Determining Connectivity of Telecommunication Networks

MATH 381, Section B, Spring 2024

Bhavana Honavalli, Riya Kulkarni, Nina Mislej, Skyelar M. Reed, Marcus Yim

June 5, 2024

Abstract

In this paper, we present our approach to modeling solutions for connecting nodes in a telecom network, ('telecom' meaning internet, cable, phone, etc). Collaborating with DFG Consulting, (a Slovenian-based company), our team worked with a library of pictorially-represented networks to test whether or not each network had a solution for total system connectivity. A connected network is important for users, providers, and city planners as it gives a definite answer to whether or not a given network can even be constructed. Using two different mathematical programming solvers, we built a model that can conclusively determine a network's connectivity and proved its efficacy using sample networks provided by DFG.

1 Introduction

1.1 DFG Consulting and Telecommunication Networks

DFG Consulting is a Slovenian firm specializing in high-quality data processing services. Operating since 1997, DFG Consulting has handled the management of complex network inventory data, which involves tracking and managing foundational components of network systems, such as routers, switches, and cables [2]. This work includes the **manipulation of data**, like *capturing* (collecting data), *converting* (changing the format of data), and *migrating* (moving data from one system to another) across power, gas, water, and road networks. DFG works with both local and international clients, ensuring that these networks run efficiently.

Our partnership with DFG Consulting focuses on translating data from visual representations of a subset of these network infrastructures; specifically **telecommunication networks**. A telecommunication network is defined as a collection of 'subscribers', (or 'users'), interconnected by wires that provide communication services such as phone lines, internet, and cable television. This collaborative effort aims to improve accuracy and operational efficiency in data migration of communication network plans.

By managing power, gas, water, and road networks, DFG Consulting provides efficient services that are critical for supporting functionality, reducing downtime, and improving service delivery. Additionally, the company's commitment to excellence in developing niche software and network data processing helps simplify operations and enhances efficiency for its clients, thereby supporting local economies and providing communities with reliable services worldwide.

Their team consists of professionals in the fields of programming, geodesy, and marketing. Our primary contact was **Vid Balek**, the lead Software Engineer at DFG Consulting. One of their ongoing projects involves migrating data from paper plans, in the form of pictures, to modern databases. While most of the data in the pictures can be parsed, there is an element of connectivity that cannot be directly read from the images. Consequently, we were tasked with developing a model to address this issue.

1.2 Problem Description

This project's objective is to connect users (houses) to a cable source (manhole) which gives the individual house access to a wider network. Cables are bundled in a protective sheathing that passes through splitters that split each bundle into smaller ones with fewer cables. When a split occurs, we are given the number of cables that are split. However, we do not know the exact cable paths. A more detailed description of the network and the reasoning behind this problem is in the next chapter.

After consultation with Vid, we formed the following goal: Given a network of **manholes, splitters, houses**, and a specified **number of cables connected at each junction**, provide a **set of disjoint cable paths** connecting houses to manholes using the exact number of cables between each junction, or state that a solution does not exist.

2 Telecommunication Network Description

As noted, DFG Consulting works with telecommunication networks. These networks are typically constructed for **optic cables**, though sometimes copper wires or other cable-like objects are used. Such cables originate from **manholes** or shafts. Manholes are underground holes where large cable bundles emerge, often providing network connections for entire areas while protecting the cables from damage. The cables connect to real structures such as apartments, **houses**, and various other buildings. These structures often have an electrical panel, usually mounted on a wall, where all the cables are managed. These are referred to as "end-users" because the cables do not extend beyond these points to other structures. To direct and split the large cable bundles from the manholes, **splitters** are used as intersections to route smaller cable bundles to individual houses. The only exception is when all cables pass through a single node, such as when one house is situated behind another. In this case, the simplest connection method is to extend the cables to the house behind it without adding an intersection. Examples of an electric panel, cables and a manhole can be found in Figures 12, 11, 13 and 14. From this point forward, the buildings of end-users will be referred to as houses, and a cable-like object containing smaller cables will be referred to as a cable bundle.

The given data takes the form of a network with each node being one of three types: **manhole, splitter, or house**. A **junction** defines the directed connection between three nodes in the network. The connections (cable bundles) between these nodes have a corresponding number of cables. Cables must terminate in a manhole or a house, but never at a splitter. DFG Consulting provided detailed network plans that included the diameter and material of the cables in each junction. For an example, please visit Figure 10 in the Appendix to learn more.

Most of the data from each picture has been previously parsed to a database and given to us in a spreadsheet with two sections, one for nodes and one for connections between them. The number of cables in a bundle has to be read directly from the picture. For easier visualization and future testing, the company provided us with an **abstracted simplified example**. We created similar spreadsheets for this case and all our test cases in order to truly capture the entire procedure of the problem, from parsing the data to interpreting the solution. The example they provided can be seen in (Figure 1).

The manholes are depicted as squares, houses as circles, and splitters as rhombuses. The number of cables in a bundle is depicted with circles and a right-angle arrow. It is important to note that the arrow shows the direction of the split, not the direction of the cable (cables are bidirectional). One of the possible solutions (Figure 2) is depicted by **yellow lines** which represent the disjoint paths.

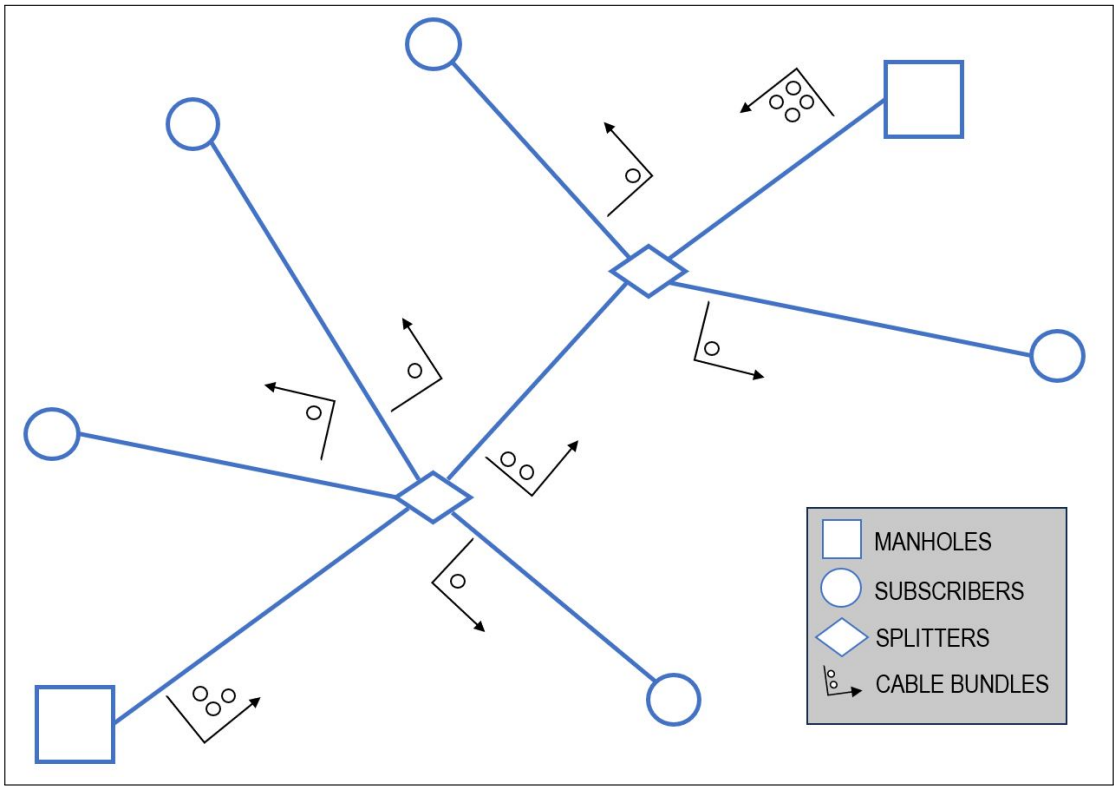

Figure 1: The simplified network provided by DFG Consulting is an example of the network.

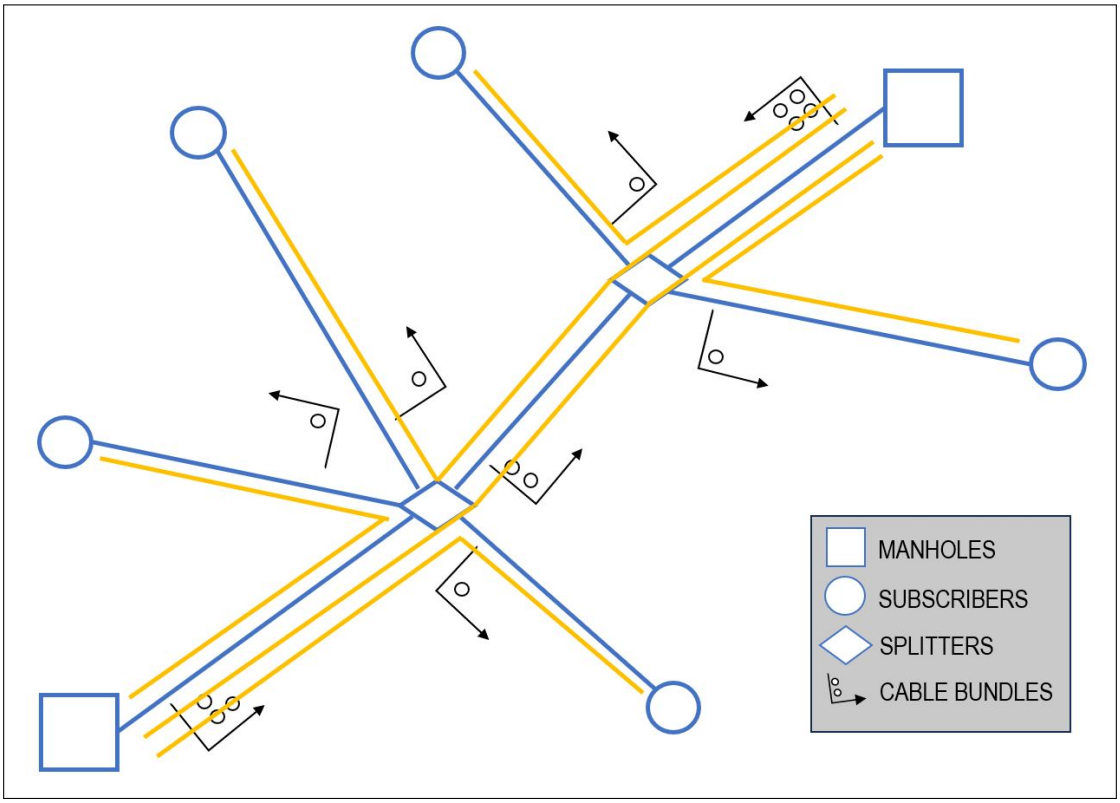

Figure 2: Solution simplified network provided by DFG Consulting.

3 Simplifications

To effectively find a practical solution, we have introduced a few simplifying assumptions to ensure our model is stable while being both applicable and relevant to most real-world scenarios. These assumptions are listed below:

1. In real networks, there exist special cases when certain cable connections end at splitter nodes. These terminations are considered a reserve. This reserve will be used if a house is built in a specified location in the future. For simplicity, our model assumes cable paths cannot end in splitter nodes (i.e. **there are no splitter nodes that terminate a path**). This simplification allows us to focus on the primary objective of connecting houses to the network without the additional layer of managing potential future connections.

2. Cables are made of different alloys and metals. It is optimal, but not necessary, to have one cable of the same material route through the same manholes and splitters. This optimization is not taken into account in our initial mathematical model, and instead, we consider that **cables of different materials can route through the same nodes** during their path. This simplification enables us to focus on finding any viable solution instead of optimizing for material consistency.

3. Cables have different diameters. It is optimal, but not necessary, to have cables of the same diameters route through the same manholes and splitters. This optimization is not taken into account in our initial mathematical model, and instead, we consider that **cables of different diameters can route through the same nodes** during their path. This simplification enables us to focus on finding, first, any viable solution, rather than being constrained by the additional complexity of diameter optimization.

4. We want to connect all houses to manholes, but sometimes such a solution is not possible or the complexity of finding this particular solution among all possible ones is too high. If so **specified by the user**, our Integer Programming algorithm will **allow paths that don't connect to any manholes**, although such solutions will be considered worse than others. The Iterative Greedy Algorithm, will not use this assumption. This simplification is necessary, as we are more concerned about finding any solution that adheres to all the constraints, for ease of data migration. In the case of such solutions, a warning to the user will be issued. Both algorithms are described in the next section.

5. We assume that houses are end (leaf) nodes, meaning **no cables route through them**. This assumption is based on the presence of splitter nodes, which serve the function of routing cables, while houses do not. There is one exception to this rule: when a house connects to another house located behind it, either all cables route through the front house, or none do. If this assumption causes a feasibility issue in the network, a splitter node can be added at the problematic location. One should then solve the model for the modified network to obtain a solution.

These simplifications were formalized to aid in future improvements to streamline our model at a later stage.

4 Mathematical Models

To tackle our objective, we developed a mathematical model that leverages Integer Programming and an Iterative Greedy Algorithm inspired by Dijkstra's Algorithm and the A* Algorithm.

We initiated our analysis with **Integer Programming** (IP) to explore all potential solutions. IP is an optimization technique that determines the maximum or minimum of a given function, while ensuring that the resulting solution is an integer value, without fractions, decimals, or roots (see [5] for detailed technical definitions). The rationale behind employing this algorithm lies in the deterministic nature of our problem, where all constraints are predetermined, and there are no stochastic, or random events involved. By adopting this method first, we gained insights into which heuristics would be beneficial, identified solutions that are impractical in real-world scenarios, and assessed how solutions evolve as the number of nodes in the network increases. This approach provided us with a deeper understanding of edge cases and details that require careful consideration.

Drawing from this understanding, we devised an Iterative Greedy Algorithm that integrates elements of Dijkstra's Algorithm, A* search, and state-space searches to ascertain the optimal cable paths within a given telecommunication graph with confidence. Dijkstra's Algorithm and the A* Algorithm are systematic search algorithms that can identify the shortest path from any specified starting node to any other node in the network (see [6]). Both algorithms are classified as greedy algorithms, as they prioritize nodes closest to the starting point. A state-space search involves a methodical enumeration of all feasible solutions (in our case, cable paths) to a problem, with a focus on identifying the optimal state. Our Iterative Greedy Algorithm systematically explores potential combinations of cable paths to identify a feasible state that connects all nodes in the network [6].

4.1 Comparing the Algorithms

We decided to create two algorithms to tackle the following two objectives: 1) **analyze the feasible region of solutions for each network** and 2) **find solutions quickly without searching all possibilities**.

Integer Programming tackles objective 1) by providing all plausible solutions that satisfy the constraints, and thus determining the connectivity of any given network. Our IP model enumerates every possible combination of junctions, such that the constraints are fulfilled. Thus, if an IP model fails to find a solution, it is assumed that the Iterative Greedy Algorithm cannot find one either. However, as mentioned in Section 1, DFG manages a complex network inventory, which contains hundreds of network plans. Thus, for efficiency purposes, the processing speed of our model must be within a reasonable time frame. In general, finding every solution takes a great deal of time and thus is ineffectual in addressing objective 2). We have two options: either limit the number of solutions saved by the IP model, which our IP solver supports, or develop a simpler solver called the Iterative Greedy Algorithm. We enable both options because the Greedy Algorithm simplifies the integration process for the company. It is less dependent on the programming language and does not require a detailed understanding of the mathematical concepts behind IP models.

The Iterative Greedy Algorithm serves as a pragmatic solution, offering speed and efficiency. Moreover, from a programming perspective, the simplicity and minimal external dependencies of this algorithm make it easier to integrate into the existing DFG Consulting system. The company primarily uses the **C++ programming** language, which is considered one of the fastest programming languages as its code is compiled directly into machine code. We used **Python** to implement our algorithm, which is considered slower, so when the code is implemented into their environment the quality of performance time should be guaranteed. Additionally, we require the analysis of the IP model for instances when the Iterative Greedy Algorithm fails, as this algorithm is not guaranteed to find a solution, even when one might exist.

In order to increase the probability of identifying a solution, the algorithm can be reiterated with different random seed nodes. If the Iterative Greedy Algorithm fails to produce a satisfactory solution within an acceptable time frame, alternative solutions suggested by the Integer Programming model can be examined. This approach ensures a comprehensive exploration of potential solutions while balancing computational efficiency and solution quality.

4.2 Notation and Terminology

We will be using common Graph Theory terminology in the next few sections. A **graph** $\boldsymbol{G}$ is a pair $\boldsymbol{(V, E)}$ of 2 sets. Set $\boldsymbol{V}$ represents vertices (also known as nodes) and set $\boldsymbol{E}$ represents edges, where each edge is a pair of vertices. We can interpret this mathematical structure as a diagram showing pairwise relations (edges) between different objects (vertices). More explicit descriptions of these concepts can be found in Graph Theory, 2008 [1, p. 2].

A graph can be directed or undirected. We will often use the notation $\boldsymbol{ab} \in \boldsymbol{E}$, where $\boldsymbol{ab}$ is short for an ordered pair of vertices $\boldsymbol{a, b} \in \boldsymbol{V}$. This way we can determine the direction and position of the edge from the notation alone. When allowing multiple edges to go from one vertex to another, the structure is called a multigraph. If more details about the connections are needed, edge weights can be used. These provide additional information

about edges and can take up any real value [1, p. 50]. Because these weights are often associated with the cost of traversing the edge, we will denote the set of weights $\boldsymbol{C}$ and for each edge $\boldsymbol{e} \in \boldsymbol{E}$ we denote $\boldsymbol{c_e} \in \boldsymbol{C}$ the weight on edge $\boldsymbol{e}$.

The manholes, subscribers, and splitters in the telecommunications network are vertices and the cable bundles are edges. The graph structure modeled directly from the data provided by the company is undirected. The number of cables in each bundle is interpreted as a weight on the edge associated with the corresponding bundle.

A remarkably important term in this problem description is the **junction** which does not have a standard naming convention. For the context of this paper, the junction is a tuple of 3 nodes (i. e. 2 directed edges) $\boldsymbol{a, b, c} \in \boldsymbol{V}$ so that there exist edges $\boldsymbol{ab, bc} \in \boldsymbol{E}$. We shorten the notation of a tuple $\boldsymbol{(a, b, c)}$ to $\boldsymbol{abc}$. It is also important to note that a junction is directed.

In both algorithms, there are some modifications to the original graph structure. We modified each undirected edge in a network into a bidirectional edge in the corresponding directed edge. The Iterative Greedy Algorithm works with a multigraph. Instead of having one edge for the cable bundle, we rather use multiple edges with one edge representing one cable in the bundle. The reasoning behind the modifications is provided in the next 2 sections.

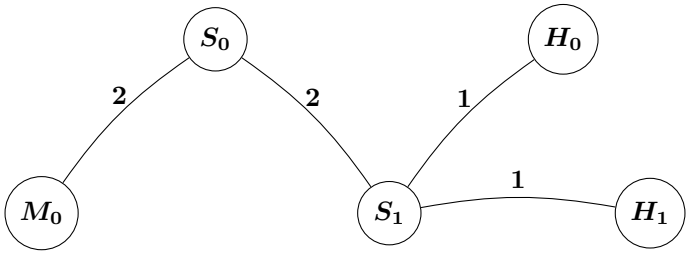

Figure 3: A simple example of our data interpreted as a graph.

Above in Figure 3 we have a small example of an undirected graph, similar to the one from our original data. Houses are marked with $\boldsymbol{H}$, manholes with $\boldsymbol{M}$, and splitters with an $\boldsymbol{S}$. To illustrate some of the terminology introduced, one example of a node would be $\boldsymbol{H_0}$. An example of an edge going from this node is $\boldsymbol{H_0S_1}$. If we treat this graph as directed, then the corresponding edge going in the opposite direction is $\boldsymbol{S_1H_0}$. An edge weight for this edge is $\boldsymbol{c_{H_0S_1}}$ and its value is $\boldsymbol{1}$. Finally, a directed junction that starts on this edge is $\boldsymbol{H_0S_1S_0}$ and the opposite direction of this is $\boldsymbol{S_0S_1H_0}$.

We can also illustrate the first (Figure 4), fourth (Figure 5) and fifth (Figure 6) **simplifying assumption** using graphs. The red cross marks denote the ones that are *not* allowed as an input, while the green check mark represents the ones that *are* allowed.

Figure 4: A visualization of the 1st simplifying assumption. A splitter should not terminate a path.

Figure 5: A visualization of the 4th simplifying assumption. Paths with houses connected to manholes are preferred.

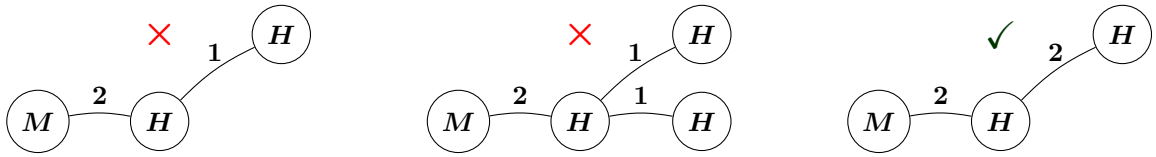

Figure 6: A visualization of the 5th simplifying assumption. A house either routes all the cables forward or none, and it cannot act as a splitter.

4.3 Integer Programming

4.3.1 Model Overview

Our IP optimization technique consists of a **linear objective function**, with linear equality and inequality **constraints**[5, p. 49]. The components of these are called variables; because this is an integer programming problem, they can only take on integer values for coefficients for each of the objective functions, equalities, and inequalities; as well as variables. Many theorems and definitions come from Linear Programming [5, p. 92]. For those familiar with some linear algebra, the Linear Program, (as well as the Integer Program), can always be written in standard matrix form:

$$\begin{aligned} \min \quad & \boldsymbol{c}^\intercal \boldsymbol{x} \\ \text{s.t.} \quad & \boldsymbol{A}\boldsymbol{x} = \boldsymbol{b} \\ & \boldsymbol{x} \geq \boldsymbol{0} \end{aligned}$$

The $\boldsymbol{x} \in \mathbb{Z}^{\boldsymbol{n}}$ is the vector of variables, $\boldsymbol{c} \in \mathbb{Z}^{\boldsymbol{n}}$ is the vector of the coefficients of the objective function, $\boldsymbol{b} \in \mathbb{Z}^{\boldsymbol{m}}$ is the vector representing the right side of the constraints, and $\boldsymbol{A} \in \mathbb{Z}^{\boldsymbol{m} \times \boldsymbol{n}}$ is the matrix with the coefficients of each inequality. Many different graph modeling problems can be solved using Linear Programming or Integer Programming techniques.

The input to our IP Algorithm is a telecommunication network, represented as a weighted and directed graph $\boldsymbol{G}$ being modified as described in the previous section.

$$\boldsymbol{x} = [\ \boldsymbol{M_0S_0S_1}\ ,\ \boldsymbol{S_1S_0M_0}\ ,\ \boldsymbol{S_0S_1H_0}\ ,\ \boldsymbol{H_0S_1S_0}\ ,\ \boldsymbol{H_1S_1S_0}\ ,\ \boldsymbol{H_1S_1S_0}\ ,\ \boldsymbol{H_1S_1H_0}\ ,\ \boldsymbol{H_0S_1H_1}\ ] \quad (1)$$

Variables of the IP model correspond to junctions in the graph. This means that for 2 existing edges $\boldsymbol{ab}, \boldsymbol{bc} \in \boldsymbol{E}$ there is a variable $\boldsymbol{x_{abc}}$. The exception is when node $\boldsymbol{b}$ is a manhole node. Because cables originate from manholes, we do not wish to connect any cables through manhole nodes, only splitters, and houses; therefore we can ignore the variables with a manhole node in the center of the junction. This value represents the number of cables routing from edge $\boldsymbol{ab}$ to edge $\boldsymbol{bc}$; which justifies why we need a directed graph because the cables could be routed in either direction (e.g. from $\boldsymbol{cb}$ to $\boldsymbol{ba}$). All the variables from Figure 3 are in (1).

$$\boldsymbol{M_0S_0S_1 + S_1S_0M_0 + H_0S_1S_0 + S_0S_1H_0 + H_1S_1S_0 + H_1S_1S_0 = 2} \quad (2)$$

Our **constraints** are inspired by the constraint usually used in network flow problems, wherein the flow coming into a node should be the same as the flow leaving that node, and the flow through one edge should not be bigger than the corresponding edge weight [5, p. 174]. In our case, the sum of cables routing from $\boldsymbol{e} \in \boldsymbol{E}$ to anywhere, and the cables routing from anywhere to $\boldsymbol{e}$ should equal the weight on the $\boldsymbol{e}$ edge for any $\boldsymbol{e}$. This means that the right side of our constraints (vector $\boldsymbol{b}$ in the equation) is the graph weights and the coefficients (matrix A) are **1** if either one of the edges in the junction is the edge of which the weight is on the right side and **0** otherwise. One implicit constraint is that all variables should be non-negative. This is our reasoning for using

Integer Programming and not Linear Programming, as the number of cables in each bundle has to be an integer. An example of a constraint is (3) from Figure 3.

$$\min \boldsymbol{H_1 S_1 H_0 + H_0 S_1 H_1} \tag{3}$$

The **objective function** is the sum of all $\boldsymbol{x_{abc}}$ where $\boldsymbol{a, c}$ are nodes representing subscribers which finds the minimal number of junctions connecting two houses. This minimizes the probability of cable paths that do not traverse manholes (due to simplification 4). After obtaining the list of solutions, we iterate through them and check if every given path connects to a manhole. We stop after finding the first solution that has such paths. This is a **time complexity optimization**, on account of this particular objective function, solutions at the start of the list have a higher likelihood of having this characteristic satisfied. Example of the objective function for Figure 3 is (3).

This model produces numerous solutions, however, many are the same except for the direction of the paths. We can reconstruct each solution to match real-world structures by drawing connections so that the number of connections at one junction corresponds to the value of the variable representing the junction. We interpreted the result as a multigraph with different **colored disjoint paths**, so the solution is readable from the graph itself. We also provided cable paths as lists, as visualization is not a key component of the solution.

To formulate our problem and search for a solution using our Integer Programming approach, we applied **Gurobi Optimizer**. This optimization solver has been widely used to solve problems in Mixed Integer Programming, Linear Programming, and Quadratic Programming. It includes several facilities such as pre-solve techniques, advanced heuristics, and parallel computing capabilities. This functionality highly reduces computation time making Gurobi suitable for handling large-scale, complex optimization problems. Gurobi's ability to handle a large number of variables and constraints allowed us to explore numerous configurations for our IP approach to find the best possible method for a solution.

4.3.2 Solution Interpretation

To obtain the paths from junction decision variables after we solved the IP problem, we defined an algorithm that **stitches the edges** from junctions together into a path. This is part of our **interpretation** of the mathematical solution into a real-world solution. One junction will exist in as many paths as its junction variable value. For example, if the variable $\boldsymbol{x_{abc}}$ is **2** it will exist in 2 paths. Once we added the junction to all possible paths, we say the junction has been connected. We start with a set of paths $\boldsymbol{P}$ and a set of variables $\boldsymbol{U}$ that have already been connected. In the beginning, both are empty. Here is one step of the algorithm:

1. We take one of the variables that is not in $\boldsymbol{U}$ and start building a path from it by adding all **3** nodes in the junction to the path.
2. We subtract **1** from the variable value (obtained in the solution) and if it reaches zero, it means the junction is connected, so we add it to $\boldsymbol{U}$.
3. We iterate through all the other variables not in $\boldsymbol{U}$:
    - There are four cases in which we stitch the junction into the path. The first two are if the junction matches the start of the path (either in the direction of the path or the opposite) and the second two if it matches the end of the path.
    - If a junction was added, we subtract **1** from the junction variable value; and if it reaches zero, we add it to $\boldsymbol{U}$, just like before. A special case occurs when we don't add a junction but subtract **1**. This happens when a cycle is found.
    - We continue iterating until some junction is added. Once no more changes are made, (we made a full iteration without any junction being stitched), we have exhausted all options, and the path is added to $\boldsymbol{P}$.

4. We then return to the first step. When there are no options to choose from, (because all of them are in $\boldsymbol{U}$), the algorithm stops. This means that all junctions have been connected (i.e. each junction is in as many paths as is its variable value).

5. Finally, we must add specific cases that are not captured in junction variables. After we stitched all of the junctions, we had to add paths that consist of **one edge only** (therefore no junction, so there is no variable). These are the paths that start at a manhole and go to a house where they terminate immediately, and one edge path connects 2 manholes together.

We are left with a set of paths $\boldsymbol{P}$ which is our interpretation of the IP solution.

4.4 Iterative Greedy Algorithm

4.4.1 Heuristics

A **heuristic** is any algorithm that ranks a set of things. For example, a kid's opinion of ice cream flavors. If we ask this kid to compare two arbitrary flavors of ice cream over and over again, we could use their opinions comparing each to create a ranked list of flavors.

Our algorithm can be altered to use any desired heuristic. The currently implemented heuristic ranks edges between nodes based on their weight, which signifies the number of cables needed along that edge. Lower weights are ranked higher, so an edge with weight 1 would be ranked higher than an edge with weight 2. This heuristic, from here on, will be referred to as the minimum edge weight heuristic.

4.4.2 Overview of the Algorithm

The second of our two solutions is a modified version of Dijkstra's Algorithm that uses a heuristic [3]. In essence, we run this algorithm iteratively until our constraints are satisfied. This algorithm allows for the quick determination of a solution (if one exists) at the cost of optimizing constraints such as cable material, diameter, or preferentially choosing paths with manholes. Traditional greedy algorithms often fail to guarantee a solution, especially when the solution space is limited to begin with. To address this limitation, we developed a program that applies a randomly seeded greedy algorithm until a solution is found. Graphs with many valid solutions can be solved quickly with the Iterative Greedy Algorithm, while graphs with few valid solutions may be solved after many iterations. The algorithm is described below, in brief:

- Select one manhole node from a random sequence as a seed node
- Rank all neighboring nodes using some heuristic (e.g. lowest-cable edge)
- Follow the edge with the highest ranking and add that edge to a path
- Stop when we reach a leaf node
- Save that path and repeat the process to create a list of disjoint paths
- Evaluate the disjoint set
- Repeat the algorithm with a different sequence of manhole nodes until a valid set is found

One of the solutions we explored is using an Iterative Greedy Algorithm. In brief, an iterative greedy algorithm does not backtrack and makes decisions based on **local information**.

As mentioned in Section 3, all cable paths found by the Iterative Greedy Algorithm include manholes. To find solutions with both manhole paths and non-manhole paths, we would require some mechanism to rank the

solutions. This ranking would take the form of optimizing the solutions. As greedy algorithms are not conducive to do optimization, we assume all cable paths must include manholes.

The **no backtracking** rule means that our algorithm will only ever consider nodes one time. After the algorithm has evaluated a node, we will never re-evaluate it. This is a desirable feature for finding solutions quickly, at the cost of not checking every possible solution.

The second part of an Iterative Greedy Algorithm is only making decisions locally. What that means is that when our algorithm is at a given node, we will decide which future node to travel to based only on the given node's neighbors. This means fewer nodes to evaluate and a faster algorithm at the cost of having an over-arching goal. This also means that the algorithm can create an unsolvable situation, as it does not consider an entire cable path from manhole to subscriber. Similar to chess, where it is easy to fall into a trap if you only look one move ahead.

For the real-world problem, where graphs are not significantly difficult and time is of the greatest importance, Greedy Algorithms fit it well.

5 Solution For Theoretical Example

The analysis of a solution is **illustrated using the theoretical graph** presented by DFG in Figure 1. In Figure 7, we depict our input for both algorithms, derived from the data in spreadsheet format and converted into a graph. This visualization is generated directly from the program and is reproducible. The circles depicting the number of cables in a bundle in Figure 1 are converted to edge weights. The manholes are represented with the letter $\boldsymbol{M}$, the splitters with $\boldsymbol{S}$, and the subscribers with $\boldsymbol{H}$.

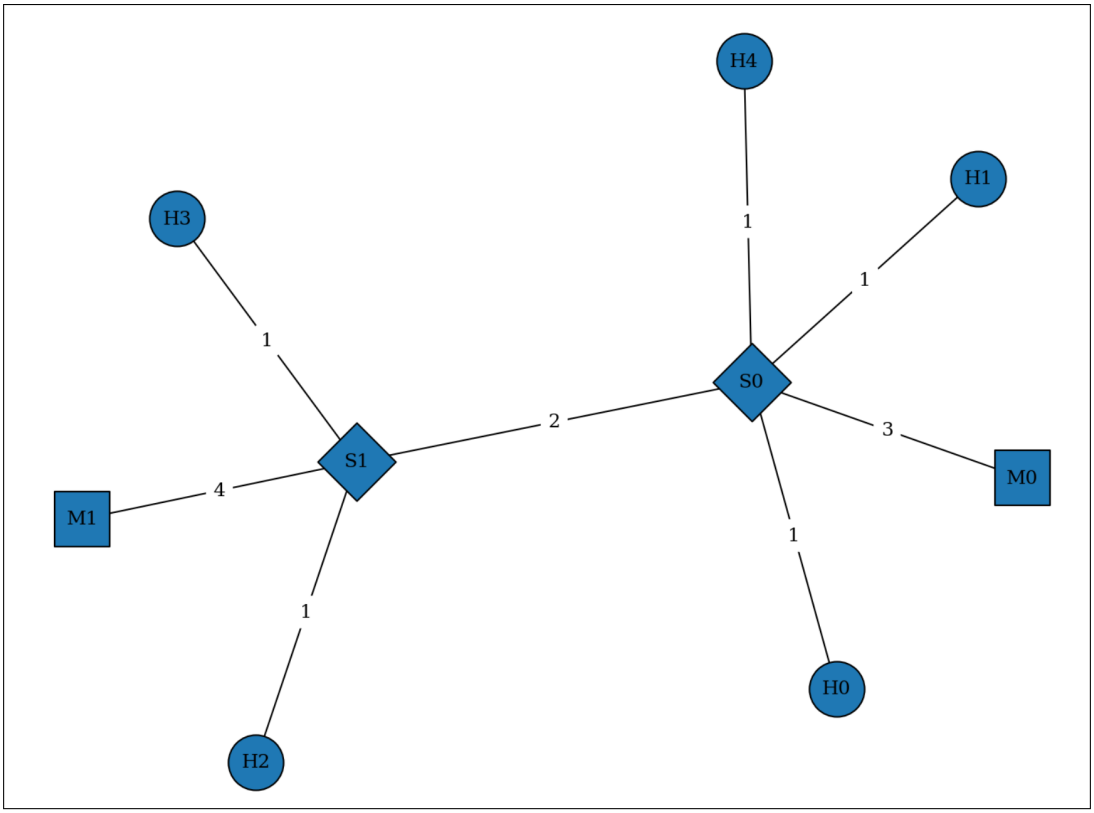

Figure 7: This is the abstracted network DFG company provided us implemented as a `networkx` undirected graph object with weights and our input for the model.

In Tables 1 and 2 the data is presented in its **raw form**. This format mirrors the data structure provided by DFG Consulting, thereby encapsulating the entire process from parsing onward. The column names remain faithful to those in their respective databases; however, only pertinent data is displayed, with other columns omitted.

str_name	ref_name
H0	OS
H1	OS
H2	OS
H3	OS
H4	OS
S0	TS
S1	TS
M0	PJ
M1	PJ

Table 1: This table represents the nodes for the graph depicted in Figure 7 in spreadsheet form. The first column denotes node names, while the second column denotes node types.

span_name	to_str_name	from_str_name	weights
S0H0	S0	H0	1
S0H1	S0	H1	1
S0H4	S0	H4	1
S0M0	S0	M0	3
S0S1	S0	S1	2
S1H2	S1	H2	1
S1H3	S1	H3	1
S1M1	S1	M1	4

Table 2: This spreadsheet provides the edge details for the graph depicted in Figure 7. The first column lists the edge names, the second column indicates the end node of each edge, the third column represents the start node, and the fourth column displays the corresponding edge weights.

In the sections below, we present solutions derived from both the IP model and the Iterative Greedy Algorithm based on this input.

5.1 Output: Integer Programming

There are **32** different variables utilized for solving this graph, along with **9** constraints. The integer programming model yielded **1008** solutions, however, only **576** of these have paths that all connect to a manhole. Upon iterating through solutions and verifying connectivity to at least one manhole for all paths, we identified that the **first** saved solution met this criterion.

```
M0 S0 H1: 1.0
M0 S0 H4: 1.0
M1 S1 H2: 1.0
M1 S1 H3: 1.0
S0 S1 M1: 2.0
S1 S0 H0: 1.0
S1 S0 M0: 1.0
```

Here we have the direct output of the **junction variable values** that are non-zero. This means that we want to route **1** cable from $\boldsymbol{M_0}$ through $\boldsymbol{S_0}$ to $\boldsymbol{H_1}$. Another example, **2** cables go from $\boldsymbol{S_0}$ through $\boldsymbol{S_1}$ to $\boldsymbol{M_1}$. The procedure is the same for the other nodes. Now we have to convert these values into corresponding paths using the algorithm we described earlier in section 4.3.2.

```
0: M0, S0, H1
1: M0, S0, H4
2: M1, S1, H2
3: M1, S1, H3
4: H0, S0, S1, M1
5: M1, S1, S0, M0
```

These paths are generated from the aforementioned variables. For instance, the last path begins at manhole $\boldsymbol{M_1}$, proceeds through splitter $\boldsymbol{S_1}$ and splitter $\boldsymbol{S_0}$, and terminates at another manhole $\boldsymbol{M_0}$. By manually inspecting these paths and counting the occurrences of each edge within all paths, we can confirm that they indeed adhere to the constraints.

We can visualize this result on a multigraph. Each of the disjoint paths is represented with a different color.

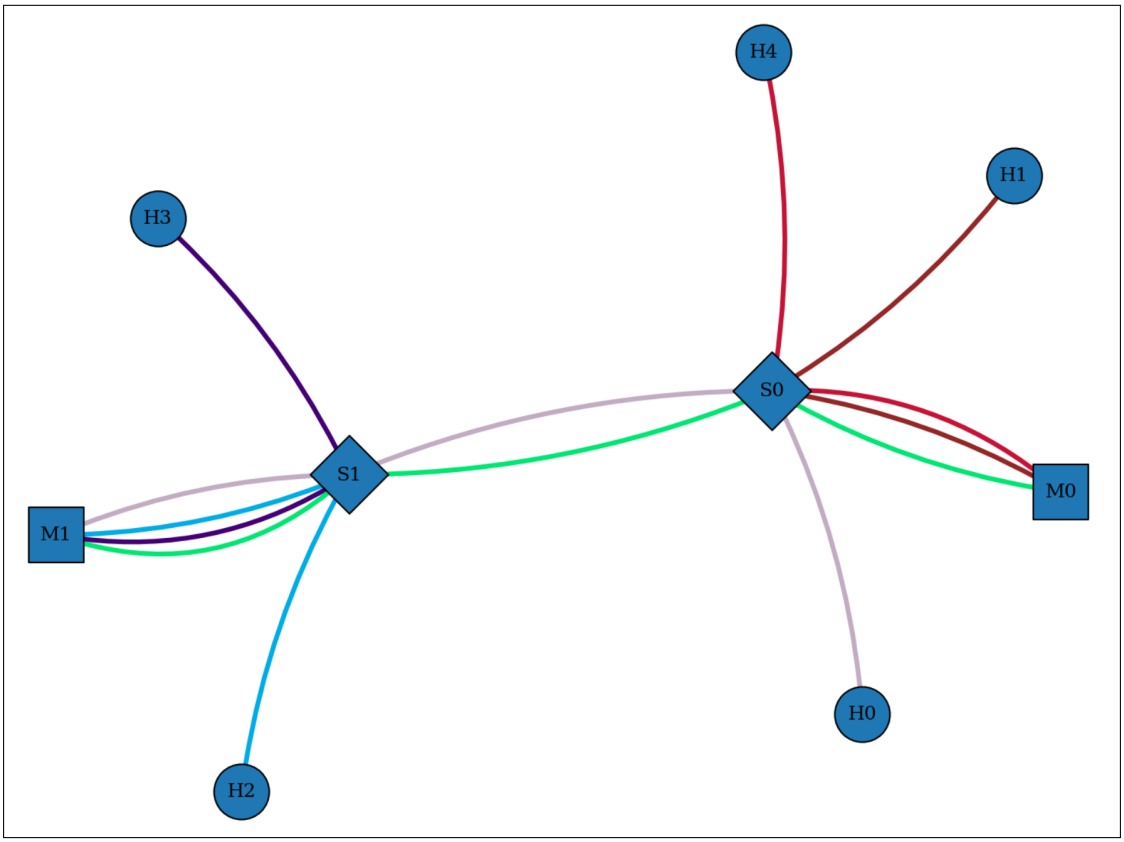

Figure 8: The solution provided by the IP to the network in Figure 7 is presented with each cable path represented in a different color.

We state that we found **1008** solutions, so if we examine more solutions from the IP, we can find solutions different from the one in the above figure.

```
0: H0, S0, S1, M1
1: H1, S0, M0
2: H4, S0, M0
3: M1, S1, H2
4: M1, S1, H3
5: M0, S0, S1, M1
```

For example, this is the **2. solution** the model saved. This is an example of an equal solution, where the difference is just direction. Here are exactly three paths in which direction differs from the previous solution. These are the ones with indexes **1**, **2** and **5**. We mentioned such examples when describing the model.

After further analysis, most of the solutions, yield the same paths it is just the direction of the paths that is different. Given that we have **6** paths that can go in either direction, that means **64** identical options from changing direction alone.

We can see that IP served us as a great analytical tool, but we wanted to offer another solution that is simpler to integrate and easier to develop further.

5.2 Output: Iterative Greedy Algorithm

```
0: M0, S0, H1
1: M1, S1, H3
2: M1, S1, H2
3: M1, S1, S0, H0
4: M0, S0, S1, M1
5: M0, S0, H4
```

We have implemented 2 different heuristic options for the algorithm. Presented here is the solution generated by the Iterative Greedy Algorithm, with the **minimum edge weight heuristic**. This solution routes **1** cable from $\boldsymbol{M_0}$ through $\boldsymbol{S_0}$ to $\boldsymbol{H_1}$. Another example, **2** cables go from $\boldsymbol{M_1}$ through $\boldsymbol{S_1}$ then split to reach $\boldsymbol{H_3}$ and $\boldsymbol{H_2}$. The procedure is the same for the other nodes.

The result produced by the Iterative Greedy Algorithm with minimum edge weight heuristic is **the same** as the first Integer Programming solution, shown above. Upon further observation, it can be seen that this generated solution matches precisely with the theoretical solution shown previously in Figure 1. It is important to note that this is not guaranteed, and is in this example just a coincidence.

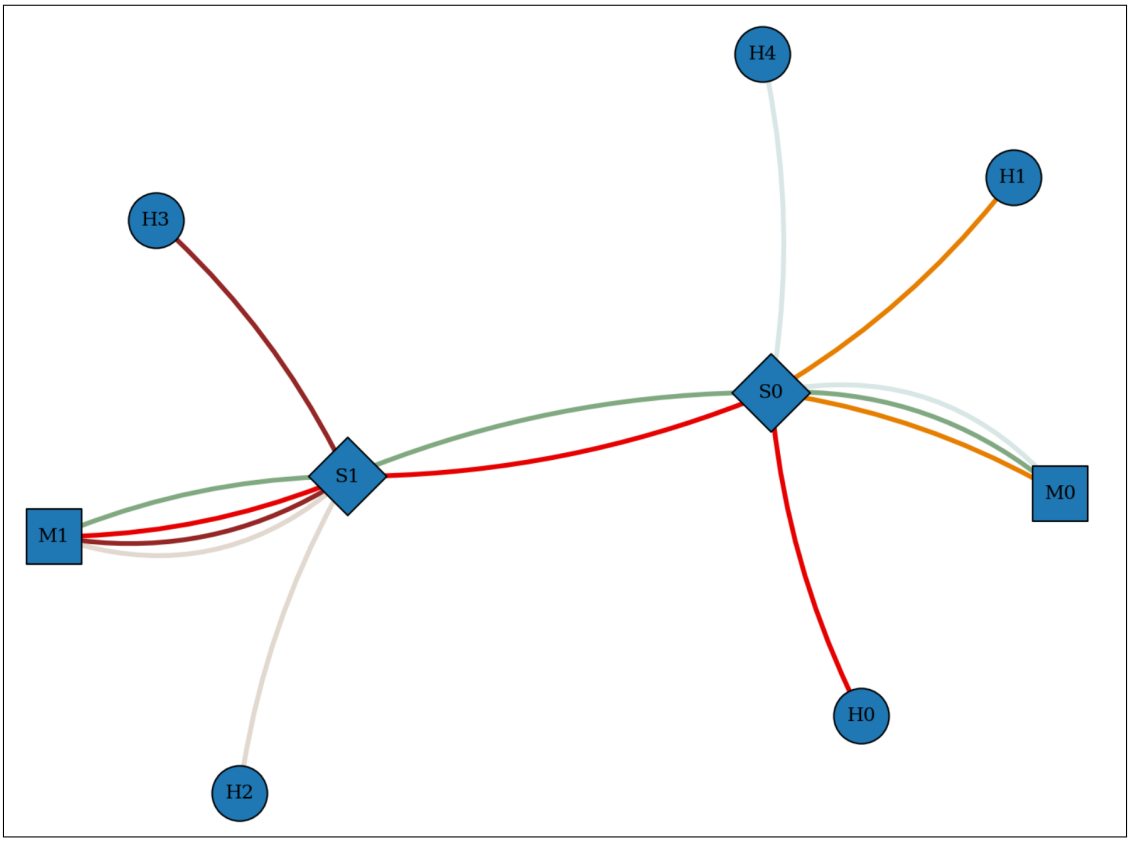

Figure 9: The solution provided by the Iterative Greedy Algorithm with a minimum edge weight heuristic to the network in Figure 7 is presented. Each cable path is represented in a different color.

```
0: M0, S0, S1, M1
1: M1, S1, H3
2: M1, S1, S0, H1
3: M0, S0, H0
4: M0, S0, H4
5: M1, S1, H2
```

Here is the solution generated by the Iterative Greedy Algorithm, with **random heuristic**. This solution differs greatly from both distinct solutions we examined so far, as here $\boldsymbol{H_1}$ and $\boldsymbol{H_0}$ are connected to a different manhole than in the previous solutions; $\boldsymbol{H_1}$ and $\boldsymbol{H_0}$ switched their manhole connections, which can imply some changes in connectivity for both users.

6 Results

Both of these models are implemented in `Python` and are available as write-ups and standalone scripts in our **GitHub repository**, along with our test cases and their descriptions.[4]. We also provided a shell script, for those who are familiar with it, to batch test both of our models.

We came to an agreement with DFG to test our algorithms on **theoretical cases** instead of the two real network plans given to us. In order to use real networks, we have to make the cases anonymous, which slows down the process of obtaining a large enough sample. Furthermore, during the development of our mathematical model, we did not have access to DFG's existing systems for network visualization. Thus, computing edge weights for each edge from the existing system was out of the scope of our project. Therefore, after much discussion, we jointly concluded that we would produce theoretical test cases to ensure our model works on key exceptions and edge cases as seen in real networks. These edge cases were approved by Vid Balek, our community contact.

To validate our mathematical model, we developed **15 test cases** to ensure our solvers can determine feasibility for numerous types of networks. While developing these cases, we addressed various edge cases such as cyclic connections, isolated nodes, etc. while simultaneously increasing the number of nodes in each case. Currently, the largest network we have tested has **35 nodes**. Visualizations of all 15 test cases and their respective solutions can be found in Section 11.3 of the Appendix.

Test Case	**Number of Nodes**	**Has Cycles**	**Connected Components**	**All paths have a Manhole**	**Number of Solutions (from IP)**
1	9	No	1	Yes	1008
2	5	No	1	Yes	216
3	4	Yes	1	Yes	27
4	3	Yes	1	Yes	25
5	7	No	1	Yes	18
6	4	No	1	Yes	1
7	10	No	1	Yes	54
8	13	No	1	Yes	1536
9	15	No	1	Yes	175,104
10	15	No	1	Yes	N/A
11	5	Yes	1	Yes	32
12	11	Yes	2	Yes	512
13	5	No	1	No	75
14	19	No	1	Yes	162,000
15	35	No	2	Yes	N/A

Table 3: Table describing the test cases and the number of solutions found by the IP model. If the number of solutions is denoted as N/A, it means that calculating all possible solutions was computationally exhaustive, and the program was terminated after 10 minutes.

Here are a few key details about the test cases described in Table 3 above:

- Test Case **1**: this network is the same as the theoretical case in the Solution For Theoretical Example.
- Test Case **2**: This network is a **straight line** with **5** nodes, where each edge has a greater weight than the total number of junctions. This was used to test path-stitching from the IP model solution, as the algorithm has to make more than one iteration over the **3** junction variables.
- Test Case **3**: This network is an **even cycle**
- Test Case **4**: This network is an **odd cycle**.
- Test Case **6**: This network has **no junction variables** because it is a star-shaped graph with a manhole

connected directly to each of the three houses in the network.

- Test Cases **8** and **12**: The Iterative Greedy Algorithm does not find a solution using the minimum edge weight heuristic, but does find a solution using the random heuristic.
- Test Cases **14** and **15**: Our community partner estimated the network could have up to 30 nodes. Therefore, both networks aim to **scale** the problem up to a larger number of nodes, with 19 and 35 nodes respectively.
- Test **12** and **15**: These networks have more than one connected component. This means that all nodes are not connected in one graph, but rather are separated into two connected graphs.
- Test Case **13**: This case cannot be solved with either heuristic from the Iterative Greedy Algorithm. Reiterating on of our key variances between the IP model, and the Greedy Algorithm, the IP model is able to construct paths without manhole connections, while the Greedy Algorithm must construct every path with a manhole connection. This variance is noted in test case 13, where none of the **75** solutions produced by the IP model ensure that every cable path is connected to a manhole. Thus, neither heuristic of the Iterative Greedy Algorithm is guaranteed to find a solution.

During our testing, we were interested in determining an in-practice **runtime** of both our algorithms to evaluate the runtime difference between the Iterative Greedy Algorithm and the IP model. Additionally, we wanted to evaluate how variance in heuristic type affected runtime. We compared two heuristics: the minimum edge weight heuristic and the random heuristic that ranks nodes randomly (visit Section Mathematical Models for a detailed explanation of both heuristics). Our summarized results for all test cases are shown in Table 4 below.

To ensure comparable testing, all three of these were run sequentially on an 11th Gen i7-11850H processor with 32 GB RAM and an Nvidia RTX A3000 graphics card from a shell script that is available in the GitHub repository[4]. We allowed the Iterative Greedy Algorithm to make **10000** runs and set the same value for the number of solutions the IP model saved. This ensures that one algorithm is not more favored than the other, and makes our runtime comparison of both algorithms more accurate.

Test Case	**Runtime Integer Programming (sec)**	**Runtime Minimum Edge Weight Heuristic (sec)**	**Runtime Random Heuristic (sec)**
1	0.2694	0.2382	0.2320
2	0.2163	0.2667	0.2388
3	0.2321	0.2530	0.2385
4	0.2383	0.2315	0.2383
5	0.2163	0.2531	0.2227
6	0.2405	0.2318	0.2387
7	0.2315	0.2476	0.2386
8	0.2319	N/A	0.2384
9	0.6165	0.2472	0.2291
10	0.6261	0.2478	0.2284
11	0.2360	0.2384	0.2318
12	0.2314	N/A	0.2382
13	0.2318	N/A	N/A
14	0.6290	N/A	0.4979
15	0.7885	N/A	0.8247
Average Runtime:	0.3490	0.2455	0.2954

Table 4: Table showcasing runtimes of both algorithms. The N/A entry marks cases in which a solution was not found. Such entries are excluded from the average.

After a few calculations, we found the average runtimes of each algorithm. The Integer Programming model has an average runtime of **0.3490** seconds. It is important to keep in mind that this is the average time the IP model takes to produce **10000** solutions, not all of them; if we wish to obtain all solutions to the IP model,

the algorithm can take significantly longer to complete. For example, with test case 9, the IP model generated 175,104 solutions, which took a total of 272.1416 seconds. The average runtime of the Minimum Edge Weight heuristic, excluding cases 8, 12, 13, 14, and 15, is **0.2455** seconds. The average runtime of the Random heuristic, excluding test case 13, is **0.2954** seconds. This shows that the Greedy Algorithm is indeed **faster** in most cases but less **reliable** at finding a solution. Regarding time complexity, our team feels that DFG Consulting will have space to make any desired improvements to this algorithm, as the Greedy Algorithm is not as mathematically difficult and is therefore easier to understand and modify.

To confirm that these solutions are indeed correct and that they meet all the specified junction constraints, we manually traced the solution to every test case on the original network. Additionally, we shared the test cases and their solutions with Vid to verify their accuracy, all of them were approved as correct solutions.

7 Improvements

Future work will center on increasing accessibility, and finding optimized solutions to network plans, that account for cable qualifications, and random chance in telecommunication graphs.

- Cables are made of different alloys and have different diameters. It is optimal, but not necessary, to have cables of similar materials and diameters route through the same manholes and splitters. Future implementations should include these soft constraints to pick ideal cable paths.
- Due to using an Iterative Greedy Algorithm, our method creates an opportunity to determine that no solution exists to a graph, when, in truth, connectivity is possible. This outcome can be mitigated by integrating more non-deterministic search patterns.
- The current heuristic choices in our Iterative Greedy Algorithm are limited and may not always yield the best possible solutions. Future work should focus on developing and incorporating more intelligent heuristic choices to enhance the efficiency and accuracy of the algorithm in identifying optimal cable paths.
- To ensure the robustness and applicability of our algorithm, it is essential to test it on real-life telecommunication networks. This will help validate its effectiveness in diverse and practical scenarios.

8 Conclusions

By utilizing **two different** solvers, we were able to solve a model that can conclusively determine a network's **connectivity**. We proved their efficacy using **sample networks** provided by DFG and networks generated ourselves. The average runtime for our Integer Programming Model was **0.35 seconds**, successfully solving **100%** of the test cases. Moreover, for the Iterative Greedy Algorithm, we evaluated two heuristics: one that selects the next node based on the minimum edge weight and another that selects the next node randomly. The minimum edge weight heuristic had an average runtime of **0.25 seconds**, excluding test cases 8, 12, and 13, 14, 15 and solved **73%** of the test cases. The random choice heuristic had an average runtime of **0.30 seconds**, excluding test case 13 and solved **93%** of the test cases. For implementation reasons, we recommend DFG Consulting implements the **Iterative Greedy Algorithm**. If the **Iterative Greedy Algorithm** cannot find a solution for a particular graph, then the **Integer Programming** model is guaranteed to find a solution or state that one does not exist.

9 Acknowledgements

We would like to thank Vid Balek, the lead Software Engineer at DFG Consulting, for taking the time to provide invaluable guidance and support throughout our project. His expertise was instrumental in helping us develop our proposed idea, and he was always open to assisting with any challenges we encountered. We would also like to extend our heartfelt gratitude to Professor Sara Billey from the University of Washington-Seattle's Department of Mathematics for her indispensable feedback, insightful ideas, and guidance throughout our project.

10 References

[1] A. Bondy and U.S.R. Murty. *Graph Theory*. Springer, 2008. URL: https://link.springer.com/book/10.1007/978-1-84628-970-5.

[2] *DFG Consulting Website*. URL: https://dfgcon.si/en/.

[3] Peter E Hart. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". In: *IEEE Transactions on Systems Science and Cybernetics* (1968). URL: https://doi.org/10.1109/TSSC.1968.300136.

[4] Bhavana Honavalli et al. *Github Repository*. 2024. URL: https://github.com/Edelwy/dfg-project-math381/.

[5] Bernhard Korte and Vygen Jens. *Combinatorial Optimization: Theory and Algorithms*. 4th. Springer, 2008. URL: https://doi.org/10.1007/978-3-540-71844-4.

[6] Amit Patel. *Introduction to A**. 2024. URL: https://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html.

11 Appendix

11.1 Real Network Example

In Figure 10, we see manholes at the top-right and bottom-left, with houses along the line. This example has been anonymized. The number of cables in the cable bundle can be read from the picture, as well as edges, houses, manholes, and splitters. The triangle symbols represent subscribers, and manholes are labeled 'PJ'. The splitters are represented as small red crosses.

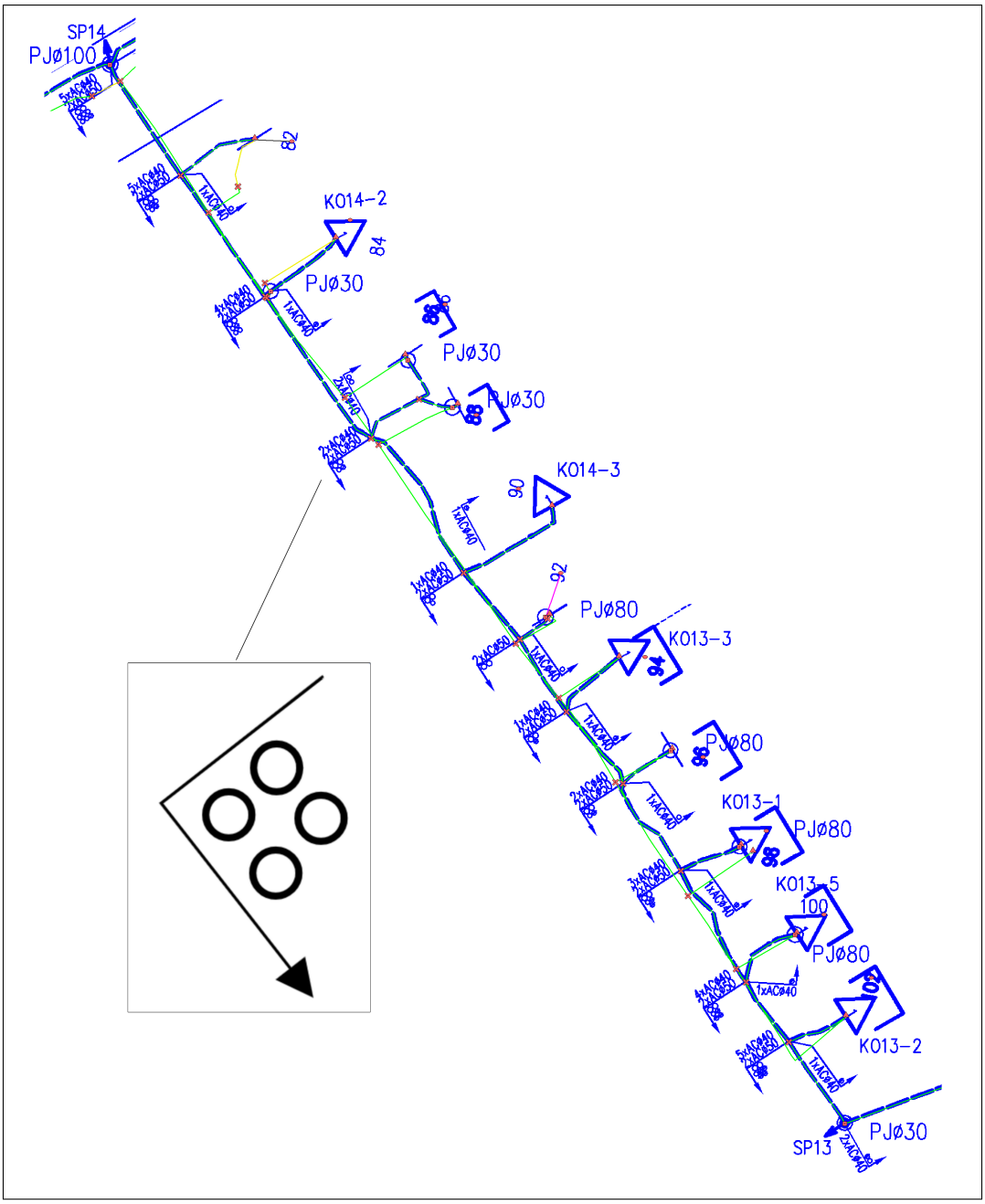

Figure 10: Example of the original data for one network, provided by DFG. The corner symbol (zoomed and in black) dictates the number of cables in a bundle (i.e. 4) that are contained in the connection.

Figures 11, 12, 13 and 14 are pictures taken by DFG on one of their projects.

Figure 11: Example of a manhole, which is sometimes referred to as a shaft. These are represented as square nodes on our theoretical graphs. This picture was taken from one of the streets in Slovenia.

Figure 12: Example of a manhole from underground, with an above-average number of cables. This is where all the bigger cable bundles originate from.

Figure 13: Another example of a manhole, but with a focus on cable bundles. These cables later route to different buildings.

Figure 14: Example of an end-user panel, where cables end. Even though it seems like this might be an apartment or some larger building, this would still be referred to as a house in our theoretical model.

11.2 Implementations

11.2.1 Data Parsing

DFG Consulting provided us with data in Excel files. There are 2 sheets in a single file for a single network. One sheet is for nodes, and the other is for links. The first column of the **nodes** sheet is `str_name`, which labels each node. The second column `ref_name` is the type of node. There are 3 supported types: **PJ** is a manhole, **TS** is a splitter, and **OS** is a house.

The **links** sheet has 4 columns. The first one, `span_name`, is a unique label for an edge. The second, `to_str_name`, is an end node; and the third, `from_str_name`, is the start node (this is the order and layout of the data provided by DFG). The company does not provide the fourth column so the data must be gathered by reading from the corresponding picture provided and is called `weights`, short for 'edge weights'.

We are using a `pandoc` package to read the data, so any of the supported formats can be used.

```
# Function for reading the input.
# The input is in 2 spreadsheets one with links and one with nodes.
def parseInput(filepath):

    # We create a dataframe reading the input in a spreadsheed form.
    nodes_dataframe = pd.read_excel(filepath, sheet_name="nodes").to_numpy()
    edges_dataframe = pd.read_excel(filepath, sheet_name="links").to_numpy()
    house_nodes = [] # Initializing the list of nodes type house.
    manhole_nodes = [] # Initializing the list of nodes type manhole.
    splitter_nodes = [] # Initializing the list of nodes type splitte.
    edges = [] # Initializing the list of edges.

    # Reading the data from the spreadsheed and adding the nodes to
    # their corresponding list, based on the type.
    for row in nodes_dataframe:
        node_type = row[1]
        if node_type == "PJ":
            manhole_nodes.append(row[0])
        if node_type == "TS":
            splitter_nodes.append(row[0])
        if node_type == "OS":
            house_nodes.append(row[0])

    # Adding the edges from the spreadshit to the list.
    for row in edges_dataframe:
        edges.append((row[1], row[2], row[3]))

    # Returning all four lists.
    return house_nodes, manhole_nodes, splitter_nodes, edges
```

Next up, we implemented a function that creates a graph from the data we parsed earlier. Mind that this is an undirected graph. We will modify it into a directed graph later.

```
# Function that creates a graph.
def createGraph(house_nodes, manhole_nodes, splitter_nodes, edges):
    G = nx.Graph() # Creating an empty networkx graph.

    # Adding all of the nodes.
    G.add_nodes_from(house_nodes)
    G.add_nodes_from(manhole_nodes)
    G.add_nodes_from(splitter_nodes)

    # Adding all of the edges.
    for edge in edges:
        G.add_edge(edge[0], edge[1], capacity=edge[2])
    return G
```

11.2.2 Integer Programming

We implemented the IP model using an optimization program called **Gurobi** alongside the Python package **networkx** in order to manipulate graphs. Here is a quick summary of the code, with a lot of comments to make the model implementation easier to read. Snippets of code are removed for improved readability but all of our code is publicly available in the **Jupyter Notebook**[4].

First, we set up the environment and import packages. **networkx** is used for working with graphs and their elements, without having to implement the object structure ourselves. We used it mostly for working with nodes and edges and acquiring their neighbors, etc. The **gurobipy** package is crucial to our model as it is the core of solving the IP problem. It is widely used for linear programming optimization. It provides 2 main algorithms, the barrier, and the simplex method, but these are used for continuous models (real variable values), while we have integer programming (integer variable values). This problem is solved via a series of linear programming relaxations.

```
import gurobipy as gp # Used for the IP model.
import numpy as np # Used for basic math operations.
from gurobipy import GRB # Used for setting the model to integer programing.
import networkx as nx # Used for graph manipulation.
import matplotlib.pyplot as plt # Used for vizualization of graphs.
import pandas as pd # Used for the data parsing.
import copy # We need this when creating paths.
import random # For vizualization.
import matplotlib.colors as mcolors # For vizualization.
import time # For timing the execution.
```

Now we have a few functions using **Gurobi** to define our model. First, we define the **variables**. The algorithm traverses each directed edge, to initialize all possible junction variables, except those with a manhole as the center node. Each variable is stored in a dictionary, whose keys are pairs of edges that create a junction. If two edges share a common node, then the variable is added to our dictionary. We only allow for **integer** values because the number of cables is an integer, therefore the number of connected edges at a junction has to be an integer.

```
# Function that creates the variables.
def createVariables(model, G_directed, manholes): # We have a directed graph.
    x = {} # Initializing an empty dictionary for storing the variables.
    # Iterating through first edges in junctions.
    for start in G_directed.edges():
        # Iterating through last edges in junctions.
        for end in G_directed.edges(start):
            # If these 2 edges have a common middle
            # node we add them to the dictionary.
            if end[1] != start[0] and start[1] == end[0] and start[1] not in manholes:
                x[start, end] = \
                model.addVar(name=f"x[{start[0]} {start[1]} {end[1]}]", \
                             vtype=GRB.INTEGER)
    return x
```

Then we have the constraints. For each `start` edge, the corresponding constraint is the sum of all variables that route to the `end` node. So if `start` is $\boldsymbol{ab} \in \boldsymbol{E}$ then `end` can be any edge $\boldsymbol{bc} \in \boldsymbol{E}$. Furthermore, we must also add the variables where the cable is wired from `end` to `start` edge, meaning $\boldsymbol{cb} \in \boldsymbol{E}$ routing to $\boldsymbol{ba}$. We also ensure that every variable value is possible by adding this constraint to the model here.

```
# Function for creating constraints.
# Creating constraints for the start cable bundles.
# x is the variable vector, G_directed is the directed graph.
def createConstraints(model, G_directed, x, weights):
    # Fixed edge, the constraint of which we are using.
    for start in G_directed.edges():
        constr = 0 # Initialize the constraint.

        # All other possible junction from this fixed edge.
        for end in G_directed.edges(start):
            # Check if the variable is in the dictionary.
            if (start, end) in x:
                var_start = x[start, end]
                constr = constr + var_start # Add to the sum.
                model.addConstr(var_start >= 0) # Implicit constraint.
            # Check if the junction variable for the junction going
            # in the opposite direction is in the dictionary.
            if ((end[1], end[0]),(start[1], start[0])) in x:
                var_end = x[(end[1], end[0]),(start[1], start[0])]
                constr = constr + var_end # Add to the sum.
                model.addConstr(var_end >= 0) # Implicit constraint.

        # The sum of variables on the fixed edge must be equal to the
        # weight on that edge.
        constr = constr == weights[start]

        # In the case where the key is not in the dictionary,
        # The constraint will be 0 which equals false, we dont want that.
        if constr != False:
            model.addConstr(constr)
```

We then add the **objective function** to the model, to minimize the number of paths that do not connect to a manhole. We simply iterate through all variables and add the ones that route from a subscriber to a subscriber through any node. Furthermore, we will prune the solutions with paths not connected to a manhole later, but this ensures that the solutions with a minimized number of *house-node-house* junctions are at the top of the solution pool. This is useful as a time optimization because we will need to prune fewer solutions before we find one that suits us.

```
# Setting the objective function.
def createObjective(model, house_nodes, x):
    min = 0 # Initialize the objective function.
    for (start, end), var in x.items(): # Iterate through variables.
        # If the start node of the junction and the end node of the junction
        # are both houses, then we add it to the sum.
        if start[0] in house_nodes and end[1] in house_nodes:
            min += var
    model.setObjective(min, sense = gp.GRB.MINIMIZE)
```

Finally, we evaluate our model and save a number of solutions equal to the `pool_solutions` value. We simply iterate through all variables and add the ones that route from a subscriber to a subscriber through any node.

```
# Finding the solution to our model.
def evaluateModel(model, solution_name, pool_solutions):
    model.setParam("OutputFlag", False) # We don't want an output printed.
    model.setParam(GRB.Param.PoolSearchMode, 2) # Searching for all optimal solutions.
    model.setParam(GRB.Param.PoolSolutions, pool_solutions) # Number of solutions saved.
    model.optimize() # Actually optimizing the model and finding the solution.
```

The next segment is for **creating paths from junction variables** in the solution. We need this to get the bigger picture. The value of a variable in a junction represents the number of cables connecting there, but this does not mean that we know how the paths are constructed yet. We have to stitch together the edges in these junctions to obtain the paths.

We have a `varsInPath` set that stores all the variables that have been stitched into a path already. This prevents duplicates. We have a fixed variable, and we iterate over all the other variables over and over again stitching them together until no more changes occur. Then we repeat the process for any other variable that is not in the `varsInPath` set.

If a variable is stitched it does not go directly to the `varsInPath` set. Its value is decreased by one because that means one of the connections was stitched, but there could be more, so we have to keep that option. A variable is added to the set when its value reaches 0, meaning no more connections go through that junction.

```
# Helper function that updates the variable when it is added to the path.
def updateVariable(vars, varKey, varsInPaths):
    vars[varKey] = vars[varKey] - 1 # Decrease number of connections in junction.
    if vars[varKey] == 0: # If there is no more options, we cannot use this anymore.
        varsInPaths.add(varKey)
```

```
# A function that changes junction variables into paths.
def junctionsToPaths(model, G, manhole_nodes, weights):
    # This function changes the name from IP variable names, to
    # just nodes in the junction so it is more readable in the path.
    # Name goes from "x[A B C]"" to "A B C".
    vars = copy.deepcopy(modifyVariableName(model))
    varsInPaths = set() # So we don't create paths from variables we already added.
    paths = [] # We will save paths created here.
    ixVar = 0
    while varsInPaths != set(vars):
        firstVarKey = list(vars.keys())[ixVar]

        # Check if the fixed variable is already in some path.
        if firstVarKey not in varsInPaths:
            firstNodes = firstVarKey.split(" ") # First three nodes of the path.
            path = [firstNodes[0], firstNodes[1], firstNodes[2]] # Add them.
            updateVariable(vars, firstVarKey, varsInPaths)

            # We are going through the variables until there are changes made,
            # to the path. Once no new path is added, we can be sure the path is done.
            change = True
            while change:
                # We check for a change every iteration.
                change = False
                for varKey in vars.keys():
                    if(varKey == firstVarKey):
                        continue
                    nodes = varKey.split(" ") # Getting all 3 nodes out of the junction.
                    # If this key is not used up already we can check whether,
                    # it matches the path we are building.
                    if varKey not in varsInPaths:
                        if addNodeToPath(path, nodes):
                            updateVariable(vars, varKey, varsInPaths)
                            change = True
            paths.append(path)
        # We move onto the next variable modulo the length of our list,
        # because we are looping through it.
        ixVar = (ixVar + 1) % len(vars)
    addOneEdgePaths(G, paths, manhole_nodes, weights)
    return paths
```

We stitch a junction to a path in **four cases**. If the junction has the same direction as the path we are creating, we stitch it if either the first edge in the junction matches the last in the path or the last edge in the junction matches the first one in the path. We add the edge that is not yet in the path (the one that is not matching). The second case is when the direction is opposite. Then either the last junction edge matches the last path edge or the first junction edge matches the first path edge. We have to continue this until all variables cannot connect anywhere because their value is **0**.

```
# Helper function to determine if junction is in path.
# We have for different options.
# The last clause in the boolean expression is to work with cycles.
def addNodeToPath(path, nodes):
    # We add to the front of the path and the direction
    # of the junction is the same as the path.
    if path[0] == nodes[1] and path[1] == nodes[2]:
        if nodes[0] != path[-2]:
            path.insert(0, nodes[0])
        return True
    # We add to the front of the path and the direction
    # of the junction is the opposite as the path.
    elif path[0] == nodes[1] and path[1] == nodes[0]:
        if nodes[2] != path[-2]:
            path.insert(0, nodes[2])
        return True
    # We add to the end ont of the path and the direction
    # of the junction is the same as the path.
    elif path[-2] == nodes[0] and path[-1] == nodes[1]:
        if nodes[2] != path[1]:
            path.append(nodes[2])
        return True
    # We add to the end ont of the path and the direction
    # of the junction is the opposite as the path.
    elif path[-2] == nodes[2] and path[-1] == nodes[1]:
        if nodes[0] != path[1]:
            path.append(nodes[0])
        return True
    else:
        return False
```

The next function deals with edge cases when there are no junction variables. There can be *manhole-node* paths consisting of just one edge that we need to take into account.

```
# Adding paths consisting of just 1 edge.
def addOneEdgePaths(G, paths, manhole_nodes, weights):
    # Going throught every manhole node and checking its neighbors.
    for node in manhole_nodes:
        for neighbor in G.neighbors(node):
            # If the list of neighbors of the neighbor is just one element,
            # it means that the neighbor is a leaf node, so we have to add
            # this one edge path to the paths list.
            next_neighbors = list(G.neighbors(neighbor))
            if len(next_neighbors) == 1:
                for i in range(weights[node, neighbor]):
                    paths.append([node, neighbor])
            # If we have 2 manholes connecting to eachother we also must add them.
            # There is no junctions to connect here.
            elif neighbor in manhole_nodes and [node, neighbor] not in paths:
                for i in range(weights[neighbor, node]):
                    paths.append([neighbor, node])
```

One thing we have to be careful about when stitching the path is cycles. If we encounter a cycle we must decrease the junction variable value, because this means we connected the cycle with this being the final junction connection, but we should not add the edge to the path because we don't want duplicates.

Finally, the code that does it all. Before all we had were just function definitions, this is the actual **script we are executing**. First, we parse the input, then create a graph and make it bidirectional. We create a weights vector, and a variable vector, and add the constraints and the objective function. After all this, the model is evaluated and paths are created from the solution. If we are not allowing paths with no manholes, then solutions are checked until the first satisfactory solution is found. If there is none we return a solution, but warn the user that it is not optimal.

```
# First we parse the input file provided in the arguments.
# From that we get nodes of type house, splitter and manhole.
# We also get edges and edge weights.
house_nodes, manhole_nodes, splitter_nodes, edges = parseInput(input_file)

# Then we create an undirected graph using these nodes and edges.
G = createGraph(house_nodes, manhole_nodes, splitter_nodes, edges)
G_directed = G.to_directed() # We also create a directed version of the graph.
# We extract the weights vector for all edges (directed).
weights = nx.get_edge_attributes(G_directed, 'capacity')
model = gp.Model("DFG") # Initialiting the model.

x = createVariables(model, G_directed, manhole_nodes) # Creating the variables vector.
# Adding constraints to the model.
createConstraints(model, G_directed, x, weights)
createObjective(model, house_nodes, x) # Adding the objective function to the model.
evaluateModel(model, "DFG", pool_solutions = pool_solutions) # Evaluating the solution.

if model.SolCount == 0:
    print("No solution was found for this model.")
else:
    paths = junctionsToPaths(model, G, manhole_nodes, weights) # Getting the paths.
    # If we don't want paths with house-node-house paths, we iterate through
    # the solutions and find the first one that satisfies this constraint.
    # If we dont find any we give a warning, that perhaps we should either save more
    # solutions or lose the constraint.
    solution_index = 0
    if not checkPaths(paths) and not allow_all_paths:
        for i in range(1, model.SolCount):
            model.setParam(GRB.Param.SolutionNumber, i)
            paths = junctionsToPaths(model, G, manhole_nodes, weights)
            path_check = checkPaths(paths)
            if path_check:
                solution_index = i
                break
            if not path_check and i == model.SolCount - 1:
                print("No solution with a manhole in every path found. \
                    Printing last solution saved. If this is not satisfactory \
                    consider increasing the number of solutions saved.")

    #Printing out the report.
    writeReport(model, paths, solution_index, execution_time, junction_variables)
```

11.2.3 Iterative Greedy Algorithm

This algorithm begins by retrieving the **sum of all edges** connected to every manhole in the network, where the sum of edges represents the number of cables connected between two nodes. Next, it chooses a start node, at random, from the list of manholes. Then, the algorithm identifies the best edge to follow based on an arbitrarily chosen heuristic. For example, a chosen heuristic might be selecting edges based on the greatest available weight. It then follows a chosen edge to the corresponding connected node and decrements its edge weight by **1**. If there is no "best" node (e.g. leaf nodes), the `get_best_node()` function returns $\mathbf{-1}$, and the path terminates there. This process is **repeated** until the sum of edge weights for every manhole is exactly **0**; a resulting **0** implies that the algorithm has connected every cable and node in the network.

The first step is to identify all **valid manhole nodes** as we will be using them as seed nodes. We do this by summing all edge weights for each respective manhole. Manholes with **0** total weight are disregarded.

```
# this method gets the sum edge weights of each manhole
def sum_manhole_weights(G_copy, manholes):
    manhole_weights = {}
    for manhole in manholes: # loop through all manholes
    # sums the edge weight for a manhole
        weight = sum([G_copy[manhole][x]["weight"] for x in G_copy.adj[manhole]])
        if weight > 0:     # remove any zero-weight manholes
            manhole_weights[manhole] = weight
    return manhole_weights
```

The next step is to define a method to **rank edges**. The framework for this method is very simple, just have a comparison method for each edge and add them to a priority queue. However, we cannot have this method deterministic as it is very easy for the Iterative Greedy Algorithm to get stuck and be unable to find a solution. We decided to have the seed nodes chosen in random order, but a random heuristic may also work.

```
# this method iterates through a graph, removing weight as it goes
# it utilizes a heuristic to guide its path
def remove_weight_greedy(G_copy, manholes, random_heuristic):
    disjoint_sets = []
    manhole_weights = sum_manhole_weights(G_copy, manholes)
    # repeat while manholes have edge weights
    while sum(manhole_weights.values()) > 0:
       # begin path with random manhole
        path = [random.choice(list(manhole_weights.keys()))]
        # find the highest-ranked node and remove one from the edge weight
        while len(G_copy.adj[path[len(path) - 1]]) > 0:
            bestNode = get_best_node(G_copy, path, random_heuristic)
            # stop path if there are no best nodes (e.g. at a leaf node)
            if bestNode == -1:
                break
            # add the best node to the current path
            path.append(bestNode)
        # add the path to the set
        disjoint_sets.append(path)
        manhole_weights = sum_manhole_weights(G_copy, manholes)
    return disjoint_sets
```

To seed randomly, we set the first node in our path to a randomly chosen manhole. We then follow the edge given by `get_best_node()` and append it to our path. We repeat this process until there are no more edges to follow e.i. `bestNode` is $-\mathbf{1}$. We find these paths until all edges out of manhole nodes are exhausted, therefore the total weight is **0**. We also have a boolean variable to represent if the user selected the **random heuristic** option. Should they choose this, we will choose a random neighbor edge every time rather than the lowest-weight edge.

The `get_best_node()` method is designed to take a given node and search all of its neighbors for the **highest-ranked node**. This ranking comes from the `heuristic()` method and is easily modifiable. We add the output of the `heuristic()` to the priority queue as its rank which will return the highest rank. We simply take the highest value edge, subtract 1 from its edge weight, and return its corresponding node. One downside to this approach is that some graphs' solutions can never be found by only choosing the lowest-weight edge. The random heuristic solves this.

The **random heuristic** ranks the neighbor nodes randomly, thus choosing a random node. This heuristic is preferable in situations where a solution must be found as this random heuristic will not get stuck in a situation where no solution can ever be found.

An important thing to note is that this method also disallows backtracking onto nodes already visited. That is, any node added to the priority queue cannot be in the current path. This is required for non-directed graphs.

```
# this method identifies the best node by using a given heuristic
def get_best_node(G_copy, path, random_heuristic):
    start = path[len(path) - 1]
    adj = G_copy.adj[start]
    queue = []

    # Add all nodes to a priority queue ranked by the heuristic method
    for end in adj.keys():
        if G_copy[start][end]["weight"] > 0 and end not in path:
            # if the user chooses the random heuristic, choose the best node at random
            if random_heuristic:
               heapq.heappush(queue, (random.random(), end))
            # otherwise, use the lowest-weight heuristic
            else:
               heapq.heappush(queue, (heuristic(G_copy, start, end), end))

    # return -1 to indicate no neighbor nodes
    if len(queue) == 0:
        return -1

    # find the highest ranked node and remove 1 from the edge to that node
    best_node = queue.pop()[1]
    G_copy[start][best_node]["weight"] = G_copy[start][best_node]["weight"] - 1
    return best_node
```

Finally, we must apply this Iterative Greedy Algorithm until a **valid solution** is found. Unfortunately, every iteration must create a deep copy of the `networkx` graph. This is an unavoidable issue with using this package. It would be much better to use sparse adjacency matrices or something similar.

Another important aspect of this method is the ability to restrict the number of iterations we run. Ultimately, if the iterations required are too computationally great, the IP solution may be more desirable. However, this is only the case if the graph has very few solutions.

```
# this method repeats the greedy algorithm until it finds a valid solution
# throws errors if no manholes are passed in, or if there are isolated nodes
# prints a message and returns an empty list if a solution could not be found
# in the # of iterations
# num_iterations defaults to -1, running until it stops
def find_disjoint_set(G, manholes, splitters, random_heuristic, num_iterations=-1):
    # throw error if no manholes are present
    if len(manholes) == 0:
        raise ValueError("No manholes passed in")

    # throw error if there are disconnected nodes
    if len(list(nx.isolates(G))) != 0:
        raise ValueError("Isolated nodes in graph")

    # need to copy the graph to avoid pass-by-reference errors
    G_copy = copy.deepcopy(G)
    sets = remove_weight_greedy(G_copy, manholes, random_heuristic)

    # if -1 is chosen, repeat the greedy algorithm until a valid solution is found
    if num_iterations == -1:
        while not is_valid_disjoint(G_copy, sets, splitters):
            G_copy = copy.deepcopy(G)
            sets = remove_weight_greedy(G_copy, manholes, random_heuristic)
    # if a number is chosen, iterate the greedy algorithm that many times
    else:
        for i in range(num_iterations - 1):
            G_copy = copy.deepcopy(G)
            sets = remove_weight_greedy(G_copy, manholes, random_heuristic)

            if is_valid_disjoint(G_copy, sets, splitters):
                break

    # print a message if valid solution is not found
    if not is_valid_disjoint(G_copy, sets, splitters):
        print("Could not find valid solution")
        return []

    return sets
```

```
# this method evaluates a disjoint set for validity:
# - paths cannot end on a splitter
# - all edges must be fully used
# easy to add additional conditions
def is_valid_disjoint(G_copy, disjoint_sets, splitters):
    if G_copy.size(weight="weight") > 0:
        return False

    for path in disjoint_sets:
        if path[-1] in splitters:
            return False
    return True
```

The way we determine if a given disjoint set is valid is through the `is_valid_disjoint()` method. We verify if all graph edge weights have been reduced to **0** and if no paths end in splitters.

It would be beneficial to add something to the heuristic that attempts to avoid ending paths in splitters to decrease the number of iterations required. Our Greedy approach also utilizes `networkx` heavily. In the future, it would be wise to use custom graph representations, as `networkx` severely limits the algorithm's performance.

11.3 Test Cases

Here are the **15 test cases** we used for our model verification, excluding the first one already described in Solution For Theoretical Example. We present one of the solutions, the visualizations can be found below.

Test Case 2: **Figure 16**

```
0: M0, S1, S2, S3, M1     |  2: M0, S1, S2, S3, M1     |  4: M1, S3, S2, S1, M0
1: M1, S3, S2, S1, M0     |  3: M0, S1, S2, S3, M1
```

Test Case 3: **Figure 18**

```
0: M0, H2, H1, H0, M0     |  1: M0, H0, H1, H2, M0
```

Test Case 4: **Figure 20**

```
0: M0, H0, H1, M0         |  2: M0, H0, H1, M0
1: M0, H1, H0, M0         |  3: M0, H1, H0, M0
```

Test Case 5: **Figure 22**

```
0: M0, H0, H3             |  2: M0, H2, H5            |  4: M0, H1, H4
1: M0, H1, H4             |  3: M0, H0, H3
```

Test Case 6: **Figure 24**

```
 0: M0, H0                |  2: M0, H1                |  4: M0, H2
 1: M0, H0                |  3: M0, H1
```

Test Case 7: **Figure 26**

```
0: M1, S2, H2, M2         |  3: M1, S2, H2, M2        |  6: M3, M2
1: M2, S1, H4             |  4: M3, H3, H1            |  7: M3, M2
2: M3, H3, H1             |  5: M3, M2                |  8: M3, H5
```

Test Case 8: **Figure 28**

```
 0: H4, S3, S2, M2        |  2: H7, S1, M1           |  4: M1, S1, S2, S3, H3
 1: H6, S3, S2, M3        |  3: M1, S1, H1           |  5: M3, S2, S3, H5
                                                     |  6: M3, H2
```

Test Case 9: **Figure 30**

```
 0: H1, S1, M1            |  3: H5, S2, S1, M1      |  6: H8, S4, S3, S2, M2
 1: H2, S1, M1            |  4: H6, S4, M3          |  7: M2, S2, S3, H4
 2: H3, S3, S2, M2        |  5: H7, S3, S2, M2      |  8: M1, S1, S2, M2
```

Test Case 10: **Figure 32**

```
0: H2, S1, M1             |  3: H6, S4, M3          |  6: M2, S2, S3, H7
1: H3, S3, S2, M2         |  4: M1, S1, H1          |  7: M1, S1, S2, S3, S4, H8
2: H5, S2, M2             |  5: M1, S1, S2, S3, H4  |  8: H5, S2, M2
                                                    |  9: H5, S2, M2
```

Test Case 11: **Figure 34**

```
0: H2, S0, H1, H0, M0     |  1: H2, H0, M0
```

Test Case 12: **Figure 36**

```
0: M0, H0, H1, S0, M0     |  2: M1, H3, S1, H6      |  4: H5, S1, H3, M1
1: H2, S0, M0             |  3: H4, S1, H3, M1
```

Test Case 13: **Figure 38**

```
 0: H1, S0, H0            |  2: H2, S0, M0          |  3: M0, S0, H0
 1: H1, S0, H2
```

Test Case 14: **Figure 40**

```
 0: M1, S5, H2            |  8: M3, S4, S2, S1, M1  | 16: M1, H1
 1: M4, S3, M2            |  9: M4, S3, M2          | 17: M2, H6
 2: M4, S3, S2, S1, M1    | 10: M4, S3, S2, S1, M1  | 18: M2, H7
 3: M1, S1, S2, S3, M4    | 11: M4, S3, M2          | 19: M3, H3
 4: M1, S1, S2, S4, H9    | 12: M4, S3, M2          | 20: M3, H3
 5: M1, S1, S2, S4, H8    | 13: M4, S3, M2          | 21: M3, H4
 6: M1, S1, S2, S4, M3    | 14: M1, H10             | 22: M3, H4
 7: M4, S3, S2, S1, M1    | 15: M1, H10             | 23: M3, H5
```

Test Case 15: **Figure 42**

```
0: M1, S7, H3             | 9: M2, S3, S4, S5, M8   | 17: M1, H2
1: M6, S4, S3, S2, S1, M1 | 10: M6, S4, S3, M2      | 18: M1, H1
2: M1, S1, S2, S6, H19    | 11: M6, S4, S3, M5      | 19: M2, H11
3: M1, S1, S2, S6, H10    | 12: M4, S6, S2, S1, M1  | 20: M2, H12
4: M1, S1, S2, S6, M4     | 13: H14, S3, S2, S1, M1 | 21: M2, H13
5: M1, S1, S2, S3, H14    | 14: M2, S3, S4, M7      | 22: M3, H4
6: H14, S3, S2, S1, M1    | 15: M7, S4, S3, M5      | 23: M3, H4
7: M2, S3, S4, M6         | 16: M5, S3, S4, M7      | 24: M3, H5
8: M2, S3, S4, M7                                   | 25: M3, H5
```

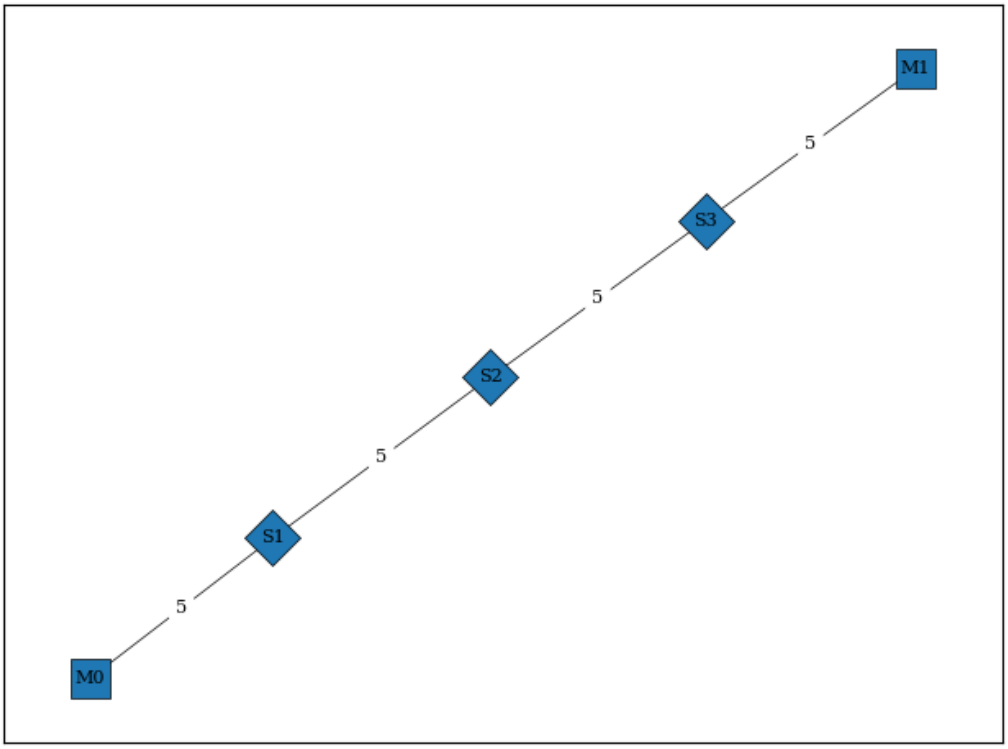

Figure 15: Test case 2 input.

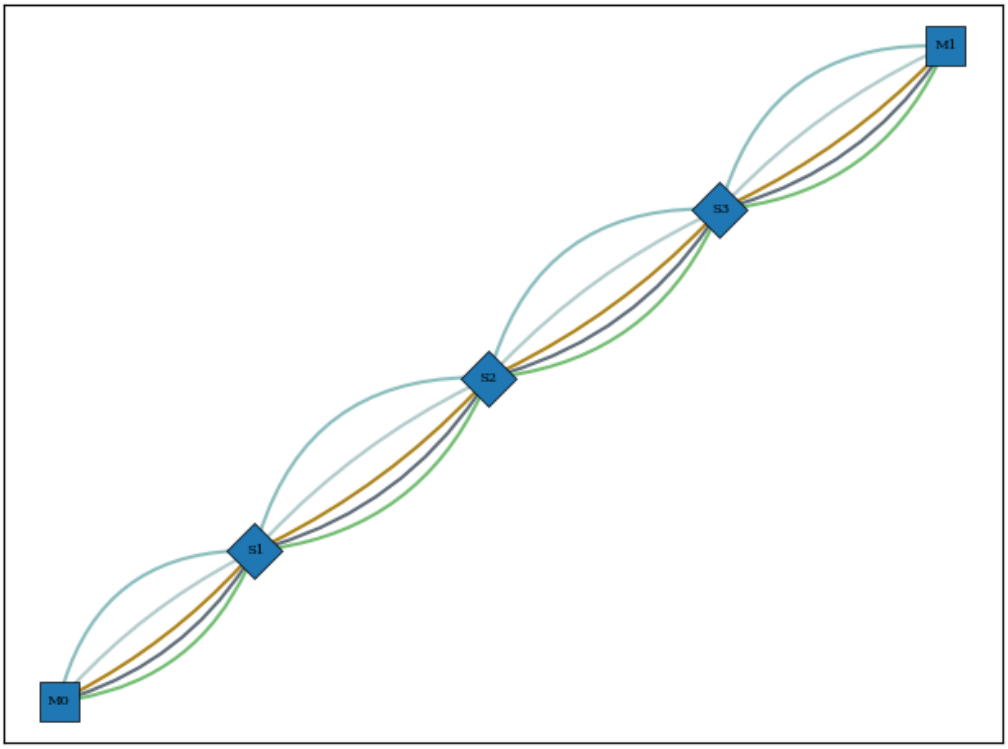

Figure 16: Test case 2 output.

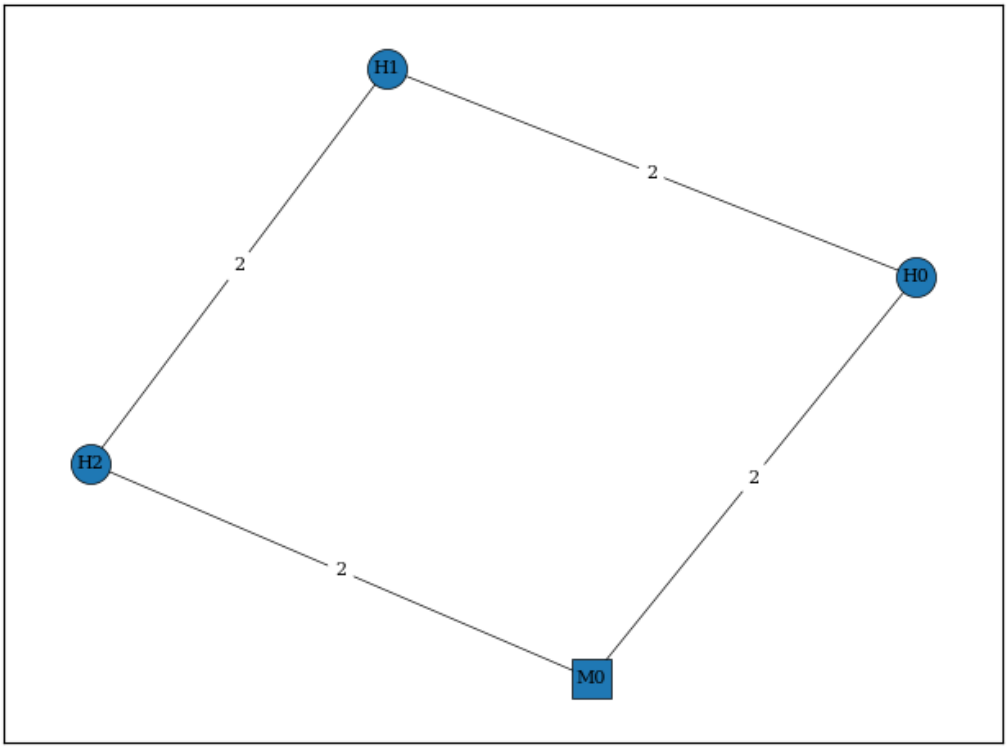

Figure 17: Test case 3 input.

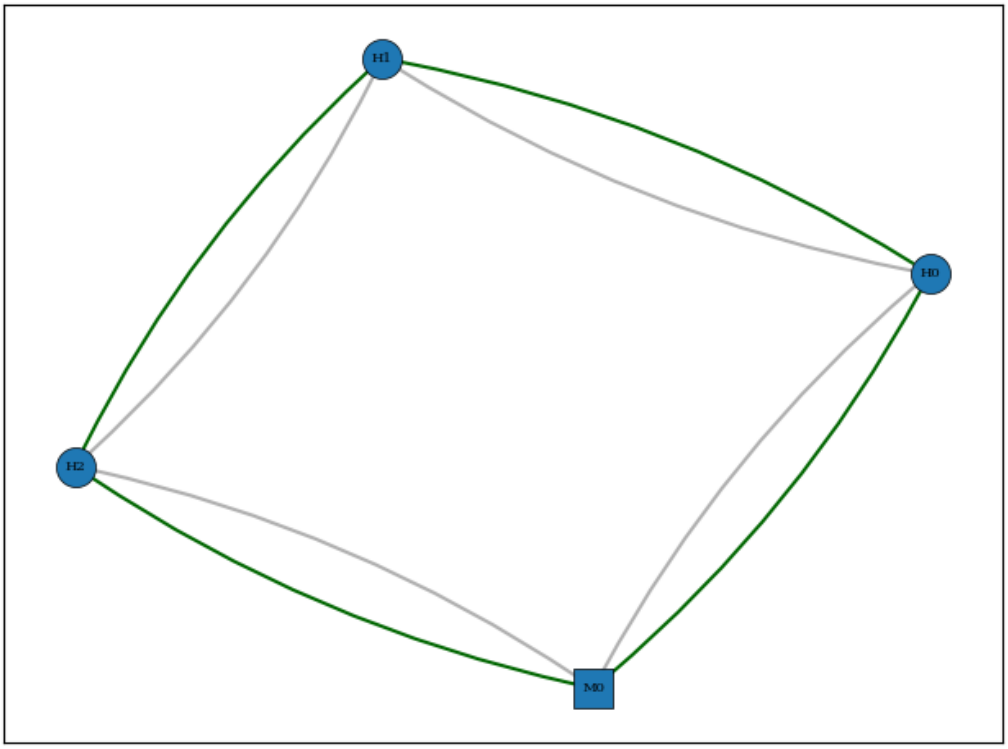

Figure 18: Test case 3 output.

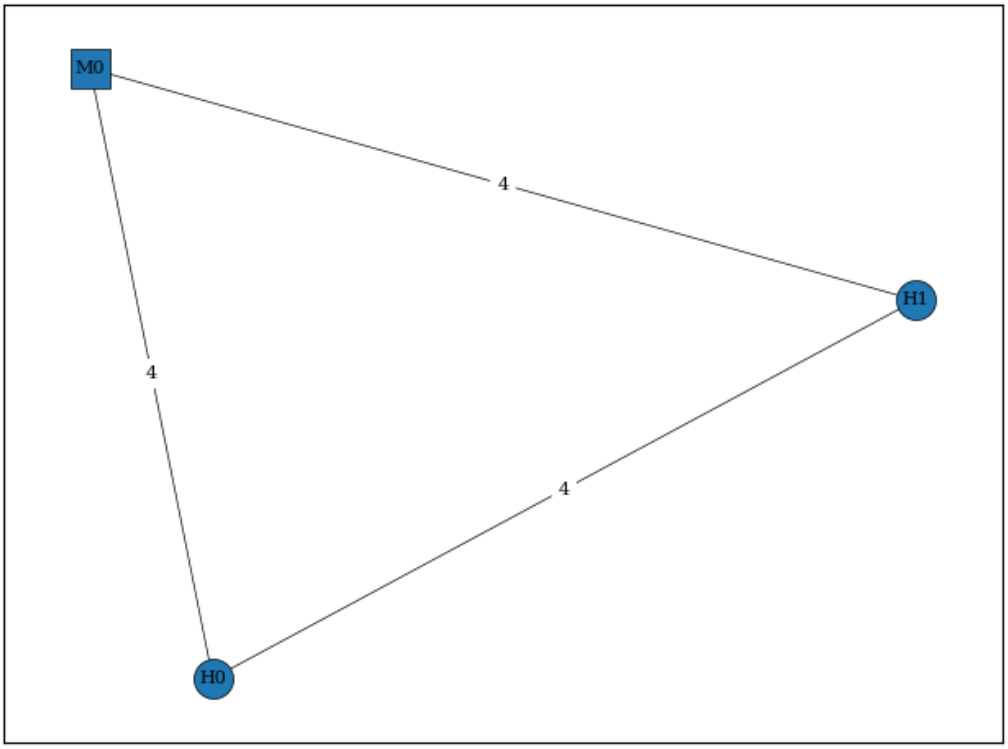

Figure 19: Test case 4 input.

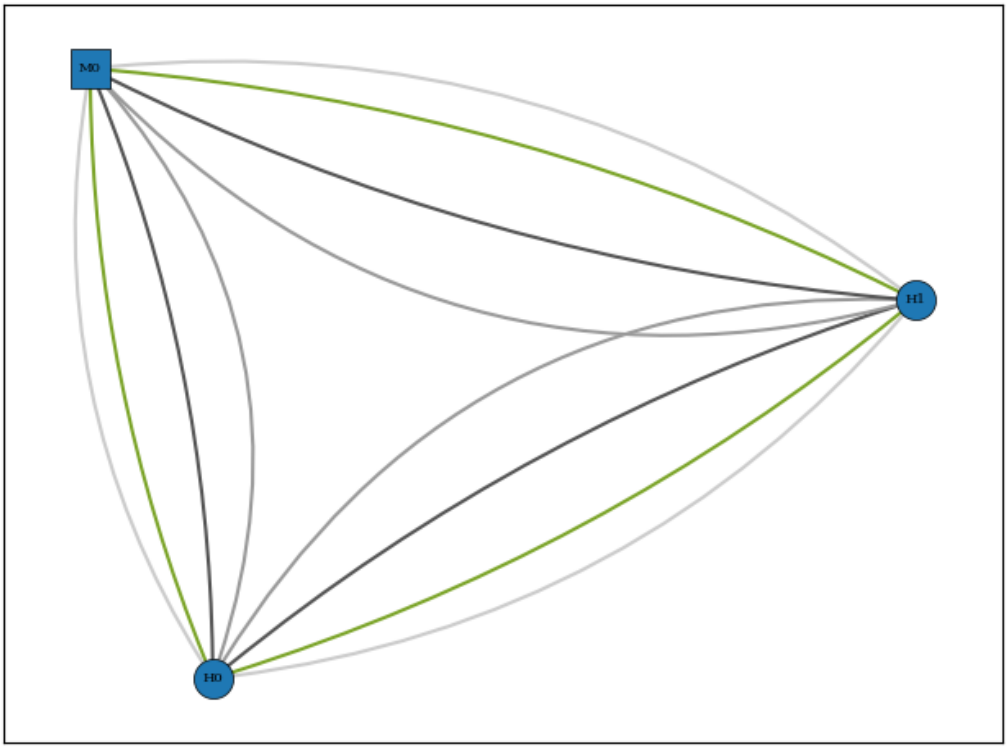

Figure 20: Test case 4 output.

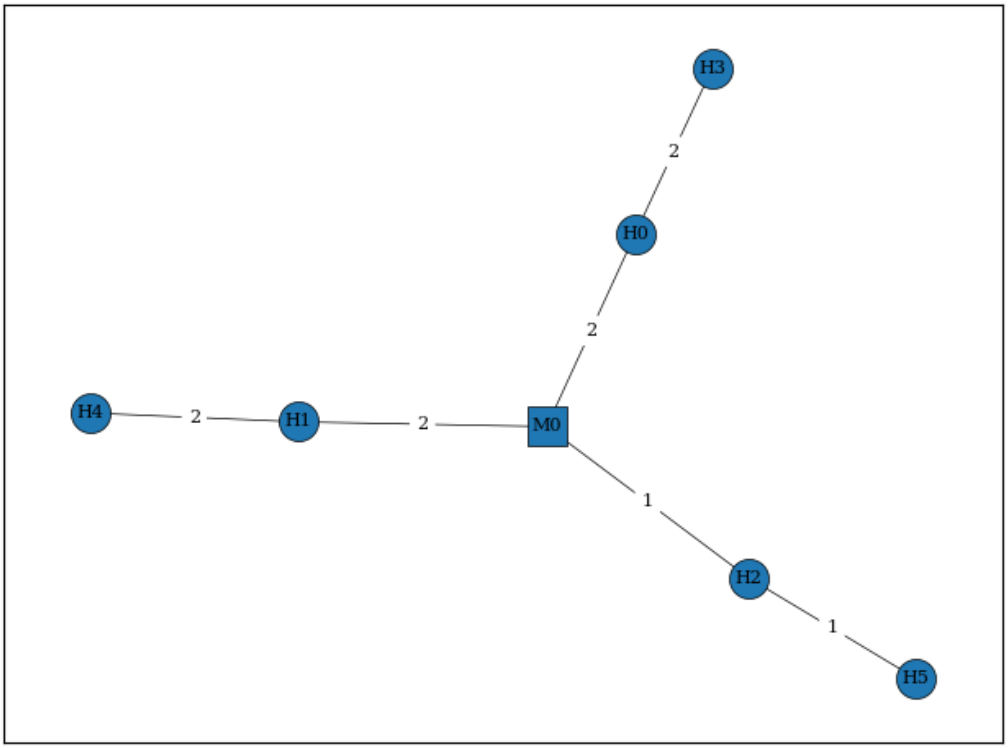

Figure 21: Test case 5 input.

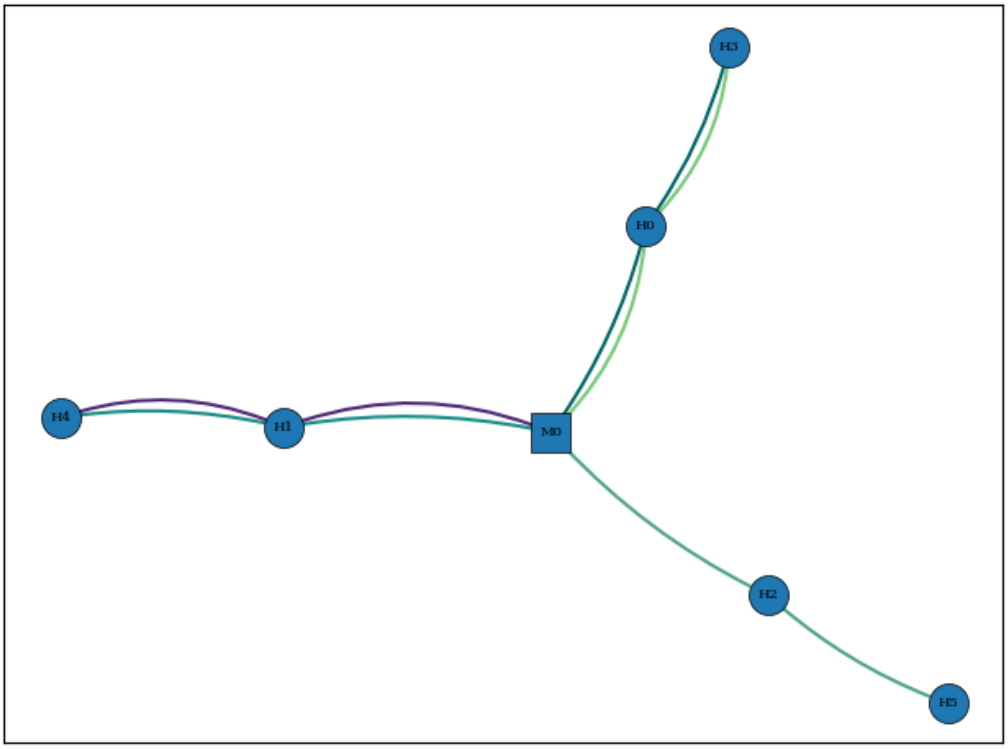

Figure 22: Test case 5 output.

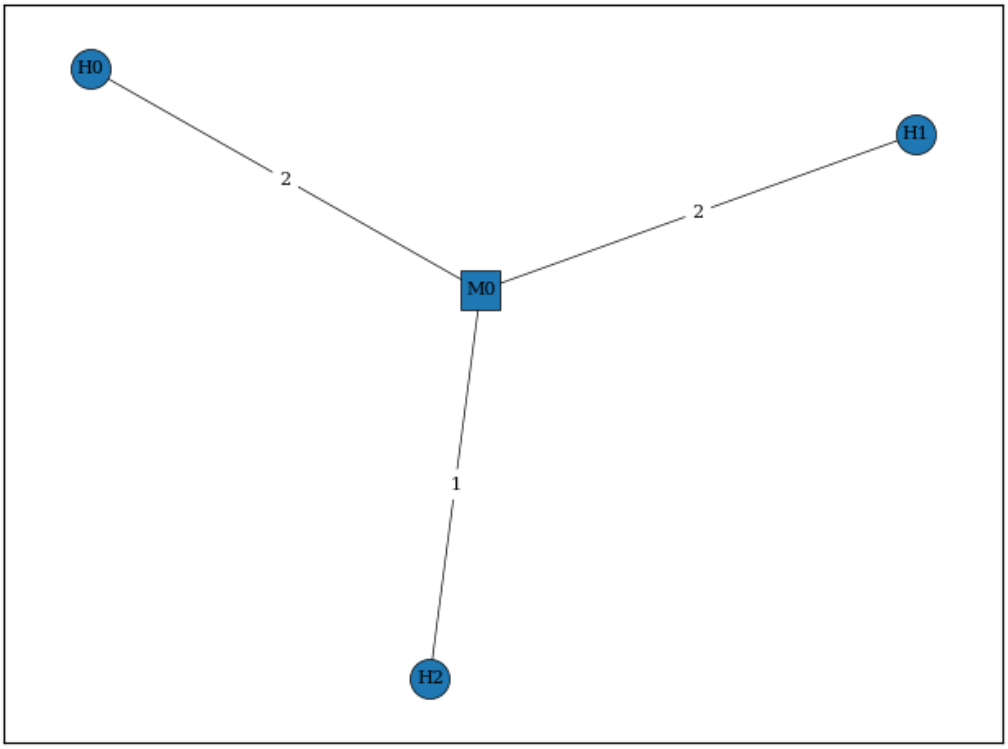

Figure 23: Test case 6 input.

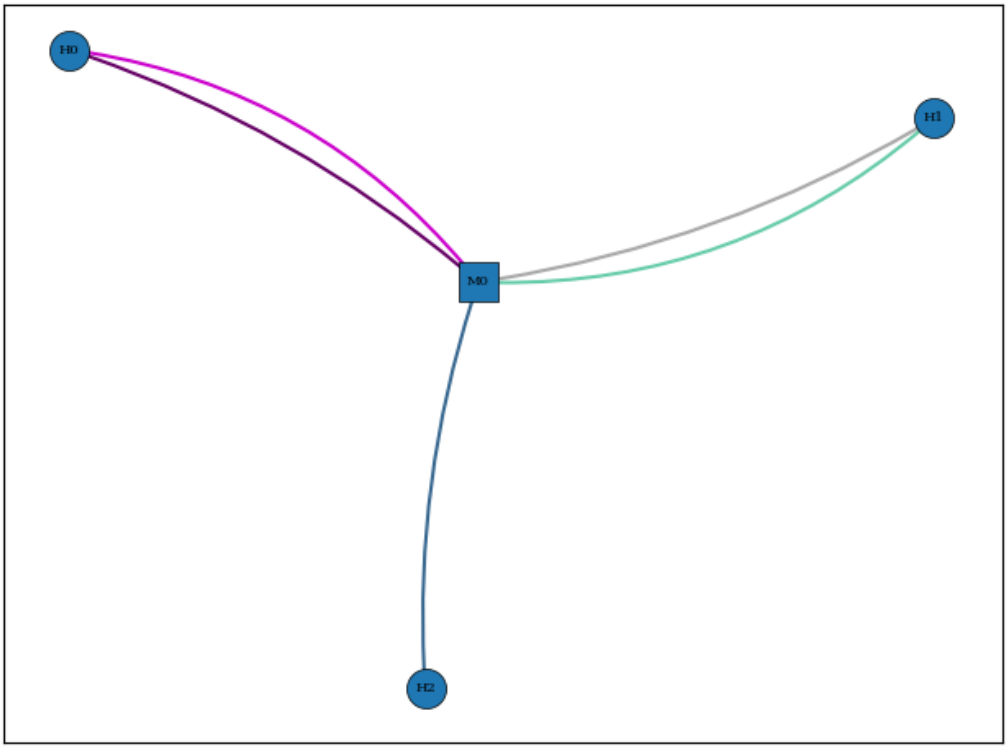

Figure 24: Test case 6 output.

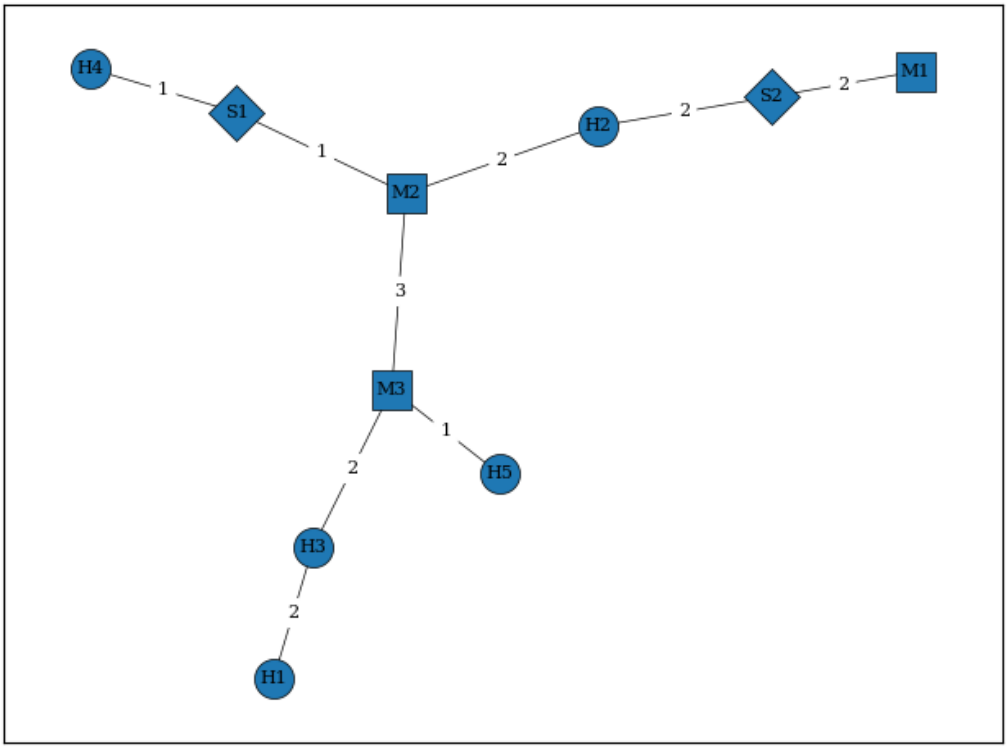

Figure 25: Test case 7 input.

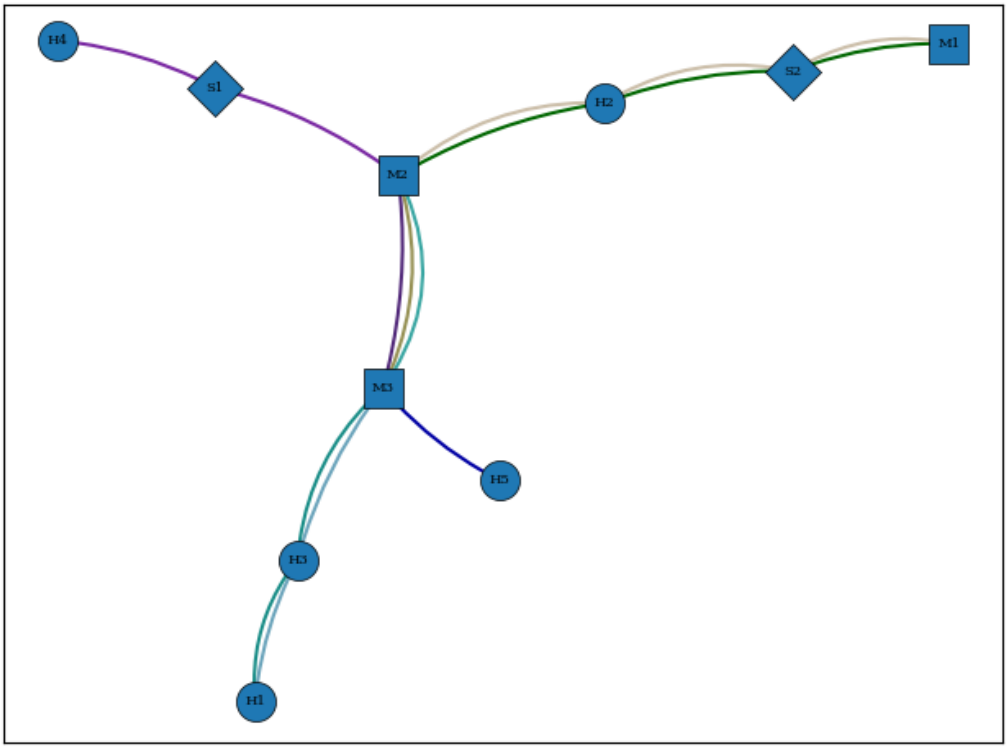

Figure 26: Test case 7 output.

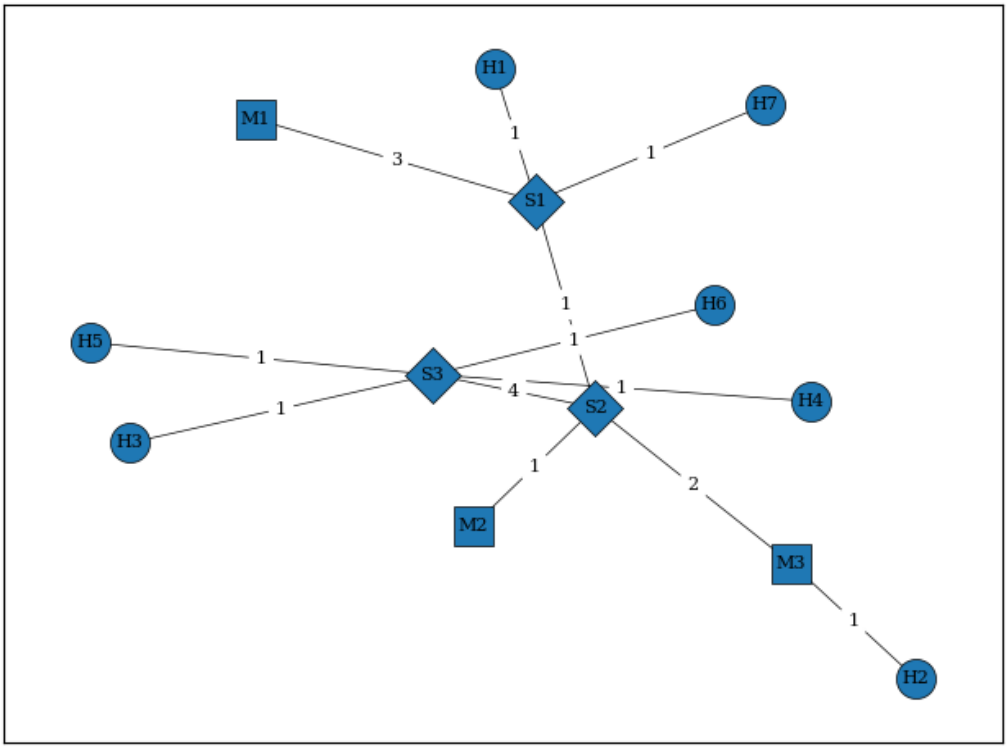

Figure 27: Test case 8 input.

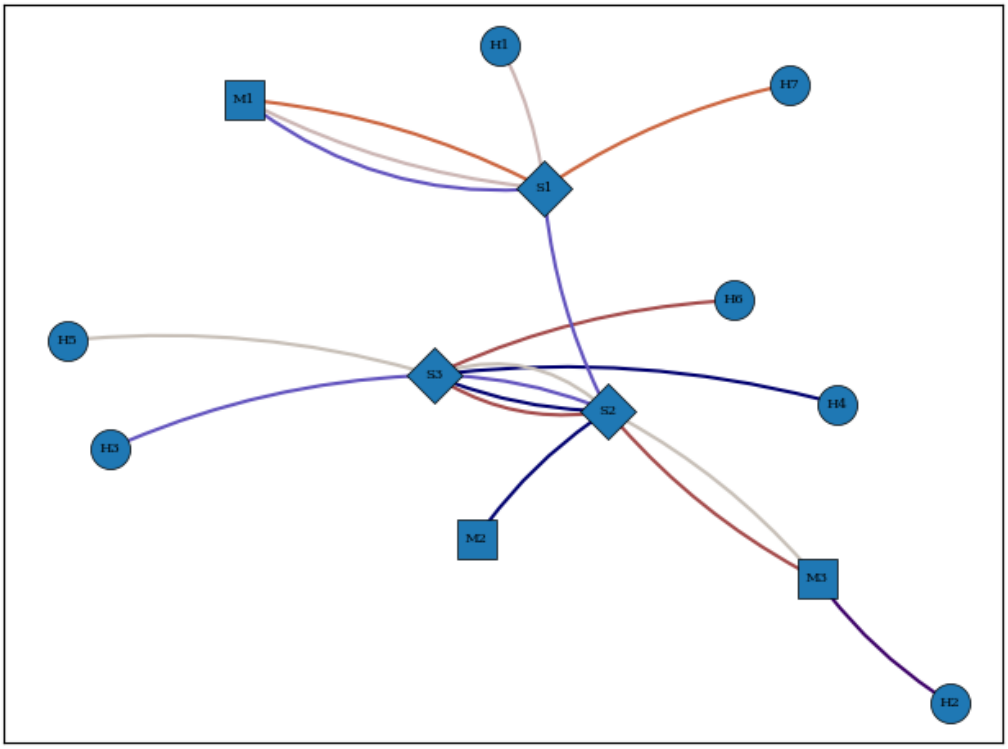

Figure 28: Test case 8 output.

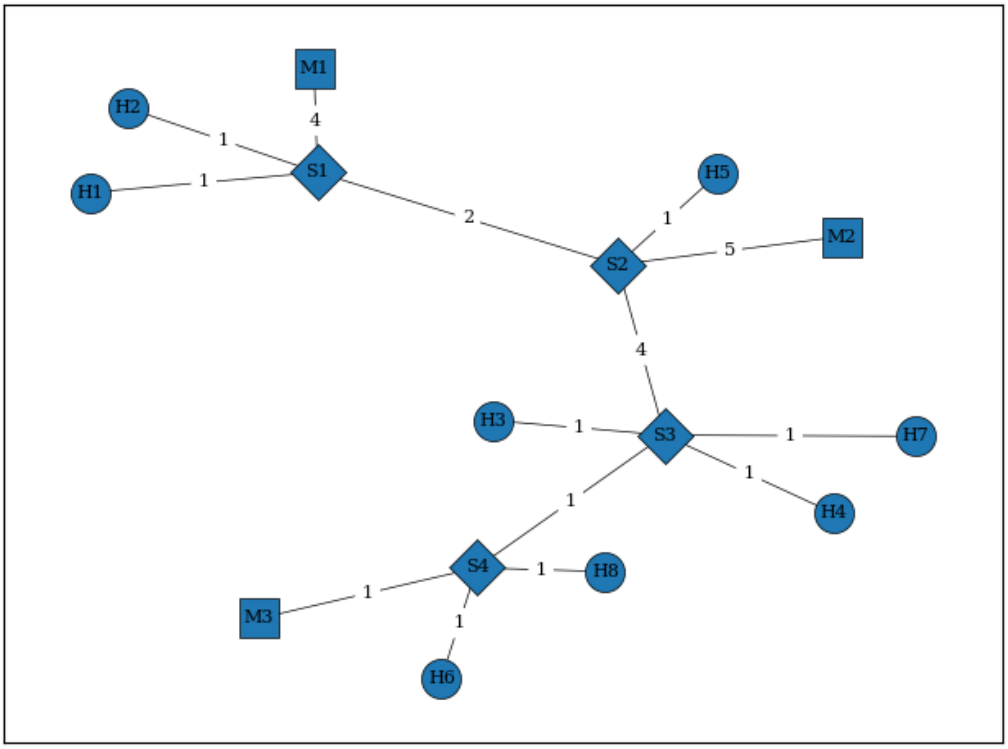

Figure 29: Test case 9 input.

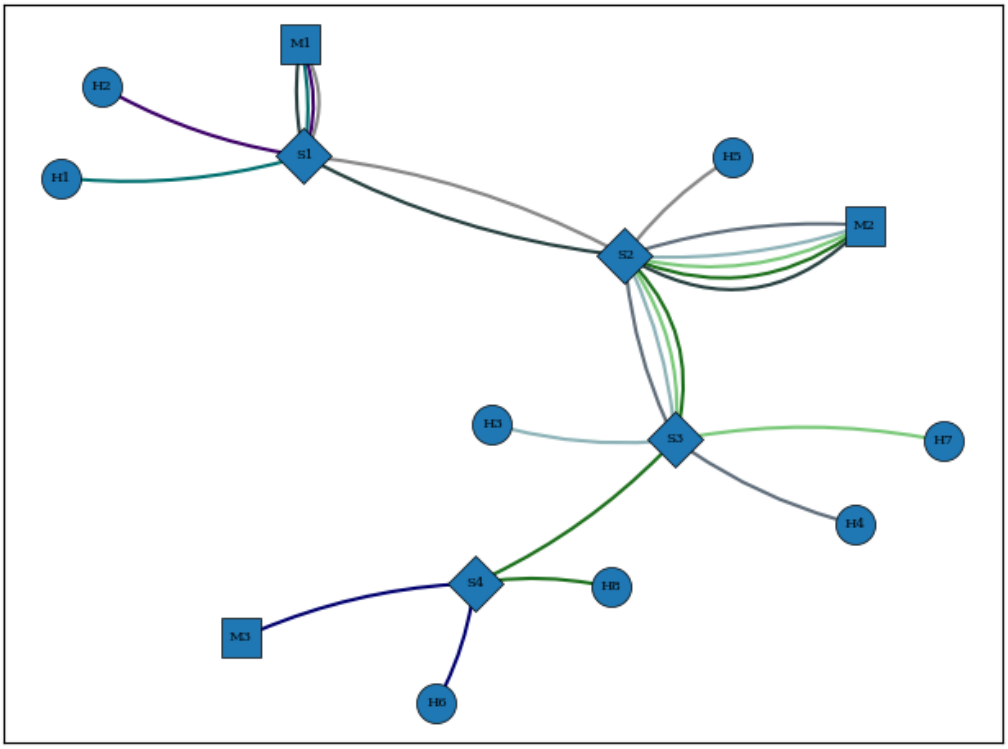

Figure 30: Test case 9 output.

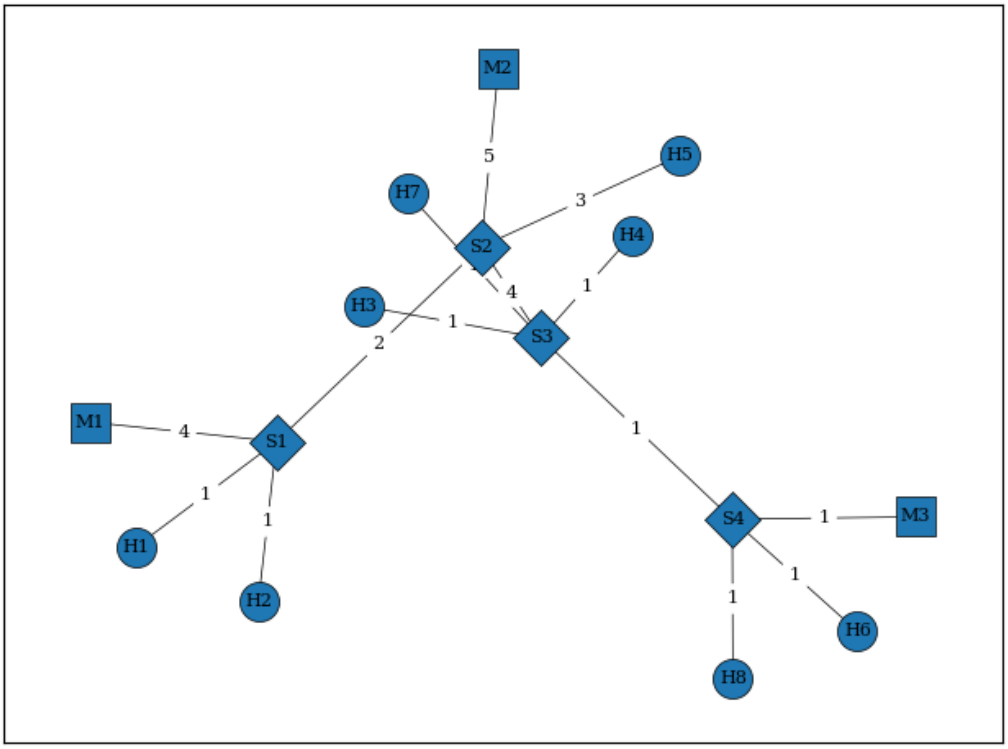

Figure 31: Test case 10 input.

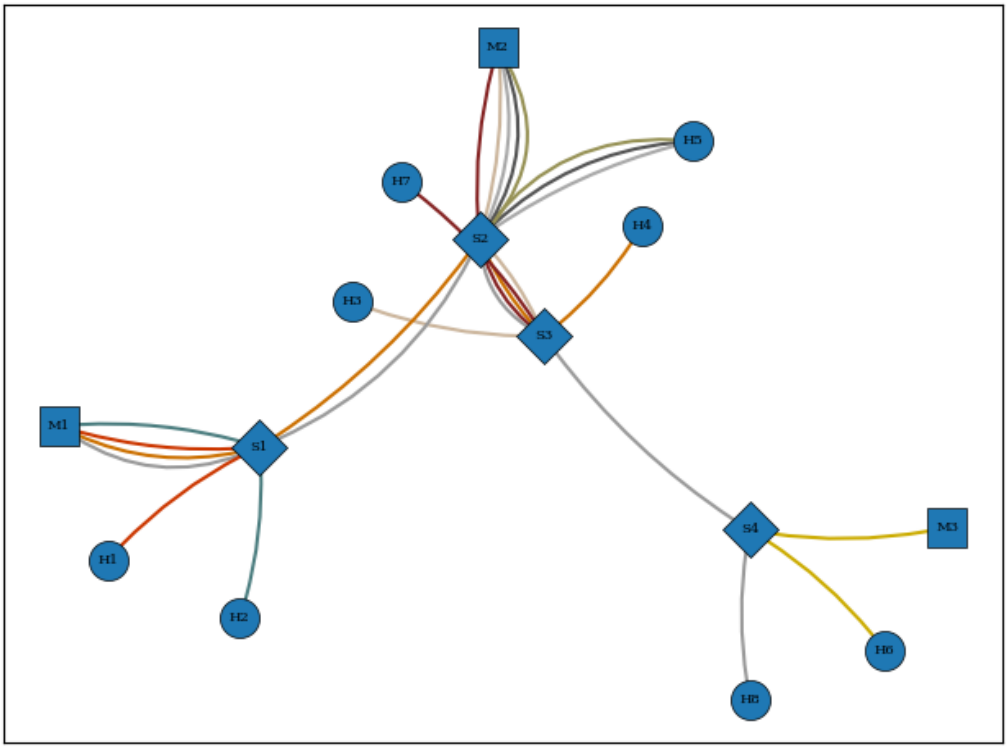

Figure 32: Test case 10 output.

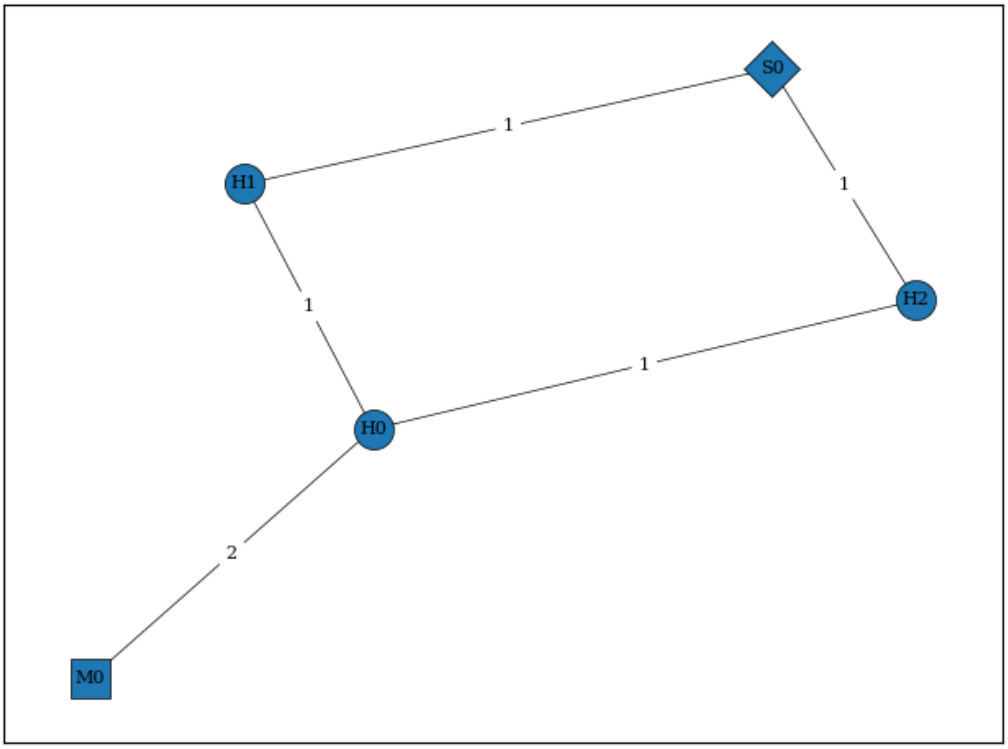

Figure 33: Test case 11 input.

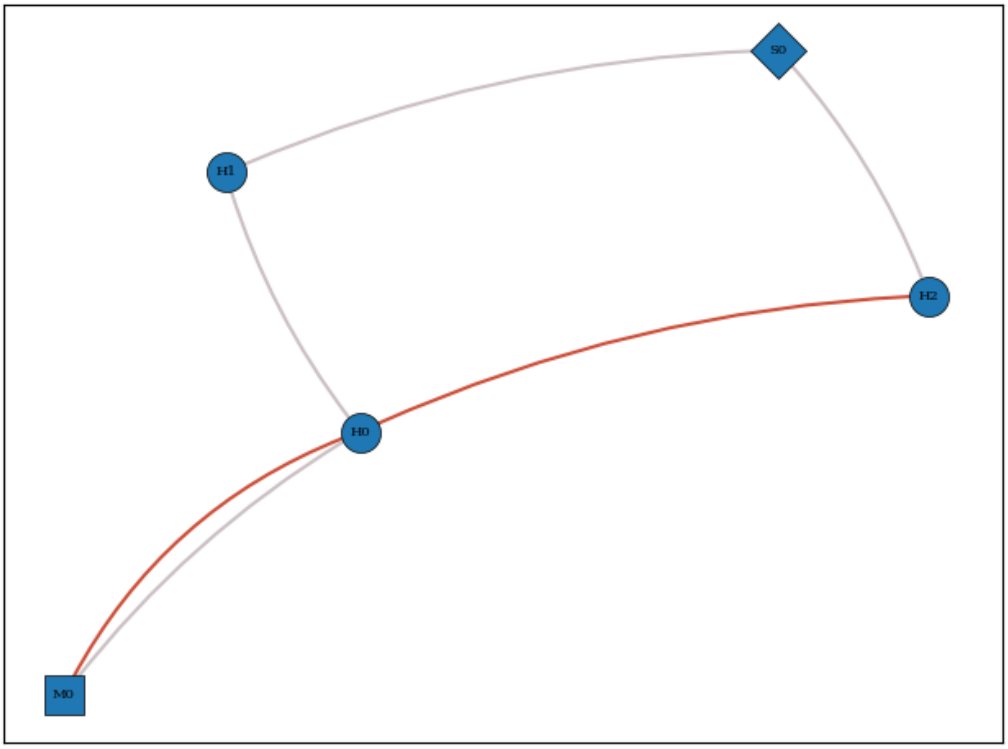

Figure 34: Test case 11 output.

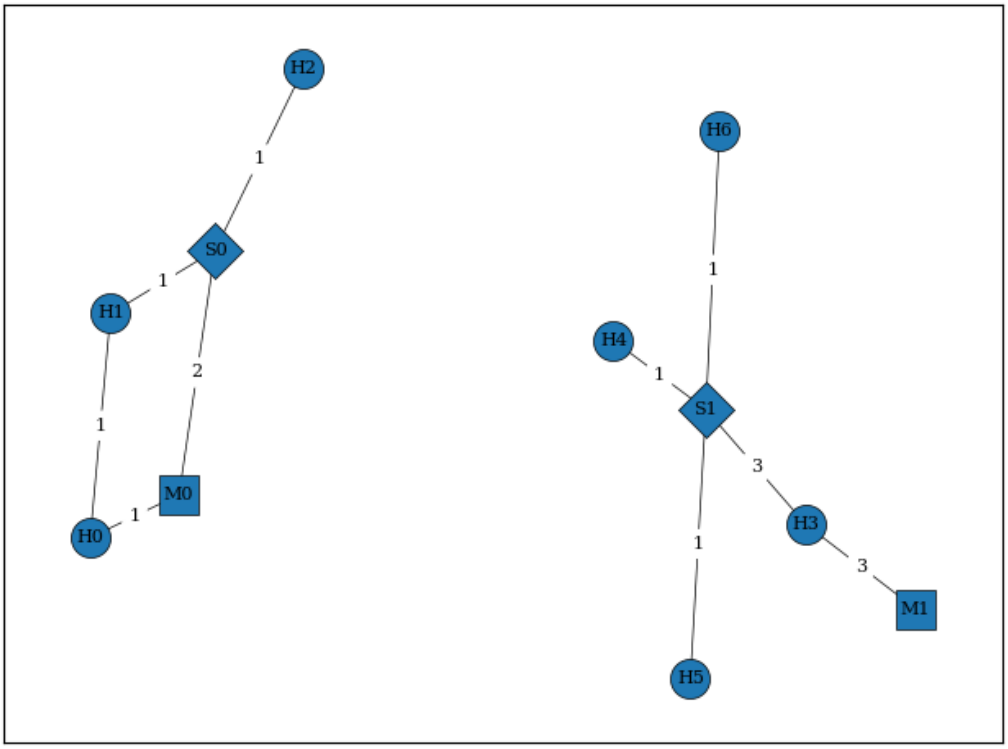

Figure 35: Test case 12 input.

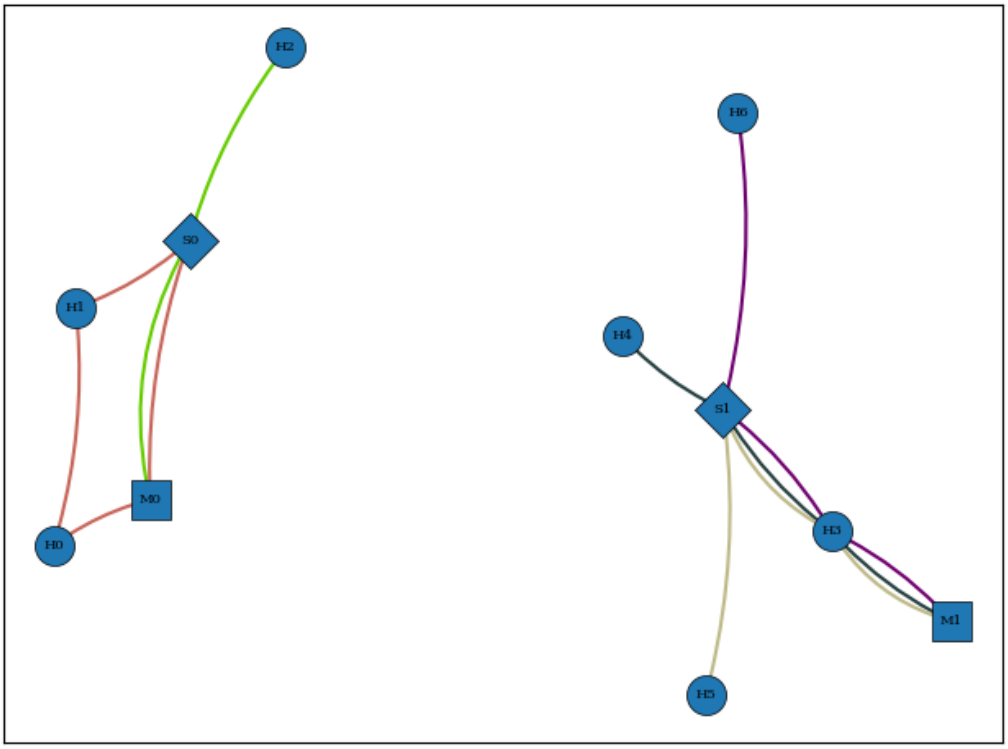

Figure 36: Test case 12 output.

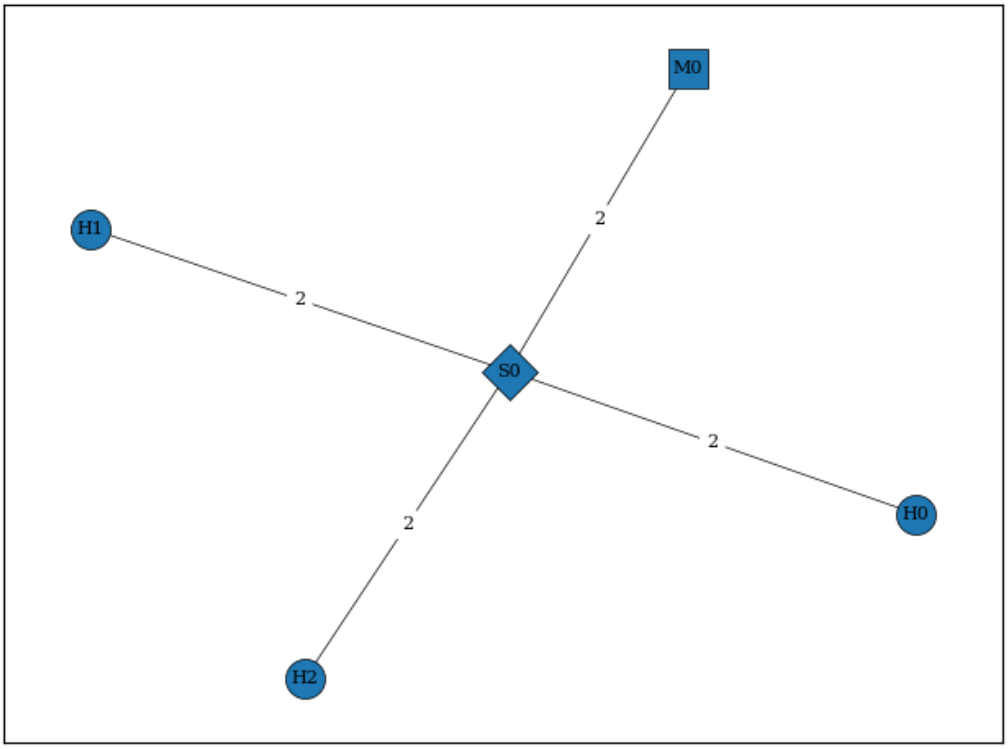

Figure 37: Test case 13 input.

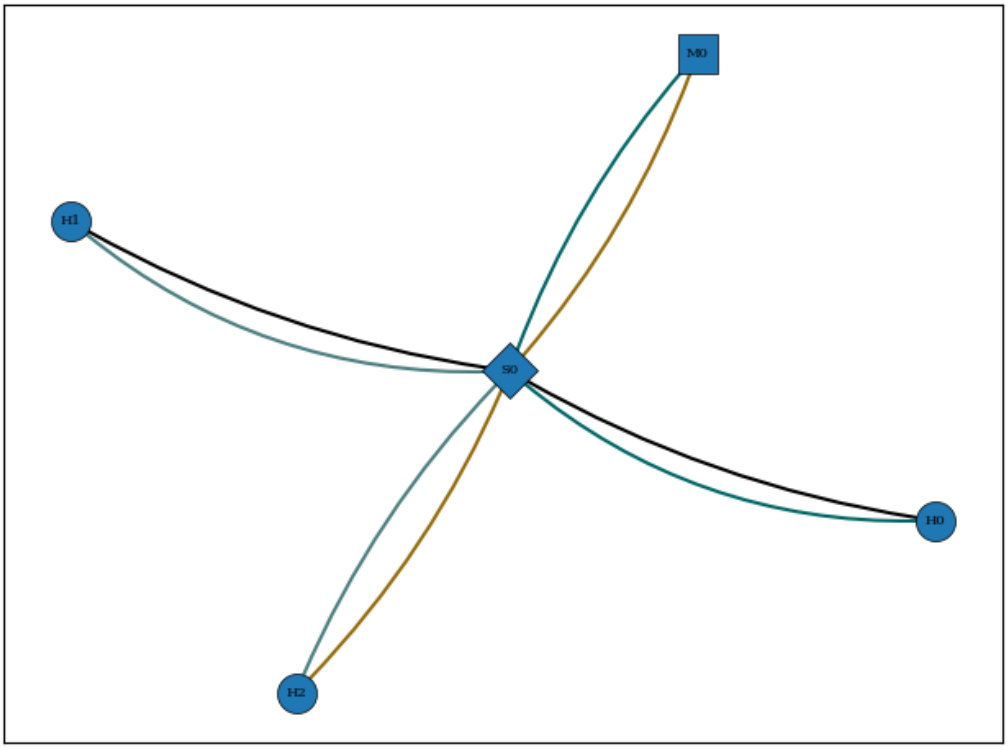

Figure 38: Test case 13 output.

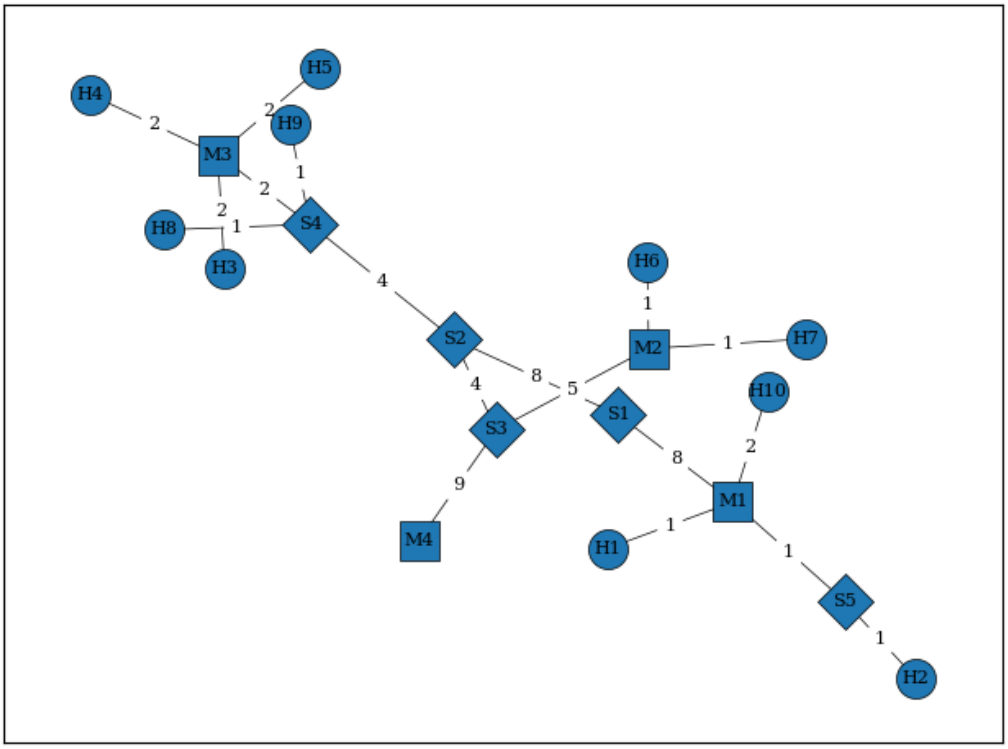

Figure 39: Test case 14 input.

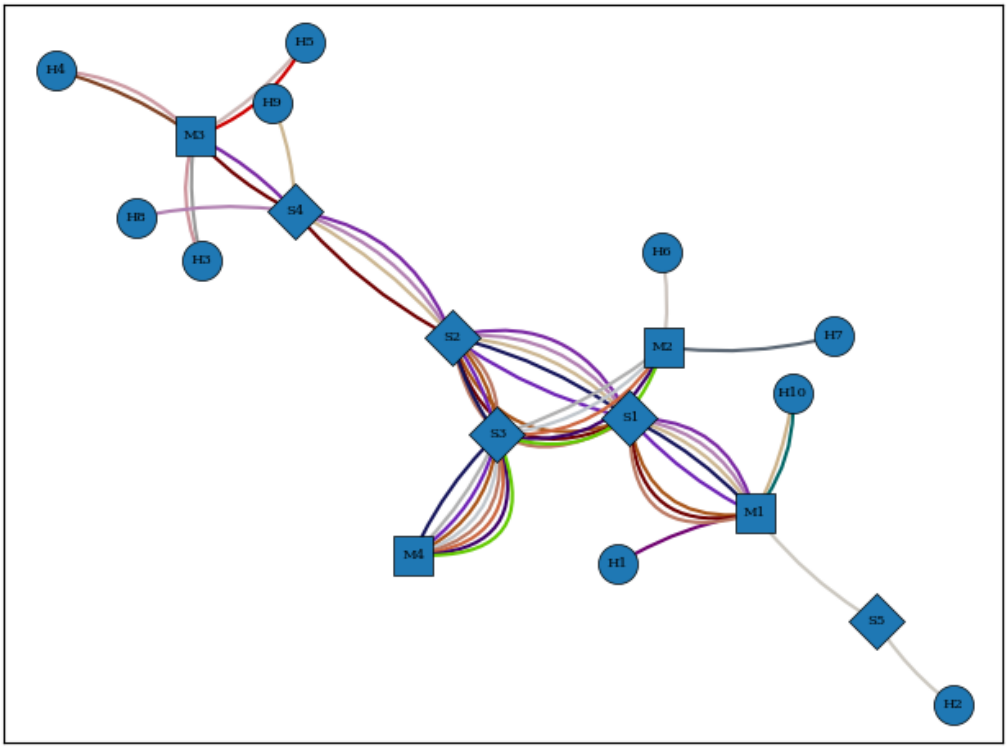

Figure 40: Test case 14 output.

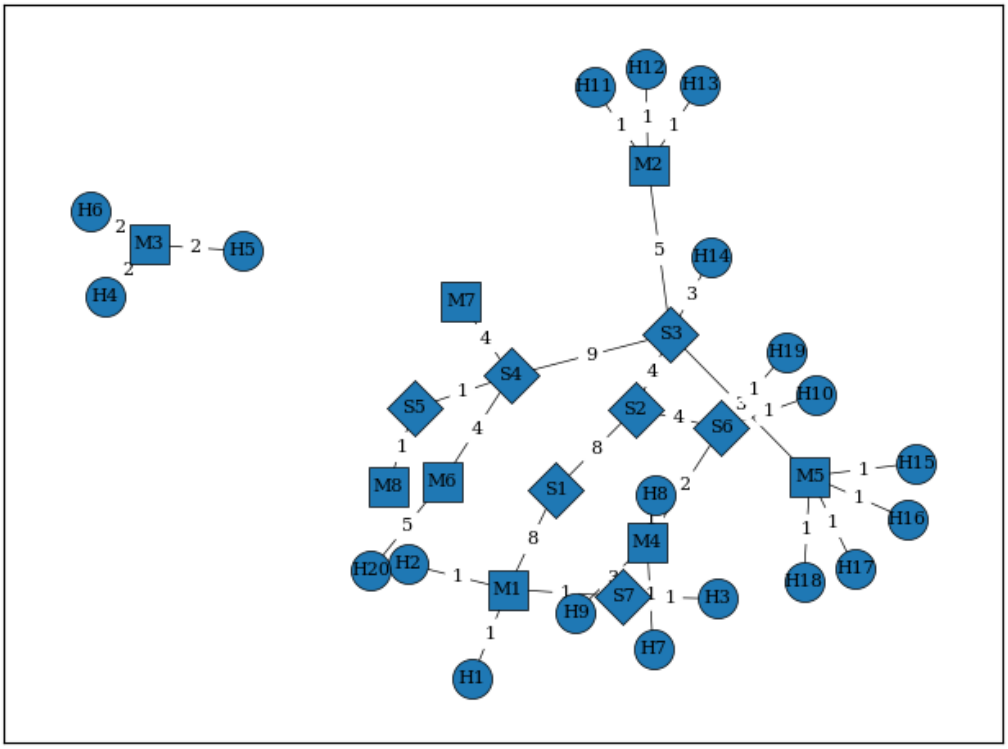

Figure 41: Test case 15 input.

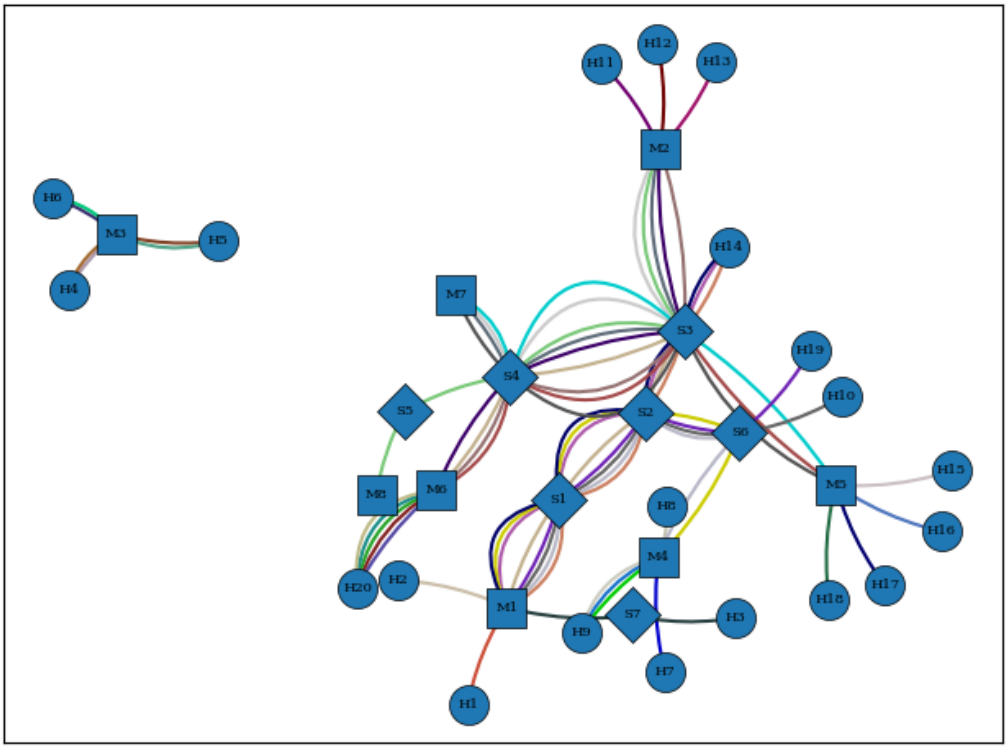

Figure 42: Test case 15 output.