benchmarked graphs. (b) The computation time for calculating the MIS size and for counting the number of all independent sets (ISs), the number of MISs, the number of independent sets having size $\alpha(G)$ and $\alpha(G) - 1$, and finding 100 largest set sizes. (c) The computation time for calculating the independence polynomials with different approaches. (d) The computation time for configuration enumeration, including single MIS configuration, the enumeration of all independent set configurations, all MIS configurations, all independent sets, and all independent set configurations having size $\alpha(G)$ and $\alpha(G) - 1$.

An example of increased contraction complexity for the standard tensor network notation

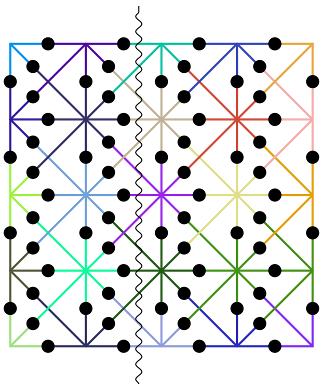
In the standard Einstein's notation for tensor networks in physics, each index appears precisely twice: either both are in input tensors (which will be summed over) or one is in an input tensor and another in the output tensor. Hence a tensor network can be represented as an open simple graph, where an input tensor is mapped to a vertex, a label shared by two input tensors is mapped to an edge and a label that appears in the output tensor is mapped to an open edge. A standard tensor network notation is equivalent to the generalized tensor network in representation power. A generalized tensor network can be converted to a standard one by adding a δ tensors at each hyperedge, where a δ tensor of rank d is defined as

$$\delta_{i_1,i_2,\ldots,i_d} = \begin{cases} 1, & i_1 = i_2 = \ldots = i_d, \\ 0, & \text{otherwise.} \end{cases} \tag{1}$$

In the following example, we will show this conversion might increase the contraction complexity of a tensor network. Let us consider the following King's graph.

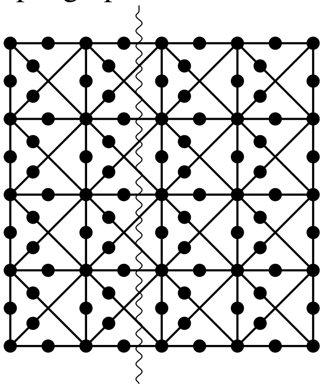


The generalized tensor network for solving the MIS problem on this graph has the following hypergraph representation, where we use different colors to distinguish different hyperedges.



Vertex tensors are not shown here because they can be absorbed into an edge tensor and hence do not change the contraction complexity. If we contract this tensor network in the column-wise order, the maximum intermediate tensor has rank $\sim L$, which can be seen by counting the number of colors at the cut.

By adding δ tensors to hyperedges, we have the standard tensor network represented as the following simple graph.



In this diagram, the additional δ tensors can have ranks up to 8. If we still contract this tensor network in a column-wise order, the maximum intermediate tensor has rank $\sim 3L$, i.e. the space complexity is $\approx 2^{3L}$, which has a larger complexity than using the generalized tensor network notation.

Bounding the MIS enumeration space

When using the algebra in Eq. (**??**) to enumerate all MISs, the program often stores significantly more intermediate configurations than necessary. To reduce the space overhead, we will show how to bound the searching space using the MIS size $\alpha(G)$. The bounded contraction consists of three stages as shown in Fig. 2. (a) We first compute the value of $\alpha(G)$ with tropical algebra and cache all intermediate tensors. (b) Then, we compute a Boolean mask for each cached tensor, where we use a Boolean `true` to represent a tensor element having a contribution to

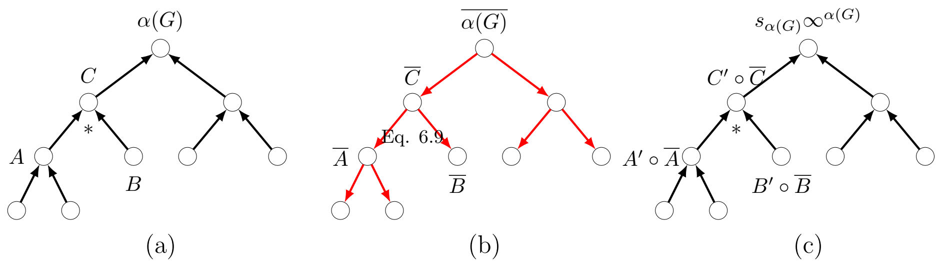


Figure 2: Bounded enumeration of maximum independent sets. Here, a circle is a tensor, an arrow specifies the execution direction of a function, $\overline{A}$ is the Boolean mask for A and $\circ$ is the Hadamard (element-wise) multiplication. (a) is the forward pass with tropical algebra (Eq. (**??**)) for computing $\alpha(G)$. (b) is the backward pass for computing Boolean gradient masks. (c) is the masked tensor network contraction with tropical algebra combined with sets (Eq. (**??**)) for enumerating configurations.

the MIS and Boolean `false` otherwise. (c) Finally, we perform masked tensor network contraction (i.e. discarding elements masked `false`) using the element type with the algebra in Eq. (**??**) to obtain all MIS configurations. The crucial part is computing the masks in step (b). Note that these masks correspond to tensor elements with non-zero gradients to the MIS size; we can compute these masks by back-propagating the gradients. To derive the back-propagation rule for tropical tensor contraction, we first reduce the problem to finding the back-propagation rule of a tropical matrix multiplication $C = AB$. Since $C_{ik} = \bigoplus_j A_{ij} \odot B_{jk} = \max_j A_{ij} \odot B_{jk}$ with tropical algebra, we have the following inequality

$$A_{ij} \odot B_{jk} \leq C_{ik}. \tag{2}$$

Here $\leq$ on tropical numbers are the same as the real-number algebra. The equality holds for some j', which means $A_{ij'}$ and $B_{j'k}$ have contributions to C_{ik}. Intuitively, one can use this relation to identify elements with nonzero gradients in A and B, but if doing this directly, one loses the advantage of using BLAS libraries [1] for high performance. Since $A_{ij} \odot B_{jk} = A_{ij} + B_{jk}$, one can move B_{jk} to the right hand side of the inequality:

$$A_{ij} \leq C_{ik} \odot B_{jk}^{\circ -1} \tag{3}$$

where $^{\circ -1}$ is the element-wise multiplicative inverse on tropical algebra (which is the additive inverse on real numbers). The inequality still holds if we take the minimum over k:

$$A_{ij} \leq \min_k (C_{ik} \odot B_{jk}^{\circ -1}) = \left(\max_k \left(C_{ik}^{\circ -1} \odot B_{jk}\right)\right)^{\circ -1} = \left(\bigoplus_k \left(C_{ik}^{\circ -1} \odot B_{jk}\right)\right)^{\circ -1} = \left(C^{\circ -1} B^{\mathsf{T}}\right)_{ij}^{\circ -1}. \tag{4}$$

On the right-hand side, we transform the operation into a tropical matrix multiplication so that we can utilize the fast tropical BLAS routines [1]. Again, the equality holds if and only if the element A_{ij} has a contribution to C (i.e. having a non-zero gradient). Let the gradient mask for C be $\overline{C}$; the back-propagation rule for gradient masks reads

$$\overline{A}_{ij} = \delta\left(A_{ij}, \left(\left(C^{\circ -1} \circ \overline{C}\right) B^{\mathsf{T}}\right)^{\circ -1}_{ij}\right), \tag{5}$$

where δ is the Dirac delta function that returns one if two arguments have the same value and zero otherwise, $\circ$ is the element-wise product, Boolean false is treated as the tropical number $\mathbb{0}$, and Boolean true is treated as the tropical number $\mathbb{1}$. This rule defined on matrix multiplication can be easily generalized to tensor contraction by replacing the matrix multiplication between $C^{\circ -1} \circ \overline{C}$ and B^{T} by a tensor contraction. With the above method, one can significantly reduce the space needed to store the intermediate configurations by setting the tensor elements masked false to zero during contraction.

The discrete Fourier transform approach to computing the independence polynomial

In Appendix **??**, we show that the independence polynomial can be obtained by solving the linear equation Eq. (**??**) using the finite field algebra. One drawback of using finite field algebra is that its matrix multiplication is less computationally efficient compared with floating-point matrix multiplication. Here, we show an alternative method with standard number types but with controllable round-off errors. Instead of choosing x_i as random numbers, we can choose them such that they form a geometric sequence in the complex domain $x_j = r\omega^j$, where $r \in \mathbb{R}$ and $\omega = e^{-2\pi i/(\alpha(G)+1)}$. The linear equation thus becomes

$$\begin{pmatrix} 1 & r & r^2 & \ldots & r^{\alpha(G)} \\ 1 & r\omega & r^2\omega^2 & \ldots & r^{\alpha(G)}\omega^{\alpha(G)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & r\omega^{\alpha(G)} & r^2\omega^{2\alpha(G)} & \ldots & r^{\alpha(G)}\omega^{\alpha(G)^2} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{\alpha(G)} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{\alpha(G)} \end{pmatrix}. \tag{6}$$

Let us rearrange the coefficients r^j to a_j, the matrix on the left side becomes the discrete Fourier transform matrix. Thus, we can obtain the coefficients by inverse Fourier transform $\vec{a}_r = \mathrm{FFT}^{-1}(\omega) \cdot \vec{y}$, where $(\vec{a}_r)_j = a_j r^j$. By choosing different r, one can obtain better precision for small j by choosing $r < 1$ or large j by choosing $r > 1$.

Computing maximum sum combination

Given two sets A and B of the same size n. It is known that the maximum n sum combination of A and B can be computed in time $O(n \log(n))$. The standard approach to solve the sum combination problem requires storing the variables in a heap — a highly dynamic binary tree structure that can be much slower to manipulate than arrays. In the following, we show an algorithm with roughly the same complexity but does not need a heap. This algorithm first sorts both A and B and then uses the bisection to find the n-th largest value in the sum combination. The key point is we can count the number of entries greater than a specific value in the sum combination of A and B in linear time. As long as the data range is not exponentially large, the bisection can be done in $O(\log(n))$ steps, giving the time complexity $O(n \log(n))$. We summarize the algorithm as in Algorithm 1.

In this algorithm, function `collect_geq` is similar the `count_geq` except the counting is replace by collecting the items to a set. Inside the function `count_geq`, variable k monotoneously increase while q monotoneously decrease in each iteration and the total number of iterations is upper bounded by $2n$. Here for simplicity, we do not handle the special element $-\infty$ in A and B and the potential degeneracy in the sums. It is nevertheless important to handle them properly in a practical implementation.

Technical guides

This appendix covers some technical guides for efficiency, including an introduction to an open-source package GenericTensorNetworks [2] implementing the algorithms in this paper and the gist about how this package is implemented. One can install `GenericTensorNetworks` in a Julia REPL, by first typing `]` to enter the `pkg>` mode and then typing

```
pkg> add GenericTensorNetworks
```

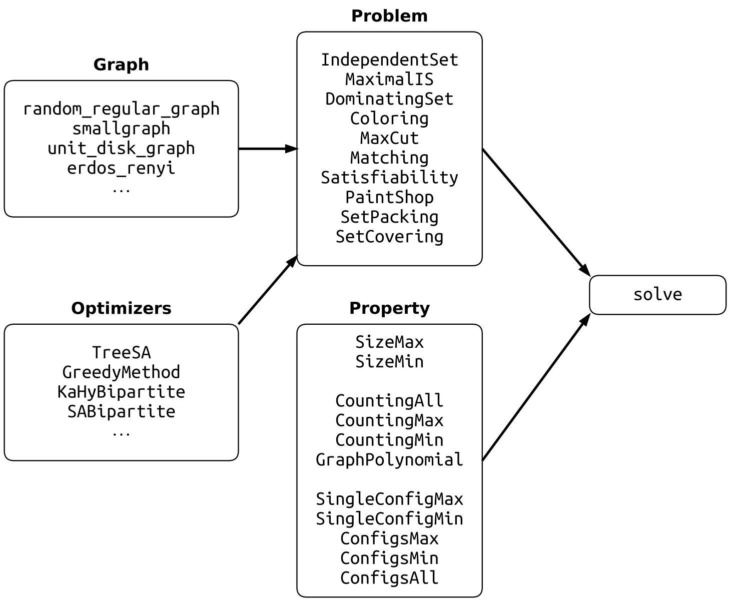
followed by an `<ENTER>` key. To use it for solving solution space properties, just go back to the normal mode (type `<BACKSPACE>`) and type

```
julia> using GenericTensorNetworks, Graphs

julia> # using CUDA

julia> solve(
           IndependentSet(
               Graphs.random_regular_graph(20, 3);
               optimizer = TreeSA(),
               weights = NoWeight(),
               openvertices = ()
           ),
           GraphPolynomial();
           usecuda=false
       )
0-dimensional Array{Polynomial{BigInt, :x}, 0}:
Polynomial(1 + 20*x + 160*x^2 + 659*x^3 + 1500*x^4 + 1883*x^5 + 1223*x^6 + 347*x^7 + 25*x
     ^8)
```

Here the main function `solve` takes three inputs: the problem instance of type `IndependentSet`, the property instance of type `GraphPolynomial` and an optional key word argument `usecuda` to decide to use GPU or not. If one wants to use GPU to accelerate the computation, "`using CUDA`" must be uncommented. The problem instance takes four arguments to initialize: the only positional argument is the graph instance one wants to solve, the keyword argument `optimizer` is for specifying the tensor network optimization algorithm, the keyword argument `weights` is for specifying the weights of vertices as either a vector or `NoWeight()`, and the keyword argument `openvertices` is for specifying the degrees of freedom not summed over. Here, we use the `TreeSA` method as the tensor network optimizer, and leave `weights` and `openvertices` as default values. The `TreeSA` algorithm, which was invented in Ref. [7], performs the best in most of our applications. The first execution of this function will be a bit slow due to Julia's just-in-time compilation. After that, the subsequent runs will be faster. The following diagram lists possible combinations of input arguments, where functions in the `Graph` are mainly defined in the package `Graphs`, and the rest can be found in `GenericTensorNetworks`.



The code we will show below is a gist of how the above package was implemented, which is mainly for pedagogical purpose. It covers most of the topics in the paper without caring much about performance. It is worth mentioning that this project depends on multiple open source packages in the Julia ecosystem:

OMEinsum and OMEinsumContractionOrders are packages providing the support for Einstein's (or tensor network) notation and state-of-the-art algorithms for contraction order optimization, which includes the one based on KaHypar+Greedy [6, 8] and the one based on local search [7].

TropicalNumbers and TropicalGEMM are packages providing tropical number and efficient tropical matrix multiplication.

Graphs is a foundational package for graph manipulation in the Julia community.

Polynomials is a package providing polynomial algebra and polynomial fitting.

Mods and the Primes package providing finite field algebra and prime number manipulation.

They can be installed in a similar way to `GenericTensorNetworks`. After installing the required packages, one can open a Julia REPL, and copy-paste the following code snippet into it.

```julia
using OMEinsum, OMEinsumContractionOrders
using Graphs
using Random

# generate a random regular graph of size 50, degree 3
graph = (Random.seed!(2); Graphs.random_regular_graph(50, 3))

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

# an einsum contraction without a contraction order specified is called `EinCode`,
# an einsum contraction having a contraction order (specified as a tree structure) is
     called `NestedEinsum`.
# assign each label a dimension-2, it will be used in the contraction order optimization
# `uniquelabels` function extracts the tensor labels into a vector.
size_dict = Dict([s=>2 for s in uniquelabels(code)])
# optimize the contraction order using the `TreeSA` method; the target space complexity
     is 2^17
optimized_code = optimize_code(code, size_dict, TreeSA())
println("time/space complexity is $(OMEinsum.timespace_complexity(optimized_code,
     size_dict))")

# a function for computing the independence polynomial
function independence_polynomial(x::T, code) where {T}
   xs = map(getixsv(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 THE MAXIMUM INDEPENDENT SET SIZE AND ITS COUNTING/DEGENERACY
     ###########

# using Tropical numbers to compute the MIS size and the 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 THE 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)[])")

########## FINDING MIS CONFIGURATIONS ###########

# define the set algebra
struct ConfigEnumerator{N}
    # NOTE: BitVector is dynamic and it can be very slow; check our repo for the static
     version
    data::Vector{BitVector}
```

```julia
end
function Base.:+(x::ConfigEnumerator{N}, y::ConfigEnumerator{N}) where {N}
    res = ConfigEnumerator{N}(vcat(x.data, y.data))
    return res
end
function Base.:*(x::ConfigEnumerator{L}, y::ConfigEnumerator{L}) where {L}
    M, N = length(x.data), length(y.data)
    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
    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)])

# the algebra sampling one of the configurations
struct ConfigSampler{N}
    data::BitVector
end

function Base.:+(x::ConfigSampler{N}, y::ConfigSampler{N}) where {N}  # biased sampling:
     return `x`
    return x  # randomly pick one
end
function Base.:*(x::ConfigSampler{L}, y::ConfigSampler{L}) where {L}
    ConfigSampler{L}(x.data .| y.data)
end

Base.zero(::Type{ConfigSampler{N}}) where {N} = ConfigSampler{N}(trues(N))
Base.one(::Type{ConfigSampler{N}}) where {N} = ConfigSampler{N}(falses(N))

# 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(uniquelabels(code))])
    N = length(vertex_index)  # number of vertices
    xs = map(getixsv(code)) do ix
        T = all ? CountingTropical{Float64, ConfigEnumerator{N}} : CountingTropical{
     Float64, ConfigSampler{N}}
        if length(ix) == 2
            return [one(T) one(T); one(T) zero(T)]
        else
            s = falses(N)
            s[vertex_index[ix[1]]] = true  # one hot vector
            if all
                [one(T), T(1.0, ConfigEnumerator{N}([s]))]
            else
                [one(T), T(1.0, ConfigSampler{N}(s))]
            end
        end
    end
   return code(xs...)
end

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

# direct enumeration of configurations can be very slow; please check the bounding
    version in our Github repo.
println("all optimal configurations are $(mis_config(optimized_code; all=true)[].c)")
```

References

[1] https://github.com/TensorBFS/TropicalGEMM.jl.

[2] https://github.com/QuEraComputing/GenericTensorNetworks.jl.

[3] B. Courcelle, *The monadic second-order logic of graphs. i. recognizable sets of finite graphs*, Information and computation, 85 (1990), pp. 12–75, https://doi.org/10.1016/0890-5401(90)90043-H.

[4] F. V. Fomin and K. Høie, *Pathwidth of cubic graphs and exact algorithms*, Information Processing Letters, 97 (2006), pp. 191–196, https://doi.org/10.1016/j.ipl.2005.10.012.

[5] F. V. Fomin and P. Kaski, *Exact exponential algorithms*, Communications of the ACM, 56 (2013), pp. 80–88, https://doi.org/10.1145/2428556.2428575.

[6] 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.

[7] G. Kalachev, P. Panteleev, and M.-H. Yung, *Multi-tensor contraction for xeb verification of quantum circuits*, 2021, https://arxiv.org/abs/2108.05665.

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

[9] J. M. Robson, *Algorithms for maximum independent sets*, Journal of Algorithms, 7 (1986), pp. 425–440, https://doi.org/10.1016/0196-6774(86)90032-5.

[10] R. E. Tarjan and A. E. Trojanowski, *Finding a maximum independent set*, SIAM Journal on Computing, 6 (1977), pp. 537–546, https://doi.org/10.1137/0206038.

Algorithm 1: Fast sum combination without using heap

```
Let $A$ and $B$ be two sets of size $n$;
// sort $A$ and $B$ in ascending order
$A \leftarrow$ sort($A$);
$B \leftarrow$ sort($B$);
// use bisection to find the $n$-th largest value in sum
   combination
high $\leftarrow A_n + B_n$;
low $\leftarrow A_1 + B_n$;
while true do
    mid $\leftarrow$ (high + low)/2;
    $c \leftarrow$ count_geq($n, A, B,$ mid);
    if $c > n$ then
        low $\leftarrow$ mid;
    else if $c = n$ then
        return collect_geq($n, A, B,$ mid);
    else
        high $\leftarrow$ mid;
    end
end
function count_geq(n, A, B, v)
    $k \leftarrow 1$;
    ; // number of entries in $A$ s.t. $a + b \geq v$
    $a \leftarrow A_n$;
    ; // the smallest entry in $A$ s.t. $a + b \geq v$
    $c \leftarrow 0$;
    ; // the counting of sum combinations s.t. $a + b \geq v$
    for $q = n, n - 1 \ldots 1$ do
        $b \leftarrow B_{n-q+1}$;
        while $k < n$ and $a + b \geq v$ do
            $k \leftarrow k + 1$;
            $a \leftarrow A_{n-k+1}$;
        end
        if $a + b \geq v$ then
            $c \leftarrow c + k$;
        else
            $c \leftarrow c + k - 1$;
        end
    end
    return c;
end
```