Spanning Trees on a Lattice

Report by
Adithya A Rao
National Institute of Technology, Surat

Supervisor
Prof. Sourendu Gupta
Tata Institute of Fundamental Research, Mumbai.

December 2023 - January 2024

Acknowledgement

I would like to express my sincere gratitude to Prof. Sourendu Gupta for agreeing to supervise this project and his invaluable guidance and supervision throughout this project. His expertise and insights have been instrumental in shaping this research work. His support has been crucial in making this project possible and contributing to my academic growth.

ADITHYA A RAO

Contents

1 Introduction

Lattice field theories are a powerful computational framework for studying quantum field theories in a non-perturbative regime. In this approach, continuous spacetime is discretized into a lattice of points, allowing for numerical simulations of complex quantum systems. This discretization provides a natural ultraviolet cutoff, regularizing the theory and making it amenable to computational methods. Lattice field theories have been particularly successful in the study of quantum chromodynamics (QCD), the theory of strong interactions, where they have provided insights into phenomena such as quark confinement and chiral symmetry breaking. The lattice formulation also serves as a bridge between the continuum theory and its discrete counterpart, offering a systematic way to approach the continuum limit and extract physical observables.

In this work, we consider the spanning tree gauge fixing on the lattice. Specifically, we show that fixing a spanning tree fixes the gauge, and then we proceed to provide algorithms to count and enumerate all the spanning trees on a hypercubic lattice with periodic boundary conditions, and also to find the local gauge transformation that transforms one spanning tree to another.

2 Gauge Fixing on the Lattice

On the lattice, the matter fields live on the lattice sites, and the gauge fields play the role of comparator owing to the local gauge symmetry and therefore live on the links connecting the lattice sites.

Local gauge invariance on the lattice means we have the freedom to assign any group element (that leads to the local gauge transformation) $G(n_i)$ to each lattice site. But such an assignment would also affect the link variables, transforming them according to eq (1).

$$U(n_i, n_j) \to G(n_i)U(n_i, n_j)G(n_j)^{-1} \tag{1}$$

Therefore in the path integral, one would be overcounting the different configurations by considering their gauge-transformed configurations too.

In the continuum case with continuous groups, this is a huge problem since the relevant fields are lie-algebra valued. The lie algebra is non-compact which implies that the path integral counts infinitely many copies. In the lattice version of the same, the link variables that are group-valued are relevant, and since the groups are compact, the volume is finite, and we are simply getting a normalization factor equal to the (finite)volume of the gauge group, $\mathcal{N} = V^N$ (where N is the total number of lattice sites).

The gauge fixing procedure is not a necessity for the observables which are gauge invariant, since the finite volume factor gets absorbed into the normalization. On the other hand for gauge variant observables, a class to which many important observables belong, to compare values in different configurations, it becomes necessary to implement some gauge fixing procedure that fixes the gauge and allows us to compare the different configurations even in the presence of local gauge transformations.

3 Spanning Trees

Let us start with a random configuration of U on the lattice. Initially, we assume all $G(n)$ to be set to $\mathbb{I}$. We start with a link $U_{\mu_0}(n_0)$ and set it to $\mathbb{I}$. This can be done by setting $G(n_0 + \mu_0) = U_{\mu_0}(n_0)$ keeping all other gauge elements at $\mathbb{I}$. This transforms the link $U_{\mu_0}(n_0)$ as $U_{\mu_0}(n_0) \to U_{\mu_0}(n_0)G(n_0 + \mu_0)^{-1} = \mathbb{I}$. Now we can fix another link starting from $n_0 + \mu_0$ to $\mathbb{I}$ by fixing the group element at the other end of the link and so on. The whole procedure can be repeated until we reach a link whose other end-point is already fixed and can't be modified.

This procedure generates a cluster of links that are set to $\mathbb{I}$ which does not contain closed loops. Such a cluster is called as as a spanning tree on the lattice. This fixes the values of group elements on all the sites, at the same time, since it does not contain closed loops, preserves the values of gauge invariant observables.

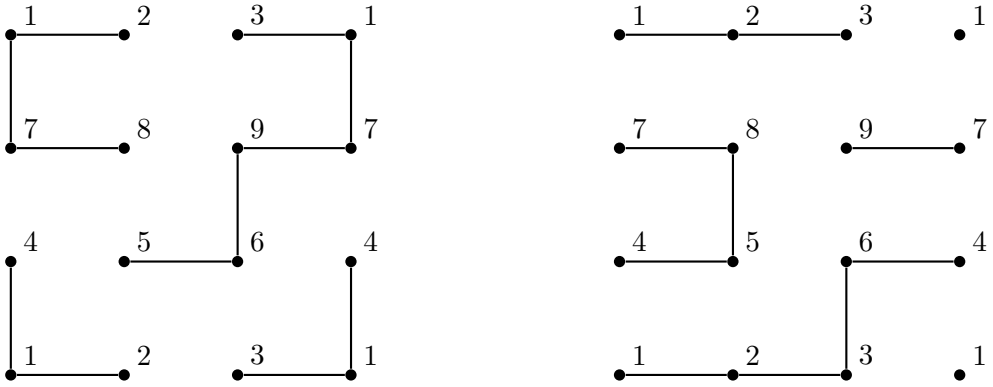


Figure 1: Examples of two spanning trees on a 3×3 lattice with periodic boundary conditions

Deciding a spanning tree for a lattice is equivalent to fixing the gauge for the lattice since for each configuration we are choosing one single gauge transformation that transforms the links to the spanning tree. We prove this statement in the following subsection.

3.1 Spanning Tree Fixes the Gauge

Consider a spanning tree gauge fixing done on a lattice with n sites, each site labeled by a natural number $m \in \{1, \ldots, n\}$ (the order of labeling, along with the labeling convention of adjacent

neighbors is irrelevant to the discussion and is not considered). Consider a gauge fixing being done by a spanning tree $\mathfrak{T}$ starting from $G(1) = \mathbb{I}$.
On such a gauge fixing, the gauge elements on sites m_1 and m_2, $G(m_1)$ and $G(m_2)$ are related by the ordered product of links in the subtree $\mathcal{S}_{m_1 m_2} \subset \mathfrak{T}$ connecting m_1 to m_2.
In other words

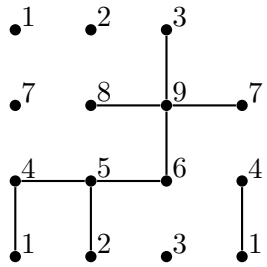
$$G(m_2) = G(m_1)U(m_1, i)U(i, j)\dots U(k, l)U(l, m_2), \quad U \in \mathcal{S}_{m_1,m_2} \tag{2}$$

In particular, since $G(1) = \mathbb{I}$, $G(m) = U(1, \cdot)\dots U(\cdot, m),\ U \in \mathcal{S}_{1,m}$.

Therefore, when a local gauge transformation $\Omega(n) = g_n$ is applied on the lattice, to preserve the spanning tree, the gauge fixing group elements on each site will transform as (since the intermediate site gauge transformations cancel out in the product of connected links)

$$G(m) \to G'(m) = g_1 G(m) g_m^{-1} \tag{3}$$

For example, in the lattice below



we start with $G(1) = \mathbb{I}$.
To fix the link $U(1, 4)$, we set $G(4) = U(1, 4) \implies U(1, 4) \to \mathbb{I} \cdot U(1, 4) \cdot U(1, 4)^{-1} = \mathbb{I}$.
Now, to fix $U(4, 5)$, we set $G(5) = U(1, 4) \cdot U(4, 5) \implies U(4, 5) \to U(1, 4) \cdot U(4, 5) \cdot (U(1, 4) \cdot U(4, 5))^{-1} = \mathbb{I}$, and so on

Therefore, one can easily see that $G(m) = U(1, \cdot)\dots U(\cdot, m)$.
As an example, under a local gauge transformation $G(5)$ transforms as $G(5) \to G'(5) = g_1 G(5) g_5^{-1}$.

Now consider an unfixed link $U(m_1, m_2)$ in the spanning tree. The value of the link before gauge transformation (with spanning tree gauge fixing) is

$$U(m_1, m_2) \to G(m_1) U_{uf}(m_1, m_2) G(m_2)^{-1} \tag{4}$$

Using eq (3), we see that in the locally gauge transformed lattice, the value of the unfixed link

would become

$$\begin{aligned} U'(m_1,m_2) =& G'(m_1)U'_{uf}(m_1,m_2)G'(m_2) = g_1G(m_1)g_{m_1}^{-1}\ g_{m_1}U_{uf}(m_1,m_2)g_{m_2}^{-1}\ g_{m_2}G(m_2)^{-1}g_1^{-1} \\ =& g_1G(m_1)U_{uf}(m_1,m_2)G(m_2)g_1^{-1} \end{aligned} \tag{5}$$

Therefore, under a local gauge transformation, all unfixed links on the lattice will transform as

$$U(m_1,m_2) \to g_1U(m_1,m_2)g_1^{-1} \tag{6}$$

Therefore, under spanning tree gauge fixing, a local gauge-transformed lattice becomes a global gauge-transformed configuration. In other words, the spanning tree gauge fixing maps local gauge copies of a configuration to global gauge transformed copies of the same configuration.

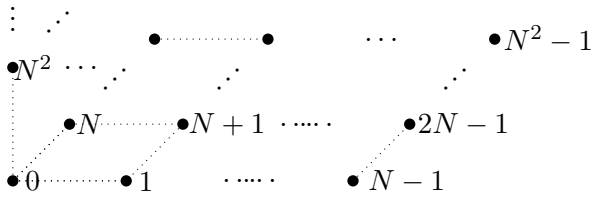
4 Enumerating the Spanning Trees

The Matrix Tree Theorem of Kirchhoff states that given a connected graph (of which the lattice we consider is also an example), the number of spanning trees of the graph is given by any cofactor of the Laplacian matrix for the graph.
The Laplacian Matrix of a graph is given as

$$Q = D - A \tag{7}$$

where D is the degree matrix, which is a diagonal matrix stating how many connections start or end at the i^{th} node, and $A = \{a_{ij}\}$ is the adjacency matrix, which is valued $a_{ij} = 1$ if i^{th} and j^{th} nodes have a connection between them, and 0 otherwise.

For the lattice in consideration, we decide to number the nodes as follows:



In this lattice, every node is adjacent to 2 nodes in each dimension. Along the first dimension, which is taken to be the one where node numbering increases by one as we traverse, the neighbors of n^{th} node are $n+1$ and $n-1$. If the node belongs to the *forward boundary* $(N-1,\ 2N-1, \ldots,\ N^d-1)$ then the neighbors of n are $n-(N-1)$ and $n+1$. Similarly for node belonging the *backward boundary*, $(0,\ N,\ 2N,\ \ldots,\ N^{d-1}(N-1))$, the neighbors are $n+1$ and $n+(N-1)$. Along the

second dimension, where the increment takes place by N, the nodes (not in boundary) have two neighbors $n+N$ and $n-N$. For nodes on *forward boundary* ($N^2-1,\ N^2-2,\ldots,\ N^2-N,\ 2N^2-1,\ 2N^2-2,\ldots,\ 2N^2-N,\ \ldots,\ N^d-1,\ N^d-2,\ \ldots,\ N^d-N$) in the second dimension, the neighbors are $n-N(N-1)$ and $n-N$, while for those in the *backward boundary* the neighbors are $n+N$ and $n+N(N-1)$. Similar analysis holds for the other dimensions too.

As an example, in a 3×3 lattice, each node is adjacent to 4 other nodes, with them being

$$0 \to \{1, 2, 3, 6\},\ 1 \to \{2, 0, 4, 7\},\ 2 \to \{0, 1, 5, 8\},$$
$$3 \to \{4, 5, 6, 0\},\ 4 \to \{5, 3, 7, 1\},\ 5 \to \{3, 4, 8, 2\},$$
$$6 \to \{7, 8, 0, 3\},\ 7 \to \{8, 6, 1, 4\},\ 8 \to \{6, 7, 2, 5\}$$

Therefore, for the lattice in consideration, the degree matrix, which gives the number of nodes adjacent to the said node, is simply $2\times d\times \mathbb{I}_{N^d}$ where d is the number of dimensions and N is the number of lattice sites in each dimension.
The adjacency matrix can be constructed with the knowledge of the neighbors of a given site, and therefore, one can calculate how many spanning trees exist in a given lattice.

As an example, for a 3×3 lattice, with the above given neighbours, the adjacency matrix and degree matrix are given as

$$A=\begin{pmatrix} 0&1&1&1&0&0&1&0&0\\ 1&0&1&0&1&0&0&1&0\\ 1&1&0&0&0&1&0&0&1\\ 1&0&0&0&1&1&1&0&0\\ 0&1&0&1&0&1&0&1&0\\ 0&0&1&1&1&0&0&0&1\\ 1&0&0&1&0&0&0&1&1\\ 0&1&0&0&1&0&1&0&1\\ 0&0&1&0&0&1&1&1&0 \end{pmatrix},\ D=2\times 2\begin{pmatrix} 1&0&0&0&0&0&0&0&0\\ 0&1&0&0&0&0&0&0&0\\ 0&0&1&0&0&0&0&0&0\\ 0&0&0&1&0&0&0&0&0\\ 0&0&0&0&1&0&0&0&0\\ 0&0&0&0&0&1&0&0&0\\ 0&0&0&0&0&0&1&0&0\\ 0&0&0&0&0&0&0&1&0\\ 0&0&0&0&0&0&0&0&1 \end{pmatrix} \tag{8}$$

The $(1,1)$ cofactor of $D-A$ is 11664, which gives the number of spanning trees on the 3×3 lattice. The number of spanning trees grows exponentially with the number of lattice sites available,

- A 3×3 lattice has 11,664 spanning trees
- A 4×4 lattice has 4,24,67,328 spanning trees
- A $3\times 3\times 3$ lattice has 25,29,99,02,31,17,90,46,912 spanning trees

The Mathematica code for counting the number of spanning trees on a hypercubic lattice with periodic boundary conditions is available in Appendix A.

4.1 Bruteforce enumeration algorithm

A general algorithm can be written down to enumerate all possible spanning trees as follows:

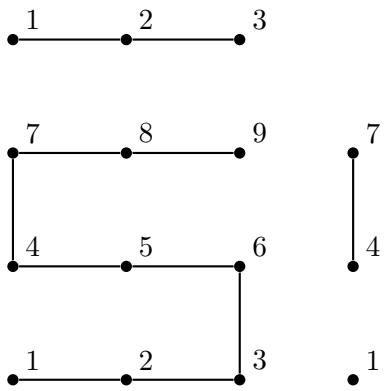
```
trees = collection of the spanning tree, visited sites, checked sites, along with the
  information if the spanning tree is completed or not
while there is at least one incomplete tree do
    for each tree in the set of trees do
        for all sites in tree do
            if site is not already checked then
                Add the neighbors that are not in the tree to the possible neighbors set
            end
            for each combination of the possible neighbors do
                Create a copy of the current tree and add the new links to the tree
                Add the said neighbors to the visited set of the tree
                if all the sites in the lattice belong to the visited set then
                    Mark the tree as completed
                end
                Append the new tree to the set of new trees
            end
        end
    end
    Delete duplicates from the set trees and replace the current set of trees with the set of
     new trees
end
```

The Mathematica code for enumerating all the spanning trees on a given hypercubic lattice with periodic boundary conditions can be found in Appendix B.

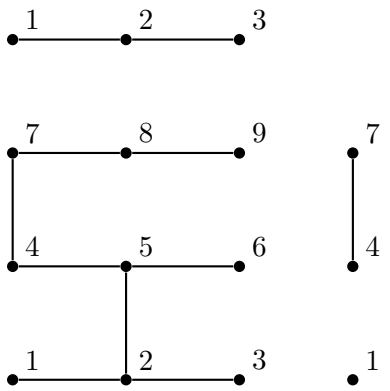
5 Gauge Transformations Between Two Spanning Trees

As a simple example behind the idea, consider two spanning trees,
Since in going from tree A to B, the unaltered links should have the same group elements on the sites, we have the constraints $G_1 = G_2 = G_3$, $G_6 = G_5 = G_4 = G_7 = G_8 = G_9$

Assume the convention $(i, j) \equiv U(i, j)$.
To do the gauge transformation, we use the requirement $(2, 5) \to G_2(2, 5)G_5^{-1} = \mathbb{I}$ to set $G_2 =$



(a) Maximal tree A



(b) Maximal tree B

$(2,5)^{-1}$ and $G_5 = \mathbb{I}$.
Setting $G_2 = (2,5)^{-1}$, we get $G_1 = (2,5)^{-1}$ & $G_3 = (2,5)^{-1}$, from which the link $(3,6) \to (2,5)^{-1}$. Therefore we can go from tree A to B with the local gauge transformation

$$G_n = \mathbb{I}, \forall n \neq 1,2,3; \quad G_1 = G_2 = G_3 = (2,5)^{-1} \tag{9}$$

In this example, the local gauge transformation is small. But there are trees between which the gauge transformation can become arbitarily large. As an example, the trees in Figure 1 are related by gauge transformation that involves a product of 28 group elements.

Following is a general procedure to find the gauge transformation between two trees:

```
modifiedLinks = the links of the gauge transformed lattice, set according to the first tree
toAdd = tree2 - tree1; i.e. the set of links in tree2 and not in tree1, and therefore to be
  added to tree1
gaugeTransformation = 𝕀(n_i); initialise the gauge transformation to the identity.
equalityConstraints = for a given site, gives the list of sites to be modified to preserve the
  unchanged links in the tree.
for every link in toAdd do
    Set the second site of the link (i.e., if the link is (n_i, n_j), then n_j) to be equal to
     U(n_i, n_j) (from the modified links) and the first site to be equal to 𝕀 to add the link.
    for every site in equalityConstraints[n_j] do
        multiply the existing gauge element at the site by U(n_i, n_j) to preserve the
         unchanged links
    end
end
```

In this code, we perform the transformation by adding the links of the second tree, which do not

exist in the first tree, to the first tree. We do this while respecting the equality constraints, i.e., for the links that are common to both the trees, the gauge elements on the sites at the two ends of the link should be equal. Adding the new links not in the first tree while respecting the equality constraints simply generates the second tree.
The Mathematica code implementing this procedure is available in Appendix C.

6 Conclusion

We have discussed the spanning tree method for fixing the gauge on a given lattice. This method provides a systematic approach to eliminate gauge redundancy while preserving the physical content of the theory. Further, we have also discussed brute force procedures for counting and finding all the spanning trees on a given lattice, which can be computationally intensive but comprehensive. We have also explored procedures to find gauge transformation between two spanning trees.

Additionally, we have touched upon the computational aspects of these methods, including their implementation in Mathematica. Future work could focus on optimizing these algorithms for larger lattices and exploring their applications in more complex gauge theories.

References

[1] P. Raff, "Spanning trees in grid graphs," Sept. 2008. arXiv:0809.2551 [math].

[2] R. P. Stanley, *Enumerative combinatorics. 1.* Cambridge studies in advanced mathematics, Cambridge: Cambridge University Press, 2. ed ed., 2012.

[3] C. Gattringer and C. B. Lang, *Quantum Chromodynamics on the Lattice: An Introductory Presentation*, vol. 788 of *Lecture Notes in Physics*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010.

A Counting the number of Spanning Trees on a Lattice

Get on GitHub

```
In[•]:= getBoundaries[nSites_, nDim_] :=
          Module[{ForwardBoundary, BackwardBoundary, d, n, n1},
           ForwardBoundary = {};
           For[d = 1, d ≤ nDim, d = d + 1,
            AppendTo[ForwardBoundary, {}];
            For[n = 1, n ≤ nSites^(nDim - d), n = n + 1,
             For[n1 = 0, n1 < nSites^(d - 1), n1 = n1 + 1,
              AppendTo[ForwardBoundary[[d]], n * nSites^d - n1 - 1]
             ]
            ]
           ];
           BackwardBoundary = {};
           For[d = 1, d ≤ nDim, d = d + 1,
            AppendTo[BackwardBoundary, {}];
            Do[AppendTo[BackwardBoundary[[d]], elem - (nSites^(d - 1)) (nSites - 1)],
             {elem, ForwardBoundary[[d]]}
            ]
           ];
           {ForwardBoundary, BackwardBoundary}
          ];

        getNeighbours[nSites_, nDim_] := Module[
           {boundaries, ForwardBoundary, BackwardBoundary, neighbour, nTot, n, d, nNeigh},
           boundaries = getBoundaries[nSites, nDim];
           ForwardBoundary = boundaries[[1]];
           BackwardBoundary = boundaries[[2]];
           nTot = nSites^nDim;
           neighbour = <||>;
           For[n = 0, n < nTot, n = n + 1,
            nNeigh = {};
            For[d = 0, d < nDim, d = d + 1,
             (* Forward *)
             If[Not[MemberQ[ForwardBoundary[[d + 1]], n]],
              AppendTo[nNeigh, n + nSites^d],
              AppendTo[nNeigh, n - (nSites^(d)) (nSites - 1)]];
             (* Backward *)
             If[Not[MemberQ[BackwardBoundary[[d + 1]], n]],
              AppendTo[nNeigh, n - nSites^d],
              AppendTo[nNeigh, n + (nSites^(d)) (nSites - 1)]]
            ];
            AppendTo[neighbour, n → nNeigh];
           ];
           neighbour];

        numberOfSpanningTrees[nSites_, nDim_] := Module[
           {adj, neighbour, nTot, deg, Q},
           adj = {};
           neighbour = getNeighbours[nSites, nDim];
           nTot = nSites^nDim;
           Do[
```

```
    AppendTo[adj, {}];
    Do[
     AppendTo[adj[[i + 1]], If[MemberQ[neighbour[i], j], 1, 0]], {j, Range[0, nTot - 1]}
    ],
    {i, Range[0, nTot - 1]}];
   deg = 2 * nDim * IdentityMatrix[nSites^nDim];
   Q = adj - deg;
   Det[Drop[Q, {1}, {1}]]
  ];
```

In[•]:=

```
(*For a 3x3 lattice*)
nSites = 3;
nDim = 2;
neighbour = getNeighbours[nSites, nDim]
Print["Number of Spanning Trees: "]
numberOfSpanningTrees[nSites, nDim]
```

Out[•]=

```
⟨|0 → {1, 2, 3, 6}, 1 → {2, 0, 4, 7}, 2 → {0, 1, 5, 8}, 3 → {4, 5, 6, 0},
 4 → {5, 3, 7, 1}, 5 → {3, 4, 8, 2}, 6 → {7, 8, 0, 3}, 7 → {8, 6, 1, 4}, 8 → {6, 7, 2, 5}|⟩
```

```
Number of Spanning Trees:
```

Out[•]=

```
11 664
```

In[•]:=

```
(*For a 4x4 lattice*)
nSites = 4;
nDim = 2;
neighbour = getNeighbours[nSites, nDim]
Print["Number of Spanning Trees: "]
numberOfSpanningTrees[nSites, nDim]
```

Out[•]=

```
⟨|0 → {1, 3, 4, 12}, 1 → {2, 0, 5, 13}, 2 → {3, 1, 6, 14}, 3 → {0, 2, 7, 15},
 4 → {5, 7, 8, 0}, 5 → {6, 4, 9, 1}, 6 → {7, 5, 10, 2}, 7 → {4, 6, 11, 3},
 8 → {9, 11, 12, 4}, 9 → {10, 8, 13, 5}, 10 → {11, 9, 14, 6}, 11 → {8, 10, 15, 7},
 12 → {13, 15, 0, 8}, 13 → {14, 12, 1, 9}, 14 → {15, 13, 2, 10}, 15 → {12, 14, 3, 11}|⟩
```

```
Number of Spanning Trees:
```

Out[•]=

```
-42 467 328
```

In[•]:=
```
(*For a 3x3x3 lattice*)
nSites = 3;
nDim = 3;
neighbour = getNeighbours[nSites, nDim]
Print["Number of Spanning Trees: "]
numberOfSpanningTrees[nSites, nDim]
```

Out[•]=

```
⟨| 0 → {1, 2, 3, 6, 9, 18}, 1 → {2, 0, 4, 7, 10, 19}, 2 → {0, 1, 5, 8, 11, 20},
 3 → {4, 5, 6, 0, 12, 21}, 4 → {5, 3, 7, 1, 13, 22}, 5 → {3, 4, 8, 2, 14, 23},
 6 → {7, 8, 0, 3, 15, 24}, 7 → {8, 6, 1, 4, 16, 25}, 8 → {6, 7, 2, 5, 17, 26},
 9 → {10, 11, 12, 15, 18, 0}, 10 → {11, 9, 13, 16, 19, 1}, 11 → {9, 10, 14, 17, 20, 2},
 12 → {13, 14, 15, 9, 21, 3}, 13 → {14, 12, 16, 10, 22, 4}, 14 → {12, 13, 17, 11, 23, 5},
 15 → {16, 17, 9, 12, 24, 6}, 16 → {17, 15, 10, 13, 25, 7}, 17 → {15, 16, 11, 14, 26, 8},
 18 → {19, 20, 21, 24, 0, 9}, 19 → {20, 18, 22, 25, 1, 10}, 20 → {18, 19, 23, 26, 2, 11},
 21 → {22, 23, 24, 18, 3, 12}, 22 → {23, 21, 25, 19, 4, 13}, 23 → {21, 22, 26, 20, 5, 14},
 24 → {25, 26, 18, 21, 6, 15}, 25 → {26, 24, 19, 22, 7, 16}, 26 → {24, 25, 20, 23, 8, 17} |⟩
```

```
Number of Spanning Trees:
```

Out[•]=

```
2 529 990 231 179 046 912
```

B Enumerating Spanning Trees

Get on GitHub

```
In[ ]:= SetDirectory[NotebookDirectory[]]

In[ ]:= (* To find the neighbours of a given site,
      given the number of site elements in one dimension and the number of dimensions. *)

      (*In each dimension, there are two bounding surfaces. Here we determine the elements
       of the two surfaces for all the dimensions. The boundaries are in the format
       {{x axis boundary points}, {y axis boundary points} ... } *)
      getBoundaries[nSites_, nDim_] :=
        Module[{ForwardBoundary, BackwardBoundary, d, n, n1},
         ForwardBoundary = {};
         For[d = 1, d ≤ nDim, d = d + 1,
          AppendTo[ForwardBoundary, {}];
          For[n = 1, n ≤ nSites^(nDim - d), n = n + 1,
           For[n1 = 0, n1 < nSites^(d - 1), n1 = n1 + 1,
            AppendTo[ForwardBoundary[[d]], n * nSites^d - n1 - 1]
           ]
          ]
         ];
         BackwardBoundary = {};
         For[d = 1, d ≤ nDim, d = d + 1,
          AppendTo[BackwardBoundary, {}];
          Do[AppendTo[BackwardBoundary[[d]], elem - (nSites^(d - 1)) (nSites - 1)],
           {elem, ForwardBoundary[[d]]}
          ]
         ];
         {ForwardBoundary, BackwardBoundary}
        ];

      (* Each site element will have 2*nDim neighbours,
      i.e. 2 in each dimension. Here we determine the
       2 neighbours of each element in each of the dimensions *)
      getNeighbours[nSites_, nDim_] :=
        Module[{boundaries, ForwardBoundary, BackwardBoundary, neighbour, nTot, n, d, nNeigh},
         boundaries = getBoundaries[nSites, nDim];
         ForwardBoundary = boundaries[[1]];
         BackwardBoundary = boundaries[[2]];
         nTot = nSites^nDim;
         neighbour = <||>;
         For[n = 0, n < nTot, n = n + 1,
          nNeigh = {};
          For[d = 0, d < nDim, d = d + 1,
           (* Forward *)
           If[Not[MemberQ[ForwardBoundary[[d + 1]], n]],
            AppendTo[nNeigh, n + nSites^d], AppendTo[nNeigh, n - (nSites^(d)) (nSites - 1)]];
           (* Backward *)
           If[Not[MemberQ[BackwardBoundary[[d + 1]], n]],
```

```
        AppendTo[nNeigh, n - nSites^d],
        AppendTo[nNeigh, n + (nSites^(d)) (nSites - 1)]]
      ] ×
      AppendTo[neighbour, n → nNeigh];
    ];
    neighbour];

treeEqualQ[Tree1_, Tree2_] := (Tree1[[1]] === Tree2[[1]] && Tree1[[3]] === Tree2[[3]]);

getTrees[nSites_, nDim_] := Module[
    {neighbour, Trees, tree, change, count, generatedTrees, newTrees, visited, completed,
     checked, neighbours, unvisitedNeighbours, possibleCombinations, newTree,},

    neighbour = getNeighbours[nSites, nDim];

    Trees = {{{}, {0}, {}, False}}; (*links in the tree ,
    visited sites,checked sites, complete or incomplete*)
change = True;
    SetSharedVariable[change];
While[change,
     change = False; (* while there is atleast one incomplete tree *)
     generatedTrees = WaitAll[
       ParallelTable[(*For each tree*)
        newTrees = {};
        visited = tree[[2]];
        checked = tree[[3]];
        completed = tree[[4]];
        If[completed, AppendTo[newTrees, tree],
         change = True;
         Do[(*For each visited site*)
          If[Not[MemberQ[checked, visitedSite]], (*If site is not already checked *)
           neighbours = neighbour[visitedSite];
           unvisitedNeighbours = Complement[neighbours, visited];
           If[unvisitedNeighbours == {}, , possibleCombinations =
             Subsets[unvisitedNeighbours, {1, Length[unvisitedNeighbours]}];

            Do[(*For each possible combination of links that can be added*)
             newTree = tree;
             Do[(*Add the links to the tree,
              and the sites to visited*)AppendTo[newTree[[1]], {visitedSite, site}];
              newTree[[1]] = Sort[newTree[[1]]];
              AppendTo[newTree[[2]], site];
              AppendTo[newTree[[3]], visitedSite];
              newTree[[3]] = Sort[newTree[[3]]];
              , {site, combination}];
             (*Check if the newTree created has visited all elements*)
             If[Sort[newTree[[2]]] == Range[0, nSites^nDim - 1], newTree[[4]] = True, ,];
```

```
            (* Append the new tree
             to the net of newtrees *)AppendTo[newTrees, newTree];
            , {combination, possibleCombinations}
           ]
          ]]
         , {visitedSite, visited}
        ]
       ]; newTrees
       , {tree, Trees}
      ]];
     Trees = DeleteDuplicates[Flatten[generatedTrees, 1], treeEqualQ];
     (* delete duplicates and replace the current set of trees by the new set *)
    ];
    Trees];


(* Generating all possible trees for a 3x3 lattice *)

nSites = 3;
nDim = 2;
neighbour = getNeighbours[nSites, nDim];
Trees = getTrees[nSites, nDim]
```

In[•]:=
```
DumpSave[StringJoin[{"Trees_", ToString[nSites] , "_" , ToString[nDim], ".mx"}], Trees]
```

Out[•]=

```
{{{{{0, 1}, {1, 2}, {2, 5}, {3, 4}, {4, 7}, {5, 3}, {6, 8}, {7, 6}}, {0, 1, 2, 5, 3, 4, 7, 6, 8},
   {0, 1, 2, 3, 4, 5, 6, 7}, True}, {{{0, 1}, {1, 2}, {2, 5}, {3, 4}, {4, 7}, {5, 3}, {7, 8}, {8, 6}},
   {0, 1, 2, 5, 3, 4, 7, 8, 6}, {0, 1, 2, 3, 4, 5, 7, 8}, True}, ··· 11660 ···,
  {{{0, 1}, {0, 2}, {0, 3}, {0, 6}, {4, 5}, {6, 7}, {6, 8}, {7, 4}}, {0, 1, 2, 3, 6, 7, 8, 4, 5},
   {0, 0, 0, 0, 4, 6, 6, 7}, True}, {{{0, 1}, {0, 2}, {0, 3}, {0, 6}, {5, 4}, {6, 7}, {6, 8}, {8, 5}},
   {0, 1, 2, 3, 6, 7, 8, 5, 4}, {0, 0, 0, 0, 5, 6, 6, 8}, True}}}
```

Full expression not available (original memory size: 16 MB)

In[•]:=
```
DumpSave[
 StringJoin[{"neighbours_", ToString[nSites] , "_" , ToString[nDim], ".mx"}], neighbour]
```

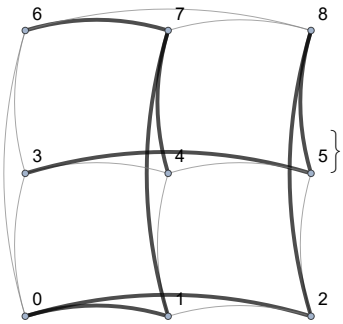
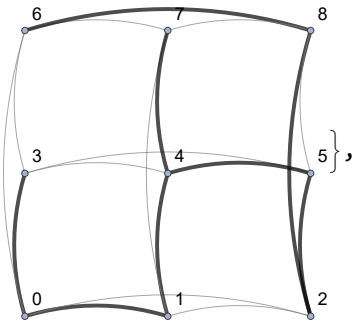
Out[•]=

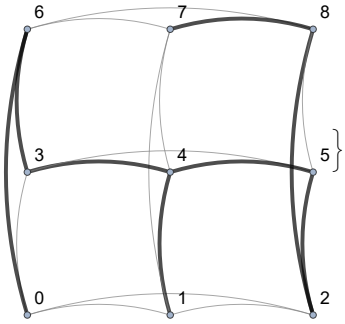
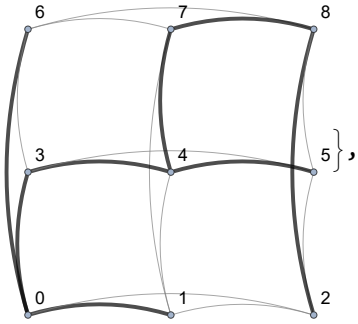
```
<|0 → {1, 2, 3, 6}, 1 → {2, 0, 4, 7}, 2 → {0, 1, 5, 8}, 3 → {4, 5, 6, 0},
  4 → {5, 3, 7, 1}, 5 → {3, 4, 8, 2}, 6 → {7, 8, 0, 3}, 7 → {8, 6, 1, 4}, 8 → {6, 7, 2, 5}|>
```

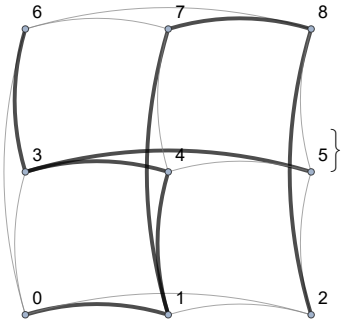
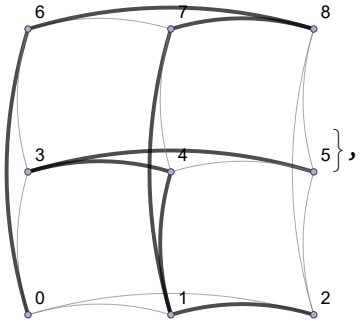
In[•]:=
```
Get[StringJoin[{"Trees_", ToString[nSites] , "_" , ToString[nDim], ".mx"}]]
(*Get the object "Trees"*)
```

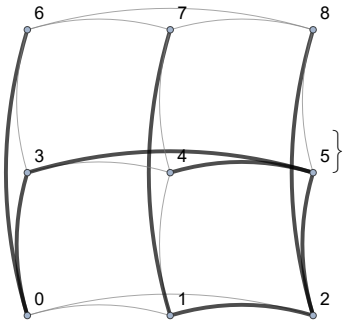
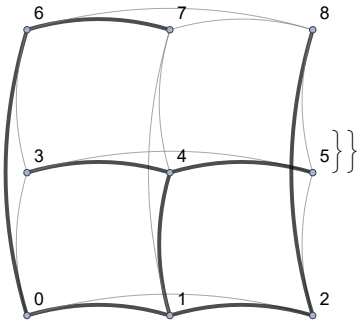
```
In[•]:= getVertexCoords[nSites_, nDim_] := Module[ (*Only for 2D lattices*)
          {nTot, n, x, y},
          vertexPos = {};
          nTot = nSites^nDim;
          x = 0;
          y  = 0;
          For[n = 0, n < nTot, n = n + 1,
           AppendTo[vertexPos, n → {x, y}];
           If[Mod[n, nSites] == nSites - 1, x = 0; y = y + 1, x = x + 1];
          ];
          vertexPos
         ]
        showTree[nSites_, nDim_, tree_] := Module[
          {links, neighbour, n, nTot, treeLinks},
          links = {};
          nTot = nSites^nDim;
          neighbour = getNeighbours[nSites, nDim];
          For[n = 0, n < nTot, n = n + 1,
           Do[
            AppendTo[links,
             UndirectedEdge[Sort[{n, i}][[1]], Sort[{n, i}][[2]]], {i, neighbour[n]}
           ]
          ];
          treeLinks = Table[UndirectedEdge[link[[1]], link[[2]]], {link, tree}];
          HighlightGraph[ Graph[DeleteDuplicates[links], VertexLabels → "Name",
            VertexCoordinates → getVertexCoords[nSites, nDim], EdgeShapeFunction → "CurvedEdge",
            EdgeStyle → {Gray}], Style[treeLinks, {Black, Thick}], ImageSize → Small]
         ]

In[•]:= (* Showing 8 random trees from the generated
          trees. The thin gray lines are the links of the lattice,
        and the thick black lines are the links set to I via spanning tree*)
        Table[index = RandomChoice[Range@Length@Trees];
         {StringJoin["Index: ", ToString@index] → showTree[nSites, nDim, Trees[[ index, 1]]]},
         {i, Range[1, 8]}]
```

Out[•]=

{{Index: 4974 →  , {Index: 5664 →  ,

{Index: 3572 →  , {Index: 10960 →  ,

{Index: 981 →  , {Index: 3987 →  ,

{Index: 9421 →  , {Index: 6761 →  }}

C Gauge Transformation Between Spanning Trees

Get on GitHub

```
In[•]:= SetDirectory[NotebookDirectory[]]

In[•]:= (* In the modules below,
        we have cosidered the notation that {l1, l2} = {l2, l1}^{-1} *)

        (*List which site is connected to which other sites via the spanning tree*)
        getConnections[nSites_, nDim_, tree_] := Module[{treeConnections, sites},
           treeConnections = <||>;
           sites = Range[0, nSites^nDim - 1];
           Do[
            AppendTo[treeConnections, site → {}];
            Do[
             If[ IntersectingQ[{Sort[{site, site1}]}, tree],
              AppendTo[treeConnections[site], site1]
             ]
             , {site1, sites}]
            , {site, sites}];
           treeConnections];

        (* We follow the norm that the 0th site is set to I. Starting from this,
        we traverse the tree and obtain the gauge elements
         on each site that gives rise to the said spanning tree *)
        getGaugeTransformation[nSites_, nDim_, tree_] := Module[
           {GaugeTransformation, sites, treeConnections, fixedSites, newFixedSites},
           sites = Range[0, nSites^nDim - 1];
           GaugeTransformation = <|sites[[1]] → {}|>;
           (* Group elements on each site, site → element*)

           treeConnections = getConnections[ nSites, nDim, tree];
           fixedSites = {{sites[[1]]}}; (*Starting from 0*)
           While[Not[Equal[Sort[Flatten[fixedSites]], sites]],
            (* While all sites are not fixed*)
            newFixedSites = {};
            (*Start from 0,
            and fix all the other end points of links starting from 0. These new end points
             that are fixed go to the newFixedSites, on which the next loop will run *)
            Do[
             Do[
              If[Not[KeyExistsQ[GaugeTransformation, connectedSite]],
               (*If site is not already fixed*)
               AppendTo[GaugeTransformation, connectedSite →
                 Join[GaugeTransformation[site], {{site, connectedSite}}] ]; (*on the left,
               the gauge transformation on 'site' will be acting. To set the link to I,
               we need to set the gauge transformation on the 'connectedSite' to '
                GaugeTransformation[site] x link' *)AppendTo[newFixedSites, connectedSite]]
              , {connectedSite, treeConnections[site]}
             ],
             {site, Last[fixedSites]}];
            AppendTo[fixedSites, newFixedSites];
           ];
           GaugeTransformation];
```

```
(* Given two trees,
equalityConstraints[site] gives the sites to be modified in order to preserve
 the unchanged linkes while setting the value of the lattice site 'site' *)
getEqualSites[ nSites_, nDim_, tree1_, tree2_] :=
  Module[{sites, unchanged, equalityConstraints},
   sites = Range[0, nSites^nDim - 1];
   unchanged = Intersection[tree1, tree2];
   equalityConstraints = <||>;
   Do[
    AppendTo[equalityConstraints, site → {}];
    Do[
     If[ IntersectingQ[{Sort[{site, site1}]}, unchanged],
      AppendTo[equalityConstraints[site], site1]
     ]
     , {site1, sites}]
    , {site, sites}];
   equalityConstraints];

(* Returns all the links (unidirectional) for a given lattice *)
getLinks [nSites_, nDim_, neighbour_] := Module[{sites, links},
   sites = Range[0, nSites^nDim - 1];
   links = <||>;
   (* Association of all links → values. Please mind the abuse of notations. I
     am using (l1, l2) for denoting both the connection between l1 and l2,
   and also the value of the link element between l1 and l2*)
   Do[
    Do[
     If [Not[KeyExistsQ[links, Sort[{site, siteNeighbour}]]]
      , AppendTo[links, {site, siteNeighbour} → {site, siteNeighbour} ]
     ]
     , {siteNeighbour, neighbour[site]}]
    , {site, sites}];
   links];

(* Given a spanning tree,
obtain the gauge transformation using getGaugeTransformation,
and act upon the links by the gauge transformation*)
getModifiedLinks[nSites_, nDim_, links_, tree_] :=
  Module[{modifiedLinks, gaugeTransformation},
   modifiedLinks = <||>;
   gaugeTransformation = getGaugeTransformation[ nSites, nDim, tree];
   Do[
    AppendTo[modifiedLinks, link → Join[gaugeTransformation[link[[1]]],
       {link}, Reverse[gaugeTransformation[link[[2]]], {1, 2}]] ]
    (*The inverse of a product of gauge elements reverses the order
     of multiplication. At the same time we also reverse the
     order in the links since {l1, l2}^{-1} = {l2, l1}*)
    , {link, links}];
   modifiedLinks];
```

```
(* Given two spanning trees,
this obtains the gauge transformation relating one to another *)
gaugeTransFormationBetweenTwoTrees[nSites_, nDim_, tree1_, tree2_, neighbour_] :=
  Module[{links, modifiedLinks, toAdd, gaugeTransformation1to2,
    equalityConstraints, addedSites, added, newAdded, equalSites},

   links = getLinks[nSites, nDim, neighbour];
   modifiedLinks = getModifiedLinks[nSites, nDim, links, tree1];
   toAdd = Complement[tree2, tree1];
   (*The links that are to be added to the first tree*)
   gaugeTransformation1to2 = <||>;
   Do[AppendTo[gaugeTransformation1to2, site → {}],
    {site, Range[0, nSites^nDim - 1]}];
   (* We start off with an identity element,
   i.e. for each site there is no gauge element *)

   equalityConstraints = getEqualSites[nSites, nDim, tree1, tree2];

   Do[(*For each link to be added*)

    (* for a link {l1, l2},
    we set G[l1] = I and G[l2] = {l1, l2} to enforce the gauge transformation *)

    (* here we obtain first, all the sites that should
     be set equal to G[l2] according to the equality constraints *)
    addedSites = {{link[[2]]}};
    added = True;
    While[added,
     added = False;
     newAdded = {};
     Do[
      Do[
       If[Not@MemberQ[Flatten@addedSites, connected],
        added = True; AppendTo[newAdded, connected]]
       , {connected, equalityConstraints[site]}]
      , {site, Last[addedSites]}];
     AppendTo[addedSites, newAdded];
    ]; (* The above while loop, for every site added,
    checks the equality constraints and adds the other required sites also *)
    equalSites = Flatten@addedSites;

    Do[(* For each site to be set for the given link,
     i.e. for each site in equalSites*)
     gaugeTransformation1to2[site] =
       Join[gaugeTransformation1to2[[site]], modifiedLinks[link] ];
     , {site, equalSites}];

    , {link, toAdd}];
   gaugeTransformation1to2
  ];
```

```
In[•]:= nSites = 3;
        nDim = 2;
        Get[StringJoin[{"neighbours_", ToString[nSites] , "_" , ToString[nDim], ".mx"}]]
        (*Get the object "neighbour"*)
        Get[StringJoin[{"Trees_", ToString[nSites] , "_" , ToString[nDim], ".mx"}]]
        (*Get the object "Trees"*)

In[•]:= tree1 = Map[Sort, Trees[[11, 1]]]
        tree2 = Map[Sort, Trees[[11664, 1]]]

Out[•]=
        {{0, 1}, {1, 2}, {2, 5}, {3, 4}, {3, 6}, {3, 5}, {6, 7}, {6, 8}}

Out[•]=
        {{0, 1}, {0, 2}, {0, 3}, {0, 6}, {4, 5}, {6, 7}, {6, 8}, {5, 8}}

In[•]:= gaugeTransFormationBetweenTwoTrees[ nSites, nDim, tree1, tree2, neighbour]

Out[•]=
        ‹|0 → {}, 1 → {}, 2 → {{0, 2}, {2, 1}, {1, 0}},
         3 → {{0, 2}, {2, 1}, {1, 0}, {0, 3}, {3, 5}, {5, 2}, {2, 1}, {1, 0}}, 4 → {},
         5 → {{0, 1}, {1, 2}, {2, 5}, {5, 3}, {3, 4}, {4, 5}, {5, 2}, {2, 1}, {1, 0}},
         6 → {{0, 1}, {1, 2}, {2, 5}, {5, 3}, {3, 4}, {4, 5}, {5, 2}, {2, 1}, {1, 0},
            {0, 1}, {1, 2}, {2, 5}, {5, 8}, {8, 6}, {6, 3}, {3, 5}, {5, 2}, {2, 1}, {1, 0}},
         7 → {{0, 1}, {1, 2}, {2, 5}, {5, 3}, {3, 4}, {4, 5}, {5, 2}, {2, 1}, {1, 0},
            {0, 1}, {1, 2}, {2, 5}, {5, 8}, {8, 6}, {6, 3}, {3, 5}, {5, 2}, {2, 1}, {1, 0},
            {0, 1}, {1, 2}, {2, 5}, {5, 8}, {8, 6}, {6, 3}, {3, 5}, {5, 2}, {2, 1}, {1, 0}},
         8 → {{0, 6}, {6, 3}, {3, 5}, {5, 2}, {2, 1}, {1, 0}, {0, 6}, {6, 3}, {3, 5}, {5, 2}, {2, 1},
            {1, 0}, {0, 1}, {1, 2}, {2, 5}, {5, 8}, {8, 6}, {6, 3}, {3, 5}, {5, 2}, {2, 1}, {1, 0}}|›
```