# Determinantal Factorizations of Trees: An Implementation Overview

Author Name Ahmet Halıcı

**Abstract**

Trees, as fundamental structures in combinatorial mathematics, offer unique insights into various mathematical and computational phenomena. One of the central mathematical representations of trees is the Laplacian matrix, which encapsulates crucial structural information about the tree. This paper delves into determinantal factorizations of the Laplacian matrix associated with trees, both directed and undirected. Drawing inspiration from the seminal work "A combinatorial theorem for trees," we present detailed Python implementations that illuminate the intricate relationships between tree structures and their Laplacian matrices. Specifically, we explore the effects of vertex and edge removals on the Laplacian matrix, providing practical algorithms and computational techniques. Furthermore, we distinguish between factorization approaches for directed and undirected trees, highlighting the nuances and complexities associated with each. Through our comprehensive exploration, we aim to bridge the gap between theoretical combinatorial properties and practical computational methods, offering readers both a deep understanding of tree factorizations and the tools to apply these concepts in real-world scenarios.

# 1 Introduction

Determinantal factorizations play a pivotal role in combinatorial mathematics, providing insights into the properties and behaviors of various mathematical structures. Among these, trees, fundamental in both theoretical and applied contexts, present unique opportunities for factorization. This whitepaper offers an implementation perspective on the determinantal factorizations of trees, spotlighting the Laplacian matrix's pivotal role.

# 2 Background

## 2.1 Mathematical Formulations

The Laplacian matrix of a graph, especially a tree, encapsulates vital structural information. It is defined as:

$$L(G) = D(G) - A(G)$$

where $L(G)$ is the Laplacian matrix, $D(G)$ is the degree matrix, and $A(G)$ is the adjacency matrix of the graph $G$.

The factorization of the Laplacian matrix, particularly the Cholesky factorization for positive-definite matrices, allows us to express the Laplacian as a product of matrices. This factorization can provide insights into the structure and properties of the underlying graph.

Trees are connected acyclic graphs, foundational in both computer science and discrete mathematics due to their inherent hierarchical structure and efficient traversal properties. They play pivotal roles in algorithms, data storage, networking, and numerous other computational domains. A critical tool in understanding and analyzing trees is the Laplacian matrix. This matrix, derived directly from a tree's structure, encodes essential information about the tree's vertices and edges. The Laplacian matrix's properties, such as its eigenvalues and determinants, offer deep insights into the tree's topological and combinatorial characteristics. For instance, the number of spanning trees, connectivity, and resistance distances can all be inferred from this matrix. As such, understanding the Laplacian matrix and its factorizations opens avenues to a plethora of applications and analyses in both theoretical and applied realms.

# 3 Combinatorial Properties of Laplacian Matrices

The Laplacian matrix of a graph encodes crucial structural information about the graph. Modifications to the graph, such as the removal of vertices or edges, have predictable effects on its Laplacian matrix. In this section, we detail Python implementations that compute the Laplacian matrix of a graph after such modifications.

## 3.1 Laplacian After Vertex Removal

Given the edges of a tree and a vertex to be removed, the following function computes the Laplacian matrix of the resulting tree:

```
import numpy as np
import networkx as nx

def laplacian_after_vertex_removal(tree_edges, vertex_to_remove):
    modified_edges = [edge for edge in tree_edges if
       vertex_to_remove not in edge]
    G_modified = nx.Graph(modified_edges)
    L_modified = nx.laplacian_matrix(G_modified).toarray()
    return L_modified
```

## 3.2 Laplacian After Edge Removal

Similarly, given the edges of a tree and an edge to be removed, the next function computes the Laplacian matrix of the resulting subgraphs:

```python
def laplacian_after_edge_removal(tree_edges, edge_to_remove):
    modified_edges = [edge for edge in tree_edges if edge !=
        edge_to_remove and edge[::-1] != edge_to_remove]
    G_modified = nx.Graph(modified_edges)
    components = list(nx.connected_components(G_modified))
    L_blocks =
        [nx.laplacian_matrix(G_modified.subgraph(component)).toarray()
        for component in components]
    L_modified = np.block([[L if i == j else np.zeros_like(L) for
        j, L in enumerate(L_blocks)] for i, _ in
        enumerate(L_blocks)])
    return L_modified
```

These functions utilize the `networkx` library, a powerful tool for the creation, manipulation, and study of complex networks.

# 4 Directed Tree Factorization

The factorization of the Laplacian matrix for directed trees requires a distinct approach compared to undirected trees. In this section, we present a Python class that handles this factorization for directed trees.

## 4.1 Initialization and Laplacian Computation

We initialize a directed graph from the provided edges and compute its Laplacian matrix as follows:

```python

import numpy as np
import networkx as nx

class DirectedTreeFactorization:
    def __init__(self, directed_edges):
        self.graph = nx.DiGraph()
        self.graph.add_edges_from(directed_edges)
        self.laplacian = None
        self.C = None

    def compute_laplacian(self):
        n = len(self.graph.nodes())
        self.laplacian = np.zeros((n, n))
        for edge in self.graph.edges():
            i, j = edge
            self.laplacian[i, i] += 1
            self.laplacian[i, j] -= 1
```

## 4.2 Matrix C Construction and Laplacian Factorization

For the factorization process, we construct an auxiliary matrix $C$ and validate the factorization of the Laplacian matrix:

```
    def construct_matrix_C(self):
        n = len(self.graph.nodes())
        self.C = np.zeros((n, n - 1))
        for col, edge in enumerate(self.graph.edges()):
            i, j = edge
            self.C[i, col] = 1
            self.C[j, col] = -1

    def factorize_laplacian(self):
        if self.laplacian is None:
            self.compute_laplacian()
        if self.C is None:
            self.construct_matrix_C()
        return np.allclose(self.laplacian, np.dot(self.C, self.C.T))
```

The factorization checks if the Laplacian matrix can be expressed as $C \times C^T$ where $C$ is the constructed matrix.

## 4.3 Algorithm Descriptions

Given the mathematical formulations, we can define specific algorithms to compute the modified Laplacian matrices, factorize them, and analyze their properties. The provided Python implementations offer a practical perspective on these algorithms, highlighting the computational methods to achieve the theoretical concepts.

# 5 Laplacian Factorization for Undirected Trees

Factorizing the Laplacian matrix of undirected trees provides insights into the tree's structure and combinatorial properties. The following Python class demonstrates the factorization process:

### 5.1 Initialization and Laplacian Computation

The initialization involves creating an undirected graph and computing its Laplacian matrix:

```
class TreeFactorization:
    def __init__(self, edges):
        self.graph = nx.Graph()
        self.graph.add_edges_from(edges)
        self.laplacian = None
        self.laplacian_perturbed = None
        self.lower_triangular = None

    def compute_laplacian(self):
        self.laplacian = nx.laplacian_matrix(self.graph).toarray()
```

### 5.2 Laplacian Perturbation and Factorization

To ensure the Laplacian matrix's positive definiteness, we introduce a minor perturbation to its diagonal. This enables us to perform Cholesky factorization:

```
    def perturb_laplacian(self, epsilon=1e-10):
        if self.laplacian is None:
            self.compute_laplacian()
        self.laplacian_perturbed = self.laplacian + epsilon *
           np.eye(self.laplacian.shape[0])

    def factorize_laplacian(self):
        if self.laplacian_perturbed is None:
            self.perturb_laplacian()
        self.lower_triangular =
           np.linalg.cholesky(self.laplacian_perturbed)

    def reconstruct_laplacian(self):
        if self.lower_triangular is None:
            self.factorize_laplacian()
        reconstructed = np.dot(self.lower_triangular,
           self.lower_triangular.T)
        return reconstructed
```

The class provides methods for perturbing the Laplacian matrix, factorizing it using Cholesky decomposition, and reconstructing the original matrix from the factorized form.

## 6 Conclusion

In this work, we presented an in-depth exploration of determinantal factorizations of the Laplacian matrices associated with trees. We provided a comprehensive overview of the combinatorial properties of Laplacian matrices, detailing the effects of vertex and edge

removals. The factorization processes for both directed and undirected trees were meticulously discussed, with Python implementations highlighting the computational aspects. These factorizations offer profound insights into the structure and combinatorial properties of trees, underscoring their significance in both theoretical and applied contexts. The provided Python implementations serve as a practical toolset for researchers and practitioners aiming to delve deeper into the realm of graph theory and its intricate interplay with matrix factorizations.

# 7 References


1. "A combinatorial theorem for trees" - Thomas Colcombet.

2. NetworkX Developers. *NetworkX: Python software for the study of the structure and dynamics of social networks*. Available at: `https://networkx.org/`

3. Numpy Developers. *Numpy: The fundamental package for scientific computing with Python*. Available at: `https://numpy.org/`


# Acknowledgements


The authors would like to acknowledge the invaluable contributions and insights from the broader research community in the field of combinatorial mathematics and graph theory. Special thanks go to Thomas Colcombet from CNRS, Universite Paris Diderot for his input and feedback.