

Programming guide for solving constraint satisfaction problems with tensor networks

Xuanzhao Gao(高煊钊), Xiaofeng Li(李晓锋), and Jinguo Liu(刘金国)

Citation: Chin. Phys. B, 2025, 34 (**5**): 050201. DOI: 10.1088/1674-1056/adbee2
Journal homepage: http://cpb.iphy.ac.cn; http://iopscience.iop.org/cpb

What follows is a list of articles you may be interested in

Graph distillation with network symmetry

Feng Lin(林峰) and Jia-Lin He(何嘉林)
Chin. Phys. B, 2025, 34 (**4**): 040204. DOI: 10.1088/1674-1056/adbeda

Self-similarity of multilayer networks

Bing Wang(王冰), Huizhi Yu(于蕙芷), and Daijun Wei(魏代俊)
Chin. Phys. B, 2025, 34 (**1**): 010202. DOI: 10.1088/1674-1056/ad8a47

Opinion consensus incorporating higher-order interactions in individual-collective networks

Shun Ye(叶顺), Li-Lan Tu(涂俐兰), Xian-Jia Wang(王先甲), Jia Hu(胡佳), and Yi-Chao Wang(王薏潮)
Chin. Phys. B, 2024, 33 (**7**): 070201. DOI: 10.1088/1674-1056/ad3b84

Influence of random phase modulation on the imaging quality of computational ghost imaging

Chao Gao(高超), Xiao-Qian Wang(王晓茜), Hong-Ji Cai(蔡宏吉), Jie Ren(任捷), Ji-Yuan Liu(刘籍元), Zhi-Hai Yao(姚治海)
Chin. Phys. B, 2019, 28 (**2**): 020201. DOI: 10.1088/1674-1056/28/2/020201

Distance-based formation tracking control of multi-agent systems with double-integrator dynamics

Zixing Wu(吴梓杏), Jinsheng Sun(孙金生), Ximing Wang(王希铭)
Chin. Phys. B, 2018, 27 (**6**): 060202. DOI: 10.1088/1674-1056/27/6/060202

SPECIAL TOPIC — Computational programs in complex systems

Programming guide for solving constraint satisfaction problems with tensor networks

Xuanzhao Gao(高煊钊)[1,2], Xiaofeng Li(李晓锋)[1], and Jinguo Liu(刘金国)[1,†]

[1] *Hong Kong University of Science and Technology (Guangzhou), Guangzhou* 511453, *China*
[2] *Hong Kong University of Science and Technology, Hong Kong SAR, China*

(Received 30 December 2024; revised manuscript received 1 March 2025; accepted manuscript online 11 March 2025)

Constraint satisfaction problems (CSPs) are a class of problems that are ubiquitous in science and engineering. They feature a collection of constraints specified over subsets of variables. A CSP can be solved either directly or by reducing it to other problems. This paper introduces the Julia ecosystem for solving and analyzing CSPs with a focus on the programming practices. We introduce some important CSPs and show how these problems are reduced to each other. We also show how to transform CSPs into tensor networks, how to optimize the tensor network contraction orders, and how to extract the solution space properties by contracting the tensor networks with generic element types. Examples are given, which include computing the entropy constant, analyzing the overlap gap property, and the reduction between CSPs.

Keywords: tensor networks, constraint satisfaction problems, problem reductions, Julia

PACS: 02.10.Ox, 02.10.Xm, 01.50.hv

DOI: 10.1088/1674-1056/adbee2 **CSTR:** 32038.14.CPB.adbee2

1. Introduction

A constraint satisfaction problem (CSP) is a class of problems that are ubiquitous in science and engineering. These problems include, for example, the independent set problem,[1] the cutting problem,[2] the dominating set,[1] set packing,[3] set covering,[4] vertex coloring,[5,6] K-SAT,[7,8] and the vertex cover problem.[9] These problems have a wide range of applications in scheduling, logistics, wireless networks and telecommunication, and computer vision, among others.[10,11] Finding an optimum solution for these problems is typically NP-hard in the worst case.[12]

How to solve these problems efficiently is a longstanding challenge in computer science, and has become increasingly connected to physics. The performance of an algorithm is closely related to the problem size and the solution space properties or energy landscape.[13] For example, if a problem exhibits the overlap gap property, then any local search algorithm may fail to find the global optimum in sub-exponential time.[14] It also applies to quantum algorithms, for example, the quantum variational optimizer for the independent set problem can be polynomially faster than the classical algorithm when the solution space is uniformly connected.[15,16] So understanding the solution space properties is crucial for designing efficient classical and quantum algorithms. A CSP can also be addressed by reducing it to other CSPs so that solvers for these problems can also be used. In this case, the overhead of the reduction is critical for the performance as well. Efficient reduction algorithms are developed in recent years due to the rising trend in using physical systems to solve general CSPs through reductions to other problems, such as spin glass on Ising machines,[17,18] QUBO on quantum annealing processor[19] and independent set problems on King's sub-graph.[16,20,21] While problem reductions are fundamental concepts in computer science, these techniques were not widely used by physicists in the past.

To reduce the barrier for physicists to study CSPs, we introduce the Julia[22] ecosystem for reducing and analyzing CSPs. Julia is a high performance programming language designed for scientific computing. It is as easy to use as Python, but with the performance of C.[23] The main tool we will use is `GenericTensorNetworks.jl`, which is a tensor network based CSPs solution framework.[24,25] At the time of writing, its version is 4.0.1. Numerous other scientific computing software packages have been developed based on Julia, such as the `ITensors.jl`[26] for tensor network based many-body physics, the `Yao.jl`[27] for variational quantum circuits, `JuMP.jl`[28] for mathematical optimization, and `DifferentialEquations.jl`[29] for solving differential equations. The `GenericTensorNetworks.jl` enables us to extract the solution space properties by contracting the tensor networks with generic element types, where the supported solution space properties include the partition function, the solution size, the number of solutions, the solution enumeration and sampling. The framework is shown in Fig. 1. `GenericTensorNetworks.jl` is built on top of the tensor network contraction library `OMEinsum.jl` and the

†Corresponding author. E-mail: jinguoliu@hkust-gz.edu.cn

© 2025 Chinese Physical Society and IOP Publishing Ltd. All rights, including for text and data mining, AI training, and similar technologies, are reserved.

problem reduction library `ProblemReductions.jl`. The `ProblemReductions.jl` library provides a set of problem definitions and reduction interfaces for CSPs. Since it is not practical to implement all the problems and reductions in the library, it provides a flexible interface for users to define their own problems and reductions. The `OMEinsum.jl` library provides the tensor network contraction engine, with the state-of-the-art contraction order optimization technique. Its tensor network contraction supports the CUDA backend, which is powered by Julia's dynamic and generic approach for GPU programming.[30]

In the following discussion, we first introduce the CSP problems in Section 2, with a focus on the reduction between CSPs. Then we introduce the tensor network representation of CSPs in Section 3, and show how to optimize the tensor network contraction order in Section 4. Finally, we show how to extract the solution space properties by contracting the tensor networks with generic element types in Section 5. If you have a Julia REPL, you can follow the code examples in the following sections by copy-pasting the codes.

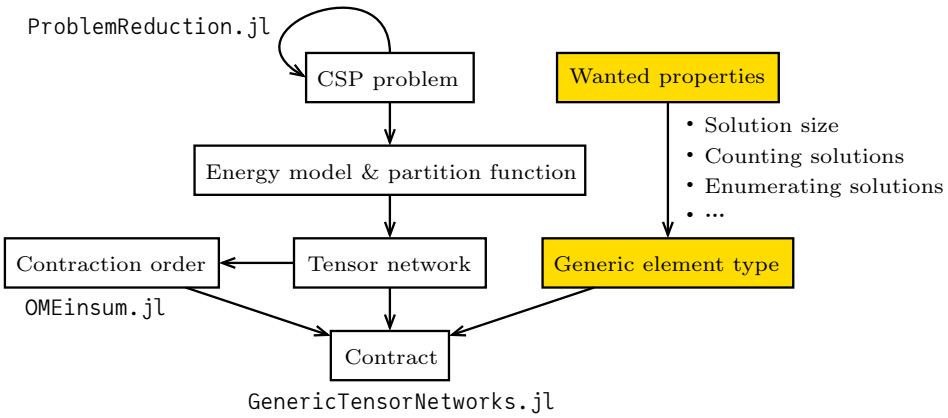


Fig. 1. The generic tensor network framework for solution space analysis of constraint satisfaction problems (CSPs).

2. Constraint satisfaction problem

Among the computationally hard problems, the constraint satisfaction problems (CSPs) are a class of problems that are closely related to physical models. `ProblemReductions.jl` provides an interface for defining CSPs and reductions between them as shown in Fig. 2.

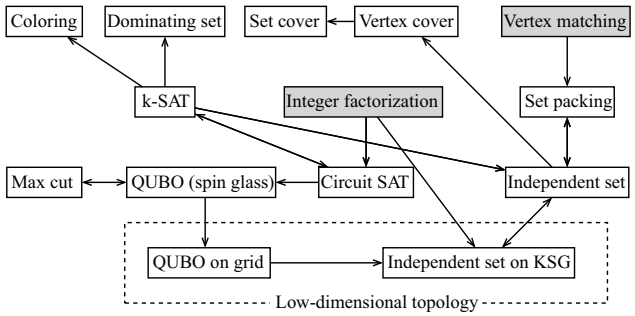


Fig. 2. A gallery of a partial list of CSPs and their reductions in `ProblemReductions.jl`. The arrows denote the reductions. The problem pointed by the tail can be reduced to the problem pointed by the head. Only the problems with gray background are not NP-complete.

In this section, we introduce the definition of some famous CSPs, and show how these problems are defined and reduced to each other in `ProblemReductions.jl`.

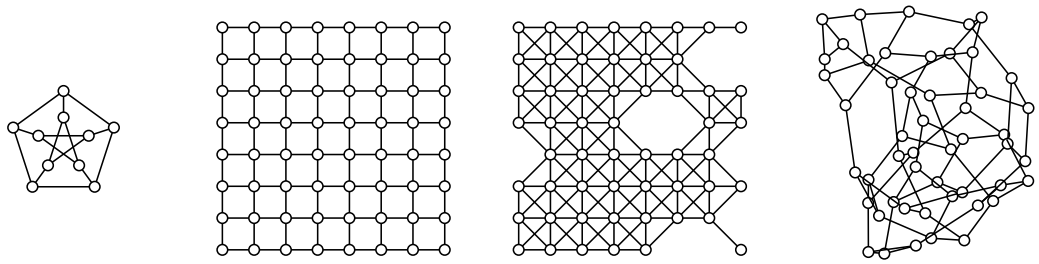
2.1. Problems definitions

The CSPs problems implemented in `ProblemReductions.jl` are listed in Fig. 2, below we will introduce these problems one by one.

2.1.1. Problems on graphs

We start by defining the graph topologies used in this work. Most CSPs introduced in this work are defined on graphs, which are not limited to simple graphs. A simple graph can be relaxed to *hyper-graphs*, where an edge can connect any number of vertices, or restricted to *unit disk graphs*, where an edge can only connect two vertices with distance less than a given threshold in a low-dimensional embedding space. To simplify the discussion, we will focus on unit disk graphs in Figs. 3(b) and 3(c), and simple graphs in Figs. 3(a) and 3(d). Their code implementation is as follows.

```
julia> using Graphs, GenericTensorNetworks
julia> petersen = Graphs.smallgraph(:petersen);
julia> square(L) = GenericTensorNetworks.random_square_lattice_graph(L, L, 1.0);
julia> ksg(L) = GenericTensorNetworks.random_diagonal_coupled_graph(L, L, 0.8);
julia> regular3(n) = Graphs.random_regular_graph(n, 3);
```



(a) Petersen graph (b) Square lattice graph (c) King's subgraph (d) 3-regular graph

Fig. 3. The types of graphs used in the partition function calculation: (a) the famous Petersen graph, (b) a square lattice graph, (c) a King's sub-graph, and (d) a random 3-regular graph, where each vertex has degree 3.

Spin glass[31,32] system is a type of disordered magnetic system that exhibits glassy behavior. The Hamiltonian of the system on a simple graph $G = (V, E)$ is given by

$$H(G, \sigma) = \sum_{(i,j)\in E} J_{ij}\sigma_i\sigma_j + \sum_{i\in V} h_i\sigma_i, \tag{1}$$

where $J_{ij} \in \mathbb{R}$ is the coupling strength between spins i and j, $h_i \in \mathbb{R}$ is the external field on spin i, and $\sigma_i \in \{-1, 1\}$ is the spin variable. The following code defines a spin glass on a Petersen graph with unit coupling strength and zero external field.

```
julia> using ProblemReductions
julia> spin_glass = SpinGlass(
           petersen,     # graph
           ones(15),     # J, in order of edges
           zeros(10)     # h, in order of vertices
       );
julia> energy(spin_glass, [0,0,0,0,0,1,1,1,1,1])
5.0
```

One can get a detailed description from the docstring of the function. Just type `?` in the Julia REPL followed by the function name, e.g., what's more, we can use `fieldnames` function to print the fields of a problem type.

```
julia> ?SpinGlass
julia> fieldnames(SpinGlass)
(:graph, :J, :h)
```

An equivalent problem to the spin glass problem is the **quadratic unconstrained binary optimization (QUBO)**,[31] which can be described as a quadratic form $f_Q(x) = x^{\mathrm{T}}Qx$, where x is a vector of binary decision variables and Q is a square matrix. The matrix Q can be interpreted as the adjacency matrix of a graph, where the diagonal elements are zero. We emphasize the constraint that elements of x are binary variables is crucial. When relaxing this constraint, the problem of finding the minimum or maximum of $f_Q(x)$ can be solved in polynomial time by finding the eigen-decomposition of Q. With this constraint, finding the minimum or maximum of $f_Q(x)$ becomes NP-hard and cannot be solved efficiently by any existing algorithm. Due to its close connection to Ising models, QUBO constitutes a central problem class for adiabatic quantum computation, where it is solved through a physical process called quantum annealing. In `ProblemReductions.jl`, we can use type `QUBO` to define the QUBO problem.

```
julia> Q = [1 1; 0 0]
2×2 Matrix{Int64}:
 1  1
 0  0

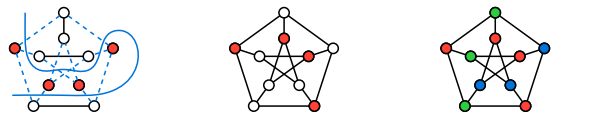
julia> QUBO(Q);  # matrix Q as argument
```

Another related NP-hard problem is the **max cut** problem,[2] which is to partition the vertices into two subsets that maximize the total weight of the edges connecting the two subsets. This problem is equivalent to the spin glass problem with positive uniform coupling strength and zero external field, and the ground state of the spin glass problem is the solution to the max cut problem. An example is given below, where a max cut problem is defined on a Petersen graph, and a solution is shown in Fig. 4(a).

```
julia> MaxCut(
           petersen,                # graph
           UnitWeight(ne(petersen)) # weights of edges
       );
```

Independent sets[1] are the subsets of vertices of a graph, where no two of which are adjacent, and the independent set problem is to find such sets of a given graph. The problem is closely related to the hard-core lattice gas model[33] in statistical physics. A maximum independent set (MIS) is an independent set of largest possible size for a given graph G, and the optimization problem of finding such a set is called the maximum independent set problem. Finding an MIS for a given graph is an NP-complete problem[34] and various exponential-time algorithms have been developed for solving it.[35] It has

garnered significant attention in recent decades due to its natural mapping to Rydberg atom-based quantum computing.[21] The following code defines the independent set problem on a Petersen graph, and a solution is shown in Fig. 4(b).



(a) Max-cut problem (b) MIS problem (c) 3-coloring problem

Fig. 4. Solutions of different problems defined on the Petersen graph: (a) the solution of the max cut problem, where the red and white vertices are the two subsets separated by the cut (blue line). The blue dashed lines are the edges across the cut. The corresponding cut size is 12. (b) The solution of the independent set problem, where the red vertices are in the independent set. The size of the maximum independent set is 4. (c) A valid solution of the vertex coloring problem. The vertices can be colored with 3 colors.

```
julia> IndependentSet(
           petersen,                # graph
           UnitWeight(nv(petersen)) # weights of vertices
       );
```

The second field is the weights of the vertices, which are set to `UnitWeight` by default, representing the unweighted version. The *size* of the independent set is the sum of the weights of its vertices. In the following, we will not explicitly state this for each problem in the rest of the paper for the sake of simplicity. A variant of the independent set problem is the **maximal independent set (MIS)** problem, where the "maximal" means that the independent set cannot be extended by adding any other vertex. Its solution space contains all the maximal independent sets, which is smaller than the solution space of the independent set problem.

```
julia> MaximalIS(
           petersen,                # graph
           UnitWeight(nv(petersen)) # weights of vertices
       );
```

Another problem that is closely related to the independent set problem is the **minimum vertex cover** problem. They are treated as the same problem in many literatures. A vertex cover is a set of vertices such that every edge in the graph has at least one endpoint in this set, and the minimum vertex cover problem is to find the smallest such set. The complement of an independent set must be a vertex cover, since an edge cannot be covered by two vertices in the independent set. Conversely, the complement of a vertex cover is an independent set. Therefore, the complement of the maximum independent set is the minimum vertex cover. In the following example, we define the minimum vertex cover problem on a Petersen graph.

```
julia> VertexCovering(
           petersen,                # graph
           UnitWeight(nv(petersen)) # weights of vertices
       );
```

Vertex coloring[5,6] refers to the problem of assigning colors to the vertices of a graph in such a way that no two adjacent vertices share the same color. The vertex coloring problem seeks to determine whether it is possible to color all vertices of a given simple graph using k colors while ensuring that adjacent vertices are not colored the same. The problem of finding the minimum number of colors required to color a given graph is one of Karp's 21 NP-complete problems.[36] One of the major applications of graph coloring is register allocation in compilers.[37] In the following example, we define the vertex coloring problem on a Petersen graph using 3 colors, and a solution is shown in Fig. 4(c).

```
julia> Coloring{3}(                 # 3 kinds of colors
           petersen,                # graph
           UnitWeight(ne(petersen)) # weights on edges
       );
```

Dominating set[1] is a problem to identify a dominating set that minimizes either the number of vertices or the total weight. A dominating set for a graph G is a subset D of its vertices such that every vertex in G is either included in D or is adjacent to at least one vertex in D. It is a classical NP-complete decision problem and has a wide application in fields such as wireless networking,[38] document summarization, and designing secure systems for electrical grids. Below is the code that defines the dominating set problem on a Petersen graph.

```
julia> DominatingSet(
           petersen,                # graph
           UnitWeight(nv(petersen)) # weights of vertices
       );
```

Matching[39] is a subset of the edges of the graph such that no two edges in the set have the same vertex, and the vertex matching problem is to find the matching with maximum number of edges or maximum total weight. Finding the maximum matching for a simple graph $G = (V, E)$ can be solved by Edmonds' blossom algorithm in $O(|V|^2|E|)$ time,[40] hence no hard problem can be reduced to the matching problem. However, the counting version of the matching problem is in the complexity class #P-complete.[41] The following code defines the matching problem on a Petersen graph.

```
julia> Matching(
           petersen,                # graph
           UnitWeight(ne(petersen)) # weights of edges
       );
```

2.1.2. Boolean satisfiability problems

Satisfiability problem[7,8] is the first problem that was proven to be NP-complete, which is to determine whether a given boolean formula is satisfiable. The satisfiability problem is a fundamental problem in computer science and has wide applications in fields such as artificial intelligence, cryptography, and automated reasoning. There is no known algorithm that can solve the SAT problem in time faster than exponential time.[42,43] In `ProblemReductions.jl`, the boolean variables, i.e., the literals, are defined by the `BoolVar` function as follows.

```
julia> x, y, z, a, b = BoolVar.(["x", "y", "z", "a", "b"]); # variables
```

With these boolean variables, a general satisfiability problem is defined as follows, where the boolean formula is given in the conjunctive normal form (CNF).

```
julia> cnf = (x ∨ y ∨ z) ∧ (a ∨ b ∨ ¬y ∨ z) ∧ (¬x ∨ ¬a)
(x ∨ y ∨ z) ∧ (a ∨ b ∨ ¬y ∨ z) ∧ (¬x ∨ ¬a)
julia> ?∨    # type `?` to enter help mode
"∨" can be typed by \vee<tab>
julia> sat_problem = Satisfiability(
           cnf,                    # CNF
           UnitWeight(length(cnf)) # weights of clauses
       );
```

If the number of literals in a clause is fixed to k, using the K-satisfiability problem is preferred. In the following example, we define a 3-SAT problem.

```
julia> ksat_problem = KSatisfiability{3}((x ∨ y ∨ z) ∧ (a ∨ b ∨ ¬y) ∧ (¬x ∨ ¬a ∨
     ¬b));
```

Circuit satisfiability (circuit SAT)[9] is a variant of the satisfiability problem, where the boolean formula is represented by a circuit. In `ProblemReductions.jl`, we can use macro expression `@circuit` to define a circuit and then use it as the input of the CircuitSAT problem.

```
julia> circuit = @circuit  begin
           c = x ∧ y
           d = x ∨ (c ∧ ¬z)
           d = true
       end
Circuit:
| c = ∧(x, y)
| d = ∨(x, ∧(c, ¬(z)))
| d = true
julia> CircuitSAT(circuit);
```

2.1.3. Other problems

Integer factorization (factoring) problem[9,44] is to find the decomposition of a positive integer into a product of integers. The corresponding decision problem is to determine whether a given integer n has a factor, other than 1, that is smaller than k, which is classified as NP-intermediate (less difficult than NP-complete but more difficult than P). Many cryptographic protocols are based on the presumed difficulty of factoring large composite integers or a related problem, for example, the RSA problem.[45] An algorithm that efficiently factors an arbitrary integer would render RSA-based public-key cryptography insecure. In `ProblemReductions.jl`, one can define the factoring problem by the `Factoring` function as follows.

```
julia> Factoring(
           2,  # the number of bits to store the first factor
           3,  # the number of bits to store the second factor
           15  # the integer to be factored
       );
```

Set cover problem[4] is defined on a set of elements and a collection of subsets, and the goal is to find the minimum number of subsets that cover all the elements. It is a hypergraph generalization of the vertex cover problem, however, its decision version is in the complexity class NP-complete.[36] An example is given below, where the set of elements is $\{1,2,3,4,5\}$ and the subsets are $\{\{1,2,3\},\{1,3,5\},\{1,2\},\{4\},\{4,5\}\}$.

```
julia> subsets = [[1,2,3],[1,3,5],[1,2],[4],[4,5]];
julia> SetCovering(
           subsets,                    # a collection of subsets
           UnitWeight(length(subsets)) # weights of subsets
       );
```

Set packing[3] is also defined on a set of elements and a collection of subsets. The goal is to find the maximum number of subsets that are pairwise disjoint, meaning that no two subsets share any common elements. It is a hypergraph generalization of the independent set problem, and its decision version is also NP-complete.[36] The following example shows how to define a set packing problem.

```
julia> sets = [[1,2,3],[1,3,5],[1,2],[4],[4,5]];
julia> SetPacking(
           sets,                    # a collection of sets
           UnitWeight(length(sets)) # weights of sets
       );
```

2.2. Reduction interfaces

In this section, we introduce the interfaces for problem reduction in `ProblemReductions.jl`. If a problem A can be reduced to problem B, then we can use the solution of B to solve A, i.e., $A \leq_p B$, meaning B is not easier than A. The reduction is not only a tool to show the hardness of a problem, but also a tool to connect different problems. For example, it allows us to solve a problem by reducing it to an Ising problem or an independent set problem on low-dimensional grid that is implementable on physical hardware, e.g., D-Wave quantum annealing processor, Ising machines[17] and Rydberg atom arrays.[16,20,21]

In Fig. 2, we show some common reductions between constraint satisfaction problems (CSPs). Since it is not practical to construct the reduction between all the CSPs, in `ProblemReductions.jl`, we only implement a finite set of reductions. A problem can be reduced to another if a reduction path exists to connect them.

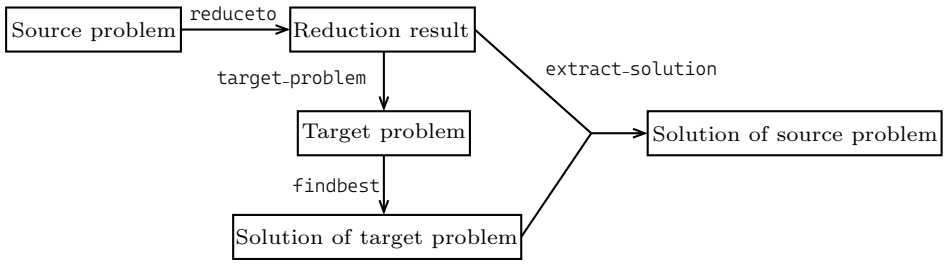


Fig. 5. The workflow of problem reduction in `ProblemReductions.jl`.

As shown in Fig. 5, a workflow of problem reduction can be separated into four parts: reduction, target problem extraction, solving, and solution extraction. In the reduction phase, we call the `reduceto` function to reduce a source problem to a target problem. The reduction result contains not only the target problem, but also intermediate information to help convert the solution back. The target problem can be extracted by the `target_problem` function. To solve it, we resort to the unified solver interface `findbest`. Finally, we can extract the solution for source problem by the `extract_solution` function. In the following example, we convert an integer factorization problem to a circuit satisfiability problem with a direct reduction rule.

```
julia> using ProblemReductions, GenericTensorNetworks
julia> factoring = Factoring(2, 3, 15);
julia> reduction_res = reduceto(CircuitSAT, factoring);  # direct reduction
julia> result = findbest(
           target_problem(reduction_res),    # target problem
           GTNSolver()                       # solver
       );
julia> solution = extract_solution(reduction_res, result[1]);
julia> ProblemReductions.read_solution(factoring,solution) # returns two factors
(3, 5)
```

Here, the `findbest` function is used to solve the circuit satisfiability problem. The built-in solver is `BruteForce`. After `GenericTensorNetworks.jl` is loaded, a tensor network based solver `GTNSolver` will also be available.

In some other cases, a direct reduction rule is not available, then a reduction path must be provided by the user. It can be constructed by the `reduction_paths` function that returns all the possible reduction paths connecting two problem types. In the following example, we convert an integer factorization problem to a spin glass problem with an automatically generated reduction path.

```
julia> all_paths = reduction_paths(Factoring, SpinGlass)
julia> reduction_res = reduceto(first(all_paths), factoring)
julia> spin_glass = target_problem(reduction_res);
julia> problem_size(spin_glass)
(num_vertices = 63, num_edges = 137)
julia> result = findbest(spin_glass, GTNSolver()); # returns all best solutions
julia> solution = extract_solution(reduction_res, result[1]);
julia> ProblemReductions.read_solution(factoring,solution) # returns two factors
(3, 5)
```

In this example, there exist multiple reduction paths from the factoring problem to the spin glass problem, which is `Factoring` → `CircuitSAT` → `SpinGlass`.[21] The reduction transforms the factoring problem into 63 vertices spin glass problems. Although the reduction is not free and requires some overhead, it is a powerful tool that enables us to fully utilize existing solvers to solve new problems.

3. Tensor network representation of constraint satisfaction problems

In previous sections, we have introduced the CSPs and the reductions between them, but a general framework to solve these problems is still missing. In this section, we introduce a tensor network representation of the constraint satisfaction problems (CSPs) to facilitate their analysis and solution. We show that both the energy model and the partition function of the CSPs can be effectively represented by tensor networks, and an example tensor network representation of the independent set problem is given.

3.1. Energy model and partition function

Let us consider a constraint satisfaction problem on a hyper-graph $G = (V, E)$, where V is the set of vertices associated with variables and E is the set of hyper-edges associated with constraints. The energy model of the problem is defined as

$$\mathcal{E}(G, \boldsymbol{s}) = \sum_{e \in E} h(e, \boldsymbol{s}), \tag{2}$$

where $\boldsymbol{s}$ is an assignment of variables and $h(e, \boldsymbol{s})$ is an energy term associated with hyper-edge $e \in E$ representing the constraint on e. Invalid configurations due to the hard constraints in some problems, e.g., the independence constraint in the independent set problem, are characterized by infinite energy penalty. The partition function for the energy model at inverse temperature β is defined as

$$Z(G, \beta) = \sum_{\boldsymbol{s}} \mathrm{e}^{-\beta \mathcal{E}(G, \boldsymbol{s})} = \sum_{\boldsymbol{s}} \prod_{e \in E} \mathrm{e}^{-\beta h(e, \boldsymbol{s})}. \tag{3}$$

Since the energy term $h(e, \boldsymbol{s})$ only involves the spins on the vertices that belong to the hyper-edge e, $\mathrm{e}^{\beta h(e, \boldsymbol{s})}$ is a tensor of rank $|e|$ and the partition function is a tensor network, where two tensors are connected if and only if they have shared variables in their corresponding hyper-edges. In the infinite temperature limit, $\beta \to 0$, the partition function is equivalent to the number of valid configurations. For certain problems, although finding one best solution is easy, counting the number of valid configurations is hard, e.g., counting the number of satisfying assignments of 2-SAT is #P-hard.[41]

3.2. Example: Tensor network representation of the independent set problem

In the following, we will show the tensor network representation of the independent set problem. Given a graph $G = (V, E)$, the energy model of the independent set problem is

$$H(G, \boldsymbol{n}) = -\sum_{v \in V} w_v n_v + \sum_{(u,v) \in E} \infty n_u n_v, \tag{4}$$

where n_v is the number of vertices in the independent set, i.e., $n_v = 1$ if v is in the independent set, and $n_v = 0$ otherwise, w_v is the weight of vertex v. The larger the size of the independent set, the lower the energy. An example is given below. The partition function at inverse temperature β can be represented as

$$Z(G, \beta) = \sum_{\boldsymbol{n}} \prod_{(u,v) \in E} \underbrace{\mathrm{e}^{-\beta \infty n_u n_v}}_{B(n_u, n_v)} \prod_{v \in V} \underbrace{\mathrm{e}^{\beta w_v n_v}}_{W(n_v)}. \tag{5}$$

It can be represented as a tensor network. For each edge $(u, v) \in E$, the pairwise interaction can be represented as a

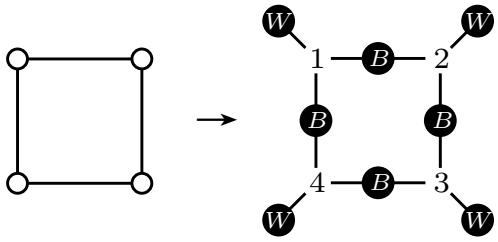
rank-2 tensor

$$B(n_u, n_v) = \begin{pmatrix} \mathbb{1} & \mathbb{1} \\ \mathbb{1} & \mathbb{0} \end{pmatrix} , \tag{6}$$

where $\mathbb{1}$ and $\mathbb{0}$ are the multiplicative identity and zero, respectively. For each vertex $v \in V$, a rank-one tensor $W(n_v)$ is associated with each vertex $v \in V$, which is given by

$$W(n_v) = \begin{pmatrix} \mathbb{1} \\ e^{\beta w_v} \end{pmatrix} . \tag{7}$$

As an example, we consider computing the partition function of a square graph with 4 vertices and 4 edges. The graph and its corresponding tensor network for the independent set problem are shown below.



The resulting tensor network is equivalent to the following sum-product form:

$$Z = \sum_{n_1,n_2,n_3,n_4} B(n_1,n_2)B(n_1,n_4)B(n_2,n_3)B(n_3,n_4)W(n_1)W(n_2)W(n_3)W(n_4) . \tag{8}$$

Using `GenericTensorNetworks`, the tensor network can be constructed automatically as follows.

```
julia> using GenericTensorNetworks, OMEinsum, Graphs
julia> problem = IndependentSet(square(2), ones(4));
julia> energy(problem, [1, 1, 1, 0])  # violation of hard constraints
Inf
julia> energy(problem, [1, 0, 0, 1])
-2.0
julia> network = GenericTensorNetwork(problem)
GenericTensorNetwork{IndependentSet{SimpleGraph{Int64}, Int64, UnitWeight}, DynamicNestedEinsum{Int64}, Int64}
- open vertices: Int64[]
- fixed vertices: Dict{Int64, Int64}()
- contraction time = 2^5.17, space = 2^2.0, read-write = 2^6.19
julia> fieldnames(typeof(network))
(:problem, :code, :fixedvertices)
julia> OMEinsum.flatten(network.code)
1∘2, 1∘4, 2∘3, 3∘4, 1, 2, 3, 4 ->
```

The resulting tensor network has 3 fields: `problem`, `code`, and `fixedvertices`. The `problem` field is the problem instance of type `IndependentSet`. The `code` field is the einsum notation defined by `OMEinsum` with optimized contraction order. Einsum notation is a compact way to represent the tensor networks topology, and the contraction order is optimized by `OMEinsum` to minimize the number of operations and memory usage. The `OMEinsum.flatten` function is used to remove the contraction order, such that we can see the underlying tensor network structure more clearly. In the flattened form, the input tensors and output tensor are separated by the arrow symbol "->". The output tensor is associated with an empty string, which means the output tensor is a scalar. The input tensors are separated by commas, and the indices of the input tensors are separated by the "∘" symbol. In this example, there are 4 input tensors of rank 2 and 4 input tensors of rank 1. The rank 2 tensors are associated with the edge tensors $B(n_u, n_v)$, and the rank 1 tensors are associated with the vertex tensors $W(n_v)$. The last field `fixedvertices` is a dictionary that stores the fixed vertices and their corresponding values. In this example, there are no fixed vertices, so the dictionary is empty.

4. Choose the right tensor network contraction order optimizer

After representing CSPs as tensor networks, we can extract the solution space properties by contracting the tensor networks with generic element types. However, contracting a tensor network can be a challenging task, since naively looping over all indices is $O(2^n)$ in time complexity, where n is the number of indices. To reduce the complexity, we need to find a good contraction order, which is the order of con-

tracting the indices. Different contraction orders lead to different complexities, while finding the optimal contraction order, i.e., the contraction order with minimal complexity, is NP-complete.[46] Luckily, a close-to-optimal contraction order is usually acceptable, which could be found in a reasonable time with a heuristic optimizer. In the past decade, methods have been developed to optimize the contraction orders, including both exact ones and heuristic ones. Among these methods, multiple heuristic methods can handle networks with more than 10^4 tensors efficiently.[47,48] In this section, we will first introduce the basic concepts and then review the methods for finding a good contraction order.

4.1. Contraction order optimization

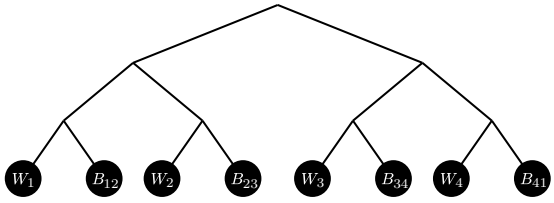
A contraction order can be represented as a rooted tree, where the leaves are the tensors to be contracted and the root is the final result. In actual calculation, we prefer binary contractions, i.e., contracting two tensors at a time, so that we can make use of BLAS[49] libraries to speed up the calculation by converting these two tensors as matrices. In this way, a given contraction order can be represented as a binary tree, where the leaves are the original tensors, the nodes are the intermediate tensors after contracting two tensors, and the root is the final result. For a given contraction order, the following three metrics are defined to describe its quality:

Time complexity (`tc`): the total number of operations.

Space complexity (`sc`): the maximum number of elements in the largest intermediate result.

Read-write complexity (`rwc`): the total number of elements to be read and written from memory.

The goal of contraction order optimization is to find a binary contraction order called the optimal contraction order, which minimizes the weighted sum of these complexity metrics. In the example shown in Subsection 3.2, the optimal contraction order is given by the following binary tree.



The largest tensor during contraction has a rank of 2, which is the smallest among all possible contraction orders. In the following, we define the unoptimized tensor network topology manually with `OMEinsum` and then show how to optimize the contraction order.

```
julia> using OMEinsum
julia> eincode = EinCode([[1, 2], [2, 3], [3, 4], [4, 1], [1], [2], [3], [4]],
    Int[])
1∘2, 2∘3, 3∘4, 4∘1, 1, 2, 3, 4 ->
julia> size_dict = uniformsize(eincode, 2)
Dict{Int64, Int64} with 4 entries:
  4 => 2
  2 => 2
  3 => 2
  1 => 2
julia> contraction_complexity(eincode, size_dict)
Time complexity: 2^4.0
Space complexity: 2^0.0
Read-write complexity: 2^4.643856189774725
```

The `EinCode` is the default constructor for specifying the tensor network topology. It takes indices for the input tensors as the first argument and the output tensor as the second argument. The `uniformsize` function is used to specify the uniform dimension of each index, which returns a dictionary mapping each index to its dimension. The `contraction_complexity` function is used to compute the time complexity, space complexity, and read–write complexity of the contraction. The time complexity is 2^4 because the default contraction naively looks over all indices without creating intermediate tensors. This leads to a large time complexity of 2^n for an n-index contraction. Moreover, the non-binary contraction does not make use of BLAS libraries, leading to a large overhead. In the following, we call the `optimize_code` function to optimize the contraction order.

The first argument of the `optimize_code` function is the `EinCode` to be optimized, the second argument is the size dictionary, and the third argument is the optimizer. The `TreeSA` is a heuristic optimizer based on the local search method. It returns a `SlicedEinCode`, which is a contraction order with sliced indices. Slicing is a technique to reduce the space complexity by looping over a subset of indices.

Here, since we did not set how many indices to slice, the first field of the `SlicedEinCode` is empty. Since the demonstrated graph is too small, the time complexity is not reduced. In the following subsection, we will introduce more methods for optimizing the contraction order. They can be used to replace the `TreeSA()` method in the above example.

```
julia> optcode = optimize_code(eincode, size_dict, TreeSA())
SlicedEinsum{Int64, DynamicNestedEinsum{Int64}}(Int64[], 1, 1 ->
├─ 4∘1, 4∘1 -> 1
│  ├─ 4∘1
│  └─ 2∘4, 1∘2 -> 4∘1
│     ├─ 3∘2, 3∘4 -> 2∘4
│     │  ├─ 2, 2∘3 -> 3∘2
│     │  │  ├─ 2
│     │  │  └─ 2∘3
│     │  └─ 4∘3, 4 -> 3∘4
│     │     ├─ 3∘4, 3 -> 4∘3
│     │     │  ⋮
│     │     └─ 4
│     └─ 1∘2
└─ 1
)
julia> contraction_complexity(optcode, size_dict)
Time complexity: 2^5.087462841250339
Space complexity: 2^2.0
Read-write complexity: 2^6.108524456778169
```

4.2. Methods for optimizing contraction order

`OMEinsum.jl` includes various methods to automatically find a good contraction order and serves as a backend of `GenericTensorNetworks.jl`. We list some methods for optimizing contraction order in Fig. 6. Two of the methods give the contraction order with the optimal space complexity, they are the exact tree-width solver and the state compression method that are implemented in `TensorOperations.jl`.[50] Both require a time exponential in the number of tensors to optimize the contraction order, thus they are not suitable for large tensor networks with more than 50 tensors. For larger networks such as those from a quantum circuit,[46] probabilistic inference problem[48] or the combinatorial optimization,[25,51] we usually resort to faster heuristic methods. There is a trade-off between the time for optimization and the quality of the contraction order. The better contraction order is usually obtained at the cost of more time to optimize the contraction order.

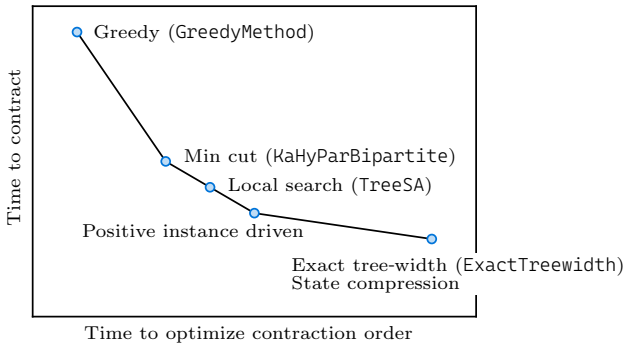


Fig. 6. The time to contract the tensor network versus the time to optimize the contraction order. The method types implemented in `OMEinsum.jl` are annotated in the parentheses.

In the following, we will introduce the methods implemented in `OMEinsum.jl` in detail.

4.2.1. Greedy method

The Greedy method is one of the simplest and fastest methods for optimizing the contraction order. The idea is to greedily select the pair of tensors with the smallest cost to contract at each step. In each step, for all possible pairs of tensors, the cost of the contraction is evaluated, which is defined as

$$\mathcal{L}(T_i, T_j) = \text{size}(T_i * T_j) - \alpha(\text{size}(T_i) + \text{size}(T_j)), \quad (9)$$

where T_i and T_j are the tensors to be contracted, $T_i * T_j$ is the intermediate tensor after contracting T_i and T_j, and α is a parameter. Then the pair with the smallest cost is selected and then contracted, which forms a new tensor. This process is repeated until all tensors are contracted. This method is fast, however it is easy to be trapped in local minima. It mainly targets the space complexity, while the time complexity is also reduced due to the smaller intermediate tensors.

A variant of the greedy method is called the hyper-greedy method,[47] where in each step one samples according to the Boltzmann distribution given by $\mathcal{P}(T_i, T_j) = \mathrm{e}^{-\mathcal{L}(T_i,T_j)/T}$ instead of directly selecting the pair with the smallest cost, where T is the temperature. If setting $T = 0$, the hyper-greedy method is equivalent to the greedy method. In this case, it is possible for the process to escape from local minima. Then the process is repeated multiple times and the best result is selected.

In `OMEinsum.jl`, the greedy method is implemented as `GreedyMethod`, with three parameters: α, `temperature`, and `nrepeat`. Their default values are set to 0.0, 0.0, and 10, respectively.

```
julia> using OMEinsum
julia> greedy = GreedyMethod(
    α = 0.0,                # the parameter of the cost function
    temperature = 0.0,      # the temperature of the hyper-greedy method
    nrepeat = 10,           # the number of trials
);
```

4.2.2. Local search method

The local search method[52] is a heuristic method based on the idea of simulated annealing. The method starts from a random contraction order and then applies the following four possible transforms as shown in Fig. 7, which correspond to the different ways to contract three sub-networks:

$$(A*B)*C = (A*C)*B = (C*B)*A,$$
$$A*(B*C) = B*(A*C) = C*(B*A),$$

where we slightly abuse the notation "$*$" to denote the tensor contraction, and A, B, C are the sub-networks to be contracted.

Due to the commutative property of the tensor contraction, such transformations do not change the result of the contraction. Even though these transformations are simple, all possible contraction orders can be reached from any initial contraction order. The local search method starts from a random contraction tree. In each step, the above rules are randomly applied to transform the tree and then the cost of the new tree is evaluated, which is defined as

$$\mathcal{L} = \mathtt{tc} + w_{\mathrm{s}}\mathtt{sc} + w_{\mathrm{rw}}\mathtt{rwc}, \tag{10}$$

where w_{s} and w_{rw} are the weights of the space complexity and read–write complexity compared to the time complexity, respectively. The optimal choice of weights depends on the specific device and tensor contraction algorithm. One can freely tune the weights to achieve the best performance for their specific problem. Then the transformation is accepted with a probability given by the Metropolis criterion, which is

$$p_{\mathrm{accept}} = \min\left(1, \mathrm{e}^{-\beta\Delta\mathcal{L}}\right), \tag{11}$$

where β is the inverse temperature, and $\Delta\mathcal{L}$ is the difference of the cost of the new and old contraction trees. During the process, the temperature is gradually decreased, and the process stops when the temperature is low enough. Additionally, the `TreeSA` method supports the slicing technique. When the space complexity is too large, one can loop over a subset of indices, and then contract the intermediate results in the end. Such a technique can reduce the space complexity, but slicing n indices will increase the time complexity by 2^n.

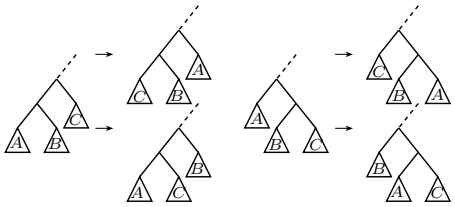


Fig. 7. The four basic local transformations on the contraction tree, which preserve the result of the contraction.

In `OMEinsum.jl`, the local search method is implemented as `TreeSA`, as shown in the following example.

```
julia> using OMEinsum
julia> treesa = TreeSA(
    sc_target = 20,          # the target space complexity
    βs = 0.01:0.05:15,       # the inverse temperatures
    ntrials = 10,            # the number of trials
    niters = 50,             # the number of iterations at each temperature
    sc_weight = 1.0,         # the relative weight of the space complexity
    rw_weight = 0.2,         # the relative weight of the read-write complexity
    initializer = :greedy,   # the initializer of the contraction order
    nslices = 0,             # the number of sliced indices
    fixed_slices = Any[],    # the manually fixed sliced indices
    greedy_config = GreedyMethod(0.0, 0.0, 1), # the method used as initializer
);
```

4.2.3. Binary partition

Binary partition is a heuristic method[47] that builds the contraction tree in a top-down manner. A tensor network can be represented as a weighted hypergraph, where the tensors are the vertices and the shared indices are the hyperedges. The weight of the hyperedge associated with index i is given by $w_i = \log_2 \mathrm{size}(i)$, where $\mathrm{size}(i)$ is the dimension of index i. Then by finding a balanced min cut on the hypergraph, we can partition the hypergraph into two parts, which represent two sub-networks to be contracted with each other. In each step, the indices associated with the cut edges together with the open indices are looped over during the contraction. Since the open indices are fixed, minimizing the total weight of the cut edges is equivalent to minimizing the time complexity of a single contraction step. Such a bipartition is repeated recursively until the resulting sub-networks are small enough to be optimized by other simpler methods. An example of the bipartition process is shown in Fig. 8. The tensor network is first partitioned into two sub-networks by cutting edges i and k. The resulting sub-networks are then further bipartitioned until each sub-network is small enough.

In the past few decades, the graph community has developed many algorithms for the balanced min cut problem and provided the corresponding software packages, such as `KaHyPar`.[53] In `OMEinsum.jl`, the binary partition based methods are implemented as `KaHyParBipartite` and `SABipartite`, which solve the balanced min cut problem by `KaHyPar` and simulated annealing method, respectively.

```
julia> using OMEinsum, KaHyPar
julia> kahyparbiparatite = KaHyParBipartite(
    sc_target = 25,               # the target space complexity
    imbalances = 0.0:0.005:0.8,   # the imbalances of the partition
    max_group_size = 40,          # the maximum size of the partition
    sub_optimizer = GreedyMethod(0.0, 0.0, 10),  # the sub-optimizer
);
julia> sabiparatite = SABipartite(
    sc_target = 25,          # the target space complexity
    ntrials = 50,            # the number of trials
    βs = 0.1:0.2:14.9,       # the inverse temperatures
    niters = 1000,           # the number of iterations at each temperature
    max_group_size = 40,     # the maximum size of the partition
    sub_optimizer = GreedyMethod(0.0, 0.0, 10),   # the sub-optimizer
    initializer = :random,   # the initializer of the contraction order
);
```

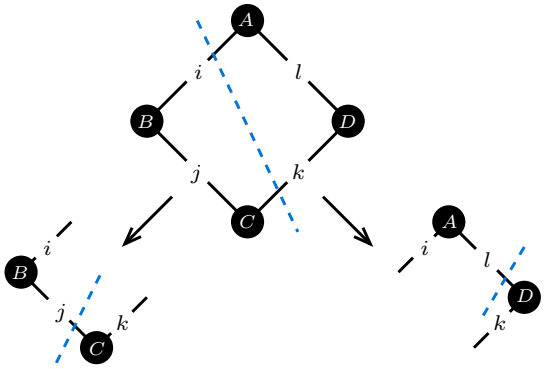


Fig. 8. An example of the recursive bipartition based tensor network contraction order optimization. The blue dashed lines represent the cuts in each step.

4.2.4. Line graph tree decomposition

Motivated by the results of Markov and Shi,[46] the contraction order can be obtained by solving the tree decomposition of the line graph of the tensor network. An index elimination order can be obtained from the tree decomposition, which gives a contraction order with contraction complexity upper bounded by the width of the tree decomposition.[47,52] Among all tree decompositions, the tree decomposition with minimal width is called the *optimal tree decomposition*, and its width is called the *treewidth*. The treewidth is an upper bound on time complexity of the single contraction step, thus the contraction order with a minimal time complexity can be derived from the optimal tree decomposition.

`OMEinsum.jl` provides an exact treewidth based method `ExactTreewidth`, which implements the Bouchitté–Todinca algorithm.[54] Solving the exact treewidth for an arbitrary graph is NP-hard and takes exponential time, hence it only works for small networks. Since the tree decomposition does not guarantee the contraction tree to be binary, a greedy method is used to convert the non-binary contractions into binary ones.

```
julia> using OMEinsum
julia> exacttreewidth = ExactTreewidth(
    greedy_config = GreedyMethod(nrepeat = 1),  # the sub-optimizer
);
```

5. Tensor networks contraction for solution space analysis

5.1. Interface

This section introduces how to use the generic tensor network for solution space analysis. The generic tensor network method serves as a unified framework for solving constraint satisfaction problems through tensor networks, by linking the desired properties of the solution space with the algebraic operations employed in tensor network contraction. Its Julia implementation is included in `GenericTensorNetworks.jl`, and the main feature is included in a single function `solve`.

```
julia> using GenericTensorNetworks, Graphs#, CUDA
julia> solve(
           GenericTensorNetwork(          # convert the CSP problem to a tensor network
               IndependentSet(            # CSP problem: the independent set problem
                   square(20),            # 20x20 square lattice
                   UnitWeight(400)        # default: uniform weight 1
               );
               optimizer = TreeSA(),      # contraction order optimizer
               openvertices = (),         # default: no open vertices
               fixedvertices = Dict()     # default: no fixed vertices
           ),
           PartitionFunction(0.0);        # wanted property: partition function at β = 0.0 (infinite temperature)
           usecuda=false                  # default: not using CUDA
       )
0-dimensional Array{Float64, 0}:
9.589790366629295e71
```

Table 1. Tensor element types and the independent set properties that can be computed using them. Every property with `Max` in name has its `Min` counterpart. The `LaurentPolynomial` is used as the return value if negative sizes are involved, e.g., in the case of spin glass. The `tree_storage` option can be used in any property for configuration enumeration for memory saving.

Property	**Elementtype&Fields**	**Description**
`SizeMax()`	`Tropical` `-n: size`	Largest solution size
`SizeMax(k)`	`ExtendedTropical{k}` `-orders : sizes`	Largest k solution sizes
`PartitionFunction(`β`)`	`Real`	Partition function at inverse temperature β
`CountingAll()`	`Real`	Number of solutions of all sizes
`GraphPolynomial()`	`Polynomial` `-coeffs : coefficients`	Number of solutions at different sizes (positive)

`GraphPolynomial(; method=:laurent)`	`LaurentPolynomial`
	`-coeffs : coefficients`
	`-order : lowestorder`
		Number of solutions at different sizes

`CountingMax()`	`CountingTropical`
	`-n: size`
	`-c: counting`
		Number of solutions with largest size

`CountingMax(k)`	`TruncatedPoly{k}`
	`-coeffs : coefficients`
	`-maxorder : highestorder`
		Number of solutions with largest k sizes

`SingleConfigMax()`	`CountingTropical{`
	`Float64,`
	`<:ConfigSampler}`
	`-n: size`
	`-c: configuration`
	`-data : vector`
		Oneconfiguration for the largest solution size

`SingleConfigMax(k)`	`ExtendedTropical{k,`
	`<:CountingTropical{`
	`Float64,`
	`<:ConfigEnumerator}}`
	`-orders : sizes`
		Oneconfiguration for each largest k sizes

`ConfigsMax()`	`CountingTropical{`
	`Float64,`
	`<:ConfigEnumerator}`
	`-n: size`
	`-c: configurations`
	`-data : vectorofvectors`
		All solutions with largest size

`ConfigsMax(k)`	`TruncatedPoly{k},`
	`<:ConfigEnumerator`
	`-orders : sizes`
		All solutions with largest k sizes

`ConfigsAll()`	`ConfigEnumerator` `-data : vectorofvectors`	All solutions

`ConfigsAll(; tree_storage=true)`	`SumProductTree`
	`-tag : node type`
	`-count : numberofsolutions`
	`-data : vector`
	`-left : leftchild`
	`-right : rightchild`
		All solutions as an expression tree

The `solve` function takes two positional arguments, one is the target CSP problem, the other is the wanted property. Here, the target problem is the unweighted independent set problem defined on a 20×20 square lattice, and the wanted property is the partition function at infinite temperature $Z(\beta = 0)$. One extra keyword argument `usecuda=false` is used to specify not using CUDA for the tensor network contraction. What is happening behind the `solve` function is shown in Fig. 1, the CSP problem is converted to an energy model and then to a tensor network. Then the contraction order is optimized. Depending on the wanted property, the tensor network is contracted with the corresponding algebra. Here, the wanted property is the partition function, which corresponds to the standard real number algebra. The full list of the properties and their associated algebra is shown in Table 1. The output is an array. Here, a 0-dimensional array represents a scalar, which corresponds to the infinite temperature partition function, or the number of all independent sets. From the output, we can see that it is an extremely large number, which is way beyond the data range of a 64-bit integer. Thus, here the output is represented as floating point numbers by default.

5.2. Tropical tensor network

In the zero temperature limit, the logarithm of the partition function (Eq. (3)) can be rewritten as

$$\lim_{\beta\to\infty} \log Z(G,\beta) = \max_{s} \sum_{e\in E} h(e,s)\,, \tag{12}$$

from which the tropical semiring algebra emerges.[25] The tropical semiring is a semiring of extended real numbers with the operations of minimum (or maximum) and addition replacing the usual operations of addition and multiplication, respectively.

By contracting the tensor network with the tropical semiring, we can get the maximum size. The Julia programming language allows us to define a new type `Tropical` to represent the tropical semiring as demonstrated in the following code.

```
julia> fieldnames(Tropical)  # fields of the Tropical number
(:n,)
julia> a, b = Tropical(2.0), Tropical(3.0)
(2.0ₜ, 3.0ₜ)
julia> a.n
2.0
julia> a + b, a * b         # `+` maps to `max`, `*` maps to `+`
(3.0ₜ, 5.0ₜ)
julia> zero(a), one(a)      # additive identity and multiplicative identity
(-Infₜ, 0.0ₜ)
julia> GenericTensorNetworks.generate_tensors(Tropical(1.0), IndependentSet(smallgraph(:petersen)))
25-element Vector{Array{Tropical{Float64}}}:
 [0.0ₜ 0.0ₜ; 0.0ₜ -Infₜ]
 [0.0ₜ 0.0ₜ; 0.0ₜ -Infₜ]
 ⋮
 [0.0ₜ, 1.0ₜ]
 [0.0ₜ, 1.0ₜ]
```

The property associated with the tropical semiring is `SizeMax`, as shown in Table 1. So we can use the following code to get the maximum size of independent sets of the Petersen graph.

```
julia> net_petersen = GenericTensorNetwork(IndependentSet(petersen))
GenericTensorNetwork{IndependentSet{SimpleGraph{Int64}, Int64, UnitWeight}, OMEinsum.
DynamicNestedEinsum{Int64}, Int64}
- open vertices: Int64[]
- fixed vertices: Dict{Int64, Int64}()
- contraction time = 2^8.0, space = 2^4.0, read-write = 2^8.704
julia> res1 = solve(net_petersen, SizeMax())[]
4.0ₜ
```

5.3. Solution space properties

Continuing from the previous section, we introduce more properties that are associated with the counting and enumeration. The rigorous definition of the relevant algebra can be found in Ref. [51]. The `SizeMax(k)` property is used for obtaining the largest k sizes of independent sets of the Petersen graph with the extended tropical semiring. It is useful for obtaining the lowest lying solutions of the weighted problems.

```
julia> res2 = solve(net_petersen, SizeMax(2))[]
ExtendedTropical{2, Tropical{Float64}}(Tropical{Float64}[4.0ₜ, 4.0ₜ])
```

The output is a vector of two identical tropical numbers, which is due to the degeneracy of the largest two sizes. The `CountingMax` property is used for counting how many maximum-size solutions there are.

```
julia> res3 = solve(net_petersen, CountingMax())[]
(4.0, 5.0)ₜ
julia> res4 = solve(net_petersen, CountingMax(2))[]
30.0*x^3 + 5.0*x^4
```

When not specifying the number of solutions, the return type is `CountingTropical` and the counting is stored in the `c` field, otherwise the return type is `TruncatedPoly` and the counting is stored in the `coeffs` field. The `CountingAll` property is used for counting all solutions.

```
julia> res5 = solve(net_petersen, CountingAll())[]
76
```

This counting property is similar to the `PartitionFunction(0.0)` property, but uses integer with arbitrary precision as the return type. The `GraphPolynomial` property is used for counting the number of solutions at different sizes. Graph polynomial is an important concept in algebraic graph theory. Popular graph polynomials include the independence polynomial,[55] the matching polynomial,[56] and the chromatic polynomial.[57] The generic tensor network method provides the `GraphPolynomial` property for CSP problems with integer sizes, which returns a polynomial that stores the number of solutions at different sizes in its coefficients.

```
julia> res6 = solve(net_petersen, GraphPolynomial())[]
Polynomial(1 + 10*x + 30*x^2 + 30*x^3 + 5*x^4)
```

The `SingleConfigMax` property is used for obtaining one solution with the largest size. The return type is `CountingTropical` and the solution is stored in the `c` field. In Julia, a type can be parameterized by another type, which is called a parameterized type. Here, `CountingTropical` is parameterized by `ConfigSampler`, which is a type that stores the solution as a binary string. The `SingleConfigMax(k)` property is used for obtaining one solution for each largest k size. The return type is `ExtendedTropical` and the solutions are stored in the `orders` field. The solutions are represented as a vector of `CountingTropical` parameterized by `ConfigSampler`.

```
julia> res7 = solve(net_petersen, SingleConfigMax())[]
(4.0, ConfigSampler{10, 1, 1}(1010000011))ₜ
julia> res8 = solve(net_petersen, SingleConfigMax(2))[]
ExtendedTropical{2, CountingTropical{Float64, ConfigSampler{10, 1, 1}}}(CountingTropical{Float64,
ConfigSampler{10, 1, 1}}[(4.0, ConfigSampler{10, 1, 1}(1001001100))ₜ, (4.0, ConfigSampler{10, 1, 1
}(0100100110))ₜ])
```

```
julia> res9 = solve(net_petersen, ConfigsMax())[]
(4.0, {0010111000, 0101010001, 1010000011, 0100100110, 1001001100})ₜ
julia> res10 = solve(net_petersen, ConfigsMax(2))[]
{0100100010, 1001001000, 0100110000, 0000111000, 0101010000, 0001011000, 1010000010, 0010100010,
 1010001000, 0010110000, 0010011000, 0010101000, 1000000011, 0100000011, 1001000001, 0001010001,
 0100010001, 0101000001, 0010000011, 1010000001, 0010010001, 1000000110, 0000100110, 0100000110,
 1001000100, 1000001100, 0100100100, 0000101100, 0101000100, 0001001100}*x^3 + {0010111000,
 0101010001, 1010000011, 0100100110, 1001001100}*x^4
julia> res11 = solve(net_petersen, ConfigsAll())[]
{1000000010, 0100100010, 0000100010, 0100000010, 0000000010, 1001001000, 1001000000, 1000001000,
1000000000, 0100110000, 0000111000, 0000110000, 0101010000, 0001011000, 0001010000, 0100010000,
0000011000, 0000010000, 0100100000, 0000101000, 0000100000, 0101000000, 0001001000, 0001000000,
0100000000, 0000001000, 0000000000, 1010000010, 0010100010, 0010000010, 1010001000, 1010000000,
0010111000, 0010110000, 0010011000, 0010010000, 0010101000, 0010100000, 0010001000, 0010000000,
1000000011, 0100000011, 0000000011, 1001000001, 1000000001, 0101010001, 0001010001, 0100010001,
0000010001, 0101000001, 0001000001, 0100000001, 0000000001, 1010000011, 0010000011, 1010000001,
0010010001, 0010000001, 1000000110, 0100100110, 0000100110, 0100000110, 0000000110, 1001001100,
1001000100, 1000001100, 1000000100, 0100100100, 0000101100, 0000100100, 0101000100, 0001001100,
0001000100, 0100000100, 0000001100, 0000000100}
```

The `ConfigsMax` property is used for obtaining all solutions with the largest size. The return type is `CountingTropical` and the solutions are stored in the `c` field, which has type `ConfigEnumerator`. The `ConfigsMax(k)` property is used for obtaining all solutions with the largest k sizes. The return type is `TruncatedPoly` and the solutions are stored in the `coeffs` field. Similarly, the `ConfigsAll` property is used for obtaining all solutions. The return type is an iterable of type `ConfigEnumerator`.

```
julia> res12 = solve(net_petersen, ConfigsAll(tree_storage=true))[]
+ (count = 76.0)
├─ + (count = 58.0)
│  ├─ + (count = 53.0)
│  ...
└─ * (count = 18.0)
julia> collect(res12)
76-element Vector{StaticBitVector{10, 1}}:
 0101000100
 0101000001
 ⋮
 0010110000
 0010100010
julia> generate_samples(res12, 3)
3-element Vector{StaticBitVector{10, 1}}:
 0101000100
 1000000110
 0010000010
```

For larger solution spaces, the `tree_storage` option can be used to save memory. The return type is `SumProductTree` and the solutions are stored in the sum product expression tree so that the memory usage is significantly reduced. To extract the solutions from the tree, we can use the `collect` function. Interestingly, you can obtain a set of unbiased samples from the tree by using the `generate_samples` function without enumerating the solutions.

This `tree_storage` option is also available for the `ConfigsMax` and `ConfigsMin` properties.

6. Applications

6.1. Hard square entropy constant

The *hard square entropy constant* is a quantity arising in statistical mechanics of hard-square lattice gases[58,59] to understand phase transitions for these systems. It is defined as $\lim_{L\to\infty} F(L)^{1/L^2}$, where $F(L)$ is the number of independent sets of a given lattice dimension $L\times L$. Since the number of independent sets of a 1-dimensional lattice of length n is the nth Fibonacci number, $F(L)$ forms a well-known integer sequence (OEIS A006506), which is thought as a two-dimensional generalization of the Fibonacci numbers. Unlike the 1D Fibonacci numbers, $F(L)$ has no known polynomial-time algorithm to compute. The following code computes the hard square entropy constant for square lattice graphs of size $L\times L$.

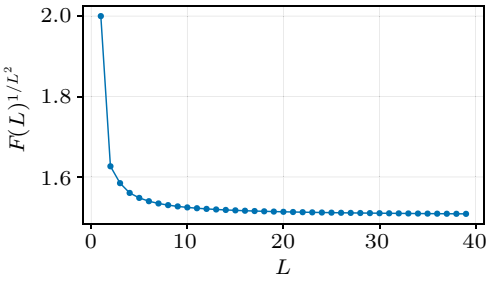


Fig. 9. The entropy constant for the square lattice graph.

```
julia> F(L) = solve(GenericTensorNetwork(IndependentSet(square(L))), PartitionFunction(0.0))[]^(1/L^2)
```

The generic tensor network method allows us to compute the hard square entropy constant for square lattice graphs of size $L\times L$ with L up to more than 39.

6.2. Configuration enumeration and sampling

The solution space analysis is crucial for designing better algorithms. This section uses the independent set problem as an example to demonstrate the overlap gap property of the solution space, and how it can be different for different types of graphs. How to characterize the hardness of a problem instance is one of the most important questions in the field of constraint satisfaction problems.

Unless the problem size is very small, enumeration of solutions at different energies is intractable. To tell if the enumeration of solutions is feasible or not, we first count the number of solutions at different energies. This task is closely related to the graph polynomials. For example, the independence polynomial of a graph G is a polynomial that counts the number of independent sets at different sizes

$$I(G,x) = \sum_{k=0}^{|\alpha(G)|} c_k x^k, \tag{13}$$

where c_k is the number of independent sets of size k and $\alpha(G)$ is the size of the largest independent set of G. If we can resolve the coefficients of the independence polynomial, we also know the number of independent sets at different sizes. In the program, we use the property name `GraphPolynomial` to denote the graph polynomial, including the independence polynomial.

```
julia> solve(GenericTensorNetwork(IndependentSet(petersen)), GraphPolynomial(; method=:finitefield))
0-dimensional Array{Polynomial{BigInt, :x}, 0}:
Polynomial(1 + 10*x + 30*x^2 + 30*x^3 + 5*x^4)
```

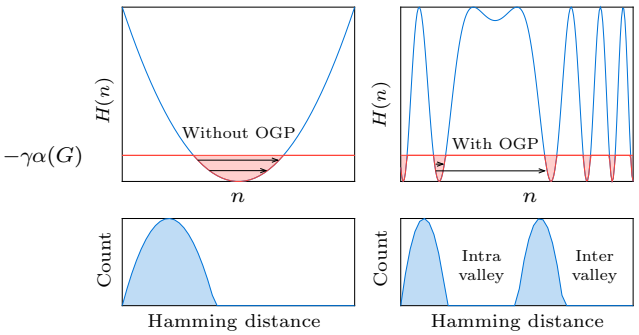


Fig. 10. The pair-wise Hamming distance distribution of solutions at different energies for graphs with and without overlap gap property.

The `GraphPolynomial` property is accessible for problems with integer sizes. It has a keyword argument `method`, which can be `:finitefield`, `:polynomial`, `:laurent`, `:fft` or `:fitting`. Here, we use the default method `:finitefield` to compute the independence polynomial of the Petersen graph, which has arbitrary precision. However, this method is unable to handle the problems with negative sizes, such as the spin glass. In such cases, we can use the `:laurent` method, which is based on the Laurent polynomial and can handle the negative polynomial orders.

We plot the number of solutions at different sizes for a 20×20 King's subgraph at filling 0.8 and a 180 vertices 3-regular graph in Figs. 11(a) and 11(b). The numbers of solutions at different sizes grow exponentially. The solutions with the largest sizes are the most important, since they are highly related to the hardness of the problem. However, even for the largest solutions, the number of solutions is still too large to be enumerated. We use the `tree_storage` option of `ConfigsMax` property to save memory. The following code generates 10000 samples from the largest two solutions of a 20×20 King's subgraph at filling 0.8 and gets the pair-wise Hamming distance distribution.

```
julia> samples = generate_samples(sum(solve(
           GenericTensorNetwork(IndependentSet(ksg(20))),
           ConfigsMax(2; tree_storage=true)
       )[].coeffs), 10000);
julia> hamming = hamming_distribution(samples, samples);
```

Similar results can be obtained for the 180 vertices 3-regular graph. The results are shown in Figs. 11(c) and 11(d). We can see that their Hamming distance distributions are very different. A random King's subgraph is more likely to have a small Hamming distance distribution, with a clear single peak. While the 3-regular graph has a more uniform distribution, with multiple peaks, which is an evidence of the overlap gap property.

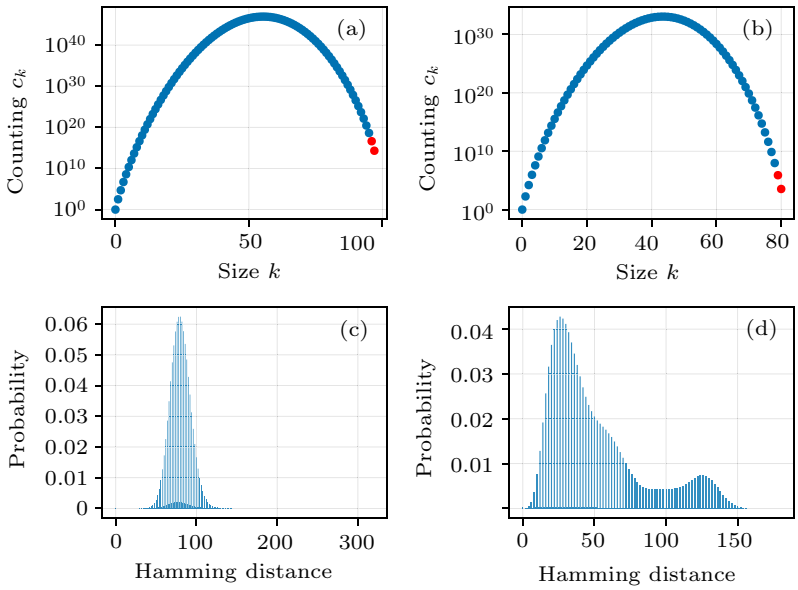


Fig. 11. The number of solutions at different sizes for (a) a 20×20 King's subgraph at filling 0.8 and (b) a 180 vertices 3-regular graph. (c) and (d) The Hamming distance distributions of the solutions with the largest sizes (red dots in (a) and (b)).

6.3. Reduce factoring to Rydberg atoms for solution space analysis

In this section, we introduce an extension of `ProblemReductions.jl`, the `UnitDiskMapping.jl` package, that contains extra reductions rules which allow users to reduce the factoring problem to the computational hard problems to that on a unit disk graph.[21] The ultimate goal is to reduce a computational hard problem to the ground state finding problem of a physical

system, such that computational hard problems can be solved by cooling a physical system to its ground state.[16] Reducing the reduction overhead and understanding the change of solution space properties are crucial for designing better physics based algorithms. To reduce the factoring problem to the weighted independent set problem on King's subgraph, the best known reduction requires $O(n^2)$ vertices,[21] where n is the number of bits. To use the extension, we just use two packages together. The following code reduces the factoring problem to the weighted independent set problem on King's subgraph and gets the overlap gap property.

```
julia> using ProblemReductions, UnitDiskMapping, GenericTensorNetworks
julia> factoring_problem = Factoring(4, 5, 221);
julia> result = reduceto(IndependentSet{ProblemReductions.GridGraph{2}, Int,
    Vector{Int}}, factoring_problem);
julia> mapped_problem = target_problem(result)
IndependentSet{ProblemReductions.GridGraph{2}, Int64, Vector{Int64}}(
    ProblemReductions.GridGraph{2}([(14, 1), (16, 1), (32, 1), (34, 1), (50, 1)
    , (52, 1), (18, 2), (36, 2), (54, 2), (6, 3)   ...  (49, 56), (50, 56), (60,
     56), (62, 56), (67, 56), (68, 56), (6, 58), (24, 58), (42, 58), (60, 58)],
     2.8567113959936523), [2, 2, 2, 2, 2, 2, 2, 2, 2, 1   ...  3, 3, 3, 3, 3, 3,
     1, 1, 1, 1])
julia> problem_size(mapped_problem)
(num_vertices = 740, num_edges = 1559)
julia> config = findbest(mapped_problem, GTNSolver());
julia> result = ProblemReductions.read_solution(factoring_problem,
    extract_solution(result, first(config)))
(13, 17)
```

One can easily verify that $13 \times 17 = 221$.

```
julia> samples = generate_samples(sum(solve(GenericTensorNetwork(mapped_problem)
    , ConfigsMax(2; tree_storage=true))[].coeffs), 10000);
```

From the generated samples, we can visualize the Hamming distance distribution of the solutions with the largest 2 sizes in Fig. 12. It turns out that the Hamming distance distribution has multiple peaks, which is an evidence of the overlap gap property.

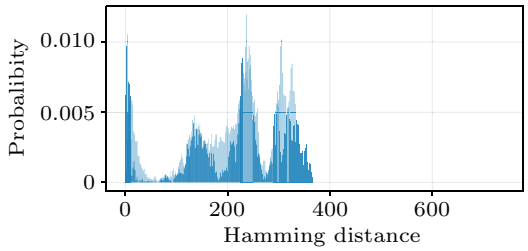


Fig. 12. The Hamming distance distribution for the maximum 2 solutions of the independent set problem reduced from the factoring problem.

6.4. Ground state degeneracy of the Buckyball structure

As a final example, we show how to use generic tensor network method to conquer the challenge problem released in the Song Shan Lake Spring School 2019. The problem statement is as follows.

Problem 1 In the Buckyball structure (fullerene) illustrated in Fig. 13, we assign an Ising spin to each vertex, with neighboring spins interacting through an antiferromagnetic coupling of unit strength. (a) Get $\ln(Z)/N$, where N is the number of vertices, and Z is the partition function at temperature 1.0. (b) Count the ground state degeneracy.

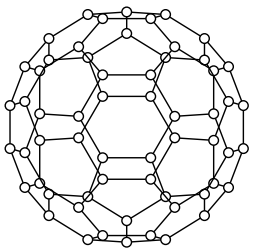


Fig. 13. The Buckyball structure (fullerene).

This problem can be easily solved by using the `GenericTensorNetworks.jl` package as follows.

```
julia> using GenericTensorNetworks, Graphs, ProblemReductions
julia> function fullerene()  # construct the fullerene graph in 3D space
           th = (1+sqrt(5))/2
           res = NTuple{3,Float64}[]
           for (x, y, z) in ((0.0, 1.0, 3th), (1.0, 2 + th, 2th), (th, 2.0, 2th + 1.0))
               for (a, b, c) in ((x,y,z), (y,z,x), (z,x,y))
                   for loc in ((a,b,c), (a,b,-c), (a,-b,c), (a,-b,-c), (-a,b,c), (-a,b,-c), (-a,-b,c), (-a,-b,-c))
                       if loc not in res
                           push!(res, loc)
                       end
                   end
               end
           end
           return res
       end
fullerene (generic function with 1 method)
julia> fullerene_graph = UnitDiskGraph(fullerene(), sqrt(5)); # construct the unit disk graph
julia> spin_glass = SpinGlass(fullerene_graph, UnitWeight(ne(fullerene_graph)), zeros(Int, nv(fullerene_graph)));
julia> problem_size(spin_glass)
(num_vertices = 60, num_edges = 90)
julia> log(solve(spin_glass, PartitionFunction(1.0))[])/nv(fullerene_graph)
1.30736845776607942
julia> solve(spin_glass, CountingMin())[]
(-66.0, 16000.0)ₜ
```

The `UnitDiskGraph` type is a type that represents the unit disk graph, which is a graph that each vertex is a point in Euclidean space. Two vertices are connected by an edge if the distance between them is less than or equal to the given radius. By solving the `PartitionFunction` and `CountingMin` properties, we find that the logarithm of the partition function per vertex is ~ 1.30737 and the ground state degeneracy is 16000, which is consistent with the expected results.

7. Conclusion and future work

In this paper, we introduce the Julia ecosystem for solving constraint satisfaction problems with tensor networks, including the reduction between constraint satisfaction problems, the tensor network representation of constraint satisfaction problems, the tensor network contraction order optimizer, and the solution space analysis. The example codes are available at https://github.com/ArrogantGao/tensornetwork4csp. Source code is available in the following GitHub repositories.

`GenericTensorNetworks.jl`: https://github.com/QuEraComputing/GenericTensorNetworks.jl
`OMEinsum.jl`: https://github.com/under-Peter/OMEinsum.jl
`ProblemReductions.jl`: https://github.com/GiggleLiu/ProblemReductions.jl

Questions and contributions are welcome in the form of issues and pull requests. In the future, we will continue to improve the performance of the tensor network contraction, and to extend the tensor network method to more constraint satisfaction problems through problem reduction. At the same time, we have to warn the readers that the tensor network method is usually not the best choice when users only need a single solution, especially for a problem with a high-dimensional graph topology. The branching algorithm works better as an exact solver in many cases.[60,61] While for approximate solvers, local search and evolutionary algorithms[62] can be more efficient. The generic tensor network method is designed for counting the number of solutions and analyzing the solution space properties.

Acknowledgements

We thank Chen-Guang Guan for actively contributing to the package `ProblemReductions.jl` in the open source promotion plan 2024. We thank Andreas Peter and other contributors for their valuable contributions to `OMEinsum.jl`

and `GenericTensorNetworks.jl` ecosystem. This work is partially funded by the National Key R&D Program of China (Grant No. 2024YFE0102500), the National Natural Science Foundation of China (Grant No. 12404568), the Guangzhou Municipal Science and Technology Project (Grant No. 2023A03J00904), the Quantum Science Center of Guangdong–Hong Kong–Macao Greater Bay Area, China and the Undergraduate Research Project from HKUST (Guangzhou).

References

[1] Clark B N, Colbourn C J and Johnson D S 1991 *Annals of Discrete Mathematics* **48** 165
[2] Ding C H, He X, Zha H, Gu M and Simon H D 2001 *Proceedings 2001 IEEE international conference on data mining* (IEEE) pp. 107–114
[3] Crescenzi P, Kann V and Halldórsson M 1995 *A Compendium of NP Optimization Problems*
[4] Chvatal V 1979 *Mathematics of Operations Research* **4** 233
[5] Jensen T R and Toft B 2011 *Graph Coloring Problems* (John Wiley & Sons)
[6] Malaguti E and Toth P 2010 *International Transactions in Operational Research* **17** 1
[7] Biere A, Heule M and van Maaren H 2009 *Handbook of Satisfiability* Vol. 185 (IOS press)
[8] Schaefer T J 1978 *Proceedings of the tenth annual ACM symposium on Theory of computing* pp. 216–226
[9] Moore C and Mertens S 2011 *The Nature of Computation* (Oxford University Press)
[10] Butenko S and Pardalos P M 2003 *Maximum Independent Set and Related Problems, with Applications* PhD thesis (University of Florida)
[11] Wu Q and Hao J K 2015 *European Journal of Operational Research* **242** 693
[12] Hastad J 1996 *Proceedings of 37th Conference on Foundations of Computer Science* (IEEE) pp. 627–636
[13] Mézard M, Parisi G, Sourlas N, Toulouse G and Virasoro M 1984 *Journal de Physique* **45** 843
[14] Gamarnik D 2021 *Proc. Natl. Acad. Sci. USA* **118** e2108492118
[15] Cain M, Chattopadhyay S, Liu J G, Samajdar R, Pichler H and Lukin M D 2023 arXiv:2306.13123
[16] Ebadi S, Keesling A, Cain M, Wang T T, Levine H, Bluvstein D, Semeghini G, Omran A, Liu J G, Samajdar R, *et al.* 2022 *Science* **376** 1209
[17] Mohseni N, McMahon P L and Byrnes T 2022 *Nature Reviews Physics* **4** 363
[18] Lucas A 2014 *Frontiers in Physics* **2** 5
[19] Ushijima-Mwesigwa H, Negre C F A and Mniszewski S M 2017 arXiv:1705.03082
[20] Pichler H, Wang S t, Zhou L, Choi S and Lukin M D 2018 *Science* **376** 6598
[21] Nguyen M T, Liu J G, Wurtz J, Lukin M D, Wang S T and Pichler H 2023 *PRX Quantum* **4** 010316
[22] Bezanson J, Karpinski S, Shah V B and Edelman A 2012 arXiv:1209.5145
[23] Bezanson J 2015 *Why is Julia fast? Can it be faster?* (JuliaCon India)
[24] Liu J G, Wang L and Zhang P 2021 *Phys. Rev. Lett.* **126** 090506
[25] Liu J G, Gao X, Cain M, Lukin M D and Wang S T 2023 *SIAM Journal on Scientific Computing* **45** A1239
[26] Fishman M, White S and Stoudenmire E 2022 *SciPost Physics Codebases*
[27] Luo X Z, Liu J G, Zhang P and Wang L 2020 *Quantum* **4** 341
[28] Lubin M, Dowson O, Dias Garcia J, Huchette J, Legat B and Vielma J P 2023 *Mathematical Programming Computation* **15** 581
[29] Rackauckas C and Nie Q 2017 *Journal of Open Research Software* **5** 15
[30] Besard T, Foket C and Sutter B D 2017 arXiv:1712.03112
[31] Glover F, Kochenberger G and Du Y 2018 arXiv:1811.11538
[32] Qiu X, Zoller P and Li X 2020 *PRX Quantum* **1** 020311
[33] Scott A D and Sokal A D 2005 *J. Stat. Phys.* **118** 1151
[34] Tarjan R E and Trojanowski A E 1977 *SIAM Journal on Computing* **6** 537
[35] Xiao M and Nagamochi H 2013 *Theoretical Computer Science* **469** 92
[36] Karp R M 2010 *Reducibility Among Combinatorial Problems* (Springer)
[37] Chaitin G J 1982 *ACM Sigplan Notices* **17** 98
[38] Stojmenovic I, Seddigh M and Zunic J 2002 *IEEE Transactions on Parallel and Distributed Systems* **13** 14
[39] Lovász L and Plummer M D 2009 *Matching Theory* Vol. 367 (American Mathematical Soc.)
[40] Edmonds J 1965 *Canadian Journal of Mathematics* **17** 449
[41] Valiant L G 1979 *SIAM Journal on Computing* **8** 410
[42] Baumgartner P *et al.* 2002 *AI in the new Millenium, Morgan Kaufmann, Seattle*
[43] Biere A, Heule M, van Maaren H and Walsh T 2009 *Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications* pp. 131–153
[44] Montgomery P L 1994 *CWI Quarterly* **7** 337
[45] Mumtaz M and Ping L 2019 *Journal of Discrete Mathematical Sciences and Cryptography* **22** 9
[46] Markov I L and Shi Y 2008 *SIAM Journal on Computing* **38** 963
[47] Gray J and Kourtis S 2021 *Quantum* **5** 410
[48] Roa-Villescas M, Gao X, Stuijk S, Corporaal H and Liu J G 2024 *Phys. Rev. Res.* **6** 033261
[49] Lawson C L, Hanson R J, Kincaid D R and Krogh F T 1979 *ACM Transactions on Mathematical Software (TOMS)* **5** 308
[50] Lukas Devos Maarten Van Damme J H and Contributors 2023 arXiv:2406.07052
[51] Liu J G, Wurtz J, Nguyen M T, Lukin M D, Pichler H and Wang S T 2024 *unpublished*
[52] Kalachev G, Panteleev P and Yung M H 2021 arXiv:2108.05665
[53] Schlag S, Heuer T, Gottesbüren L, Akhremtsev Y, Schulz C and Sanders P 2023 *ACM Journal of Experimental Algorithmics* **27** 1
[54] Bouchitté V and Todinca I 2001 *SIAM Journal on Computing* **31** 212
[55] Ferrin G M 2014 https://scholarcommons.sc.edu/etd/2609/
[56] Farrell E J 1979 *Journal of Combinatorial Theory, Series B* **27** 75
[57] Read R C 1968 *Journal of Combinatorial Theory* **4** 52
[58] Baxter R J, Enting I G and Tsang S K 1980 *J. Stat. Phys.* **22** 465
[59] Pearce P A and Seaton K A 1988 *J. Stat. Phys.* **53** 1061
[60] Fomin F V and Kaski P 2013 *Communications of the ACM* **56** 80
[61] Gao X Z, Wang Y J, Zhang P and Liu J G 2024 arXiv:2412.07685
[62] Lamm S, Sanders P, Schulz C, Strash D and Werneck R F 2017 *J. Heuristics* **23** 207