

```
1 begin
2     using GenericTensorNetworks, PlutoUI, WGLMakie
3     # `show_graph` and `show_gallery` are for visualizing graphs
4     using GenericTensorNetworks.LuxorGraphPlot: show_graph, show_gallery
5     # Graphs is a package for graph operations
6     import GenericTensorNetworks.Graphs
7 end
```

# Tensor networks for computational hard problems

---

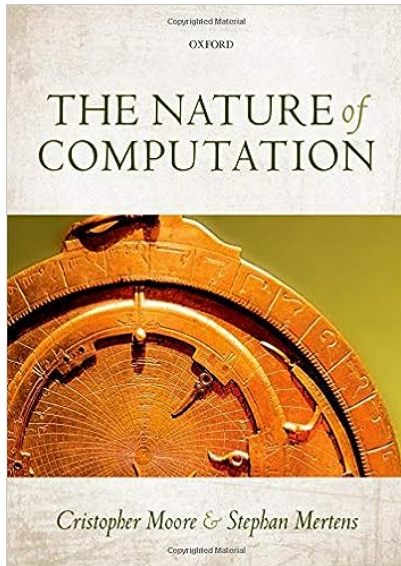
This notebook could be downloaded [here](#).

Jinguo Liu ([Hong Kong University of Science and Technology \(Guangzhou\)](#), [Function hub](#), [Advanced material thrust](#))

[jinguoliu@hkust-gz.edu.cn](mailto:jinguoliu@hkust-gz.edu.cn) (PhD applications welcomed)

- [Liu J G, Gao X, Cain M, et al. Computing solution space properties of combinatorial optimization problems via generic tensor networks\[J\]. SIAM Journal on Scientific Computing, 2023, 45\(3\): A1239-A1270.](#)

To get a systematic understanding of computational complexity theory and its relation with statistical physics.



The Nature of Computation  
By Christopher Moore & Stephan Mertens

**Section 5**

Who is the hardest one of All?  
NP-Completeness

**Section 13**

Counting, sampling and statistical physics

## Contents of this lecture

---

1. Defining hard-core lattice gases.
2. The solution space properties of hard-core lattice gases.
3. The statistical properties (at finite temperature) of hard-core lattice gases.
4. Given a hard-core lattice gas ground state solving oracle, the integer factoring problem can be solved efficiently.

## Hard Core Lattice Gas - A Thorough Guide

---

Hard-core lattice gas refers to a model used in statistical physics to study the behavior of particles on a lattice, where the particles are subject to an exclusion principle known as the "hard-core" interaction that characterized by a blockade radius. Distances between two particles can not be smaller than this radius.

- Nath T, Rajesh R. Multiple phase transitions in extended hard-core lattice gas models in two dimensions[J]. Physical Review E, 2014, 90(1): 012120.
- Fernandes H C M, Arenzon J J, Levin Y. Monte Carlo simulations of two-dimensional hard core lattice gases[J]. The Journal of chemical physics, 2007, 126(11).

Let define a  $10 \times 10$  triangular lattice, with unit vectors

$$\vec{a} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

$$\vec{b} = \begin{pmatrix} \frac{1}{2} \\ \frac{\sqrt{3}}{2} \end{pmatrix}$$

```
((1, 0), (0.5, 0.866025))
```

```
1 a, b = (1, 0), (0.5, 0.5*sqrt(3))
```

```
(10, 10)
```

```
1 Na, Nb = 10, 10
```

```
filling_factor = 1.0
```

```
1 filling_factor = 1.0
```

```
sites =
```

```
[(1.5, 0.866025), (2.5, 0.866025), (3.5, 0.866025), (4.5, 0.866025), (5.5, 0.866025), (6.5
```

```
1 # triangle lattice site locations
2 sites = [a .* i .+ b .* j for i=1:Na, j=1:Nb][rand(Na * Nb) .< filling_factor]
```

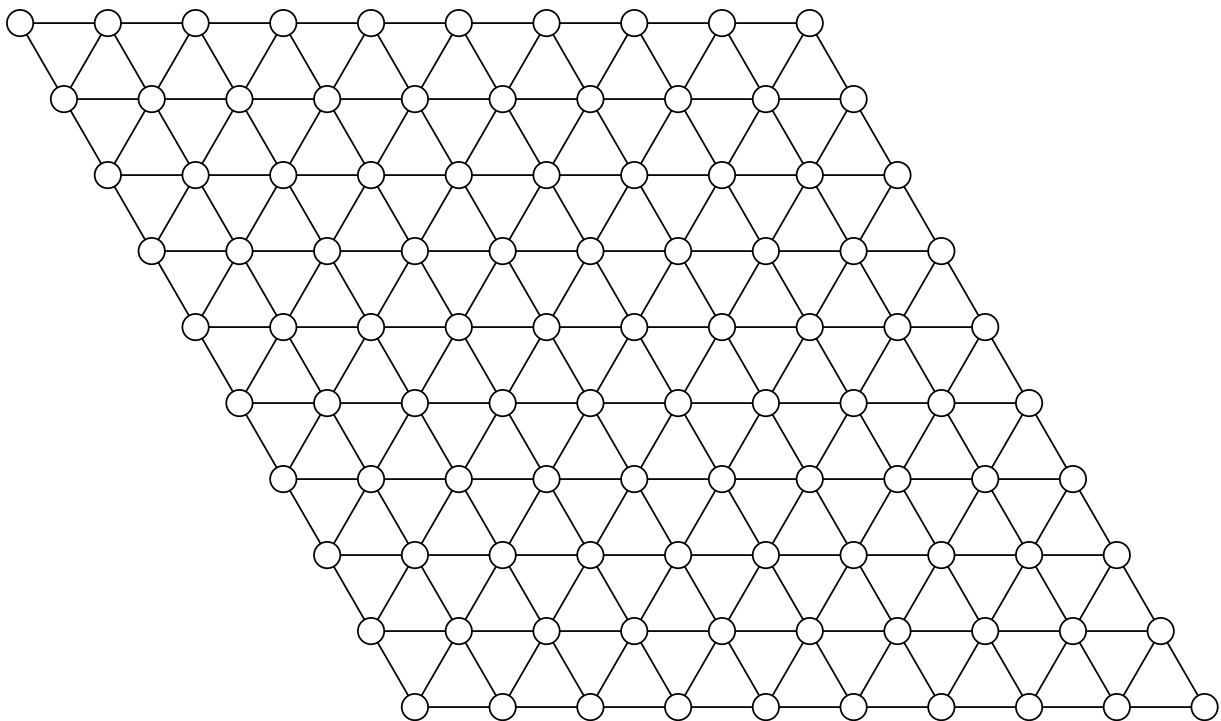
Since triangular lattice is a unit-disk graph, we can generate the graph topology with the location information and the blockade radius.

```
blockade_radius = 1.1
```

```
1 blockade_radius = 1.1
```

```
graph = {100, 261} undirected simple Int64 graph
```

```
1 graph = unit_disk_graph(vec(sites), blockade_radius)
```



```
1 show_graph(graph; locs=sites, texts=fill("", length(sites)))
```

Let  $G = (V, E)$  be a graph, where  $V$  is the set of vertices and  $E$  be the set of edges. The energy model for the hard-core lattice gas problem is

$$E(\mathbf{n}) = - \sum_{i \in V} w_i n_i + \infty \sum_{(i,j) \in E} n_i n_j$$

where  $n_i \in \{0, 1\}$  is the number of particles at site  $i$ , and  $w_i$  is the weight associated with it. For unweighted graphs, the weights are uniform.

## Solution Space Properties

- **Q1:** How many atoms we can fit into this lattice?
- **Q2:** Let us denote the maximum number of particles be  $G$ , how many such configurations can we find?
- **Q3:** How many configurations have size  $\alpha(G) - 1$  (with **1** defect)? Can they be related by "swap" operations?
- **Q4:** How does the energy landscape look like?

The solution space hard-core lattice gas is equivalent to that of an independent set problem. The independent set problem involves finding a set of vertices in a graph such that no two vertices in the set are adjacent (i.e., there is no edge connecting them). One can create a tensor network based modeling of an independent set problem with package `GenericTensorNetworks.jl`.

The `IndependentSet` constructor not only create a tensor network representation, but also optimizes the contraction order of this tensor network. Here, we choose `GreedyMethod` optimizer. A list of available optimizers are listed in package `OMEinsumContractionOrders.jl`

```
problem =
```

```
IndependentSet(62◦51◦85◦95◦73◦74◦63◦53◦52◦75, 53◦63◦51◦52◦62◦95◦75◦74◦85◦73 ->  
└─ 53◦62◦52◦51◦44◦65◦85◦95◦75◦46◦56◦45. 56◦52◦53◦44◦45◦46◦75◦73◦74◦63◦65 -
```

```
1 problem = IndependentSet(graph; optimizer=GreedyMethod())
```

### Note

Explain the tensor network structure for hard-core lattice gases on the blackboard. If we have time, I will explain the tensor contraction order finding algorithms.

One can check the time, space and read-write complexity. The return values are log2 values of the number of operations.

```
(17.7181, 12.0, 15.7956)
```

```
1 timespacereadwrite_complexity(problem)
```

## The best solution

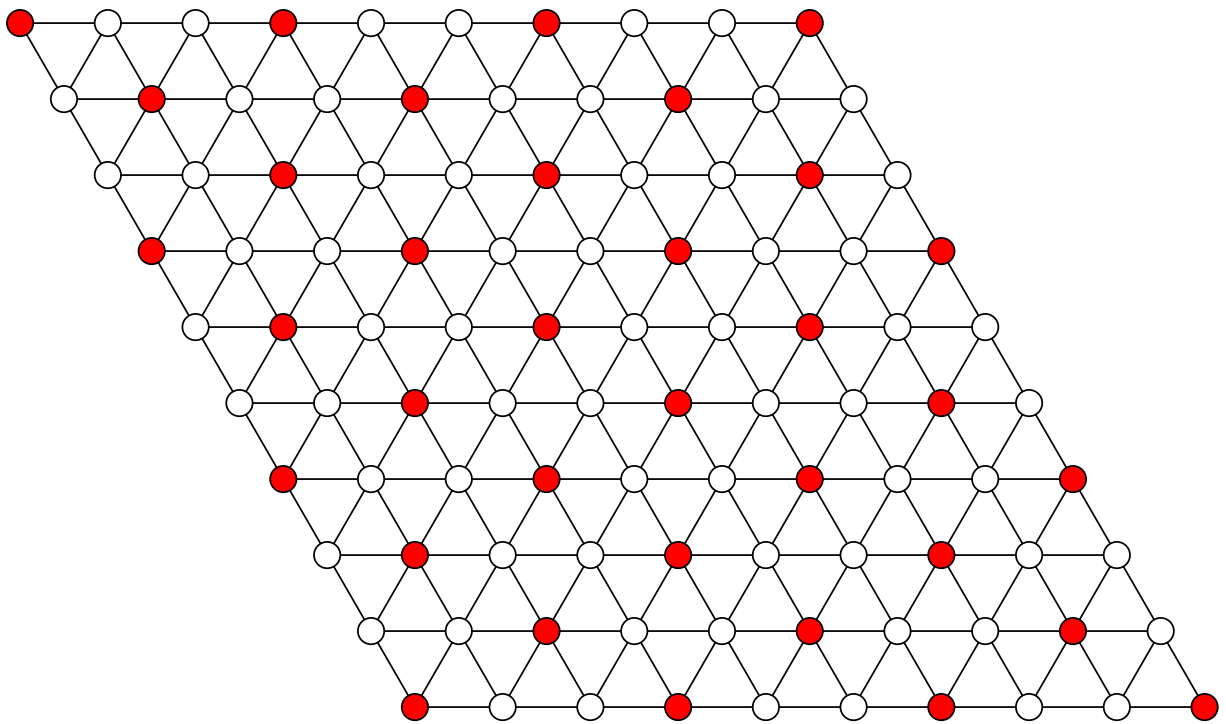
A standard task of a combinatorial optimization problem is to get one of the best solutions.

`GenericTensorNetworks` provides a unified interface for solution space property solving `solve(problem, property)`. The resulting value is always a tensor, its rank is equal to the number of open indices. Since we do not have open vertices in our demo problem, the return value is a rank-0 tensor.

```
bestconfig =
```

```
(34.0, GenericTensorNetworks.ConfigSampler{100, 1, 2}(100100100101001001000010010010100100:
```

```
1 bestconfig = solve(problem, SingleConfigMax())[]
```



```
1 show_config(sites, graph, bestconfig.c.data)
```

If one is only interested in finding the maximum solution size, the `SizeMax` property can be much faster to evaluate.

```
maximum_size = 34.0_t
```

```
1 maximum_size = solve(problem, SizeMax())[0]
```

If one is also interested in knowing how many best solutions are there, they should use `CountingMax` property.

```
(34.0, 1.0)_t
```

```
1 solve(problem, CountingMax())[0]
```

There is no degeneracy.

## Solution space visualization

---

```
122.0*x^33 + 1.0*x^34
```

```
1 solve(problem, CountingMax(2))[0]
```

```

max2configs = + (count = 122.0)
               + (count = 121.0)
                 + (count = 119.0)
                   + (count = 117.0)
                     + (count = 38.0)
                       + (count = 37.0)
                         :
                       * (count = 1.0)
                         :
                     + (count = 79.0)
                       * (count = 69.0)
                         :
                       * (count = 10.0)
                         :
                   * (count = 2.0)
                     * (count = 1.0)
                       * (count = 1.0)
                         :
                       * (count = 1.0)
                         :
                     * (count = 2.0)
                       + (count = 2.0)
                         :
                       * (count = 1.0)
                         :

```

```
1 max2configs = solve(problem, ConfigsMax(2; tree_storage=true))[]
```

**suboptimal =**

```
[GenericTensorNetworks.StaticBitVector{100, 2}: [0x000000000000000001, 0x0000000000000000,
```

```
1 suboptimal = collect(max2configs[1].coeffs[1])
```

**optimal =**

```
[GenericTensorNetworks.StaticBitVector{100, 2}: [0x000000000000000001, 0x0000000000000000,
```

```
1 optimal = collect(max2configs[1].coeffs[2])
```

In the following, we visualize the solution space spanned by states with optimal and suboptimal energies.

config\_graph = {123, 204} undirected simple Int64 graph

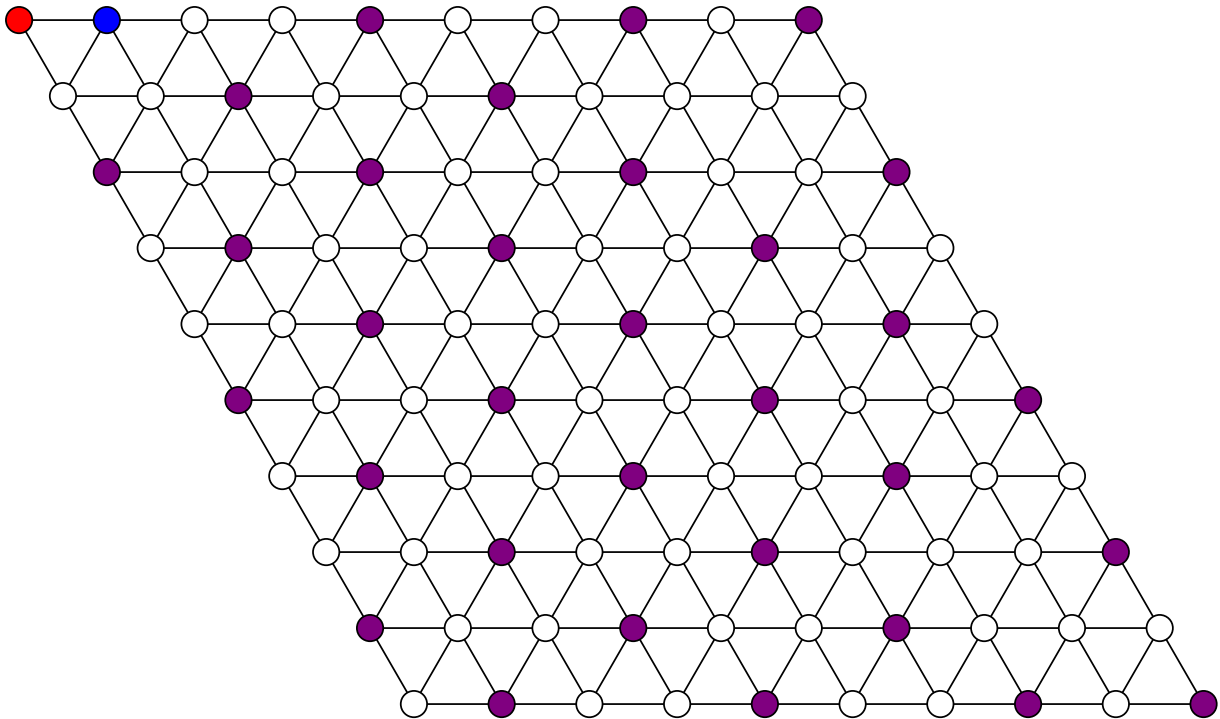
```
1 config_graph = let
2   vertices = [suboptimal..., optimal...]
3   config_graph = Graphs.SimpleGraph(length(vertices))
4   for (i, v) in enumerate(vertices), (j, w) in enumerate(vertices)
5     if connected_by_swap_on_edge(v, w, graph) || connected_by_flip(v, w, graph)
6       Graphs.add_edge!(config_graph, i, j)
7     end
8   end
9   config_graph
10 end
```

connected\_by\_swap\_on\_edge (generic function with 1 method)

```
1 function connected_by_swap_on_edge(v, w, graph)
2   diff = v ⊖ w
3   count_ones(diff) != 2 && return false
4   i, j = findall(==(1), diff)
5   return Graphs.has_edge(graph, i, j)
6 end
```

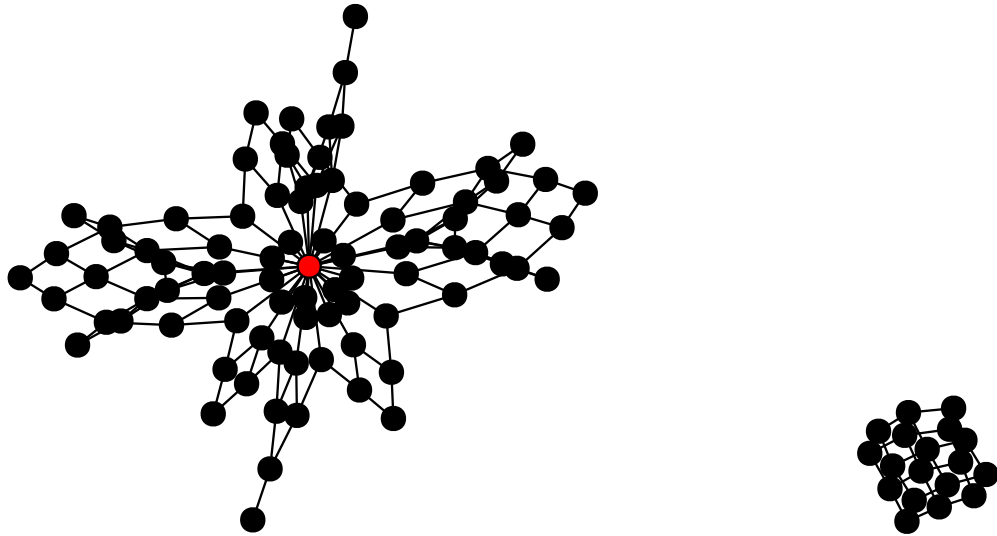
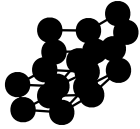
connected\_by\_flip (generic function with 1 method)

```
1 function connected_by_flip(v, w, graph)
2   diff = v ⊖ w
3   return count_ones(diff) == 1
4 end
```



```
1 compare_configs(sites, graph, suboptimal[2], suboptimal[3])
```





```
1 show_graph(config_graph, vertex_colors = [fill("black", length(suboptimal))...,  
fill("red", length(optimal))...], texts=fill("", Graphs.nv(config_graph)),  
vertex_size=0.1)
```

## Overlap gap property

---

- Gamarnik D. The overlap gap property: A topological barrier to optimizing over random structures[J]. Proceedings of the National Academy of Sciences, 2021, 118(41): e2108492118.

```

1 let
2   configs = max2configs.coeffs[1] + max2configs.coeffs[2]
3   distri = hamming_distribution(generate_samples(configs, 1000),
4   generate_samples(configs, 1000))
5   barplot(0:length(distri)-1, distri)
6 end

```

Observable 5147176927807856510 not found :(

## Independence polynomial

---

The math behind the library is related to the independence polynomial. The coefficients of an independence polynomial is the number of independent sets with different sizes.

$$I(G, x) = \sum_{k=0}^{\alpha(G)} a_k x^k,$$

where  $\alpha(G)$  is the maximum independent set size,  $a_k$  is the number of independent sets of size  $k$ . The total number of independent sets is thus equal to  $I(G, 1)$ .

```

poly = 1 + 100·x + 4689·x2 + 137120·x3 + 2805825·x4 + 42731866·x5 + 503008566·x6 + 4691761040·x7 +
35273226469·x8 + 216308210306·x9 + 1090944080545·x10 + 4550157262320·x11 + 15746922343282·x12 +
45287451876730·x13 + 108232333753792·x14 + 214631162623338·x15 + 352167834261209·x16 +
476104968091884·x17 + 527379514805796·x18 + 475279052672370·x19 + 345492781985186·x20 +
200505939920852·x21 + 91786224030339·x22 + 32687347631096·x23 + 8917493332547·x24 +
1833609141990·x25 + 279823949501·x26 + 31341820714·x27 + 2572045219·x28 + 156603436·x29 +
7244610·x30 + 258526·x31 + 6911·x32 + 122·x33 + x34

```

```

1 poly = solve(problem, GraphPolynomial())[]

```

## Statistical properties

---

The package `TensorInference.jl` is mainly designed for Bayesian inference. It can also be used to analyse physical systems.

- Murphy K P. Machine learning: a probabilistic perspective[M]. MIT press, 2012.


The standard inference tasks include

1. calculate the partition function (also known as the probability of evidence).
2. compute the marginal probability distribution over each variable given evidence.
3. find the most likely assignment to all variables given evidence.
4. find the most likely assignment to a set of query variables after marginalizing out the remaining variables.
5. draw samples from the posterior distribution given evidence.

```
1 using TensorInference
```

```
`TensorInference` loaded `GenericTensorNetworks` extension successfully,  
`TensorNetworkModel` and `MMAPModel` can be used for converting a `GraphProblem`  
to a probabilistic model now.
```

We need to convert a combinatorial optimization problem instance in `GenericTensorNetworks` to a probabilistic model in `TensorInference`. The conversion is a bit hacky, which involves adding some unity tensors to the network. For simplicity, this conversion recomputes the contraction order. The reference of `TensorInference` will come out soon, if I forgot to update, please write an email to [jinguoliu@hkust-gz.edu.cn](mailto:jinguoliu@hkust-gz.edu.cn).



```
1 @bind β Slider(0:0.1:10; default=3.0)
```

```
pmodel =  
TensorInference.TensorNetworkModel{Int64, OMEinsum.DynamicNestedEinsum{Int64}, Array{Float64,  
variables: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, :  
contraction time = 2^17.943, space = 2^12.0, read-write = 2^15.814
```

```
1 pmodel = TensorNetworkModel(problem, β)
```

## The partition function

$$Z = \sum_{\mathbf{n}} e^{-\beta E(\mathbf{n})},$$

where the summation runs over all  $2^{|V|}$  configurations of  $\mathbf{n}$ .

```
Z = 22021.38467088478
```

```
1 Z = solve(problem, PartitionFunction(-2.0))[]
```

## Marginal probabilities

---

The marginal probability tells how likely a site is occupied by a particle.

```
mars =
```

```
[[0.306254, 0.693746], [0.864143, 0.135857], [0.594706, 0.405294], [0.640934, 0.359066], [
```

```
1 mars = marginals(pmodel)
```

## Sampling from thermal equilibrium

---

```
TensorInference.Samples{Int64}: [view(::Matrix{Int64}, :, 1): [0, 0, 1, 0, 1, 0, 0, 1, 0,
```

```
1 sample(pmodel, 10)
```

```
sample_sets (generic function with 1 method)
```

```
1 function sample_sets(problem, βs, nsample)
2     n = solve(problem, SizeMax())[] .n
3     pss = Vector{Int}[]
4     for β in βs
5         pmodel = TensorNetworkModel(problem, β)
6         ns = dropdims(sum(sample(pmodel, nsample); dims=1); dims=1)
7         push!(pss, [count(==(i), ns) for i=0:n])
8     end
9     return pss
10 end
```

```
samples =
```

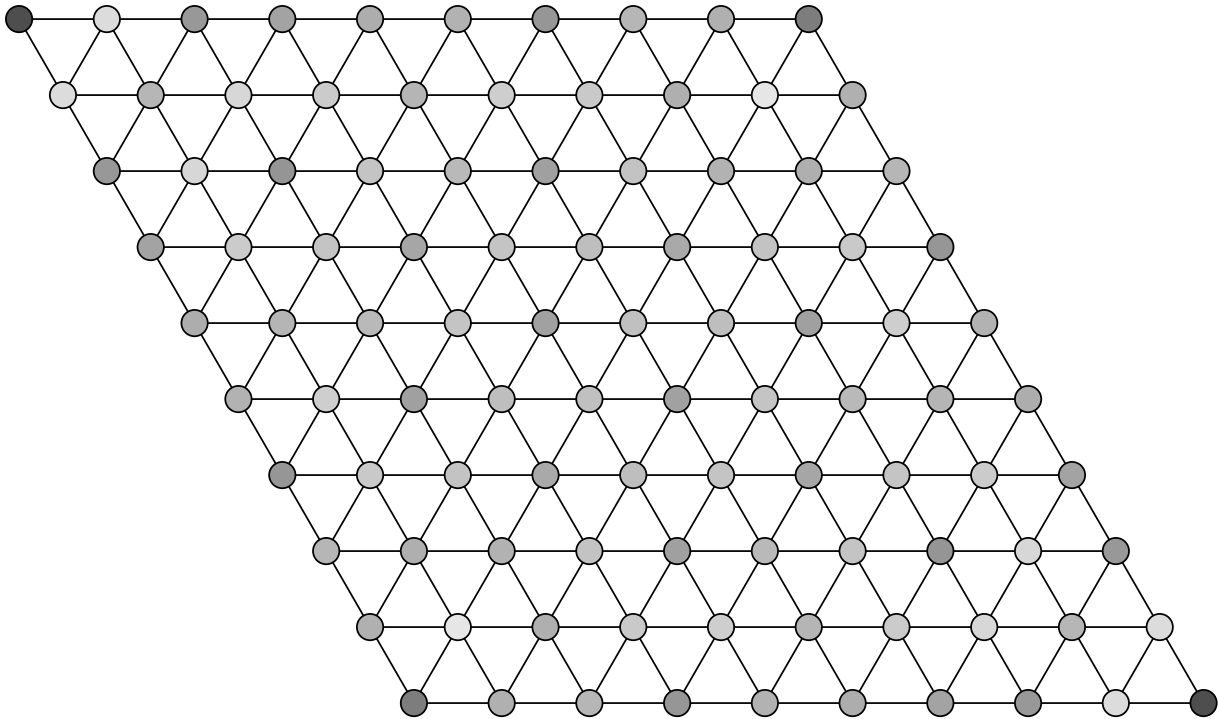
```
[[0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  more ,0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  more ,0], [0, 0, 0,
```

```
1 samples = sample_sets(problem, 0:2:6, 1000)
```

```

1 with_theme(plot_theme) do
2   ns = 0:length(poly.coeffs)-1
3   fig, ax, plt = plot(ns, log.(poly.coeffs))
4   ax.xlabel="number of atoms"
5   ax.ylabel="log(number of configurations)"
6   for (color,  $\beta$ , sample) in zip([RGBAf(0.8, 0.6, 0.1, 0.5), RGBAf(0.5, 0.7, 0.3,
7     0.5), RGBAf(0.6, 0.3, 0.3, 0.5), RGBAf(0.3, 0.7, 0.7, 0.5)], 0:2:6, samples)
8     log_sample = log.(max.(1, sample))
9     barplot!(ns, log_sample, color=color, opacity=0.4, label=" $\beta = \$\beta$ ")
10  end
11  axislegend("Samples", position = :lt)
12  fig
13 end

```



```

1 show_graph(graph; locs=sites, vertex_colors=[(1-b, 1-b, 1-b) for b in getindex.
(mars, 2)], texts=fill("", Graphs.nv(graph)))

```

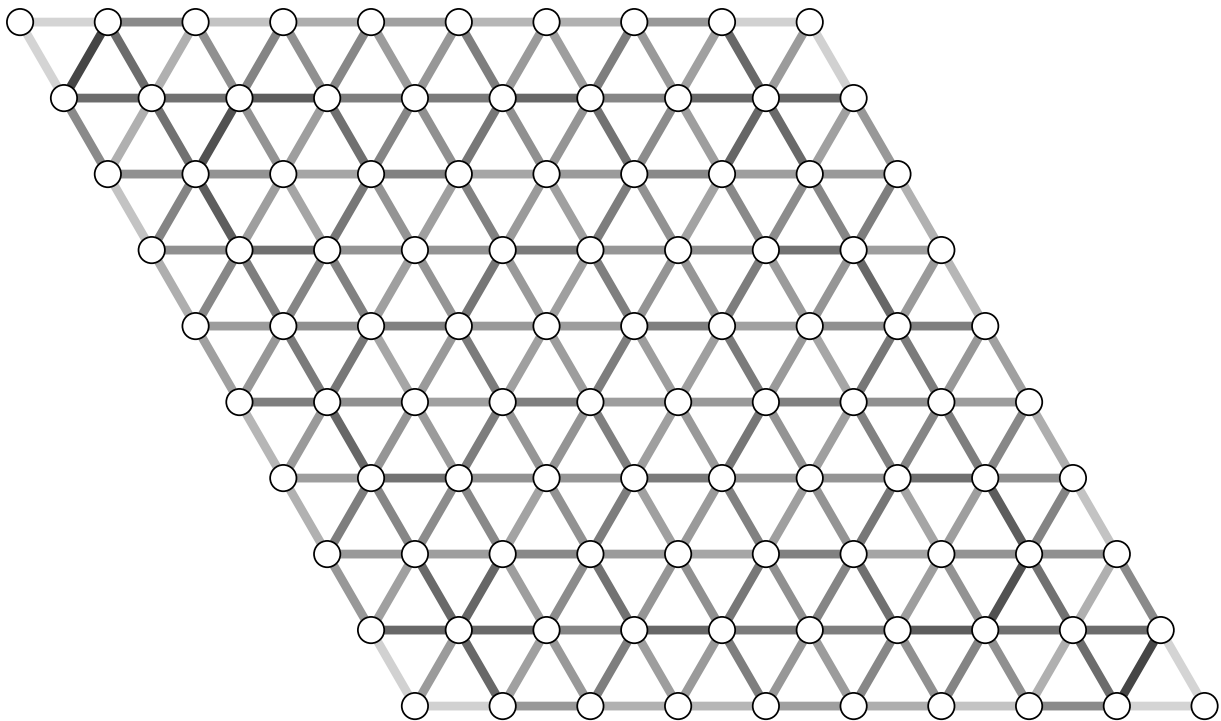
## Joint marginal probabilities

```
pmodel2 =  
TensorInference.TensorNetworkModel{Int64, OMEinsum.DynamicNestedEinsum{Int64}, Array{Float64,  
variables: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, :  
contraction time = 2^33.392, space = 2^24.0, read-write = 2^26.885
```

```
1 pmodel2 = TensorNetworkModel(problem,  $\beta$ ; mars=[[e.src, e.dst] for e in  
Graphs.edges(graph)])
```

```
1 mars2 = marginals(pmodel2);
```

Here, we plot the probability that two vertices have the same configuration.



```
1 show_graph(graph; locs=sites, edge_colors=[(b=mar[1, 1]; (1-b, 1-b, 1-b)) for mar in  
mars2], texts=fill("", Graphs.nv(graph)), edge_line_width=5)
```

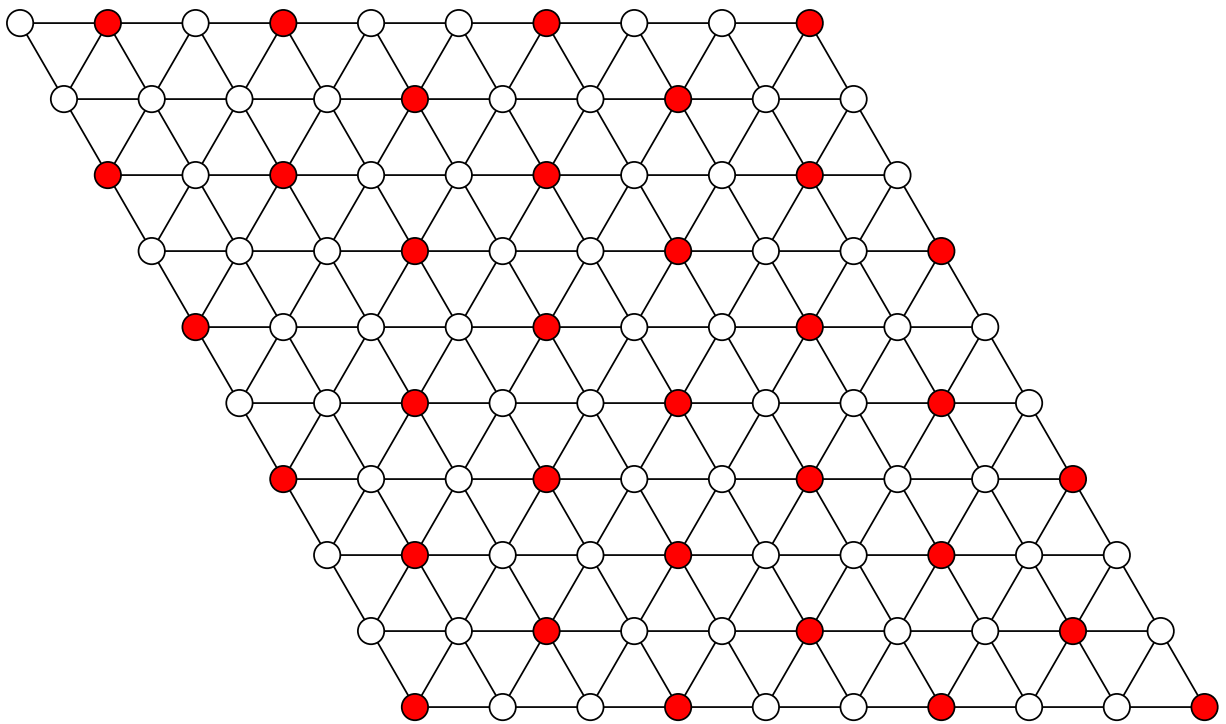
## Maximum a posterior estimation

The most probable configuration has  $\log(p) = 102$ , where  $p$  is the unnormalized partition function.

```
(102.0, [1, 0, 0, 1, 0, 0, 1, 0, 0, more ,1])
```

```
1 most_probable_config(pmodel)
```

Let us fix the first spin to state 0, and check the most probable configuration.



```

1 let
2   pmodel = TensorNetworkModel(problem, β; evidence=Dict(1=>0))
3   lop, cfg = most_probable_config(pmodel)
4   show_config(sites, graph, cfg)
5 end

```

## Logic Gadgets

- Nguyen M T, Liu J G, Wurtz J, et al. Quantum optimization with arbitrary connectivity using rydberg atom arrays[J]. PRX Quantum, 2023, 4(1): 010316.

Let  $\mathcal{C}$  be a constraint, and  $G = (V, E, \delta)$  be a weighted graph, where  $V$  is the vertex set,  $E$  is the edge set and  $\delta = (\delta_1, \delta_2, \dots, \delta_{|V|})$  are the weights associated with vertices. If the MISs of  $G$  encodes this constraint, i.e. the assignment satisfies the  $\mathcal{C}$  if and only if the corresponding set is a MIS of  $G$ ,  $(\mathcal{C}, G)$  forms a weighted MIS gadget.

## NOT gate

```
locs_not = [(0.0, 0.0), (0.0, 1.0)]
```

```
1 locs_not = [(0.0, 0.0), (0.0, 1.0)]
```

`graph_not = {2, 1}` undirected simple Int64 graph

```
1 graph_not = unit_disk_graph(locs_not, 1.1)
```



```
1 show_graph(graph_not; locs=locs_not, texts=fill("", Graphs.nv(graph_not)))
```



```
1 let
2   configs = solve(IndependentSet(graph_not), ConfigsMax())[]
3   show_gallery(graph_not, (1, 2); locs=locs_not, vertex_configs=configs.c,
4   texts=fill("", Graphs.nv(graph_not)))
5 end
```

`show_all_ground_states` (generic function with 1 method)

```
1 function show_all_ground_states(locs, weights, grid)
2   graph = unit_disk_graph(locs, 1.1)
3   configs = solve(IndependentSet(graph; weights), ConfigsMax())[]
4   show_gallery(graph, grid; locs=locs, vertex_configs=configs.c)
5 end
```

## NOR gate

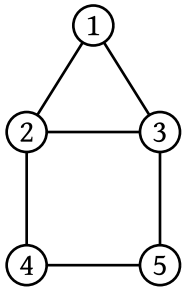
`locs_nor = [(0.0, -0.8), (-0.5, 0.0), (0.5, 0.0), (-0.5, 1.0), (0.5, 1.0)]`

```
1 locs_nor = [(0.0, -0.8), (-0.5, 0.0), (0.5, 0.0), (-0.5, 1.0), (0.5, 1.0)]
```

`graph_nor = {5, 6}` undirected simple Int64 graph

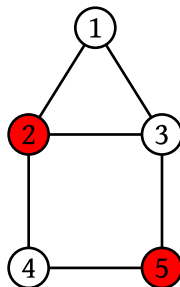
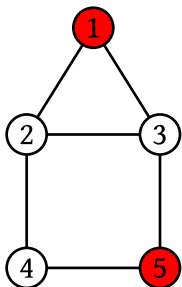
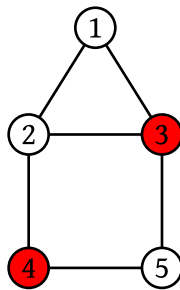
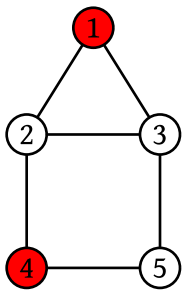
```
1 graph_nor = unit_disk_graph(locs_nor, 1.1)
```





```
1 show_graph(graph_nor; locs=locs_nor)
```

$$n_3 = \neg(n_1 \vee n_5)$$



```
1 show_all_ground_states(locs_nor, ones(Int, 5), (2, 2))
```

# NOT and NOR are universal

$$x \vee y = \neg \text{NOR}(x, y)$$
$$x \wedge y = \text{NOR}(\neg x, \neg y)$$

## Composibility of logic gates

---

The weighted MIS reduction scheme is rooted on the fact that weighted gadgets are composable. By adding up energy terms, the ground state of the composed model encodes the solution to the constraint satisfiability problem that all clauses are satisfied. For any two weighted MIS gadgets  $(C_1, G_1)$  and  $(C_2, G_2)$ , we have

$$\text{compose} : ((C_1, G_1), (C_2, G_2)) \mapsto (C_1 \wedge C_2, G_1 + G_2),$$
$$G_1 + G_2 = (V_1 \cup V_2, E_1 \cup E_2, \delta : v \mapsto (\delta_1)_v + (\delta_2)_v),$$

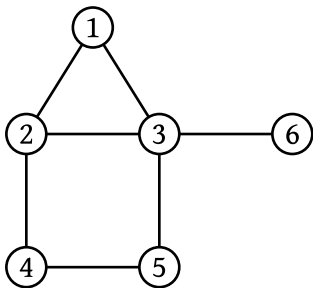
where the weight of a vertex in the new graph is a summation of the weights of this vertex in  $G_1$  and  $G_2$ .  $\wedge$  is the logical-and operator, which returns true if constraints on both sides are satisfied. If vertex  $v$  is absent in  $G_k$ ,  $(\delta_k)_v$  returns zero.

```
locs_or = [(0.0, -0.8), (-0.5, 0.0), (0.5, 0.0), (-0.5, 1.0), (0.5, 1.0), (1.5, 0.0)]
```

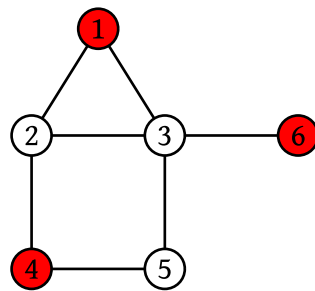
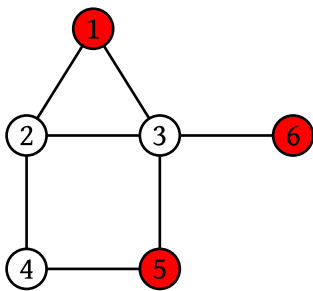
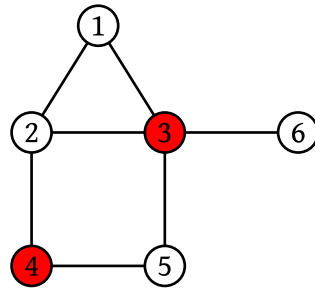
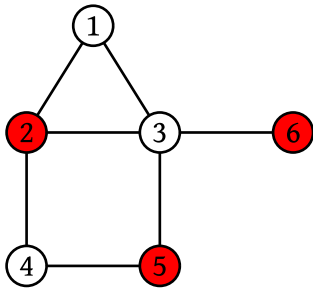
```
1 locs_or = [(0.0, -0.8), (-0.5, 0.0), (0.5, 0.0), (-0.5, 1.0), (0.5, 1.0), (1.5, 0.0)]
```

```
graph_or = {6, 7} undirected simple Int64 graph
```

```
1 graph_or = unit_disk_graph(locs_or, 1.1)
```



```
1 show_graph(graph_or; locs=locs_or)
```



```
1 show_all_ground_states(locs_or, (x=ones(Int, 6); x[3]=2; x), (2, 2))
```

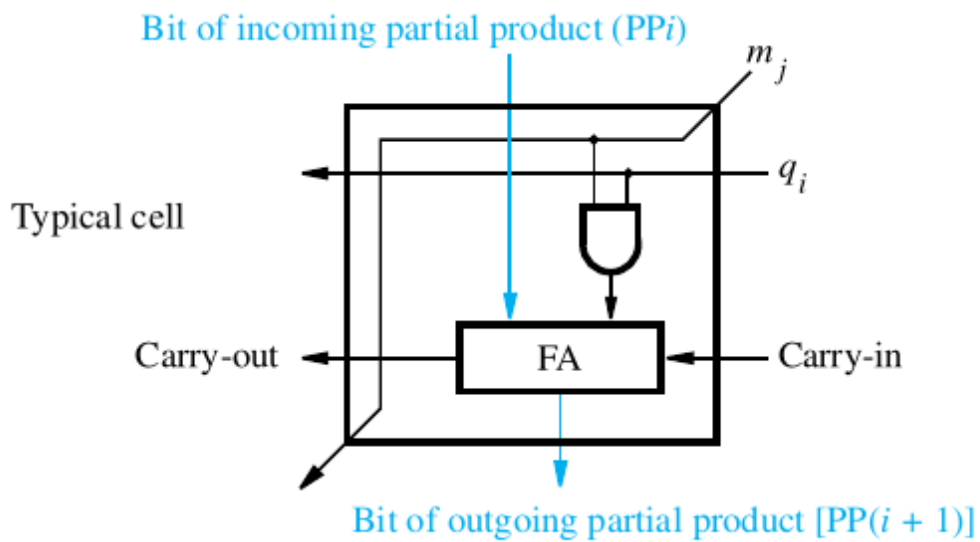
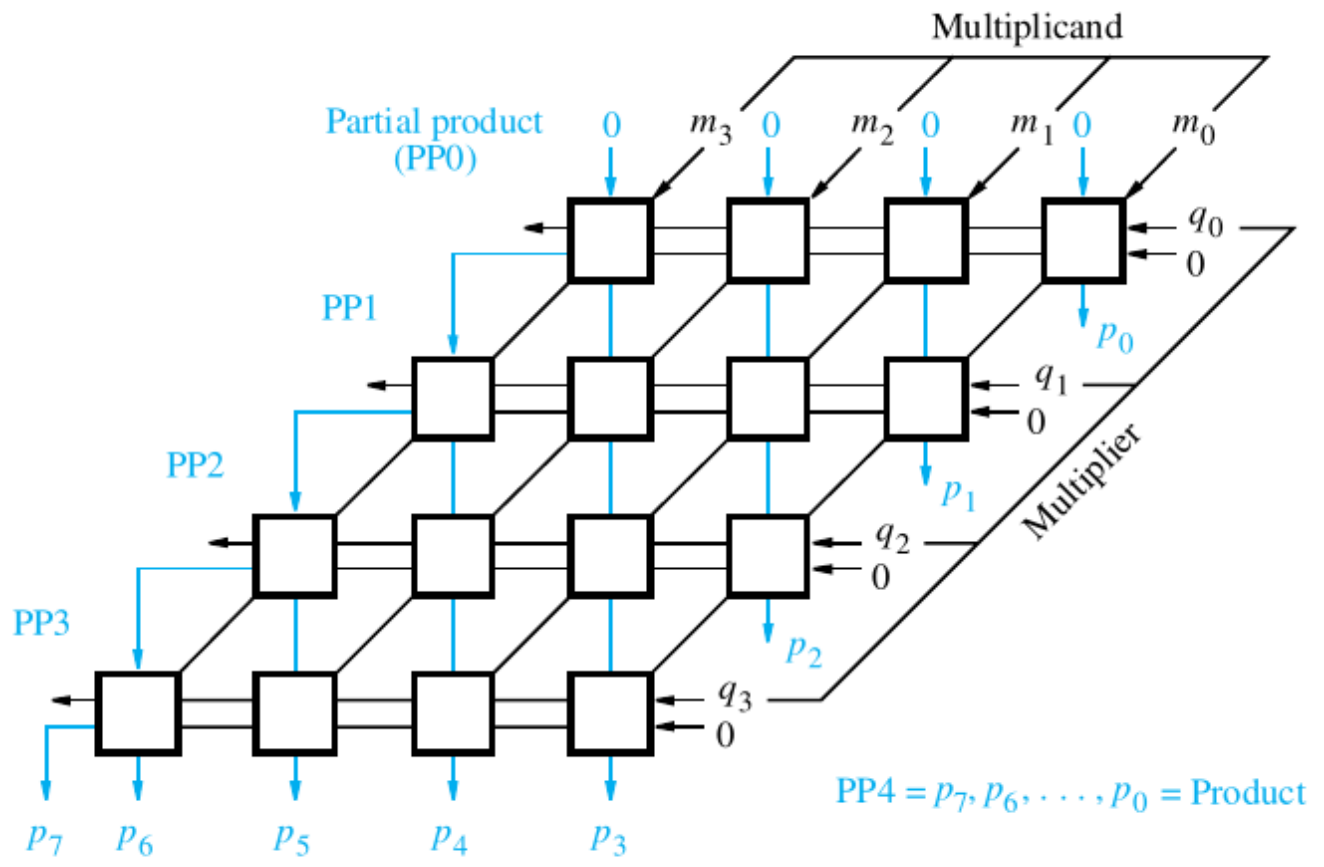
$$x_6 = x_1 \vee x_5$$

# Solving the integer factoring problem

The integer factoring problem is a mathematical problem that involves finding the prime factors of a given composite integer. Given a composite number  $N$ , the task is to determine its prime factors, which are the prime numbers that multiply together to give  $N$ .

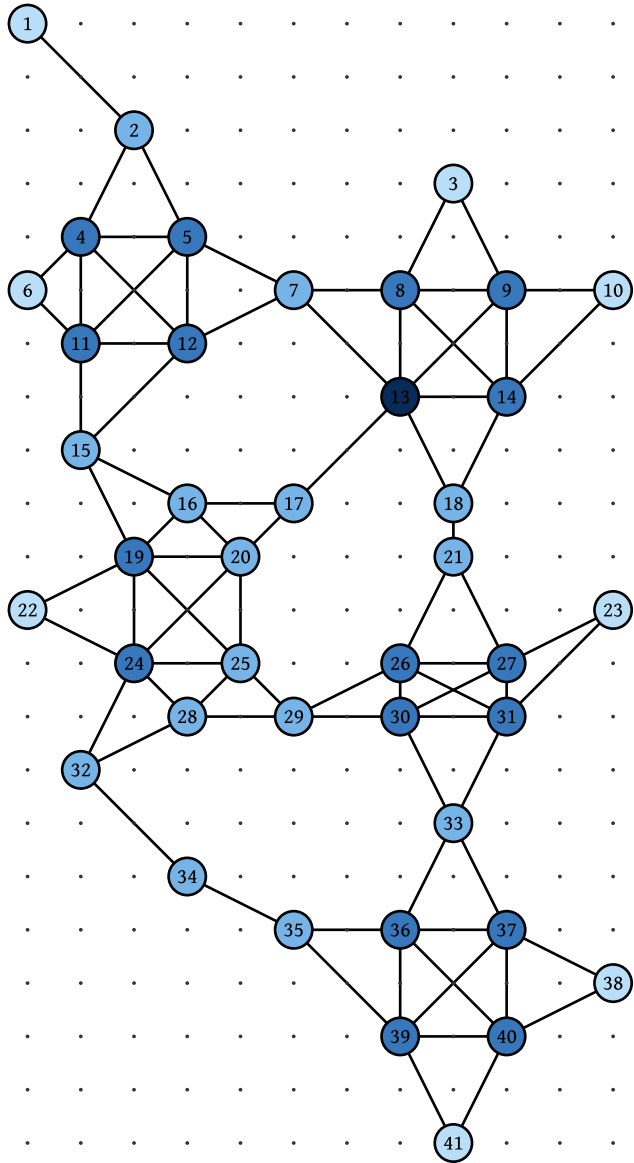
RSA encryption relies on the assumption that factoring large composite numbers is difficult. The RSA algorithm uses the product of two large prime numbers as the public key, while the private key involves the knowledge of the prime factors.

## Array multiplier



(b) Array implementation

**Figure 9.6** Array multiplication of unsigned binary operands.



```

1 let
2   graph, pins = UnitDiskMapping.multiplier()
3   texts = fill("", length(graph.nodes))
4   texts[pins] .= ["x$i" for i=1:length(pins)]
5   show_grayscale(graph; unit=20)
6 end

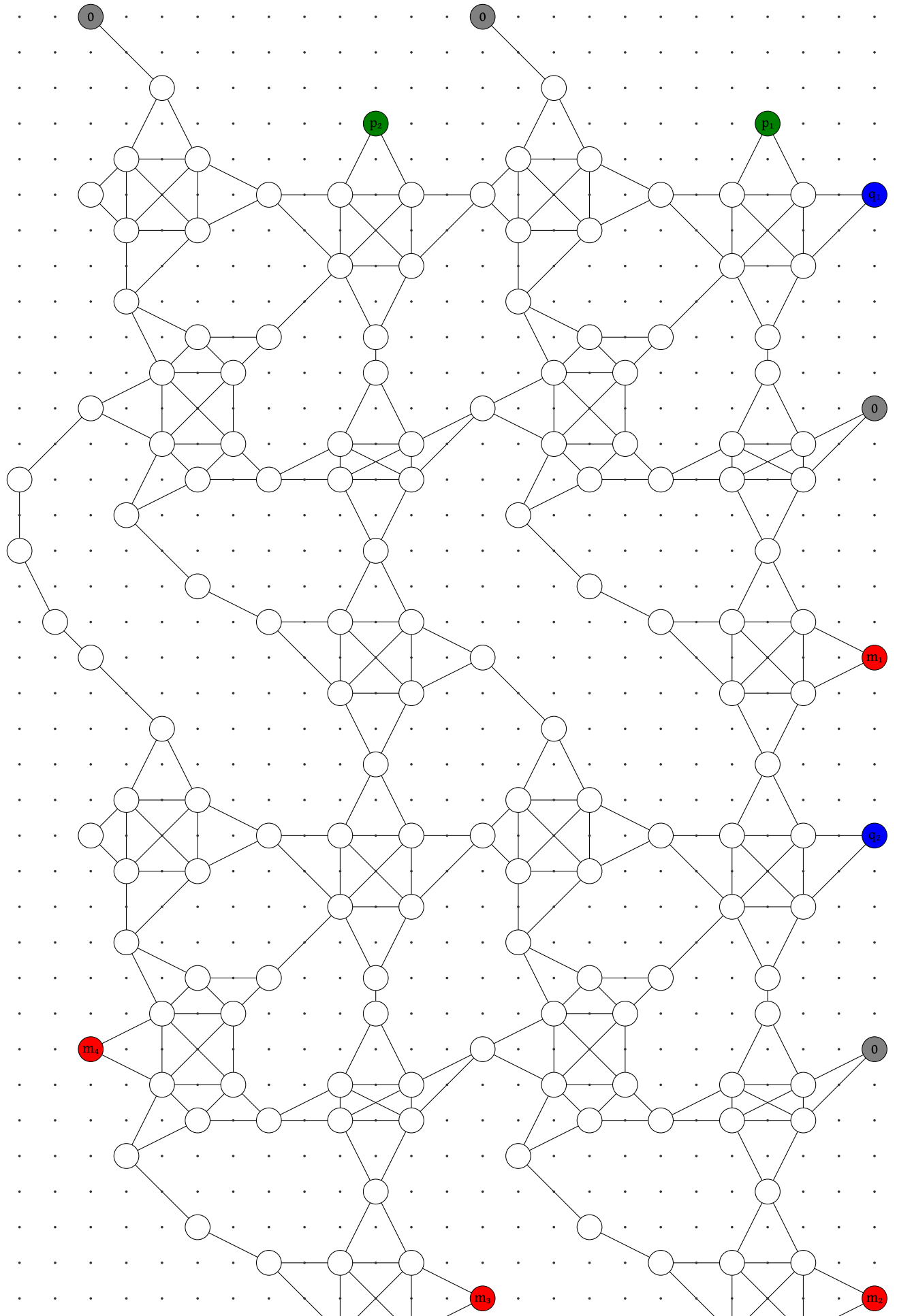
```

One can call `map_factoring(M, N)` to map a factoring problem to the array multiplier grid of size (M, N). In the following example of (2, 2) array multiplier, the input integers are  $p = 2p_2 + p_1$  and  $q = 2q_2 + q_1$ , and the output integer is  $m = 4m_3 + 2m_2 + m_1$ . The maximum independent set corresponds to the solution of  $pq = m$

```

1 mres = UnitDiskMapping.map_factoring(2, 2);

```



```
1 show_pins(mres)
```

To solve this factoring problem, one can use the following statement:

```
multiplier_output = (3, 2)
```

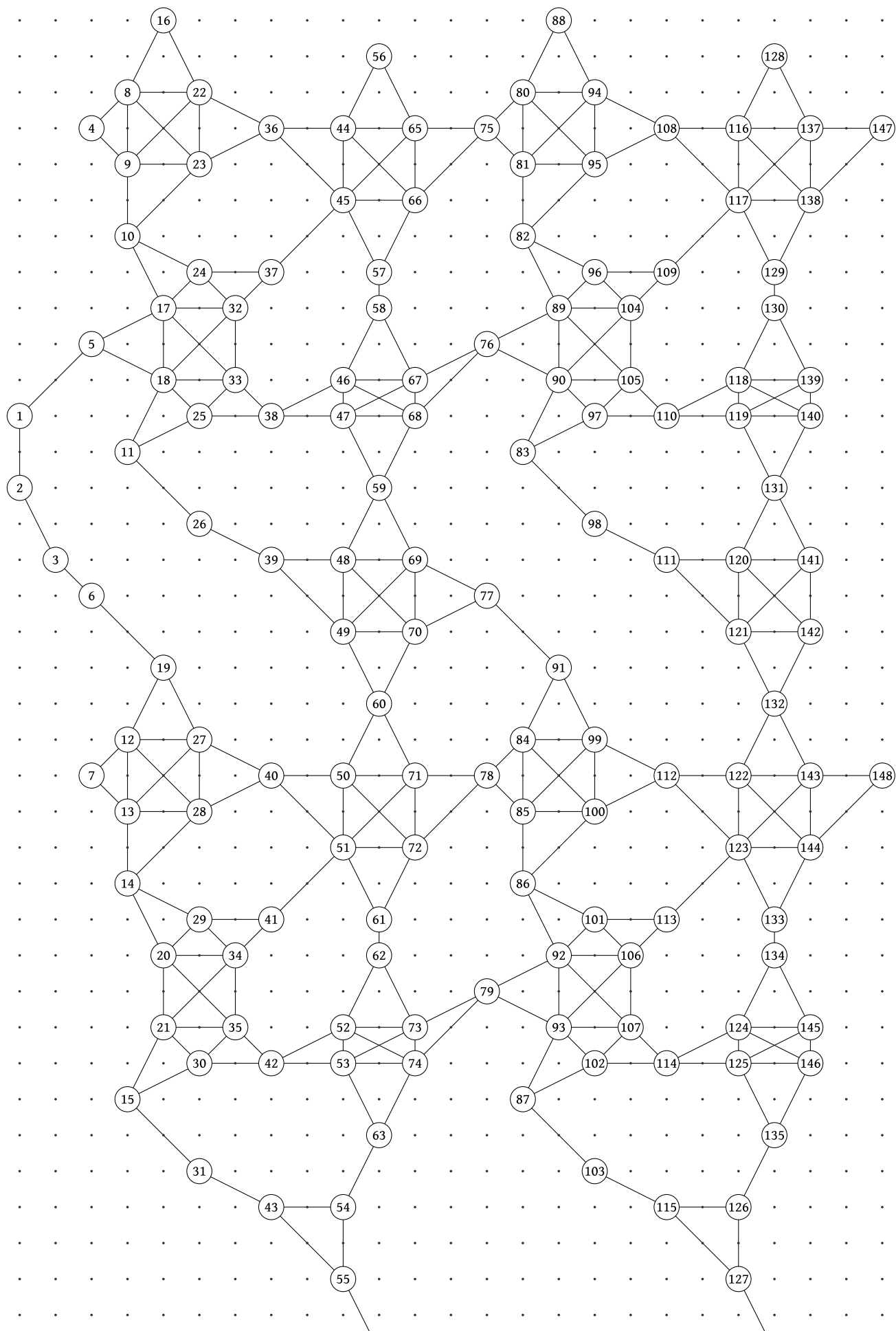
```
1 multiplier_output = UnitDiskMapping.solve_factoring(mres, 6) do g, ws
2   collect(Int, solve(IndependentSet(g; weights=ws), SingleConfigMax())[].c.data)
3 end
```

This function consists of the following steps:

1. We first modify the graph by inspecting the fixed values, i.e., the output  $m$  and  $\theta$ s:
  - o If a vertex is fixed to 1, remove it and its neighbors,
  - o If a vertex is fixed to 0, remove this vertex.

The resulting grid graph is

```
1 mapped_grid_graph, remaining_vertices = let
2   g, ws = graph_and_weights(mres.grid_graph)
3   mg, vmap = UnitDiskMapping.set_target(g, [mres.pins_zeros...,
4     mres.pins_output...], 6 << length(mres.pins_zeros))
5   GridGraph(mres.grid_graph.size, mres.grid_graph.nodes[vmap],
6     mres.grid_graph.radius), vmap
7 end;
```

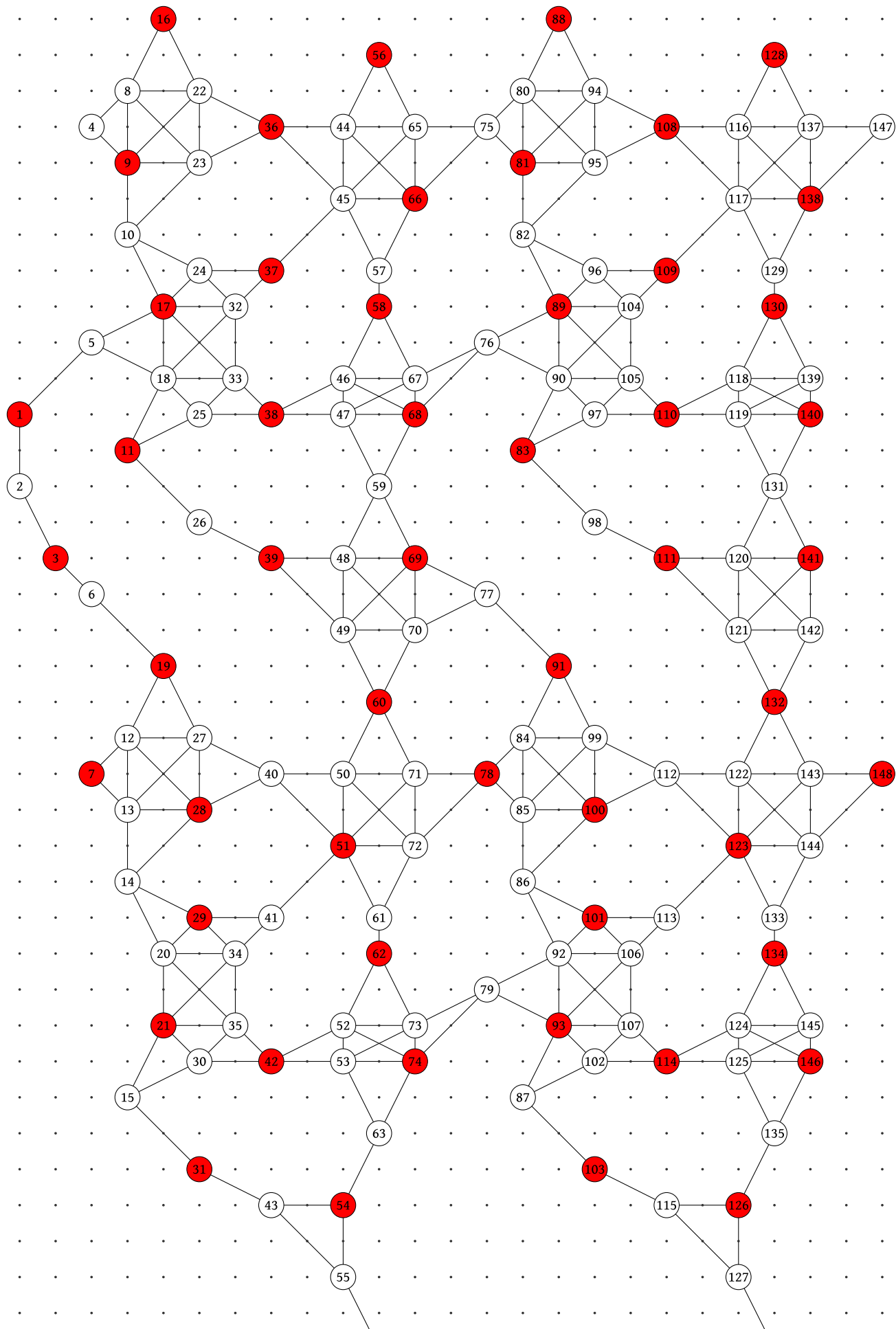




```
1 show_graph(mapped_grid_graph)
```

2. Then, we solve this new grid graph.

```
1 config_factoring6 = let
2   mg, mw = graph_and_weights(mapped_grid_graph)
3   solve(IndependentSet(mg; weights=mw), SingleConfigMax())[].c.data
4 end;
```



```
1 UnitDiskMapping.show_config(mapped_grid_graph, config_factoring6)
```

3. It is straightforward to read out the results from the above configuration. The solution should be either (2, 3) or (3, 2).

(3, 2)

```
1 let
2   cfg = zeros(Int, length(mres.grid_graph.nodes))
3   cfg[remaining_vertices] .= config_factoring6
4   bitvectors = cfg[mres.pins_input1], cfg[mres.pins_input2]
5   UnitDiskMapping.asint.(bitvectors)
6 end
```

## Conclusion

---

Solving the ground state of a many-body system must be difficult, otherwise someone cleverer than you could crack your password and steal your money.

## Solving the challenge problem

---

### Step 1: construct the target problem graph - the line graph of C60 structure

fullerene (generic function with 1 method)

```
1 # Returns the C60 structure 3D coordinates
2 function fullerene()
3   φ = (1+√5)/2
4   points = NTuple{3,Float64}[]
5   for (x, y, z) in ((0.0, 1.0, 3φ), (1.0, 2 + φ, 2φ), (φ, 2.0, 2φ + 1.0))
6     for (α, β, γ) in ((x,y,z), (y,z,x), (z,x,y))
7       for loc in ((α,β,γ), (α,β,-γ), (α,-β,γ), (α,-β,-γ), (-α,β,γ), (-α,β,-γ),
8                 (-α,-β,γ), (-α,-β,-γ))
9         if loc ∉ points
10          push!(points, loc)
11        end
12      end
13    end
14  return points
15 end
```

line\_graph (generic function with 1 method)

```
1 # get the line graph of target graph
2 function line_graph(g::Graphs.SimpleGraph)
3     edges = [(e.src, e.dst) for e in Graphs.edges(g)]
4     ne = Graphs.ne(g)
5     ledges = [(i, j) for i=1:ne, j=1:ne if j > i && !isempty(edges[i] ∩ edges[j])]
6     return Graphs.SimpleGraph([Graphs.Edge(i, j) for (i, j) in ledges])
7 end
```

nearest\_pairs (generic function with 1 method)

```
1 # get the distance-k vertex pairs
2 function nearest_pairs(g, k::Int)
3     res = Tuple{Int,Int}[]
4     for v in Graphs.vertices(g)
5         for (w, d) in Graphs.neighborhood_dists(g, v, k)
6             if d == k && v < w
7                 push!(res, (v, w))
8             end
9         end
10    end
11    return res
12 end
```

points\_c60 =

(0.0, 1.0, 4.8541), (0.0, 1.0, -4.8541), (0.0, -1.0, 4.8541), (0.0, -1.0, -4.8541), (1.0,

```
1 points_c60 = fullerene()
```

g\_c60 = {60, 90} undirected simple Int64 graph

```
1 g_c60 = unit_disk_graph(points_c60, sqrt(5))
```

g\_c60line = {90, 180} undirected simple Int64 graph

```
1 g_c60line = line_graph(g_c60)
```

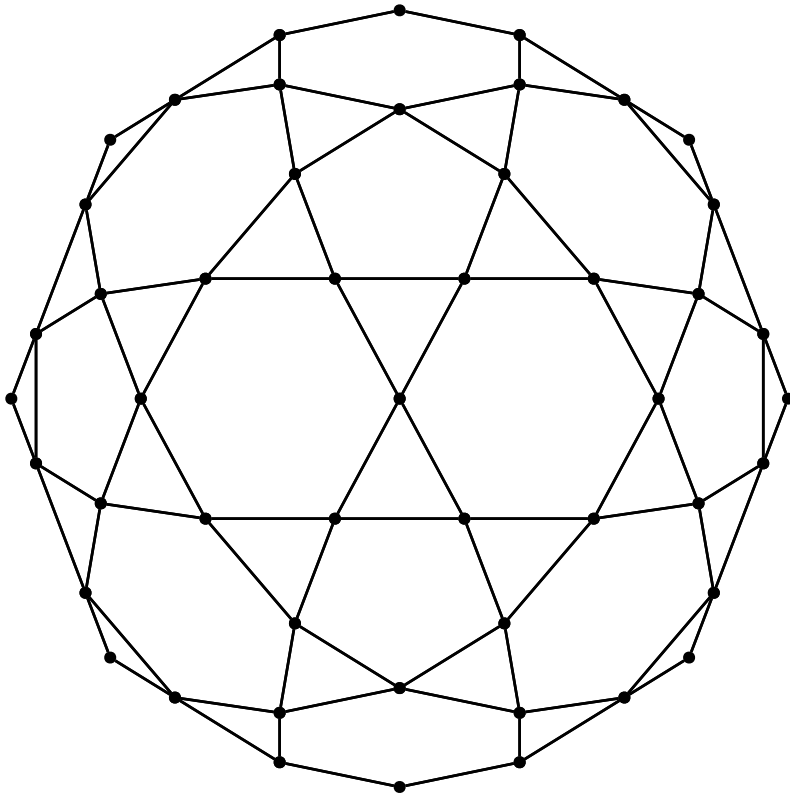
Let us visualize it.

```
1 using Rotations
```

project\_graph (generic function with 1 method)

```
1 function project_graph(locs, graph, Rx, Ry, Rz; colors)
2     vertices, edges = Int[], Tuple{Int,Int}[]
3     # rotate
4     rlocs = [RotXYZ(Rx, Ry, Rz) * [loc...] for loc in locs]
5     # filter
6     show_graph(graph; locs=rlocs, unit=15, texts=fill("", length(rlocs)),
7     vertex_colors=colors, vertex_line_width=0)
7 end
```

X:  , Y:  , Z:



```

1 let
2   dual_points = [points_c60[e.src] .+ points_c60[e.dst] for e in
   Graphs.edges(g_c60)]
3   #allpoints = [points..., dual_points...]
4   #project_graph(allpoints, join(g_c60, dual), Rx, Ry, Rz)
5   project_graph(dual_points, g_c60line, Rx, Ry, Rz; colors=fill("black",
   Graphs.nv(g_c60line)))
6 end

```

## Step 2: Use GenericTensorNetworks to solve this problem

Create a Spin-Glass problem instance.

sg =

```
SpinGlass(MaxCut(SlicedEinsum([], 90, 90 ->
  | 90
```

```

1 sg = SpinGlass(g_c60line; edge_weights=ones(Int, Graphs.ne(g_c60line)),
  optimizer=TreeSA())

```

```
(:target, :J, :h)
```

```

1 fieldnames(typeof(sg))

```

Time complexity (number of element-wise multiplications) =  $2^{15.272994203135088}$   
 Space complexity (number of elements in the largest intermediate tensor) =  $2^{10.0}$   
 Read-write complexity (number of element-wise read and write) =  $2^{14.105335637060879}$

```

1 contraction_complexity(GenericTensorNetworks.target_problem(sg))

```

```
1 subtypes(GenericTensorNetworks.AbstractProperty)
```

0-dimensional Array{TropicalF64, 0}:  
-60.0<sub>t</sub>

```
1 solve(sg, SizeMax())
```

(-60, 1628081942737728018)<sub>t</sub>

```
1 solve(sg, CountingMax(; T=Int64))[]
```

$$2 \cdot x^{-180} + 180 \cdot x^{-176} + 480 \cdot x^{-174} + 9270 \cdot x^{-172} + 46848 \cdot x^{-170} + 419560 \cdot x^{-168} + 2614080 \cdot x^{-166} + 18182010 \cdot x^{-164} + 115962880 \cdot x^{-162} + 745057596 \cdot x^{-160} + 4675291680 \cdot x^{-158} + 29106524430 \cdot x^{-156} + 178520901120 \cdot x^{-154} + 1077708658320 \cdot x^{-152} + 6370469242496 \cdot x^{-150} + 36731232620730 \cdot x^{-148} + 205743778022400 \cdot x^{-146} + 1115871054856020 \cdot x^{-144} + 5842401162962400 \cdot x^{-142} + 29450066371862910 \cdot x^{-140} + 142557191150567680 \cdot x^{-138} + 661059737578267800 \cdot x^{-136} + 2929699509020487360 \cdot x^{-134} + 12381157737462628850 \cdot x^{-132} + 49788089743444187136 \cdot x^{-130} + 190122177499298200380 \cdot x^{-128} + 688094606388563843360 \cdot x^{-126} + 2356051695249917895510 \cdot x^{-124} + 7619107171202333045760 \cdot x^{-122} + 23233475890018028884448 \cdot x^{-120} + 66706446493011743343360 \cdot x^{-118} + 180076900120108896531750 \cdot x^{-116} + 456470204537566643906560 \cdot x^{-114} + 1085139539491015595887740 \cdot x^{-112} + 2416290414384392738567328 \cdot x^{-110} + 5033618861513515307542850 \cdot x^{-108} + 9798255073434063426627840 \cdot x^{-106} + 17799236430998165430070680 \cdot x^{-104} + 30133280771255186651368640 \cdot x^{-102} + 47471799500305584979696974 \cdot x^{-100} + 69477733827114321906877440 \cdot x^{-98} + 94288054072995040829427860 \cdot x^{-96} + 118394170613682809213800800 \cdot x^{-94} + 137209291434341456482115850 \cdot x^{-92} + 146338594636220418087593472 \cdot x^{-90} + 143149352704622795682218640 \cdot x^{-88} + 127924999593442336301665920 \cdot x^{-86} + 103951509150741068893900190 \cdot x^{-84} + 76385883350557081101358080 \cdot x^{-82} + 50423275241157566093097852 \cdot x^{-80} + 29663146154802249892074400 \cdot x^{-78} + 15400230533800114631592330 \cdot x^{-76} + 6970645087708714249731840 \cdot x^{-74} + 2708390188149782846741480 \cdot x^{-72} + 885064458463260236818752 \cdot x^{-70} + 236542831948740086046150 \cdot x^{-68} + 49643092979071831552000 \cdot x^{-66} + 7671008351073013792500 \cdot x^{-64} + 775849436920187100000 \cdot x^{-62} + 38521570090156831250 \cdot x^{-60}$$

```
1 solve(sg, GraphPolynomial(; method=:finitefield))[]
```

3.8521570090156835e19

```
1 38521570090156831250.0
```

We emphasize that the spin-glass problem is reduced to the Max-Cut problem. Since we work on the target problem, the generated samples need to be extracted.

```

configs_max = + (count = 3.8521570090156835e19)
               + (count = 1.9260785045078417e19)
               + (count = 6.349759056726331e18)
               + (count = 4.3135214314492667e18)
               + (count = 2.211376739182998e18)
               + (count = 1.891540268216193e18)
               :
               * (count = 3.19836470966805e17)
               :
               + (count = 2.1021446922662687e18)
               + (count = 1.9419795235111644e18)
               :
               * (count = 1.601651687551044e17)
               :
               + (count = 2.0362376252770652e18)
               + (count = 1.8763193897936627e18)
               + (count = 1.570962997434282e18)
               :
               * (count = 3.053563923593808e17)
               :
               * (count = 1.599182354834025e17)
               + (count = 1.599182354834025e17)
               :
               * (count = 1.0)
               :

```

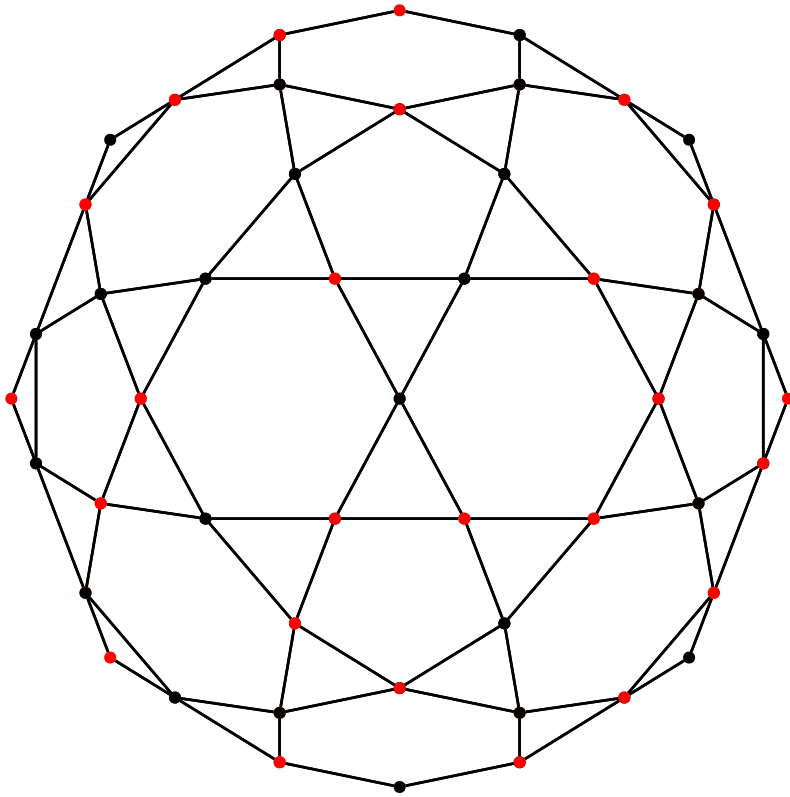
```
1 configs_max = solve(sg, ConfigsMax(; tree_storage=true))[].
```

best\_samples =

```
[GenericTensorNetworks.StaticBitVector{90, 2}: [0x0000000000000001, 0x0000000000000001, 0
```

```
1 best_samples = generate_samples(configs_max, 100)
```

```
1 @bind sample_index Slider(1:length(best_samples); show_value=true)
```



```

1 let
2   dual_points = [points_c60[e.src] .+ points_c60[e.dst] for e in
   Graphs.edges(g_c60)]
3   project_graph(dual_points, g_c60line, Rx, Ry, Rz; colors=[c == 1 ? "red" :
   "black" for c in best_samples[sample_index]])
4 end

```

`g_c60line2 = {90, 540} undirected simple Int64 graph`

```

1 g_c60line2 = let
2   g = deepcopy(g_c60line)
3   second_nearest = nearest_pairs(g, 2)
4   for (i, j) in second_nearest
5     Graphs.add_edge!(g, i, j)
6   end
7   g
8 end

```

`sg2 =`

```

SpinGlass(MaxCut(SlicedEinsum([], 53, 53 ->
  └─ 33◦53. 53◦33 -> 53

```

```

1 sg2 = SpinGlass(g_c60line2; edge_weights=ones(Int, Graphs.ne(g_c60line2)),
  optimizer=TreeSA())

```

target space complexity not found, got: 26.0, with time complexity 35.16506687852741, read-write complexity 29.568223538704967.



Time complexity (number of element-wise multiplications) =  $2^{35.16506687852741}$   
Space complexity (number of elements in the largest intermediate tensor) =  $2^{26.0}$   
Read-write complexity (number of element-wise read and write) =  $2^{29.56822353870497}$

```
1 contraction_complexity(sg2.target)
```

```
julia> @time solve(sg2, CountingMax(); usecuda=true, T=Int)
0.469537 seconds (224.23 k allocations: 10.290 MiB)
0-dimensional CuArray{CountingTropical{Int64, Int64}, 0, CUDA.Mem.DeviceBuffer}:
(-200, 67441356)†
```

```
julia> @time solve(sg2, GraphPolynomial(); usecuda=true)
1529.071057 seconds (322.69 M allocations: 14.157 GiB, 0.72% gc time)
0-dimensional Array{LaurentPolynomial{BigInt, :x}, 0}:
LaurentPolynomial(2*x-540 + 180*x-528 + 1080*x-518 + 6930*x-516 + 1880*x-510 + 6240*x-508 + 76920*x-506 + 151180*x-504 + 5880*x-502 + 19284*x-500 + 170760*x-498 + 625080*x-496 + 2315640*x-494 + 2179180*x-492 + 587808*x-490 + 2221020*x-488 + 8518000*x-486 + 23227680*x-484 + 40438920*x-482 + 29308568*x-480 + 34838400*x-478 + 105522840*x-476 + 260553920*x-474 + 471589140*x-472 + 554607600*x-470 + 627335990*x-468 + 1306930320*x-466 + 2913882240*x-464 + 5152434920*x-462 + 7157976078*x-460 + 9372922200*x-458 + 16350102360*x-456 + 31875303240*x-454 + 54825135420*x-452 + 81005127272*x-450 + 117330574080*x-448 + 193485616320*x-446 + 342572352400*x-444 + 567140330160*x-442 + 865795535436*x-440 + 1316769126960*x-438 + 2137552097040*x-436 + 3565110703680*x-434 + 5738857808980*x-432 + 8891316451200*x-430 + 13851091018800*x-428 + 22226145464560*x-426 + 35878183618380*x-424 + 56785288064160*x-422 + 88622863747140*x-420 + 139237645261440*x-418 + 221172964613160*x-416 + 350664150439680*x-414 + 550660027478640*x-412 + 861979477645584*x-410 + 1354780284595860*x-408 + 2134576988391600*x-406 + 3352933663254750*x-404 + 5248349553226040*x-402 + 8214661992245676*x-400 + 12874470821728560*x-398 + 20165800781591550*x-396 + 31527084026597160*x-394 + 49237001050232280*x-392 + 76893772319856960*x-390 + 120042828446598990*x-388 + 187201084459427640*x-386 + 291613346853442760*x-384 + 453944143873789680*x-382 + 706137478557067104*x-380 + 1097313297682723920*x-378 + 1703210349485918280*x-376 + 2640766088689698720*x-374 + 4089820414600375610*x-372 + 6325889278904451168*x-370 + 9770746369063749360*x-368 + 15069530152127684360*x-366 + 23206333797337734540*x-364 + 35677673400106584840*x-362 + 54754644093725588248*x-360 + 83876523376899240120*x-358 + 128236886020418483940*x-356 + 195654466380310314200*x-354 + 297867038794331094780*x-352 + 452443499784446370048*x-350 + 685593598818321383340*x-348 + 1036281882071507764200*x-346 + 1562229967228396885680*x-344 + 2348625051779594468120*x-342 + 3520684203132709833858*x-340 + 5261688443639047343280*x-338 + 783871701483352206560*x-336 + 11639125059348899930880*x-334 + 17221887397762277382750*x-332 + 25389337409722217981648*x-330 + 37286572724309226471720*x-328 + 54537957099711684067320*x-326 + 79432707533590305548980*x-324 + 115174466125246436331840*x-322 + 166212785050149836443596*x-320 + 238677323813989905290880*x-318 + 340937599075454678880180*x-316 + 484311356421117396770520*x-314 + 683941594048797557364640*x-312 + 959853841334229687536352*x-310 + 1338188760685311456441240*x-308 + 1852576919114243135670280*x-306 + 2545578882269320541059440*x-304 + 3470050466494856118704880*x-302 + 4690207331499817757241724*x-300 + 6282054602409528827405640*x-298 + 8332720901152424990062080*x-296 + 10938105382986333978071200*x-294 + 14198136948859624756530480*x-292 + 18208899621601546179632088*x-290 + 23050958509360339672133420*x-288 + 28773506964372365188576320*x-286 + 35374535646678751426233090*x-284 + 42778174376256449042653160*x-282 + 50811701829656863839124776*x-280 + 59186368843625045456450160*x-278 + 67487864538672704026527230*x-276 + 75183441723146639463736800*x-274 + 81652609749913059838959060*x-272 + 86245942787616501266164496*x-270 + 88371153333586497042646230*x-268 + 87597114912246056022024720*x-266 + 83756461718738418428184300*x-264 + 77019251615225670490794120*x-262 + 67909044567214572263807184*x-260 + 57243104864531722077311360*x-258 + 46000430966319315040322940*x-256 + 35147872674254551256288760*x-254 + 25473034862715530108084940*x-252 + 17471965728926713968235920*x-250 + 11318572516127946440464080*x-248 + 6911779787025839129563480*x-246 + 3971139180862867851342090*x-244 + 2142544458406893403045680*x-242 + 1083285538069762699249604*x-240 + 512118278588744100231120*x-238 + 225784139285006246065260*x-236 + 92558328317957865343680*x-234 + 35157049287919243784880*x-232 + 12321910103095777972056*x-230 + 3965164474956979380840*x-228 + 1164644101193222398440*x-226 + 310030953597431387400*x-224 + 74171258404206425280*x-222 + 15788368475984541048*x-220 + 2955232097232551040*x-218 + 479802610380448240*x-216 + 66529900824874200*x-214 + 7744714698343980*x-212 + 742676055934152*x-210 + 57357632728380*x-208 + 3454913154840*x-206 + 153604518320*x-204 + 4514839680*x-202 + 67441356*x-200)
```

# Helper functions

---

```
1 begin # helper functions
2     TableOfContents()
3     plot_theme = Theme(
4         Axis = (
5             xlabelsize=28,
6             ylabelsize=28,
7             xticklabelsize=20,
8             yticklabelsize=20,
9         )
10    );
11    # utility for showing a configuration
12    function show_config(locs, graph, config)
13        show_graph(graph; locs, texts=fill("", Graphs.nv(graph)), vertex_colors=[x
14            == 1 ? "red" : "white" for x in config])
15    end
16    # utility for comparing two configurations
17    function compare_configs(locs, graph, config1, config2)
18        colormap = Dict((0, 0)=>"white", (0, 1)=>"blue", (1, 0)=>"red", (1,
19            1)=>"purple")
20        show_graph(graph; locs, texts=fill("", Graphs.nv(graph)), vertex_colors=
21            [colormap[Int.((x, y))] for (x, y) in zip(config1, config2)])
22    end
23 end;
```