Application of Matroid Intersection for Cost-Effective Load Balancing

Benedict Presley – 13523067
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
presleybenedict04@gmail.com, 13523067@std.stei.itb.ac.id

***Abstract*— This paper addresses the degree-constrained minimum spanning tree problem, a combinatorial optimization challenge that arises in network design and resource allocation. By leveraging the framework of matroid theory, specifically the weighted matroid intersection problem, this paper presents a solution that guarantees an optimal spanning tree while satisfying degree constraints on specific vertices to ensure balanced load distribution across the network.**

***Keywords*— degree, matroid theory, minimum spanning tree, weighted matroid intersection**

I. Introduction

Network reliability is a study that focus on designing efficient networks and ensuring their functionality despite potential failures, resource constraints, or dynamic changes. It examines how to design, maintain, and optimize networks to achieve robust performance, often balancing trade-offs between cost, efficiency, and redundancy. Network reliability is a critical aspect of modern infrastructure, used in many fields such as telecommunications, distributed systems, and cloud computing. A key aspect of these studies is constructing networks that minimize the risk of overload and maximize operational efficiency under given constraints.

One problem in this domain involves constructing minimum-weight spanning trees that satisfy specific degree constraints on certain nodes, ensuring cost-effective load balancing across the network. Traditional algorithms for minimum spanning trees, such as Prim's or Kruskal's, focus solely on minimizing cost without accounting for degree constraints, limiting their applicability in real-world scenarios where node capacities may be bounded. Furthermore, there are few algorithms capable of solving this problem both optimally and efficiently, making it a challenging and open area of research in combinatorial optimization.

This paper explores the application of matroids for solving constrained optimization problems. A matroid can be thought of as a structure that captures the dependencies among elements in a set, providing a systematic way to identify optimal subsets under given conditions. Matroid intersection extends this idea, enabling the simultaneous consideration of two matroids to find a common independent set that satisfies both structures. This capability is particularly relevant for network design problems where multiple constraints, such as cost and degree limits, must be addressed simultaneously. By leveraging matroid intersection, this paper introduces a novel approach for constructing degree-constrained minimum spanning trees, enabling cost-effective load balancing across the network.

II. Theoretical Basis

A. *Set Theory*

1. Definition and Notation

   A set is a collection of distinct objects, which can be finite or infinite. Objects in set are called elements.

   A set is typically denoted by a pair of curly braces containing its elements. For example: $S = \{a, b, c\}$ represents a set with three elements, a, b, and c. An empty set contains no elements and is denoted by $\emptyset$. If an element a is part of a set S, it is denoted by $a \in S$.

   The size of a set S is the number of elements contained in S, denoted by $|S|$.

2. Subsets and Power Sets

   A subset is a set whose elements all belong to another set. If $A \subseteq B$, then every element of A is also an element of B. The set of all subsets of S is called the power set, denoted as $P(S)$.

   For example:
   $S = \{a, b\}$, then $P(S) = \{\emptyset, \{a\}, \{b\}, \{a, b\}\}$

3. Set Operations

   The following are fundamental operations on sets
   - Union
     The union of two sets A and B, denoted $A \cup B$, is the set of all elements that are in A, B, or both. Formally, $A \cup B = \{x : x \in A \lor x \in B\}$.
   - Intersection
     The intersection of two sets A and B, denoted $A \cap B$, is the set of all elements that are in both A and B. Formally, $A \cup B = \{x : x \in A \land x \in B\}$.
   - Difference
     The difference between two sets A and B, denoted $A \setminus B$, is the set of elements that are in A but not in B. Formally, $A \setminus B = \{x : x \in A \land x \notin B\}$.
   - Symmetric Difference
     The symmetric difference of two sets A and B, denoted $A \Delta B$, is the set of elements that are in A or B but not both. Formally, $A \Delta B = (A \setminus B) \cup (B \setminus A)$
   - Cartesian Product

The Cartesian product of two sets A and B, denoted $A \times B$, is the set of all ordered pairs (a, b), where $a \in A$ and $b \in B$. Formally $A \times B = \{(a, b): a \in A, b \in B\}$.
For example: $A = \{1,2\}$ and $B = \{x, y\}$, then $A \times B = \{(1, x), (1, y), (2, x), (2, y)\}$.

B. Graph Theory

A graph is a fundamental structure consisting of vertices (or nodes) connected by edges.

1. Definition

A graph G is formally defined as an ordered pair $G = (V, E)$, where V is the set of vertices (or nodes) and E is the set of edges, where each edge connects two vertices.

An edge $e \in E$ can be represented as:

- Undirected edge: $e = \{u, v\}$, indicating a bidirectional connection between u and v.

Figure 1. Undirected edge e = {1,2}

- Directed edge: $e = (u, v)$, indicating a directed connection from u to v.

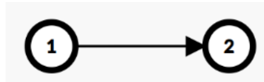


Figure 2. Directed edge e = (1, 2)

2. Properties of Graphs

- Undirected Graph
  Graph whose edges have no direction.
- Directed Graph (Digraph)
  Graph whose edges have a direction.

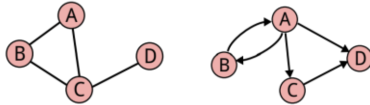


Figure 3. Undirected graph (left) and directed graph (right) [Source: https://osn.toki.id/data/pemrograman-kompetitif-dasar.pdf]

- Unweighted Graph
  Graph whose edges don't have an assigned weight (or cost).
- Weighted Graph
  Graph whose edges have an assigned weight (or cost), such as $w(e)$.

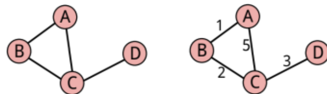


Figure 4. Unweighted graph (left) and weighted graph (right) [Source: https://osn.toki.id/data/pemrograman-kompetitif-dasar.pdf]

Note that a graph can be a combination of the elements above.

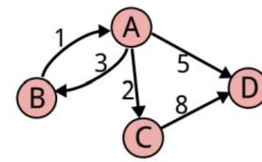


Figure 5. directed weighted graph [Source: https://osn.toki.id/data/pemrograman-kompetitif-dasar.pdf]

3. Features of Graphs

- Degree of a vertex
  The degree of a vertex, denoted as deg (v), is the number of edges incident to it. If the edges are directed, degree can be categorized into in-degree (the number of edges directed towards the vertex) and out-degree (the number of edges directed away from the vertex).
- Path
  A sequence of vertices such that each adjacent pair is connected by an edge. A simple path contains no repeated vertices.
- Cycle
  A path that starts and ends at the same vertex. A graph with no cycle is called acyclic graph.
- Negative Cycle
  A cycle whose total weight of edges is negative.

4. Special Graphs

- Tree
  A tree is a connected, acyclic graph. If T is a tree with n vertices, it has $n - 1$ edges.

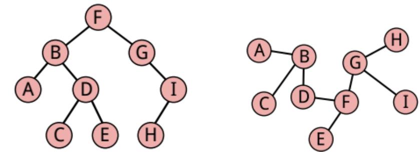


Figure 6. Examples of Trees [Source: https://osn.toki.id/data/pemrograman-kompetitif-dasar.pdf]

- Spanning Tree
  A spanning tree of a graph G is a subgraph that includes all vertices of G and is a tree.

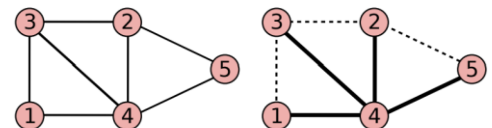


Figure 7. Graph (left) and its spanning tree (right) [Source: https://osn.toki.id/data/pemrograman-kompetitif-dasar.pdf]

- Bipartite Graph
  A graph is bipartite if its vertex set V can be partitioned into two disjoint sets U and W, such that every edge only connects a vertex in U to one in W.

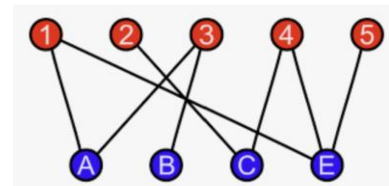


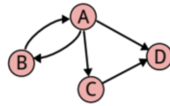
Figure 8. Bipartite graph [Source: https://en.wikipedia.org/wiki/Bipartite_graph]

5. Graph Representation

Graph can be represented in various ways.

- Adjacency Matrix

  A 2D matrix A of size $n \times n$ (where n is the number of vertices) is used. $A_{ij} = 1$ if there is an edge between vertex i and vertex j and $A_{ij} = 0$ otherwise.

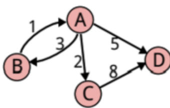


	A	B	C	D
A	0	1	1	1
B	1	0	0	0
C	0	0	0	1
D	0	0	0	0

Figure 9. Unweighted adjacency matrix [Source: https://osn.toki.id/data/pemrograman-kompetitif-dasar.pdf]

- Weighted Adjacency Matrix

  A 2D matrix A of size $n \times n$ (where n is the number of vertices) is used. $A_{ij} = w(e)$ if there is an edge between vertex i and vertex j and $A_{ij} = \infty$ otherwise.

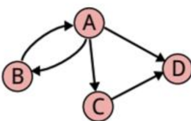


	A	B	C	D
A	∞	3	2	5
B	1	∞	∞	∞
C	∞	∞	∞	8
D	∞	∞	∞	∞

Figure 10. Weighted adjacency matrix [Source: https://osn.toki.id/data/pemrograman-kompetitif-dasar.pdf]

- Adjacency List

  For each vertex, a list of its neighbors is stored.

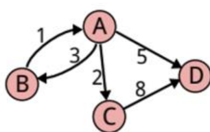


A	$[B, C, D]$
B	$[A]$
C	$[D]$
D	$[\,]$

Figure 11. Unweighted adjacency list [Source: https://osn.toki.id/data/pemrograman-kompetitif-dasar.pdf]

- Weighted Adjacency List

  For each vertex, a list of its neighbors and edge weight is stored

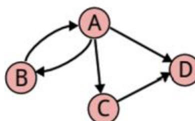


A	$[\langle B,3\rangle, \langle C,2\rangle, \langle D,5\rangle$
B	$[\langle A,1\rangle]$
C	$[\langle D,8\rangle]$
D	$[\,]$

Figure 12. Weighted adjacency list [Source: https://osn.toki.id/data/pemrograman-kompetitif-dasar.pdf]

- Edge List

  A list of edges of the graph.



$\langle A,B\rangle$,
$\langle A,C\rangle$,
$\langle A,D\rangle$,
$\langle B,A\rangle$,
$\langle C,D\rangle$

Figure 13. Edge list [Source: https://osn.toki.id/data/pemrograman-kompetitif-dasar.pdf]

- Weighted Edge List

  A list of edges and the respective weights of the graph.

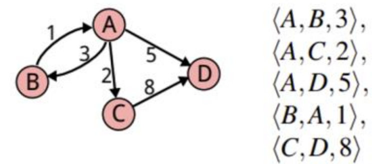


Figure 14. Weighted edge list [Source: https://osn.toki.id/data/pemrograman-kompetitif-dasar.pdf]

C. *Minimum Spanning Tree Problem*

The Minimum Spanning Tree (MST) problem is a fundamental problem in graph theory and combinatorial optimization. It involves finding a subset of edges in a weighted, connected, and undirected graph that:

- Connects all the vertices in the graph.
- Forms a tree
- Minimizes the total weight of the edges in the tree.

Formally,

Given a graph $G = (V, E)$, where V is the set of vertices and E is the set of edges with associated weights $w(e)$ for each edge $e \in E$, the objective is to find a subset $T \subseteq E$ such that:

- T spans all vertices in V (i.e., it connects all vertices).
- T forms a tree.
- The total weight $\sum_{e \in T} w(e)$ is minimized.

This problem can be solved using various algorithms such as Kruskal's algorithm, Prim's algorithm, or Boruvka's algorithm.

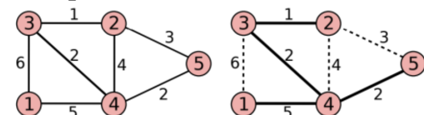


Figure 15. Graph (left) and its MST (right) [Source: https://osn.toki.id/data/pemrograman-kompetitif-dasar.pdf]

D. *Shortest Path Problem and Bellman-Ford Algorithm*

The shortest path problem involves finding the path between two vertices in a graph such that the sum of the weights of the edges along the path is minimized. Formally,

Given a graph $G = (V, E)$. Let $s \in V$ be the source vertex, and let $t \in V$ be the target vertex. The goal is to find a path $P = (s, v_1, v_2, \ldots, t)$ where:

- $P \subseteq E$ (all edges in the path belong to E)
- $\sum_{(u,v) \in P} w(u, v)$ is minimized.

For the single-source shortest path problem, the objective is to compute the shortest path $P_s(v)$ for every vertex $v \in V$, such that $d(s, v) = \min \sum_{(u,v) \in P} w(u, v)$, where $d(s, v)$ is the shortest distance from s to v.

There are various algorithm that can be utilized to solve shortest path problem, one such algorithm is the Bellman-Ford algorithm.

The Bellman-Ford algorithm works by iteratively relaxing all edges in a graph to compute the shortest paths from a single source vertex to all other vertices. It initializes the distance to the source as 0 and all other

vertices as infinity. Then, for $|V|-1$ iterations, it checks each edge (u, v) and updates the distance to vertex v if a shorter path through u is found.

```
1: function BELLMAN-FORD(s)
2:     dist ← new integer[V + 1]

3:     FILLARRAY(dist, ∞)
4:     dist[s] ← 0
5:     for i ← 1, V − 1 do
6:         for ⟨u, v⟩ ∈ edgeList do
7:             if dist[v] > dist[u] + w[u][v] then
8:                 dist[v] = dist[u] + w[u][v]
9:             end if
10:        end for
11:    end for

12:    return dist
13: end function
```

Figure 16. Pseudocode of Bellman-Ford algorithm [Source: https://osn.toki.id/data/pemrograman-kompetitif-dasar.pdf]

E. Disjoint Set Union (DSU)

Disjoint Set Union (DSU), also known as the Union-Find data structure, is a data structure for managing a collection of disjoint sets. It supports two key operations efficiently:

1. Find: Determine the representative (or leader) of the set containing a particular element.
2. Union: Merge two sets into one, ensuring that the resulting sets remain disjoint.

DSU can be used to maintain and check the connectivity of nodes in graph. DSU can also be used to detect cycles in a graph.

F. Matroid Theory

1. Definition

A matroid M is formally defined as an ordered pair (S, I) where S is a finite set called the ground set and I is a family of subsets of S, called the independent sets, that satisfy the following axioms:

a. Non-empty property: $\emptyset \in I$
b. Hereditary property: If $A \in I$ and $B \subseteq A$, then $B \in I$.
c. Exchange property: If $A, B \in I$ and $|A| > |B|$, then there exists $x \in A \backslash B$ such that $B \cup \{x\} \in I$.

These rules represent the idea of independence, similar to how some sets of vectors in linear algebra are independent or how some groups of edges in a graph form structures without cycles.

Example:
$M = (S, I)$ where $S = \{x, y, z\}$ and $I = \{\emptyset, \{x\}, \{y\}, \{z\}, \{x, y\}, \{x, z\}\}$.

2. Basis, Circuits, and Rank

- Basis

A basis of a matroid is a maximal independent set. All bases of a matroid have the same size, otherwise we can add something to the smaller basis from a greater basis by the exchange property. Any independent set is a subset of some basis (by hereditary property and exchange property), thus knowing all the bases of a matroid describes the whole matroid. We can construct a basis of a matroid using Rado-Edmonds algorithm.

- Circuit

A circuit is a minimal dependent set, a dependent set where removing any element makes it independent.

- Rank function

The rank of a matroid is the size of its bases. The rank of a matroid can also be defined more flexibly. Formally, $r(A)$ is the rank of a set A in matroid $M = (S, I)$.

The rank function satisfies

- $r(A) \leq |A|$ for all $A \subseteq S$.
- $r(A) \leq r(B)$ if $A \subseteq B \subseteq S$.
- $r(A \cup B) + r(A \cap B) \leq r(A) + r(B)$.

3. Types of Matroid

Matroids are abstract structures that can represent a wide variety of problems. Matroids can be created or tailored to model specific problems by defining the ground set and the independence rule. Below are some common types of matroid.

- Graphic Matroid

A graphic matroid is derived from a graph $G = (V, E)$, where the ground set S is the set of edges E of the graph and the independent sets I are the subsets of S that do not form a cycle.

- Colorful Matroid

A colorful matroid arises when elements are grouped into distinct categories or colors. The ground set consists of colored elements. Each element has exactly one color. Set of elements is independent if no pair of included elements share a color.

- Uniform matroid

The ground set S of a uniform matroid is any finite set. The independent sets I are all subsets of S with size at most k, where k is a fixed integer.

4. Matroid Intersection

The matroid intersection problem seeks a maximum common independent set between two matroids $M_1 = (S, I_1)$ and $M_2 = (S, I_2)$ defined on the same ground set S. Formally, the problem can be stated as finding an independent set I that satisfies

$$|I| = \max\{|I'|: I' \in I_1 \cap I_2\}$$

Matroid intersection is useful because it allows us to find a structure that satisfies two different constraints simultaneously. By identifying the largest set of elements that are independent in both matroids, it provides a formal framework for solving problems where multiple requirements must be met together.

The weighted matroid intersection problem extends the matroid intersection problem by assigning weights to the elements of the ground set. The goal is to find a common independent set between two matroids that maximizes the total weight of its elements. Formally, the problem can be stated as finding an independent set I that satisfies

$$\sum_{e \in I} w(e) = \max\left\{\sum_{e \in I'} w(e) : I' \in I_1 \cap I_2\right\}$$

where $w(e)$ is the weight of edge e.

In order to solve matroid intersection problem, there are two results that play a central role in the solution of matroid intersection problem.

First, the min-max relation theorem establishes that the maximum size of a common independent set $I \in I_1 \cap I_2$ between two matroids $M_1 = (S, I_1)$ and $M_2 = (S, I_2)$ is given by

$$\max|I| = \min_{A \subseteq S}(r_1(A) + r_2(S \setminus A))$$

where r_1 and r_2 are the rank functions of the matroids. This equality provides a theoretical limit for the size of the independent set and confirms the optimality of the solution. It guarantees that the optimal solution is reached by ensuring tha the theoretical upper bound is achieved.

The Edmonds-Lawler theorem extends this concept to the weighted case, where each element in the ground set S is assigned a weight, and the goal is to find a maximum-weight common independent set. This is achieved using an augmenting path algorithm on an "exchange graph", where vertices represent elements in S, and edges indicate opportunities to exchange elements between the current independent set and the rest of the ground set. The algorithm iteratively finds augmenting paths to increase the total weight until no further improvements can be made. The min-max relation ensures that this process converges to the theoretical maximum.

III. Problem Definition and Solution

A. Problem Definition

The problem tackled in this paper involves finding a cost-effective spanning tree in a graph while satisfying specific degree constraints on certain vertices. This type of problem arises in network design, where constraints like capacity limits or load balancing must be incorporated into the optimization process. Formally the problem statement is given below.

Given an undirected, connected, and weighted graph $G = (V, E)$ where:

1. V is the set of n vertices,
2. E is the set of edges,
3. $w(e)$ is the weight of each edge $e \in E$.

Construct a spanning tree $T \subseteq E$ that satisfies the following conditions:

1. T is a spanning tree, meaning it connects all vertices in V without forming cycles.
2. For a subset of vertices $V_k \subseteq V$ (where $|V_k| = k$), each vertex $v \in V_k$ has a degree constraint $d(v)$, specifying that the degree of v in the spanning tree T must not exceed $d(v)$.
3. The total weight of the edges in T is minimized.

B. Solution

In order to solve the degree-constrained minimum spanning tree problem, we adopt a matroid intersection solution that combines a graphic matroid (which ensures acyclicity) and a degree matroid (which enforces degree constraints on the special vertices). The first part of this solution is that we first fix a small "forest" among the special vertices (since the number of possible forests on a small set of special vertices is limited), and then we attempt to complete the MST by adding other edges in a way that respects both acyclicity and degree bounds.

We enumerate all such candidate forests T. Each forest T represents a pre-selected set of edges between special vertices. Once T is fixed, only the remaining edges—those with zero or one endpoint among the special vertices—are considered for inclusion in the final MST. During this enumeration, if a forest is already invalid (e.g., it contains cycles among the special vertices or violates some degree constraint immediately), we discard it. Otherwise, we proceed to the matroid intersection phase.

After choosing a valid forest T among the special vertices, we want to find additional edges to form a minimum spanning tree that satisfies both the acyclicity (graphic matroid) and the degree constraints (degree matroid). In matroid intersection, we maintain an independent set that already satisfy both matroid properties. To improve or "augment" this set, we construct an exchange graph whose vertices represent edges in the ground set, and we add directed edges between these vertices according to specific feasibility rules:

1. From unused edges to used edges if adding the unused edge (and removing some used edge if necessary) still satisfies both the graphic and degree matroid constraints.
2. From used edges to unused edges similarly, when removing a used edge and adding an unused one leads to a feasible solution.

Finding a sequence of swaps (edges in the exchange graph) that improves the total cost is done via an augmenting path procedure. We search for a path in the exchange graph that begins at some "entering" edge (i.e., an unused edge that can potentially be added) and ends at a suitable "exiting" edge. This ensures that we get a larger independent that is closer towards the optimal answer.

Unlike simple BFS-based augmenting path algorithms for unweighted matroid intersection, our problem involves weighted edges, hence the costs must be factored into each swap. To account for these weights, we track potential improvements using a Bellman–Ford style relaxation loop. Each vertex in the exchange graph (representing a ground-set edge) holds a distance value that indicates how much improvement (cost reduction) can be obtained by flipping edges along a path ending at that vertex. If there are multiple ways to obtain a path with optimal value, we choose the path with the least edges travelled since that minimizes the number of unnecessary exchanges. Once no further improvement is possible (i.e., no augmenting path improves the solution), we have reached a local optimum under the chosen forest T.

Having successfully performed matroid intersection with respect to the fixed forest T, we obtain a candidate MST that respects the degree constraints on the special vertices. We repeat this process for every possible forest T among the special vertices while keeping track of the best solution. This final best solution is guaranteed to be a degree-constrained MST that satisfies all of our problem requirements. We can reconstruct the tree by using the edges in the final independent set we maintained throughout the matroid intersection process.

IV. IMPLEMENTATION

The implementation of the solution is written in C++ for efficiency reasons. Due to the extensive length of the code, it cannot be included in full within this paper. However, readers can access the complete implementation and related documentation on the project's GitHub repository at https://github.com/BP04/weighted-matroid-intersection.

A. Disjoint Set Union (DSU) Class

The DSU class is responsible for tracking connected components in the graph and checking for acyclicity. The data structure acts as an "oracle" to efficiently check if the set being build is independent (have no cycle). Each edge addition corresponds to a union operation, and each cycle check is a matter of comparing representatives.

B. prepare() Function

This subprogram is invoked before each augmentation step to ensure consistency:

1. It initializes or clones the DSU state to match the current set of chosen edges. All used edges are "union-ed," ensuring we have the correct connectivity relation between nodes.
2. It resets the degree array so that each vertex's remaining capacity is reflected accurately. For every edge in the MST, the degrees of its endpoints are decremented.
3. With these updates, subsequent checks in augment() can reliably determine whether adding a new edge would break either the cycle-free condition or a degree limit.

C. augment() Function

This procedure is the core of the matroid intersection strategy, seeking augmenting paths in an exchange graph:

1. Preparation: A call to prepare() resets the DSU and degree arrays to reflect which edges are currently in the partial MST.
2. Feasibility Checks: For each edge in the ground set, we mark whether adding it would create a cycle (safe1) and whether adding it is permissible under the remaining degree constraints (safe2).
3. Exchange Graph Construction: Each edge in the ground set is treated as a node in a directed graph; edges between these nodes represent potential "swaps" (one edge leaves the MST, another enters) that maintain both acyclicity and degree constraint safisfaction.
4. Bellman–Ford Routine: Since edges have weights, we apply a Bellman–Ford routine that looks for negative-cost paths in the exchange graph. Finding a negative-cost path corresponds to a cost-improving series of swaps—i.e., an augmenting path.
5. Flipping Edges: If such a path exists, we flip the chosen edges between "used" and "unused" status, updating the MST cost. If no such path remains, augmentation halts for this configuration.

D. calculate() Function

Once we fix a forest among special vertices, we must incorporate additional edges from the ground set to complete a spanning tree of all N vertices. The calculate function manages this integration:

1. It begins with a certain number of edges already placed, so need indicates how many more edges are necessary. The initial cost of the current forest is init_cost.
2. Repeatedly, the function calls augment(), which attempts to improve the current set of edges via exchange. If an augmentation is successful, calculate() checks whether the resulting cost is promising (i.e., better than the best-known answer) and whether the correct number of edges has been reached.
3. If all needed edges can be integrated successfully while respecting degree constraints and avoiding cycles, calculate returns the final cost. Otherwise, it returns -1 to indicate infeasibility.

E. build_answer() Function

This function implements the enumeration of every possible forests among the first k special vertices. Internally, build_answer() works recursively. The function keeps track of which special vertex it is working on right now. When the edges have been assigned to this vertex, the function move on to the next vertex. When all potential edges has been assigned between all special vertices, the code verify if these chosen edges form a valid forest using DSU, and whether the configuration satisfy degree constraints on each special vertex. If valid, we progress to finishing the MST with the calculate() function.

V. TESTS AND RESULTS

Given the testcase below

```
# Number of vertices and special vertices respectively
10 5
# Label of special vertices and degree constraint
0 5
1 3
2 4
3 2
4 1
# Adjacency Matrix
29 49 33 12 55 15 32 62 37
61 26 15 58 15 22 8 58
37 16 9 39 20 14 58
10 15 40 3 19 55
53 13 37 44 52
23 59 58 4
69 80 29
89 28
48
```

Running the above test case we get

```
Minimum total weight = 95

Edges:
u = 0, v = 4, w = 12
u = 3, v = 7, w = 3
u = 5, v = 9, w = 4
u = 1, v = 8, w = 8
u = 2, v = 5, w = 9
u = 2, v = 8, w = 14
u = 0, v = 6, w = 15
u = 3, v = 5, w = 15
u = 1, v = 6, w = 15
```

Hence we've constructed the degree-constrained minimum spanning tree. Below is the visualization of the tree given by the code.

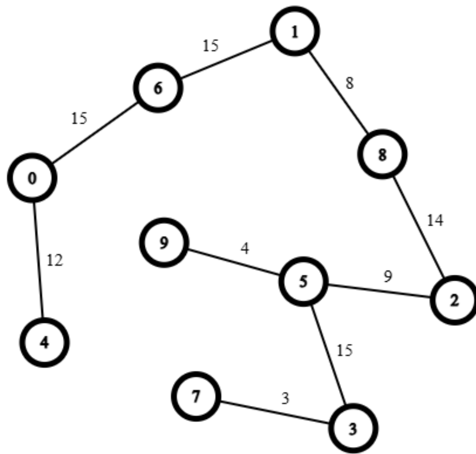


Figure 17. Visual of the constructed degree-constrained MST

The implementation was also tested on the benchmark problem DIY Tree from Codeforces. This problem provides a well-built test cases for evaluating the correctness and efficiency of the solution under realistic constraints. The test cases include various graphs with degree constraints and edge weights, ensuring that the implementation handles a wide range of scenarios effectively. The implementation passed all test cases thus demonstrating its correctness and ability to efficiently solve the problem while adhering to the specified constraints.

Lang	Verdict	Time	Memory
C++17 (GCC 7-32)	Happy New Year!	249 ms	84 KB

Figure 18. Result of testing implementation to benchmark problem

VI. Conclusion

In this paper, we explored the application of matroid theory and the weighted matroid intersection algorithm to solve the degree-constrained minimum spanning tree problem. By leveraging the Edmonds-Lawler theorem and the min-max relation, we developed an approach that guarantees an optimal solution, as demonstrated through testing on the benchmark problem. The implementation successfully handles a wide range of scenarios, consistently providing the correct and optimal spanning tree.

However, while the solution is theoretically sound and achieves optimality, the computational time required to compute the tree can become significant for larger graphs. This shows a potential area for improvement, where further optimizations or heuristic approaches could be explored.

Despite these limitations, the methodology presented here showcases the power of matroid theory in solving constrained optimization problems.

VII. Appendix

GitHub: https://github.com/BP04/weighted-matroid-intersection

VIII. Acknowledgments

The author sincerely thanks God Almighty for providing the strength and opportunity to complete this paper successfully. The author also extends deep appreciation to Dr. Ir. Rinaldi, M.T., lecturer of the IF1220 Discrete Mathematics course, for his guidance, encouragement, and support throughout the semester and during the preparation of this paper. Additionally, the author extends thanks to Mr. Ilya Zylev (ATSTNG) for their insightful and comprehensive blog on matroid intersection, which served as a valuable resource in developing this paper.

References

[1] R. Munir, "Himpunan (Part 1)," Informatika STEI ITB, Bandung, Indonesia. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/02-Himpunan(2024)-1.pdf. [Accessed: Jan. 6, 2025].A

[2] R. Munir, "Himpunan (Part 2)," Informatika STEI ITB, Bandung, Indonesia. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/03-Himpunan(2024)-2.pdf. [Accessed: Jan. 6, 2025].

[3] R. Munir, "Graf (Bagian 1)," Informatika STEI ITB, Bandung, Indonesia. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf. [Accessed: Jan. 6, 2025].

[4] R. Munir, "Graf (Bagian 2)," Informatika STEI ITB, Bandung, Indonesia. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/21-Graf-Bagian2-2024.pdf. [Accessed: Jan. 6, 2025].

[5] R. Munir, "Pohon (Bagian 1)," Informatika STEI ITB, Bandung, Indonesia. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/23-Pohon-Bag1-2024.pdf. [Accessed: Jan. 6, 2025].

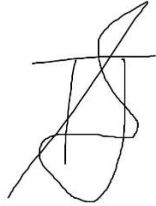
[6] J. Oxley, "Matroids," Cornell University, Ithaca, NY, USA. [Online]. Available: https://www.cs.cornell.edu/courses/cs6820/2022fa/Handouts/oxley-matroids.pdf. [Accessed: Jan. 7, 2025].

[7] TOKI, "Pemrograman Kompetitif Dasar," OSN TOKI. [Online]. Available: https://osn.toki.id/data/pemrograman-kompetitif-dasar.pdf. [Accessed: Jan. 7, 2025].

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 7 Januari 2025



Benedict Presley (13523067)