4-2022

The Enumeration of Minimum Path Covers of Trees

Merielyn Sher
William & Mary

The Enumeration of Minimum Path Covers of Trees

A thesis submitted in partial fulfillment of the requirement
for the degree of Bachelor of Science in
Mathematics from William & Mary

by

Merielyn Sher

Accepted for Honors
(Honors, No Honors)

Charles R. Johnson
Charles Johnson, Advisor

Eric Swartz
Eric Swartz

Weizhen Mao
Weizhen Mao

Williamsburg, VA

The Enumeration of Minimum Path Covers of Trees

Merielyn Sher
Advisor: Dr. Charles Johnson

Department of Mathematics
College of William & Mary
Williamsburg, VA
April, 2022

Abstract

A path cover of a tree T is a collection of induced paths of T that are vertex disjoint and cover all the vertices of T. A minimum path cover (MPC) of T is a path cover with the minimum possible number of paths, and that minimum number is called the path cover number of T. A tree can have just one or several MPC's. Prior results have established equality between the path cover number of a tree T and the largest possible multiplicity of an eigenvalue that can occur in a symmetric matrix whose graph is that tree. We hope to gain insights into the different ways that maximum multiplicity occurs among the multiplicity lists of T by enumerating its MPC's. The overall strategy is to divide and conquer. Given any tree T, several techniques are introduced to decompose T into smaller components. Then, the number of MPC's of these smaller trees can be calculated and recombined to obtain the number of MPC's for the original tree T.

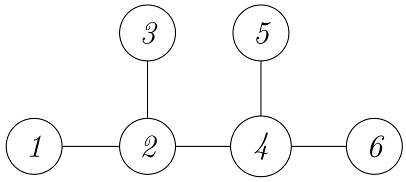
Acknowledgements

I would like to express my sincere gratitude to Professor Charles Johnson for his continuous support and guidance through my undergraduate career of study and research. I would also like to thank Professor Eric Swartz and Professor Weizhen Mao for serving on my committee and providing valuable feedback.

1 Introduction

Let A $= (a_{ij})$ be an $n \times n$ symmetric matrix. The graph of A, denoted G(A), is the simple undirected graph on n vertices with an edge $\{i, j\}$ if and only if $i \neq j$ and $a_{ij} \neq 0$. We use $\mathcal{S}(G)$ to denote the set of all symmetric matrices whose graph is G.

Example 1.1. *Here, G is a tree on 6 vertices. $A \in \mathcal{S}(G)$, and $a_{ij} \neq 0$.*



G

$$A = \begin{array}{c c} & \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} & \begin{pmatrix} a_{11} & a_{12} & 0 & 0 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & a_{24} & 0 & 0 \\ 0 & a_{32} & a_{33} & 0 & 0 & 0 \\ 0 & a_{42} & 0 & a_{44} & a_{45} & a_{46} \\ 0 & 0 & 0 & a_{54} & a_{55} & 0 \\ 0 & 0 & 0 & a_{64} & 0 & a_{66} \end{pmatrix} \end{array}$$

Our goal is to further understand the list of possible multiplicities of eigenvalues that occur for matrices in $\mathcal{S}(G)$ and how they might be related to the combinatorial features of G. Specifically, we focus on the case where G is a tree, and there has been extensive amount of research done on this topic. Notation-wise, given a tree T, let $\deg_T(v)$ denote the **degree** of a vertex v in T, which is the number of neighboring vertices of v. Let A be a matrix in $S(T)$. Then, for a vertex v in T, $A(v)$ denotes the principal submatrix of A resulting from deleting the row and column indexed by v. For a subtree T_i of T, $A[T_i]$ denotes the principal submatrix of A that includes only the rows and columns indexed by the vertices in T_i. A fundamental theorem on this subject is the Parter-Wiener theorem, and we include a generalization for it here [1].

Theorem 1.2. *Let T be a tree and let A be a matrix in $S(T)$. Suppose that there is a vertex v of T and a real number λ such that $\lambda \in \sigma(A) \cap \sigma(A(v))$, where $\sigma(A)$ denotes the spectrum of A. Then*

1. *there is a vertex u of T such that $m_{A(u)}(\lambda) = m_A(\lambda) + 1$, where $m_A(\lambda)$ and $m_{A(u)}(\lambda)$ are the respective algebraic multiplicity of λ in A and $A(u)$;*

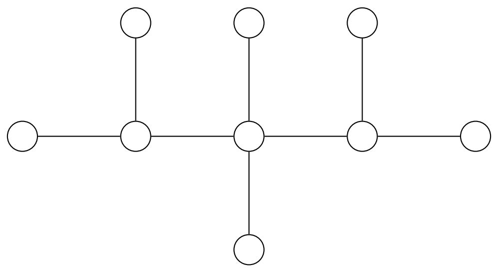
2. *if $m_A(\lambda) \geqslant 2$, then the prevailing hypothesis is automatically satisfied and u may be chosen so that $deg_T(u) \geqslant 3$ and so that there are at least three components T_1, T_2, and T_3 of $T - u$ such that $m_{A[T_i]}(\lambda) \geqslant 1$, $i = 1, 2, 3$; and*

3. *if $m_A(\lambda) = 1$, then u may be chosen so that $deg_T(u) \geqslant 2$ and so that there are two components T_1 and T_2 of $T - u$ such that $m_{A[T_i]}(\lambda) = 1$, $i = 1, 2$.*

A vertex v in T meeting the requirements of Theorem 1.2 is called a **Parter** vertex of T for λ.

The **multiplicity list** of a $n \times n$ matrix is a simple partition of n in which the parts are the multiplicities of the distinct eigenvalues. A multiplicity list is **ordered** when the multiplicities are ordered by the numerical values of the underlying eigenvalues. It is **unordered** when we list the multiplicities in descending order based on their own values. For example, for a matrix A on 8 vertices that has eigenvalues {-3, -1, -1, 1, 2, 2, 2, 5}, its ordered multiplicity list is {1, 2, 1, 3, 1}, and its unordered list is {3, 2, 1, 1, 1}. We sometimes write an unordered multiplicity list in its abbreviated form, that is, we remove all 1's from the list. Given a graph G, its **catalog** is the set of all unordered multiplicity lists of the matrices in $\mathcal{S}(G)$. One major constraint on the catalog of $\mathcal{S}(G)$ is the maximum multiplicity, $M(G)$, that is, the largest possible multiplicity that occurs for eigenvalues of matrices in $\mathcal{S}(G)$. The example here is drawn from Appendix A of [2], which is a database that records the catalogs for trees on fewer than 12 vertices.

Example 1.3. *Given a tree T on 9 vertices,*



its catalog with all 1's removed is {3 3; 3 2 2; 3 2; 3; 2 2 2; 2 2 ; 2}, and $M(T) = 3$.

Remarkable results have been obtained for trees on the relationship between their maximum mutiplicity and certain combinatorial features [2].

For a tree T, a **residual path maximizing set** is a collection of q vertices of T, whose removal from T leaves a forest of p paths such that $p - q$ is a maximum, and we use $\Delta(T)$ to denote the maximum value. A **path cover** of a tree T is a collection of induced paths of T that are vertex disjoint and cover all the vertices of T. A **minimum path cover** (MPC) of T is defined as a path cover with the minimum possible number of paths. The path cover number of T, denoted $P(T)$, is the number of paths in an MPC. A significant result is introduced in [3], in which a four-way equality between $M(T), P(T), \Delta(T)$, and $n - mr(T)$ is established, where n denotes the number of vertices of T and $mr(T)$ denotes the minimum rank among matrices in $\mathcal{S}(T)$.

Theorem 1.4. *[3] For each tree T on n vertices,*

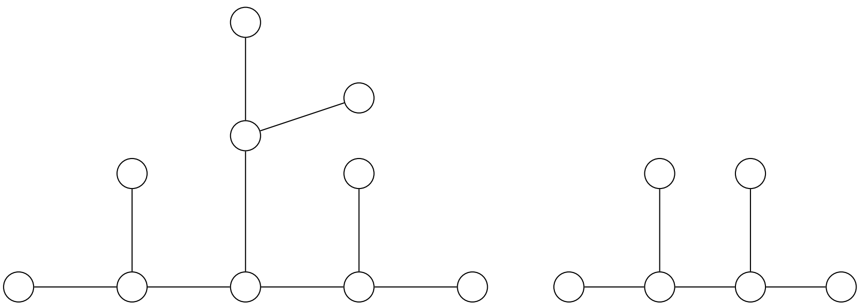
$$M(T) = P(T) = \Delta(T) = n - mr(T).$$

This allows us to obtain $M(T)$ directly from T. Since the proof for $M(T) = P(T)$ is completed through showing their respective equality to $\Delta(T)$, and the algorithms recorded in [2] for computing $P(T)$ is also through first identifying $\Delta(T)$, a more intuitive understanding of the direct relationship between $M(T)$ and $P(T)$ is desired.

In Example 1.3, notice that $M(T) = 3$ occurs in four different multiplicity lists. We are interested in characterizing the different ways $M(T)$ can occur for matrices in $\mathcal{S}(T)$. Because it has been established that $M(T) = P(T)$, a natural starting point and the main focus of this thesis is to investigate the different ways that $P(T)$ occurs for a given tree T, from which we then attempt to describe the multiple occurrences of $M(T)$.

Let $\boldsymbol{\mathcal{P}}(T)$ denote the set of all MPC's of T, and let $N(T)$ denote the number of distinct MPC's of T. So, $N(T) = |\boldsymbol{\mathcal{P}}(T)|$. A tree can have just one or several MPC's. The value of $N(T)$ is more closely related to the specific organization of vertices and edges in T than the number of vertices in T. For example, for a path T, $N(T) = 1$ regardless of its length. $N(T)$ increases drastically when the structure of T gets more complicated, as shown in the following example. Therefore, when enumerating $N(T)$, the main strategy here is to divide and conquer.

Example 1.5. *T_1 shown below has 19 distinct MPC's, and T_2 has a unique MPC.*



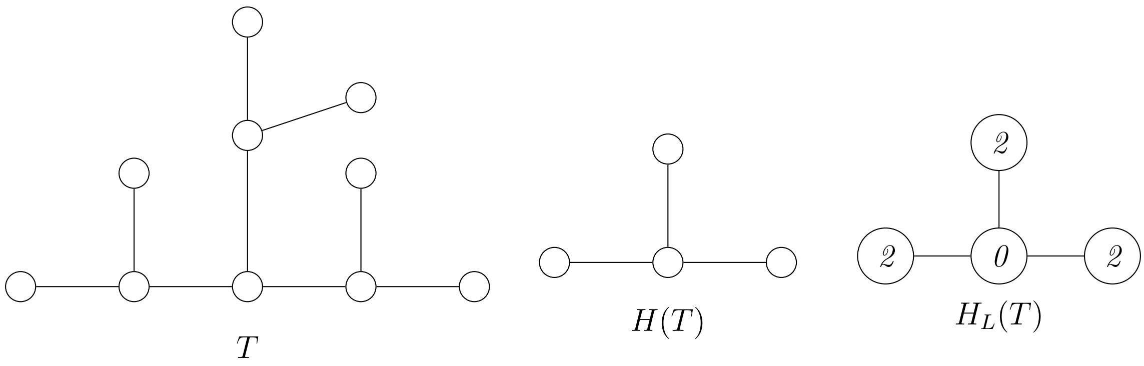
$P(T_1) = 4; N(T_1) = 19$ $\qquad$ $P(T_2) = 2; N(T_2) = 1$

We first introduce some background definitions and observations in Section 2 that associate certain properties of vertices and edges in a tree T with $N(T)$. Demonstrated in more detail in Sections 3 and 4, we characterize trees based on the structures of their vertices and edges and classify trees into different categories. Different techniques are discussed for different classes of trees that decompose a complex tree into smaller components $\{T_1, T_2, \cdots, T_k\}$ with rules to obtain $N(T)$ through recombining the values of $N(T_i)$'s. Section 5 deals with trees that cannot be further decomposed using the methods included in the previous two sections. A new algorithm is introduced to inductively enumerate $N(T)$ for such trees. Section 6 provides a complete algorithm, which combines techniques introduced in previous sections, that calculates $N(T)$ for any given tree T. An example is also included.

2 Background Observations

In a tree T, a **high degree vertex** (HDV) is one of degree at least 3. Otherwise, the vertex is of **low degree**. A **pendent vertex** of T is a vertex of degree 1. A tree is **linear** if all its HDV's lie on a single induced path. Otherwise it is **nonlinear**. The **Hi-graph**, $H(T)$, of T is the subgraph induced by its HDV's. $H(T)$ is a forest with one or more components (each of which is a tree). The **incremental degree** of a vertex v, $\delta(v)$, is the difference between its degrees in T and in $H(T)$. A **high-incremental degree** (HID) vertex in $H(T)$ is one of incremental degree at least 2; otherwise it is of **low-incremental degree** (LID). It is sometimes helpful to have all vertices in $H(T)$ labeled with their respective incremental degrees, and we call this a **labeled Hi-graph**, denoted $H_L(T)$.

Example 2.1. *Given T, its Hi-graph and labeled Hi-graph are shown on the right.*



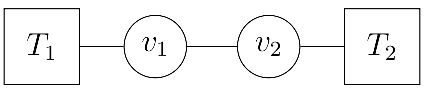
It is also helpful to characterize how edges are used in different MPC's of T when calculating $N(T)$. For a tree T with $N(T) = 1$, each of its edges is either included in the MPC or not. For a tree T with multiple MPC's, they differ from one another by including different sets of edges. We distinguish 3 statuses for edges in T.

Definition 2.2. *An edge is **absent** if it is used in no MPC of T. An edge is **required** if it is used in all MPC's. An edge is **discretionary** if it occurs in some but not all MPC's.*

We will look at ways of identifying edge statuses and their contributions to our knowledge of $N(T)$ both here and in later sections.

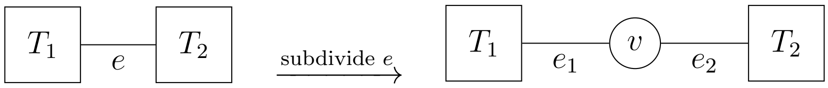
Lemma 2.3. *Any edge between two low degree vertices is required.*

Proof. Shown below is the general structure of two adjacent low degree vertices v_1 and v_2 in a tree T. T_1 and T_2 represent the subtrees connected respectively to v_1 and v_2 in T. One or both of them can be empty.



We will complete the proof by contradiction. Let $\mathcal{C}_1$ be an MPC of T. Assume that the edge (v_1, v_2) is not used in $\mathcal{C}_1$. Then, the two paths in $\mathcal{C}_1$ that respectively contain v_1 and v_2 also terminate at v_1 and v_2. Through merging these two path into one by including (v_1, v_2) and thus connecting v_1 and v_2, we construct a new path cover for T, $\mathcal{C}_2$. All the other paths in $\mathcal{C}_2$ are the same as in $\mathcal{C}_1$. The new path cover, $\mathcal{C}_2$, covers all the vertices in T and contains one fewer path than $\mathcal{C}_1$. Since we assumed $\mathcal{C}_1$ to be an MPC, a contradiction is reached. Therefore, (v_1, v_2) must be included in all MPC's of T. $\square$

Edge subdivision is the process where a new degree-2 vertex is positioned along an existing edge.



Lemma 2.4. *If T' is obtained by subdividing a required edge in T, then $N(T') = N(T)$.*

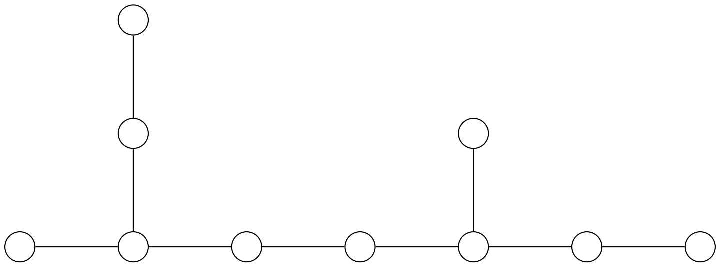
Proof. Suppose the edge e in the illustration above is required in T, we will first show that after subdividing e, the resulting edges e_1 and e_2 are both required in the new tree T'. Since we can always construct a path cover for T' from an MPC of T by including e_1, v, and e_2 in the path that originally includes e and keeping other paths unchanged, we have $P(T') \leqslant P(T)$. Without loss of generality, assume that e_1 is not required in T'. Then, for an MPC of T', $\mathcal{C}_{T'}$, that does not use e_1, either e_2 is used and v is included in a path that goes into T_2 or it is not used and v_2 is included in the MPC as a singleton. If e_2 is used in $\mathcal{C}_{T'}$, we can construct a corresponding path cover $\mathcal{C}_T$ for T that preserves the paths in $\mathcal{C}_{T'}$ except that the path that originally terminates at v now terminates at the vertex in T_2 that is connected to e. If e_2 is not used in $\mathcal{C}_{T'}$, we can also construct a corresponding path cover $\mathcal{C}_T$ for T that preserves all the paths in $\mathcal{C}_{T'}$ excluding v as a singleton. For both cases, $|\mathcal{C}_T| \leqslant |\mathcal{C}_{T'}| \leqslant P(T)$, meaning $\mathcal{C}_T$ is a *minimum* path cover of T. However, we assumed e to be required. A contradiction is reached. Therefore, e_1 and e_2 are both required in T, which means that e_1, v, and e_2 must be included in the same path in every MPC of T', making them equivalent to e, a single edge, in T. Therefore, we have $N(T') = N(T)$. $\square$

Corollary 2.5. *For a tree T, the value of $N(T)$ is independent of the lengths of the paths induced by the low degree vertices in T.*

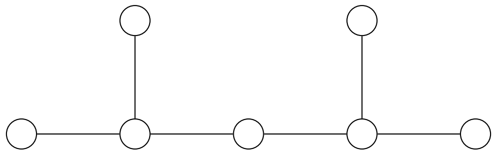
Theorem 2.6. *If for two trees, T_1 and T_2, $H_L(T_1)$ is isomorphic to $H_L(T_2)$, then $P(T_1) = P(T_2)$, and $N(T_1) = N(T_2)$.*

Proof. If $H_L(T_1)$ is isomorphic to $H_L(T_2)$, the only possible difference between T_1 and T_2 is the lengths of the paths induced by the low degree vertices in them. All other aspects of the two structures are the same. Then, by Corollary 2.5, $N(T_1) = N(T_2)$. $\square$

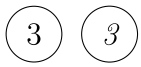
Example 2.7. *Here, even though T_1 and T_2 are two different trees, they share the same labeled Hi-graph and $N(T_1) = N(T_2) = 9$.*



T_1



T_2



$H_L(T_1)\&H_L(T_2)$

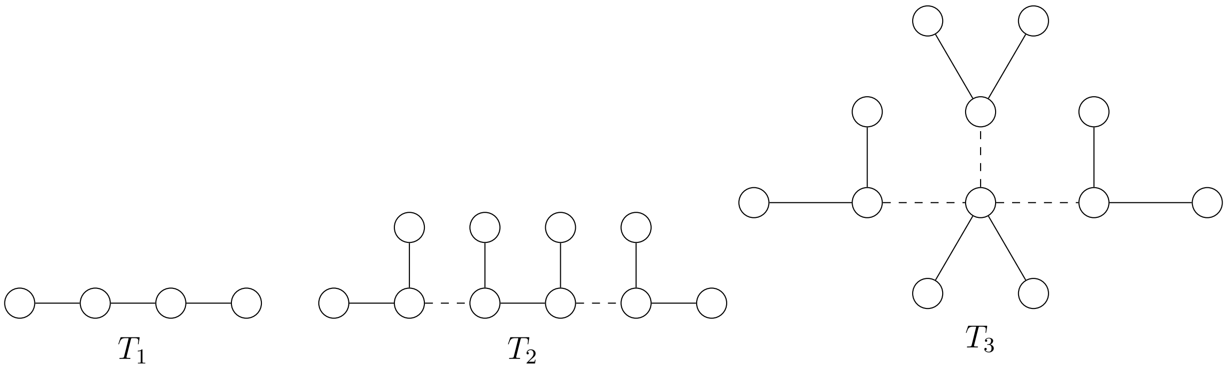
Remark 2.8. *For a tree T, an LID vertex cannot be a pendent vertex of $H(T)$. Furthermore, a vertex of incremental degree 0 in $H(T)$ must have at least 3 pendent arms. Otherwise, they would not have been included in the Hi-graph.*

By Theorem 2.6, $N(T)$ is entirely determined by its labeled Hi-graph. Therefore, while the examples and the proofs here often use trees with degenerate pendent paths for the sake of simplicity, the results can always be generalized to all trees with the same labeled Hi-graph.

2.1 Trees with a Unique MPC

We will deviate slightly before delving into the different techniques for enumerating MPC's. One problem that we have been interested in is to characterize all trees that have a unique MPC. However, among trees that have a unique MPC, there still exist a variety of structures. Both linear and nonlinear trees can have a unique MPC, as shown in the following example. Using the techniques developed in the remaining sections, it can be determined whether a given tree T has a unique MPC. However, we have not been able to directly describe all trees with a unique MPC using overt combinatorial features of T.

Example 2.9. *For trees T_1, T_2, and T_3, $N(T_1) = N(T_2) = N(T_3) = 1$.*
$P(T_1) = 1, P(T_2) = 3$, and $P(T_3) = 4$. The dashed edges are absent, and all the other edges are required.



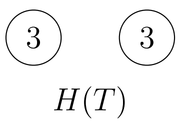
Remark 2.10. *A tree T has a unique MPC if and only if every edge in T is either required or absent.*

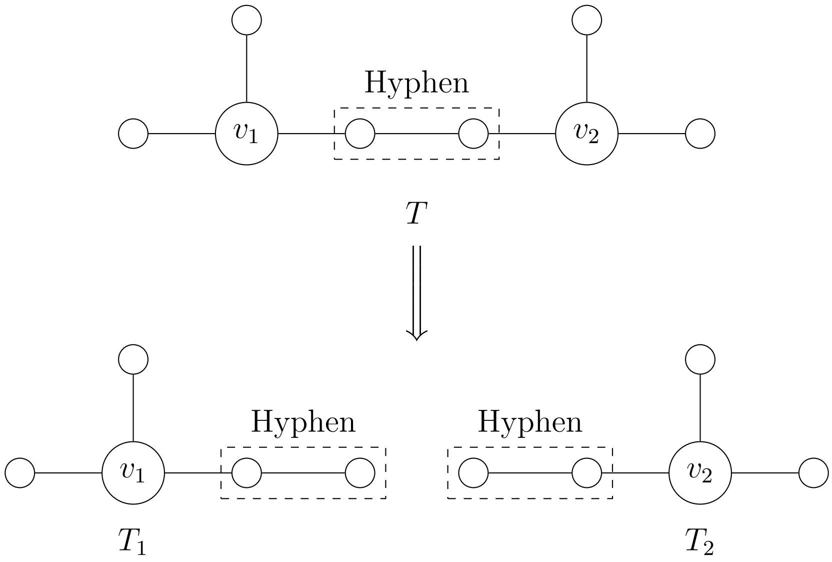
3 Trees with Multiple-Component Hi-graphs

The Hi-graph of a tree T has two or more components when there are one or more low-degree vertices on a single path between two of the HDV's. We will call such a path, induced by the low-degree vertex or vertices between two HDV's in the original tree, a **hyphen**.

Remark 3.1. *By Lemma 2.3 and Corollary 2.5, the edges in a hyphen are required and the value of $N(T)$ is independent of the lengths of the hyphens.*

Here we will first consider the case in which $H(T)$ is composed of two disjoint components. Note that in this case, there is exactly one hyphen in T. We consider the hyphen as a shared boundary between the two neighboring components. The two HDV's connected by the hyphen are denoted v_1 and v_2, respectively. The process of **hyphen decomposition** is defined as follows: we first separate the two components by removing the edge between the hyphen and v_2, with the resulting tree that contains v_1 denoted T_1; Similarly, T_2 is obtained by removing the edge between the hyphen and v_1 and selecting the part that includes v_2. Note that after the separation, both T_1 and T_2 include the hyphen. Below is a tree with a two-component Hi-graph with each component being a singleton of incremental degree 3. The hyphen is labeled, and the process of hyphen-decomposing T into T_1 and T_2 is displayed.





It is easier to enumerate $N(T_1)$ and $N(T_2)$ than to directly calculate $N(T)$. In fact, we will show that $N(T)$ is the product of the two.

Lemma 3.2. *Suppose that the Hi-graph of a tree T has two components. The hyphen in T is always included in a single path in every MPC of T_1 and T_2.*

Proof. Since T_1 and T_2 are named arbitrarily, we will only present the proof for T_1.

When the hyphen is a degenerate path, the statement is trivially true.

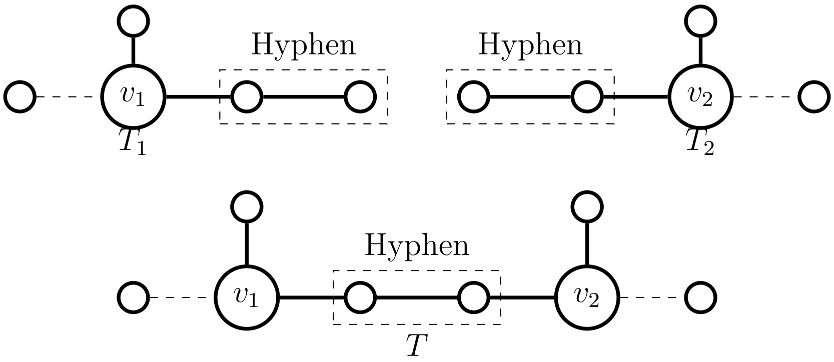
When the hyphen is composed of two or more vertices, every vertex in it is of low degree. Therefore, by Lemma 2.3, every edge in the hyphen is required, meaning the hyphen must be included in a single path in every MPC of T_1. The same argument applies to T_2. $\square$

Lemma 3.3. *Let $\mathcal{C}_1$ and $\mathcal{C}_2$ be two MPC's of T_1 and T_2, respectively. Then $\mathcal{C}_1$ and $\mathcal{C}_2$ can be merged into a path cover of T by taking the two paths in $\mathcal{C}_1$ and $\mathcal{C}_2$ that both contain the hyphen and merging them into one path. Note that two such paths must exist by Lemma 3.2. The other paths in $\mathcal{C}_1$ and $\mathcal{C}_2$ are unchanged by the merge. The resulting path cover is an MPC of T.*

Proof. An example of merging MPC's of T_1 and T_2 is shown below. We will complete the proof by contradiction. Suppose that the resulting path cover of T is not an MPC. Thus, the number of paths in it, k, is greater than $P(T)$. We have

$$k = |\mathcal{C}_1| + |\mathcal{C}_2| - 1 > P(T).$$

Because the edges in the hyphen are required in T, the reverse of the merging process can be applied on an MPC of T to obtain two path covers for T_1 and T_2. We apply the reverse-merging on an MPC of T, and the total number of paths in these two path covers is $P(T)+1$. But we have $|\mathcal{C}_1|+|\mathcal{C}_2|>P(T)+1$, suggesting that at least one of $\mathcal{C}_1$ and $\mathcal{C}_2$ is not minimum. A contradiction is reached. Therefore, the resulting path cover of T from merging $\mathcal{C}_1$ and $\mathcal{C}_2$ is an MPC of T. □



We now consider the relationship between $\boldsymbol{\mathcal{P}}(T_1)$, $\boldsymbol{\mathcal{P}}(T_2)$, and $\boldsymbol{\mathcal{P}}(T)$, the sets of MPC's of T_1, T_2, and T.

Theorem 3.4. *There exists a bijection between $\boldsymbol{\mathcal{P}}(T_1)\times\boldsymbol{\mathcal{P}}(T_2)$ and $\boldsymbol{\mathcal{P}}(T)$.*

Proof. Let $\mathcal{C}_1$ and $\mathcal{C}_2$ be two MPC's of T_1 and T_2, respectively. We will first construct a function f that maps from $\boldsymbol{\mathcal{P}}(T_1)\times\boldsymbol{\mathcal{P}}(T_2)$ to $\boldsymbol{\mathcal{P}}(T)$.

Define $f:\boldsymbol{\mathcal{P}}(T_1)\times\boldsymbol{\mathcal{P}}(T_2)\to\boldsymbol{\mathcal{P}}(T)$. The function f merges $\mathcal{C}_1$ and $\mathcal{C}_2$ in the same way as described in Lemma 3.3. Let $\mathcal{C}\in\boldsymbol{\mathcal{P}}(T)$ be the resulting MPC. We then have $f(\mathcal{C}_1,\mathcal{C}_2)=\mathcal{C}$ for $\mathcal{C}_1\in\boldsymbol{\mathcal{P}}(T_1),\mathcal{C}_2\in\boldsymbol{\mathcal{P}}(T_2),\mathcal{C}\in\boldsymbol{\mathcal{P}}(T)$.

For $\mathcal{C}_1,\mathcal{C}_1'\in\boldsymbol{\mathcal{P}}(T_1)$ and $\mathcal{C}_2,\mathcal{C}_2'\in\boldsymbol{\mathcal{P}}(T_2)$, if $(\mathcal{C}_1,\mathcal{C}_2)=(\mathcal{C}_1',\mathcal{C}_2')$, we show that $\mathcal{C}=f(\mathcal{C}_1,\mathcal{C}_2)=f(\mathcal{C}_1',\mathcal{C}_2')=\mathcal{C}'$. Given the merging procedure described above, it is impossible for two sets of identical MPC's of T_1 and T_2 to become distinct MPC's of T after the merge. Therefore, f is well-defined.

Now show that f is a bijection. We first prove that f is injective by showing that if $\mathcal{C}_1\neq\mathcal{C}_1'$ or $\mathcal{C}_2\neq\mathcal{C}_2'$, then $\mathcal{C}\neq\mathcal{C}'$. This is true by the merging process described above. Let f^{-1} be defined as the reverse merging process. An MPC of T gets split up into MPC's of T_1 and T_2. T_1 is the component that contains v_1 after the removal of the edge between the hyphen and v_2; Similarly, T_2 is the component that contains v_2 after the removal of the edge between the hyphen and v_1. Then, for each $\mathcal{C}^{(i)}\in\boldsymbol{\mathcal{P}}(T)$, we get $f^{-1}(\mathcal{C}^{(i)})=(\mathcal{C}_1^{(i)},\mathcal{C}_2^{(i)})$, with $(\mathcal{C}_1^{(i)},\mathcal{C}_2^{(i)})\in\boldsymbol{\mathcal{P}}(T_1)\times\boldsymbol{\mathcal{P}}(T_2)$. Therefore, f is surjective.

Because f is both injective and surjective, it is a bijection. □

Corollary 3.5. *Let T be a tree with a two-component Hi-graph, and let T_1 and T_2 be the results of hyphen-decomposing T. Then, $N(T)=N(T_1)N(T_2)$.*

Proof. As is shown in Lemma 1.3, there exists a bijection between $\boldsymbol{\mathcal{P}}(T_1) \times \boldsymbol{\mathcal{P}}(T_2)$ and $\boldsymbol{\mathcal{P}}(T)$. The cardinalities of the two sets are the same. Hence,
$N(T_1) \cdot N(T_2) = |\boldsymbol{\mathcal{P}}(T_1)||\boldsymbol{\mathcal{P}}(T_2)| = |\boldsymbol{\mathcal{P}}(T_1) \times \boldsymbol{\mathcal{P}}(T_2)| = |\boldsymbol{\mathcal{P}}(T)| = N(T)$. □

We generalize Corollary 3.5 to trees with Hi-graphs of k components and $k-1$ hyphens. For this type of tree, hyphen-decomposition can be applied at each of the $k-1$ hyphens in a sequential order to decompose T into k parts.

Algorithm 3.6. *Given a tree T with a k-component Hi-graph and $k-1$ hyphens,*

1. *Identify all the hyphens and assign indices $1, 2, \cdots, k-1$ to them. The order of the assignment does not matter as long as each hyphen is assigned exactly once.*

2. *Starting from $i = 1$, while $i \leqslant k-1$,*

   (a) decompose the component that includes the i-th hyphen into two subparts through applying hyphen-decomposition at the i-th hyphen, and

   (b) increment i by 1.

3. *When Step 2 is completed, we have decomposed T into k subparts.*

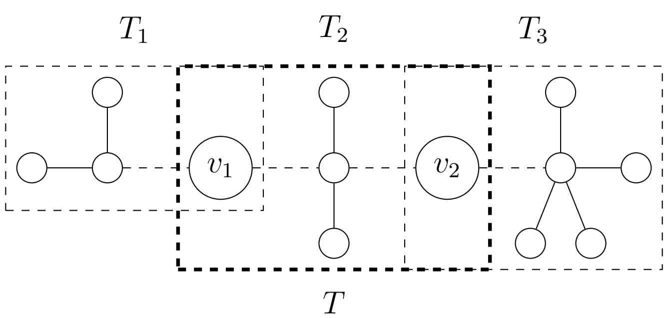
Theorem 3.7. *Suppose that the Hi-graph of a tree T consists of k disjoint components. If Algorithm 3.6 is applied and T is hyphen-decomposed into k disjoint components, $T_1, T_2, \cdots, T_k$, then*

$$N(T) = \prod_{i=1}^{k} N(T_i).$$

Proof. This is a direct result of Corollary 3.5 and the construction of Algorithm 3.6. □

Example 3.8. *For the tree T shown below that has a three-component Hi-graph, v_1 and v_2 are identified as the two hyphens. We then apply hyphen decomposition to T and get T_1, T_2, and T_3 as the resulting components. By Theorem 3.7,*

$$N(T) = N(T_1)N(T_2)N(T_3) = \binom{3}{2} \times \binom{4}{2} \times \binom{5}{2} = 180.$$



In this example, the resulting components are all stars. We will describe the calculation of $N(T)$ for trees with a connected Hi-graph in the next section.

4 Trees with Single-Component Hi-graphs

We are now able to calculate $N(T)$ for a tree T with a Hi-graph of multiple components when given $N(T_i)$'s for all its components resulting from hyphen decomposition. In this section, we will look only at trees with a connected Hi-graph and show how to enumerate $N(T)$.

First, there are certain simple trees of which we can calculate $N(T)$ without having to decompose further. An example is **generalized stars**.

Definition 4.1. *A* ***generalized star*** *(g-star) is a tree with at most one HDV.*

Proposition 4.2. *Let T be a g-star with d arms. Then,*

$$P(T) = d - 1 \quad \text{and} \quad N(T) = \binom{d}{2}.$$

Proof. In order to minimize the number of paths used in a path cover for T, one of the paths includes two of the arms as well as the central vertex. The other $(d-2)$ arms are included as disjoint paths in the MPC. Therefore, $P(T) = d - 1$. Because there are $\binom{d}{2}$ ways to select two arms to lie on the same path in an MPC, $N(T) = \binom{d}{2}$. □

For a tree T with a single-component Hi-graph that is not a g-star, there are ways to decompose it into smaller pieces, $T_1, T_2, \cdots, T_k$, enumerate $N(T_i)$ for each piece, and then calculate $N(T)$ given rules for recombination. **Absent-edge decomposition**, a process defined in Lemma 4.3, plays an important role when decomposing such trees.

Lemma 4.3. *Absent-edge decomposition is the process where a tree T is decomposed into smaller components $T_1, T_2, \cdots, T_k$ through the removal of all of its absent edges. Then,*

$$P(T) = \sum_{i=1}^{k} P(T_i) \quad \text{and} \quad N(T) = \prod_{i=1}^{k} N(T_i).$$

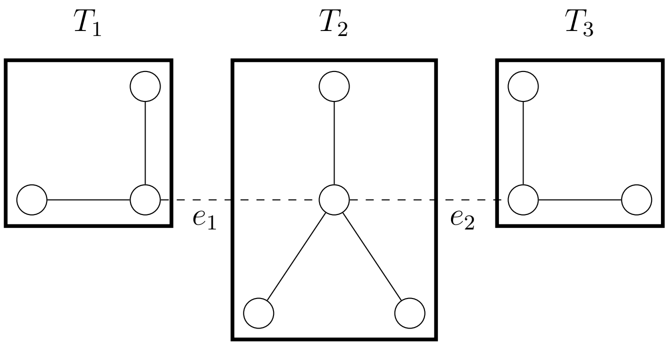
Proof. The absent edges of T are, by definition, not used in any MPC of T. Through removing all of them, an MPC of T can be viewed as a union of the MPC's of $T_1, T_2, \cdots, T_k$. Therefore, $P(T) = \sum_{i=1}^{k} P(T_i)$. The choice of which MPC of $\boldsymbol{\mathcal{P}}(T_i)$ to use for constructing an MPC for T is independent across different T_i's. Thus, we can write $\boldsymbol{\mathcal{P}}(T)$ as the product of $\boldsymbol{\mathcal{P}}(T_1), \boldsymbol{\mathcal{P}}(T_2), \cdots, \boldsymbol{\mathcal{P}}(T_k)$,

$$N(T) = |\boldsymbol{\mathcal{P}}(T)| = |\boldsymbol{\mathcal{P}}(T_1)||\boldsymbol{\mathcal{P}}(T_2)| \cdots |\boldsymbol{\mathcal{P}}(T_k)| = \prod_{i=1}^{k} N(T_i).$$

□

Example 4.4. *For the tree below, e_1 and e_2 are absent. Therefore, we apply absent-edge decomposition and decompose T into T_1, T_2, and T_3. By Theorem 2.3,*

$$P(T) = \sum_{i=1}^{3} P(T_i) = 1 + 2 + 1 = 4 \quad \textit{and} \quad N(T) = \prod_{i=1}^{3} N(T_i) = 1 \cdot \binom{3}{2} \cdot 1 = 3.$$



Our goal now is to identify all absent edges in a given tree.

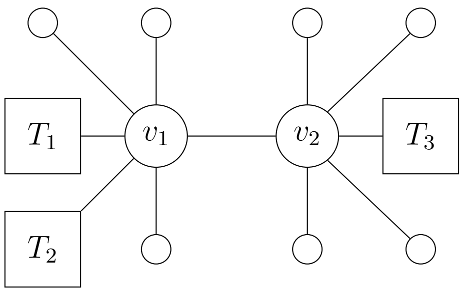
4.1 Identifying Absent Edges

We first use $H(T)$ and incremental degrees to directly identify some absent edges for a tree T.

Proposition 4.5. *In a tree T, an edge between two HID vertices is an absent edge.*

Proof. Suppose that in a tree T with a connected Hi-graph, there is one or more pairs of adjacent HID vertices. We complete the proof by contradiction.

Assume that in T, the edge between two adjacent HID vertices, v_1 and v_2, is used in an MPC $\mathcal{C}$. We use (v_1, v_2) to denote the edge and l to denote the path in $\mathcal{C}$ that contains v_1, v_2, and (v_1, v_2). Suppose that $\delta(v_1) = d_1$ and $\delta(v_2) = d_2$, with $d_1, d_2 \geq 2$, and that $P(T) = k$. Because $H(T)$ is connected, v_1 and v_2 have d_1 and d_2 pendent paths in T.



The tree shown above demonstrates a sample structure of T. The nodes denoted as T_i's represent the HDV's (besides v_1 and v_2) adjacent to either v_1 or v_2 and the subtrees connected to them. In order to minimize the number of paths used, l must include one neighboring edge at v_1 and one at v_2 besides (v_1, v_2). These two edges at v_1 and v_2 can either connect them with a pendent path or an adjacent HDV depending on the overall structure of T. Either way, there is at least one pendent path left at each vertex that is included in $\mathcal{C}$ separately. A new MPC, $\mathcal{C}'$, can be constructed from $\mathcal{C}$. The process is as follows.

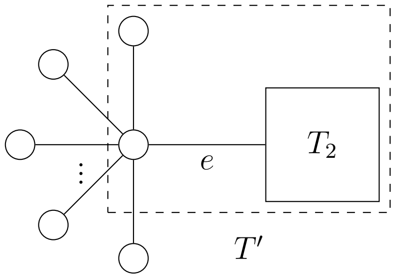
The path, l, first gets split into two by removing (v_1, v_2) from $\mathcal{C}$. Then, we connect the path that contains v_1 to one of its "available" pendent paths and apply the same to v_2. During this process, the total number of paths increases by 1 when (v_1, v_2) is removed and decreases by 2 when v_1 and v_2 are respectively connected to the two pendent paths that were originally separate paths in $\mathcal{C}$. Therefore, $|\mathcal{C}'| = k + 1 - 2 = k - 1 < P(T)$. We have now reached a contradiction. Therefore, (v_1, v_2) is absent in T. □

Not all absent edges are between two HID vertices. In order to identify the rest of them, we reduce a tree to a smaller one without changing the statuses of the edges with the help of **pendent g-stars**. A pendent g-star in a tree T is a g-star induced by a **peripheral HDV** and its pendent paths. An HDV v is peripheral if and only if there is exactly one branch of T at v that contains all the other HDVs in T.

Lemma 4.6. *In a tree T, an edge connecting a pendent g-star to the rest of T is never required.*

Proof. Let e be an edge connecting a pendent g-star of incremental degree d to the rest of T, T_2. Assume that e is a required edge. Then, in every MPC of T, there is always a path that includes e, the central HDV of the pendent g-star, as well as one of its pendent paths in order to minimize the number of paths used. Suppose that in an MPC of T, other than the path that uses e, there are k paths that cover the rest of T_2. Then,
$P(T) = k + 1 + (d - 1) = k + d$.

We now construct a new path cover for T, where we use the same set of paths to cover T_2, except that the path that originally included e now terminates at the vertex in T_2 neighboring e. This means that there are still $(k + 1)$ paths covering T_2. We then need $(d - 1)$ paths to cover the pendent g-star by including two of its pendent paths and the central vertex in the same path and the rest of the pendent paths as separate ones. This new path cover of T uses $(k + 1 + d - 1) = k + d$ paths, which is equal to $P(T)$. We have reached a contradiction, meaning that e must not be a required edge. □



Proposition 4.7. *Removing a pendent g-star from a tree T does not change the statuses of the rest of the edges in T.*

Proof. A pendent g-star is arbitrarily selected to be removed from T along with e, the edge that connects it with the rest of T. The resulting tree is denoted T_2. v denotes the vertex in T' that is the immediate neighbor of e. By Lemma 4.6, e can only be absent or discretionary.

If e is an absent edge, none of the original MPC's uses e. It is obvious that removing it does not change the statuses of edges in T_2.

Now suppose that e is discretionary. We can then partition $\boldsymbol{\mathcal{P}}(T)$, the set of all MPC's of T, into two subsets, one with all the MPC's that use e and the other with MPC's that do not use e. Suppose that the selected pendent g-star has incremental degree k. For the subset that does not include e, removing e does not affect the structures of the MPC's, and thus the edges in T_2 have the same status as in T. In this case, we write $P(T) = P(T_2) + (k-1)$.

For the subset of $\boldsymbol{\mathcal{P}}(T)$ where e is always used, the path that includes e in each of the MPC's also goes through a pendent path of the pendent g-star to minimize the number of paths used. Let T' denote the subtree of T induced by e, T_2, and the pendent path. Then, each MPC in $\boldsymbol{P}(T)$ is consisted of $k-1$ paths as well as an MPC in $\boldsymbol{P}(T')$. Thus, $P(T) = (k-1) + P(T') = P(T_2) + (k-1)$. We then get $P(T') = P(T_2)$.

If there is an MPC in $\boldsymbol{P}(T')$ where e is not used, the aforementioned pendent path must be included as a separate path. We get $P(T') = P(T_2) + 1$ as a result, which is contradictory to the result in the previous paragraph. This suggests that e must be a required edge in T'. Therefore, when the pendent g-star and e is removed from T, the statuses of the edges left remain unchanged. □

The **internal tree**, $I(T)$, of a tree T is the subtree induced by the vertices and edges in T when all of its pendent g-stars as well as the edges connecting them to the rest of T are removed.

Corollary 4.8. *For a tree T and its internal tree $I(T)$, the edges that are included in both trees share the same statuses.*

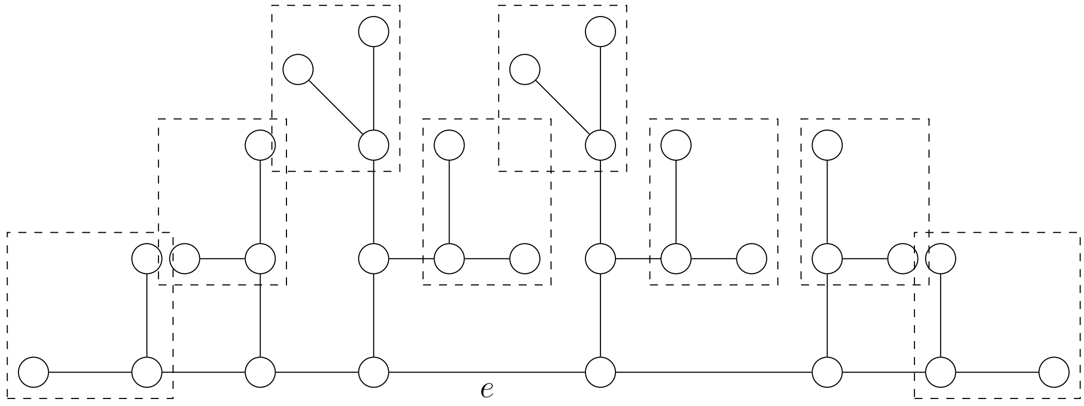
For a tree T where no absent edge can be immediately identified, we can now reduce it to a smaller tree by removing a pendent g-star and check if Proposition 4.5 is applicable. The following algorithm results in identifying all absent edges for a tree T.

Algorithm 4.9. *A tree T is given.*

1. *Let $E_{absent} = \{\}$ denote the set of absent edges in T and set $F = T$.*
2. *Let $T_1, \cdots, T_m$ denote the disjoint components in F. While there exists a connected component in F that has more than one HDV:*
   (a) *For $T_i \in F$, if T_i has only one HDV, we remove T_i from F since it does not include an absent edge. Otherwise, T_i has two or more HDV's. We update F and proceed to the next step.*
   (b) *Let E denote the set of all edges in F. Iterate over all edges in E. If an edge e_i is between two HID vertices in the connected tree T_i that it belongs to, set $F = F - e_i$ and add e_i to E_{absent}.*
   (c) *Apply the following steps to each of the T_i's. Initially, T_i is at the 0th iteration and is represented using $T_i^{(0)}$.*
      i. *During the t-th iteration, every edge in $T_i^{(t)}$ is checked for whether it is absent. In order to determine the status for an edge e_{ij} of $T_i^{(t)}$, remove all pendent g-stars of $T_i^{(t)}$ as well as the edges that connect the pendent g-stars to the rest of $T_i^{(t)}$. However, if e_{ij} is in a pendent g-star itself, that pendent g-star does not get removed.*
      ii. *$T(i)^{(t+1)}$ is obtained after removing all the pendent g-stars of $T_i^{(t)}$. F is updated accordingly when newly identified absent edges are removed. Repeat the entire process in Step 2 on $T(i)^{(t+1)}$ until the initial condition becomes unsatisfied.*
3. *At the completion of the previous steps, all absent edges in T are included in E_{absent}.*

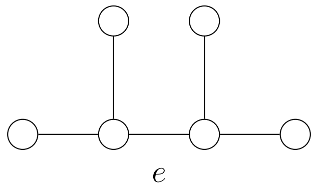
In the worst case scenario, Algorithm 4.9 has complexity $O(n!)$. Since we need to remove all pendent g-stars and iterate over all the edges to identify which of them are absent for each recursive step, each iteration has linear performance. This process is then applied recursively at each of the resulting trees, which gives us a performance of factorial time.

Example 4.10. *Determine whether e is an absent edge for the following tree T.*



We first attempt to identify any edge that is between two HID vertices. Since there is no such edge in T, we proceed to Step 2(c) of Algorithm 4.9.

Since our goal is to identify whether e is absent, all pendent g-stars of T as well as the edges that connect them to the rest of T are removed. The resulting tree is shown below. Since it still includes more than one HDV, we repeat Step 2 of Algorithm 4.9 on the updated tree and find that e is now between two HID vertices. We conclude that e is an absent edge in the original tree T. In fact, e is the only absent edge in T, which can be verified by applying the algorithm thoroughly on T.



5 Prime Trees

In this section, we will present an algorithm to enumerate $N(T)$ for trees that cannot be further decomposed using hyphen decomposition or absent-edge decomposition.

Definition 5.1. *A tree T is* ***prime*** *if $H(T)$ is connected and there are no absent edges in T.*

Lemma 5.2. *For a prime tree T, an edge e connecting a pendent g-star to the rest of T is discretionary.*

Proof. Since T is a prime tree, e must not be absent. By Lemma 4.6, e must not be required. Therefore, e is a discretionary edge. □

Lemma 5.3. *Let T be a prime tree with one or more pendent g-stars. If an edge e connects a pendent g-star of incremental degree k to the rest of T, and if T' is the resulting* *subtree after removing any $k-1$ pendent paths of the pendent g-star from T, then e is a required edge in T'.*

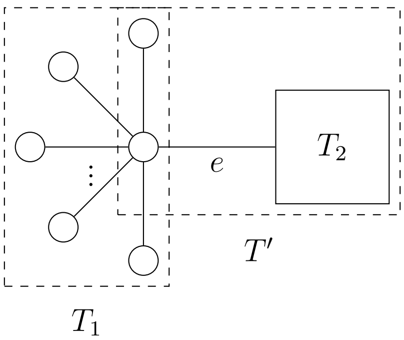
Proof. Included in the proof for Proposition 4.7. $\square$

Algorithm 5.4. *In order to enumerate MPC's for a prime tree T, we first identify an edge e connecting a pendent g-star, T_1, with the rest of T. By Lemma 5.2, e is discretionary. We then consider partitioning $\boldsymbol{\mathcal{P}}(T)$ into two subsets where e is either always used or never used. For the subset $\boldsymbol{\mathcal{P}}_N(T)$, where e is not used, consider the two trees T_1 and T_2 as the result of removing e from T. We have $|\boldsymbol{\mathcal{P}}_N| = N(T_1) \times N(T_2)$.*

Now consider the subset $\boldsymbol{\mathcal{P}}_U(T)$, where e is always used. In this case, in order to minimize the number of paths used, for every MPC in $\boldsymbol{\mathcal{P}}_U(T)$, the path that includes e must also go through the central vertex of T_1 as well as one of its pendent paths. We construct a subtree T' through removing $(k-1)$ pendent paths of the pendent g-star from T. There are $\binom{k}{k-1} = k$ ways of doing so. For each of the k ways, we only count the number of MPC's of T' that use e to be consistent with our setup. By Lemma 3.3, e is required in T', so $N(T')$ is exactly the number of MPC's of T' that use e. The resulting trees, regardless of which $(k-1)$ paths are removed, are all automorphic to one another and thus have the same number of MPC's, $N(T')$. Therefore, $|\boldsymbol{\mathcal{P}}_U| = k \cdot N(T')$.

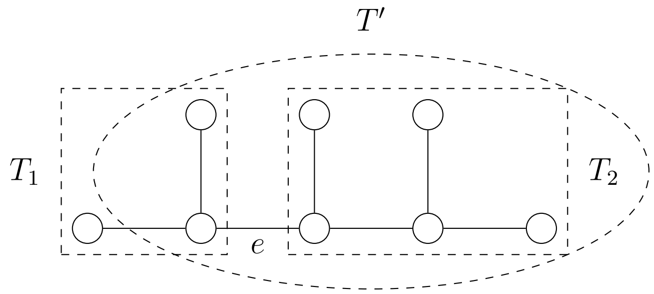
Finally, we have

$$N(T) = |\boldsymbol{\mathcal{P}}(T)| = |\boldsymbol{\mathcal{P}}_N(T)| + |\boldsymbol{\mathcal{P}}_U(T)| = N(T_1) \cdot N(T_2) + k \cdot N(T').$$



Since T_1, T_2, and T' are all on fewer vertices than T, the values $N(T_1), N(T_2)$, and $N(T')$ can also be known inductively with the same algorithm.

Example 5.5. *Calculate $N(T)$ for the following prime tree T.*



$|\boldsymbol{\mathcal{P}}_N(T)| = N(T_1)\cdot N(T_2) = 1\cdot 3 = 3$; $|\boldsymbol{\mathcal{P}}_U(T)| = k\cdot N(T') = 2\cdot 1 = 2$;
$N(T) = |\boldsymbol{\mathcal{P}}(T)| = |\boldsymbol{\mathcal{P}}_N(T)| + |\boldsymbol{\mathcal{P}}_U(T)| = 3 + 2 = 5$.

6 A General Algorithm

Algorithm 6.1. *Given a tree T,*

1. *Apply hyphen decomposition using Algorithm 3.6 to decompose T into components with connected Hi-graphs.*

2. *For each of the resulting components, identify all absent edges and apply absent-edge decomposition.*

3. *For each of the resulting components, repeat Step 1 and 2 since new hyphens and absent edges may arise after the decomposition processes. Keep a record of every decomposition applied and the components involved for Step 5. When all the resulting components become prime, go to Step 4.*

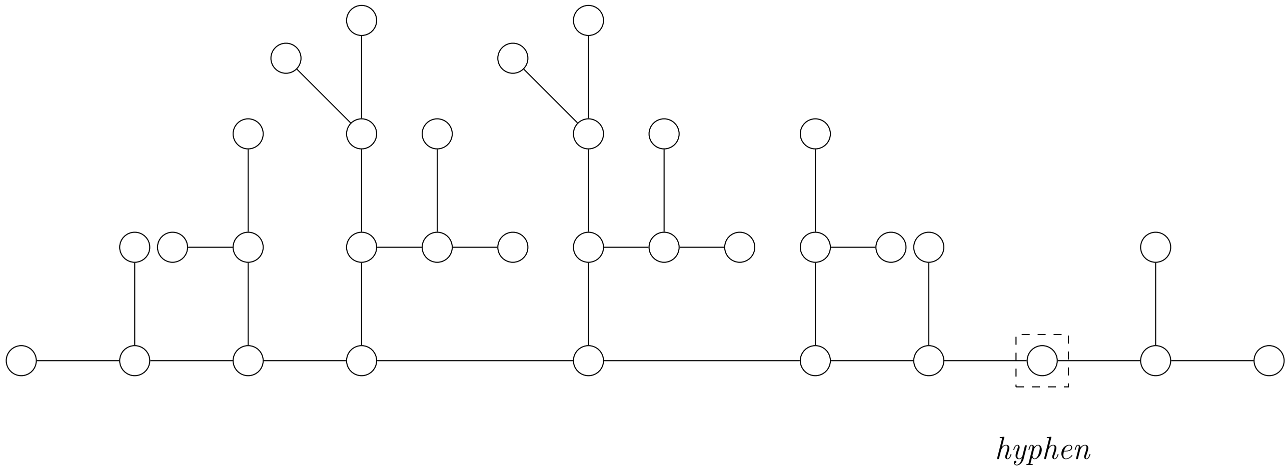
4. *Use Algorithm 5.4 to calculate the number of MPC's for prime trees inductively.*

5. *Use the values obtained from Step 4 and recombine them one step at a time using Theorem 3.7 and Lemma 4.3 to eventually obtain $N(T)$.*

Similar to Algorithm 4.9, this is a recursive process where we recursively reduce the size of a tree and compute the number of MPC's for the resulting components at each recursive step. Therefore, Algorithm 6.1 in general has factorial complexity.

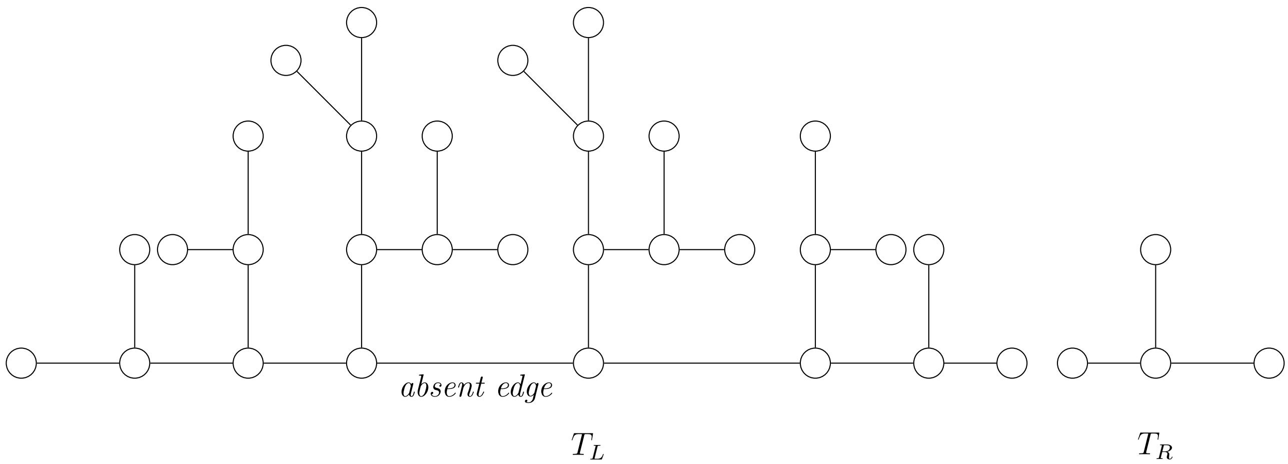
In Appendix A, the complete code that implements Algorithm 6.1 is provided.

Example 6.2. *Solve $N(T)$ for the following tree T using Algorithm 6.1.*



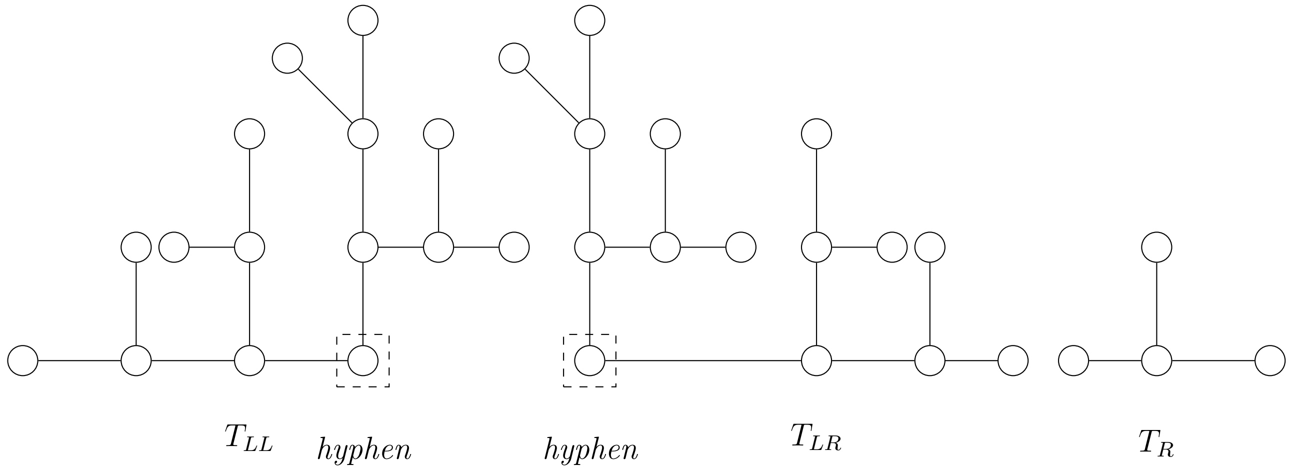
1. Apply hyphen decomposition on T and obtain T_L and T_R. Since T_R is a g-star, we use Proposition 4.2 and get $N(T_R) = \binom{3}{2} = 3$. By Theorem 3.7,

$$N(T) = N(T_L)N(T_R) = 3N(T_L).$$

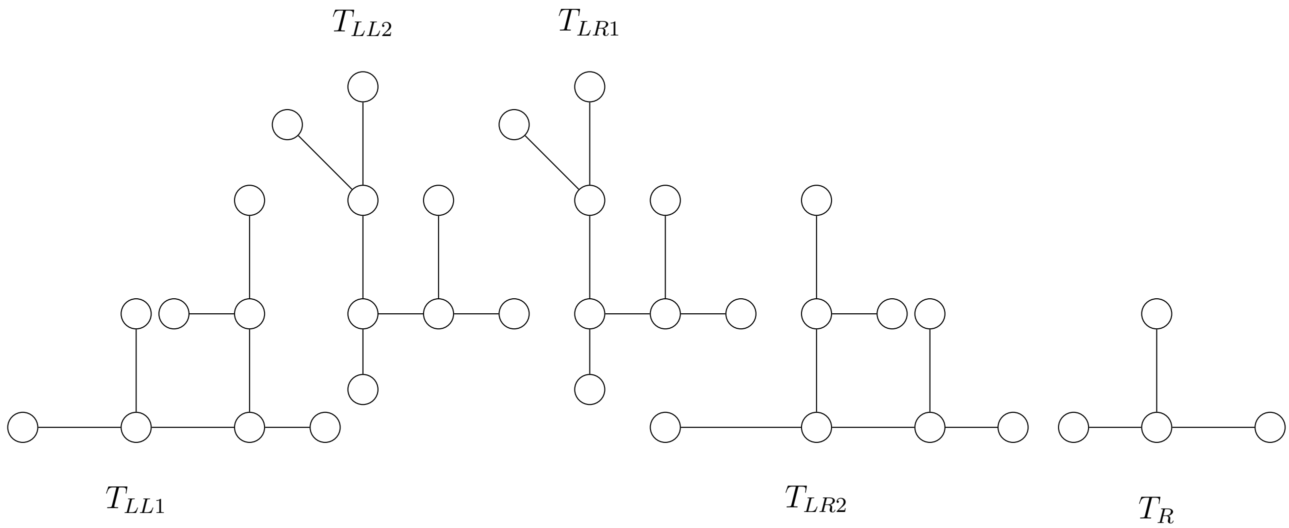


2. Apply absent-edge decomposition on T_L and obtain T_{LL} and T_{LR}. By Lemma 4.3, $N(T_L) = N(T_{LL})N(T_{LR})$. We now have

$$N(T) = 3N(T_{LL})N(T_{LR}).$$



3. Repeat steps 1 and 2 in Algorithm 6.1 for each of the resulting components. Two hyphens are identified, for which hyphen decomposition is applied. No new absent edge is found.



4. Hyphen-decomposing T_{LL} and T_{LR} results in four components, T_{LL1}, T_{LL2}, T_{LR1}, and T_{LR2}. The function for $N(T)$ becomes

$$N(T) = 3N(T_{LL})N(T_{LR}) = 3N(T_{LL1})N(T_{LL2})N(T_{LR1})N(T_{LR2}).$$

Notice that T_{LL1}, T_{LL2}, T_{LR1}, and T_{LR2} are all isomorphic to the tree in Example 5.5. All the components are now prime trees. Thus,

$$N(T_{LL1}) = N(T_{LL2}) = N(T_{LR1}) = N(T_{LR2}) = 5.$$

Finally, we have

$$N(T) = 3N(T_{LL1})N(T_{LL2})N(T_{LR1})N(T_{LR2}) = 3 \times 5^4 = 1875.$$

Appendix A Code to Enumerate $N(T)$

This appendix contains code in Python that calculates N(T) for trees. Here, trees are represented using dictionaries. For a tree T, each of its vertices is labeled and serves as a key in the dictionary for T. The values of the key that represents a vertex v are the neighboring vertices of v. $N(T)$ for a tree T can be calculated by calling the function count_mpc(graph) and providing it with a Graph object (T) as the parameter.

To run the algorithm, copy all the code included in this appendix and save it as a [filename].py file. Then, type "python [filename].py" in the terminal of a computer and press enter as long as Python is installed. The code has been tested to correctly calculate $N(T)$ for 12 trees of different sizes and structures. One caveat is that it returns $N(T) = 1$ for an empty tree. Two testing trees, the 10-vertex nonlinear tree in Example 2.1 and the tree in Example 6.2, are currently included in the code. More instructions are provided later in this appendix on how to create new testing trees and calculate the corresponding $N(T)$.

```
import copy
import operator as op
from functools import reduce


''' Code partially adopted from https://python-course.eu/applications-
                                     python/graphs-python.php '''

class Graph(object):

    def __init__(self, graph_dict=None):
        """ initializes a graph object;
            If no dictionary is given,
            an empty dictionary will be used """

        if graph_dict == None:
            graph_dict = {}
        self._graph_dict = graph_dict

        # keeps a record of the next integer name to assign when creating
                                                 new vertices
        max_v_int = -1
        for v in self._graph_dict.keys():
            if v.isdigit():
```

```
                num = int(v)
                if num > max_v_int:
                    max_v_int = num
        self._next_new_v = max_v_int + 1


        # get all HDV's of the tree
        self._HDVs = []
        for vertex in graph_dict:
            if self.is_HDV(vertex):
                self._HDVs.append(vertex)

        # get all pendent HDV's of the tree
        p_HDV = self._all_pendent_HDVs()
        self._pHDVs = p_HDV.keys()
        self._pHDV_edges = p_HDV.values()

    def get_dict(self):
        """ returns the dictionary representation of the graph """

        return self._graph_dict

    def vertex_degree(self, vertex):
        """ returns the degree of the given vertex """

        degree =  len(self._graph_dict[vertex])
        return degree

    def vertex_neighbors(self, vertex):
        """ returns all the neighboring vertices of a vertice """

        return self._graph_dict[vertex]

    def is_HDV(self, vertex):
        """ identifies whether a vertex is of high degree """

        if self.vertex_degree(vertex) >= 3:
            return True
        return False

    def get_HDVs(self):
        """ returns all HDV's of the graph """

        return self._HDVs
```

```python
def incremental_deg(self, vertex):
    """ returns the incremental degree of a vertex """

    if self.is_HDV(vertex):

        higraph_deg = 0
        for neighbor in self._graph_dict[vertex]:
            if self.is_HDV(neighbor):
                higraph_deg += 1

        return self.vertex_degree(vertex) - higraph_deg

    else:
        return -1

def is_HID(self, vertex):
    """ checks if a vertex is HID """

    if self.incremental_deg(vertex) >= 2:
        return True

    return False

def _all_pendent_HDVs(self):
    """ returns all pendent HDV's of the graph """

    p_hdvs = dict()
    for hdv in self._HDVs:
        hdv_num = [] # count HDV neighbors
        for neighbor in self.vertex_neighbors(hdv):
            if self.vertex_degree(neighbor) != 1:
                hdv_num.append(sorted([hdv, neighbor]))

        if len(hdv_num) <= 1:
            p_hdvs[hdv] = hdv_num

    return p_hdvs

def get_pendent_HDVS(self):
    return self._pHDVs

def remove_pendent_gstar(self, p_hdv):
    """ returns a new graph_dict where the pendent g-star
    with p_hdv as its center is removed """
```

```
        new_graph_dict = copy.deepcopy(self._graph_dict)


        if p_hdv in self._pHDVs:
            for neighbor in self.vertex_neighbors(p_hdv):
                if self.vertex_degree(neighbor) == 1:
                    new_graph_dict.pop(neighbor)
                else:
                    new_graph_dict[neighbor].remove(p_hdv)
            new_graph_dict.pop(p_hdv)
        else:
            print("The given vertex " + str(p_hdv) + " is not a pendent
                                                        HDV.")

        return new_graph_dict

    def remove_v_and_neighbors(self, vertex):
        """ removes the given vertex and all its neighbors from the graph
                                                   """

        for neighbor in self.vertex_neighbors(vertex):

            try:
                self._graph_dict.pop(neighbor)
            except KeyError:
                continue

            try:
                self._graph_dict.pop(vertex)
            except KeyError:
                continue

    def remove_all_pendent_gstars(self, excl_phdv = None):
        """ removes all pendent g-stars from the graph """

        new_graph_dict = copy.deepcopy(self._graph_dict)

        for p_hdv in self._pHDVs:
            if p_hdv != excl_phdv:
                for neighbor in self.vertex_neighbors(p_hdv):

                    if self.vertex_degree(neighbor) == 1:
                        new_graph_dict.pop(neighbor)
                    else:
                        new_graph_dict[neighbor].remove(p_hdv)
```

```
            new_graph_dict.pop(p_hdv)

        return new_graph_dict


    def edges(self, vertice):
        """ returns a list of all the edges of a vertice"""
        return self._graph_dict[vertice]

    def all_vertices(self):
        """ returns the vertices of a graph as a set """
        return set(self._graph_dict.keys())

    def all_edges(self):
        """ returns the edges of a graph """
        return self.__generate_edges()

    def add_pendent_vertex(self, v):
        """ Add a pendent vertex at the given v is not in
            self._graph_dict
        """
        new_v = str(self._next_new_v)
        self._graph_dict[v].add(new_v)
        self._graph_dict[new_v] = [v]
        self._next_new_v += 1


    def add_edge(self, edge):
        """ assumes that edge is of type set, tuple or list;
            between two vertices can be multiple edges!
        """

        edge = set(edge)
        vertex1, vertex2 = tuple(edge)
        for x, y in [(vertex1, vertex2), (vertex2, vertex1)]:
            if x in self._graph_dict:
                self._graph_dict[x].add(y)
            else:
                self._graph_dict[x] = [y]

    def remove_edge(self, edge):

        self._graph_dict[edge[0]].remove(edge[1])
        self._graph_dict[edge[1]].remove(edge[0])
```

```python
    def remove_vertex(self, v):
        """ remove the given vertex"""

        for n in self.vertex_neighbors(v):
            self._graph_dict[n].remove(v)
        self._graph_dict.pop(v, None)

    def __generate_edges(self):
        """ A static method generating the edges of the
            graph "graph". Edges are represented as sets
            with one (a loop back to the vertex) or two
            vertices
        """
        edges = []
        for vertex in self._graph_dict:
            for neighbour in self._graph_dict[vertex]:
                edge = sorted([neighbour, vertex])
                if edge not in edges:
                    edges.append(edge)
        return edges

    def __iter__(self):
        self._iter_obj = iter(self._graph_dict)
        return self._iter_obj

    def __next__(self):
        """ allows us to iterate over the vertices """
        return next(self._iter_obj)

    def __str__(self):
        res = "vertices: "
        for k in self._graph_dict:
            res += str(k) + " "
        res += "\nedges: "
        for edge in self.__generate_edges():
            res += str(edge) + " "
        return res

def ncr(n, r):
    # calculate the result of n choose r
    # code from https://stackoverflow.com/questions/4941753/is-there-a-
                                                  math-ncr-function-in-python
```

```
    r = min(r, n-r)
    numer = reduce(op.mul, range(n, n-r, -1), 1)
    denom = reduce(op.mul, range(1, r+1), 1)
    return numer // denom


def remove_pendent_gstar(graph, p_hdv):

    new_graph = Graph(graph.remove_pendent_gstar(p_hdv))
    return new_graph

def remove_all_pendent_gstars(graph, excl_phdv = None):
    new_graph = Graph(graph.remove_all_pendent_gstars(excl_phdv))
    return new_graph

def get_all_hyphens(graph):

    hyphen_num = 0

    for v in graph.all_vertices():
        # search for degree-two vertices
        if graph.vertex_degree(v) == 2:
            num_neighbor_HDV = 0  # number of neighboring HDV's

            n = list(graph.vertex_neighbors(v))

            if len(n) == 2:

                if graph.is_HDV(n[0]) and graph.is_HDV(n[1]):

                    graph.remove_vertex(v)
                    graph.add_pendent_vertex(n[0])
                    graph.add_pendent_vertex(n[1])
                    hyphen_num += 1

                elif graph.is_HDV(n[0]):
                    graph.remove_edge(sorted([n[1], v]))
                    hyphen_num += 1

                elif graph.is_HDV(n[1]):
                    graph.remove_edge(sorted([n[0], v]))
                    hyphen_num += 1

            else:
                print("Vertex " + v + " is not of low degree and thus is
```

```
                                                    not part of a hyphen.
                                                    ")

    return graph, hyphen_num

def get_all_absent_edges(graph):
    """ recursively return all absent edges of the given graph"""

    absent_edges = []

    # first get all edges between two HID vertices
    for edge in graph.all_edges():
        if graph.is_HID(edge[0]) and graph.is_HID(edge[1]):
            absent_edges.append(edge)
            graph.remove_edge(edge)

    if len(graph.get_pendent_HDVS()) > 1:

        without_neighbor_HID = Graph(graph.get_dict())

        # identify other absent edges by recursively removing pendent g-
                                            stars
        smaller_g = remove_all_pendent_gstars(without_neighbor_HID)
        sub_absent = get_all_absent_edges(smaller_g)
        absent_edges = sub_absent + absent_edges

        for v in graph.get_pendent_HDVS():
            without_neighbor_HID = Graph(graph.get_dict())

            # identify other absent edges by recursively removing pendent
                                            g-stars
            smaller_g = remove_all_pendent_gstars(without_neighbor_HID,
                                            excl_phdv=v)
            sub_absent = get_all_absent_edges(smaller_g)

            absent_edges = sub_absent + absent_edges

    return absent_edges


def count_mpc(graph):
    """ count the number of MPC's for any given tree """

    hdvs = graph.get_HDVs()
    if 1 == 1:
```

```
        graph, hyphen_num = get_all_hyphens(graph)
        absent_e = get_all_absent_edges(graph)

        for e in absent_e:
            try:
                graph.remove_edge(e)
            except KeyError:
                continue

        while len(absent_e) != 0 or hyphen_num != 0:

            graph, hyphen_num = get_all_hyphens(graph)
            absent_e = get_all_absent_edges(graph)
            for e in absent_e:
                graph.remove_edge(e)

    graph = Graph(graph.get_dict())

    # create a deep copy of the graph passed to this function
    new_graph = Graph(copy.deepcopy(graph.get_dict()))
    cur_N_T = 1

    for v in graph.get_HDVs():

        # paths do not matter
        deg_one_num = deg_one_n(graph, v)[0]

        # if the vertex is the center of a star
        # directly calculate its N(T) and multiply it to cur_NT
        # then remove the star from the forest
        if len(deg_one_num) == graph.vertex_degree(v):

            cur_N_T = cur_N_T * ncr(len(deg_one_num), 2)
            new_graph.remove_v_and_neighbors(v)

    new_graph = Graph(new_graph.get_dict())

    if len(new_graph.get_pendent_HDVS()) >= 1:
        for v in list(new_graph.get_pendent_HDVS())[0]:

            new_T1 = remove_pendent_gstar(new_graph, v)
            for_T2 = Graph(copy.deepcopy(new_T1.get_dict()))

            g1, N_T1 = count_mpc(new_T1)
            deg_one_num, hdv_num = deg_one_n(new_graph, v)
```

```
            if len(hdv_num) == 1:
                for_T2.add_pendent_vertex(hdv_num[0])

            g2, N_T2 = count_mpc(for_T2)
            N_T_pstar = ncr(len(deg_one_num), 2)
            net_N_T = N_T_pstar * N_T1 + N_T2 * len(deg_one_num)

        cur_N_T = cur_N_T*net_N_T

    return new_graph, cur_N_T

""" return all neighbors of degree one of the given vertex in the given
                                                    graph """
def deg_one_n(graph, v):
    deg_one_neighbor = []
    hdv_neighbor = []

    for n in graph.vertex_neighbors(v):
        if graph.vertex_degree(n) == 1:
            deg_one_neighbor.append(n)
        else:
            hdv_neighbor.append(n)
    return deg_one_neighbor, hdv_neighbor
```

The segment shown below includes the two dictionaries that respectively represent the 10-vertex nonlinear tree in Example 2.1 and the tree in Example 6.2. New trees can be added to the code using a similar format. A current limitation of the code is that it requires the label of a vertex to either be a single letter (e.g. 'a') or a stringified nonnegative integer (e.g. '20'). After the dictionary for a new tree, new_dict, is included in the code, use new_tree = Graph(new_dict) to instantiate it as a Graph object. We then call the function that calculates $N(T)$ using the line "N_T = count_mpc(new_tree)[1]". The variable N_T then gives us $N(T)$ for the new tree.

```
if __name__ == "__main__":

    # the 10-vertex nonlinear tree
    g_10  = { "a" : {"b"},
      "b" : {"a", "c", "d"},
      "c" : {"b"},
      "d" : {"b", "h", "e"},
      "e" : {"d", "f", "g"},
      "f" : {"e"},
```

```
    "g" : {"e"},
    "h" : {"d", "i", "j"},
    "i" : {"h"},
    "j" : {"h"}
}

# the tree given in Example 6.2 in the thesis
thesis_ex = { "a" : {"b"},
    "b" : {"a", "c", "d"},
    "c" : {"b"},
    "d" : {"b", "h", "e"},
    "e" : {"d", "f", "g"},
    "f" : {"e"},
    "g" : {"e"},
    "h" : {"d", "i", "p"},
    "i" : {"h", "j", "m"},
    "j" : {"i", "l", "k"},
    "k" : {"j"},
    "l" : {"j"},
    "m" : {"i", "n", "o"},
    "n" : {"m"},
    "o" : {"m"},
    "3" : {"2"},
    "2" : {"x", "3", "4"},
    "4" : {"2", "5"},
    "x" : {"p", "y", "2"},
    "5" : {"4", "6", "7"},
    "6" : {"5"},
    "7" : {"5", "8"},
    "8" : {"7"},
    "y" : {"x", "z", "1"},
    "z" : {"y"},
    "1" : {"y"},
    "p" : {"x", "q", "h"},
    "q" : {"p", "r", "u"},
    "r" : {"q", "s", "t"},
    "s" : {"r"},
    "t" : {"r"},
    "u" : {"q", "v", "w"},
    "v" : {"u"},
    "w" : {"u"},
}

# instantiate the two trees as Graph objects
graph_10 = Graph(g_10)
```

```
thesis_tree = Graph(thesis_ex)

# test the complete algorithm count_mpc()
t1, n1 = count_mpc(graph_10)
print("The number of MPC's of the 10-vertex nonlinear tree in Example
                                     1.5 is: ", n1)
t2, n2 = count_mpc(thesis_tree)
print("The number of MPC's of the tree in Example 6.2 is: ", n2)
```

Below is the output of the code:

The number of MPC's of the 10-vertex nonlinear tree in Example 1.5 is: 19

The number of MPC's of the tree in Example 6.2 is: 1875

Bibliography

[1] C. R. Johnson, A. Leal-Duarte, and C. M. Saiago, "The Parter-Wiener theorem: Refinement and generalization," *SIAM Journal on Matrix Analysis and Applications*, vol. 25(2), pp. 352–361, 2003.

[2] C. R. Johnson and C. M. Saiago, *Eigenvalues, Multiplicities, and Graphs*. Cambridge University Press, 2018.

[3] C. R. Johnson and A. Leal-Duarte, "The maximum multiplicity of an eigenvalue in a matrix whose graph is a tree," *Linear and Multilinear Algebra*, vol. 46, pp. 139–144, 1999.

[4] L. Hogben and C. R. Johnson, "Path covers of trees," unpublished note.

[5] C. R. Johnson and C. M. Saiago, "Estimation of the maximum multiplicity of an eigenvalue in terms of the vertex degrees of the graph of a matrix," *Electronic Journal of Linear Algebra*, vol. 9, pp. 27–31, 2002.

[6] C. R. Johnson, A. Leal-Duarte, and C. Saiago, "Inverse eigenvalue problems and lists of multiplicities of eigenvalues for matrices whose graph is a tree: The case of generalized stars and double generalized stars," *Linear Algebra and its Applications*, vol. 373, pp. 311–330, 2003.

[7] C. R. Johnson, A. Leal-Duarte, C. M. Saiago, B. D. Sutton, and A. J. Witt, "On the relative position of multiple eigenvalues in the spectrum of an hermitian matrix with a given graph," *Linear Algebra and its Applications*, vol. 363, pp. 147–159, 2003.

[8] C. R. Johnson and C. M. Saiago, "The trees for which maximum multiplicity implies the simplicity of other eigenvalues," *Discrete Mathematics*, vol. 306(23), pp. 3130–3135, 2006.

[9] C. R. Johnson, A. Leal-Duarte, and C. M. Saiago, "The structure of matrices with a maximum multiplicity eigenvalue," *Linear Algebra and its Applications*, vol. 429(4), pp. 875–886, 2008.

[10] C. R. Johnson, A. A. Li, and A. J. Walker, "Ordered multiplicity lists for eigenvalues of symmetric matrices whose graph is a linear tree," *Discrete Mathematics*, vol. 333, pp. 39–55, 2014.

[11] I.-J. Kim and B. L. Shader, "Smith normal form and acyclic matrices," *Journal of Algebraic Combinatorics*, vol. 29(1), pp. 63–80, 2009.