Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed
Computing

Wave Function Collapse for Graph Generation

Semester Project

Soufiane Barrada

sbarrada@ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Till Aczel, Joël Mathys
Prof. Dr. Roger Wattenhofer

July 15, 2025

Abstract

In this work, we propose a novel approach to graph generation by adapting the Wave Function Collapse (WFC) algorithm to graphs. We introduce a tiling-based method where subgraphs are extracted from an input graph using k-hop neighborhoods and are used to label it using the Weisfeiler-Lehman algorithm to make tiles. These tiles are then recombined following structural constraints to generate new graphs that preserve the local patterns of the original. We present both a 1-hop and a generalized k-hop version, provide a formalized framework, and prove that any graph consistent with the tile set can be generated with nonzero probability. Our method enables flexible graph generation while ensuring structural coherence with the input data.

Contents

Chapter 1

Introduction

Graphs are one of the most versatile and expressive data structures in computer science, used extensively across various scientific disciplines. They enable the representation of complex relationships and interactions, allowing to represent interactions, dependencies, and structures in an intuitive and efficient manner. Graphs serve as the foundation for solving intricate problems and uncovering meaningful insights.

Due to their ubiquity, graphs find applications in diverse domains, such as in bioinformatics, syntax trees in linguistics, fraud detection in financial networks, and community detection in social graphs. Each of these use cases requires specific algorithms to extract meaningful insights.

Despite the richness of real-world graph datasets, many scenarios lack publicly available data due to privacy constraints or the nascent nature of certain application areas. In such cases, synthetic graph generation becomes essential for benchmarking algorithms and modeling systems. Furthermore, even when real-world data is available, testing graph algorithms across varying magnitudes and configurations necessitates configurable graph generation tools.

Graph generators aim to address these challenges by reproducing real-world graph properties while allowing customization for specific use cases. The importance of these tools extends to benchmarking algorithms, simulating complex systems, and generating test cases for emerging methodologies. However, existing graph generation techniques often cater to specific domains or lack flexibility, necessitating a unified approach that can adapt to different requirements.

This project draws inspiration from the Wave Function Collapse (WFC) algorithm, a procedural generation technique originally developed for image synthesis. WFC constructs outputs by sequentially selecting tiles based on local constraints, ensuring global consistency. Its ability to generate visually coherent and constraint-satisfying structures makes it an intriguing candidate for graph generation.

Adapting WFC to graphs requires translating its fundamental principles to a new context. In this project, we introduce graph tiles as subgraphs representing k-

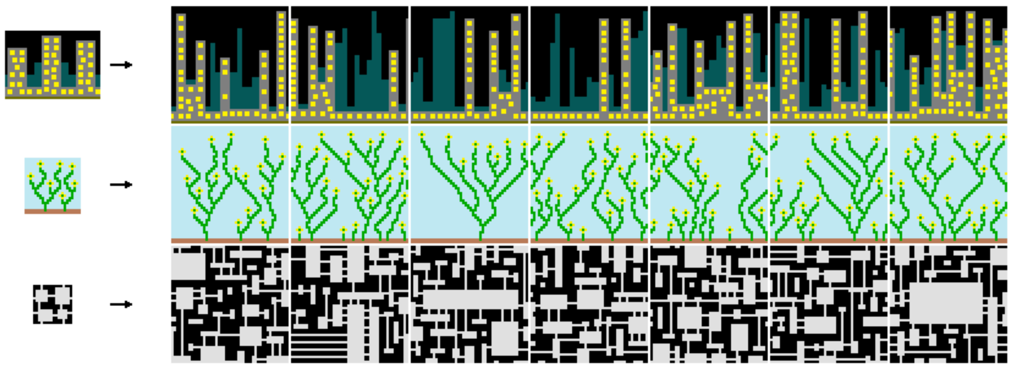


Figure 1.1: Wave Function Collapse Algorithm

hop neighborhoods around the graph's nodes. Each tile encapsulates local graph structure and has "open ends" representing potential connections to other tiles. By iteratively assembling these tiles while respecting connection constraints, we aim to generate synthetic graphs that mimic the properties of the input graph.

Chapter 2

Related Work

Graph generation has attracted some attention across various disciplines. However, the specific approach of adapting the Wave Function Collapse (WFC) algorithm to graph generation has not been explored in the literature. In this section, we provide an overview of related work, focusing on general graph generation methods, deep learning approaches, and substructure discovery techniques. While these methods offer valuable insights, none directly align with the methodology proposed in this project.

The study "Graph Generators: State of the Art" [1] categorizes modern graph generation techniques into domains such as Semantic Web, graph databases, social networks, and general graphs. Of particular interest to this project are the "general graph" generators, which are designed to reproduce properties of real graphs without being tied to specific applications. Examples include: Preferential Attachment: Generates scale-free networks by mimicking real-world degree distributions. R-MAT [2] and BTER [3]: Address community structures and clustering coefficients. Darwini [4]: A refinement of BTER, offering improved accuracy.

The survey "A Survey on Deep Graph Generation: Methods and Applications" [5] provides an in-depth exploration of deep learning techniques for graph generation. It categorizes generative models into five main types: Auto-Regressive Models, Variational Autoencoders (VAEs), Normalizing Flows, Generative Adversarial Networks (GANs), and Diffusion Models. Notable examples of these models include GraphRNN [6], which frames graph generation as a sequential process by adding nodes and edges step-by-step; GraphVAE [7], which uses a VAE-based framework to generate new graphs in a one-shot manner; MoFlow [8], which employs an invertible mapping between the input graph and latent space to generate graphs in a single step; MolGAN [9], which integrates a GAN-based approach with a discriminator to ensure desirable properties in the generated graphs; and GDSS [10], which adopts a score-based diffusion model that reconstructs graphs by denoising Gaussian noise applied to node and edge features.

The SUBDUE [11] system represents an alternative approach, focusing on substructure discovery using the Minimum Description Length (MDL) principle. By iteratively identifying and compressing substructures, SUBDUE creates a hierarchical representation of the input graph. Notable features include: Inexact Matching: Supports distortions in graph structures, enabling flexibility in substructure identification. Substructure Compression: Compresses repeated patterns into single vertices, allowing further analysis on a reduced dataset.

While SUBDUE offers valuable substructure extraction and discovery, its applicability to our project is limited because it requires labeled directed graphs as input, whereas our approach is designed for undirected, unweighted graphs. This makes our method more general and suitable for a broader range of graph generation tasks, where labeled directed graphs may not be available or appropriate.

Chapter 3

Preliminaries

In this chapter, we introduce the key concepts required to understand the rest of this report. We provide a brief overview of graphs, as well as the inclusive and exclusive K-hop neighborhoods of a vertex. Additionally, we discuss the Weisfeiler-Lehman algorithm for node labeling and conclude with an introduction to the Wave Function Collapse algorithm.

3.1 Graphs

A graph G is a mathematical structure defined as a pair $G = (V, E)$, where V is the set of vertices (nodes) and E is the set of edges, representing connections between pairs of vertices. Graphs can be classified based on their properties:

- **Directed vs. Undirected Graphs**: Directed graphs have edges with a direction, while undirected graphs represent bidirectional relationships.
- **Weighted vs. Unweighted Graphs**: Weighted graphs assign values (weights) to edges, while unweighted graphs treat all edges equally.

In this project, we focus on **unweighted, undirected graphs**, which simplifies the domain. This representation ensures its applicability to contexts such as social networks, where the directionality and weight of the edges are not critical.

3.2 K-hop Neighborhoods

The concept of a k-hop neighborhood is central to this project, as it defines the local subgraph around a node within a specified distance k. In this work, we distinguish between two definitions of k-hop neighborhoods, which differ in the treatment of edges between the outer nodes (i.e., nodes at exactly k-hop distance from the central node). These definitions are as follows:

3.2.1 Inclusive K-hop Neighborhood

The **Inclusive K-hop Neighborhood** of a node c, denoted as $N_k^{\text{inclusive}}(c)$, consists of all nodes reachable from c within k hops, along with all edges between these nodes. Formally, $N_k^{\text{inclusive}}(c) = (V_k, E_k)$, where:

$$\begin{aligned} V_k &= \{u \in V \mid \text{distance}(c, u) \le k\}, \\ E_k &= \{(u, v) \in E \mid u, v \in V_k\}. \end{aligned}$$

3.2.2 Exclusive K-hop Neighborhood

The **Exclusive K-hop Neighborhood** of a node c, denoted as $N_k^{\text{exclusive}}(c)$, is similar to the Inclusive K-hop Neighborhood but excludes edges between nodes that are both at exactly the k hop distance from the central node. Formally, $N_k^{\text{exclusive}}(c) = (V_k, E'_k)$, where:

$$\begin{aligned} V_k &= \{u \in V \mid \text{distance}(c, u) \le k\}, \\ E'_k &= \{(u, v) \in E \mid u, v \in V_k, (\text{distance}(c, u) < k) \text{ or } (\text{distance}(c, v) < k)\}. \end{aligned}$$

This definition excludes edges that directly connect two outer nodes (i.e., nodes at k-hop distance from c) while preserving all other edges within the neighborhood.

3.2.3 Extraction of K-hop Neighborhoods

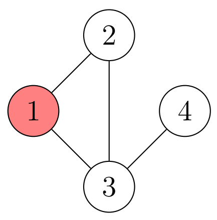
To extract these neighborhoods from a graph $G = (V, E)$, a breadth-first search (BFS) is performed starting from the central node c up to a depth of k. The edges E_k or E'_k are then determined based on the chosen definition:

- For $N_k^{\text{inclusive}}(c)$, include all edges where both endpoints are in V_k.
- For $N_k^{\text{exclusive}}(c)$, exclude edges connecting two nodes $u, v \in V_k$ such that $\text{distance}(c, u) = k$ and $\text{distance}(c, v) = k$.

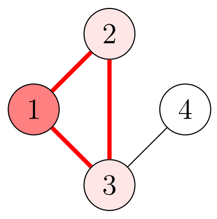
We define the K-hop neighborhood as the exclusive K-hop neighborhood by default, unless explicitly stated otherwise.

3.3 Weisfeiler-Lehman Algorithm for Node Labeling

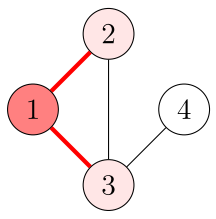
The Weisfeiler-Lehman (WL) algorithm is a graph isomorphism test that iteratively updates node labels based on their neighborhoods. This algorithm is widely used for graph representation and classification tasks due to its simplicity and effectiveness. In the context of this project, the WL algorithm is utilized to assign unique labels to graph nodes, capturing the structure of their local subgraphs.



(a) An undirected graph, with the considered central node in red



(b) Inclusive 1-hop neighborhood



(c) Exclusive 1-hop neighborhood

Figure 3.1: Illustration of the Inclusive and Exclusive 1-hop neighborhood on a simple undirected graph

3.3.1 Standard WL Algorithm

The WL algorithm operates as follows:

1. **Initialization:** Assign an initial label to each node. In the standard WL algorithm, all nodes are typically assigned the same initial label, such as 1.
2. **Neighborhood Aggregation:** At each iteration, update the label of a node based on its current label and the sorted multiset of labels of its neighbors. This can be expressed as:

   $$lab^{(t+1)}(v) = \text{hash}\left(lab^{(t)}(v), \text{sorted}\left(\{lab^{(t)}(u) \mid u \in N(v)\}\right)\right),$$

   where $lab^{(t)}(v)$ is the label of node v at iteration t, and $N(v)$ represents the neighbors of v.
3. **Termination:** We terminate when the labels stabilize (i.e., no further changes occur) or when a predefined number of iterations is reached.

The final labels encode structural information about the graph.

3.3.2 Modified WL Algorithm for Improved Labeling

To better capture the structure of subgraphs, we introduce a variation in the initialization step of the WL algorithm. Specifically, for a given spanning (central) vertex v_c, we initialize its label as 0, while all other nodes in the graph are initialized with the label 1. This modification aims to enhance the differentiation between subgraphs, particularly in cases where standard initialization may fail to capture key structural distinctions.

Consider the example of cycles of different sizes. In the standard WL algorithm, cycles of size 5 and 7 yield the same label for the spanning vertex (for the same number of iterations). However, by assigning a distinct initial label to the spanning vertex, the WL algorithm produces different final labels for the central vertex in each cycle, reflecting the structural differences.

3.3.3 Modified Initialization:

$$lab^{(0)}(v) = \begin{cases} 0 & \text{if } v = v_c, \\ 1 & \text{otherwise.} \end{cases}$$

The modified initialization allows the WL-algorithm to better differentiate between graphs.
Therefore, the labels generated for the spanning vertex will be more informative and context-aware, which is crucial for our purposes.

3.4 Wave Function Collapse Algorithm

The Wave Function Collapse (WFC) algorithm is a procedural generation technique originally designed for image synthesis. It constructs outputs by iteratively selecting and collapsing "tiles" while respecting predefined constraints. The algorithm is inspired by principles of quantum mechanics, particularly the idea of superposition, and has been widely applied in procedural content generation.

3.4.1 General Overview

The WFC algorithm operates on a grid where each position represents a "cell." Each cell initially holds a superposition of possible states (tiles), and the algorithm progressively collapses the superposition to a single state by enforcing constraints.

Steps of the Algorithm:

1. **Initialization**: Each cell is initialized with all possible tiles in a superposition. The set of tiles is determined based on the problem domain and constraints.

2. **Entropy Calculation**: The entropy of each cell is calculated based on the number of valid states (tiles) remaining in its superposition. Cells with fewer possibilities have lower entropy.

3. **Selection**: A cell with the lowest entropy is selected for collapse. In case of ties, a random choice is made.
4. **Collapse**: The selected cell is collapsed to a single tile, chosen randomly (and may be with different weights) from its valid states.
5. **Propagation**: The constraints of the collapsed tile are propagated to neighboring cells, updating their possible states and reducing entropy.
6. **Iteration**: Steps 2-5 are repeated until all cells are collapsed or no valid configurations remain.

Constraints and Propagation

Constraints define the valid states for each cell based on its neighbors. For example, in video game level generation, constraints might ensure that some tiles never neighbor each other. Propagation enforces these constraints, ensuring that invalid configurations are eliminated as the algorithm progresses.

3.4.2 Backtracking

In some cases, the WFC algorithm encounters a situation where no valid states remain for a cell due to conflicting constraints. This necessitates backtracking:

- The algorithm revisits the previously collapsed cell and resets its state to a superposition of valid tiles.
- Neighboring cells affected by the reset are also updated to restore consistency.
- Backtracking ensures that the algorithm can recover from conflicts and explore alternative configurations.

While backtracking adds computational complexity, it is a critical feature for handling highly constrained problems, such as graph generation, where conflicts are more likely to occur due to intricate local structures.

3.4.3 Applications and Adaptation to Graphs

The WFC algorithm is highly versatile and has been adapted for tasks such as texture synthesis, level design, maze generation, and, in this project, graph generation. In the context of graphs:

- Tiles correspond to nodes at the center of a typical graph substructure.

- Constraints ensure that generated graphs adhere to the structural properties of the input graph.

Chapter 4

Procedural Graph Generation

This chapter provides a high-level overview of our 1-hop and k-hop graph generation methods. We present the motivation, the key steps in the workflow, and the overall idea behind using *tiles* to build larger graphs. This chapter focuses on the **conceptual understanding** rather than detailed formalization, which will be covered in the next chapter.

4.1 The 1-hop Graph Generation method

4.1.1 Motivation and Background

Our technique aims to **capture the local structure** around each node (using its 1-hop neighborhood) and use these building blocks—the *tiles*—to incrementally assemble a larger graph. The primary goal is to ensure that any *local constraints* reflected in the original graph can also appear in newly generated graphs.

4.1.2 Labeling the Input Graph with WL

Given an undirected input graph $G = (V, E)$, we assign each node a label by doing the following:

1. **1-Hop Extraction:** For each node $v \in V$, collect its immediate neighbors $N(v)$. This forms a small induced subgraph around v.
2. **Weisfeiler–Lehman Iterations:** Run two iterations of the WL procedure on the 1-hop subgraph of each node to capture the local structure (the degree in this case).
3. **Final Node Label:** After two iterations, the label of node v in G is stored as the label of v $lab(v)$. The other node labels in the subgraph are not of interest to us.

4.1.3 Extracting Tiles

Once every node v has an assigned label $lab(v)$, we create a *tile* to encapsulate the local information around v. A tile T_v is composed of:

$$T_v = \Big(lab(v),\ \{lab(u) \mid (v, u) \in E\} \Big),$$

where the first component is the label of v, and the second component is a multiset of labels corresponding to v's immediate neighbors. Because different nodes might share the same label $lab(v)$ but have different labeled neighbors, each such node yields a distinct tile.

4.1.4 Tile Library

We collect these tiles into a *tile library* $\mathcal{T}_G$. Concretely, for each node $v \in V$, we store T_v in $\mathcal{T}_G$. Thus, $\mathcal{T}_G$ reflects all the local 1-hop configurations present in the original graph G.

Importantly, the tile library allows the same tile to appear multiple times. This design choice preserves the statistical distribution of local substructures in the input graph, ensuring that tiles that are more frequently encountered in G are also more likely to be selected during the procedural generation process. Instead of considering all unique tiles with equal probability, the tile library inherently biases tile selection towards structures that naturally occur more often.

This approach helps maintain the structural essence of the input graph by capturing the frequency of different neighborhoods, thereby guiding the generation process toward outputs that better reflect the nature of the original data. By leveraging the empirical distribution of tiles, we increase the likelihood that commonly occurring patterns in G are faithfully reproduced, reinforcing the consistency of the generated graphs.

4.1.5 Generation Algorithm

We begin graph generation with a single tile, then iteratively "concretize" one *open end* at a time. An open end is a neighbor label from a previously added tile that has not yet been matched. At each iteration, we choose an open end from the partially constructed graph and decide whether to close it by connecting to an existing tile or introduce a new tile that fulfills the required neighbor label.

Connecting Tiles to Build Graphs

When we want to build a new graph, we do so one edge at a time by *connecting tiles*. Two tiles can be connected if:

$$\text{Tile}_1.\text{label} \in \text{Tile}_2.\text{neighbor\_labels} \quad \text{and} \quad \text{Tile}_2.\text{label} \in \text{Tile}_1.\text{neighbor\_labels}.$$

This mirrors how, in the original graph, we would only connect nodes that are consistent with each other's neighborhood labels.

Probabilistic Generation

To avoid determinism and allow the algorithm to explore different structures, we use a *probabilistic* procedure:

- With probability p, we try to connect a new tile to an already existing tile in the partially built graph.
- With probability $1-p$, we introduce a **brand-new** tile (node) and connect it to the graph.

Termination

This process continues until there are no remaining open ends. Each edge addition is driven by the label constraints of the tiles and controlled by the probability p. In doing so, we ensure that every connection respects the local structure originally captured in $\mathcal{T}_G$.

4.1.6 Illustrative Example

As shown in Figure 4.1, the initial graph consists of five nodes. The labeled graph in Figure 4.2 demonstrates the Weisfeiler-Lehman lebeling process. Steps of the 1-hop graph generation are illustrated in Figures 4.3 to 4.6.

- At step 1 in Figure 4.3, the algorithm randomly chooses a tile to start with.
- At steps 2, 3, 4 in Figures 4.4 to 4.5, the algorithm decides with probability 1-p (at each step)to add new tiles.
- At step 5 in Figure 4.6, the algorithm decides to close the selected open end with probability p, and connects the two tiles.

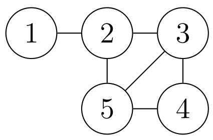


Figure 4.1: Input graph

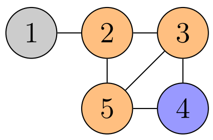


Figure 4.2: Labeled graph

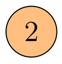


Figure 4.3: Step 1

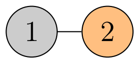


Figure 4.4: Step 2

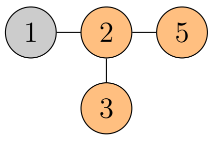


Figure 4.5: Step 3 + 4

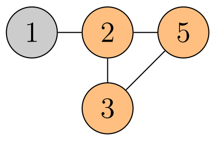


Figure 4.6: Step 5

Figure 4.7: An input graph, it's WL-labelling and some steps of the 1-hop graph generation algorithm.

4.1.7 Limitation

Strict Locality: Because each tile encodes only a node and its immediate neighbors, longer-range dependencies in the original graph are *not* explicitly captured. While this approach preserves local structures effectively, it fails to account for patterns that emerge at a larger scale, such as motifs, cycles, or hierarchical relationships between nodes that are further apart.

To address this, we extend our definition of a tile beyond the 1-hop neighborhood and introduce a more general notion in the following section.

4.2 The k-hop Graph Generation Method

To overcome the limitations of strict locality in the 1-hop neighborhood approach, we extend our methodology to incorporate *k-hop neighborhoods*. Instead of considering only a node and its immediate neighbors, we define tiles based on nodes and their surrounding structures up to a distance of k.

By increasing k, we gain a more expressive representation of local subgraphs, enabling our generation process to better reflect long-range relationships present in the input graph. The challenge, however, lies in balancing expressiveness with computational efficiency, as larger neighborhoods introduce more complexity. The following sections detail how we define, extract, and utilize these generalized tiles in our graph generation framework.

4.2.1 Labeling the Input Graph with WL for k-Hop Neighborhoods

For a given input graph $G = (V, E)$ and a chosen k, we assign labels to nodes based on their k-hop neighborhoods. The process involves:

1. **k-Hop Extraction:** For each node $v \in V$, extract the subgraph induced by all nodes within k hops of v. This includes v and all nodes reachable within k steps.
2. **Weisfeiler–Lehman Iterations:** Apply the WL procedure to the k-hop subgraph of each node to generate unique labels that capture the structural and relational information within the neighborhood.
3. **Final Node Label:** After $2*k$ iterations, the label of node v is determined, reflecting the structure of its k-hop neighborhood. This label is stored as $lab(v)$.

This process ensures that nodes with similar k-hop neighborhood structures in the input graph receive the same label, enabling consistent tile definitions.

4.2.2 Updated Tile Representation with Subgraphs

To fully capture the structural variability within a k-hop neighborhood, we enhance the tile representation to include all valid subgraphs that contain the central node. Each subgraph is uniquely identified by the Weisfeiler-Lehman (WL) label assigned to the central node when the WL algorithm is applied to the subgraph. Formally, the updated tile representation is defined as:

$$T_v = \Big(lab(v),\, \{lab(u) \mid (v,u) \in E\},\, \{lab_{\text{sub}}(v, G') \mid G' \subseteq \text{k-hop}(v) \wedge v \in G'\}\Big),$$

where:

- $lab(v)$ is the WL label of the central node v in the full k-hop neighborhood.
- $\{lab(u)\}$ is the multiset of WL labels for nodes within the k-hop neighborhood of v.

- $\{lab_{\text{sub}}(v, G')\}$ represents the set of WL labels for all subgraphs G' of the k-hop neighborhood of v that include v.

This expanded definition ensures that every tile not only captures the full k-hop neighborhood but also accounts for the structural diversity of its subgraphs.

4.2.3 Extracting Subgraphs and WL Labels

To construct this additional component of the tile, we enumerate all possible subgraphs of the k-hop neighborhood that include the central node. The WL algorithm is then applied to each subgraph, and the resulting label for the central node is stored. The following steps outline the process:

1. Compute the WL label for the full k-hop neighborhood of v.
2. Enumerate all subsets of nodes in the k-hop neighborhood that include v, excluding the empty set and disconnected subgraphs.
3. For each connected subgraph, apply the WL algorithm and record the label of v.

4.2.4 Generation Algorithm

The k-hop generation algorithm introduces a new logic to ensure that the structural consistency of k-hop neighborhoods is preserved:

1. **Initialization:** Start with a single tile chosen from $\mathcal{T}_G$.
2. **Iterative Expansion:** At each step, select an open end from the partially constructed graph and try to close it with an already existing tile with probability p, or propose a new tile to connect with probability $1 - p$.
3. **Validation of k-Hop Neighborhoods:** Before connecting to a proposed tile:
    - Ensure that the k-hop neighborhood of the proposed tile in the partially constructed graph forms a valid subgraph of the original k-hop neighborhood of that tile in G.
    - Verify that the k-hop neighborhoods of all tiles within the k-hop radius of the proposed tile remain valid subgraphs of their original k-hop neighborhoods in G.

4.2.5 Validation step

If the k-hop validation step were omitted, the algorithm might produce structures unconsistent with the input, such as smaller cycles that do not exist in the input graph's k-hop neighborhoods. By enforcing this constraint, we ensure that the generated graph faithfully reflects the input graph's k-hop patterns.

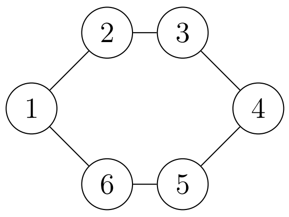


Figure 4.8: Input graph

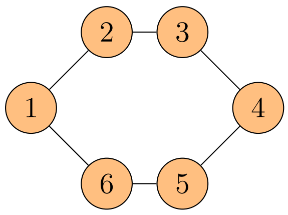


Figure 4.9: 2-hop WL-labeling of the input graph

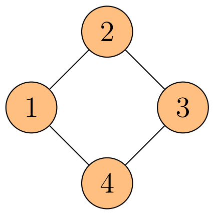


Figure 4.10: Possible output graph without the validation step

4.2.6 Advantages of the k-Hop Method

- **Captures Long-Range Dependencies:** By incorporating information from k-hop neighborhoods, this method encodes patterns that span multiple hops.
- **Consistency with Input Graph:** The validation step ensures that the generated graph respects the structural constraints of the input graph at the k-hop level.
- **Flexibility:** The parameter k allows for tuning the balance between local and global structural fidelity.

4.2.7 Limitations

While the k-hop graph generation method significantly improves the expressiveness of the generated graphs, it comes with the following limitations:

- **Computational Complexity:** The process of precomputing all possible subgraphs for each tile is computationally expensive. As the size of the k-hop neighborhood increases, the number of subgraphs grows exponentially.
- **Potential for Stalled Generation:** The generation process may encounter situations where no valid tile can be added to the partially constructed graph without violating the constraints. To address this, a robust backtracking mechanism is required. This mechanism must be capable of undoing multiple steps, potentially going back several iterations to explore alternative construction paths.

4.3 Summary

In this chapter, we have presented an accessible description of our 1-hop and k-hop graph generation approach. The next chapter will introduce a formalization of these concepts for the 1-hop case.

Chapter 5

Formalization

In this chapter, we provide a rigorous formalization of the 1-hop graph generation method. We define tiles and tile sets, present the procedural generation algorithm in pseudocode, and prove that any graph consistent with the given tile set can be generated with non-zero probability. This formal treatment establishes a solid foundation for understanding the theoretical guarantees and capabilities of our approach.

5.1 Set up

5.1.1 Graph

$G = (V, E)$ is an undirected graph without self-loops nor double edges, where:

- V is the set of vertices (or nodes).
- $E \subseteq V \times V$ is the set of edges.
- $\forall (u, v) \in E : u \neq v$.

5.1.2 Labels

We define a label set L, and a mapping lab such that:

$$lab : V \to L$$

For our purpose, we generate the label map lab by running 2 WL iterations on the 1-hop neighborhood of each node. This implies that:

$$lab(v1) = lab(v2) \Leftrightarrow deg(v1) = deg(v2)$$

5.1.3 Tiles

We define a Tile as a tuple $T = (l, N)$, where:

- $l \in L$.
- $N = \biguplus_{i \in \{0,\dots,n\}} \{l_i\}$, is a multiset, where $l_i \in L$ and $n \in \mathbb{N}$.
- $|N| > 0$

In our approach, tiles represent individual nodes with their labels and the labels of their neighbors. Two tiles $T_1 = (l_1, N_1), T_2 = (l_2, N_2)$ can be connected iff:

$$l_1 \in N_2 \text{ and } l_2 \in N_1$$

5.1.4 Tile set

The tile set $\mathcal{T}_G$ of a graph G is the multiset of tiles extracted from G. Each vertex v in G corresponds to a tile T_v defined as:

$$T_v = (lab(v), \{lab(u) \mid (v, u) \in E\}),$$

where $lab(v)$ is the label of vertex v, and $lab(u)$ represents the label of its adjacent vertices.

5.2 Algorithm

5.2.1 Data structures

We assume a data structure "Graph", that maintains a mapping from the Tiles we add to it, to corresponding graph nodes it generates. It also offers 3 methods:

- add_node(Tile t): Given a tile, this method will create a corresponding node for it in its internal representation of a graph.
- add_edge(Tile t1, Tile t2): Creates an edge between the corresponding nodes of Tiles t1 and t2.
- get_graph(): returns the generated graph.

5.2.2 Probability p

p determines the probability during graph generation of connecting a tile to other already existing tiles versus creating a new tile to connect to it $0 < p < 1$.
We assume, and without loss of generality, that p is constant.

5.2.3 Unique identity of tiles

$T_1 ... T_m$ are said to be "similar" tiles, iff

$$\forall\ i, j\ 0 \leq i \neq j \leq m : (T_i.l = T_j.l \quad \& \quad T_i.N = T_j.N)$$

We assume that any similar Tiles $T_1 ... T_m$ sampled from $\mathcal{T}_G$ are uniquely identifiable.

5.2.4 PGG

Algorithm 1: PGG: Procedural Graph Generation

```
Input: $(\mathcal{T}_G, p)$
Output: Generated graph $G$
1  $G \leftarrow$ Graph();
2  Sample a random tile $T_1$ from $\mathcal{T}_G$;
3  $G$.add_node($T_1$);
4  Open_tiles $\leftarrow \{T_1\}$ // Set of tiles with open ends
5  while $Open\_tiles \neq \emptyset$ do
6      $T_{\text{current}} \leftarrow$ random_pick(Open_tiles);
       // Decide whether to connect to an existing tile or sample
          a new one
7      Sample a random number $r \in [0, 1]$;
8      if $r < p$ and $size(Open\_tiles) > 1$ then
9          Pick $T_{\text{close}} \in$ Open_tiles $\setminus \{T_{\text{current}}\}$ such that $T_{\text{close}}$ can connect
            to $T_{\text{current}}$;
10     else
11         Sample $T_{\text{close}}$ from $\mathcal{T}_G$ such that $T_{\text{close}}$ can connect to $T_{\text{current}}$;
12         $G$.add_node($T_{\text{close}}$);
13         $Open\_tiles$.add($T_{\text{close}}$);
       // Add the edge and update open ends
14     $G$.add_edge($T_{\text{current}}, T_{\text{close}}$);
15     $T_{\text{current}}.N \leftarrow T_{\text{current}}.N \setminus \{T_{\text{close}}.\ell\}$;
16     $T_{\text{close}}.N \leftarrow T_{\text{close}}.N \setminus \{T_{\text{current}}.\ell\}$;
       // Cleanup Open_tiles
17     foreach $T \in Open\_tiles$ do
18         if $T.N = \emptyset$ then
19             Open_tiles.remove(T);
20 return $G.get\_graph()$;
```

5.3 Goal

We would like to prove that for any graph G' that generates a subset of the Tileset $\mathcal{T}_G$, it could be generated with non-zero probability by our graph generation algorithm PGG.
More formally we would like to prove that:
$\forall G, G'$ with tilesets $\mathcal{T}_G$, $\mathcal{T}'_G$:

$$\mathcal{T}'_G \subseteq \mathcal{T}_G \Leftrightarrow Pr(PGG(\mathcal{T}_{G'}, p) = G') > 0$$

5.4 Proof

5.4.1 Statement

Let $G = (V, E)$ and $G' = (V', E')$ be two graphs. Suppose $\mathcal{T}_G$ and $\mathcal{T}_{G'}$ are their corresponding tile sets, and assume $\mathcal{T}_{G'} \subseteq \mathcal{T}_G$. Consider the procedural graph generation algorithm $PGG(\mathcal{T}_G, p)$, which at each step adds exactly one edge; this edge can either connect to 2 existing tiles with probability $p \in (0, 1)$ or introduce a new tile and connect it to an already existing tile with probability $1 - p$.
We aim to prove that:

$$\mathcal{T}'_G \subseteq \mathcal{T}_G \Leftrightarrow Pr(PGG(\mathcal{T}_{G'}, p) = G') > 0$$

5.4.2 Proof

Idea: We construct an ordered sequence of edges S such that: if $PGG(\mathcal{T}_G, p)$ realizes S, it would produce G'. We then show that S has a strictly positive probability of occurring.

Step 1: Constructing the Sequence S from G'

1. **Walk on G':** Start from an arbitrary node $v_0 \in V'$, whose corresponding tile is $T_0 \in \mathcal{T}_{G'}$. Perform a walk on G', that ensures every edge in E' is visited at least once. Such a walk can, if necessary, revisit nodes and edges to ensure all edges in E' appear in the walk. Let $S_{walk} = (e_1, e_2, \ldots, e_m)$ be the sequence of edges encountered during this walk. Some edges may appear multiple times.
2. **Eliminate duplicates:** From the sequence S_{walk}, create S by including each edge from E' only the first time it appears and ignoring any subsequent appearances. After this step, $S = (edge_1, edge_2, \ldots, edge_{|E'|})$ is a sequence containing each edge of E' exactly once.

Step 2: Interpreting S as a PGG Execution

Consider running $PGG(\mathcal{T}_G, p)$ and enforcing it to add edges in the order specified by S.

- **Initial Step:** We start with an empty graph and add tile $T_0 \in \mathcal{T}_{G'}$. This is possible because $\mathcal{T}_{G'} \subseteq \mathcal{T}_G$.

- **Subsequent Steps:** For each edge $edge_i \in S$, we consider its endpoints. If an endpoint has not yet been added to the generated graph, we instantiate its corresponding tile from $\mathcal{T}_G$ and connect it to edge's other end. If both endpoints are already present, then we connect them, and that is necessarily possible as both have an open end corresponding to $edge_i$ (since this is how G' is structured). This means then that the algorithm either:

- Introduces a new node if required with probability $1 - p > 0$, or
- Closes the open ends with probability $p > 0$.

Because we have full freedom in conceptualizing the algorithm's random choices, we can imagine a scenario where at every step it picks exactly the needed tile from $\mathcal{T}_G$. And since $\mathcal{T}_{G'} \subseteq \mathcal{T}_G$, the right tile is always possible.

Step 3: Non-Zero Probability of S Occurring

- **Finite Choices with Positive Probability:** Each time we introduce a new tile (node), we pick from a finite set $\mathcal{T}_G$. Selecting any particular tile has positive probability since probabilities over finite sets are strictly greater than zero for each element.

- **Edge Addition Probability:** At each step of adding an edge from S, we either connect to an existing tile with probability $p > 0$, or instantiate a new tile with probability $1-p > 0$. Thus, every decision point has a strictly positive probability.

- **Product of Positive Terms:** The probability of the entire sequence S occurring is the product of positive probabilities (each step yields a positive probability event). Hence, the overall probability is strictly greater than zero:
$$\Pr(\mathrm{PGG}(\mathcal{T}_{G'}, p) = G') \geq \Pr(S) = \prod_{i=1}^{n} p_{step_i} > 0$$
where p_{step_i} is the probability of the i-th step.

- **Termination:** In the first step, only a single node is instantiated. From the second step onward, the algorithm creates one node and one edge, or a single edge per step following the sequence S. To fully close the generated graph, $|S| = |E'|$ steps are required. Thus, the algorithm terminates after:

$$n = |E'| + 1 \text{ steps.}$$

- **Conclusion:** Therefore, the algorithm can generate G' with non-zero probability.

Chapter 6

Conclusion

In this semester project, we developed a novel procedural graph generation framework inspired by the wave function collapse algorithm. Our method enables the generation of new graphs that locally preserve the structural patterns of an input graph while allowing novel graph generation.

We formalized the 1-hop version of our approach, providing a rigorous definition of tiles, a probabilistic generation algorithm, and a proof ensuring that any graph consistent with the tile set can appear with nonzero probability. Extending the method to k-hop neighborhoods allows to capture longer-range dependencies, improving structural fidelity at the cost of increased computational complexity.

Future work could explore optimizations in tile selection, a less constraining condition for tile connections in the k-hop case, and an efficient backtracking mechanism.

Bibliography

[1] A. Bonifati, I. Holubová, A. Prat-Pérez, and S. Sakr, "Graph generators: State of the art and open challenges," *ACM Computing Surveys*, 2020.

[2] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-mat: A recursive model for graph mining," in *Proceedings of the 4th SIAM International Conference on Data Mining*, 2004, pp. 442–446.

[3] T. G. Kolda, A. Pinar, T. Plantenga, and C. Seshadhri, "A scalable generative graph model with community structure," in *Proceedings of the IEEE International Conference on Big Data*, 2014.

[4] S. Edunov, D. Logothetis, C. Wang, A. Ching, and M. Kabiljo, "Darwini: Generating realistic large-scale social graphs," *arXiv preprint*, vol. arXiv:1610.00664, 2016.

[5] Y. Zhu, Y. Du, and Y. Wang, "A survey on deep graph generation: Methods and applications," *ACM Computing Surveys*, 2022.

[6] J. You, R. Ying, X. Ren, W. Hamilton, and J. Leskovec, "Graphrnn: Generating realistic graphs with deep auto-regressive models," in *Proceedings of the International Conference on Machine Learning (ICML)*, 2018.

[7] M. Simonovsky and N. Komodakis, "Graphvae: Towards generation of small graphs using variational autoencoders," in *Proceedings of the International Conference on Artificial Neural Networks (ICANN)*, 2018.

[8] C. Zang and F. Wang, "Moflow: An invertible flow model for generating molecular graphs," in *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 2020.

[9] N. D. Cao and T. Kipf, "Molgan: An implicit generative model for small molecular graphs," *arXiv preprint*, vol. arXiv:1805.11973, 2018.

[10] J. Jo, S. Lee, and S. J. Hwang, "Score-based generative modeling of graphs via the system of stochastic differential equations," in *Proceedings of the International Conference on Machine Learning (ICML)*, 2022.

[11] L. B. Holder, "Empirical substructure discovery (subdue)," in *Proceedings of the IEEE International Conference on Data Mining*, 1994, pp. 133–140.

[12] C. Morris, N. M. Kriege, F. Bause, K. Kersting, P. Mutzel, and M. Neumann, “Tudataset: A collection of benchmark datasets for learning with graphs,” in *ICML 2020 Workshop on Graph Representation Learning and Beyond (GRL+ 2020)*, 2020. [Online]. Available: www.graphlearning.io

APPENDIX A

Appendix A

In this appendix, we explore some of the techniques we evaluated for generating graphs and present the reasons behind our shifts in strategies. We further present some generated graphs using our approach for different input graphs.

A.1 Explored Methods: First

This method introduces the idea of adapting the Wave Function Collapse (WFC) algorithm to graph generation by defining tiles as subgraphs. A tile represents a local subgraph centered around a *spanning vertex* and includes its k-hop neighbors. Each tile encodes the local structure. We further define *open ends* of a tile as the set of external connections that it's vertices had with nodes outside of the subgraph. These open ends define how we connect the tiles to generate a new graph.

To connect a tile to another (via its open ends), the first approach was to consider all tiles as a possibility. A better approach (intuitively) is to determine the set from the input graph by looking at what vertices neighbor our tile's subgraph. This approach introduces the necessity of labeling the input graph nodes beforehand because this information is now necessary to define the external connections of a tile. To label our input graph we run the WL-Algorithm for node labelling discussed in chapter 3.

We define as *End Identifier* the series of node labels going from the Tile's Spanning vertice through the shortest path vertices until the open end.
For two tiles to connect, the following conditions must hold: - The two tiles must have an open end.
- The End Identifier from the first tile's open end, must match the reverse of the End Identifier from the second tile's open end.
For example, the Tile 2 and Tile 4 in figure A.7 can connect, because:

- Tile 2 has an open end with an End Identifier: 2, 3, 4
- Tile 4 has an open end with an End Identifier: 4, 3, 2

A.1.1 Limitation

The problem with this appraoach is that it can result in higher node degrees than those observed in the input graph. For example in figure A.7, merging tile 1 with tile 3 via their open end labeled 3 would result in higher node degree for node 2 (degree 4 instead of 3).

A.2 Explored Methods: Second

We now don't define open ends with what tile they are looking for, but rather with what internal node in the tile has the open end. This internal node will be merged when we connect the tile. This means that we merge differently, and don't make use of End identifiers anymore.Two tiles can be connected if they share the same open end.

While this resolves the issue of artificially increasing node degrees, it introduces a new limitation: the loss of local structural consistency. Ideally, the generated graphs should preserve the local structure of the original dataset while allowing for global reorganization.

To address this, we further refine the method and define open nodes based on their local neighborhood. Specifically, each open node is characterized by:

- Its own label (e.g. figure A.12 , "1").
- A list of its neighboring node labels in the input graph (e.g., "[3,3,2,2]").
- A distinction between neighbors it offers (that are part of the tile, e.g. "[3, 3]") and those it expects to receive during tile merging (that are not part of the tile, e.g. "[2, 2]").

Connection Criteria: For two tiles to merge, they must satisfy the following conditions:

1. Their open nodes must have the same label.
2. The merging process must ensure compatibility between what each tile **offers** and what the other **looks for**.

The decision process follows these rules:

- If the number of offered connections, for a certain label, exceeds what is **looked for**, the merge is not possible.
- If the number of **offered** connections exactly matches what is **looked for**, the merge is valid, and no open ends remain from the merge.

- If the number of **offered** connections is less than what is looked for, the merge is still **valid**, but a new open end is created to account for the unmet demand.

This approach ensures that local structures are better preserved, and can not result in higher degree nodes in the generated graph.

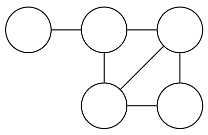


Figure A.1: Input graph

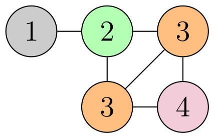


Figure A.2: Labeled input graph using WL on the 1-hop neighborhood

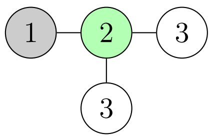


Figure A.3: Tile 1

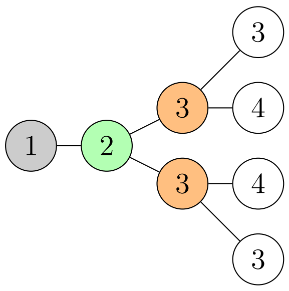


Figure A.4: Tile 2

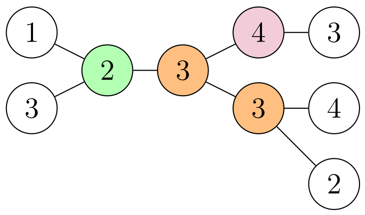


Figure A.5: Tile 3

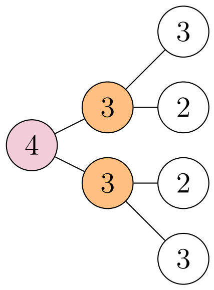


Figure A.6: Tile 4

Figure A.7: All tiles generated from an input graph using the first explored method. The uncolored nodes in the tiles correspond to open ends. The tile number corresponds to the spanning node's label.

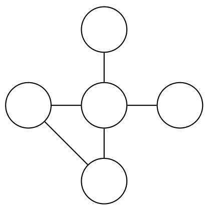


Figure A.8: Input graph

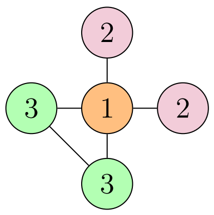


Figure A.9: Labeled input graph using WL on the 1-hop neighborhood

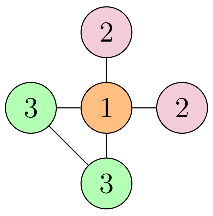


Figure A.10: Tile 1

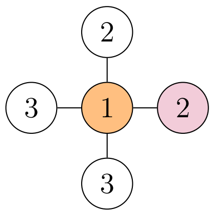


Figure A.11: Tile 2

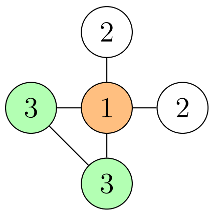


Figure A.12: Tile 3

Figure A.13: All tiles generated from an input graph using the second method. The uncolored nodes in the tiles correspond to what the open node is looking for. The tile number corresponds to the spanning node's label.

APPENDIX A

Appendix B

A.1 Visualizations

We show some generated graphs using both synthetic (handmade) graphs and real protein graph representations [12]. We limit the inputs to graphs with low node degrees, because of a termination problem for $k = 1$, and an exploding precomputation problem for $k \geq 2$. Moreover, we always set $size = 1$ when generating graphs with relatively higher node degrees (e.g. for all protein graphs).

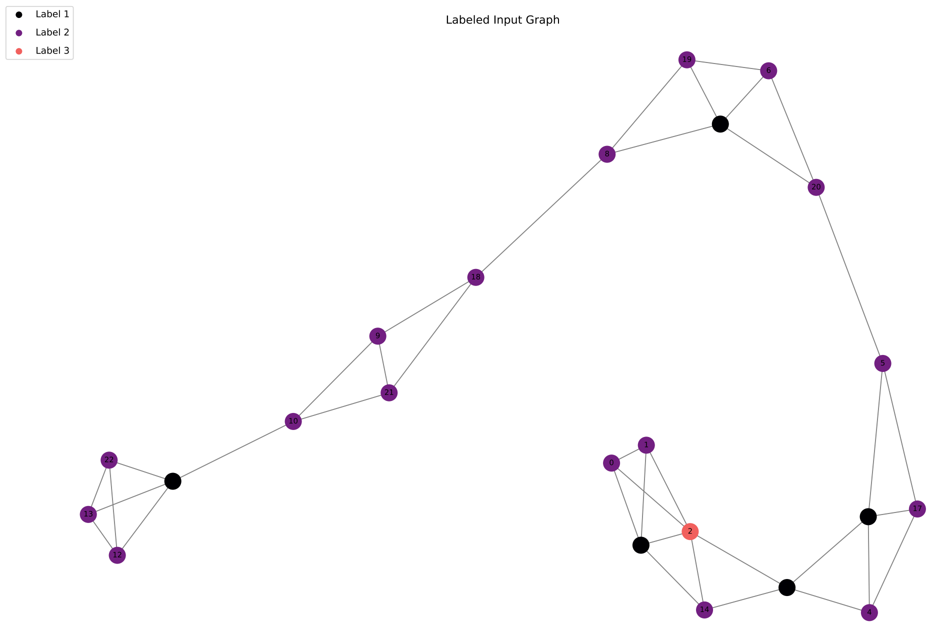


Figure A.1: Protein 1 labeled input graph. $(k = 1)$

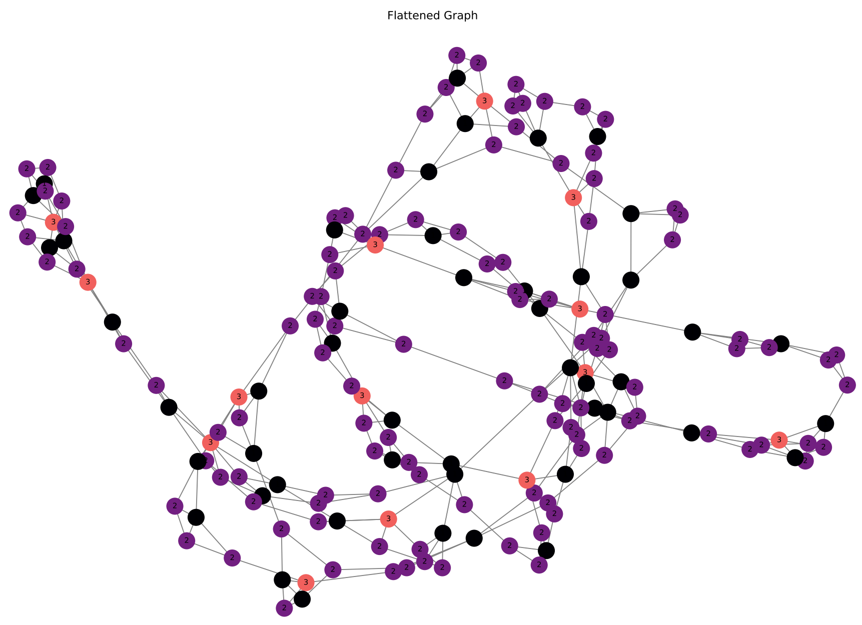


Figure A.2: Protein 1 output graph. ($k = 1$)

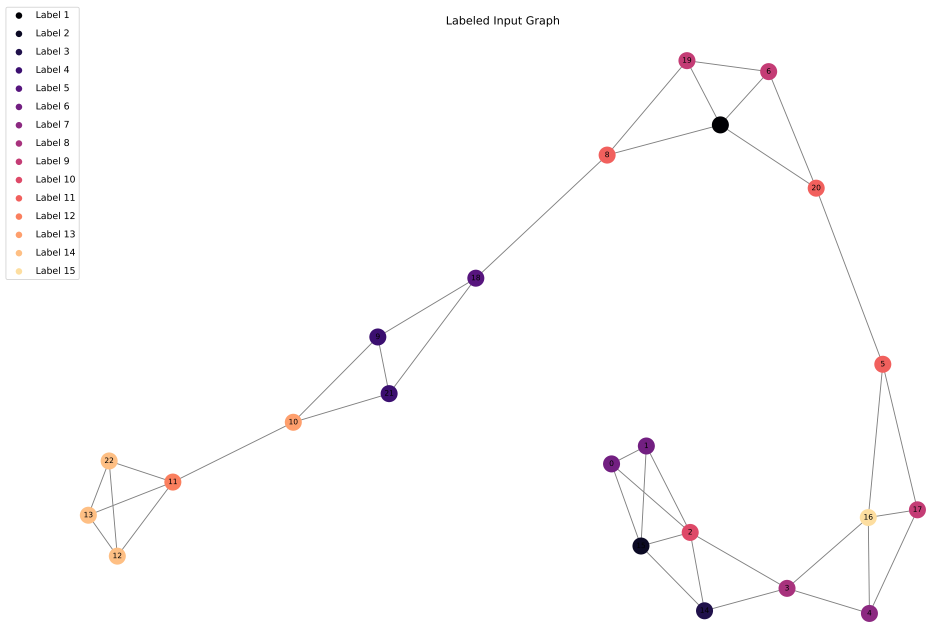


Figure A.3: Protein 1 labeled input graph. ($k = 2$)

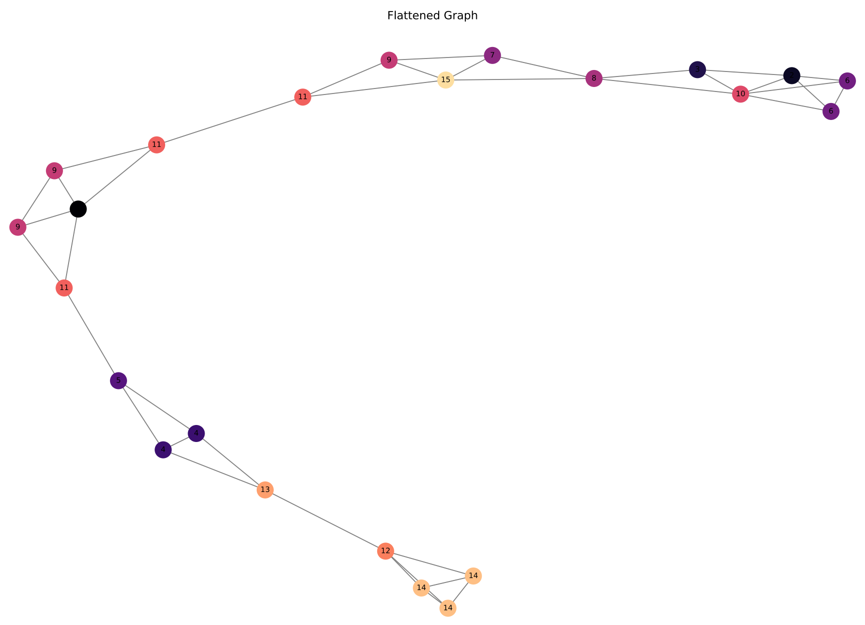


Figure A.4: Protein 1 output graph. $(k = 2)$

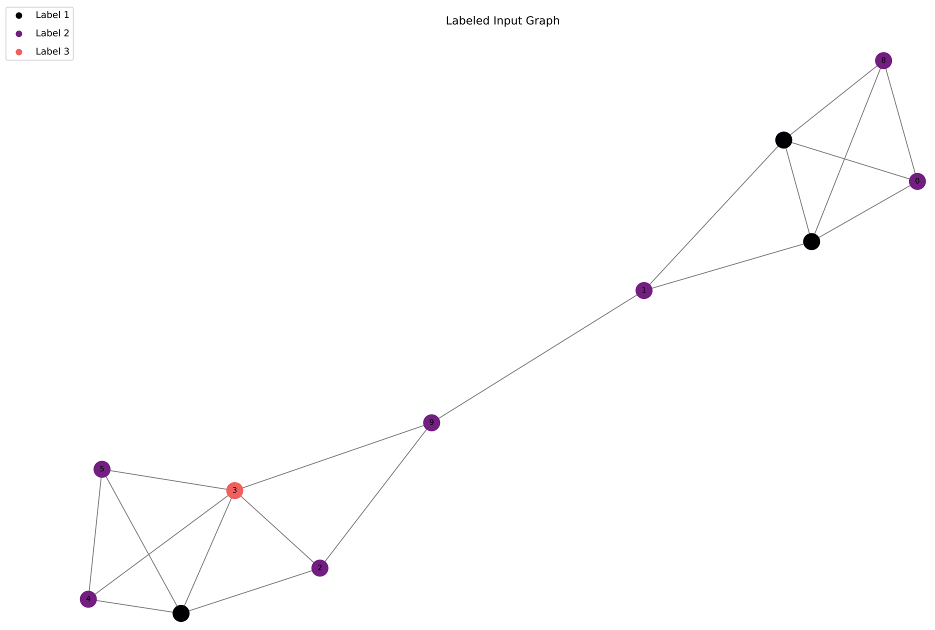


Figure A.5: Protein 2 labeled input graph. $(k = 1)$

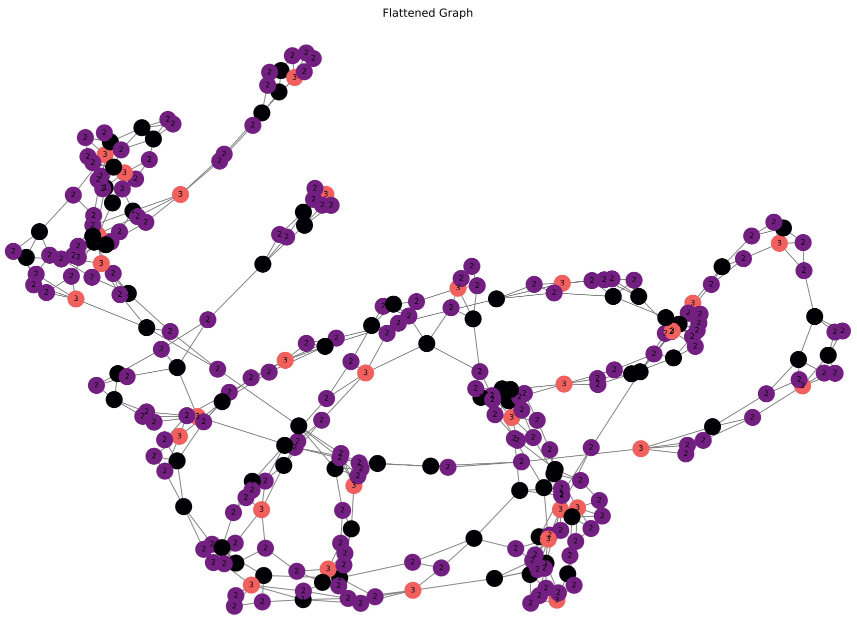


Figure A.6: Protein 2 output graph. ($k = 1$)

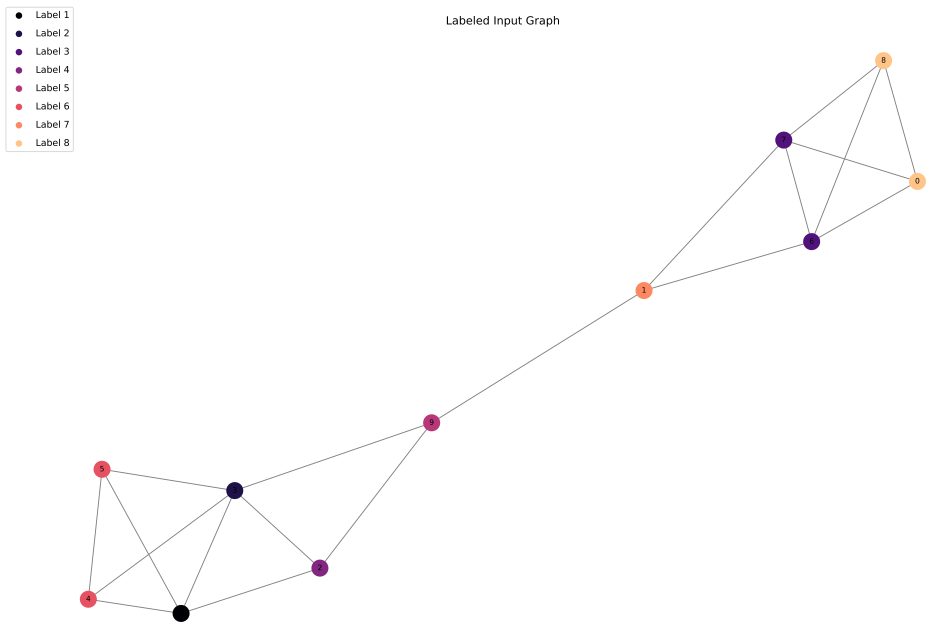


Figure A.7: Protein 2 labeled input graph. ($k = 2$)

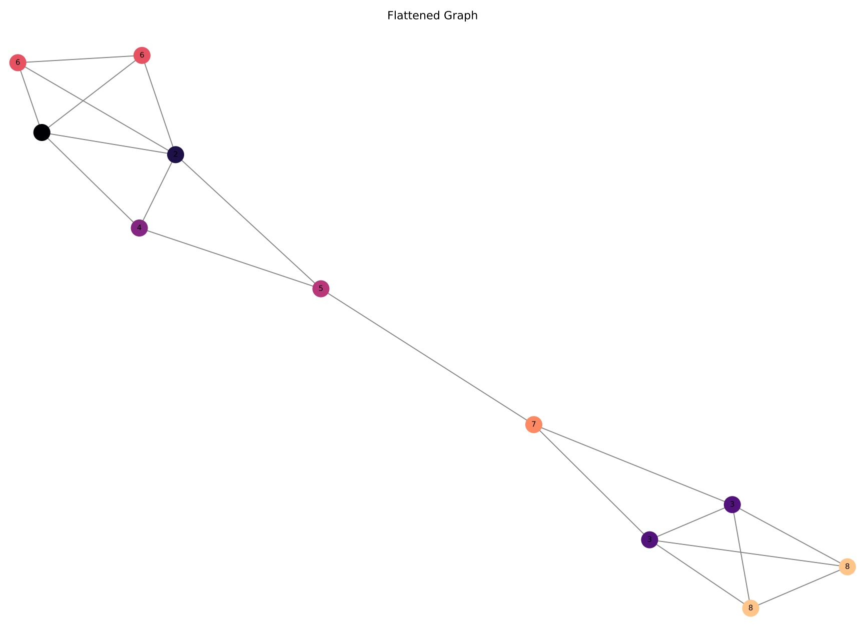


Figure A.8: Protein 2 output graph. ($k = 2$)

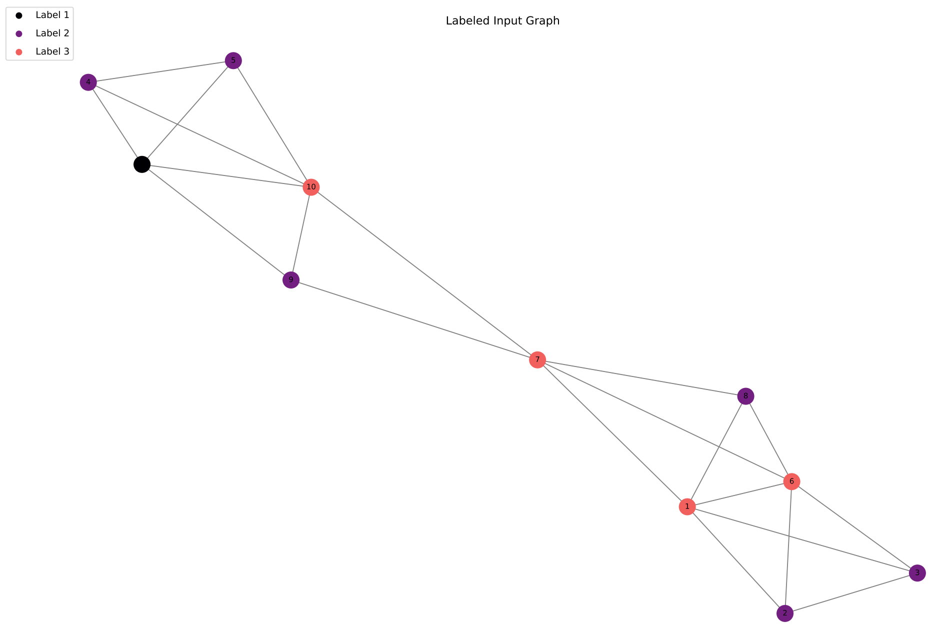


Figure A.9: Protein 3 labeled input graph. ($k = 1$)

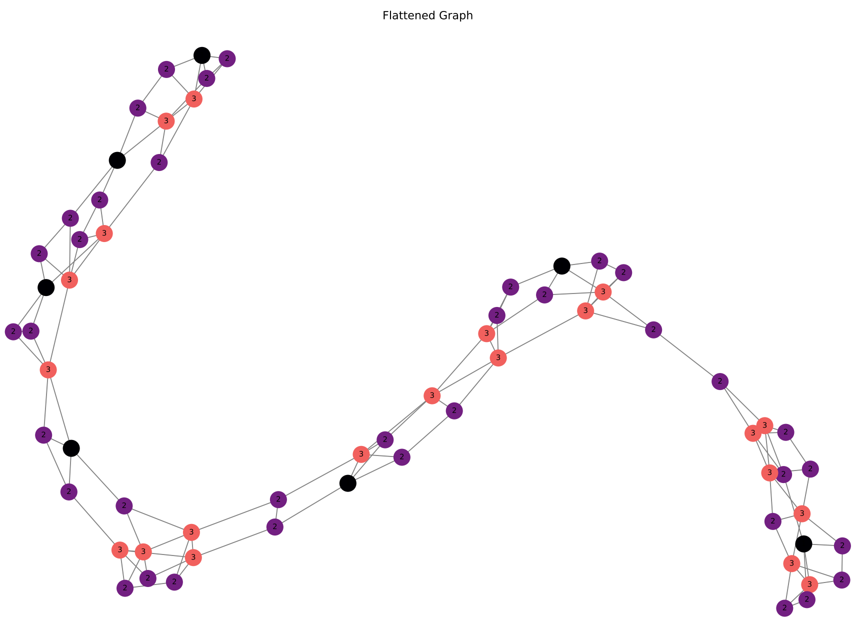


Figure A.10: Protein 3 output graph. ($k = 1$)

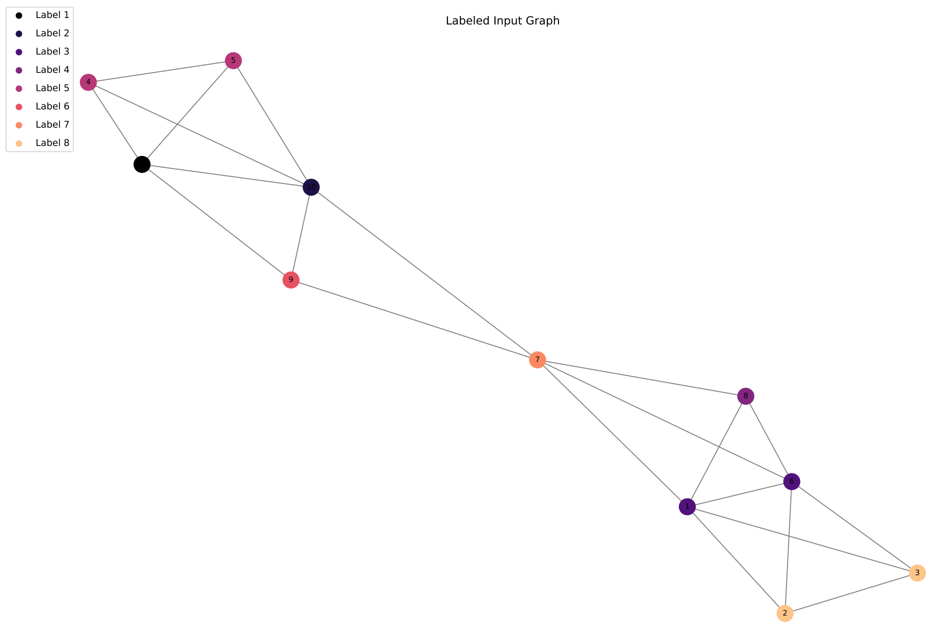


Figure A.11: Protein 3 labeled input graph. ($k = 2$)

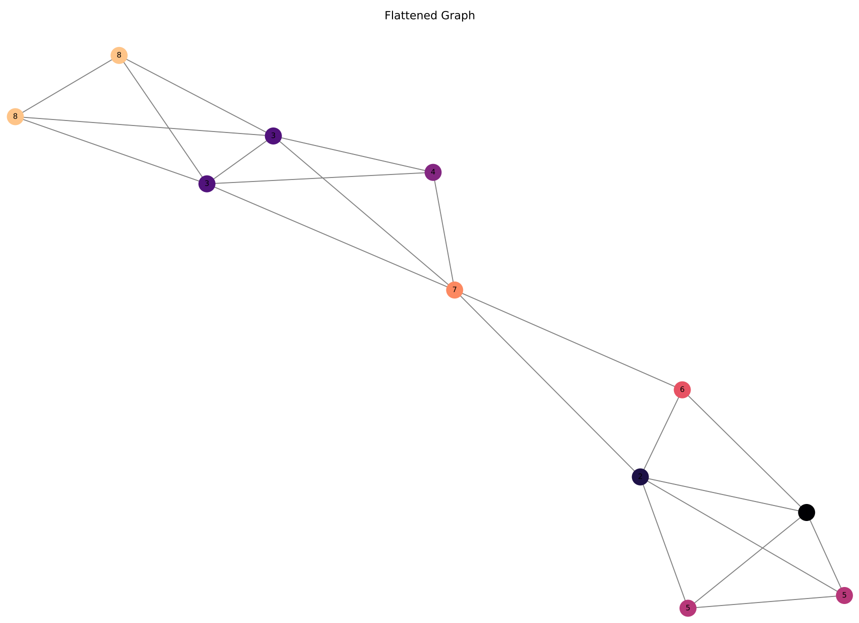


Figure A.12: Protein 3 output graph. $(k = 2)$

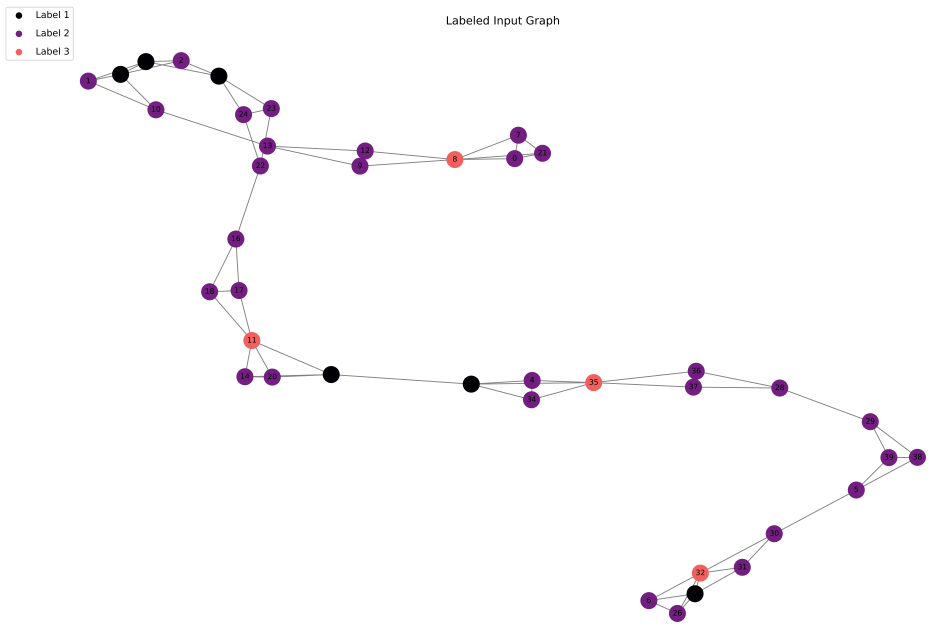


Figure A.13: Protein 4 labeled input graph. $(k = 1)$

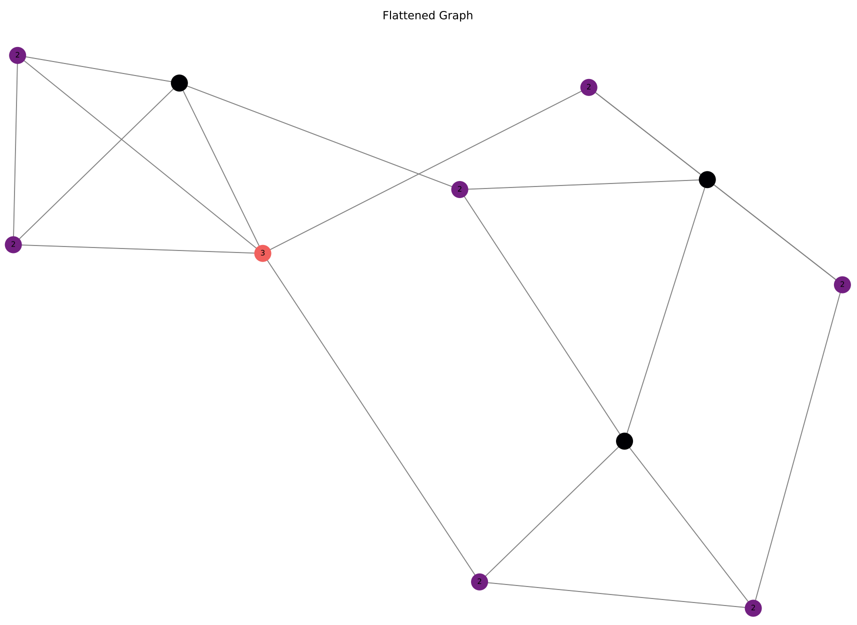


Figure A.14: Protein 4 output graph. ($k = 1$)

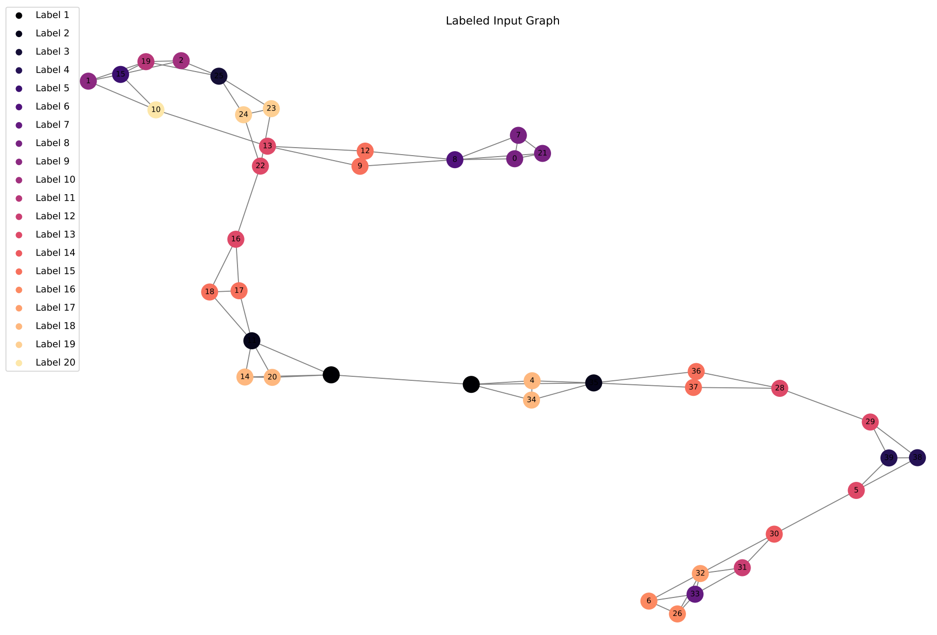


Figure A.15: Protein 4 labeled input graph. ($k = 2$)

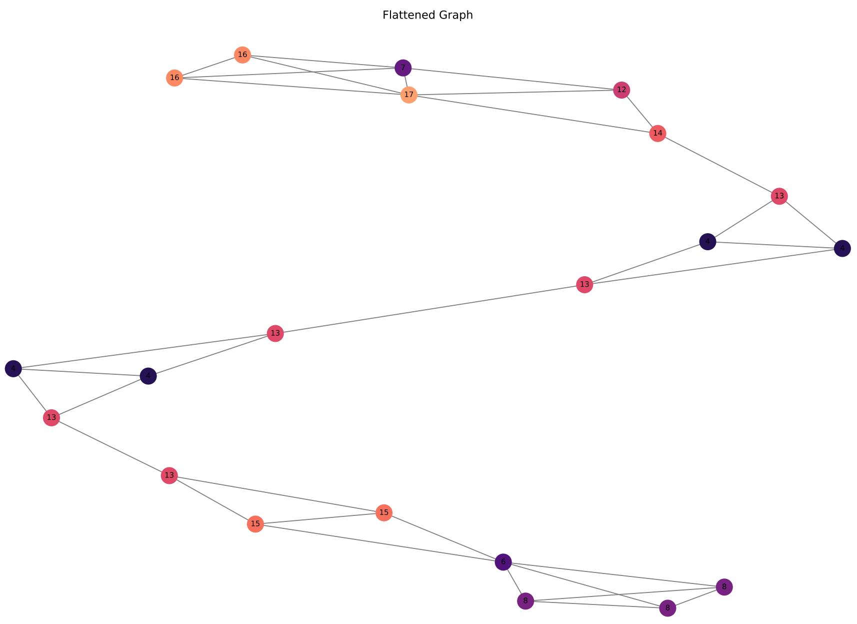


Figure A.16: Protein 4 output graph. ($k = 2$)

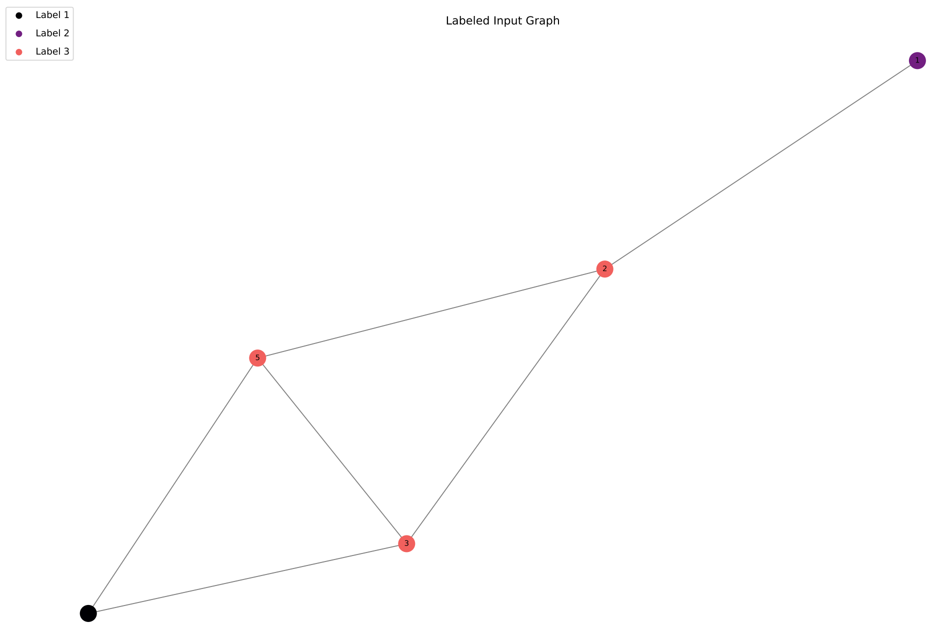


Figure A.17: Synthetic 1 labeled input graph. ($k = 1$)

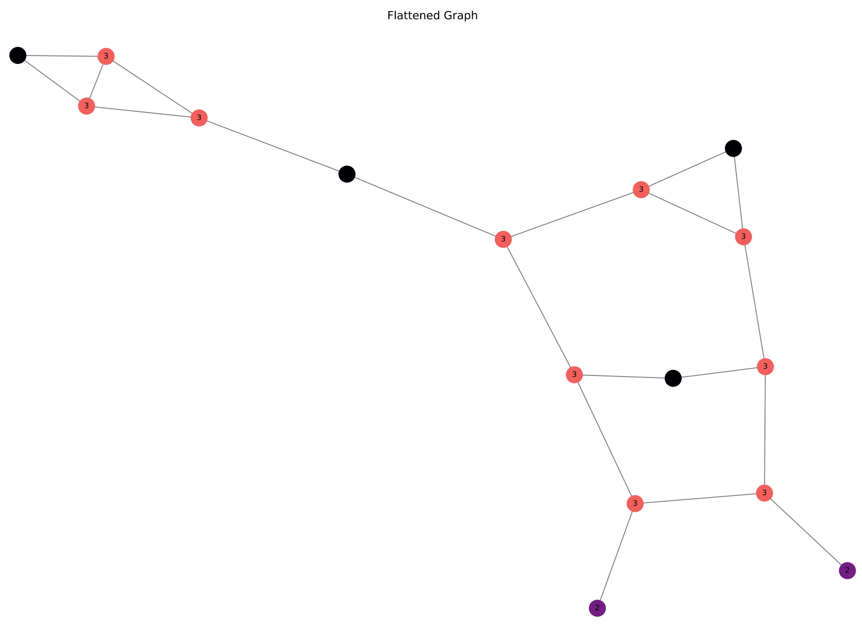


Figure A.18: Synthetic 1 output graph. $(k = 1, size = 10)$

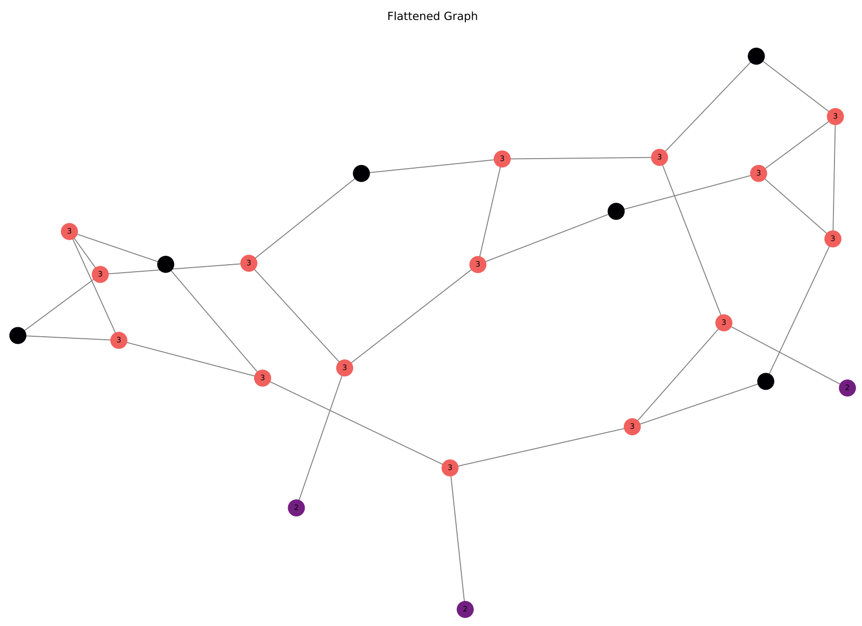


Figure A.19: Synthetic 1 output graph. $(k = 1, size = 20)$

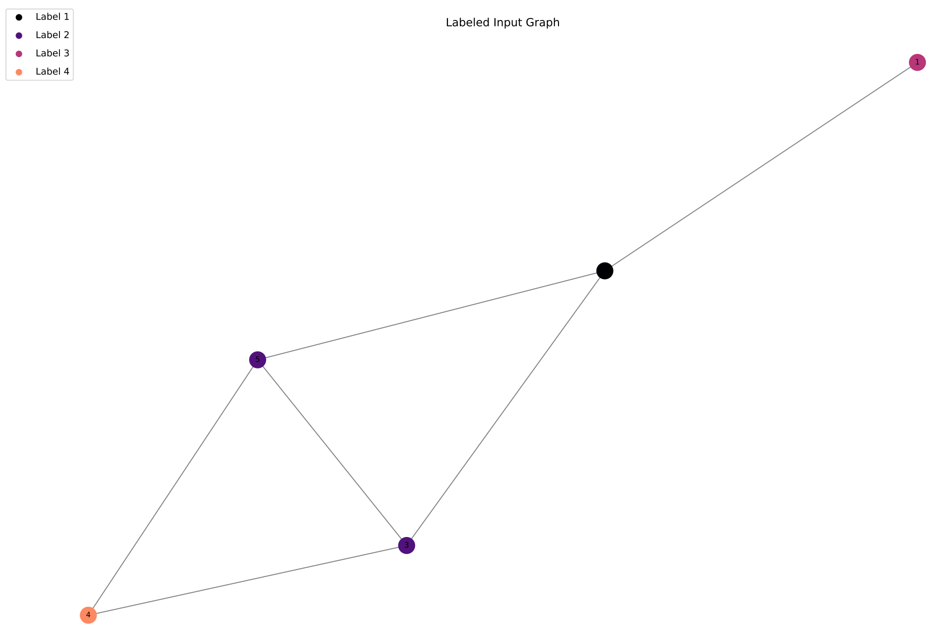


Figure A.20: Synthetic 1 labeled input graph. ($k = 2$)

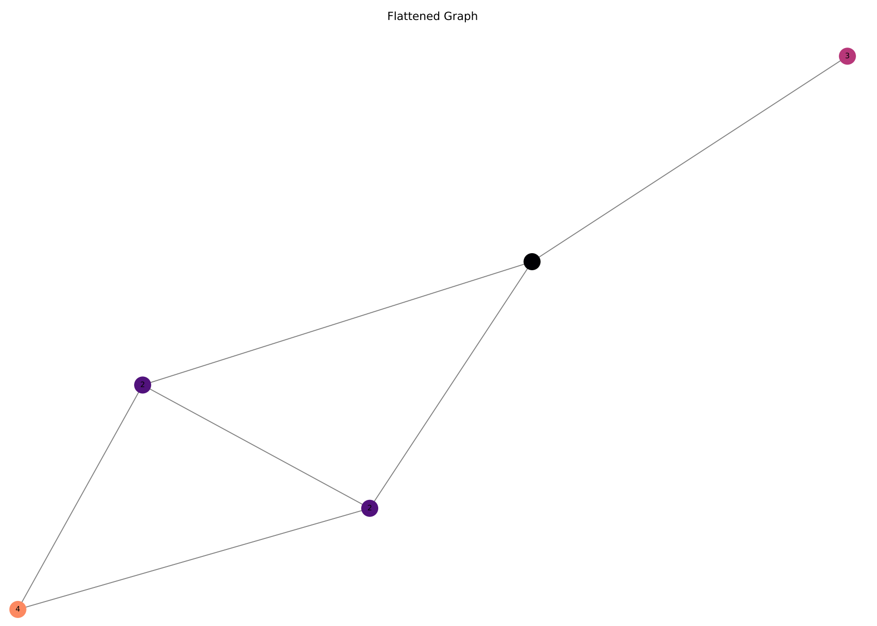


Figure A.21: Synthetic 1 output graph. ($k = 2, size = 1$)

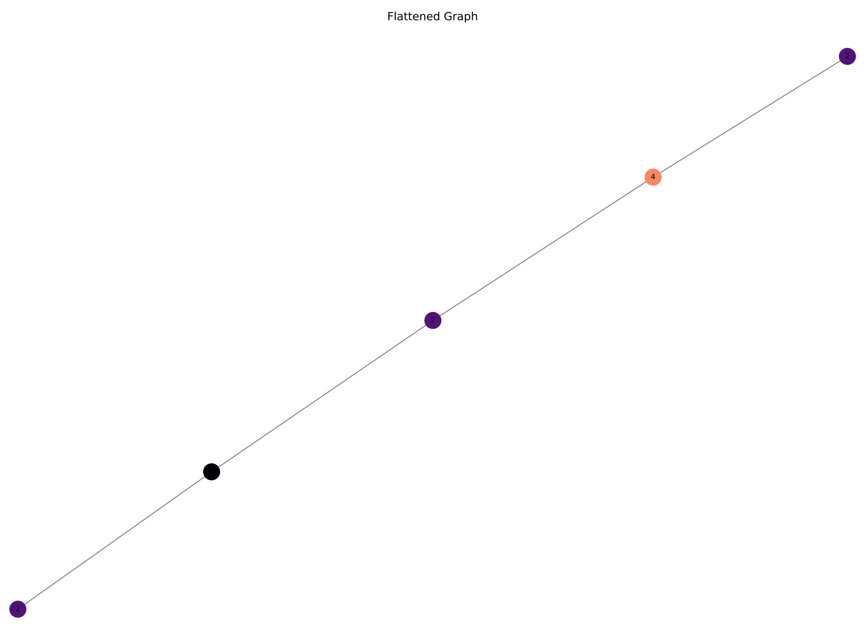


Figure A.22: Synthetic 1 output graph. It fails because backtracking isn't implemented.$(k = 2, size = 10)$

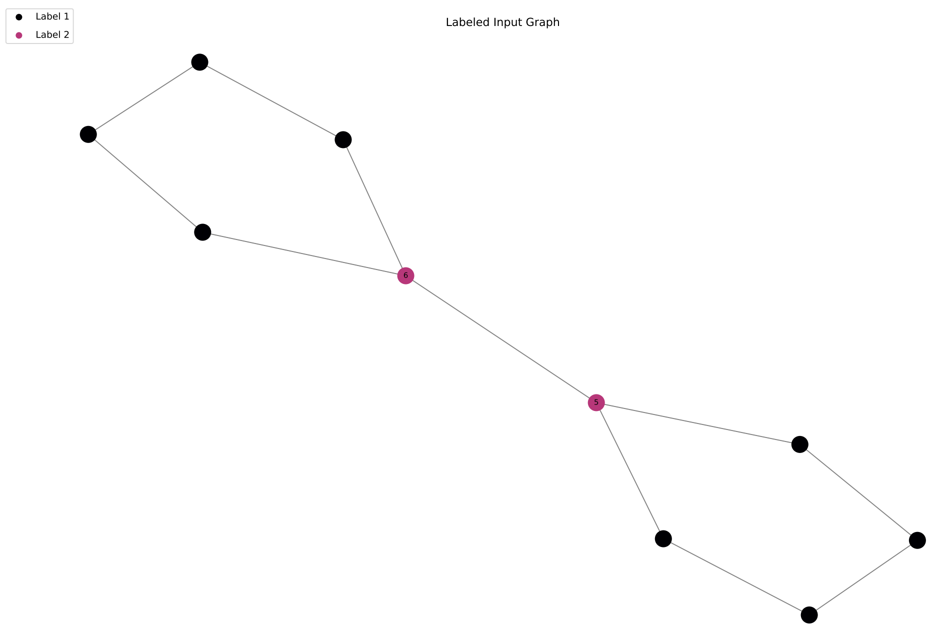


Figure A.23: Synthetic 2 labeled input graph. $(k = 1)$

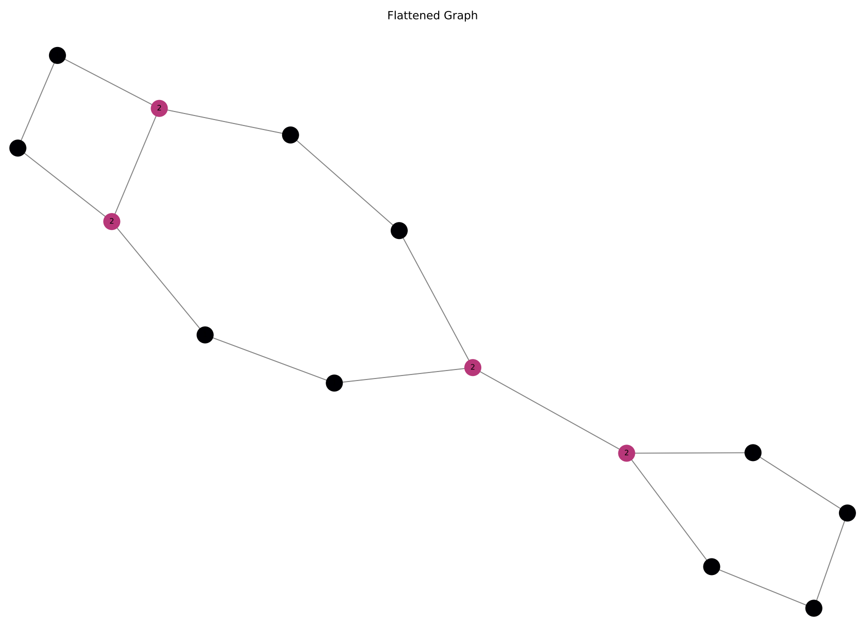


Figure A.24: Synthetic 2 output graph. $(k = 1, size = 10)$

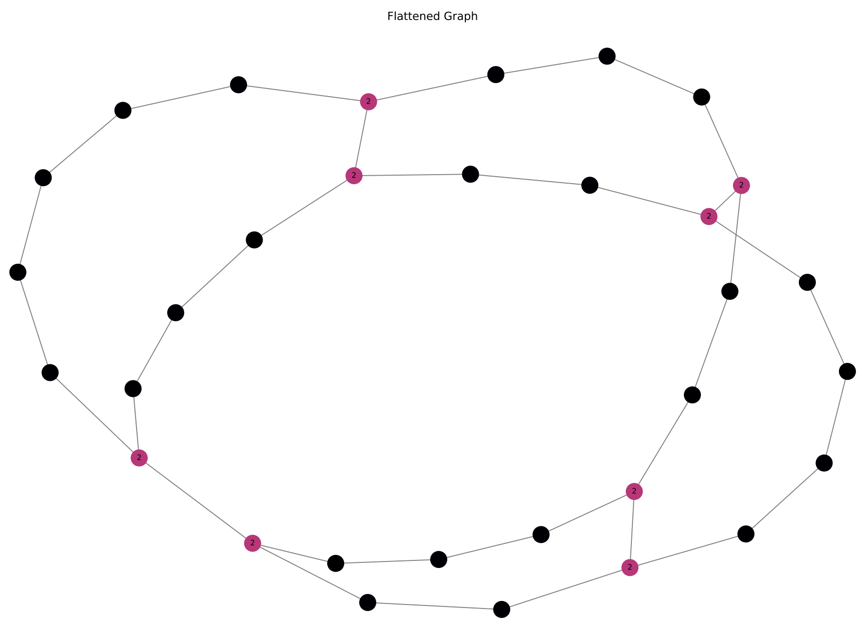


Figure A.25: Synthetic 2 output graph. $(k = 1, size = 20)$

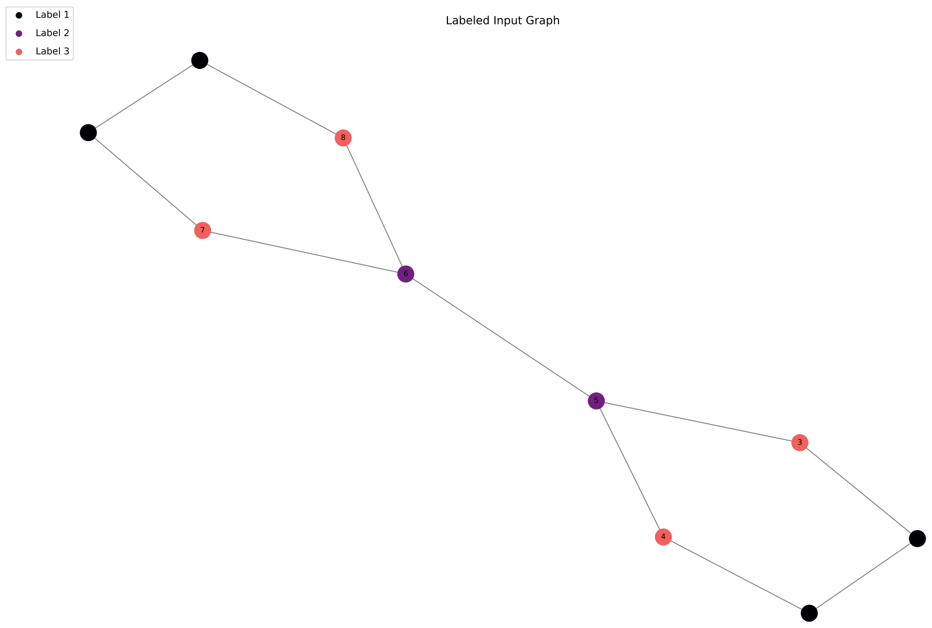


Figure A.26: Synthetic 2 labeled input graph. $(k = 2)$

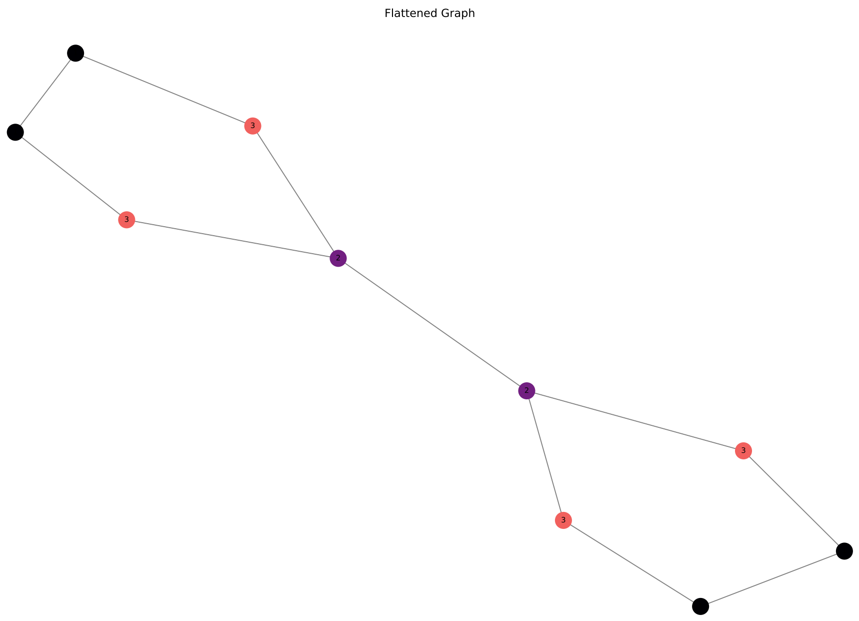


Figure A.27: Synthetic 2 output graph. $(k = 2, size = 1)$

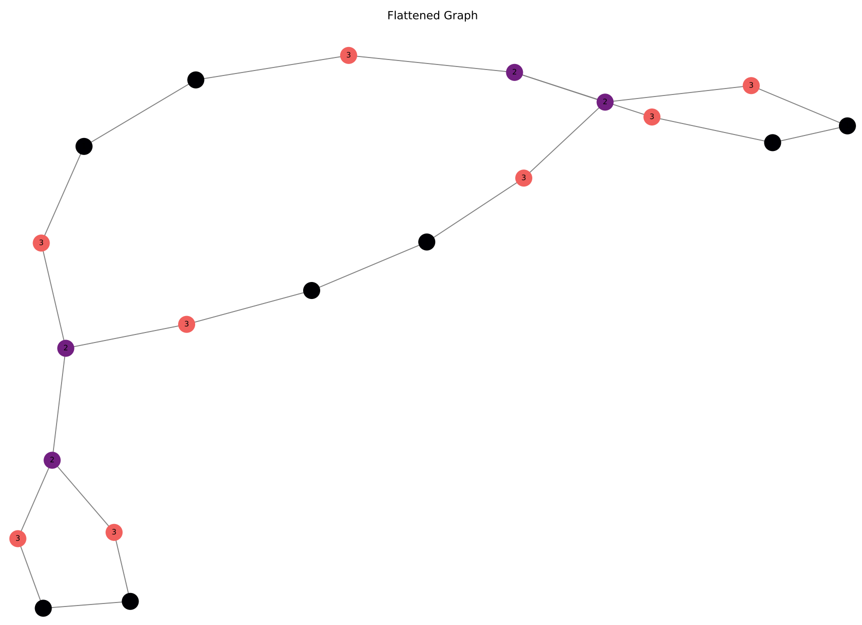


Figure A.28: Synthetic 2 output graph. $(k = 2, size = 10)$

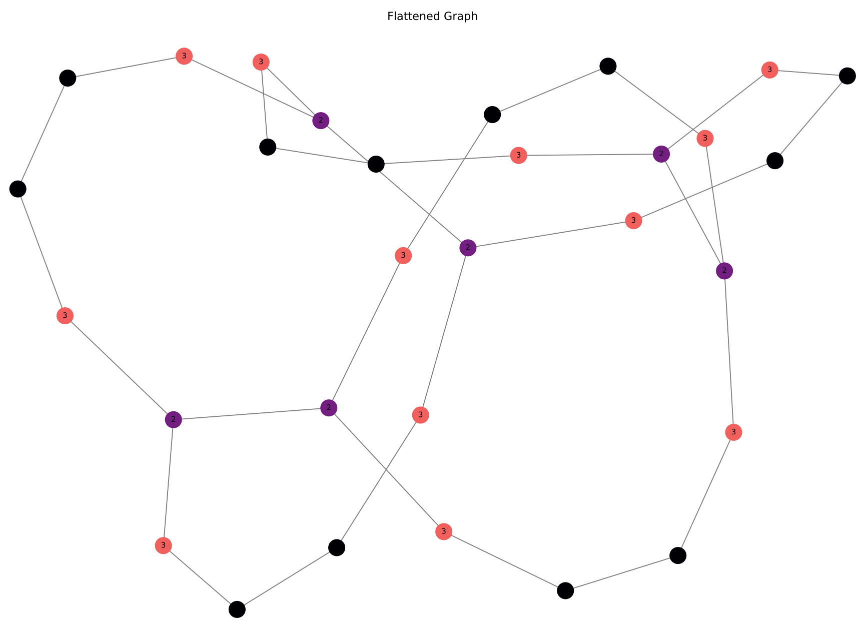


Figure A.29: Synthetic 2 output graph. $(k = 2, size = 20)$

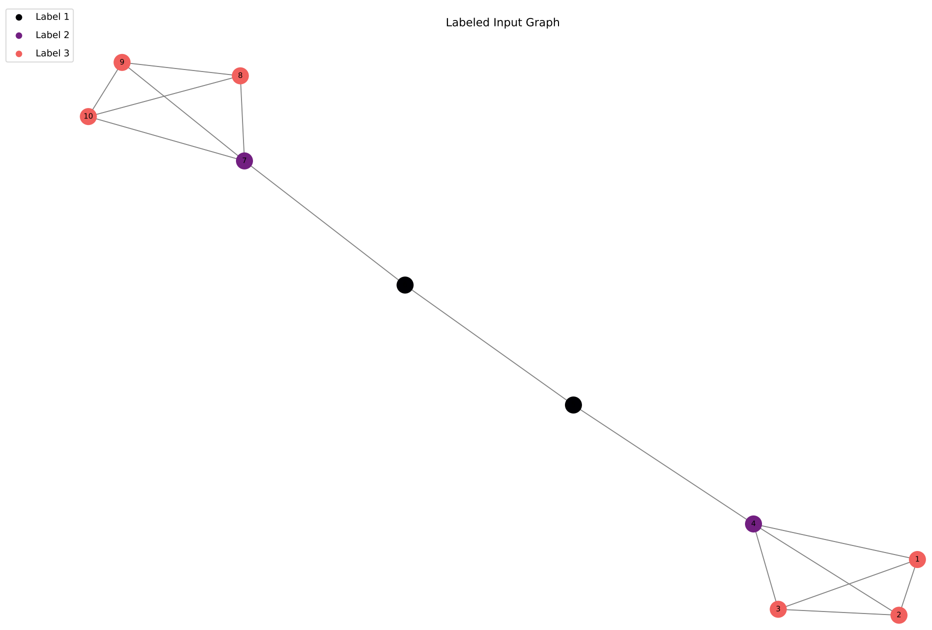


Figure A.30: Synthetic 3 labeled input graph. $(k = 1)$

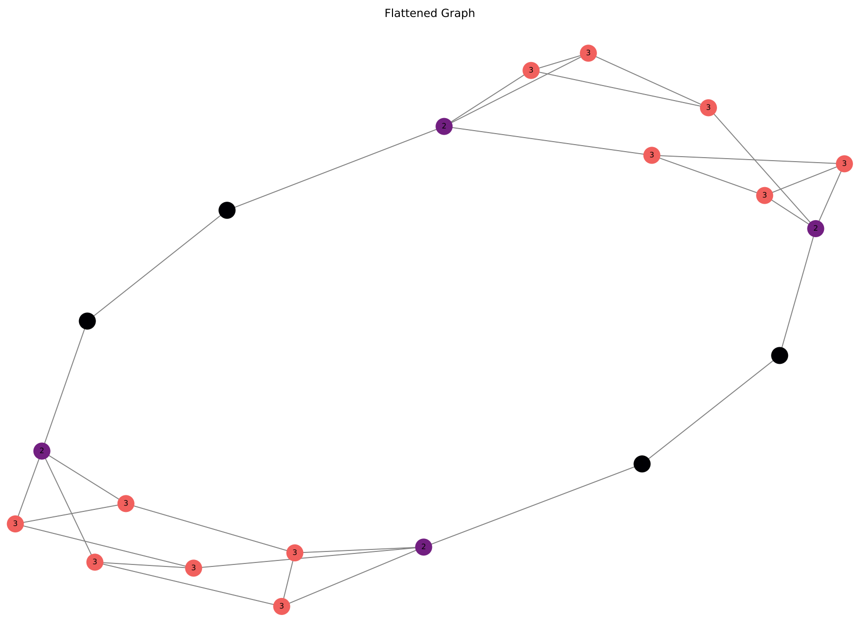


Figure A.31: Synthetic 3 output graph. $(k = 1, size = 10)$

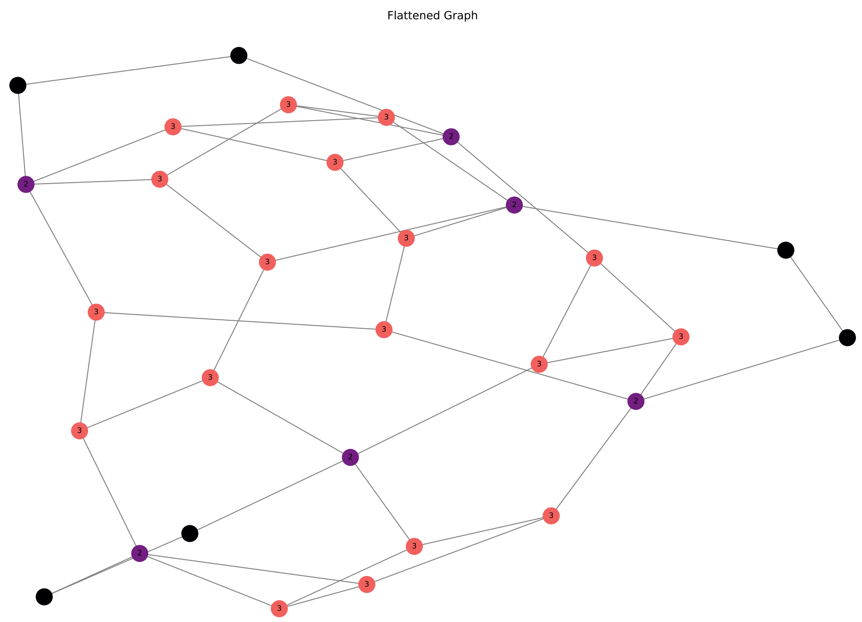


Figure A.32: Synthetic 3 output graph. ($k = 1, size = 20$)

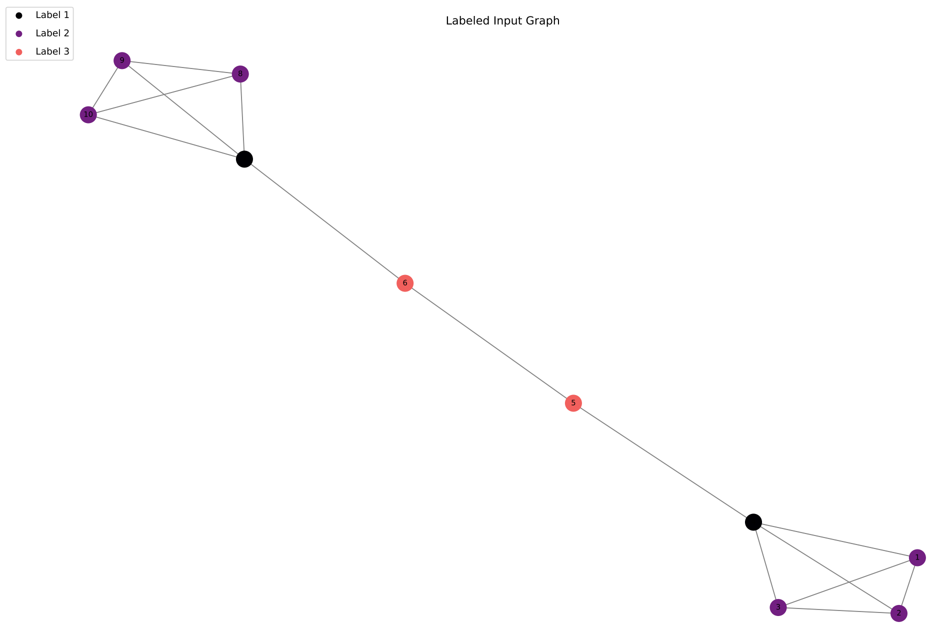


Figure A.33: Synthetic 3 labeled input graph. ($k = 2$)

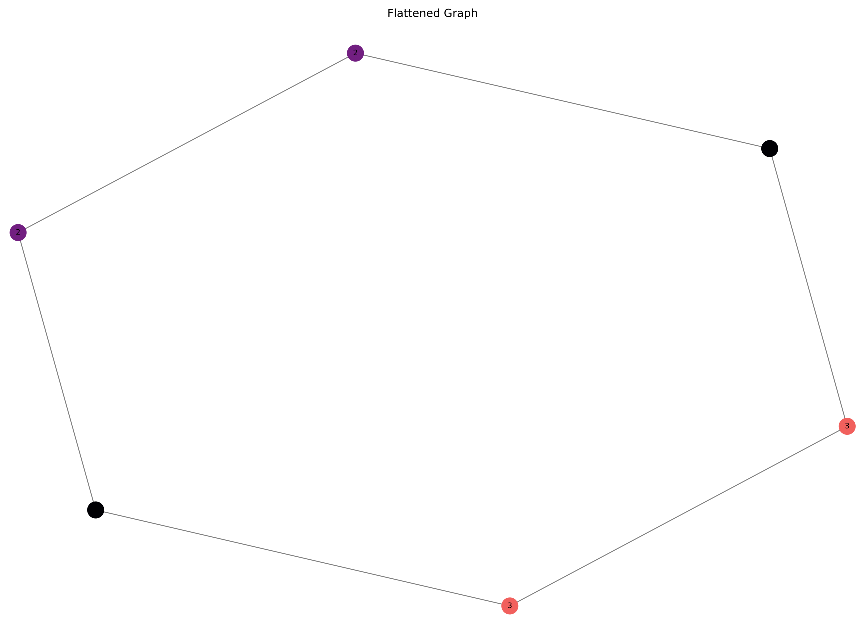


Figure A.34: Synthetic 3 output graph. ($k = 2, size = 1$)

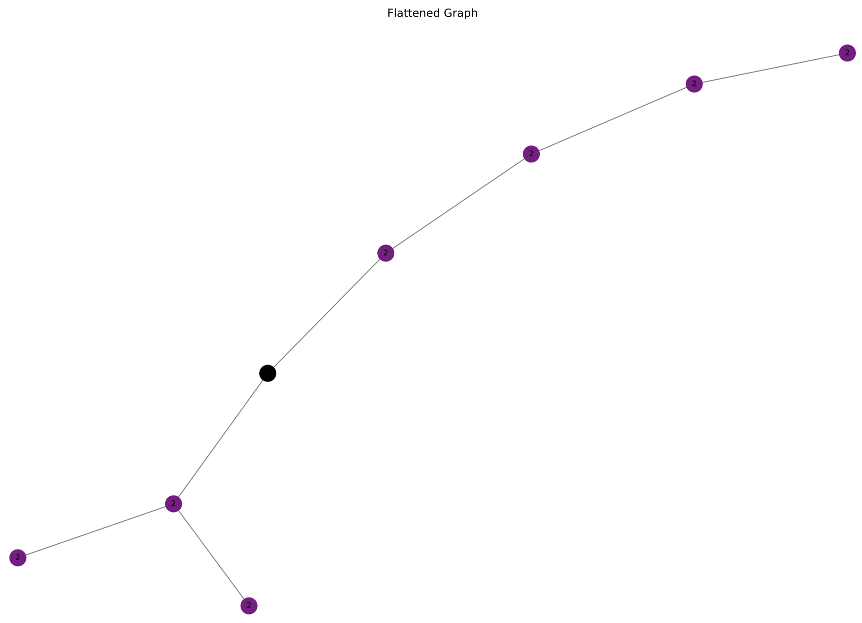


Figure A.35: Synthetic 3 output graph. It fails because backtracking isn't implemented.($k = 2, size = 10$)