Data Visualization Techniques for relational and high dimensional data

Ziv Hochman - 8454434, Stylianos Psara - 2140527 , Panagiotis Andrikopoulos - 1780743

April 10th, 2024

1 Annotations

- **Vertex:** A fundamental element in a graph representing an entity or a point.
- **Edge:** A connection between two vertices in a graph, denoting a relationship or interaction between them.
- **Graph:** A mathematical structure composed of a set of vertices and a set of edges that connect these vertices.
- **Directed Graph:** It is a type of graph in which edges have a specific direction from one vertex to another.
- **Undirected Graph:** A type of graph in which edges do not have a specific direction. For example, if we have the vertices V and U and an edge between them, we can go from V to U and from U to V.
- **Parent Vertex:** In a directed graph, a vertex that has an outgoing edge to another vertex is termed the parent vertex of the latter.
- **Child Vertex:** In a directed graph, a vertex that has an incoming edge from another vertex is termed the child vertex of the former.
- **Time Complexity:** A measure of the amount of time required for an algorithm to execute as a function of the size of its input.
- **Space Complexity:** A measure of the amount of memory space required by an algorithm to execute as a function of the size of its input.
- **Crossing resolution:** The smallest angle from crossing edges in a graph

2 Introduction

In today's age of massive data growth, it's more important than ever to derive valuable insights from information. Yet, the sheer amount and complexity of data make understanding them a significant challenge. Data visualization becomes essential, allowing us to simplify complex datasets into easy-to-understand visuals.
Nevertheless, effective data visualization has challenges. As datasets become complex, so does the challenge of constructing meaningful visual representations.
Our paper aims to address these challenges by implementing novel methodologies from scratch to visualize relational and high-dimensional datasets and empower stakeholders with deeper insights.
The remainder of the report is structured as follows: In Section 3, we delve into the process of reading and graphically representing data in a circular way (step 1). Section 4 addresses the extraction and visualization of trees, offering insights into this specific data structure (step 2). Moving on to Section 5, we explore the computation of a force-directed layout, an important technique in graph visualization (step 3). In Section 6, our focus shifts to the computation of a layered graph, where we investigate the organization of data into distinct layers (step 4). Section 7 is dedicated to the exploration of Multi-layer/clustered graphs and edge bundling techniques (step 5). The subsequent section, Section 8, delves into the presentation of graph projections, outlining different approaches (step 6). Following this, in Section 9, we introduce metrics used for assessing the quality of graph projections, providing valuable insights into the effectiveness of visualization techniques (step 7). Section 10 offers concluding remarks summarizing the report. Finally, in Section 11, we provide a link to access the code for all the steps discussed, facilitating further exploration and implementation.

2.1 Software and Libraries

Our software is being developed using Python 3.1 as the primary programming language, and we are utilizing Jupyter Notebook with Anaconda as our coding environment. To implement our algorithms and data structures, we are using the following libraries:

- numpy
- pandas
- matplotlib.pyplot
- matplotlib.colors
- pydot
- time
- math
- copy
- itertools
- sklearn.manifold
- sklearn.metrics.pairwise
- scipy.stats
- matplotlib.collections

3 Step 1: Read and draw a graph

In this step, initially, we import datasets from dotfiles into our program. Then, we proceed to implement the data structures of our classes, including Vertex, Edge, and Graph. Within the Graph class, we implement graph visualization using the function "visualize_graph" and calculate the graph using the function "place_vertices_in_circle". Subsequently, we create a function to convert the generated graph from the dot file into our custom graph format.
Finally, we visualize the original graph and then the graph for the bonus part, following the explanations.

3.1 Classes

3.1.1 Vertex

This class contains the following attributes:

1. *id*: Each vertex has its unique identifier.
2. *coordinates*: Represents the (x, y) coordinates of the vertex on a grid plot.
3. *neighbors*: A list that contains all the neighboring vertices of the current vertex, i.e., all vertices that share an edge with the current vertex.

And the following functions:

1. **"add_neighbor" Function**: This function appends a new neighbor to the list of neighbors for the current vertex.

3.1.2 Edge

This class contains two attributes: *start*: Denotes the starting vertex of the edge.
end: Denotes the ending vertex of the edge.

For example, in a directed graph G, if we have vertex u and vertex v with an edge between them, we denote the edge as {u,v}, where :

1. start = u.
2. end = v.

If we were dealing with an undirected graph, we would construct it as a directed graph, assigning each vertex pair a "start" and "end" vertex. However, during visualization, we would treat the graph as undirected by plotting all edges without directionality. For instance, if an edge exists between vertices V and U, implying bidirectional connectivity, there's no necessity to duplicate this information in our database. Instead, we would store only one instance of the edge, specifying its start as V and end as U. During visualization, we'd represent this edge as bidirectional. Hence, the terms "start" and "end" wouldn't signify the beginning and end points but instead points 1 and 2 of an edge. This method reduces storage requirements and simplifies time complexity during operations as the algorithm iterates through fewer edge objects.

3.1.3 Graph

This class contains 2 attributes:

1. *vertices* - denotes the dictionary of all the vertices in the current graph.
2. *edges* - denotes the list of all the edges in the current graph.

In addition, this class contains 7 functions (2 for the bonus part that will be explained in the last part) for managing vertices and edges, as well as for visualizing the graph.

1. **__init__**: we are initializing two empty sets, vertices, and edges.
2. **"add_vertex" AND "add_edge" Functions**: these functions add vertices and edges to the graph, respectively.
3. **"get_vertex" Function**: this function retrieves a vertex from the graph based on its ID.
4. **"visualize_graph" Function**: this function plots the graph using Matplotlib. It first places the vertices in a circular layout using the place_vertices_in_circle method, then plots each vertex as a blue dot with its ID labeled on top. The edges are plotted as grey lines connecting the start and end vertices of each edge.
5. **"visualize_bonus_graph" Function**: this function is an extended version of the visualization function. It computes the average number of neighbors for vertices in the graph and adjusts the layout accordingly using the place_vertices_in_circle_Bonus method. The average number of neighbors is printed for reference. The rest of the visualization process remains the same as in visualize_graph.
6. **"place_vertices_in_circle" Function**: this function arranges the vertices in a circular layout. It calculates the position of each vertex based on its index and the total number of vertices, placing them evenly around a circle.
7. **"place_vertices_in_circle_Bonus" Function**: this function is an extended version of the vertex placement function. It adjusts the radius of the circle based on the average number of neighbors of the vertices. Vertices with more neighbors than the average have a smaller radius, while those with fewer neighbors have a larger radius.

Overall, the Graph class provides functionality for managing and visualizing graph data structures efficiently.

3.2 Creating the graph

In the function "create_custom_graph", the data structure of the graph is constructed. Initially, vertices and edges are parsed from the generated graph into lists. Subsequently, each vertex is initialized with its ID and default coordinates. Finally, for each tuple representing an edge (source_vertex, neighbor_vertex), neighbors are added for each source vertex, and the list of edges is appended to the Graph class.

3.3 Complexity of our layout algorithm

First, we converted the dot file into a graph representation. To work effectively with this graph, we designed a custom structure that is more suitable for our requirements by using the function "create_custom_graph" which takes the generated graph and converts it into our graph. Subsequently, our algorithm proceeds as follows:
We iterate through all vertices " Vi " of the graph and plot them in a circular format See *Figure* 1 where the x and y coordinates of each vertex are calculated as follows:

$$x = \text{center}_x + \text{radius} \times \cos(\text{angle})$$
$$y = \text{center}_y + \text{radius} \times \sin(\text{angle})$$

Where center_x, center_y is the point (0,0), the radius is the distance from the center, and the "angle" which contributes to the distribution of the vertices around the circle is computed as follows:

$$\text{angle\_increment} = 2 \times \pi / \text{num\_vertices}$$
$$\text{angle} = i \times \text{angle\_increment}$$

Where *num_vertices* is the number of the vertices in the graph.

This process has a time complexity of $O(V)$ (due to the iteration through all the vertices Vi). Next, we scan through the edges of the graph. For each edge, we draw a line between the corresponding pair of vertices. This step has a time complexity of $O(E)$ since it involves iterating over all edges. Therefore, the total **time complexity** of our layout algorithm is $O(V + E)$, where V represents the number of vertices and E represents the number of edges in the graph.
In addition, the **space complexity** is $O(V + E)$:

- The class "Edge" is $O(1)$, because it only contains simple variables.
- The class "Vertex" is $O(V)$, because it contains a list of vertices, which in the worst case can contain as many vertices as the total vertices - 1. Which is $O(V)$, where V denotes the number of vertices in the graph.
- The class "Graph" is $O(V+E)$ because we have a list and dictionary, the dictionary contains all the vertices ($=O(V)$) and the second list contains all the edges ($=O(E)$), therefore, the total space complexity is $O(V + E)$, where V represents the number of vertices and E represents the number of edges in the graph.

The total space complexity is $O(V + E + V + E) = O(V + E)$

3.4 Pro's and Con's

Pro's:

- Our algorithm performs well in terms of time complexity and constructs good visualizations in small and medium size datasets (e.g. noname.dot, LeagueNetwork.dot), by representing the graphs in a circular structure.
- Our algorithm avoids overlapping vertices, which is useful, especially in large datasets (e.g. JazzNetwork.dot). We achieved that by making the distance between the vertices equal, with respect to the overall number of vertices.
- Our algorithm is very fast and efficient, with time complexity of $O(V + E)$ and space complexity of $O(V + E)$

Con's:

- Our algorithm has a problem with large datasets visualization (e.g. polblogs.dot), since placing the vertices in a circuital structure makes the visualization dense for large datasets that contain a lot of vertices and edges. Consequently, the resulting graph becomes unreadable, as the delineation of edge origins and destinations becomes obscured, impeding effective data comprehension.
- Our algorithm struggles to represent long strings understandably. For example, the graph from "polblogs.dot" has words instead of numbers as the ID of each vertex. To address this, we need to adjust the shape and size of the vertices accordingly, which leads to overlapping between the vertices.

Unfortunately, our algorithm cannot handle the visualization and management of those cons in a readable and clean way.

3.5 Bonus Part

In this part, we took the original layout of the vertex placement and adjusted it to make an improved visualized graph with clearer and more informative results, Especially for large datasets (e.g., JazzNetwork.dot). We've done so by adding layers to the calculation of the vertices' location in the original graph. The network is divided into three layers based on the number of neighbors each vertex has (See *Figure* 2). The outer layer consists of vertices with a number of neighbors equal to or fewer than the average number of neighbors divided by 3. The middle layer contains all the vertices with a number of neighbors between the average number of neighbors and the average number of neighbors divided by 3. The inner layer includes all remaining vertices. To calculate the middle and the inner layer, we multiplied the radius of the outer layer by 0.2 and by 0.4 respectively. We've done so because in our original layout (where all of the vertices are on the same layer = in the same distance from the center(0,0)) we couldn't (or it was very hard) distinguish between the vertices that have fewer neighbors than the ones that have more neighbors. We accomplished this by creating a new function called "place_vertices_in_circle_Bonus" that, in addition to placing the vertices into a circular shape, also divided them into layers (as explained above). In that function, we added a few "if" statements that helped us separate the vertices by neighbors by changing the radius (the distance from the center (0,0) according to the corresponding layer(as explained above). By changing the positions of the vertices (with the new function " place_vertices_in_circle_Bonus"), we obtained a better visualization than using the original algorithm. The new algorithm differs from the original algorithm by additional "if" statements that all of them are $O(1)$ (we are just assigning a new value to a variable), therefore the time complexity did not change and stayed $O(V)$ where V denotes the number of vertices in the graph.

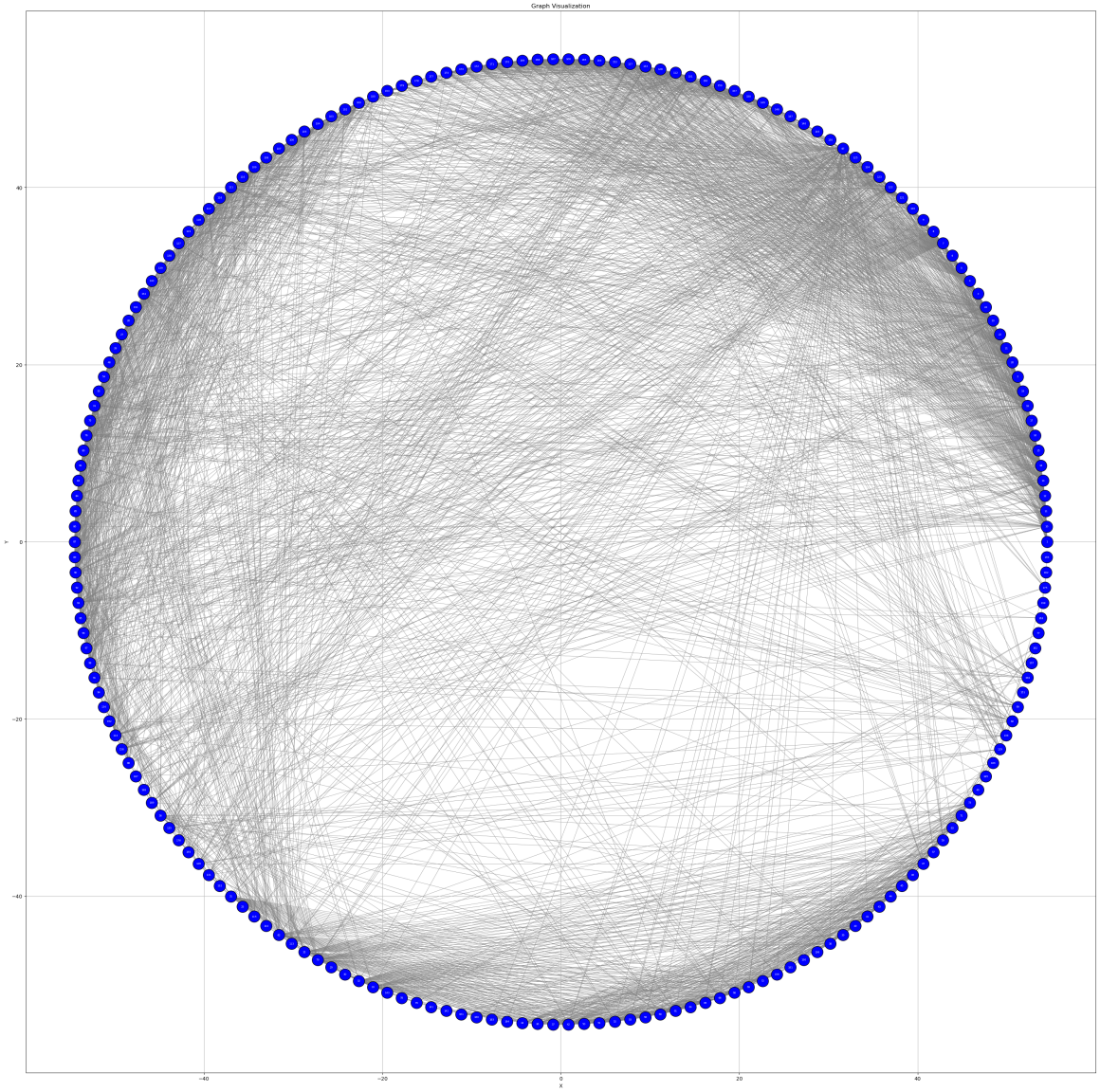


Figure 1. *Visualized the Jazz network using the original algorithm*

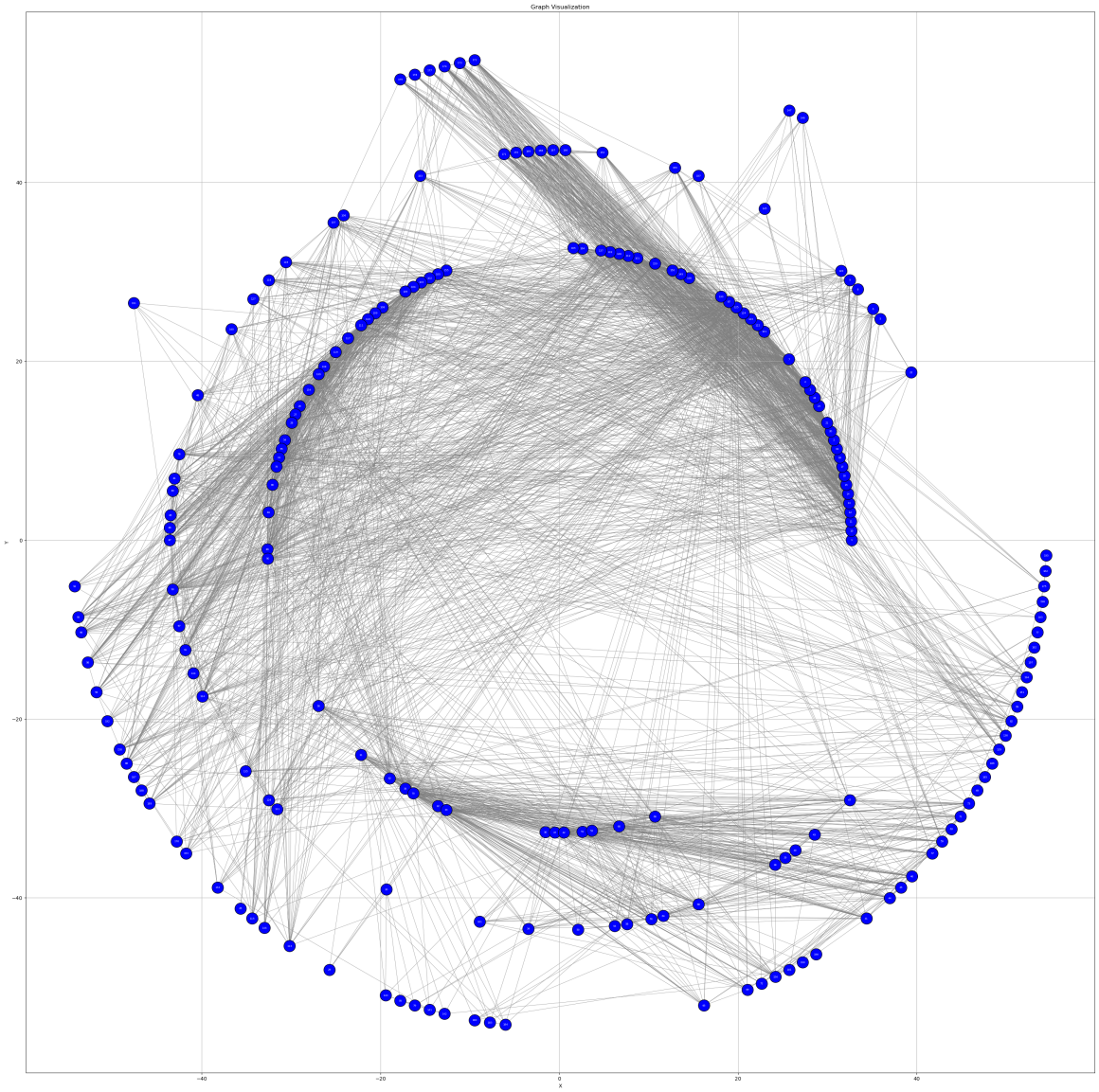


Figure 2. *Visualized the Jazz network Graph using the improved algorithm*

The running time for the bonus part on the "JazzNetwork.dot" file is 8.97 seconds
Additionally, to visualize the graph in this part, we needed to calculate additional things(the average neighbors and the running time); we still needed to iterate through all of the vertices and edges, as in the original algorithm. The **time complexity** remains $O(V + E)$ where V denotes the vertices and E denotes the edges in the graph. Although we created a new visualization function for the bonus part called "visualize_graph_Bonus", which involves computing the average and running time, we added one more loop with time complexity of $O(V)$. However, since it is not nested within the other loops, its time complexity is $O(V + V + V + E) = O(3V + E) = O(V + E)$. We didn't change any of our data structures; therefore, the space complexity didn't change either and stayed $O(V + E)$.

4 Step 2: Extract and visualize trees

In this stage, our objective is to transform our graphs into tree structures. Initially, we employ Breadth First Search (BFS) and Depth First Search (DFS) algorithms to accomplish this conversion. Following that, we decided to utilize the radial algorithm due to the circular structure of the graph, which aligns with the approach adopted in Step 1. Then, we can analyze the complexity and experiment with multiple graphs and root vertices. Finally, we incorporate non-tree edges and provide insights based on our observation of the resulting tree.

Additional class attributes required for this step:
For the implementation of the stack structure of DFS we created a class named Stack and we implemented the

following supporting functions:

- **is_empty**: returns whether the Stack is empty.
- **push**: adds an item at the top of the stack.
- **pop**: retrieves one item from the bottom of the stack.
- **top**: takes the item at the top of the stack

4.1 BFS and DFS

BFS and DFS are fundamental graph traversal algorithms used to explore and analyze graphs or trees. DFS systematically explores each branch as much as possible before backtracking. On the other hand, BFS explores the neighbor vertices at the present level before moving to the vertices at the next level. Moreover, BFS employs a queue data structure exploring level by level in contrast with DFS which is typically implemented recursively or using a stack data structure.

In this project, we implemented both BFS and DFS algorithms by creating separate data structures, respectively.

4.1.1 BFS

For the **BFS** we created a class **vertex_for_BFS** that will contain the following attributes:

1. *id* - The id of the current vertex.
2. *color* - Denotes the state of the vertex with respect to the queue:
    - 0 - White (not visited yet in the queue).
    - 1 - Gray (currently in the Queue).
    - 2 - Black (visited and already exited the Queue).
3. *distance* - The length of the shortest path between the current vertex and the root vertex.
4. *parent* - The id of the parent vertex.

For the implementation of BFS we created the function:

- **BFS**: This function takes a graph and the current root vertex, initializing a table to store vertex information like color, distance from the root, and parent vertex. It then creates a queue to track visited vertices, updates information for the root vertex, marks the current vertex as visited, and explores its unvisited neighbors. Unvisited neighbors are added to the visited list, and their information in the table is adjusted. This process repeats until no unvisited vertices remain in the list.

4.1.2 DFS

For the **DFS** we created a class **vertex_for_DFS** that will contain the following attributes:

1. *id* - The id of the current vertex.
2. *color* - Denote the state of the vertex with respect to the Queue:
    - 0 - White (not visited yet in the Queue).
    - 1 - Gray (currently in the Queue).
    - 2 - Black (visited and already exited the Queue).
3. b - The Timestamp of inserting the current vertex into the Stack.
4. f - The Timestamp of extracting the current vertex from the Stack.
5. *parent* - The id of the parent vertex.

For DFS, a separate data structure for the stack is implemented. This stack is necessary for the DFS algorithm to maintain the order of vertices to be explored. Timestamps ("b" and "f") are used in the DFS algorithm to track the order of insertion and extraction from the stack, aiding in determining the exploration order. The class **Stack** contains the following support functions:

- **is_empty**: returns whether the Stack is empty.
- **push**: adds an item at the top of the stack.
- **pop**: retrieves one item from the bottom of the stack.
- **top**: takes the item at the top of the stack

Moreover, for the implementation of DFS we created the functions:

- **"DFS" Function**: This function creates a matrix to store various information for each vertex such as id, discovery time, finishing time, finishing time, color, and the parent of the vertex. Following that, it adjusts the information on the matrix about the root vertex and calls the DFS_recursive to fill the matrix with the information for all the vertex.

4.2 Radial Layout

The Purpose of employing the radial layout is to arrange the vertices of a tree in a circular structure based on their depth. Initially, we assign a root vertex in the center of the circular structure. Following that, the allocation of the position of each child on the next layer is determined by calculating the percentage of the space that is required based on the number of descendants. This process is iteratively applied to the subsequent vertices, considering only the space allocated to their respective parents.
For our choice to implement the Radial layout, we developed two key structures:

1. **TreeVertex**
   - *value* - The id of the current vertex.
   - *children* - A list containing all children's vertices of the current vertex.
   - *point* - Denotes the coordinates (x, y) on the grid where the current vertex is positioned within the Radial layout.
   - *parent* - The id of the parent vertex.
2. **RadialPoint**
   - *vertex* - Refers to the current vertex associated with the Radial Point.
   - *point* - Specifies the coordinates (x, y) on the grid where the current vertex is positioned within the Radial layout.
   - *parent_point* - Specifies the coordinates (x, y) on the grid where the parent vertex of the current vertex is positioned within the Radial layout.

With the implementation of the radial layout we used a couple of support functions that help us with the computation of the right position for each vertex:

- **"convert_to_tree" Function** : This function takes the vertices dictionary from the BFS and DFS algorithms into a new tree using both of the new structures that we created.
- **"subTree_vertex_count" Function**: This function takes a vertex and returns the count of vertices the sub-tree from the current vertex contains.
- **"get_depth" Function**: This function takes a vertex and returns the depth of this vertex, relative to the root vertex of the tree. In other words, how far it is from the root vertex.
- **"compute_depth" Function**: This function takes a vertex and returns the height of the tree that the current vertex represents the root vertex. In other words, the length of the longest path between the current vertex and its descendants.

"radial_positions" Function - The main function that we created to implement the radial layout:
In this function we adjust the positions of the vertices according to the radial layout. This is a recursive function that places the vertex with respect to its parents and their parent's "restricted" curve on the current layer. To find the relative curve of the circle, we implemented the following equation to determine the upper bound:

$$\mu = \theta + \frac{\lambda(v)}{\lambda(u)} \times (\beta - \alpha) \tag{1}$$

- θ represents the starting point of the curve.
- $\lambda(v)$ represents the number of vertices in the sub-tree rooted at vertex V.
- $\lambda(u)$ represents the number of vertices in the sub-tree rooted at vertex U (one of V's direct children).
- β represents the upper bound of the curve from the previous layer.
- α represents the lower bound of the curve from the previous layer.

After computing both the upper and lower bounds of the current curve, we position each child vertex at the midpoint of the curve, adjusting its radius relative to the current layer. Subsequently, we update the lower bound to match the upper bound and proceed to calculate the position of the next child. This process continues until all vertices within the graph have been positioned.

4.3 Complexity of our Radial Layout

First, we compute the BFS/DFS on the original graph $O(V+E)$, then we take the dictionary from the BFS/DFS outputs and convert it into our custom tree (of TreeVrtices). We used the following functions:

- **"convert_to_tree" Function** This function converts a dictionary of vertices into our tree structure ("TreeVertex"). It computes the depth of each vertex and stores it in the depth cache. Additionally, it computes and stores the subtree vertex count for each vertex in the subtree count cache. Since each operation iterates through the vertices once, the time and space complexity are both $O(V)$. where V is the number of vertices
- **"compute_depth" Function** This function recursively computes the depth of a vertex in the tree. It visits every vertex in the sub-tree once, so its time and space complexity are both $O(V)$.
- **"get_depth" Function** This function returns the depth of a vertex in the tree from a stored cache. Therefore, its time complexity is $O(1)$ and its space complexity is $O(1)$.
- **"subTree_vertex_count" Function** This function counts the number of vertices in the subtree rooted at a vertex. It uses a queue to perform a breadth-first search on the sub-tree, so its time and space complexity are both $O(V)$.
- **"get_subTree_count" Function** This function returns the number of vertices in the subtree rooted at a vertex from a stored cache. Therefore, its time complexity is $O(1)$ and its space complexity is $O(1)$.
- **"radial_positions" Function** This function computes the radial positions of the vertices in the tree. It is called recursively for each vertex in the tree, and each call performs two $O(1)$ operations: "get_depth" and "get_subTree_count". Therefore, its time complexity is $O(V)$.
- **"visualize_radial_layout" Function** has a time complexity of $O(V + E)$, where V is the number of vertices and E is the number of edges in the graph. This is because the function visualize_radial_layout calls the function radial_positions, which is $O(V)$ as explained before, and then iterates over all the vertices and edges in the output graph, which is O(V + E). Therefore, the total time complexity is $O(V + V + E) = O(V + E)$.

4.4 Les Misérables network with BFS and DFS

As we can observe from *Figure* 3 BFS creates fewer layers (4) compared to DFS (11). This observation confirms that BFS explores levels of the tree structures more evenly, while DFS might focus on a single branch before moving to the next one. In addition, we can see that the parents in BFS have more children than the parents in DFS, who have mostly one or two. In addition, in the case of BFS, vertices are spread out around the circles at each level, which makes the graph more evenly distributed. In contrast, DFS creates multiple long chains of vertices. Finally, we can observe that there is no crossing of edges in both graphs, which makes the relationships between the vertices clearer and visible.

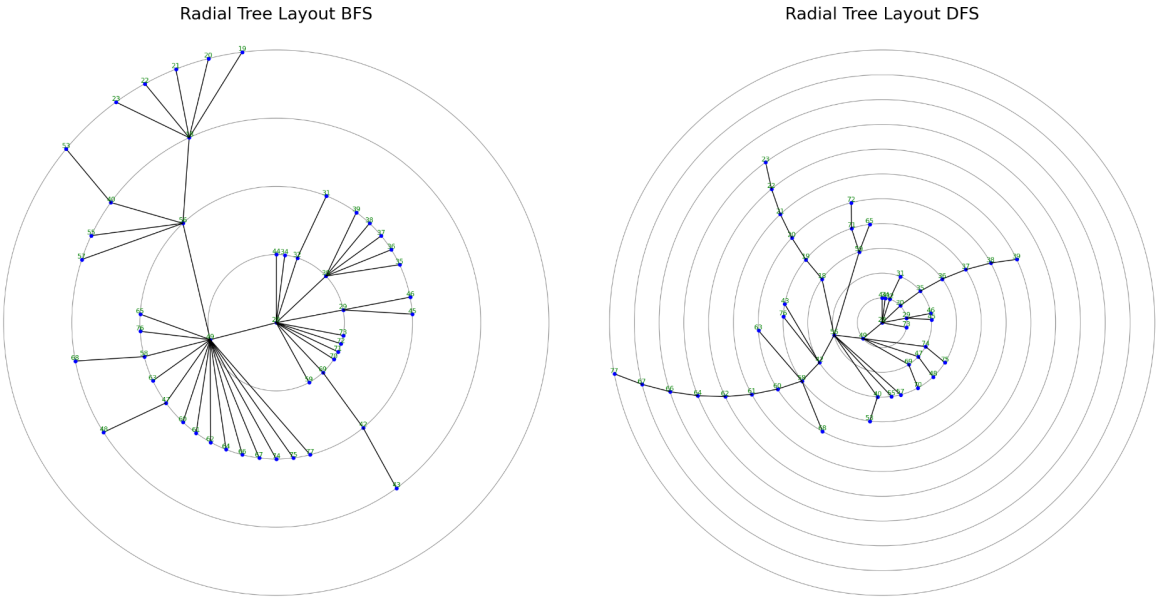


Figure 3. *Visualizing the Les Misérables Network with the root vertex "28" with BFS and DFS*

4.5 Jazz network with BFS and DFS

Similarly, in the Les Misérables network we can observe from *Figure* 4 that the BFS of the Jazz network creates 5 layers, a notable difference in comparison with the 14 layers of DFS. In addition, it is interesting to highlight that the Jazz network forms more layers than Les Misérables. This difference derives from the fact that the Jazz network has a more complex structure with more vertices than the Les Misérables network. Moreover, we can also see that in this network, the vertices are more evenly distributed with the BFS algorithm and have more depth with the DFS algorithm. It is notable to mention that, like before, there are no instances of edge crossing or edge bending.

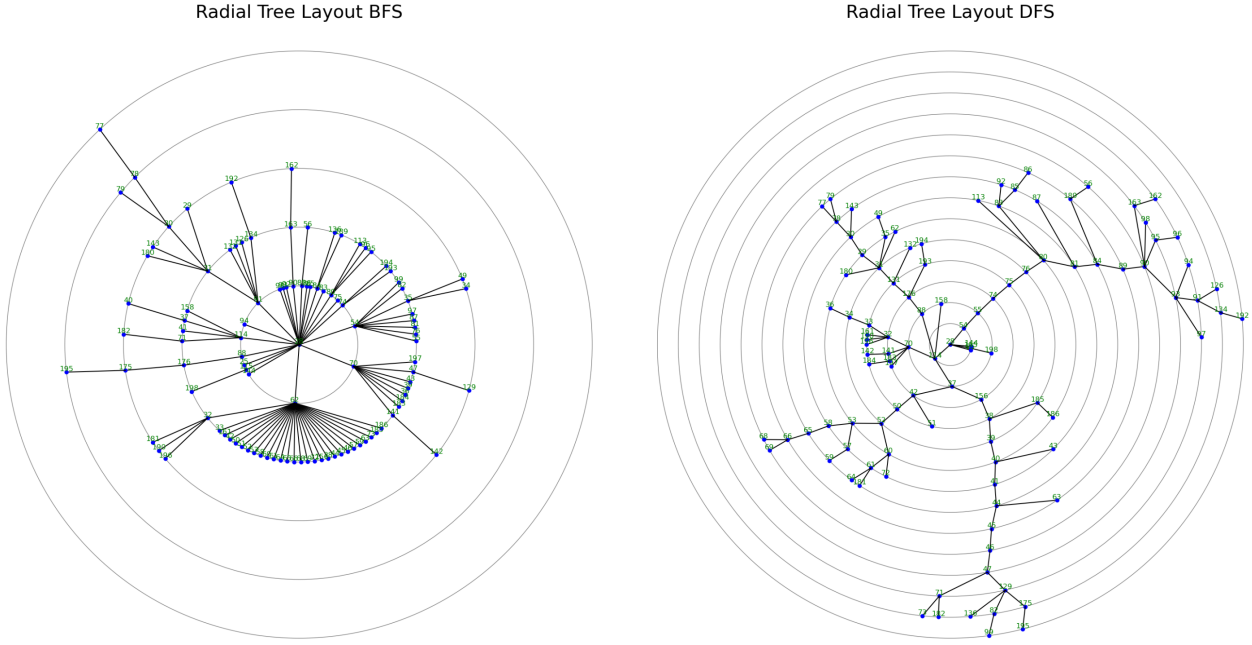


Figure 4. *Visualizing the Jazz Network with the root vertex "28" with BFS and DFS*

4.6 Quality

The overall quality of our graph is commendable. The edges are rendered as straight as possible without any crossing. The graph's dimensions are appropriate, which makes the vertices and their relationships visible without sacrificing any information. However, the only drawback is that certain vertices are positioned closely,

making them indistinguishable in a simple image *Figure* 3 (Radial Tree Layout DFS).

4.7 Comparison between BFS and DFS

When applying the radial algorithm using the output graph from DFS, it generated more layers (circles) in the output compared to when using BFS. This confirms the fact that DFS algorithms tend to travel in depth resulting in a graph with a higher level.

4.8 DFS with several root vertices

The choice of the root influences the visualization of the graph significantly. From the graphs in *Figure* 5 and *Figure* 6 with roots 56 and 28, we observe that they generate 9 and 11 layers, respectively, which is significantly fewer than when roots 11 and 1 are assigned, resulting in 17 and 21 layers respectively. Additionally, we can see that the choice of the root impacts the total number of vertices in the graph. This occurs due to the directed nature of the graphs. For instance, if we have two vertices (U and V) where V is the parent of U, selecting U as a root will exclude the parent V from the output graph. Finally, it's evident that graphs with root vertices 1 and 11 that have more vertices in total generate longer edges (*Figure* 5 and *Figure* 6), causing children to be farther from their parents. In contrast, the remaining graphs follow a pattern where the majority of children are placed close to their respective parents.
As we can observe in *Figure* 5 and *Figure* 6, the quality of visualization deteriorates or improves based on the root vertex. Moreover, after multiple tries, we have concluded that choosing a vertex with many connections for a root vertex proves to be the better choice.

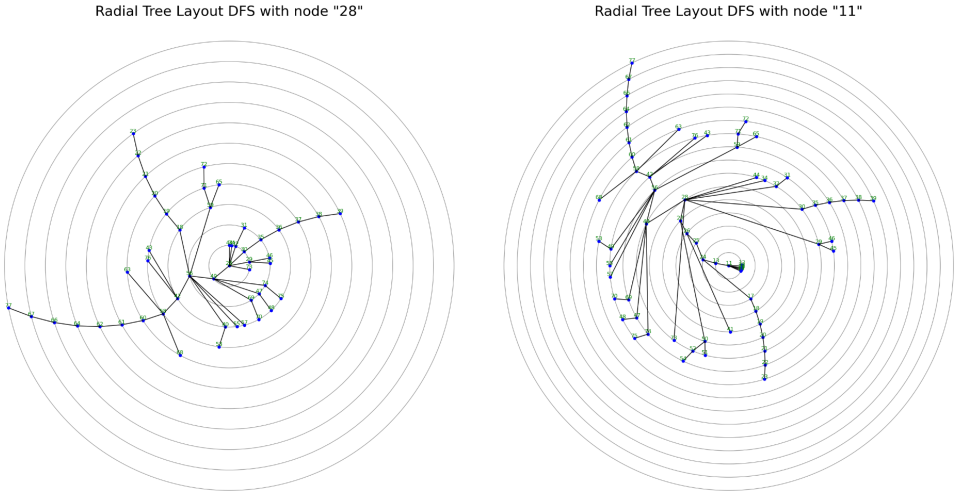


Figure 5. *Visualizing the Les Misérables Network with DFS with the root vertices "28" and "11"*

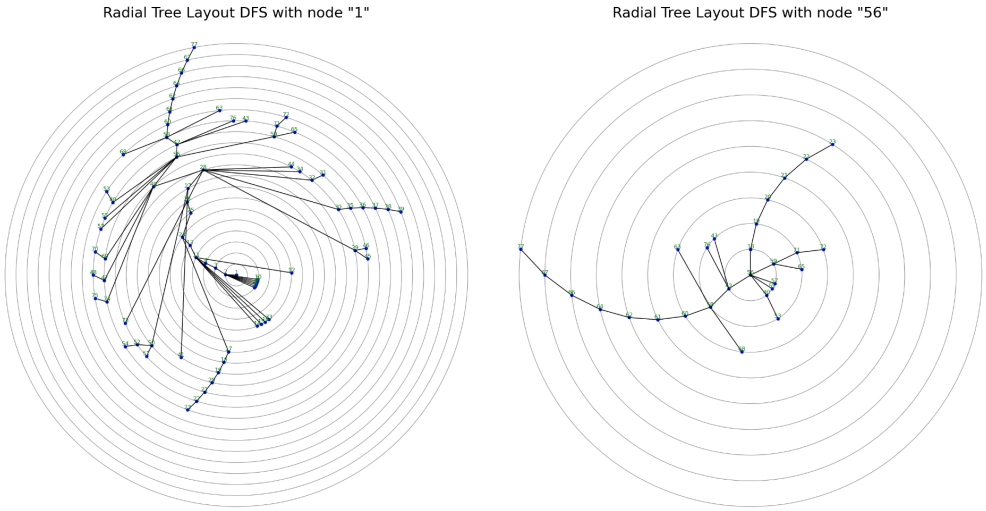


Figure 6. *Visualizing the Les Misérables Network with DFS with the root vertices "1" and "56"*

4.9 Radial Layout with non-Tree graph

Using an extracted tree and adding remaining edges may not always provide additional valuable insights, particularly for directed graphs. In the case of directed graphs, this method can be inefficient as the orientation of the edges plays a crucial role in defining the graph's structure. Choosing the root vertex becomes essential because it may lead to the removal of certain edges, potentially resulting in the loss of crucial information present in the original graph.
Conversely, for undirected graphs represented as trees, this approach can be quite beneficial. By depicting the graph as a tree, it becomes easier to discover meaningful relationships or paths between vertices that might otherwise be challenging to identify. Thus, while this method may not always be suitable for directed graphs, it can be highly effective for revealing interesting and significant connections within undirected graphs *Figure* 7.

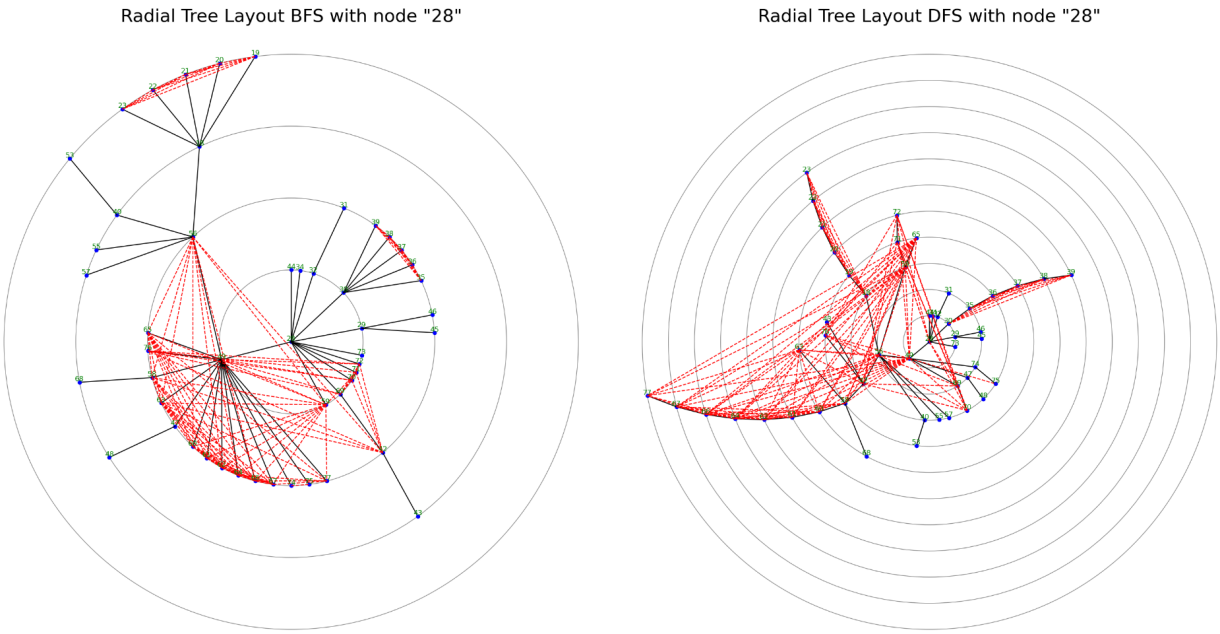


Figure 7. *Visualizing the Les Misérables Network with the root vertex "28" with BFS and DFS with the addition of the original edges (in dashed red lines)*

For this step we implemented the following functions:

- **"create_non_tree_edges_list" Function**: This function takes the original graph and the tree graph generated by BFS or DFS as inputs. It iterates through all the edges in the original graph, saving all the edges that were removed to create the tree. Finally, it returns all of those edges.
- **"visualize_radial_layout_with_non_tree_edges" Function**: With this function, we can visualize all the edges from the original graph, that the BFS/DFS removed while building the corresponding tree. These edges are shown by a dashed red line.

5 Step 3: Compute a force-directed layout

In this step, we created a layout using the Eades Spring-Embedder algorithm [7]. We applied our layout to draw the Les Misérables and the Jazz Network graphs. Firstly we will present the implementation of the Eades Spring-Embedder layout, followed by an explanation of the choice of the constants and hyperparameters that we used. Finally, we will discuss the complexity, the quality of the plotted results, and potential improvements in quality metrics.
Additional class attributes required for this step:
For this step, we created a new class named **Forced_vertex**, which consists of the id of the vertex, its position, and the force that was applied.

5.1 Eades Spring-Embedder layout implementation

The Eades Spring-Embedder algorithm is a graph layout algorithm used to visualize undirected graphs in a two-dimensional space. It simulates the behavior of springs to position the graph vertices in a visually pleasing

manner. In our implementation, first, we converted the graphs into undirected graphs by making all of the edges bidirectional. Then, we placed the vertices in a random position on the canvas. The logic is that the vertices should move according to the forces applied to them until they reach a position where the system is closest to equilibrium [7]. The forces that have to be computed are :

- The repulsive forces between two non-adjacent vertices u and v

$$f_{\text{rep}}(p_u, p_v) = \frac{C_{\text{rep}}}{\|p_v - p_u\|^2} \cdot p_u\vec{p}_v$$

- The attractive forces between adjacent vertices u and v

$$f_{\text{spring}}(p_u, p_v) = c_{\text{spring}} \cdot \log\left(\frac{|p_u - p_v|}{\ell}\right) \cdot p_v\vec{p}_u$$

- The Total force that is applied on a vertex that will cause the vertex to move

$$F_v = \sum_{u,v \notin E} f_{\text{rep}}(p_u, p_v) + \sum_{u,v \in E} f_{\text{spring}}(p_u, p_v)$$

Where p_u is the position of the vertex u and p_v is the position of the vertex v.

After the initialization of the vertices, we run a loop where the forces are computed for each iteration, and the position of the vertices is recomputed and changes according to the total force in each vertex and a predefined moving step:

$$VertexPosition = VertexPosition + F_v \cdot \delta$$

The algorithm runs until the the the movement of the vertex with the maximum movement, become smaller than ϵ.
In our layout, we used the following Constants and Hyperparameters:

Constants:

- l = the ideal length of springs between vertices, determining their attraction strength, and it pulls vertices closer if they're too far apart.
- C_{rep} = determines the strength of repulsive forces between vertices. Vertices repel each other with a force inversely proportional to the distance between them. Crep scales this repulsive force.
- C_{spring} = influences the strength of attractive forces between connected vertices. It determines how strongly vertices are pulled towards each other when they are within the ideal spring length.

Hyperparameters:

- δ = determines vertex movement in each iteration. Larger δ speeds up convergence but may lead to instability. (learning_rate)
- *damping* = Damping slows down vertex movement towards optimal positions, stabilizing the layout and preventing excessive motion.
- ϵ = serves as a convergence criterion for stopping the iteration process. If the maximum movement of any vertex in a single iteration falls below , the algorithm terminates, indicating that the layout has reached a stable configuration.

After conducting various experiments with different constants (refer to *Figures* 8, 9, 10 and 11), we concluded that the most visually appealing results for both the "Les Misérables" (see *Figure* 12) and "Jazz Network" (see *Figure* 13) graphs were achieved with the following constant values: $l = 1$, $C_{rep} = 1$, $C_{spring} = 2$. However, the hyperparameter values varied for each graph. For "Les Misérables" graph we used learning_rate = 0.1 , *dumpling* = 0.99 , $\epsilon = 0.05$, while for the Jazz Network we used learning_rate = 0.01 , $\epsilon = 0.03$, and again the same *dumpling* = 0.99. The optimal visual outcomes were attained after 232 and 578 iterations for "Les Misérables" and the "Jazz Network", respectively.

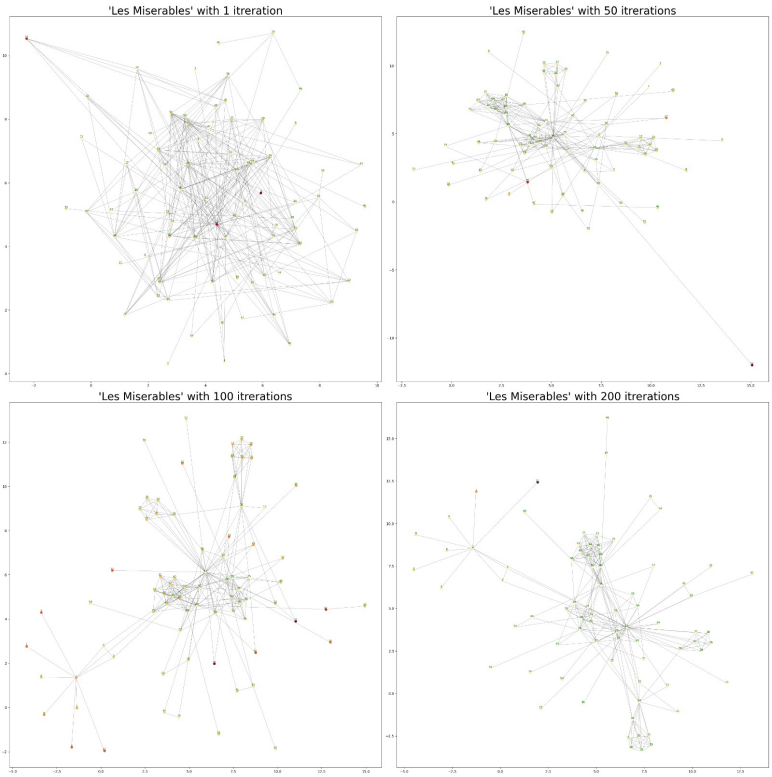


Figure 8. *Comparison between the number of iterations on the Les Misérables graph*

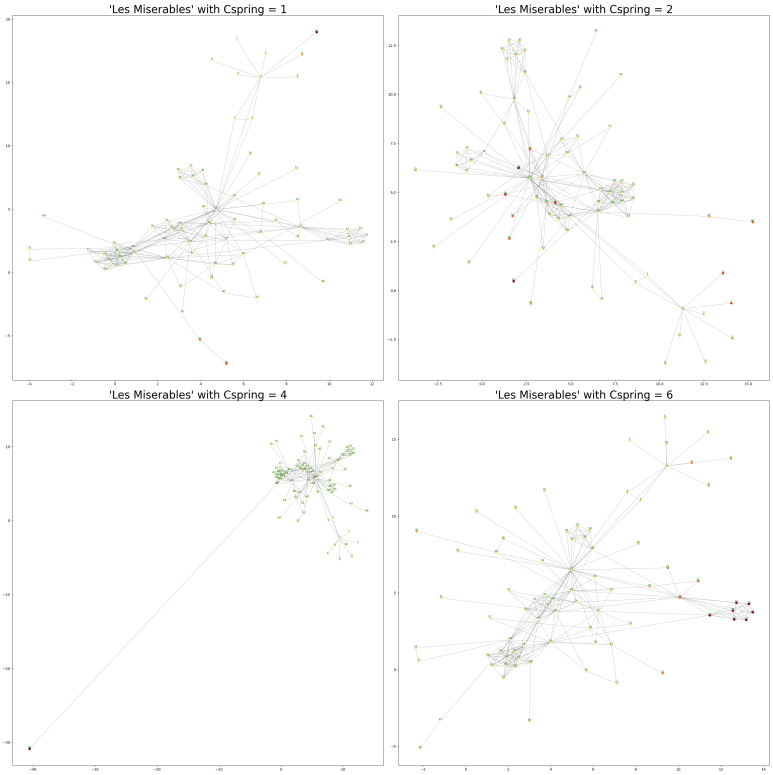


Figure 9. *Comparison between the values of $C_s pring on the Les Misérables graph*

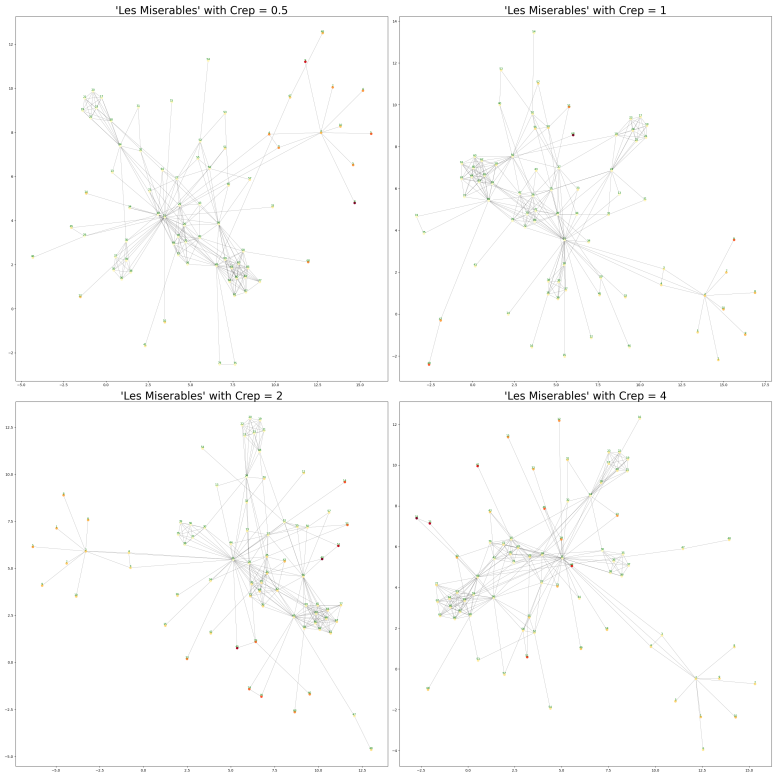


Figure 10. *Comparison between the values of* C_r*epontheLesMisérablesgraph*

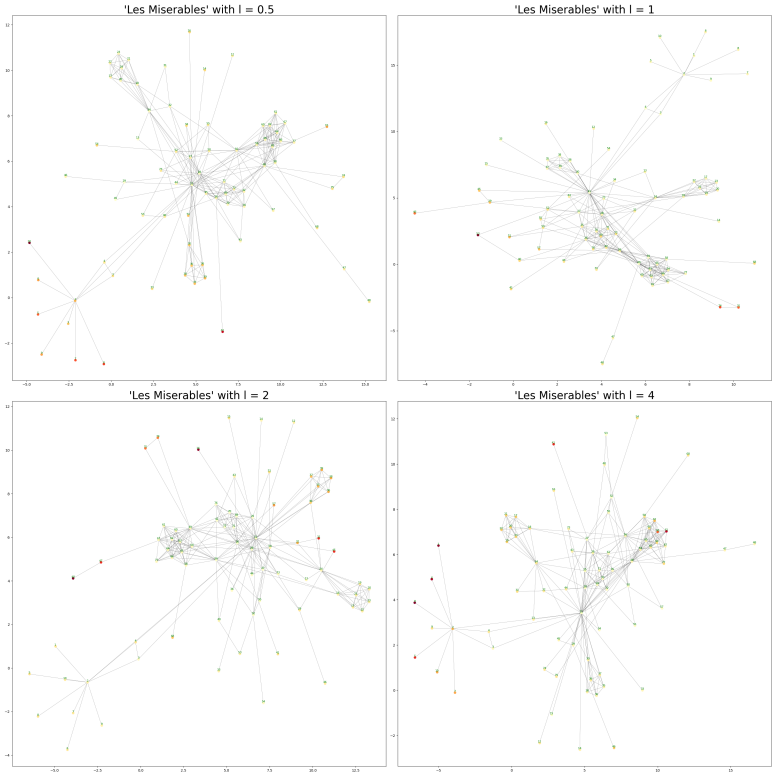


Figure 11. *Comparison between the values of* l *on the Les Misérables graph*

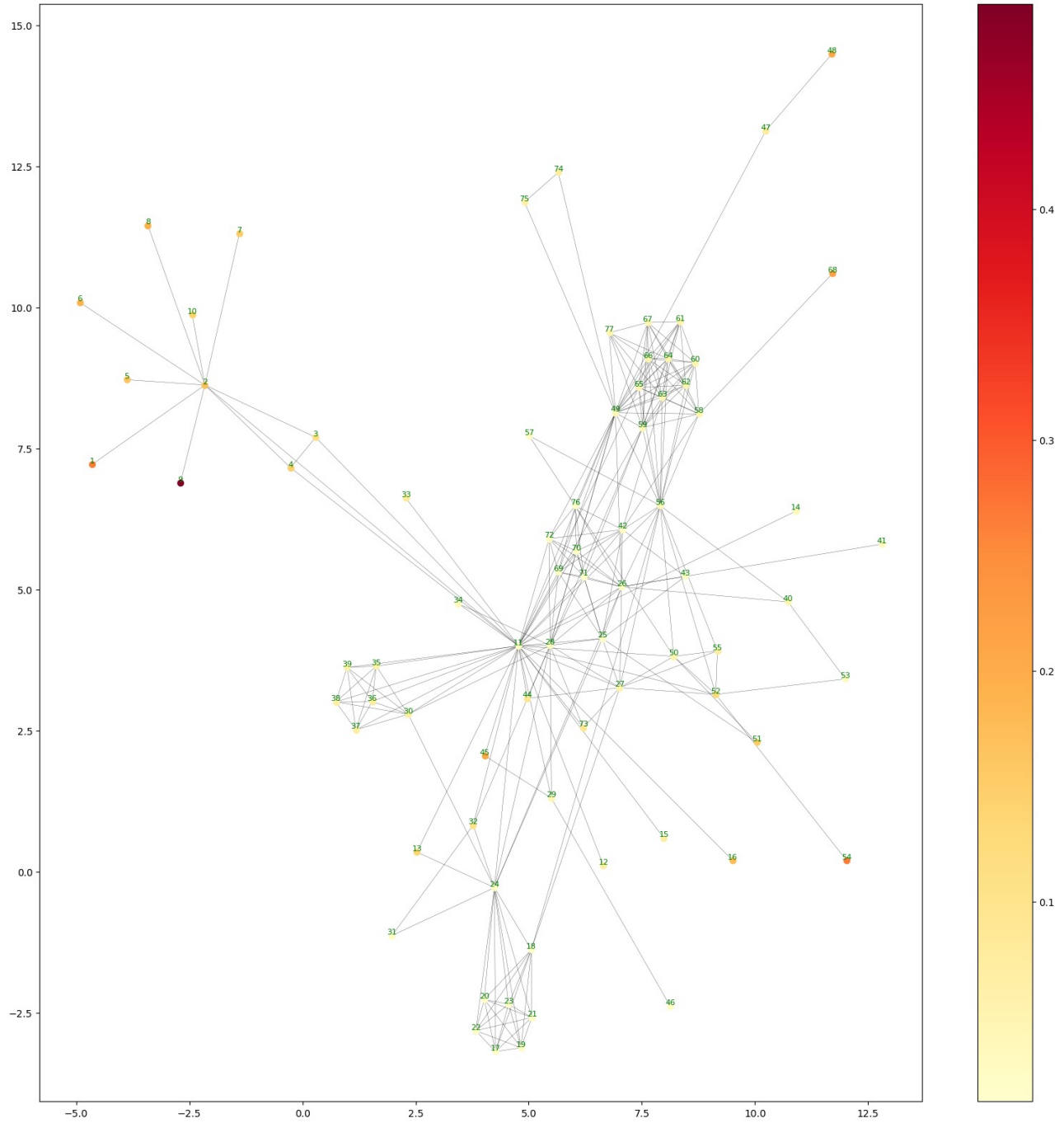


Figure 12. *Final visualization of Les Misérables force directed graph with the best hyperparameters.Number of iterations=232. (The color represents the size of the total force on the vertex on a scale from yellow to red, with yellow being the smaller force and red being the larger force)*

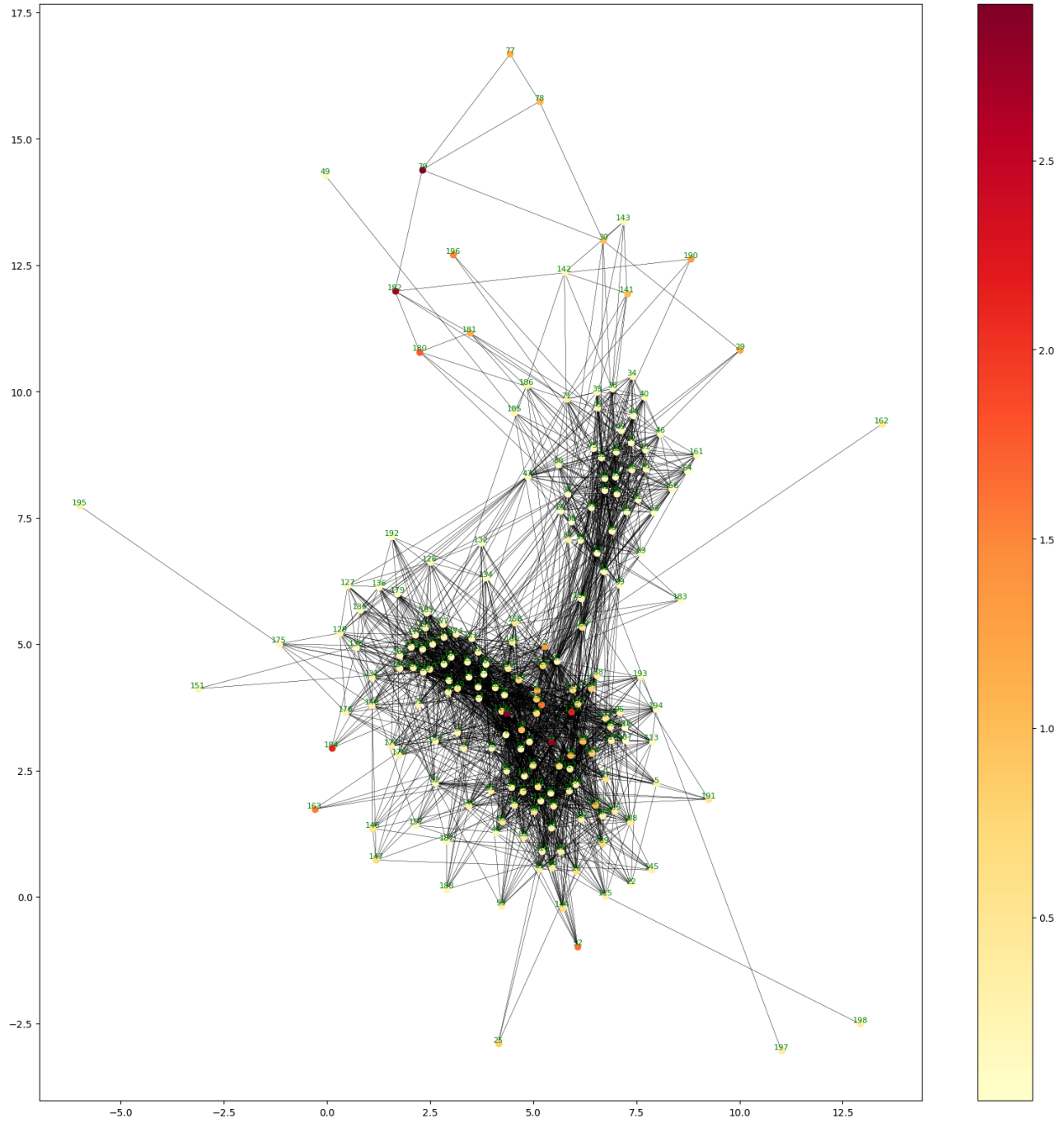


Figure 13. *Final visualization of Jazz network force directed graph with the best hyperparameters. Number of iterations=578. (The color represents the size of the total force on the vertex on a scale from yellow to red, with yellow being the smaller force and red being the larger force)*

5.2 Complexity

First, we take the original graph and modify the vertices and edges for our needs O(V+E), then we iterate (at most) k steps, while in each iteration we are computing and updating the forces for each vertex. To achieve that we used the following functions:

- **"initialize_Forced_vertices" Function**: This function converts the list of vertices from the original graph into a set of forced vertices, which are part of a new data structure that we created for this algorithm. Its time complexity is O(V), where V is the number of vertices in the original graph.
- **"is_edge_between_vertices" Function**: This function efficiently checks if the current potential edge, disregarding directions, exists in the graph's edges list (because we are working on an undirected graph). Since searching in a set is O(1) the total time complexity is O(1).
- **"update_positions" Function**: This function iterates through all of the vertices and updates their forces and positions according to the final force. Since it processes each vertex exactly once, its time complexity is O(V).
- **"compute_forces" Function**: This function is the main computation function. Initially, it resets the force for each vertex to zero, requiring O(V) operations. Next, the function iterates over each pair of vertices in the graph. For a graph with V vertices, iterating over each pair of vertices results in a time complexity of O(V^2). Within these loops, the function performs elementary actions, each with a time complexity of O(1). These actions include computing distances, norms, and other simple calculations. Therefore, the total time complexity of the function is O(V^2 + V) = O(V^2)
- **"visualize_layout" Function**: has a time complexity of O($V + E$), where V is the number of vertices and E is the number of edges in the graph. This is because the function visualize_layout iterates through all the edges and vertices in the graph to plot them according to their positions. Additionally, it computes normalized values of the forces acting on the vertices to provide a more detailed representation. This involves changing the color of each vertex based on the magnitude of the force it experiences.

Therefore, for each iteration, the worst case scenario is:
O($V + E + V + V^2 + V + E$) = O($V^2 + 3V + 2E$) ⇒ This simplifies to O(V^2).

5.3 Quality and Potential Improvements

The graph visualization quality is outstanding, facilitating clear comprehension of the forced-directed graph. Further enhancements could be achieved by refining specific parameters. For instance, optimizing the l parameter value within the Fruchterman-Reingold force-directed algorithm could be beneficial. Moreover, exploring advanced adjustments such as incorporating inertia, gravitational, and magnetic forces may yield promising results. Nonetheless, the current outcomes provide a high-quality visualization, underscoring the effectiveness of our visualization methodology.

6 Step 4: Compute a layered layout

In this step, we transformed the original directed graphs, Small Directed Network and Pro League Network into a directed acyclic graph (DAG). This conversion was achieved by applying a heuristic with guarantees to eliminate the cycles of the graph by reversing the selected edges. Moreover, to improve the visualization of the (DAG) we created a layered layout by applying a layer assignment using Height optimization. Additionally, for the edges spanning more than two layers, we introduced dummy vertices in each layer to establish a connection between the parent and child, which passes through the dummy vertices. The addition of dummy vertices significantly improved the visual clarity of the graphs since we managed to avoid edges that passed through other vertices before they reached the vertex of interest. Finally, for further optimization, we utilize the Barycenter and median heuristics algorithm to minimize the edge crossing in our graphs.
For this step, we applied the algorithms for both the Small Directed Network and the Pro League Network. Unlike the Small Directed Network, the Pro League Network is dense, making visualization challenging. To address this, we filtered the edges to include only those with weights ranging from 1 to 3 for better visualization. However, even with this filtering, the relationships within the Pro League graph remain somewhat indistinct. Therefore, we will only present the final output where dummy vertices are removed, and the edges have their original form.

Additional class attributes required for this step:
For this step we added two new arrays in the class Vertex that contain the parents and the children of each vertex. Moreover, we implemented the following supporting functions:

Class FVertex

- **"add_parents" Function**: this supporting function adds parent vertex to the parents' list of the current vertex.
- **"add_children" Function**: this supporting function adds a child vertex to the children's list of the current vertex.

Class FSGraph

- **"remove_edge" Function**: This function receives the start and vertices of an edge and deletes the edge from the graph.
- **"remove_vertex" Function**: This function receives the id of the vertex and deletes it from the graph.
- **"update_x_y" Function**: This function receives the coordinates x and y and updates the information on the graph.

6.1 Creating a directed acyclic graph

As previously mentioned, to convert the original graphs to DAGs, we implemented a heuristic algorithm with guarantees [7]. The algorithm begins by removing sink vertices (those that have only in-going edges) from the original graph, along with the associated edges of interest. This process continues until no sink vertices remain in the graph. Similarly, we repeat a comparable procedure focusing on source vertices (those that have only out-going edges) until none remain in the graph. This step aims to eliminate vertices that cannot be a part of a cycle.
After removing the sink and source vertices, we were left with vertices containing both in-going and on-going edges. If the original graph is not empty at this stage, we proceed by selecting the vertex with the maximum difference between on-going and in-going. We then remove this vertex along with its out-going edges, while retaining its in-going edges as candidates for reversal. Then we repeat the process until the graph is empty. The output graph maintains the structure of the original graph but with the selected candidate edges reversed. In *Figure* 14 we can observe that the original graph contains the cycle 2,17,16. However, in the upgraded graph after applying the heuristic with the guarantees, we noticed that the edge 2,17 has been reversed.

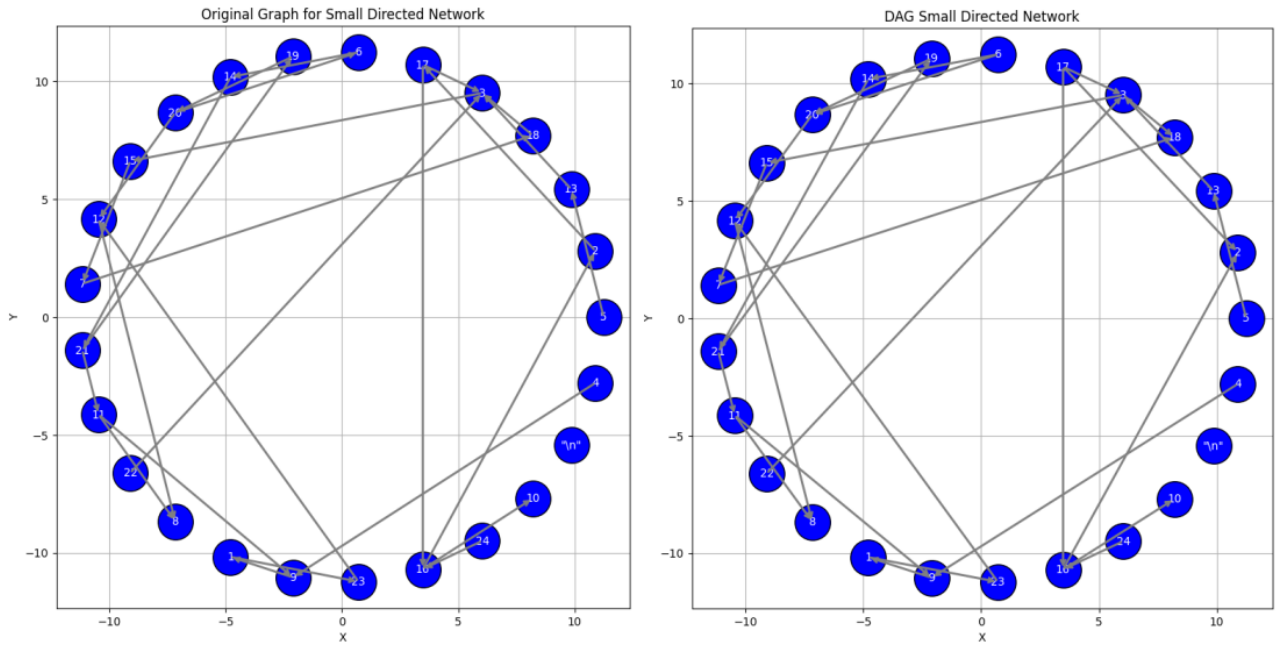


Figure 14. *Original and DAG graph*

For this step we implemented the following functions:

- **"remove_sink_vertices" Function** This function removes sink vertices along with their in-going edges and updates the graph accordingly. This process is repeated until the graph no longer contains any sink vertices. Time complexity $O(V + E)$ because we have
- **"remove_source_vertices" Function** This function removes source vertices along with their out-going edges and updates the graph accordingly. This process is repeated until the graph no longer contains any source vertices.
- **"remove_maximal_vertex" Function** If the graph is not empty, this function identifies the vertex with the maximum difference between in-going and out-going edges and removes it from the graph, along with its edges. In case of multiple vertex with the same maximal difference, the function selects the vertex with the fewest in-going edges, as minimizing the number of incoming edges is preferable since these are the edges we want to reverse. Finally, the function returns a list of the edges that need to be reversed.
- **"remove_edge_add_reverse" Function** This function takes the vertices of the edges that need to be reversed as input. It then deletes these edges and adds new reversed ones while updating information about the children and parents of each relevant vertex.
- **"heuristic_guarantees" Function** This is the main function where we invoke the previously mentioned functions until the graph is empty. Additionally, we reverse the edges in the output graph and return the new graph along with a list of the reversed edges for later use.

  the overall time complexity heuristic_guarantees

6.2 Layer assignment

To visualize the DAG resulting from the previous step, we implemented the height minimization algorithm. First, we initialize the first layer with the source vertices of the original graph. Following that, we removed the vertices (along with their outgoing edges) that we assigned to the previous layer from the original graph. Then, we initialize the next layer with the source vertices of the current graph and we repeat the process until no more source vertices remain in the graph. Furthermore, for the final graph, edges spanning more than two layers were subdivided by dummy vertices to improve the visual clarity of the graph, as some edges pass over other vertices. To accomplish this, we introduced a dummy vertex in each layer between the parent and the child.
As you can see in *Figure* 15 by adding dummy vertices we created a more clear representation of the graph structure and vertex relations. On the other hand, in the pro league network, after adding dummy vertices, we acquire a significantly expanded graph with numerous dummy vertices. This proliferation occurs because most layers consist of only one vertex with nearly every vertex being connected to every other vertex, which means that we have multiple edges that span over two layers (*Figure* 16).

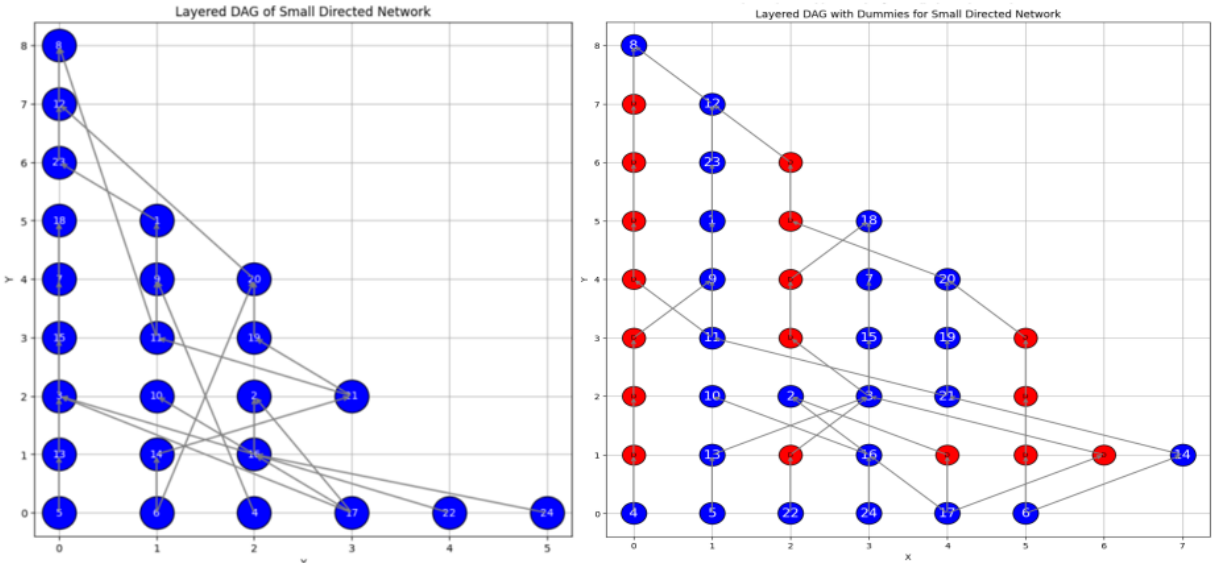


Figure 15. *Layered DAG and Layered DAG with dummies for small directed network*

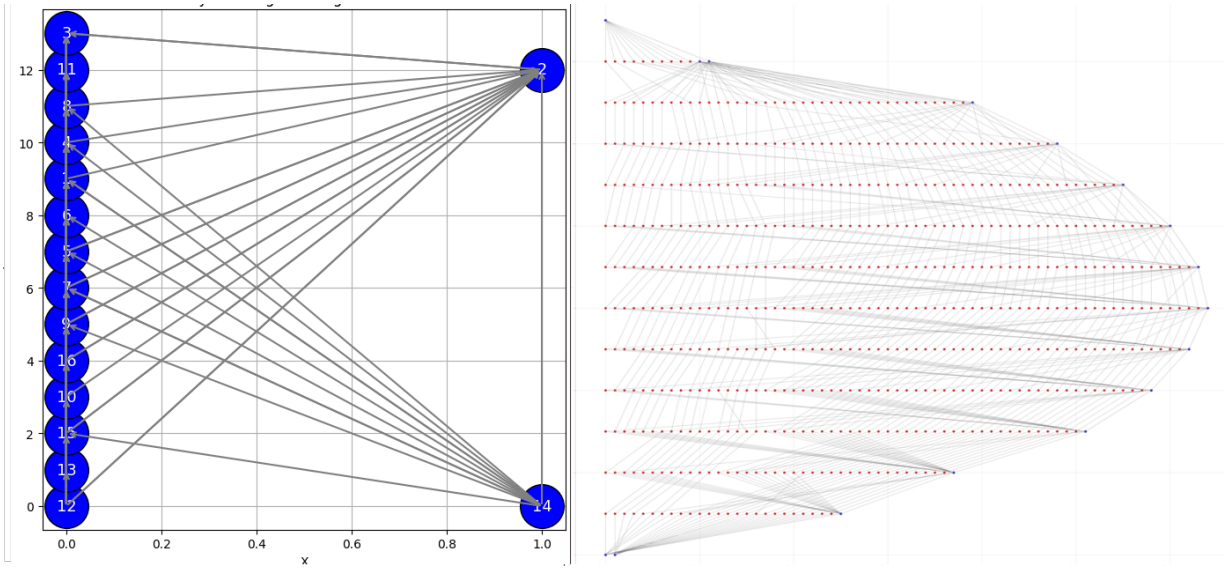


Figure 16. *Layered DAG and Layered DAG with dummies for pro league network*

For this step we implemented the following functions:

- **"update_parents_children" Function**: This supporting function receives a graph and updates the parents and children lists of the vertex based on the edges.
- **"sources_of_graph" Function** This function returns a list with the source vertices of the current graph.
- **"update_graph" Function** This function has as input the current graph and the source vertices and returns the new graph after removing the source and their corresponding edges.
- **"visualize_layered_graph" Function** We modified the visualization procedure from the previous steps to visualize the graph based on the layers, while we update the coordinates.
- **"insert_dummies" Function** In this function, for each edge, we examine if it passes over more than one layer based on the coordinates of the vertices at the ends. For every layer that these edges traverse, we introduce new dummy vertices and we adjust accordingly the graph structure to reflect the modification.
- **"layer_assignment" Function** This function is the main function for assigning the vertices to their respective layer. First, we initialize the bottom layer with all the source vertices of the graph. Following that, we iteratively refine the graph by eliminating the source vertices and their corresponding edges, and we proceed to the next level. This process continues until no additional source vertices remain within the graph.
- **"visualize_layered_graph_with_dummies" Function** We modified the visualize_layered_graph function to be able to visualize the dummy vertices in the layers and graph. The only difference is that we change the color and the size of the dummies to be noticeable.

6.3 Iterative Crossing minimization

In this optimization step, our objective is to minimize the number of crossings of the layered graph resulting from the previous step. First, we initialize the bottom layer with random ordering. Following that, for each pair of consecutive layers from bottom to top, we apply the Barycenter or Median heuristic to rearrange the ordering of the upper layer while keeping the bottom layer fixed. Once we complete this process for all layers from bottom to top, we repeat a similar procedure, but for consecutive layers starting from top to bottom. In this phase, we rearrange the bottom layer while keeping the top layer fixed. This iterative procedure continues until no further improvement is observed.
Through this process, we count the total number of edge crossings within the graph and we always keep the structure of the graphs with the minimum number of crossings. This procedure is repeated for a different ordering of the bottom layer until all the combinations have been exhausted.
For this step we implemented the following functions:

- **"get_index" Function**: This supporting function is the function that receives a layer and a target_id of a vertex and returns the position of the vertex in the layer.

- **"custom_key_a AND custom_key_b" Function**: these are supporting functions for the default function sorted in order in a case of two vertices at the same position with different degree parity to place the one with an odd number to the left.
- **"calculate_total_crossing" Function** This function gets two consecutive layers as input. For each vertex of the top layer, we examine where its parents' position is after the position of the parents of other vertices in the same layer. If this is the case, it indicates the presence of crossing edges.
- **"iterative_crossing_minimization" Function** This function is the main function for the iterative crossing minimization. For each pair of consecutive layers from bottom to top, we apply the heuristic algorithms barycenter or Median to rearrange the top layer keeping the bottom layer fixed. Similarly, the process is repeated from top to bottom, with the top layer fixed while the bottom layer is rearranged. This process is repeated for all the permutations of the bottom layer until we end up with an order that minimizes the crossing of edges.

6.3.1 Crossing minimization with Barycenter

This heuristic algorithm is designed to minimize the crossings between two consecutive layers in a graph. As previously mentioned, in each application of this algorithm, we adjust the ordering of one layer based on the position of its neighbors in the other layer, which remains fixed [8].
Specifically, in barycenter we apply the formula:

$$o_2(u) = \frac{1}{\deg(u)} \sum_{v \in N(u)} o_1(v) \tag{2}$$

Here, $o_2(u)o_1(v)$ denote the position of vertices u and v respectively, $\deg(u)$ represents the degree of the vertex u in the current graph (consisting only of the two layers), while $\sum_{v \in N(u)} o_1(v)$ sums the positions of the neighbors of u in the current graph.
For this step we implemented the following functions:

- **"rearrange_of_layers_Barycenter" AND "rearrange_of_layers_reverse_Barycenter" Functions** Both get as input two consecutive layers and compute the new position for each vertex in one layer by applying the Barycenter formula while keeping the other layers constant. The difference between them is that the "rearrange_of_layers_Barycenter" operates from bottom to top layers, where the bottom layer remains constant while the top layer is modified, whereas "rearrange_of_layers_reverse_Barycenter" operates from top to bottom, where the top layer remains fixed while the bottom layer is adjusted.

As we can observe from *Figure* 17, it is evident that the total number of crossings for the small directed network significantly decreases from 14 (see *Figure* 14) to just 1 after implementing the barycenter technique, which is a remarkable improvement. In the case of the Pro League Network, which is dense, we managed to reduce the crossing from 5593 to 1200 with the median, which is still a great improvement.

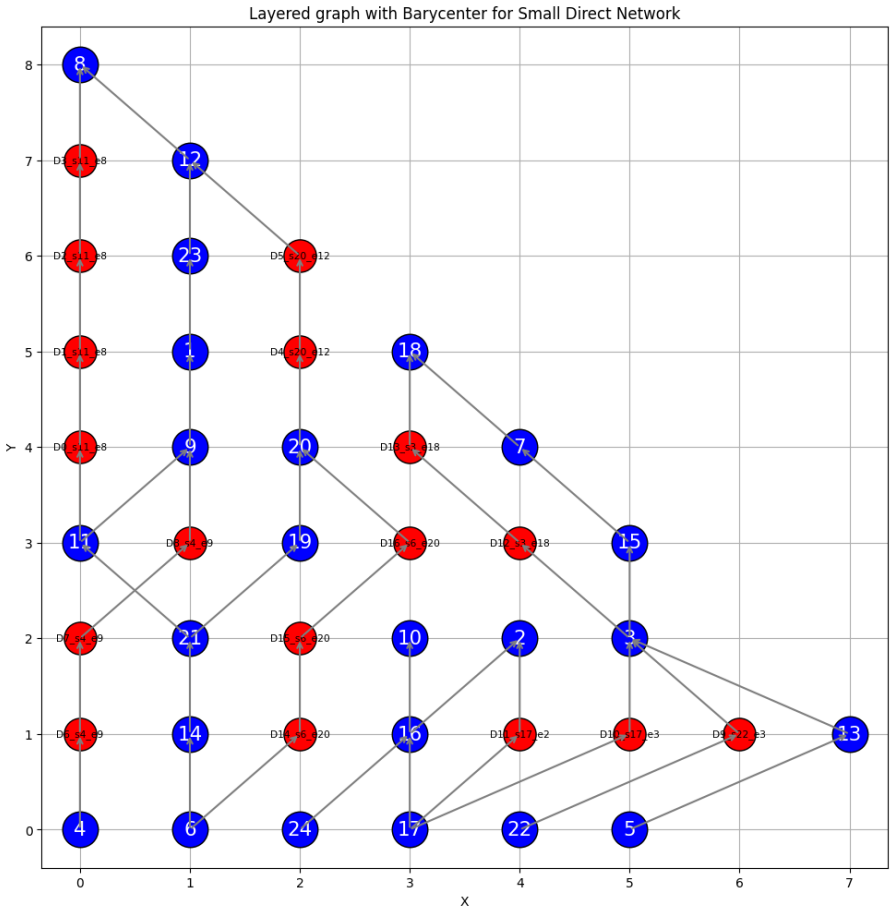


Figure 17. *Layered DAG with dummies after applying Barycenter.*

6.3.2 Crossing minimization with Median

This heuristic algorithm serves as an alternative to the Barycenter within the iterative crossing minimization algorithm with the same input. The difference of the median approach is that it positions the vertex of interest at the midpoint of its neighbors of the fixed layer. If the vertex has no neighbors, it is placed at position 0. If the vertices within the same layer have the same position but different degree parities, the vertex with an odd degree parity is placed to the left. However, if the vertices have the same degree of parity, the algorithm chooses arbitrarily which vertex will be placed first [9].
The formula is:

$$o_2(v) = \text{med}(v) = o_1\left(\left\lceil \frac{k}{2} \right\rceil\right) \tag{3}$$

Here, $o_2(u)$ denotes the position of vertex u and $o_1\left(\left\lceil \frac{dk}{2} \right\rceil\right)$ represents the position of the neighbor that is in the middle of all the k neighbors of u.
For this step we implemented the following functions:

- **"rearrange_of_layers_median" AND "rearrange_of_layers_median_reverse" Functions** Both get as input two consecutive layers and computing the new position for each vertex in one layer by applying the Median algorithm while keeping the other layers constant. The difference between them is that the "rearrange_of_layers_median" operates from bottom to top layers, where the bottom layer remains constant while the top layer is modified, whereas "rearrange_of_layers_median_reverse" operates from top to bottom, where the top layer remains fixed while the bottom layer is adjusted.

As we can observe from *Figure* 14, the total number of crossings for the small directed network significantly decreases from 14 (see figure 17) to 9 after implementing the Median technique. In the case of the Pro League Network which is dense, we managed to reduce the crossing from 5593 to 1254 with the median. This technique performed worse than barycenter which may be since the median assigns random vertices that have the same position and same parity.

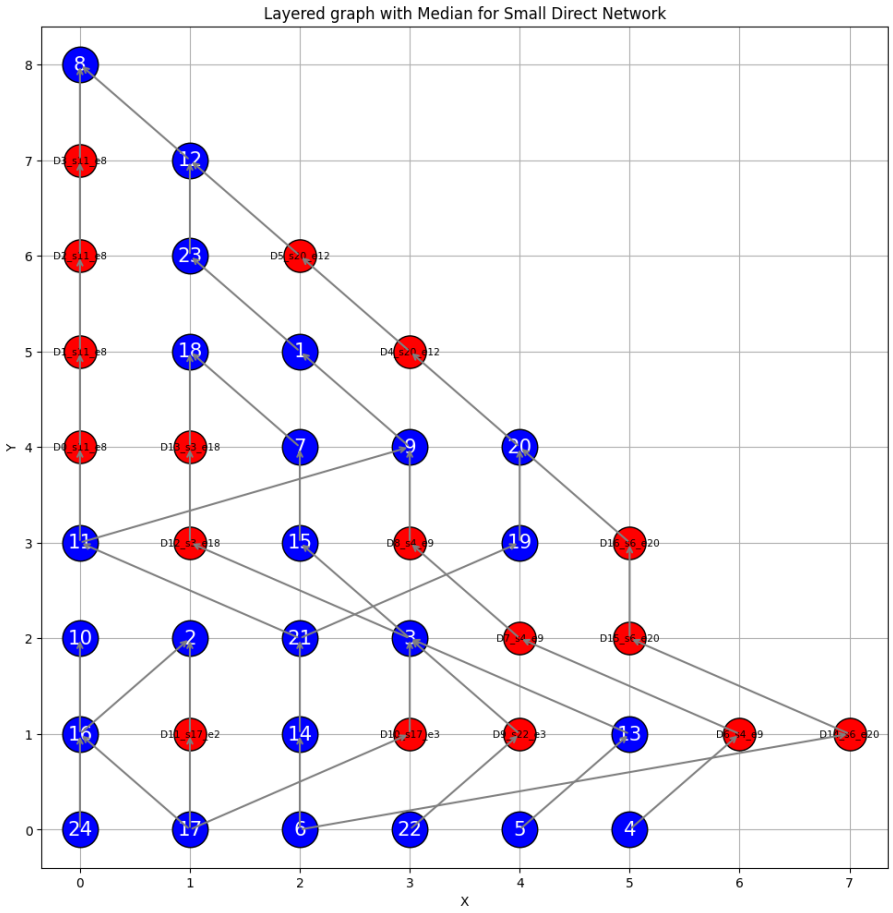


Figure 18. *Layered DAG with dummies after applying Median.*

6.4 Assign coordinates and draw edges

In this step we assign coordinates in our vertices based on their position on the layers. Moreover, we draw the edges without the dummy vertices without changing the structure of the vertices.
For this step we implemented the following functions:

- **"visualize_layered_with_crossing_without_dummies" Function** We modified the "visualize_layered_graph" function to be able to visualize the graphs without dummy vertices, however, we kept the position of the dummy vertices to preserve the structure of the graph and present the edges without the interruption of the dummies.

In *Figure* 19 we can see the output graphs for both the small directed network and pro league network after we removed the dummy vertices.

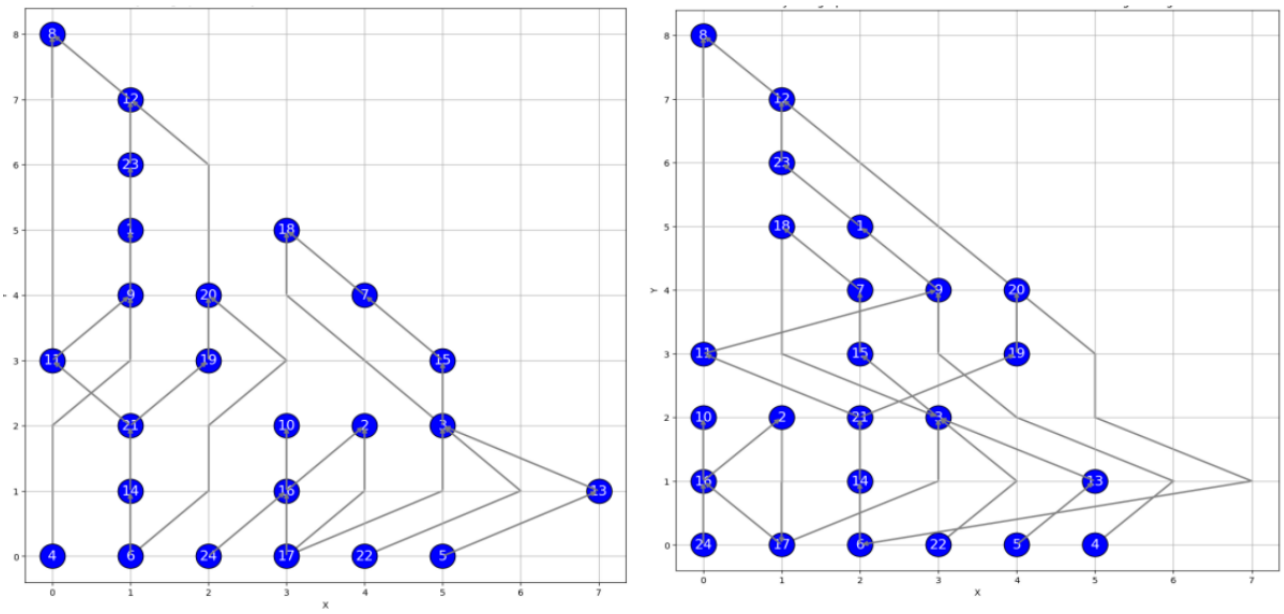


Figure 19. *Layered DAG without dummies after applying barycenter (left) and median (right)*

6.5 Reverse the edges

In this step we reverse the edges into their original direction.

- **visualize_layered_with_original_edge_directions** We modified the visualize_layered_with_crossing_without_dummies to reverse the edges into their original direction.

In *Figure* 20, we can observe the output graph after reversing the edges (2,17), (3,18), and (20,6) back to their original form. The graph reveals the cycles that were previously broken, indicating that we no longer have a DAG.

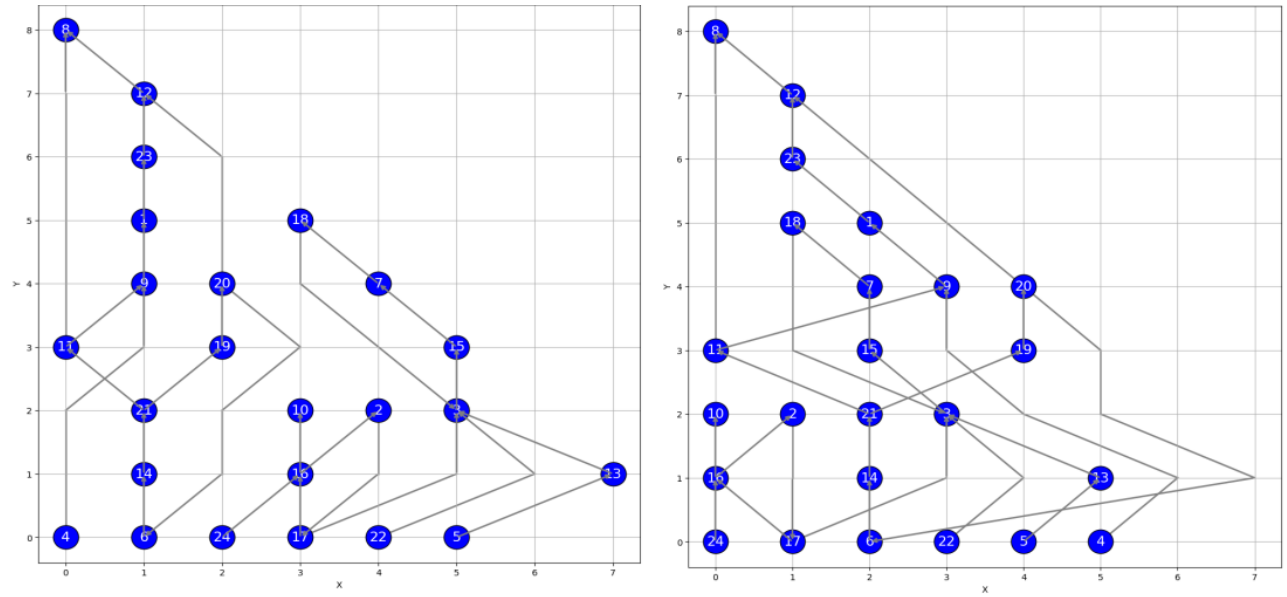


Figure 20. *Layered DAG without dummies with the original edges after applying barycenter (left) and median (right) for small directed network*

In the case of the Pro League network we reversed a total of 46 edges. The pro league graph has high density, which makes the observation challenging. However, in *Figure* 21 upon closer inspection, we notice distinct we noticed distinct differences between the two crossing minimization processes. With barycenter placement, vertices tend to follow a curved path. On the other hand, with median placement, vertices appear slightly shifted to the left and right. This deviation arises from the random selection made by the median method, which randomly determines which vertex with identical position and degree parity is placed in the left or right position.

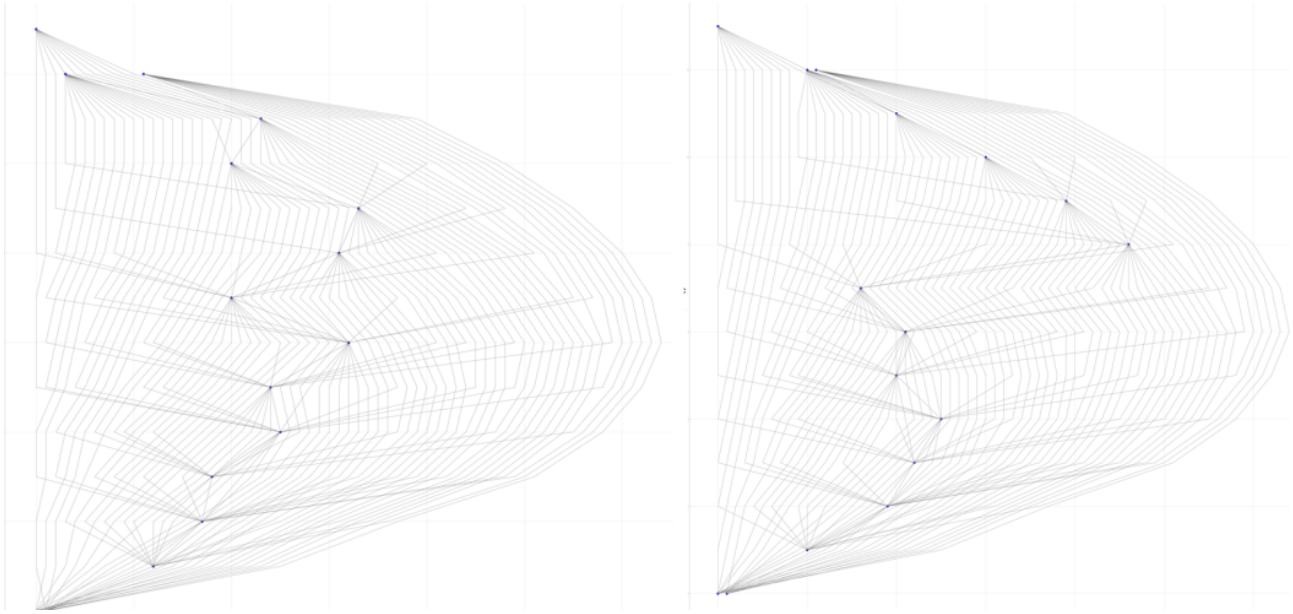


Figure 21. *Layered DAG without dummies with the original edges after applying barycenter (left) and median (right) for Pro league network*

6.6 Layered graph with curved edges (Bonus)

In this step, we utilized Bezier curves to smooth the edges and create curved edges instead of edges with angles. Moreover, as illustrated in Figure 22, we positioned the arrows in the middle of the edges to provide a clearer representation.

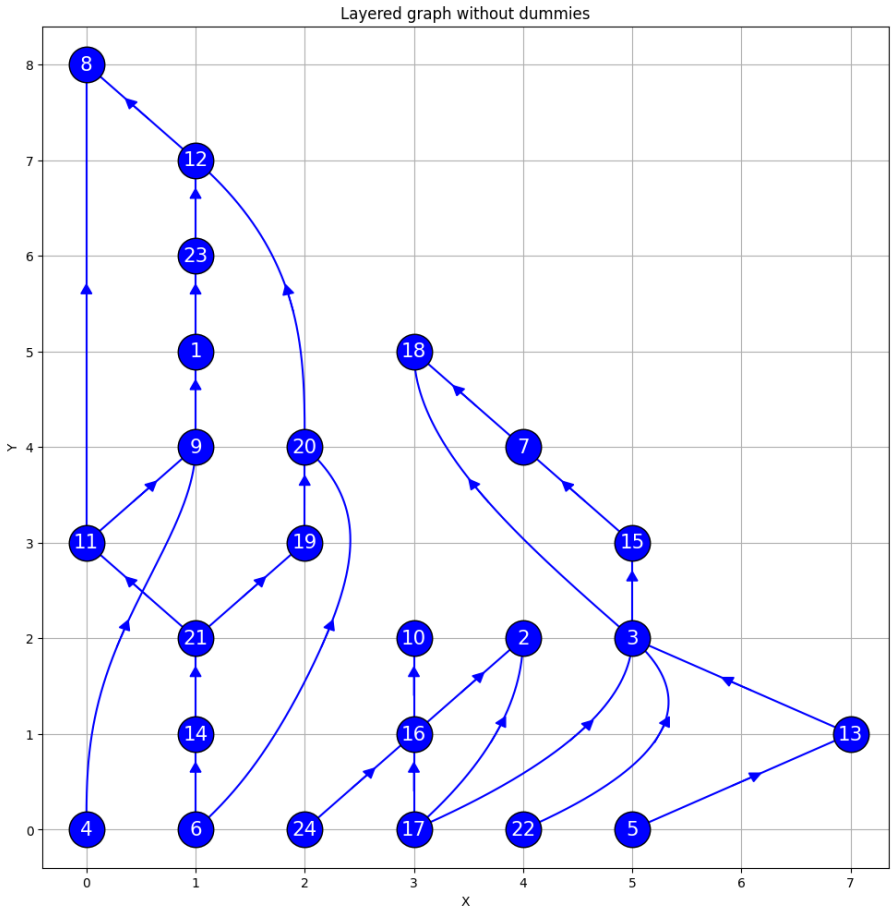


Figure 22. *Layered DAG curved edges for small directed network after applying barycenter*

6.7 Discussion

In this section we will discuss the efficiency of the layered graph using both Barycenter and Median to minimize the crossing of the edges.

Barycenter algorithm is determined in section 6.3.1 One reason behind selecting this algorithm is its simplicity. Additionally, it is relatively straightforward, leading to quick execution with satisfactory results. Furthermore, it can identify the optimal solution when the optimum crossing is 0.

Median algorithm is discussed in section 6.3.2 Similarly to the barycenter approach, the median algorithm is characterized by simplicity, straightforwardness, and efficiency, resulting in a commendable performance. Furthermore, it can identify the optimal solution when the desired outcome has zero crossings.

Both heuristics provide promising results. In small directed networks, all resulting layouts are easily readable due to the minimal number of edge crossings. As anticipated, layouts generated using the Median heuristic produce more edge crossings compared to those produced using the Barycenter heuristic. In denser networks like the pro league network, observing relationships between vertices becomes more challenging. However, we observed a significant reduction in the number of edge crossings, indicating the effectiveness of both heuristics.

7 Step 5: Multi-layer/clustered graphs and edge bundling

In this step, we took the Argumentation Network as our primary data source. First, we separate this graph using the function "separate_subgraphs", which partitions the dataset into two dictionaries and a list. The list "subgraph_vertices" and the dictionary "subgraph_edges", stores lists of vertices and edges for each of the subgraphs respectively. The second dictionary, "inter_subgraph_edges" contains lists of all the inter-subgraph edges. The key for the edges dictionary is the index of the subgraph, while the key for the inter-subgraph is a tuple representing the two sub-graphs connected by those edges.
Consequently, we iterate through all the subgraphs (with the corresponding edges and vertices), calculate their forced-directed graph, with the forced-directed algorithm from step 3 individually, and present each of them within a colored boundary box. Then for each pair of subgraphs, we apply the edge binding algorithm to smooth out these edges. This algorithm groups similar edges based on similarities and compatibility measurements. This algorithm gives a smoother and less cluttered appearance to the overall network. This algorithm has four compatibility measurements:

1. **visibility** - This compatibility considers the visibility of one edge from the perspective of other edges. The visibility metric helps determine how visible one edge is from the perspective of another edge, considering their positions and orientations.

2. **scale** - This compatibility measures how well the scales (lengths) of one edge compare to the scales of another edge. It helps determine whether two edges have similar or compatible lengths relative to each other.

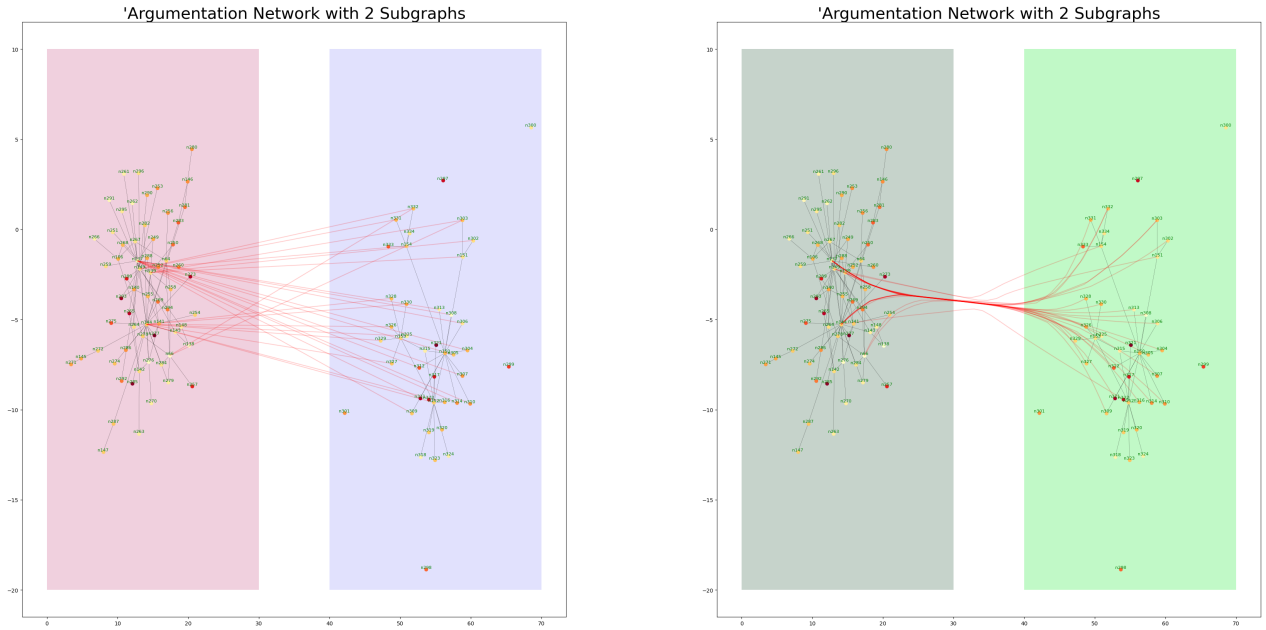
3. **angle** - This compatibility measures how well the orientations of one edge align with the orientations of another edge. It helps determine whether two edges have similar or compatible orientations relative to each other.

4. **position** -This compatibility measures how well the positions of the endpoints of one edge align with the positions of the endpoints of another edge. It helps determine whether two edges have similar or compatible positions relative to each other.

With the help of those compatibilities we can now adjust the inter-clusters (sub-graphs) edges, and visualize the connection between the subgraphs, in a more informative way. We can see (in *figure* 24b) that these two subgraphs have a strong relation between them. In *figure* 24a we still can see the relation, however, with large datasets with more than two subgraphs, finding those relations between them can be a challenging task.
To achieve those results, we used the following functions:

1. **separate_subgraphs**:

   - **Description:** This function takes a graph as input and separates it into subgraphs based on the clusters present in the graph. It identifies vertices belonging to each subgraph, collects edges within each subgraph, and identifies inter-subgraph edges connecting vertices from different clusters.
   - **Input:** A dot graph



(a) Argument Network before bundling *(b) Argument Network after bundling*

Figure 23. *Comparison of Argument Networks with two of the largest sub-graphs before and after bundling*

- **Output:**
  - A list of lists of vertices, where each inner list contains all the vertices belonging to a specific subgraph.
  - A dictionary of lists of edges, where each list contains all the edges belonging to a specific subgraph.
  - A dictionary of lists of edges, where each list contains all the inter subgraphs edges, between each pair of subgraphs.

2. **create_custom_graph_from_subgraph**:

- **Description:** This function takes the vertices and edges of a subgraph as input and creates a custom graph representation from them.
- **Input:**
  - vertices - A list of vertices belonging to the subgraph.
  - edges - A list of edges within the subgraph.
- **Output:** An instance representing the custom graph

3. **create_forced_subgraphs**:

- **Description:** This function takes a graph and computes the forced-directed graph layout in a given boundary box.
- **Input:**
  - ax - Matplotlib axes object for the first subplot.
  - graph - The input graph for which force-directed graph layout will be computed.
  - boundry_box - A dictionary containing the bounding box information, including keys 'min_x', 'min_y', 'width', and 'height'.
  - num_iterations - (optional, default=1000): The number of iterations to perform for computing the forced-directed layout.
  - Cspring - (optional, default=2.0): The spring constant used in the force-directed layout algorithm.
  - Crep - (optional, default=1.0): The repulsion constant used in the force-directed layout algorithm.
  - l - (optional, default=1.0): The ideal edge length used in the force-directed layout algorithm.
  - e - (optional, default=0.05): The threshold for stopping iterations when the maximum movement of vertices falls below this value.

- learning_rate - (optional, default=0.1): The learning rate used for updating vertex positions in each iteration.
- title - (optional, default=“”): The title of the subplot.

- **Output:** A list of the vertices of the current subgraph (with their coordinates).

4. **compact_edges** -

- **Description:** This function takes a list of inter-cluster edges, along with the corresponding subgraphs, and converts these edges into NumPy arrays. By converting the inter-cluster edges into NumPy arrays, the function prepares the data for edge bundling.
- **Input:**
  - cur_edges - A list of edges to be compacted.
  - subGraph1 - The first subgraph containing the vertices corresponding to the edges' source vertices.
  - subGraph2 - The second subgraph contains the vertices corresponding to the edges' destination vertices.
- **Output:** A NumPy array representing the compact edge representations.

5. **subdivide_edge** -

- **Description:** function subdivides each edge in a set of input edges into a specified number of points. It calculates the positions of the new points along each edge based on the desired number of points and the lengths of the segments between consecutive points.
- **Input:**
  - edges - A NumPy array representing the input edges to be subdivided.
  - num_points - The desired number of points to be generated along each edge, including the original endpoints.
- **Output:** A NumPy array containing the positions of the newly generated points along each edge.

6. **compute_edge_compatibility** -

- **Description:** This function calculates the edge compatibility measures for a set of input edges. It computes various compatibility metrics, including angle compatibility, length compatibility, distance compatibility, and visibility compatibility, to determine how well edges can be bundled together.
- **Input:**
  - edges - A NumPy array representing the input edges for which compatibility scores are to be computed.
- **Output:** A NumPy array containing the edge compatibility scores between each pair of edges.

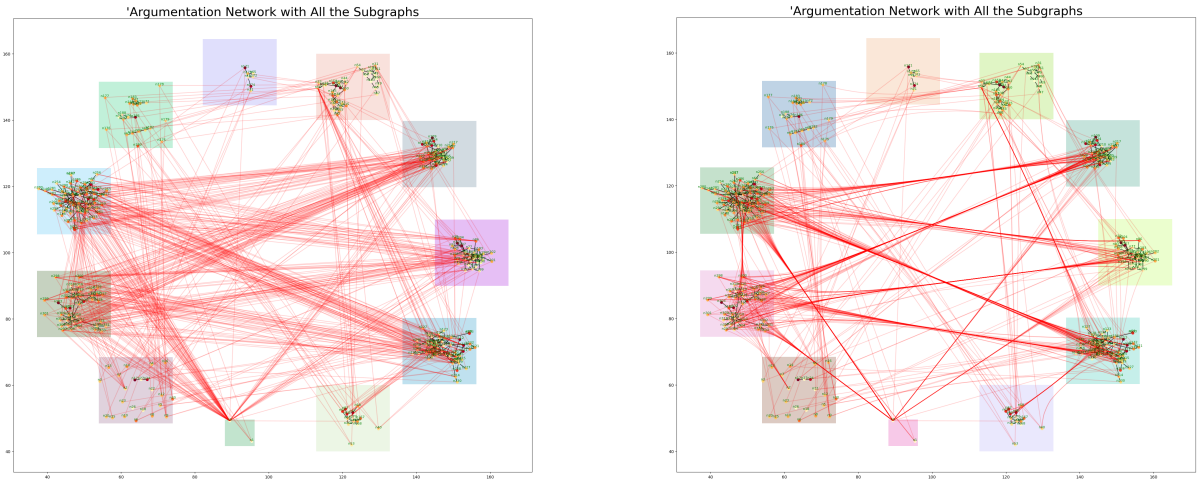
7. **compute_forces_new** -

- **Description:** function calculates the forces acting on each point (vertex) of the edges in a graph. It computes both spring forces and electrostatic forces based on the geometry and compatibility of the edges.
- **Input:**
  - edges - A NumPy array representing the edges in the graph.
  - e_compat - A NumPy array containing edge compatibilities, representing the compatibility between each pair of edges.
  - kp - A NumPy array representing the stiffness parameter controlling the strength of the spring forces.
  - electro_factor - (optional, default=0.275): A scalar factor controlling the influence of the electrostatic forces.
- **Output:** A NumPy array containing the forces acting on each point (vertex) of the edges.

8. **bundling_edges** -

- **Description:** This function takes a list of edges represented as NumPy arrays and applies the edge bundling algorithm to them. This function utilizes the functions "subdivide_edge", "compute_edge_compatibility" and "compute_forces_new" to implement the algorithm. "subdivide_edge" is used to divide each edge to the right amount of segments in each cycle while using "compute_edge_compatibility" and "compute_forces_new" to compute the compatibilities' measurements and apply them on each of the segments.
- **Input:**
  - edges - A NumPy array representing the edges in the graph.
  - K - (optional, default=0.1): The stiffness parameter controlling the strength of the spring forces during edge bundling.
  - n_iter - (optional, default=60): The number of iterations to perform during each bundling cycle.
  - n_iter_reduction - (optional, default=2/3): The reduction factor applied to the number of iterations after each cycle.
  - lr - (optional, default=0.04): The learning rate controlling the step size of the optimization process.
  - lr_reduction - (optional, default=0.5): The reduction factor applied to the learning rate after each cycle.
  - n_cycles - (optional, default=6): The number of bundling cycles to perform.
  - initial_segpoints - (optional, default=1): The initial number of segments used for edge subdivision.
  - segpoint_increase - (optional, default=2): The factor by which the number of segments increases after each cycle.
  - compat_threshold - (optional, default=0.05): The compatibility threshold is used to determine whether edges should be bundled.
- **Output:** A NumPy array containing the optimized positions of the edges after edge bundling.

Further in this step, we took the whole Argumentation Network and computed the edge bundling on all of the subgraphs resulting in the following graphs (*Figure* 24):



(a) Argument Network before bundling *(b) Argument Network after bundling*

Figure 24. *Comparison of Argument Networks before and after bundling*

As previously mentioned, when dealing with a large dataset containing more than two subgraphs, it becomes challenging to understand the relationships between them solely from *graph* a, *Figure* 24. This difficulty arises as the number of edges increases, making the graph difficult to interpret. However, in *graph* b, thanks to the edge bundling algorithm, the overall view remains unaffected by the number of edges. This feature allows for a clear understanding of the relationships between these subgraphs. Additionally, we chose to present these subgraphs in a circular form, further enhancing the visual clarity and facilitating the interpretation of inter-subgraph relationships.

8 Step 6: Projections for graphs

In this section, we delve into visualizing the Les Misérables, Jazz, and Pro League networks. Firstly, we employ the Floyd-Warshall algorithm to compute a distance matrix based on graph-theoretic distances, offering valuable insights into the connectivity patterns among the vertices. Leveraging this matrix, we project the graph onto two dimensions using multidimensional scaling (MDS) and t-distributed stochastic neighbor embedding (t-SNE) techniques. We import MDS and t_SNE from sklearn the manifold library [4,5]. Then, we tune various parameters to these models and plot the graph.

Additional class attributes required for this step:
For this step, we also added the weights as information about edges in class Edge.

8.1 Create the distance matrix

To create the distance matrix based on the graph-theoretic distances we created the floyd_warshall function. The Floyd-Warshall algorithm (see *Algorithm* 1) initializes the distance matrix by setting all the matrix elements to infinity. Then, it goes through every matrix element and updates them according to the weights of direct edges between respective vertices. After that, it sets the diagonal elements to zero. The algorithm optimizes path lengths between every pair of vertices through iterative updates, considering all vertices as potential intermediaries. This iterative process, grounded in dynamic programming principles, systematically evaluates and refines potential paths to compute the shortest distance matrix elements, represented as D[i,j]. Incorporating intermediary vertices enables the algorithm to refine distances efficiently. By iteratively assessing all possible paths and updating distances based on previously calculated shortest paths, the Floyd-Warshall algorithm yields a comprehensive distance matrix, accurately portraying optimal path lengths between all pairs of vertices upon completion.

Algorithm 1 Floyd-Warshall Algorithm

```
1: procedure FloydWarshall(G, V)
2:     for i ← 1 to V do
3:         for j ← 1 to V do
4:             if i = j then
5:                 dist[i][j] ← 0
6:             else
7:                 dist[i][j] ← ∞
8:             end if
9:         end for
10:    end for
11:    for each edge (u, v) ∈ G do
12:        dist[u][v] ← weight of (u, v)
13:    end for
14:    for k ← 1 to V do
15:        for i ← 1 to V do
16:            for j ← 1 to V do
17:                if dist[i][k] + dist[k][j] < dist[i][j] then
18:                    dist[i][j] ← dist[i][k] + dist[k][j]
19:                end if
20:            end for
21:        end for
22:    end for
23:    return dist
24: end procedure
```

8.2 Projection using MDS

Multidimensional scaling (MDS) aims to find a lower-dimensional representation of data while preserving the distances between data points from the original high-dimensional space. Broadly, MDS serves as a method for analyzing similarity or dissimilarity data, endeavoring to interpret such data as distances within a geometric space. MDS algorithms can be categorized into two main types: metric and non-metric. In metric MDS, where the input distance matrix follows a metric (in our case from theoretical distances), the distances between pairs of points in the output space are optimized to match the similarity or dissimilarity data points. Conversely, in the non-metric variant, the algorithms aim to maintain the ordinal relationship between distances, thereby seeking a monotonic correspondence between the distances in the embedded space and the similarities or dissimilarities [6]. Moreover, the goal of the algorithm is to minimize stress which is computed using the following function:

$$\sum_{i=1}^{N}\sum_{j=1}^{N}(d_{ij}^{(d)} - d_{ij}^{(m)})^2$$

Where $d_{ij}^{(d)}$ is the distance between samples i and j in the high-dimensional space, and $d_{ij}^{(m)}$ is the distance between samples i and j in the low-dimensional space.
In our implementation we used the metric approach. The parameters that we used in the MDS model are the following:

- *n_components*: The number of dimensions to represent the dissimilarities.
- *dissimilarity*: a variable that is used to define whether you want the model to compute the distances between the data points (value 'Euclidean'), or to pass the pre-computed distance matrix(value 'precomputed').
- *max_iter*: the maximum iterations of the algorithm.
- *epsilon*(*eps*): defines the threshold for the change in stress or error that indicates when the optimization algorithm should stop iterating and consider the solution as sufficiently close to the optimal one. Essentially, it controls how accurately the MDS algorithm tries to fit the data into the lower-dimensional space.

8.2.1 Les Misérables network

After conducting a series of experiments involving the manipulation of the variable eps, as depicted in *Figures* 25, 26, and 27, we noted a significant visual change from eps=10 to eps=0.1 as well as a change in stress which was around 10698 for eps=10, around 6181 for eps=1, and around 3400 for eps=0.1. Also, we observed that visual alterations in the graph were imperceptible for eps values below 0.1. However, an improvement emerged upon setting eps to 0.0001. This adjustment yielded a stress value of approximately 3031 after 164 iterations, representing a slight enhancement over the stress attained with eps set to 0.1 (approximately 3400). Nevertheless, due to inherent randomness in the model, the stress values ranged from 3000 to 3300. Additionally, the remaining variables were set as follows: n components=2 (to obtain a two-dimensional representation), dissimilarity='pre-computed' (to utilize the distance matrix), and max iter=200. The algorithm continues to run until either the maximum number of iterations is reached or until the data is fitted well enough based on the epsilon (eps) threshold. In our experiments, the model typically converged within 120-250 iterations. Also, we did not notice any discernible visual changes in the plotted graph beyond the 200th iteration. Hence, we opted to set the maximum number of iterations to 200. As we can see in *Figure* 28 the graph is represented in a readable visualization where most of the vertices are efficiently clustered.

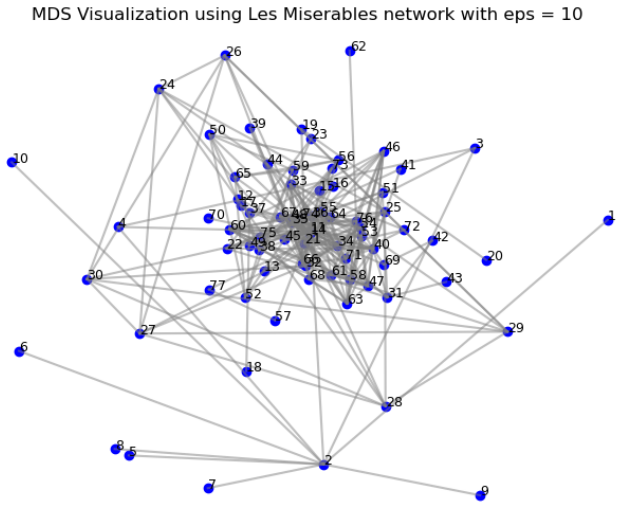


Figure 25. *MDS:Les Misérables visualization with eps = 10, resulted in stress = 10698 in 5 iterations*

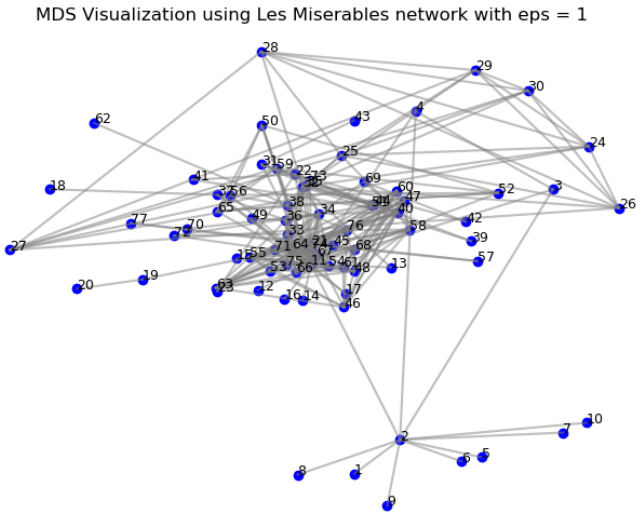


Figure 26. *MDS:Les Misérables visualization with eps = 1, resulted stress = 6181 in 13 iterations*

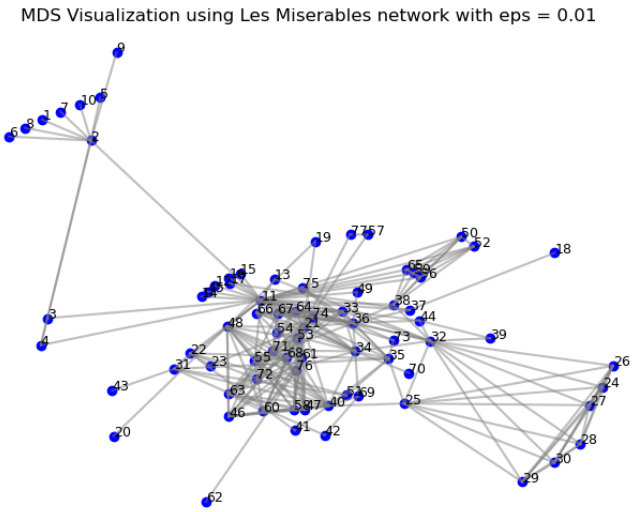


Figure 27. *MDS:Les Misérables visualization with eps = 0.1, resulted in stress = 3400 in 44 iterations*

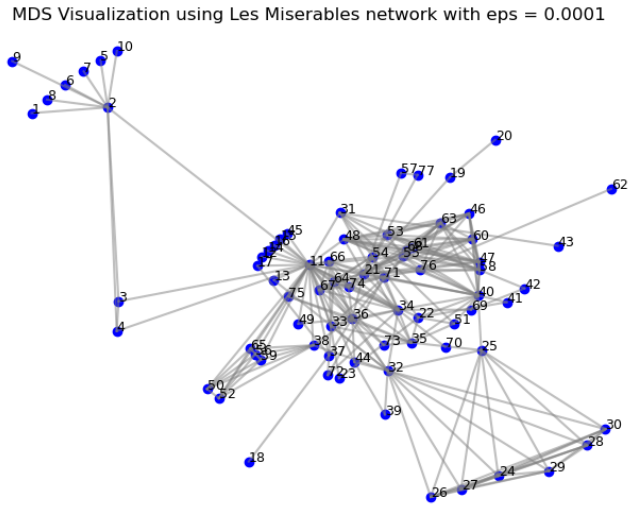


Figure 28. *MDS:Les Misérables visualization with eps = 0.0001, resulted in stress = 3031 in 164 iterations*

8.2.2 Jazz network

We also employed the MDS projection to visualize the Jazz network. Again, we tested different values for the *eps* threshold and observed the visual results. As can be inferred from the results of *Figures* 29, 30, 31, 32 and 33, neither visual changes nor significant optimization in stress value was observed. For this reason, we decided to set the maximum number of iterations to 400 and the *eps* threshold to 0.0001.

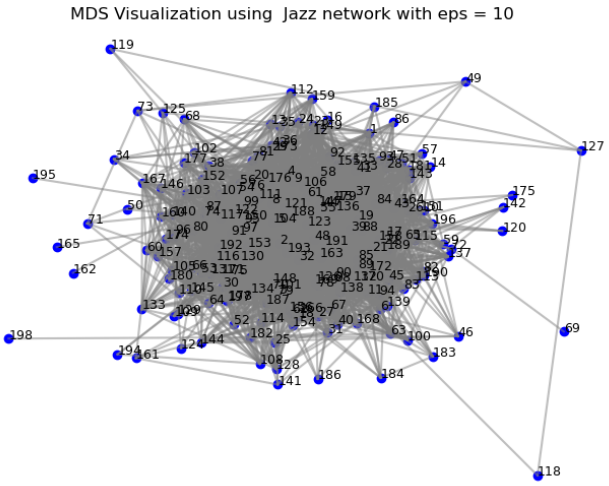


Figure 29. *MDS: Jazz network with eps = 10, resulted in stress = 19583 in 4 iterations*

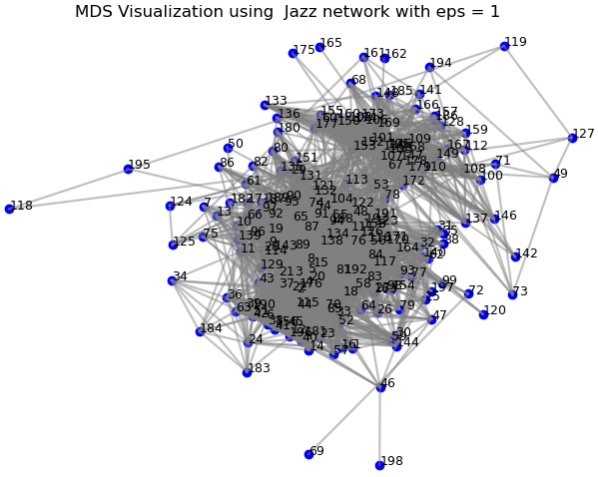


Figure 30. *MDS: Jazz network with eps = 1, resulted in stress = 11716 in 34 iterations*

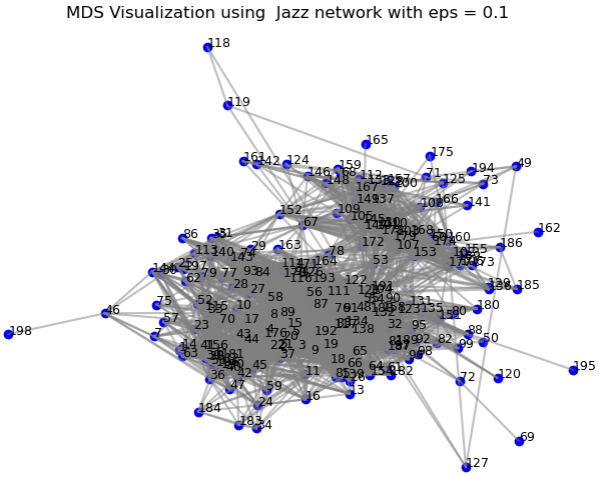


Figure 31. *MDS: Jazz network with eps = 0.1, resulted in stress = 9400 in 62 iterations*

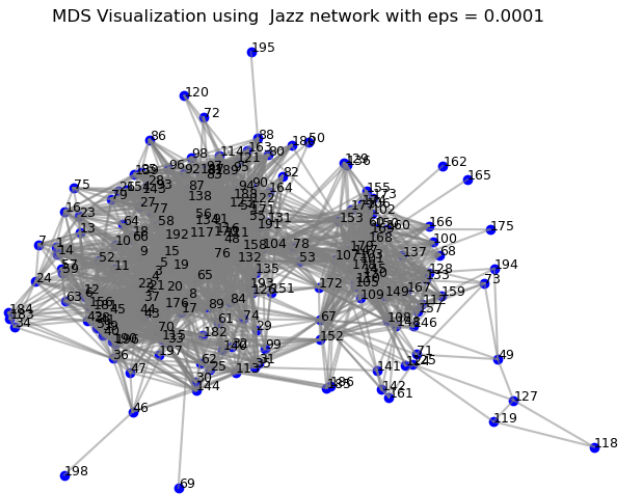


Figure 32. *MDS: Jazz network with eps = 0.0001, resulted in stress = 8045 in 342 iterations*

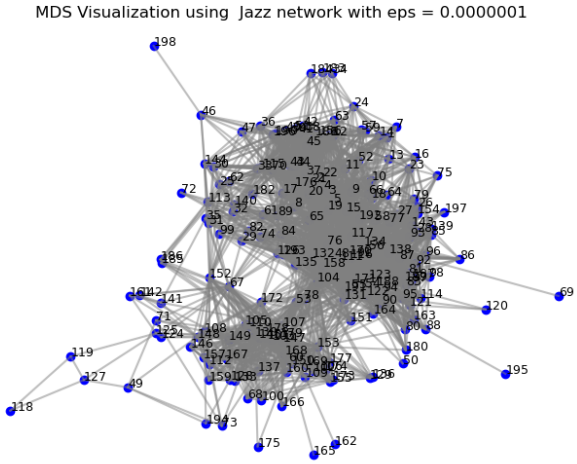


Figure 33. *MDS: Jazz network with eps = 0.0000001, resulted in stress = 7956 in 461 iterations*

8.2.3 Pro league network

Following the previous network analysis, we applied MDS projection to visualize the Pro League network, a small yet densely connected network consisting of only 16 vertices but connected with 239 edges, making it challenging to visualize. To overcome this challenge, we filtered its edges based on their weights, including only those with weights ranging from 1 to 3.

After applying MDS with various thresholds for *eps*, we observed from the *Figures* 34, 35, 36, 37 and 38 that there were no significant visual differences or notable changes in stress values. Even though we noticed a decrease in stress after decreasing the threshold and increasing iterations, the stress seemed to stabilize around the value of 30. Consequently, we decided to set the parameters based on the optimal stress value we observed, which was achieved within 95 iterations and an *eps* threshold of 0.0001.

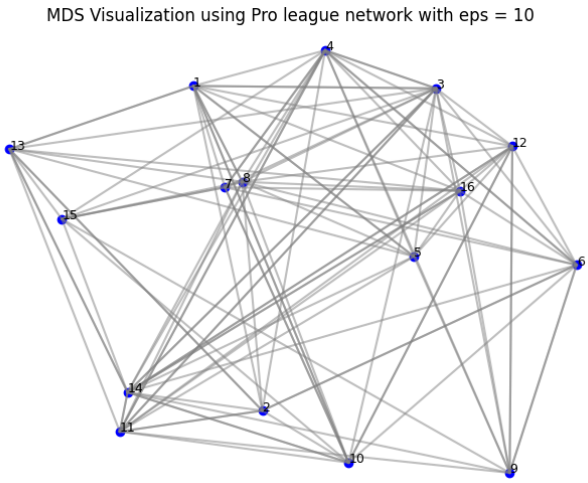


Figure 34. *MDS: Pro league network with eps = 10, resulted in stress = 62 in 2 iterations*

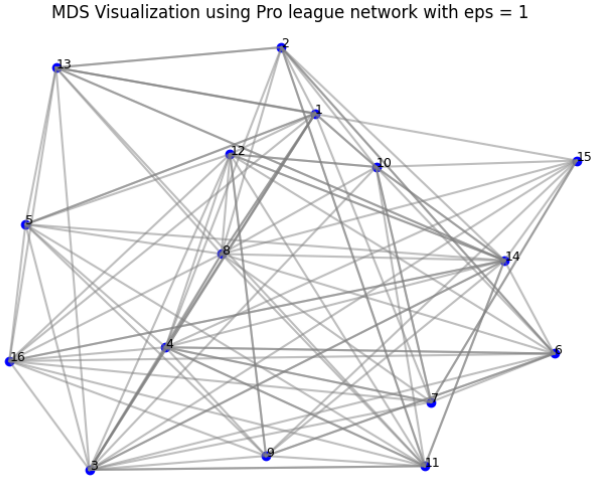


Figure 35. *MDS: Pro league network with eps = 1, resulted in stress = 61 in 3 iterations*

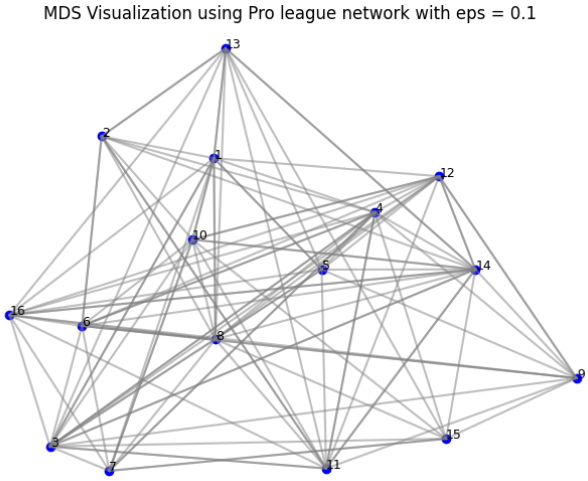


Figure 36. *MDS: Pro league network with eps = 0.1, resulted in stress = 48 in 12 iterations*

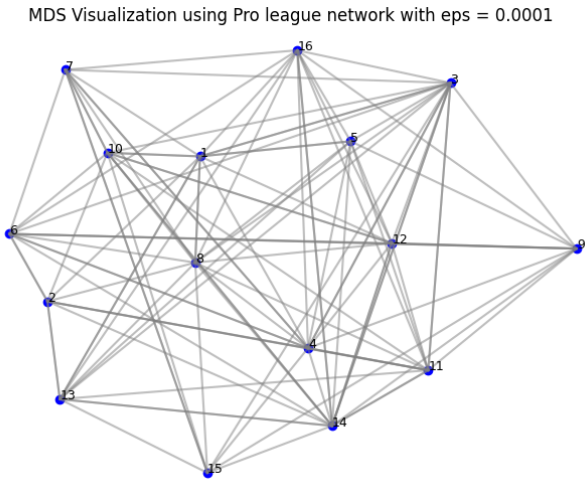


Figure 37. *MDS: Pro league network with eps = 0.0001, resulted in stress = 29 in 95 iterations*

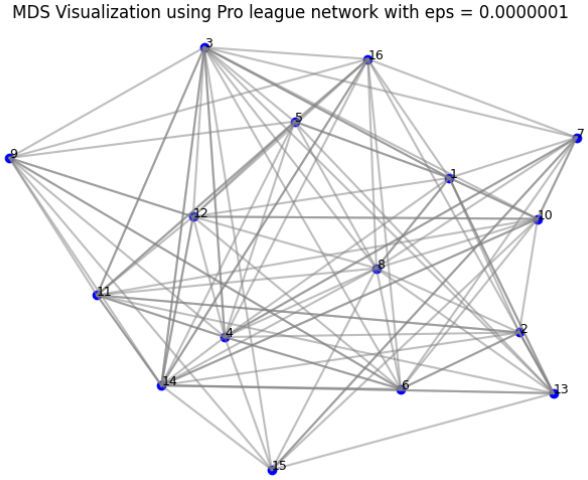


Figure 38. *MDS: Jazz network with eps = 0.0000001, resulted in stress = 29 in 186 iterations*

8.3 Projection using t-SNE

T-SNE is a technique used for visualizing data with many dimensions. It transforms similarities among data points into joint probabilities and aims to decrease the Kullback-Leibler divergence between the joint probabilities of the lower-dimensional representation and the original high-dimensional data. One notable characteristic of t-SNE is its non-convex cost function, meaning that varying initial setups can lead to varying outcomes [8]. The cost function consists of a sum of Kullback-Leibler divergences over all data points and is computed as follows:

$$C = \sum_i KL(P_i||Q_i) = \sum_i \sum_j p_{ij} \log \frac{P_{ij}}{q_{ij}}$$

where P_i represents the conditional probability distribution over all other data points given data point x_i in the high-dimensional space, Q_i represents the conditional probability distribution over all other map points given map point y_i in the low-dimensional space, while p_{ij} and q_{ij} the conditional probabilities between the high-dimensional data points x_i , x_j and low-dimensional data points y_i , y_j respectively [8]. Consequently to minimize the cost function aims to find low-dimensional data where $p_{ij} = q_{ij}$ and therefore $\log \frac{P_{ij}}{q_{ij}} = 0$.

The parameters that we choose to use in the t-SNE visualization are the following:

- *n_components*: The number of dimensions to represent the dissimilarities.
- *metric*: a variable that is used to define whether you want the model to compute the distances between the data points (value 'Euclidean'), or to pass the pre-computed distance matrix (value 'precomputed').

- *init*: parameter that determines the initialization strategy for the optimization algorithm. It specifies how the algorithm should initialize the embedding of the data points in the low-dimensional space before optimizing their positions.
- *perplexity*: a parameter that influences the optimization process and the resulting embedding and controls the effective number of neighbors that each data point considers during the optimization process.
- *n_iter*: the number of iterations the optimization algorithm will run.
- *learning_rate*: the step size used in the optimization algorithm to update the positions of data points in the low-dimensional space during each iteration.
- *min_grad_norm*: a threshold that specifies the minimum norm of the gradient vector at which the optimization process should terminate.

Our parameters values for this implementation are n_components=2,init="random",metric="precomputed", perplexity = 35, n_iter = 5000, learning_rate = 50 ,min_grad_norm $=1 \times 10^{-10}$ .Among the various parameters examined, the learning rate and perplexity emerged as the most influential in achieving optimal results (see *Figures* 39, 40, 41 and 42). Specifically, we found that a learning rate of 50 and a perplexity of 35 consistently yielded the best visual representation after 1849 iterations (see *Figure* 43). This representation further underscored a minimal divergence of approximately 0.13. Furthermore, it is worth noting that since the initialization was set to "random", there is inherent randomness in the results. However, after running the model multiple times, the observed divergence ranged from 0.1 to 0.25, while the visual representation remained consistent.

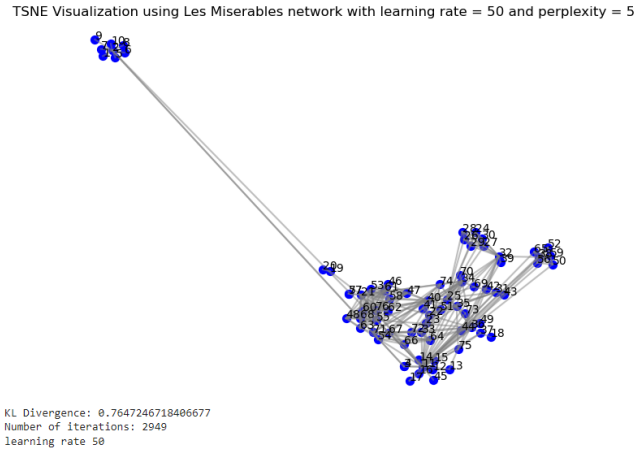


Figure 39. *TSNE Visualization using Les Misérables network with learning rate = 50 and perplexity = 5*

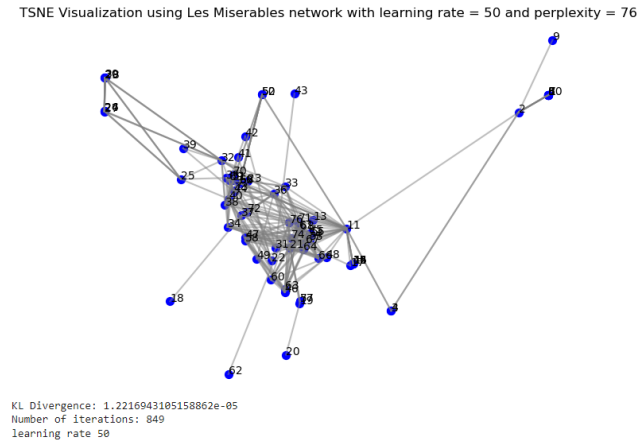


Figure 40. *TSNE Visualization using Les Misérables network with learning rate = 50 and perplexity = 76*

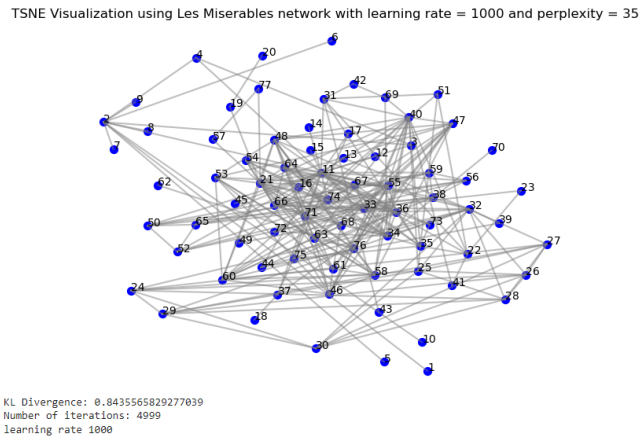


Figure 41. *TSNE Visualization using Les Misérables network with learning rate = 1000 and perplexity = 35*

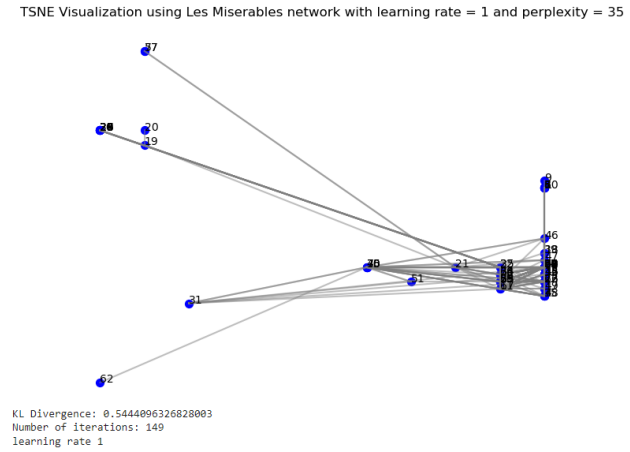


Figure 42. *TSNE Visualization using Les Misérables network with learning rate = 1 and perplexity = 35*

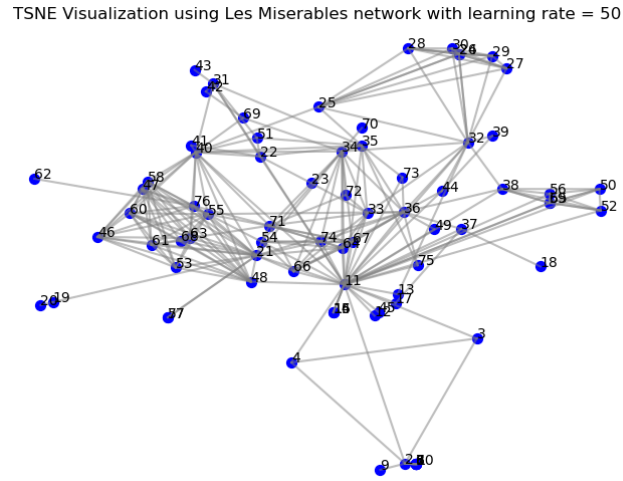


Figure 43. *TSNE Visualization using Les Misérables network with learning rate = 50 and perplexity = 35*

8.3.1 Jazz network

For visualizing the Jazz network, we adopted the same parameter values as those used for the Les Misérables network, except for the perplexity parameter. Given that the Jazz network is larger, comprising 198 vertices and 20742 edges, we adjusted the perplexity accordingly. *Figures* 44, 45, 46, and 47 present the visualizations generated using different perplexity values. Although higher perplexity values (e.g., *perplexity* = 190) optimize the divergence score (as shown in *Figure* 47), the resulting visualization appears cluttered due to the proximity

of vertices, leading to numerous overlaps. Conversely, the projection illustrated in *Figure* 46 offers a more coherent visualization despite exhibiting higher divergence.

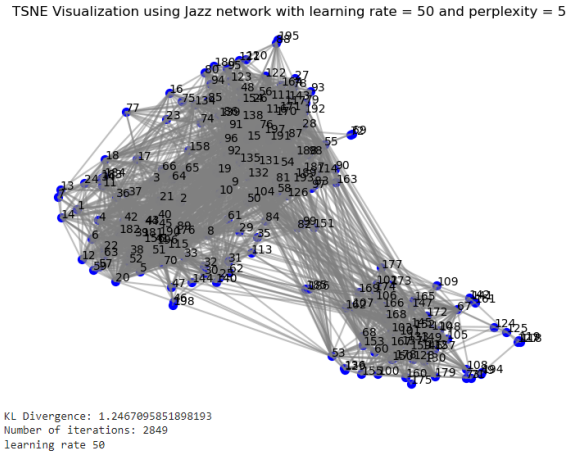


Figure 44. *TSNE Visualization using Jazz network with learning rate = 50 and perplexity = 5*

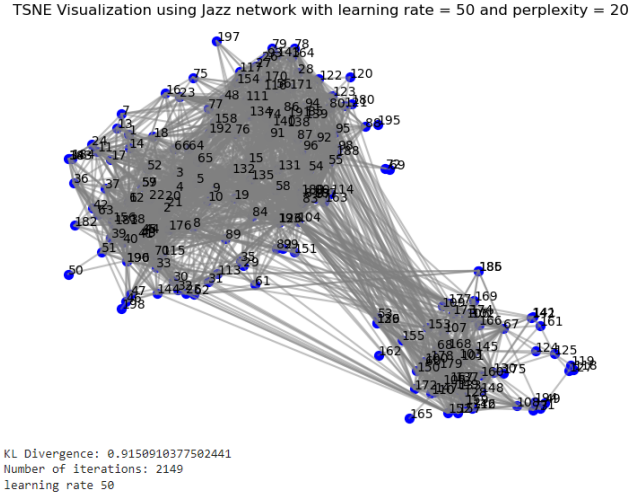


Figure 45. *TSNE Visualization using Jazz network with learning rate = 50 and perplexity = 20*

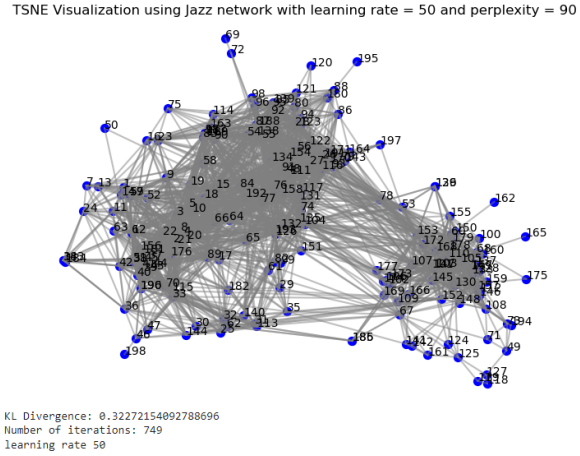


Figure 46. *TSNE Visualization using Jazz network with learning rate = 50 and perplexity = 90*

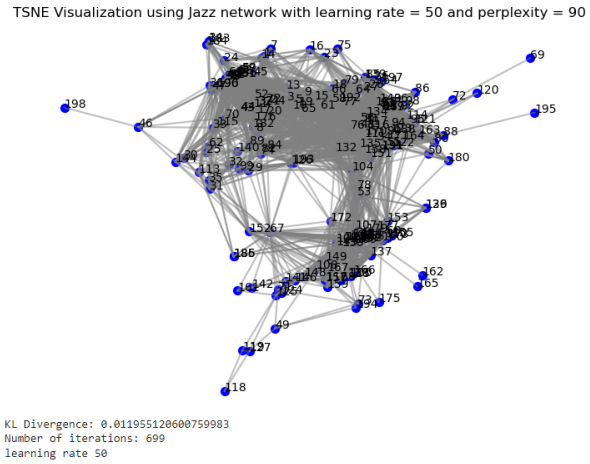


Figure 47. *TSNE Visualization using Jazz network with learning rate = 50 and perplexity = 190*

8.3.2 Pro league network

For this network, we utilized the same parameters as the two previous networks, except for the perplexity. Since the pro league network consists of 16 vertices, we explored perplexity values ranging from 1 to 15. From the output graphs presented in *Figures* 48, 49, 50, and 51, similar to the outputs of MDS, we observed no significant differences in the visualization, which can be occurred due to the small number of vertices in the network. Regarding divergence, the best result we observed was 0.31, achieved with a learning rate of 50, 1599 iterations, and a perplexity of 10 (see *Figure* 50).

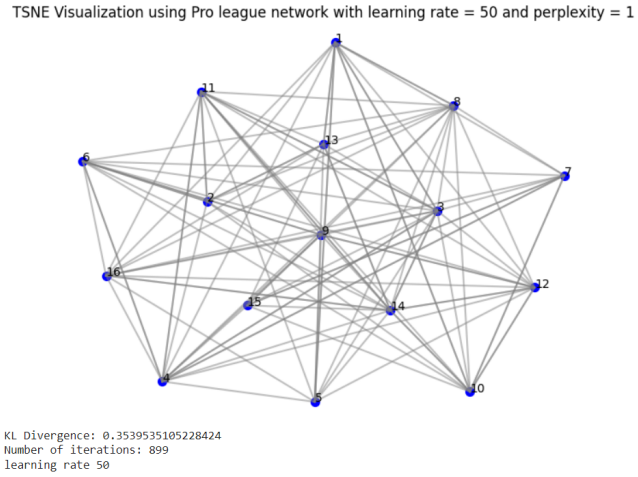


Figure 48. *TSNE Visualization using Pro league network with learning rate = 50 and perplexity = 1*

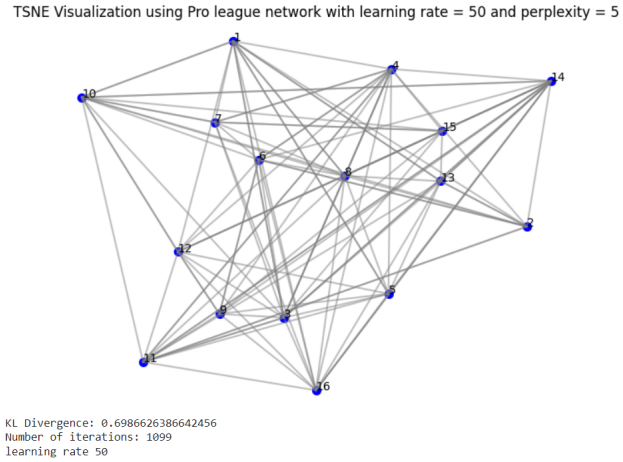


Figure 49. *TSNE Visualization using Pro league network with learning rate = 50 and perplexity = 5*

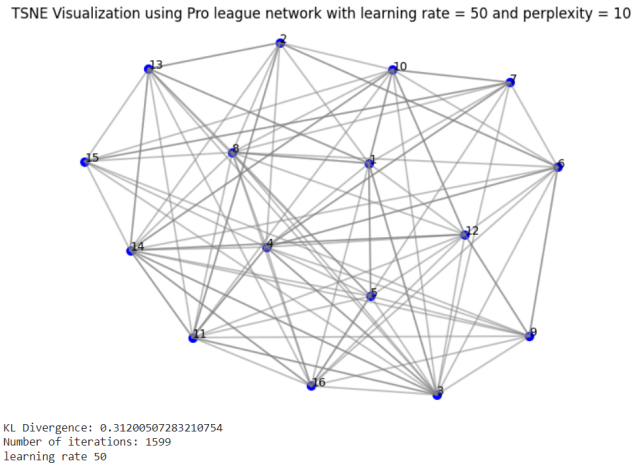


Figure 50. *TSNE Visualization using Pro league network with learning rate = 50 and perplexity = 10*

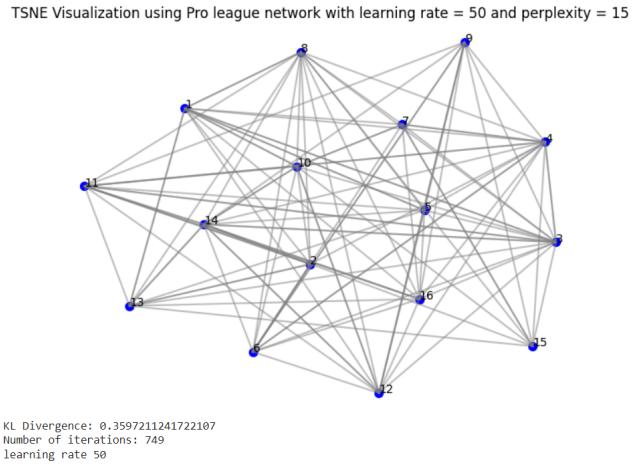


Figure 51. *TSNE Visualization using Pro league network with learning rate = 50 and perplexity = 15*

8.4 Discussion

Upon comparing the projections generated by MDS and t-SNE for the Les Misérables dataset, it becomes evident that both models yield visually appealing outcomes, as demonstrated in *Figures* 28 and 43. Both MDS and t-SNE exhibit decent clustering performance, effectively grouping the majority of vertices. However, it's noteworthy that MDS achieved optimal results within 164 iterations, whereas t-SNE required significantly more iterations, specifically 1849, to achieve its best visual representation. This substantial difference in convergence time may carry significant implications, particularly when dealing with considerably large datasets.
Nevertheless, when applied to the Jazz network, which is a larger dataset, t-SNE produced a superior visualization result compared to MDS, as the vertices are clustered more distinctly (see *Figures* 46 and 32). Furthermore, t-SNE achieved this with fewer iterations than its application on the Les Misérables dataset, concluding after 749 iterations. Although t-SNE still required approximately twice as many iterations as MDS, the difference in iteration counts between the two models decreased when applied to the Jazz network dataset.

9 Step 7: Quality measurement of graph projections

In this step we aim to compare the graphs produced by force-directed layout (step 3), layered layout (step 4), and the projections produced in step 6. For the comparison, we will construct various graph-related metrics (crossing number, crossing resolution, stress of layout) and some projection-related metrics (continuity, trustworthiness, stress, Shepard diagram, and Shepard goodness score).

9.1 Graph related metrics

Crossing number: To find the number of crossings in a graph we implemented the "count_crossings" function. The "count_crossings" function is designed to determine the number of intersections among a set of line seg-

ments represented by the edges parameter. Internally, the function employs a nested loop structure to compare each pair of line segments for potential intersections. It iterates over all pairs of edges, assessing whether any intersection occurs between them by calling the "intersect" function. If a pair of edges, e.g., edge1 and edge2, was checked, the combination of edge2 and edge1 will not be checked again. The "intersect" function determines whether two line segments intersect by considering the orientations of the segments and their respective endpoints.

Crossing resolution: Crossing resolution in a graph, is the smallest angle that occurs from crossings. For this purpose, we created the "crossing_resolution" function. In its operation, the function iterates over all possible combinations of edges if they intersect. If an intersection is detected, the function creates a vector from the start and end points of the edges and computes the angle formed by the intersecting segments by passing those vectors to the "angle_between" function. Throughout the computation, the function keeps track of the smallest angle encountered, representing the minimum degree of intersection among all pairs of intersecting edges. This minimum angle serves as an indicator of the overall angular resolution of edge crossings within the graph or diagram and offers valuable insight into the arrangement and layout of the edges. It aids in the evaluation and optimization of graphical representations to improve visual clarity and interpretability.

Stress of layout: For the calculation of the stress of layout we developed a function called calculate_stress_of_layout. The function is based on the following formula:

$$\sum_{i<j} \frac{1}{d_{ij}^2} \left(|X_i - X_j| - d_{ij}\right)^2 \tag{4}$$

where d_{ij} is the ideal (euclidean distance) between i and j and $|X_i - X_j|$ is the actual distance between i and j. The first step of the calculation of the stress of layout is to set the weights of every edge of the graph to the Euclidean distance of the coordinates of the edge start point and end point (so we set the edge' weights to the actual edge length in the layout). Then we are using the floyd_warshall algorithm that we implemented in step 6 to retrieve the table with the shortest paths according to the Euclidean distance of the graph vertices (finds the path from vertex i to j that has the smallest sum of distances according to the real length of the edges). Finally, in the calculate_stress_of_layout function we iterate through each item i,j of the table (retrieved by floyd_warshall algorithm), where $i < j$ and compute the stress by the formula (4).

9.2 Projection-related metrics

Continuity: Measures how well the relationships between points in the high-dimensional space are preserved in the low-dimensional space. It assesses whether neighboring points in the high-dimensional space remain close to each other in the low-dimensional space. For computing the continuity we created the "continuity" function which is based on the following formula:

$$C = 1 - \frac{2}{NK(2n - 3K - 1)} \sum_{i=1}^{N} \sum_{j \in V_i(K)} \left(r(i,j) - K\right) \tag{5}$$

where N is the number of vertices, K is the parameter representing the number of nearest neighbors, $V_i(K)$ is the set of neighbors of vertex i in the high-dimensional space but not in the low-dimensional space, and $r(i,j)$ is the rank of vertex j in the list of neighbors of vertex i in the low-dimensional space.
The "continuity" function, takes as inputs the low distances (D_low), the high dimensional distances (D_high), and the number of nearest neighbors (K). In its operation, the function computes the nearest neighbors for each point in both high-dimensional and low-dimensional spaces, and then for each point, find the neighbors that are present in the high-dimensional space but not in the low-dimensional space. Finally, it calculates the ranks of these neighbors and uses the formula (5) to compute the continuity metric.

Trustworthiness: Measures how well the relationships between points in the low-dimensional space are consistent with the high-dimensional space. It assesses whether neighboring points in the low-dimensional space accurately represent neighboring points in the high-dimensional space. For computing the continuity we created the "Trustworthiness" function which is based on the following formula:

$$T = 1 - \frac{2}{NK(2n - 3K - 1)} \sum_{i=1}^{N} \sum_{j \in V_i'(K)} \left(r'(i,j) - K\right) \tag{6}$$

where N is the number of vertices, K is the parameter representing the number of nearest neighbors, $V_i'(K)$ is the set of neighbors of vertex i in the low-dimensional space but not in the high-dimensional space, and $r'(i,j)$

is the rank of vertex j in the list of neighbors of vertex i in the high-dimensional space. The "Trustworthiness" function takes the same inputs as the "continuity" function and works similarly with the only difference that for each point, find the neighbors that are present in the low-dimensional space but not in the high-dimensional space, and use the equation (6) to compute the trustworthiness metric.

Normalised stress: It is calculated through the "normalized_stress" function that takes the distances in the high-dimensional space and the distances in low-dimensional space as an input and computes the normalized stress metric according to the following equation:

$$textNormalizedStress = \frac{\sum_{i,j}(d_{ij}^{(h)} - d_{ij}^{(l)})^2}{\sum_{i,j}(d_{ij}^{(h)})^2} \tag{7}$$

where $d_{ij}^{(h)}$ is the distance between points i and j in the high-dimensional space, and $d_{ij}^{(l)}$ is the distance between points i and j in the low-dimensional space.

Shepard diagram and goodness score: The Shepard diagram is a visualization tool employed to assess the preservation of distances between points in the original high-dimensional space when projected onto a lower-dimensional space. To quantify the fidelity of this preservation, the Shepard goodness score is computed using the Pearson correlation coefficient from the "scipy.stats" library.

9.3 Results

In this subsection we will compare both the visualization outcomes and the metric values for three distinct graphs: "Les Misérables", "JazzNetwork" and "LeagueNetwork".

9.3.1 Les Misérables network

The graph-related metric outcomes for the "Les Misérables" graph are the following:

Total crossings: (refer to *Figure* 54 for the visual representation of the graph)

- Force-directed layout total crossings: 905
- Layered layout total crossings: 1844
- MDS projection total crossings: 1142
- t-SNE projection total crossings: 951

Crossing resolution: (refer to *Figure* 54 for the visual representation of the graph)

- Crossing resolution in Force-directed graph: 3.58
- Crossing resolution in Layered graph: 1.21
- Crossing resolution in MDS projection: 0.53
- Crossing resolution in t-SNE projection: 6.20

Stress of layout: (refer to *Figure* 54 for the visual representation of the graph)

- Stress of layout in Force-directed graph: 886.69
- Stress of layout in Layered graph: 9218923.74
- Stress of layout in MDS projection: 1389.16
- Stress of layout in t-SNE projection: 61885.07

The projection-related metric outcomes for "Les Misérables" in MDS and t-SNE projections are the following:

Shepard score:

- Shepard Goodness score for MDS: 0.94 (See also the Shepard diagram in *Figure* 52)
- Shepard Goodness score for t-SNE: 0.85 (See also the Shepard diagram in *Figure* 53)

Trustworthiness:

- Trustworthiness for MDS: 0.89
- Trustworthiness for t-SNE: 0.94

Normalized stress:

- Normalized Stress for MDS: 0.03
- Normalized Stress for t-SNE: 0.12

Continuity:

- Continuity for MDS: 0.92
- Continuity for t-SNE: 0.93

Discussion:
As we can observe from the results of the graph-related metrics, the Forced directed layout seems to outperform all the other layouts in two of the three metrics. Particularly, it excels in "Total Crossings" and "Stress of Layout", while securing the second-best position in "Crossing Resolution". Conversely, the Layered layout yields the least favorable results, obtaining the lowest scores in both "Total Crossings" and "Stress of Layout", and the second-lowest score in "Crossing Resolution". The notable discrepancy in stress levels observed in the Layered layout may be attributed to the intricate pathing of edges traversing numerous dummy vertices before reaching their destination. From a visual perspective, as we can notice from *Figure* 54), most of the vertices in the Forced-directed layout, are well grouped into clusters and visualized clearly. Also, despite the unfavorable outcomes of the Layered layout, the visual presentation remains remarkably clear, as it facilitates easy tracking of lines. When considering the projection metrics, let us compare the MDS and t-SNE models. Upon examining the metrics outputs, it becomes evident that the two models exhibit no significant differences except for the Normalized Stress value. MDS Normalized Stress value (0.03) is four times lower than in t-SNE (0.12). However, despite this discrepancy, both visualizations offer a similar level of readability, making this observation reasonable.

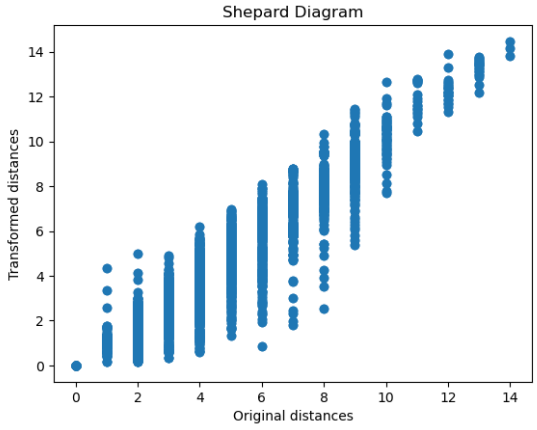


Figure 52. *MDS Shepard diagram Les Misérables*

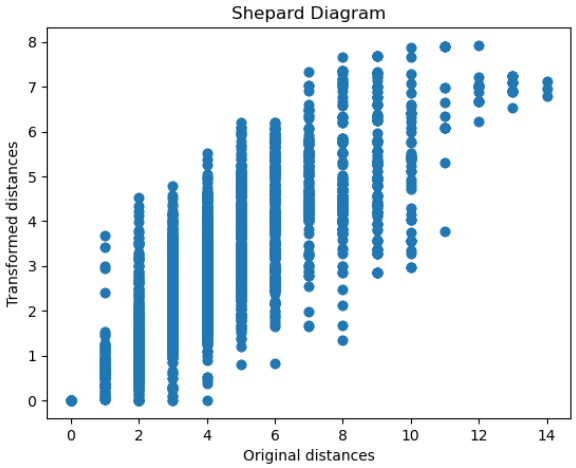


Figure 53. *t-SNE Shepard diagram Les Misérables*

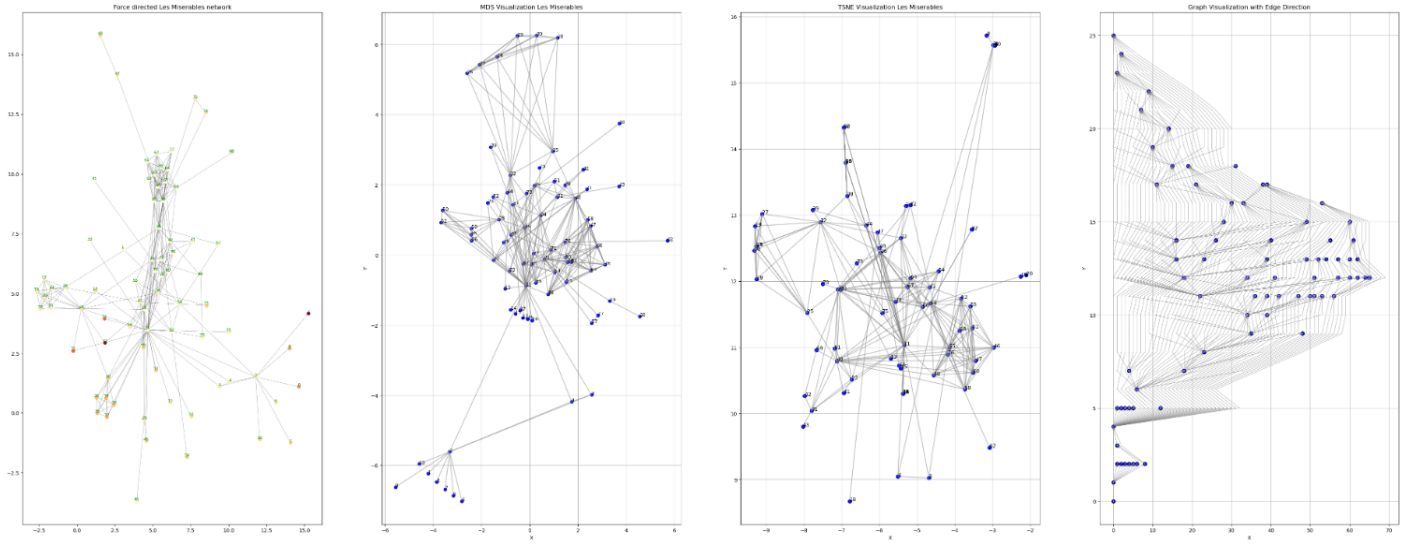


Figure 54. *First image (from left to right: Force Directed layout - second image: MDS projection - third image t-SNE projection - fourth image: Layered layout - for Les Misérables network*

9.3.2 LeagueNetwork network

For "LeagueNetwork" we need to address the density, which complicates visualization. Additionally, we want to ensure that the weights of edges are within a certain range, as negative weights are undesirable. Consequently, we have decided to include only edges with weights between 1 and 3. The graph-related metric outcomes for the "LeagueNetwork" graph are the following:

Total crossings: (refer to *Figure* 57 for the visual representation of the graph)

- Force-directed layout total crossings: 632
- Layered layout total crossings: 612
- MDS projection total crossings: 846
- t-SNE projection total crossings: 951

Crossing resolution: (refer to *Figure* 57 for the visual representation of the graph)

- Crossing resolution in Force-directed graph: 12.58
- Crossing resolution in Layered graph: 2.85
- Crossing resolution in MDS projection: 6.30

- Crossing resolution in t-SNE projection: 3.97

Stress of layout: (refer to *Figure* 57 for the visual representation of the graph)

- Stress of layout in Force-directed graph: 0.21
- Stress of layout in Layered graph: 3032703.90
- Stress of layout in MDS projection: 1.72
- Stress of layout in t-SNE projection: 0.74

The projection-related metric outcomes for "LeagueNetwork" in MDS and t-SNE projections are the following:
Shepard score:

- Shepard Goodness score for MDS: 0.75 (See also the Shepard diagram in *Figure* 55)
- Shepard Goodness score for t-SNE: 0.71 (See also the Shepard diagram in *Figure* 56)

Trustworthiness:

- Trustworthiness for MDS: 0.79
- Trustworthiness for t-SNE: 0.81

Normalized stress:

- Normalized Stress for MDS: 0.09
- Normalized Stress for t-SNE: 0.27

Continuity:

- Continuity for MDS: 0.78
- Continuity for t-SNE: 0.82

Discussion: From the results of the graph-related metrics, it is evident that the forced-directed layout outperforms all other layouts in two out of three metrics. Specifically, it excels in "Crossing Resolution" and "Stress of Layout" while securing the second-best position in "Total Crossings". In contrast, the Layered layout yields the least favorable results, obtaining the lowest scores in both "Crossing Resolution" and "Stress of Layout" but achieving the highest score in "Total Crossings." From a visual perspective, as depicted in *Figure* 57, the visual representations of the Forced Directed layout, MDS projection, and t-SNE projection are nearly identical, which makes sense given that the "LeagueNetwork" graph contains only 16 vertices. Additionally, despite the unfavorable outcomes of the Layered layout, the vertices are well-represented and organized effectively into layers.
Now, we will analyze the projection metrics for the MDS and t-SNE models. After reviewing the metrics outputs, it's clear that there are minimal disparities between the two models, except for the Normalized Stress value, which in MDS (0.09) is three times lower than in t-SNE (0.27).

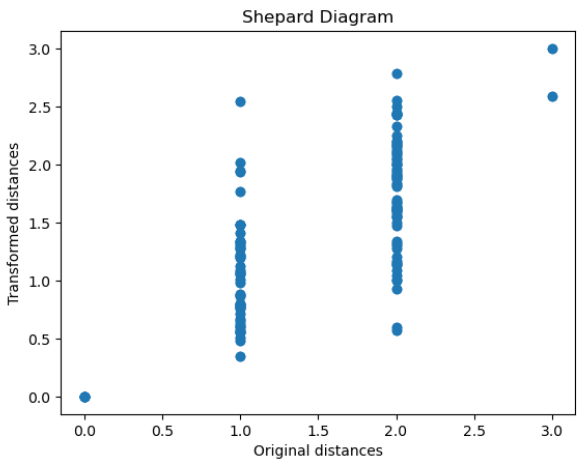


Figure 55. *MDS Shepard diagram LeagueNetwork*

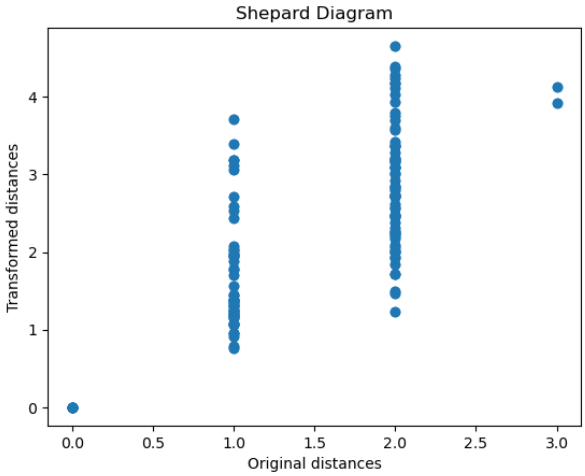


Figure 56. *t-SNE Shepard diagram LeagueNetwork*

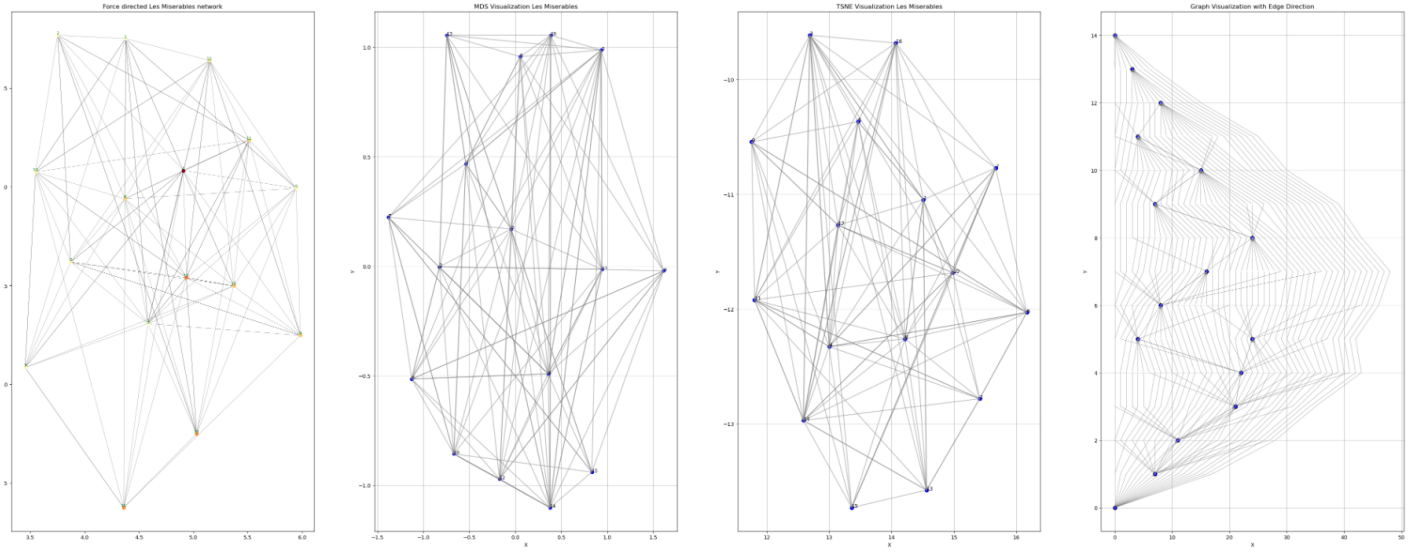


Figure 57. *First image (from left to right): Force Directed layout - second image: MDS projection - third image t-SNE projection - fourth image: Layered layout - for LeagueNetwork*

9.3.3 Jazz network

Because the "Jazz network" does not have weights on the edges, which are needed for the Floyd-Warshall algorithm, we set the weights to one to represent the edge's length. Also, due to the computational complexity of the layered layout, we did not manage to have results for the metrics, as after the insertion of dummy vertices, the graph contained 51443 vertices and 53987 edges.

Total crossings: (refer to *Figure* 60 for the visual representation of the graph)

- Force-directed layout total crossings: 216883
- MDS projection total crossings: 220699
- t-SNE projection total crossings: 186728

Crossing resolution: (refer to *Figure* 60 for the visual representation of the graph)

- Crossing resolution in Force-directed graph: 0.09
- Crossing resolution in MDS projection: 0.25
- Crossing resolution in t-SNE projection: 0.05

Stress of layout: (refer to *Figure* 60 for the visual representation of the graph)

- Stress of layout in Force-directed graph: 3271.91
- Stress of layout in Layered graph: 3032703.90
- Stress of layout in MDS projection: 4411.61
- Stress of layout in t-SNE projection: 1072.79

The projection-related metric outcomes for "Jazz network" in MDS and t-SNE projections are the following:

Shepard score:

- Shepard Goodness score for MDS: 0.82 (See also the Shepard diagram in *Figure* 58)
- Shepard Goodness score for t-SNE: 0.73 (See also the Shepard diagram in *Figure* 59)

Trustworthiness:

- Trustworthiness for MDS: 0.83
- Trustworthiness for t-SNE: 0.87

Normalized stress:

- Normalized Stress for MDS: 0.07
- Normalized Stress for t-SNE: 0.17

Continuity:

- Continuity for MDS: 0.87
- Continuity for t-SNE: 0.89

Discussion: As it can be observed from the results, t-SNE projection seems to have the best visual results, as the model achieved the least crossings (186728) and the smallest "Stress of layout" value (1072.79). Also in the projection-related metrics, t-SNE scored better in "Continuity" and "Trustworthiness", while MDS achieved better results in "Shepard score" and "Normalised stress". However, as shown in *Figure* 60, t-SNE has the best visualization outcome, as the vertices are distributed in a clearer way.

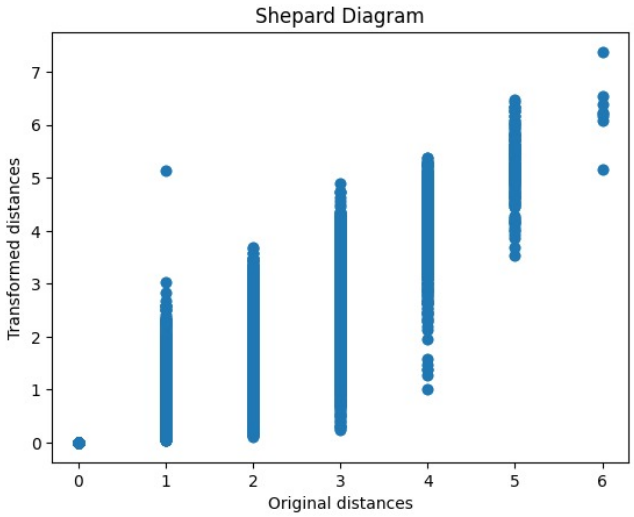


Figure 58. *MDS Shepard diagram Jazz network*

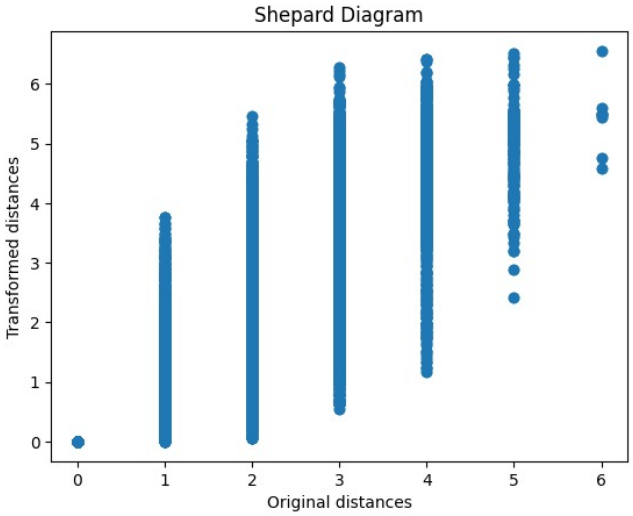


Figure 59. *t-SNE Shepard diagram Jazz network*

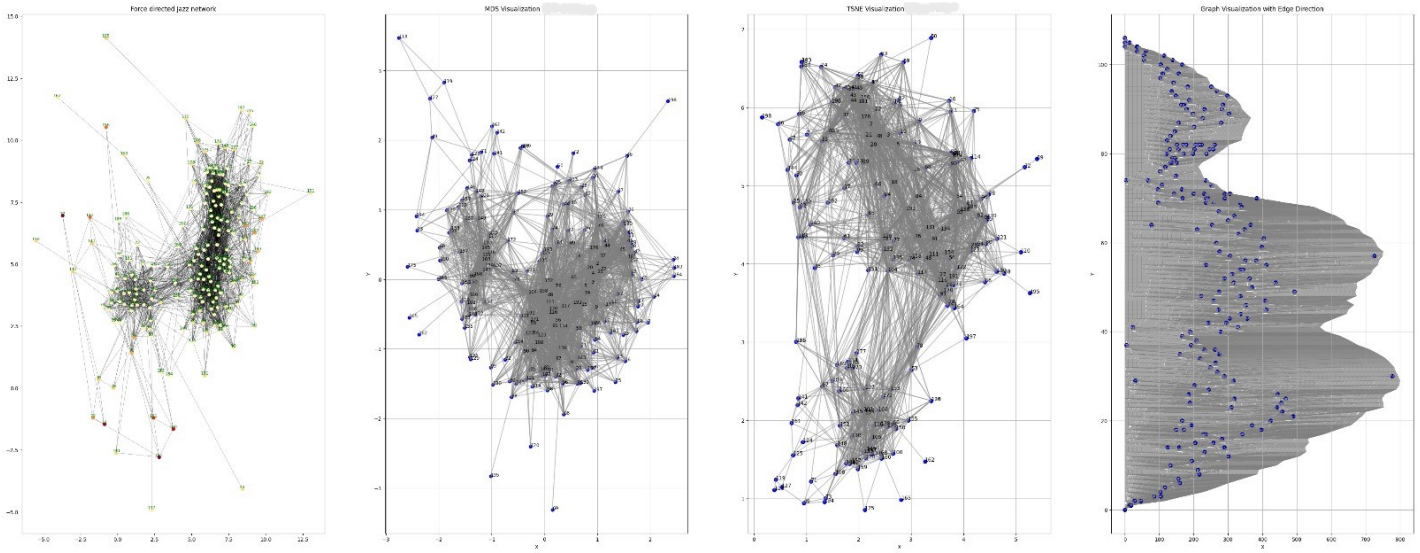


Figure 60. *First image (from left to right): Force Directed layout - second image: MDS projection - third image t-SNE projection - fourth image: Layered layout - for Jazz network*

10 Conclusion

In conclusion, in this project, we visualized multiple datasets of various sizes using various algorithms and visualization techniques. Initially, we implemented a circular structure. However, despite the absence of overlapping vertex, it was challenging to observe meaningful information about vertex relationships. To address this, we advanced our approach by segmenting the graph into circular layers. This improvement enabled us to observe better details about vertices, such as those with fewer neighbors, a task previously deemed challenging in the initial representation.

Following that, we change our focus to visualizing the graphs as trees. To accomplish this, we implemented both BFS and DFS algorithms and applied a radial layout for effective visualization. As expected, the radial layout with BFS resulted in fewer layers with vertices more evenly distributed across them, whereas the DFS layout produced more layers, allowing for a deeper visualization, including cases where only one vertex was present in a layer.

In the next step, we implemented a force layout by employing the Eades Spring-Embedded algorithm, which is inspired by natural phenomena, particularly the behavior of springs. This algorithm iteratively adjusts the positions of vertices based on attractive and repulsive forces, aiming to create a graph representation as close as possible to equilibrium. Upon applying the algorithm, we observed that the vertices formed clusters, resulting in a clearer representation of the graph.

After completing the previous step, our objective was to visualize a directed representation in a more hierar-

chical layered structure. To accomplish this, we initially needed to convert the graph into an acyclic form by applying a heuristic algorithm with guarantees. Subsequently, we assigned vertices to layers using a height minimization algorithm. To enhance the visualization of edges spanning across multiple layers and vertices, we introduced dummy vertices in each layer that the edges passed over.
Finally, we aimed to minimize edge crossings during the iterative crossing minimization phase by employing the barycenter and median algorithms to compute new positions for the vertices within the layers. Although both algorithms led to significant improvements, the barycenter method typically outperformed the median.
In the next step, we shifted our perspective again to focus on multi-layer/clustered graphs. Initially, we divided the graph into clusters/subgraphs and visualized them individually using the forced layout technique from a previous step. Following this, to enhance the clarity of edges connecting different clusters and create better visualization of connections, we applied the edge bundling algorithm to each edge linking two clusters.
Our results demonstrate that after applying edge bundling, the connections between clusters become clearer, allowing us to discern strong relations between clusters, especially in cases where there are more than two subgraphs.
Following that, we proceeded with projection models, specifically MDS and t-SNE. Projections are commonly used to simplify complex data, making it easier to understand and analyze, such as visualizing high-dimensional data in a lower-dimensional space. To employ these models, we first calculated the distance matrix by implementing the Floyd-Warshall algorithm, which identifies the shortest theoretical paths for every pair of vertices in a graph. Subsequently, we feed the matrix into both models and fine-tuned the hyperparameters. The observed results for both algorithms revealed a similar structure.
Finally, we constructed and utilized various metrics to quantitatively compare the forced layout, layered layout, and projections obtained from the previous steps. The graph-related metrics used to evaluate our graph representations included the number of edge crossings, crossing resolution (the smallest angle formed by crossing edges), and the stress of the layout. The metrics indicated that the forced-directed layout was quantitatively superior to the others. However, we also found the layered layout to provide a visually appealing representation, by organizing vertices into layers and bending the edges making it easy to follow them.
Moving on to project-related metrics, which were tested only on MDS and t-SNE projections, we employed metrics such as trustworthiness (measuring the proximity of points in the original data), continuity (evaluating the maintenance of the original data's layout), shepherd goodness score (assessing the accuracy of distance or similarity representation), and normalized stress. Upon evaluation, we observed minimal differences between the projections, except for normalized stress, which consistently remained significantly lower in MDS.

11 Links

You can find the code as well as the data and more information about them at the following GitHub repository: GitHub Link

12 References

References

[1] Eades, P., Lin, X., & Smyth, W. F. (1993). A fast and effective heuristic for the feedback arc set problem. *Information Processing Letters*, 47(6), 319–323. https://doi.org/10.1016/0020-0190(93)90079-o

[2] Sugiyama, K., Tagawa, S., & Toda, M. (1981). Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man, and Cybernetics*, 11(2), 109–125. https://doi.org/10.1109/tsmc.1981.4308636

[3] Eades, P., & Wormald, N. C. (1994). Edge crossings in drawings of bipartite graphs. *Algorithmica*, 11(4), 379–403. https://doi.org/10.1007/bf01187020

[4] https://scikit-learn.org/stable/modules/generated/sklearn.manifold.MDS.html

[5] https://scikit-learn.org/stable/modules/generated/sklearn.manifold.TSNE.html

[6] https://scikit-learn.org/stable/modules/manifold.htmlmultidimensional-scaling

[7] Eades, P. (1984). A heuristic for graph drawing. Congressus Numerantium, 42, 149{160. http://altmetrics.ceek.jp/article/ci.nii.ac.jp/naid/10026767732

[8] Van Der Maaten, L., Hinton, G. E. (2008). Visualizing Data using t-SNE. Journal of Machine Learning Research, 9(86), 2579{2605. http://isplab.tudelft.nl/sites/default/files/vandermaat

[9] Ghodsi, A. (2006). Dimensionality reduction is a short tutorial. Department of Statistics and Actuarial Science, Univ. of Waterloo, Ontario, Canada, 37(38), 2006.

[10] Eades, P., Lin, X., Smyth, W. F. (1993). A fast and effective heuristic for the feedback arc set problem. Information Processing Letters, 47(6), 319{323. https://doi.org/10.1016/0020-0190(93)90079-o.

[11] Sugiyama, K., Tagawa, S., Toda, M. (1981). Methods for visual understanding of hierarchical system structures. IEEE Transactions on Systems, Man, and Cybernetics, 11(2), 109{125. https://doi.org/10.1109/tsmc.1981.4308636.

[12] Sugiyama, K., Tagawa, S., Toda, M. (1981). Methods for visual understanding of hierarchical system structures. IEEE Transactions on Systems, Man, and Cybernetics, 11(2), 109{125. https://doi.org/10.1109/tsmc.1981.4308636.