Toward Graph Layout of Large Data Visualization: Algorithms, Evaluations and Application

by

Michael Ferron

A thesis submitted in conformity with the requirements
for the degree of Master of Science
Faculty of Graduate Studies (Computer Science)
University of Ontario Institute of Technology

Supervisor(s): Dr. K. Pu and Dr. J. Szlichta

Abstract

Toward Graph Layout of Large Data Visualization: Algorithms, Evaluations and Application

Michael Ferron

Master of Science

Faculty of Graduate Studies

University of Ontario Institute of Technology

2016

Generating layouts for millions of points on a spatially-restricted platform is a difficult task with a unique set of constraints. These layouts are traditionally generated on a server out of sight of the user. User-oriented applications would benefit from a real-time view of layout generation, which can assist user decision making and improve user experience by introducing interactivity. The literature of constraint resolution and mobile visualization is briefly surveyed to achieve an understanding of the state of the art for this problem, and motivate a solution with scenario-based examples. We formally identify the major constraints associated with this specialized layout generation problem and the special interplay between them. A pipeline-based layout generation method is defined algorithmically, coupled with the implementation of the algorithm(s). The quality of the result is analyzed on a constraint-dependent basis. Applications and future system improvements and extensions are discussed.

Acknowledgements

I would like to thank UOIT for providing a wonderful environment to conduct research and to pursue academic study.

I would like to thank my supervisors, Ken Q. Pu and Jaroslaw (Jarek) Szlichta for their support and guidance through the course of this research. I have learned greatly through their counsel; they are both incredible researchers and role-models. I would like to thank Mehdi Kargar for his assistance in the integration of his system with our work. I would like to thank my thesis committee for their informative feedback and helpful comments.

I would like to thank my colleagues and friends within UOIT for their company and for interesting discussion on research topics. I would like to thank my friends outside of UOIT for being available at odd hours of the night when a break was needed and providing support.

Lastly, I would like to thank my family for their continued support through the years, and for tolerating the impact of my abnormal schedule on their lives. I am truly grateful for their love and support.

Contents

Chapter 1

Introduction

Modern data systems often contain data on the order of big data, consisting of millions of tuples distributed across multiple tables, each with a variety of different attributes. These tuples can express relationships within the dataset through identifying indexing attributes. Data visualizations are produced to present to users and analysts these trends, patterns and relationships via aggregation. Many applications can make use of layouts that map the aggregate source points to the canvas, instead of presenting an aggregated view. Highly granular views of source datasets are primarily presented on screens of very high pixel count, in order to be unencumbered by the dimensions of the device. Due to the high volume of data, these views are not easily mappable to a two- or three-dimensional visualization medium.

With smartphones and tablets being common tools both in the user's home and the enterprise setting, it may not be possible for application developers to escape the constraint of a two- or three-dimensional visualization. Analysts often work through their mobile devices alongside their computers, and with the integration of the smartphone into both the workplace and home life, users may contend with applications that present a high volume of data as well. Efficient production of layouts suitable to these devices is a problem that will need to be resolved in order to expand the ability of data scientists to present nuanced views of the data to others. As mobile devices are restricted by multiple constraints, the problem of laying out a high granularity view of the data must take these constraints into consideration.

Primarily, layouts are generated in the backend of a system on a server, as a black box technology unavailable to the user. While this approach has its benefits regarding performance and device responsiveness, it does not provide to the user the story of the dataset's path to its destination. Datasets vary wildly in volume and in feature sets, thus automating the process of layout generation can be a difficult problem to address. Niche case layouts can be created on a dataset-by-dataset basis, but such an approach is inefficient as a tool for enterprise decision-making, where multiple views of multiple datasets may be required to drive business logic and decision-making. Automatic layout generation for big data-level datasets on mobile platforms would enable a variety of tools to benefit data scientists, analysts working directly and often with the data, and application developers, who may use such a technique to create a popular recommender system.

1.1 Contributions

The contributions of this thesis can be summarized as follows:

- **Definition of the problem of layout generation for datasets containing millions of tuples on a mobile or small screen device.** In working with millions of tuples, as mobile devices and tablets become more prevalent in the workforce as an analytical tool, the challenge of fitting each tuple to the restricted screen space becomes increasingly relevant. The problem is formulated ahead, with supporting examples and considerations to show its relevance. The IMDb dataset is used as an illustrative dataset, containing approximately 3.3 million movie entries.

- **Establishment of a meta-layout generation method using hyperparame-**

ter tuning. A pipeline approach to data visualization allows the system to be augmented to implement a variety of simulation models. The system shows the novelty of a pipeline approach being used to generate a layout by breaking the problem of ideal layout generation into multiple phases, each phase representative of a particular constraint to resolve, and a task that seeks to solve this constraint, and iterating on each constraint resolution method multiple times with hyperparameter tuning.

- **Ad-hoc layout generation via one or more user-defined indexing attributes.** Graph layouts are generated in the implemented system ad-hoc by a user-defined indexing attribute. Layouts generated for these attributes are saved, enabling large volumes of relational data to be visualized by a technique determined after layout generation.

- **Implementation of the meta-layout generation via hyperparameter tuning enabling database visualization.** The meta-layout generation process has been implemeneted in a system using ClojureScript. This system encompasses the primary contribution of this work, demonstrating the use of time-tested algorithms in a novel approach to enable database visualization.

- **Optimizations for data visualization.** The reduction of nodes to process allows the system to generate layouts quickly. Compared to many visualization techniques and systems taking hours to place millions of nodes, the system's pipeline approach can model as many tuples as these systems in a much smaller time frame. This approach provides in a timely fashion a layout that can be used to map the aforementioned points to a visualization space without incurring much delay.

- **A scalable data prepartion technique.** As part of the system, scores are calculated between aggregated nodes for a given indexing attribute on millions of tuples. The score calculation method in the system scales effectively with the number of aggregate nodes, providing a lightweight method of computing relationship data with which layouts can be generated.

- **Applications of the pipeline method for augmenting query answers using generated layouts both visually and in selection.** Using a system that provides query answers in the form of answer trees as a base, the Jaccard Similarity measure is investigated as a ranking measure for query answer selection, showing different sets of selected query answers. Additionally, a method for enriching the answers returned by the system using layouts generated by the pipeline system is described.

- **Experimental evaluation of constraint resolution.** Metrics for the evaluation of the resolution of constraints via the annealing, compacting and resizing algorithms are defined. These metrics are used to determine the effectiveness of the defined algorithms in the pipeline.

1.2 Motivating Application

Consider the IMDb dataset shown in Figure 1.1. The IMDb dataset contains millions of *actors* and millions of *movies* distributed on multiple tables. These entities are described by a number of different *attributes* corresponding to metadata. This dataset is rich in relationships, as actors can be related to each other and to movies via particular pieces of metadata, and the same holds true for movies. Further, these entities may hold multiple values for a given attribute. Visually representing such a dataset is typically done by aggregating these entities according to some indexing attribute. The aggregation process is time-consuming and relegated to a server backend. Due to the high volume of data and the multidimensional nature of the data, visualizations working with this dataset scale poorly, as there are very many calculations that must be performed each iteration. In order to efficiently provide a view of the data, an approach must be taken to lay out the

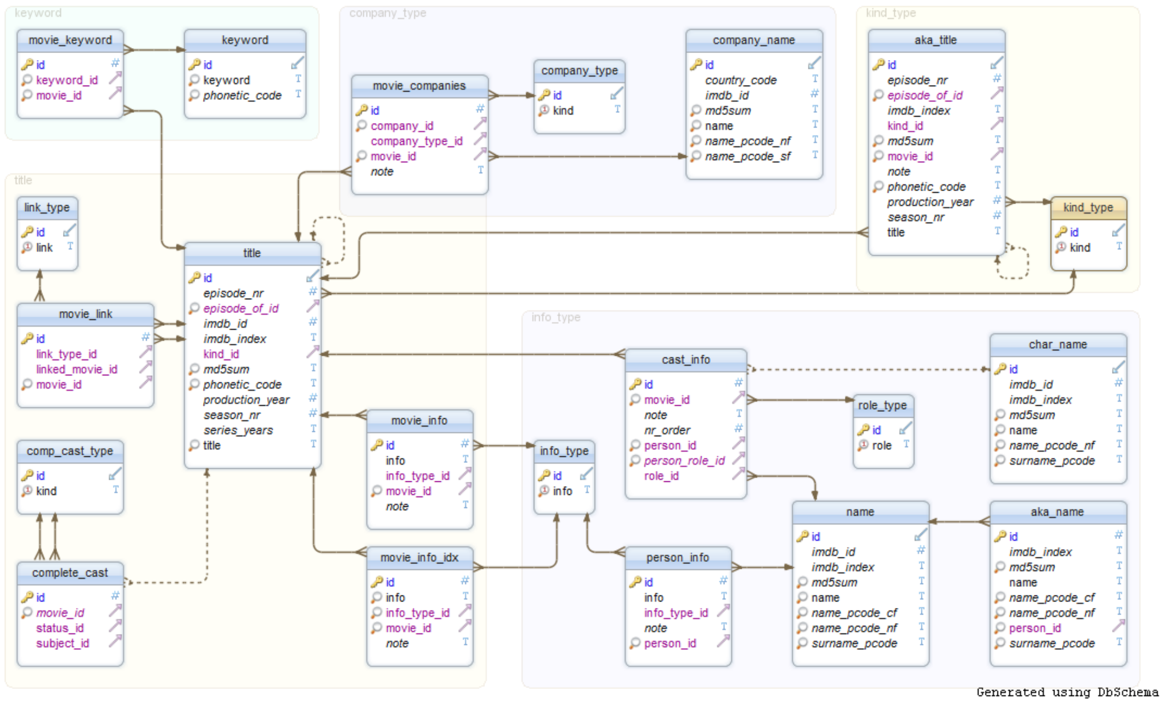


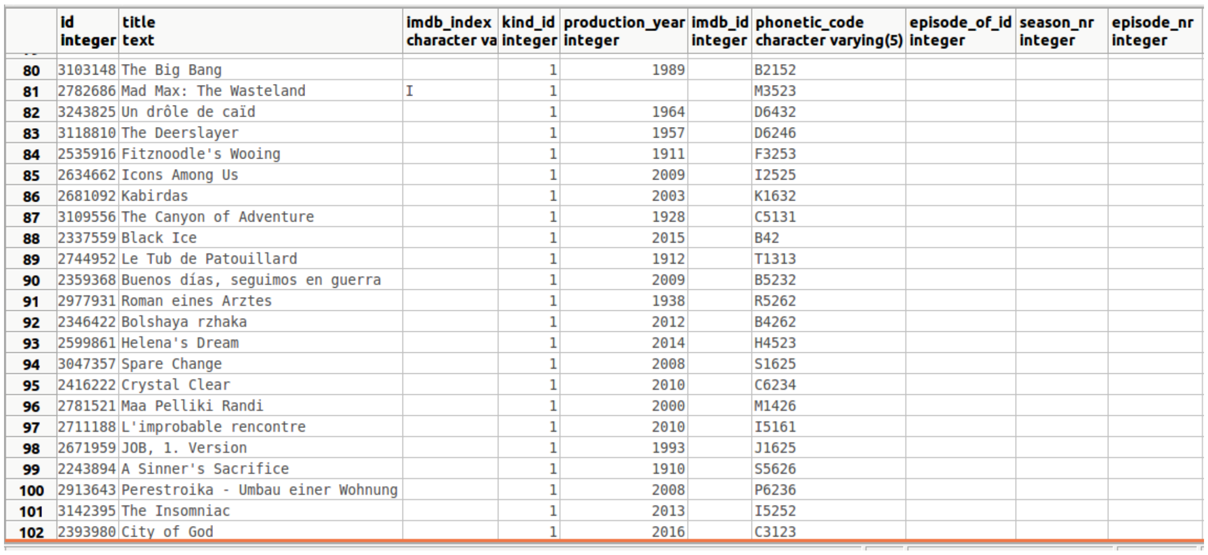
Figure 1.1: The IMDb Schema.

tuples within the database in a lightweight fashion. If an approach for quickly generating a layout of the millions of movies in the dataset can be determined, then that layout generation can be used for the presentation of data for many different tasks, including scenarios that involve cleaning the dataset or in presenting answers to a user via the layout.

In data analysis, showing the relationships between data is the motivation behind a wide number of problems and applications. Consider the task of a film critic writing an article about a prominent director, to be viewed online. The film critic may want to organize all movies based on their language(s), and then show the concentration of movies produced by the director based on their language. This visual representation could be used to show the languages that a given director favours, as well as showing the different languages that are similar to each other. Such a layout can be used to visually and intuitively portray the information discussed in the article.

1.3 Background

In a relational database, data are stored within tables in the form of *tuples.* A tuple is an n-dimensional entity with n fields. Each field has a value, either a predeclared data type or the *null* type for empty fields. These values correspond to different *attributes.* An attribute is a characteristic that can be used to describe, organize and aggregate tuples. Consider the incorporated figure below:

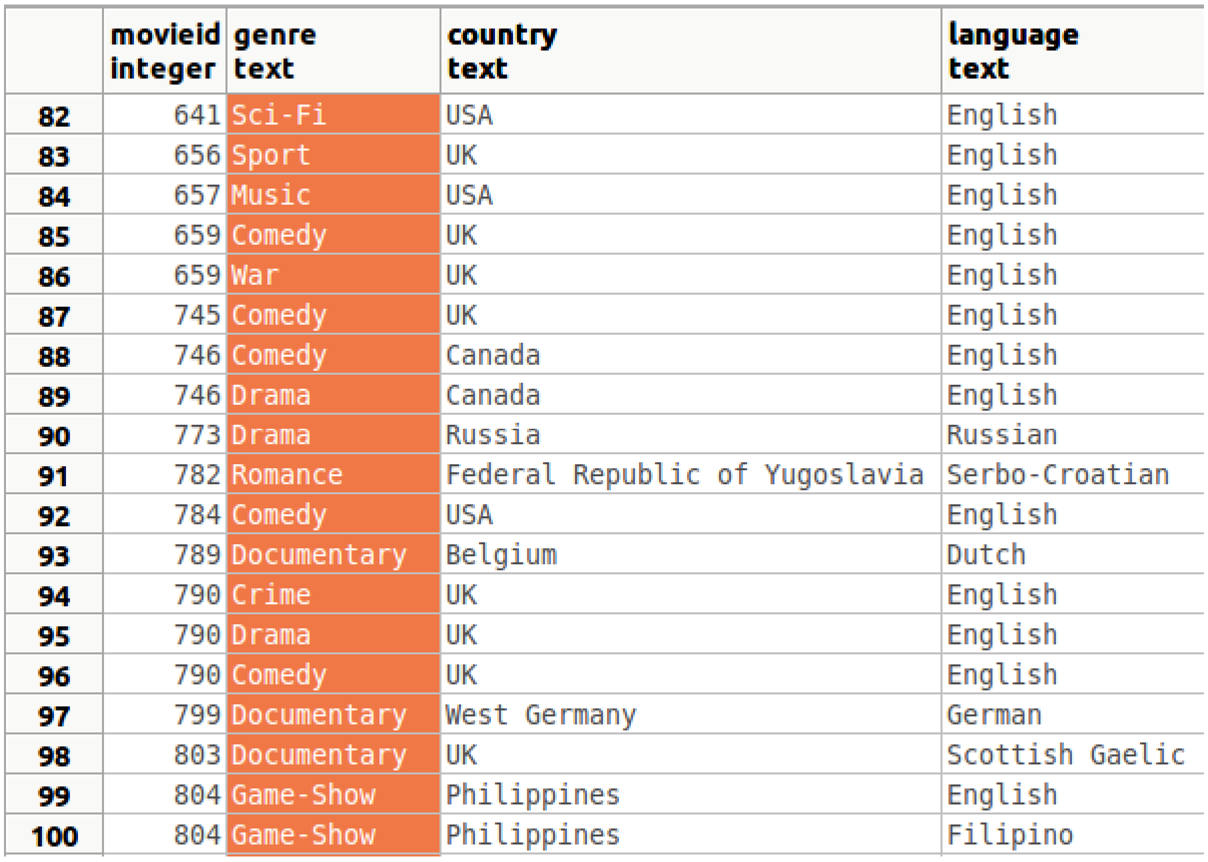


	id integer	title text	imdb_index character va	kind_id integer	production_year integer	imdb_id integer	phonetic_code character varying(5)	episode_of_id integer	season_nr integer	episode_nr integer
80	3103148	The Big Bang		1	1989		B2152			
81	2782686	Mad Max: The Wasteland	I	1			M3523			
82	3243825	Un drôle de caïd		1	1964		D6432			
83	3118810	The Deerslayer		1	1957		D6246			
84	2535916	Fitznoodle's Wooing		1	1911		F3253			
85	2634662	Icons Among Us		1	2009		I2525			
86	2681092	Kabirdas		1	2003		K1632			
87	3109556	The Canyon of Adventure		1	1928		C5131			
88	2337559	Black Ice		1	2015		B42			
89	2744952	Le Tub de Patouillard		1	1912		T1313			
90	2359368	Buenos días, seguimos en guerra		1	2009		B5232			
91	2977931	Roman eines Arztes		1	1938		R5262			
92	2346422	Bolshaya rzhaka		1	2012		B4262			
93	2599861	Helena's Dream		1	2014		H4523			
94	3047357	Spare Change		1	2008		S1625			
95	2416222	Crystal Clear		1	2010		C6234			
96	2781521	Maa Pelliki Randi		1	2000		M1426			
97	2711188	L'improbable rencontre		1	2010		I5161			
98	2671959	JOB, 1. Version		1	1993		J1625			
99	2243894	A Sinner's Sacrifice		1	1910		S5626			
100	2913643	Perestroika - Umbau einer Wohnung		1	2008		P6236			
101	3142395	The Insomniac		1	2013		I5252			
102	2393980	City of God		1	2016		C3123			

Figure 1.2: A relation, with several tuples and attributes.

In Figure 1.2, consider the very first row. This row is a *tuple* with the values $(3103148, TheBigBang, null, 1, 1989, null, B2152, null, null, null)$. Each of these values corresponds to the *attribute* named at the top of the column (i.e. 'The Big Bang' is the tuple's value for the *title* attribute. As shown, not every entry of a tuple must have a non-null value or a unique value. These values pertain to a single attribute, the constraints of which are defined on an attribute-by-attribute basis. To explore these concepts further, let us consider the below image.

In Figure 1.3, a sampling of IMDb movie data is provided. Each row is a tuple, with attribute values for each tuple shown. These attributes can have either unique values, or indistinct values. Examining the four attributes shown, all columns contain indistinct



	movieid integer	genre text	country text	language text
82	641	Sci-Fi	USA	English
83	656	Sport	UK	English
84	657	Music	USA	English
85	659	Comedy	UK	English
86	659	War	UK	English
87	745	Comedy	UK	English
88	746	Comedy	Canada	English
89	746	Drama	Canada	English
90	773	Drama	Russia	Russian
91	782	Romance	Federal Republic of Yugoslavia	Serbo-Croatian
92	784	Comedy	USA	English
93	789	Documentary	Belgium	Dutch
94	790	Crime	UK	English
95	790	Drama	UK	English
96	790	Comedy	UK	English
97	799	Documentary	West Germany	German
98	803	Documentary	UK	Scottish Gaelic
99	804	Game-Show	Philippines	English
100	804	Game-Show	Philippines	Filipino

Figure 1.3: An executed query showing some tuples in the relation.

values for each movie. Attributes that contain completely unique values are candidates for declaration as a *primary key.* A *primary key* is an attribute for which the field of a tuple most be both not null, as well as unique. Consider the column to the far left in the above image. Each entry is not null, as well as different from every other value in the column. Any attribute that functions as a primary key provides an avenue by which tuples with similar values are differentiated from each other.

The descriptive nature of attributes allows tuples to have operations performed on them based on their attribute values. To perform these operations, the database is issued a *query,* the results of which are a set of attribute values. A query is a descriptive statement that fetches a set of attribute values from a relation. These values are presented as a *query answer.* Queries can be used to modify tuples, as well as *aggregate* them. Aggregation is the process of combining together all tuples that have a given value, presenting these combinations as one unique tuple per row for every row. Aggregation of tuples provides a condensed view of the data, and can be used to show relationships within tuples on a higher level. Once a query has returned a set of attribute values across a table's tuples, data analysis can be performed by aggregating the tuples on one or more of these attributes.

To demonstrate relationships within tuples, *similarity* data between tuples is used. In order to determine similarities between tuples, the tuples must be aggregated according to an attribute of interest. This attribute is designated as the *indexing attribute*, as the data are classified according to keywords (the keywords being the attribute values of each tuple). An *indexing attribute* is an attribute by which tuples in the result set are sorted. Referring to Figure 1.3, *genre* is an example of a possible indexing attribute, as it organizes tuples based on the tuples' value[s] for genre. An important note is that an indexing attribute can contain multiple values for a given tuple, classifying that tuple in all groups where it shares the value of the group (i.e. movie ID 659 can be indexed to both Comedy and War).

Relationships between aggregations can be stored in one of a number of *data structures*. In visualization, relationships are demonstrated through a number of different *graphs*. A graph consists of a set of *vertices* connected by *edges*, as shown in Figure 1.4. A *vertex* is a point in space with positional values for each dimension. A two-dimensional space like the one shown in Figure 1.4 would produce a vertex with two positional values (x, y). Given two vertices, an *edge* is a line connecting two vertices.

To demonstrating similarity or distance scores for a set of tuples grouped by genre, for example, a *tree* of vertices and edges can be created, where the vertices are representative of the different possible values of the indexing attribute, and the edges are weighted with a *score* calculated to represent the similarity between two connected vertices. In this work, a *maximal spanning tree* is used to represent the calculated relationships between

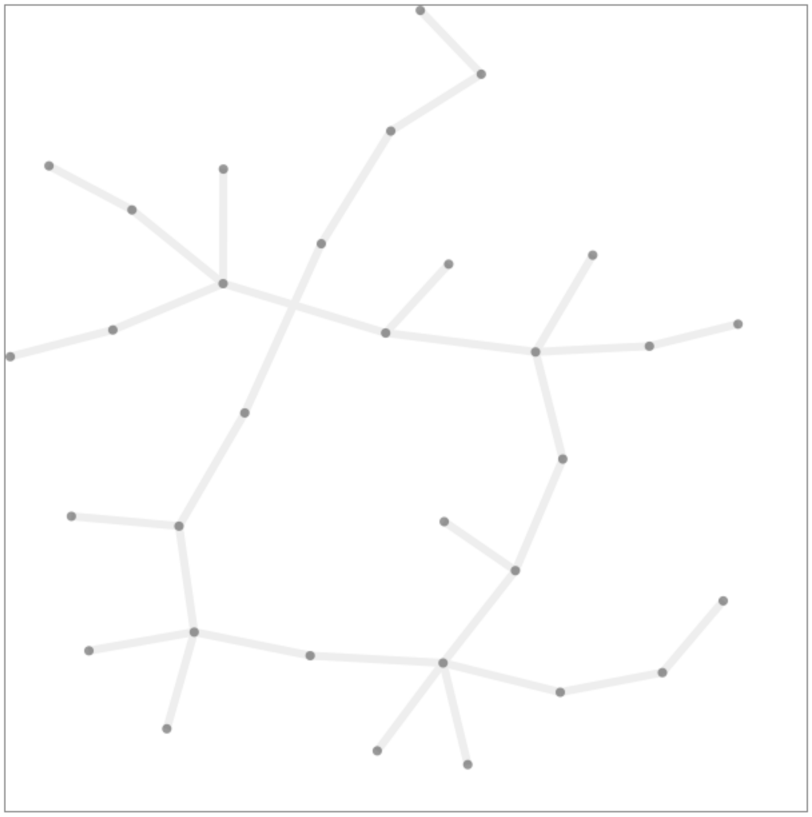


Figure 1.4: A tree, a type of graph.

attribute values. A maximal spanning tree consists of a set of vertices, representing every different possible value a tuple can have for a selected indexing attribute, and a set of edges. For a tree to be maximally-spanning, every value must be represented in the set of vertices, and each vertex must be connected to at least one other vertex. The set of a vertex's edges must contain an edge that connects it to the vertex it has the highest similarity score with.

In order to model similarities, scores must be computed that show the strength of connection between vertices, or the degree of similarity between two values in a given attribute. Scores such as the *Jaccard similarity* return a value bound between zero and one, which lends itself easily to use as a percentage. Given two sets $A = Tom, Jane, Marty, B = Tom, Lewis, Carrol$, the *Jaccard similarity* of these two sets is calculated as $\frac{A \cap B}{A \cup B}$. The result of this calculation with the given sets is $1/5$, producing a similarity of 0.2.

1.4 Related Work

With respect to visualizing high volumes of data, many different approaches have been taken in different contexts. When working with large graphs as datasets, graph decomposition and reduction may be necessary in order to present the graph to the user. Kruskal provides a method for creating a minimized structure, within which the similarities of a graph can be embedded [18]. In work presented by Ahmed et al., a method for graph decomposition is presented. This decomposition is achieved by partitioning the graph to minimize the number of neighbouring vertices [3]. Calculating similarity values on the input data set may be required before visualizing the data. A possible tree structure for use in live updating visualizations is presented by Beygelzimer et al., where the authors describe a tree which is space-constant, reducing the edges contained while maintaining the performance properties of a full navigating net [4]. In creating graphs, Tang et al. (2015) presents a method for embedding information networks in low-dimensional spaces using edge sampling and the satisfaction of an objective function that preserves network structures [26]. Our work computes input scores based on the technique described by Kruskal, focusing on the edges with the highest weights to create a maximal spanning tree instead of a minimal spanning tree [18]. This set of nodes and edges forms our input graph for layout generation.

To prepare data for visualization, the dataset may need to undergo the process of dimensionality reduction - selecting a representative subset of the set of a dataset's attributes - to fit the 2D or 3D visualization medium. Yan et al. present a graph embedding framework, defining both the objectives and framework with experimental evaluation [28]. Dimensionality reduction is further explored by Hinton and Roweis, where the authors define a method for performing dimensionality reduction to place objects in a low-dimensional space while respecting their similarities [13]. This method uses a natural cost function to create a gradient that adjusts embedded objects in the lower-dimensional space. An application of this work is present in work by Cook et al., where the stochastic neighbour embedding method is used to visualize pairwise similarities by using conditional probabilities with aspect maps [7]. Our work performs dimensional reduction by relying on a user-defined indexing attribute to determine similarity data between aggregate nodes based on the indexing attribute.

After dimensionality reduction has been performed, visualization methods designate positions of data points in two or three dimensions. An extension of SNE coupled with visualization application is presented by Van der Maaten and Hinton, describing visualizing high-dimensional data by extending the SNE method [27]. The work presented by Tang et al. (2016) tackles the visualization of high-dimensional, large-scale data by creating an approximated nearest-neighbour graph, presenting the graph in the lower-dimensional space [25]. Another visualization method is shown in work presented by Novotny [20]. Novotny describes the visual abstraction method of visualizing large data, providing discussion on the challenges faced when working with large data. With respect to data presentation methods, Singh et al. present an alternate method of showing query results with many tuples by selecting representative tuples as a representation of the query result [23]. Kargar et al. show query results by identifying and ranking answer trees based on query keywords [15]. Our pipeline layout generation uses the algorithm described by Fruchterman and Reingold, along with hyperparameter tuning, to determine positions of nodes [10]. These positions can then be used to create a visualization.

Network and graph layout algorithms have been presented in other works. Applications can use frameworks such as the one described by Fruchterman and Reingold, wherein a framework for graph drawing using a force-based simulation is described [10], an implementation of which can be found in D3.js [1], a collection of data-driven documet manipulation algorithms. Enright and Ouzounis present an automatic graph layout method

BioLayout to visualize protein sequence similarities [8]. BioLayout uses a variation of force-directed layout generation, annealing layout based on temperature reduction to slow node movement to a final position. Jacomy et al. also utilize a modification of the force-directed layout described by Fruchterman and Reingold to continuously generate layouts of networks [14]. ForceAtlas2 uses a "scaling parameter" in place of attraction and repulsion to generate layouts in a single pass. Sander creates layouts of graphs by partitioning nodes and determining their position by one of two methods, followed by determining a degree of edge bending [22]. Ko and Yen combine a modified stress majorization node placement algorithm with torque equilibrium to generate graph layouts [16]. These graph layout generation methods vary from our work as we take a meta-layout generation approach with an existing, proven layout algorithm as opposed to a single-pass layout generation approach. The pipeline graph layout method drives a force-directed layout in a multi-pass approach, utilizing hyperparameter tuning to generate layouts.

An application of network layout is the visualization of social networks. Sköld describes methods by which social networks can be analyzed and layouts for social networks can be generated, containing an implementation of a social network visualization tool [24]. Mccarty et al. compare social network visualizations generated algorithmically from alter-pairs with personal network visualizations determined manually by users [19]. Christakis and Fowler demonstrate the visualization of social networks in an epidemiological context, citing algorithms that are used to draw undirected large graphs and showing visualizations of social networks [6]. Heer and Boyd present a tool for visualizing social networks, the layout being generated using a spring embedding algorithm [12]. Boccaletti et al. investigate the characteristics of multilayer networks and the use of adjacency matrices to determine multilayer network structure, in applications such as social networks [5]. Social network visualization is restricted by attribute, where our approach to generating graph layouts can be used to organize data in an ad-hoc fashion according to a user-defined indexing attribute. The notion of a user-selected indexing attribute allows for attributes to be selected that create both sparse and dense graphs.

1.5 Problem Definition

Given a collection of relational tables $\{R_1, R_2, ..., R_n\}$ and a set of millions of entities, defined to be tuples T in a query $Q : (R_1 \bowtie R_2 \bowtie ... \bowtie R_n)$ having a key in the set of attributes $I \subseteq A\{A(R_i) : i = 1, 2, ..., k\}$, consider the tuples contained within the projection $\Pi_I(T)$. This projection restricts the fields of T present solely to the indexing attribute I.

Problem 1. **Relationship Determination:** $\Pi_I(T)$ contains a collection of tuples grouped according to I. Calculate Sim, the relationships between two nodes N_a, N_b in $\Pi_I(T)$, for all pairs of nodes, where a node N_a is the aggregate of all tuples in $\Pi_I(T)$ that contain the value a.

Problem 2. Given Sim, produce a data structure MST that contains all nodes N in $Pi_I(T)$, coupled with a set of edges E. E must contain the edge connecting a node N_i to its most similar neighbour node, for all N.

Definition 1. **Window (D):** The window is a two dimensional bounding box with width W and height H. The area of the window is $A : W \times H$. A approximates the area of a computing device screen. With respect to mobile and small screen devices, the dimensions of A are restricted greatly. The number of pixels within A is approximately the number of tuples in $\Pi_I(T)$.

Problem 2. **Node-Neighbour Positioning:** Given the projection of tuples based on the indexing attribute, map $\Pi_I(T)$ to **D**. For each node N in MST, position N such that the distance to each node N is connected to is minimized. In a spatially-restricted window, reduce the length of the edges connecting two neighbouring nodes as much as possible to conserve space.

Problem 3. **Proportional Node Sizing:** Given the number of tuples in a node in

$\Pi_I(Q)$, scale the nodes in D such that each node placed within D approximates its size in $\Pi_I(Q)$ relative to the cardinality of tuples in Q. As space on a mobile screen is restricted, each node must fit within the window while also approximating the percentage of tuples that contain the value of the node.

Definition 2. **Layout (L):** A layout is the positions of all nodes and edges contained in MST within D. Each layout consists exclusively of the positioning of the tree's components.

Problem 4. **Whitespace Reduction:** Given a layout L, nodes can have a variable amount of empty space between them. Eliminate the intermittent whitespace as much as is possible in order to most efficiently use the pixels available on a mobile device.

Chapter 2

Methodology

In this chapter we will briefly review the problem being addressed by the automatic pipeline method layout generation algorithm. The issue of data preparation encompassed by the pipeline algorithm will also be touched on. The algorithms for each step of the pipeline will then be defined, and the algorithms implemented together in a layout generation system will be presented as a visual walkthrough of the stages of the pipeline algorithm. An implementation of the discussed algorithms will be described in detail, topics of which include the relevant program environment and constructs employed to enable the system to perform tasks outlined in the algorithm discussion.

2.1 Algorithm

A set of problems have previously been defined in Chapter 1 that describe the problems associated with graph layouts of large data visualizations, both in general and on screen spatially-restricted devices such as mobile phones and tablets. What follows is a presentation of a pipeline algorithm being introduced to solve the previously-outlined constraints. The pipeline-style solution will be described, and each stage of the pipeline will be introduced in detail.

2.1.1 Pipeline Solution

As previously discussed, there are two primary problems to be addressed. The first issue is the task of generating a layout on a high volume of data, wherein the layout must adhere to environment-specific constraints. In presenting a high enough volume of tuples to be described as big data, the speed of layout generation decreases greatly. To preserve suitability for user interaction, the layout generation process must be as unhindered by increase in volume of points as possible. Even moderate decreases in responsiveness can greatly impact the user experience.

The second problem to be addressed centers around the issue of solving multiple constraints. A common method of resolving multiple constraints involves attempting to solve all constraints simultaneously, performing actions that are more complex in nature and considering all constraints with each action. In the case that some constraints are in conflict, cost-benefit and requirement analysis is performed in order to identify a constraint's importance, determining an acceptable threshold of lower quality for a given constraint. To resolve the defined problem these constraints possibly in conflict must be resolved to generate a layout, a pipeline methodology is proposed. The pipeline method of constraint resolution follows a logical ordering.

Definition 3. **Pipeline Constraint Resolution:** Given a set of constraints $C : \{C_a, C_b, ..., C_i\}$ with varying objectives for resolution, organize the constraints such that they follow a particular order O. Resolve constraints in the order they appear in O.

Pipeline constraint resolution allows for a system to deconstruct a problem that may be too complex to resolve automatically given a set of constraints, enabling components of a system to solve the problem. By decoupling conflicting constraints, a system can resolve a given constraint using simpler approaches. This approach enables automatic constraint resolution.

The pipeline approach to constraint resolution allows the problem of layout generation to be rendered automatic. A system that can automatically generate layouts provides opportunity for a generalized approach to solve for combinations of constraints for different environments. The components of the pipeline method for automatic layout generation will be outlined in this chapter.

2.1.2 Score Calculation

In order to present the relational dataset in 2D space, network data is provided to the system. In the absence of a prepared network relationship, one can be induced. This network data represents the similarity between elements of $\Pi_I \sigma_k(Q)$. In our system, similarity between nodes of I is represented by the Jaccard set containment score. The Jaccard set containment score between two nodes $(j, j)'$ in I is represented as:

$$sim(j, j') = \frac{I(j) \cap I(j')}{max(I(j), I(j'))} \tag{2.1}$$

where $I(j) = \Pi_I \sigma_j(Q)$ is the collection of keys in j, $I(j) \cap I(j')$ is the set of shared keys between j and j', and $I(j) \cup I(j')$ is the union of keys in j and j'. The result of (2.1) for any (j, j') is bound between $\big[0, 1\big]$, where 1 is complete similarity and 0 is complete dissimilarity. These similarity measures can be approximated to improve performance. In an end-to-end system, the Jaccard set containment score is approximated for efficiency.

2.1.3 Annealing

Nodes in $\Pi_I \sigma_k(Q)$ are placed. Once nodes are placed, their positions are driven through a force-directed simulation, governed by a modified equation in [10]:

$$p_j(i+1) = p_j(i) + \sum F \tag{2.2}$$

$$F = k \times p_j - g + \alpha \sum_{i=1}^{n} C \tag{2.3}$$

where p_j is the node j's position, g represents the attractive force of gravity, C is the repulsive charge force, α represents the system cooling factor, and $k \times p_j$ is the displaced spring force. Each node will have a repulsive forced applied by its geometric neighbours.

```
Input: $x$ Input Node Positions $P(N)$ and Velocities $V(N)$
Output: $x$ Updated Node Positions $P'(N)$
Energy ← IncreaseEnergy(Energy);
for $i \leftarrow 0$ to $x$ do
    $V_i = V_i + F_i$;
    $P'_i = P'_i + V_i + k_i$;
end
Input: Input Node Velocities $V(N)$
Output: Updated Node Velocities $V'(N)$
On simulation time tick: for $i \leftarrow 0$ to $x$ do
    $F(i) = \sum_0^m (k) - C + g \; V'_I = V_i - F_i$;
end
```

The force-directed simulation has its energy increased by a fixed amount. This energy influx is constant on each iteration. As energy is introduced into the simulation, the velocity of each node increases. The nodes' positions shift according to the change in their velocity. As nodes are connected, their neighbours will exert a spring force proportional to the strength of the spring and the nodes' velocities. The simulation energy cools after receiving an increase in energy. Nodes' momentums decrease, and they settle toward a position at the end of each annealing phase. After the initial injection of kinetic energy, each node slowly decelerates, the force of dragging connected nodes reducing the velocity of nodes in an inelastic environment.

2.1.4 Resizing

During the resizing phase, nodes have their size determined according to a nonlinear transformation. Resizing is governed by the following equations:

$$r = (a + s(b - a)) \times O \tag{2.4}$$

$$s = \frac{1}{e^{-c(x-0.5)} + 1} \tag{2.5}$$

The node's raw size determines the *percentile* x it belongs to. The sigmoid function 2.5 is sampled at x, scaled by c. Equation 2.4 determines the pixel size of the node, having a minimum size of a, scaled at a rate of O per resizing. Genres falling in different percentiles can be seen in scaled in figure 1.4, where more populated genres scale to larger radii.

During the resizing phase, collision detection between nodes is enabled for the duration of the pipeline. As nodes initially needed to be able to freely shift toward locations to remove edge crossings, the system does not enable collisions during said phase. Once the initial layout is determined, collision detection is enabled to prevent major restructuring of the initial layout through the compacting and resizing phases. Collision detection is performed as follows:

```
Input: Input Node Positions $P(N)$
Output: Updated Node Positions $P'(N)$
for $i \leftarrow 0$ to $x - 1$ do
    for $j \leftarrow i$ to $x$ do
        if $P_i$ overlaps with $P_j$ then
            $P_i$ and $P_j$ are shifted away from the collision point;
        else
            Only charge, gravity and spring forces act on $P_i$ and $P_j$;
        end
    end
end
```

2.1.5 Compacting

Consider equation 2.4. The cooling parameter α reduces in this phase. The magnitude of $\sum_{i=1}^{n} C$ decreases, compacting the nodes together. Figure 1.4 shows a layout compacted after the resizing phase. As demonstrated, the nodes have been drawn together by the reduction in charge, and similar genres are placed with their neighbours.

```
Input: Node Positions $P(N)$, Charge C
Output: Updated Node Positions $P'(N)$, Charge C'
On simulation time tick:;
Reduce C by a fixed interval;
for $i \leftarrow 0$ to $x$ do
    $F(N) = \sum_0^x (k) - C + g$;
    $P'(N)$ = P(N) + F(N);
end
```

2.2 Visual Walkthrough

The pipeline system is demonstrated using captures of each stage, accompanied by a brief description describing the changes in the simulation at each step.

The system constructs a maximum spanning tree from the provided similarity data, using Kruskal's algorithm [18]. Edge weights are determined based on the paired node similarity values obtained from Equation 2.1. Edges with the highest weight are selected, and if one of the edge's nodes is not in the spanning tree, the edge is added to the spanning tree. This set of edges forms the input graph used by the system in layout generation.

2.2.1 Layout Resolution

The tree is introduced to the canvas with random node placements. This introduces edge crossings between nodes and, due to the charge and gravity parameters, a high volume of inter-node whitespace. At this stage, the nodes are shifted due to the high amount of initial energy present in the simulation, their velocities degrade rapidly as charge, spring

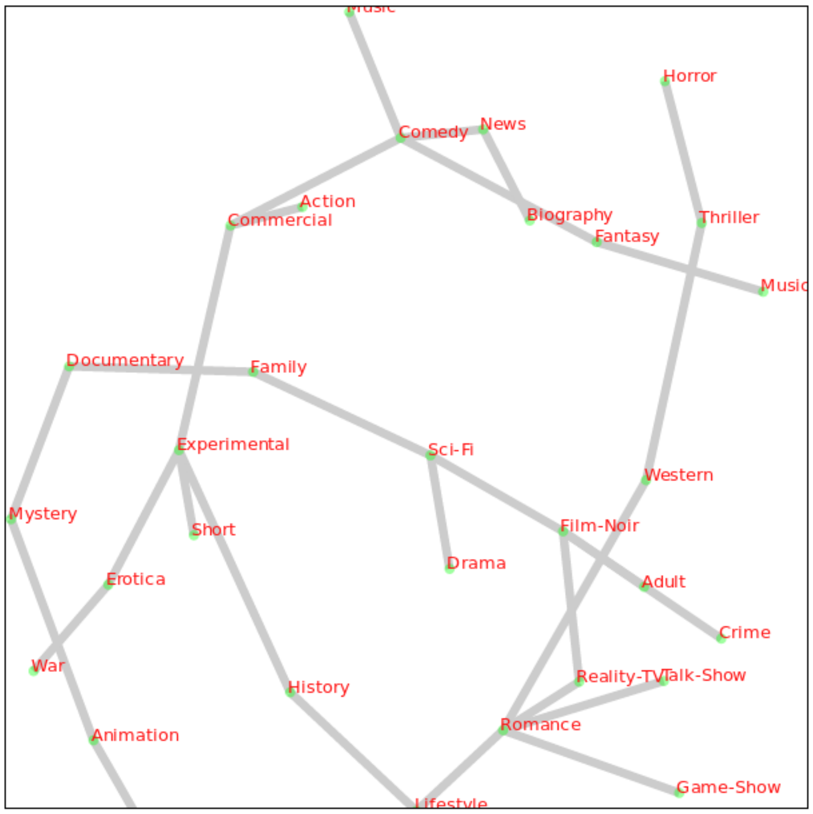


Figure 2.1: Initial placement with momentum.

force and gravity oppose their movement. Upon laying out nodes in their initial positions, edge crossings are introduced to the layout.

Definition 4. **Edge Crossing:** Given four nodes N_a, N_b, N_c, N_d and two edges connecting pairs of nodes E_{ab}, E_{cd}, an *edge crossing* occurs when the two edges E_{ab} and E_{cd} intersect i.e. there exists a point p in the window such that both E_{ab} and E_{cd} contain p. Due to the forces involved in the automatic layout generation, nodes in E_{ab} will not lie directly on nodes in E_{cd}. This means that p can only belong to the set of points in $\{E\}$ if N_a and N_b lie in between N_c and N_d but on opposite sides of E_{cd}.

Annealing

In order to remove edge crossings and reposition the nodes, the simulation is *annealed.* The simulation 'ticks', receiving an increase in energy. This energy translates to an

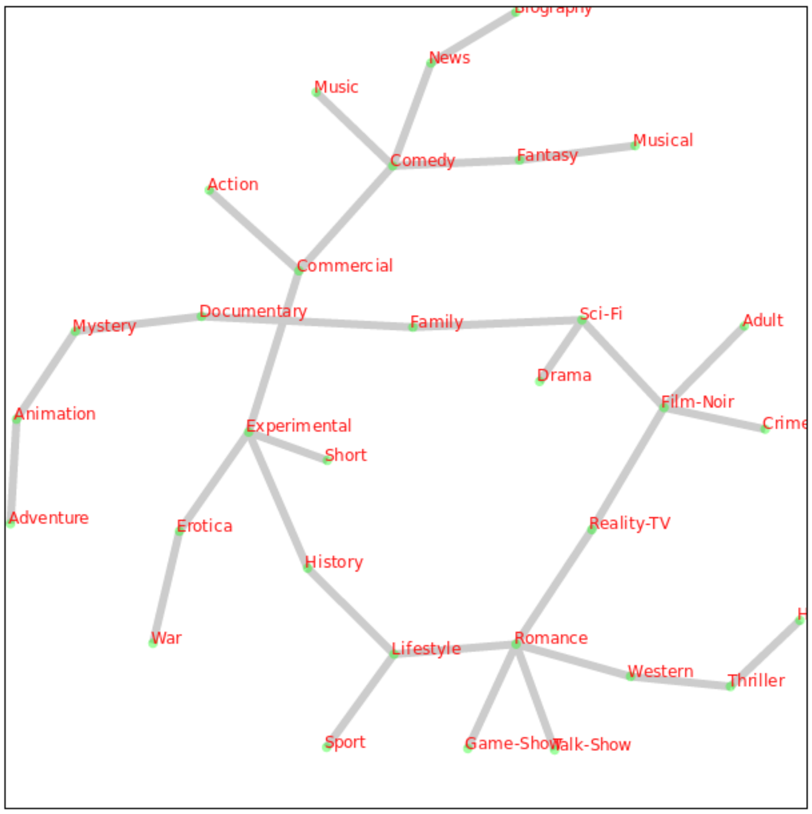


Figure 2.2: Initial annealing.

increase in momentum for the nodes of the tree. Nodes will shift around the canvas in an attempt to produce an untangled tree. At this stage, collision resolution is disabled. The goal of annealing is to satisfy the objective described in *Problem 2*. While annealing at this stage does not directly contribute to a reduction in inter-node whitespace, the removal of edge crossings will facilitate later stages in the pipeline to achieve the objective described therein.

Consider equation 2.2. This equation governs movement of nodes in the system. In the annealing phase, equation 2.3 drives the motion of all nodes on each tick of the simulation. Node position is settled by the degradation of node velocities. This is tuned by an internal decay coefficient f of 0.9. Using figure 2.2 and the IMDb dataset for reference, genre positions are calculated according to equation 2.2, and move at a rate of $v(i) = v(i-1) \times f$ to their position.

Figure 2.3 demonstrates the annealing of nodes progressing by introduction of energy

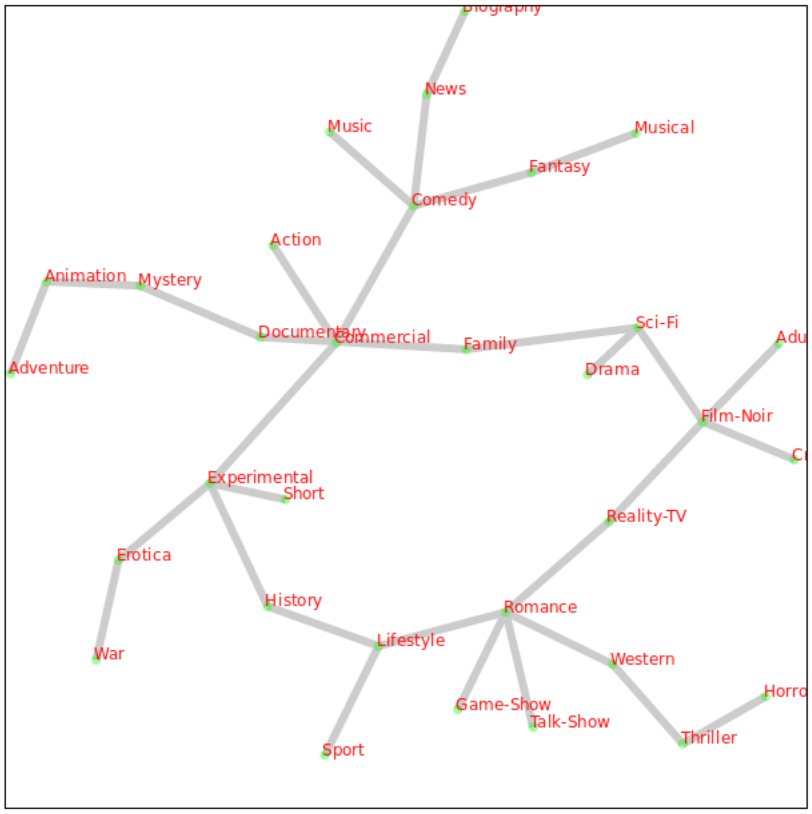


Figure 2.3: Intermediary annealing.

to the system. As the annealing phase continues, nodes are displaced by increase in momentum caused by the simulation receiving energy. These nodes drag their neighbours as they travel, contorting the shape of the tree. As nodes are shifted through this introduced energy, the secondary goal alluded to in *Problem 2* is approached - the geographical neighbouring position of nodes directly connected by an edge.

As shown in Figure 2.4, the general structural layout of the maximal spanning tree has approached a stable state. Branches are not horribly displaced further by introduction of energy to the simulation, as nodes are not pushing each other away to alter the tree's layout.

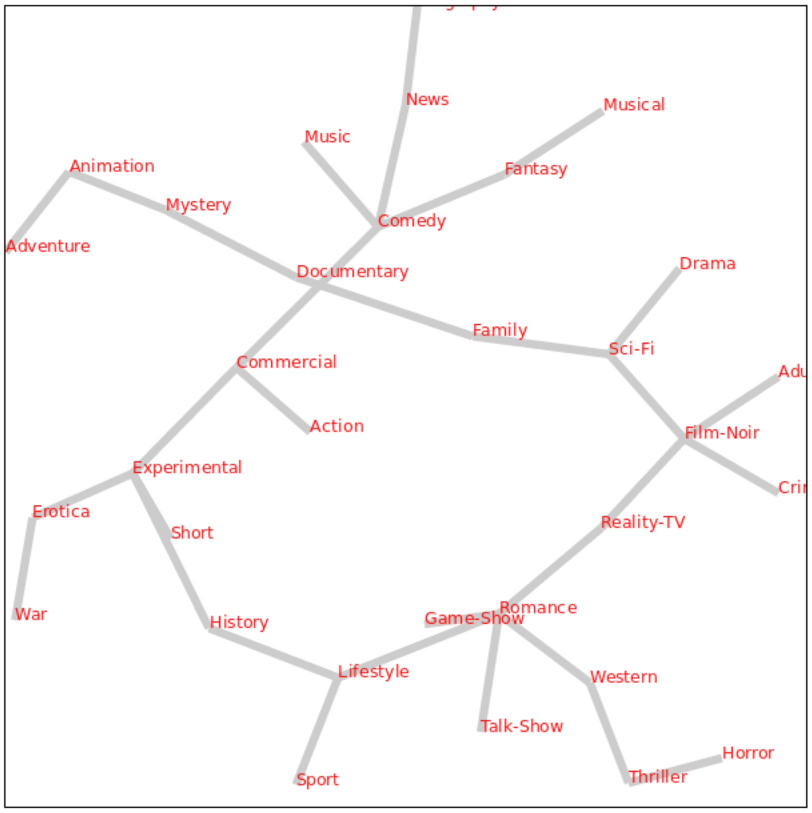
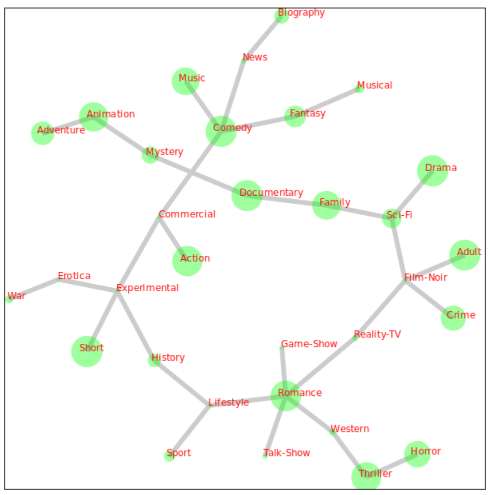


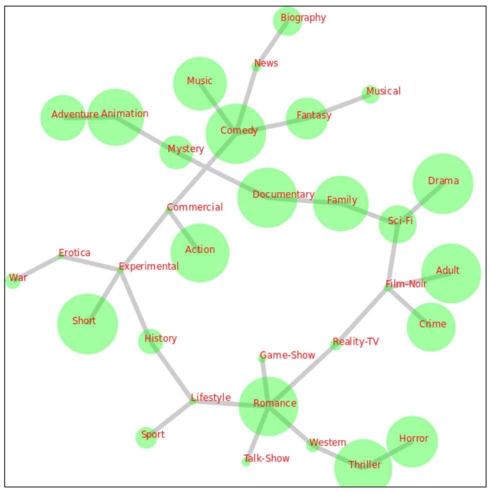
Figure 2.4: Final anneal.

Resizing

Once the node positions have been finalized, the pipeline is ready to begin solving the second constraint as described in Problem 3. This constraint is focused on ensuring the representative size of the node matches the population representation of the node's value in the source dataset. Directly, a node's representative size of the distribution of tuples that contain that node's value can be achieved using a non-linear scaling function. Window W has a finite set of pixels as mentioned in *Definition 1.*. To provide a visual indicator of the representation of a node's values in the input data, the percentage of tuples that contain said node's value can be approximately translated to the window. The percentage of pixels a node occupies in the window approximately represents the percentage of tuples that contain that node's value for the indexing attribute.



(a) Initial resize.



(b) Layout begins to perturb due to collision detection.

Figure 2.5: Two node resize steps.

Figure 2.5a shows the initial resizing of nodes. The nodes begin resizing according to Equation 2.4. Nodes are resized as a percentage of their final size, so even smaller nodes will increase in size on the first iteration. These current sizes are not indicative of the tuple value distribution due to the stepwise resizing approach, so further iterations of resizing are to be performed.

An additional resizing step is demonstrated in Figure 2.5b. With the nodes being bound to the window's walls, the tree structure will slightly warp as nodes on the periphery push against the hard boundaries. This warping effect contributes to a change in the amount of inter-node space, touching on the objective described in *Problem 2* while also moving toward completion of *Problem 3*.

The shift in tree structure is clearly visible in Figure 2.6. This conveniently reduces the amount of space some branches will have to travel during the compacting process, impacting the objective of *Problem 4*. The previous two figures show the interplay between the shift in quality of multiple constraints, an interesting side-effect of force-directed layouts being used to solve for constraints tareting positioning and space.

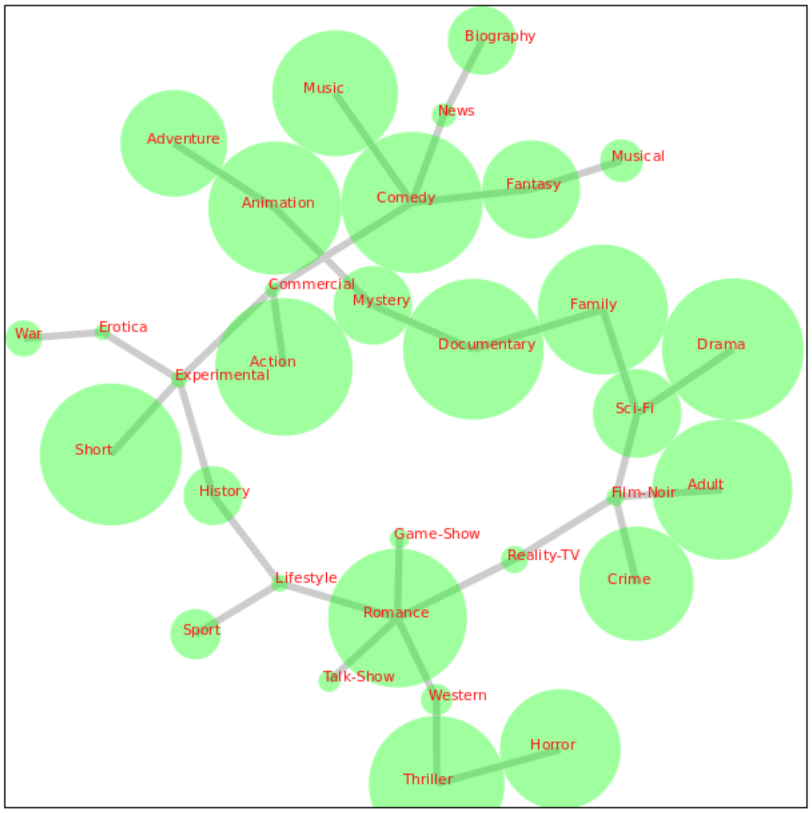


Figure 2.6: Final resize.

Compacting

Once the resizing phase has completed as shown in Figure 2.8b, the system can move to resolve the final constraint described in *Problem 4*. In order to compact, the simulation parameters are tuned as discussed previously. Gravity, charge and edge strength draw the nodes together into a cluster. Equation 2.2, as previously mentioned, governs the movement of objects in the window. During the compacting step, charge and gravity parameters in this equation are iteratively tuned to produce a stepwise compacting effect, collapsing the tree layout into itself.

Refer to Figure 2.7. As the charge parameter is reduced in a stepwise fashion, the reduction of used space is gradual in nature. This allows nodes to slowly fill into place, rather than be jarringly thrown across the canvas. A gradual compacting of the layout is

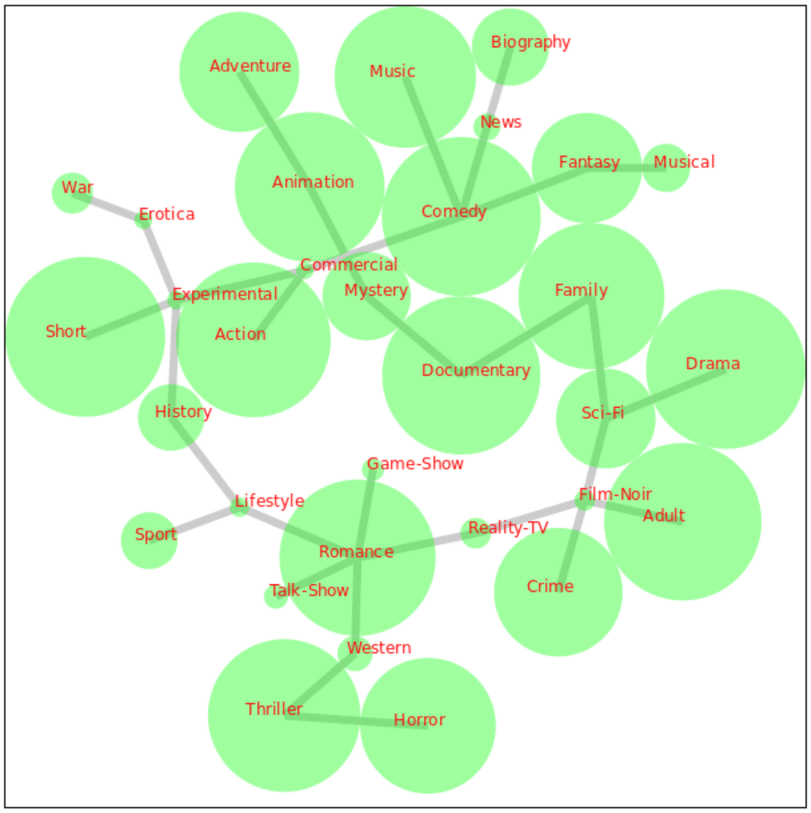
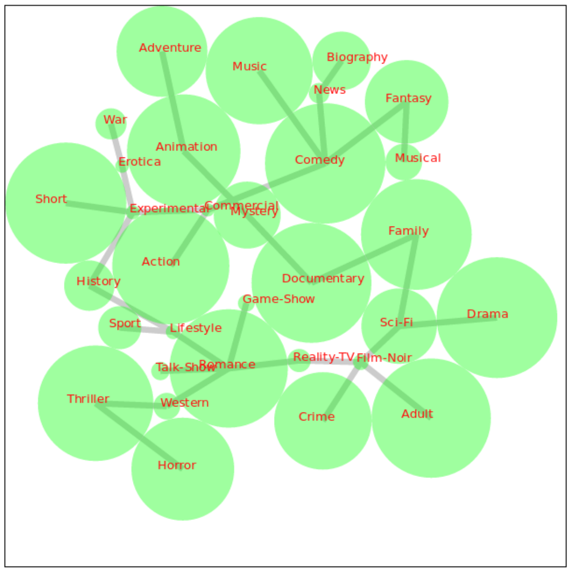


Figure 2.7: Tuples placed within the window.

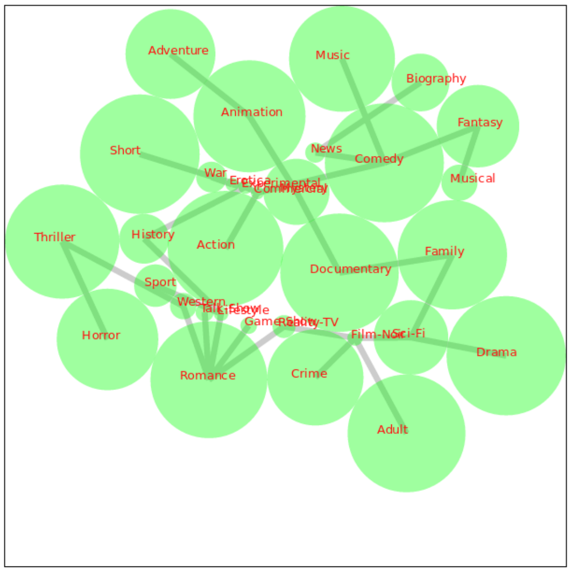
important in order to minimize the disturbance to the objective described in *Problem 2* regarding the positiong of connected nodes as neighbours.

In Figure 2.8a, the layout is mostly compacted after the second compacting step. There is still some whitespace present, indicating that *Problems 2* and *4* are in the process of being resolved, but that the constraints are not yet resolved.

Examination of the smallest nodes shows that they have shifted around the large nodes that they are nearby, or attached to, as shown in Figure 2.8b. This slightly reduces the unusable whitespace that poses an issue. When the indexing attribute does not have an excessively high number of nodes, solving the objective described in *Problem 4* does not pose a threat to the quality of *Problem 2*.



(a) Intermediate compacting.



(b) Fully compacted.

2.2.2 Population

With node positioning finalized, the system preserves the layout for future use. An example usage of the layout generated by the system is shown in Figure 2.9. The visualizatyion, implemented using the d3 *hexbin.js* plugin, distributes hexagonal cells across the area covered by the generated layout within the canvas. A cell's colour is determined chiefly by the node that it rests within. Each node within the layout has a colour associated with it, selected randomly. Any cells that lie directly within the area of a given node's position in the final generated layout are assigned the colour of the node. In the cases where a cell is positioned in between two nodes, the cell's colour is determined by the node it is closer to. Additional parameters such as border thickness and opacity are available to be leveraged in application-specific contexts. Exploration of the use of these parameters will be exercised further in Chapter 3, where a use case of the layout will be provided alongside with methods in which the characteristics of the hexbin library can be used to provide information to the user.

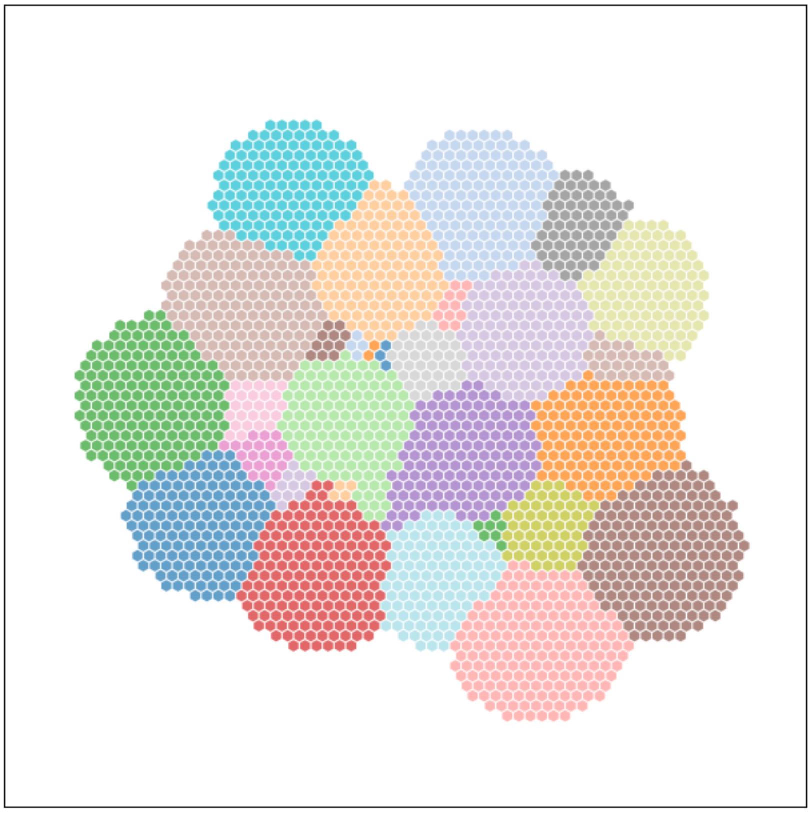


Figure 2.9: One possible use of the layout, differentiated with colour and tiling.

2.3 Implementation

A version of the described system has been implemented in a server environment. [1] In order to implement the pipeline technique, a server has been written in ClojureScript that calculates similarities for a given attribute, creating the input maximal spanning tree stored as a JSON file. Within the JSON file, the degree of similarity between each node is provided, along with the count of entries for each individual attribute value. Accepting this information enables the server implementation of the layout generation simulation to exclude the similarity generation task from the pipeline itself, making it an external component that allows the pipeline to accept data based on any possible similarity metric.

The system is implemented using a combination of Clojure and JavaScript libraries,

[1]The system is hosted at *db.science.uoit.ca/share/imdb*.

code generated through the ClojureScript compiler. The *d3.js* library is used to implement a force-based simulation that resolves the layout generation process. Clojure's *core.async* library contains several relevant constructs required for asynchronous execution. To prepare for a presentation of the core constructs and technologies used in the implementation, a discussion of Clojure is necessary.

2.3.1 Clojure

Clojure is a functional programming language that runs on the Java Virtual Machine and the JavaScript engine. The Clojure language is a Lisp dialect designed for concurrent program development. The Clojure compiler extends the Lisp philosophy that treats functions as data, which allows for code chunks to be accessed and transformed. This property of *homoiconicity* is powerful in conjunction with the core syntax of Clojure, the *S-expression*. An S-expression is a nested tree structure, the leaves of which are either atomic values or S-expressions. In Clojure, functions are written as S-expressions, with the leaves being either values, symbols, or function calls. The nested functions, being S-expressions, can also have a number of atoms or function calls as leaves. The syntax of Clojure is demonstrated in the following toy function definition.

```
(defn sum [x y]
  println (+ x y))
```

Figure 2.10: A demonstrative function definition.

The S-expression in Figure 2.10 is a very simple function that, given two parameters, prints the sum of the two input parameters. The function forms an abstract syntax tree with a height of 4. The value of the sum expression is the atom of the S-expression containing the println function call. In the system implementation, a number of more complex functions have been written that demonstrate nested S-expressions in practical contexts. Consider the functions shown, which identifies and corrects any overlapping nodes in the layout. A number of S-expressions are present in both functions, and the use of functions as data is present as well.

```
(defn resolve-collision!
  [p node]
  (let [dx (- (.-x node) (.-x p))
        dy (- (.-y node) (.-y p))
        l (sqrt (+ (* dx dx) (* dy dy)))
        r (+ (.-size node) (.-size p))]
    (if (< l r)
      (let [l (* 0.5 (/ (- l r) l))]
        (set! (.-x node) (- (.-x node) (* dx l)))
        (set! (.-y node) (- (.-y node) (* dy l)))
        (set! (.-x p)    (+ (.-x p) (* dx l)))
        (set! (.-y p)    (+ (.-y p) (* dy l)))))))
```

Figure 2.11: Functions in the system implementation resolving collisions.

The resolve-collision! function determines the distance between two given nodes as well as their combined radius. If the distance between the two nodes is less than their combined radius, a collision is detected. The determination of the distance between nodes, as well as the determination of the combined radius are both S-expressions that contain further S-expressions. The logic contained within the let statement that adjusts the positions of the nodes such that they are not colliding is another S-expression. In other environments, the adjustment of variable values along with function calls that use their updated values may be written sequentially, but S-expressions enable the nesting of these separate tasks.

2.3.2 Concurrency

When building a system that has both an active user-facing front end responsible for portraying information to and receiving input from a user, as well as data processing handling in a non-visible back end, problems impacting one end of the system can arise. Components that are self-contained operate without relying on input from any other system unit, in turn preventing other units from waiting on results of computations being performed by these units. A simple example of unit-dependent functional interference with respect to the implemented system involves the presentation of a specific step of the layout generation to the user. The front end layout can only present the state of a step of layout generation once all components of the layout that are being manipulated by a function in the pipeline have had their parameters adjusted. While even one component is still in processing, the visual presentation must wait for a response on the state of that component before updating the canvas.

With respect to the pipeline, in the case of annealing, all nodes have their positions updated based on forces acting on them both within the system as a whole and from other nodes in the vicinity, as described previously in Equation 2.3. If computation of the sum of forces on each nodes' position change is performed quickly, then calculating each node's positional change sequentially is not an obstruction - the task completes quickly and the front end will not wait on the result. However, if the calculation of the change in a node's position takes a lot of time, then performing these calculations in order will force the front end to wait longer for an updated set of positions. This model of task completion, wherein a set of tasks is executed in order, is known as *synchronous* implementation, with *sequential execution.* To bypass the previously-described scenario in which a front end is waiting on slow, ordered task completion, a different model of computation is used.

Asynchronous execution is the programming model by which tasks are completed in an unordered fashion. Consider the calculation of nodes' positions in the annealing stage of the pipeline method of layout generation. In a synchronous environment, each node's positional change would be calculated in sequence. Asynchronous execution distributes each node's positional change calculation to a concurrent construct, which are all performed simultaneously and independently of each other. The distribution of tasks across multiple worker units results in high reduction of run time investment for a high number of computations. With respect to the pipeline layout generation system, this improvement in performance via reduction in time required for computation is relevant in two ways. Firstly, as mentioned before, systems which have a front end waiting on state updates reduce the possibility of encountering a wait, maintaining a responsive environment for user interaction and application. The second relevant factor comes in the form of scalability. Distributing tasks across a set of workers allows for a system to produce layouts on attributes with a higher potential number of nodes than in synchronous implementations, which enables systems like the layout generation system to generate layouts on a larger range of attribute node cardinalities.

Ahead, a concurrent construct functioning as a cornerstone of the implementation will be introduced and discussed. The definition and functions of the *channel* will be provided, and the use case of the channel with respect to the layout generation system will be explained. *Go blocks* are also presented, a macro that implements asynchronous behaviour using channels.

2.3.3 Channels

The *channel* is a construct that processes can use to store and retrieve information. Processes and channels are defined in the *core.async* library.

Definition 5. **Process:** A process is a *concurrent* logical object that operates on observing an *event.*

The process is the basic operator through which asynchronous execution is enabled. A set of processes is generated by the *go-block* based on the number of cores of the currently-used computer. These processes are run in a thread pool with a finite, defined number of threads. Processes run on different threads in the thread pool as events on listened objects are detected, enabling asynchronous task execution. Processes listen to events occurring within *channels*.

Definition 6. **Channel:** Consider data of any type as a *message M* generated by work performed by a process P. A *channel* is a construct that accepts a messages created as a result of work done by one of a set of processes $P_1, ..., P_n$ and stores them. The number of messages a channel can hold is described as the *channel buffer*. Any given P can *put* a message in a channel, or *get* messages from a channel based on which channel/s P is watching. If a channel has more messages placed on it than its defined capacity, the thread that the channel is defined on *blocks*.

The basic functions of a channel can be shown in the code presented in Figure 2.12. As a channel is a construct that receives values and stores them to be extracted by other processes, operations and channel access occur in the context of threads.

```
(let [ch (chan 10)]
  (thread (>!! ch "Channel operations in Clojure - in action!"))
  (println (<!! ch))
  (close! ch))
```

Figure 2.12: Basic channel functions and behaviour.

The fundamental operations involved in channel creation and use are shown in Figure 2.12. Channel creation is facilitated using the *chan* function. (chan) can optionally be called with a numerical parameter; the number provided to the function defines the buffer size of the channel, or how many values a channel can hold before it forces blocking putting additional values. Two other methods of channel creation set up the channel such that the action of putting values does not block - when the buffer is filled, a value is dropped from the channel. These functions are *(chan (sliding-buffer n))*, which creates a buffer of size *n* and removes the oldest value in the buffer when it is full, and *(chan* *(dropping-buffer n))*, which is similar to *sliding-buffer*, but removes the newest value from the buffer when the buffer is full. Filling the channel is done by putting values in the channel, using expressions containing the ¿!! *blocking put* operator. These values are referenced using the ¡!! *blocking take* operator. Blocking is one of two classes of waiting, and is described further ahead. When using standard threads, put and take operators must stop execution until there is either space to put values in a channel or there is a value to remove, respectively.

2.3.4 Go-blocks

Much of the asynchronous functionality of channels comes from the go-block. Consider the following code:

```
(def ex-chan []
  (let [ch (chan)]
    (go (>! ch 4))
    (assert (= 4 (<!! (go (<! ch)))))
    (close! ch))
  )
```

Figure 2.13: Basic go-routine syntax and use.

In examining the go block in Figure 2.13, the basic semantics of go blocks are presented. A go block encapsulates an expression that interacts with a given channel. There are two go-routines present in Figure 2.13. The first go-routine places a values in the generated channel *ch* while freeing the thread for execution. This go block demonstrates the structure of a go block in a simplistic fashion - go blocks consist of the go macro followed by an expression that interacts with a channel. Go blocks can be used at any depth within an S-expression. The second go block takes the value present within the defined channel and checks whether it satisfies an expression. Both go-routines create a thread on execution

- using a go block within a doseq expression creates a thread each time a go-routine is executed. The body of the go block executes asynchronously on the thread generated by the associated go block.

The concurrency model within *core.async* defines two different types of waiting. One previously mentioned type of wait is *blocking* - a thread will hold execution of tasks until the current task is completed. This model of waiting prevents further execution, which in turn inhibits the responsiveness of an application. Large amounts of blocking can interfere with the speed of task completion - threads blocking do not perform work, which delays the completion of their task set. The second type of wait is *parking*. When a given process is waiting when park waiting is used, the waiting process is removed from the thread it is on, and another process is placed on the thread instead. The difference between parking and blocking can be summarized thusly: with *blocking*, a waiting process remains on the thread until completion, while with *parking* the waiting process is moved off of the thread and multiple process execution timings are interleaved on the thread instead, maximizing the amount of time that a given thread is allowing non-waiting processes to execute. With respect to processes working on channels, the two operations of interest are *put* and *take*. The operations >! and <! are *parking put* and *parking take*, respectively. As explained, these free the current thread to execute another process. The other two operations >!! and <!! refer to *blocking put* and *blocking take*. As discussed, blocking operations do not free up the current thread to execute another process.

Within Clojure, a go-block (or a go-loop) is a concurrent construct. A go macro signals a position where code is to be executed asynchronously. The go-block itself returns a channel immediately to the thread that calls it. The body of a go-block will then execute asynchronously, and the results from the completion of the body evaluation will be put into the returned channel, where the returned data can then be taken and processed or used elsewhere in the system.

2.3.5 Channels and Go Blocks in the Pipeline

With the two core asynchronous tools, channels and go blocks, described in detail, it is now possible to explore how these constructs are used in the system. Channels and go-routines enable an asynchronous process execution method that enable the layout generation pipeline to function.

The pipeline system implementation utilizes channels to identify which stage of the pipeline the layout generation process is currently in. The associated function is shown in the following figure. The contents of a given message are the *state* of the simulation. In the implemented simulation, a simulation state is composed of a set of nodes, edges, and simulation force parameters. Within the context of a constraint-resolving pipeline, processes are defined that operate based on the current stage of the pipeline. The messages possibly stored within a channel specify the stage of the pipeline that the simulation has currently progressed to. The system incorporates two channels; the first channel C_1 stores the progressed step of the pipeline, while the second channel C_2 stores the current step. Processes take the current state from C_2, checking what stage of the pipeline is to be executed. Once the stage has been identified, the corresponding tasks are evaluated, progressing the state of the simulation. The process that took the message from C_2 increments the message and places it in C_1. C_2 is then updated with the message in C_1. Once the message identifies a non-existent stage of the pipeline, the channels are closed. A *closed channel* does not accept any further messages put in the channel. This prevents the system from attempting to continue when tasks are not defined further.

The system's use of channels is shown in Figure 2.14. The previously-described series of operations are defined within a pipeline function. The pipeline function demonstrates the culmination of all previous implementation discussion with S-expressions containing functions as data, channel definition, putting and taking, and go-blocks that return channels as well as function body value passing to channels.

```
(defn pipeline
  [ ]
  (let [c1 (chan)
        c2 (chan)]
    (go-loop [n 0]
             (let [ping (<! c2)]
               (println "[PIPELINE] Received" ping)
               (let [stage (get-stage n)]
                 (if stage
                   (do (>! c1 stage)
                       (recur (inc n)))
                   (do (js/console.debug "Closing c1")
                       (close! c1))))))
    [c1 c2]))
```

Figure 2.14: The pipeline function.

Chapter 3

Applications

This chapter will provide a relevant application of the layouts generated by the pipeline algorithm. Systems that produce query results with extremely high volumes of tuples are commonplace and used in a variety of scenarios, including data analytics, data cleaning and business logic. One such system will be referenced and described briefly. The system in question will then have its results evaluated with layouts generated by the pipeline system. The objective is not only to show that the results of the query answer system are presented visually, but that the incorporation of layouts allows further intuitive comprehension and pattern recognition of the provided answers. This application can then extend to other systems that operate within the same domain.

3.1 Jaccard Similarity as a Ranking Measure

Consider a system such as the one described in [15]. The system, provided with a labeled graph using previously-defined node and edge weights with an accompanying set of keywords as a query, identifies a set of trees containing all the keywords in the query. Each tree is a join-network of shortest path connections between nodes. This set of trees is ranked according to a defined quantity - the edge weight objective.

Definition 9. **Edge Weight Objective (EW):** Given an answer tree T composed of a set of nodes connected by a set of edges $e_1, e_2, ..., e_m$, the answer tree's edge weight is defined as $EW(T) = \sum_{i=1}^{m} w(e_i)$ with $w(e_i)$ defined as the weight of a given edge E_i).

Answer trees in the resultant set of trees are ranked according to the edge weight objective, described previously. The minimization of the edge weight objective is a *NP-hard* problem, as shown in [11]. As the problem is NP-hard, the system utilizes a greedy algorithm to identify answer trees which satisfy a minimized edge weight objective. To efficiently determine the nodes within an answer tree, the system computes a *2-hop cover index*, storing the shortest distance between every node and each other node in the graph.

To find an answer, the system implements a variation of the root semantic approach as described in [11]. Each node in the full graph is considered as a possible root node for an answer tree. To evaluate which trees are suitable answer trees, for each node in the graph, a tree is formed with the closest node that contains the input keywords. Trees with the smallest sum of edge weights are selected as the set of answer trees. The distinct root semantic operation runs in polynomial time, making the answer selection algorithm run in polynomial time.

Further, the system also incorporates an answer selection approach using the Jaccard similarity as defined in Equation 2.1. The system thus provides three methods for returning sets of answer trees that contain input nodes. The join networks can be determined and ranked according to minimization of the edge weight, ranked according to the Jaccard similarity functioning as a weight, or with a doubly-weighted combination of both of the previously-mentioned weights, each weight scaled by an input lambda coefficient.

To show the difference in results generated by these different weights, consider the system as presented in Figures 3.2a and 3.2b. [1] The Keywords field accepts two or more input keywords contained within double quotes to signify individual keyword entries. The user can select either the edge weight minimization, the Jaccard similarity, or the

[1]The eGraphSearch system is accessible online at http://data.science.uoit.ca:8080/graphsearch/web/P1.jsp.

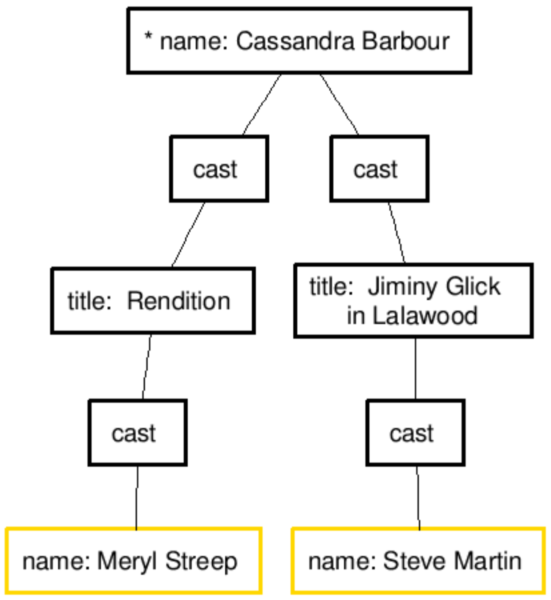
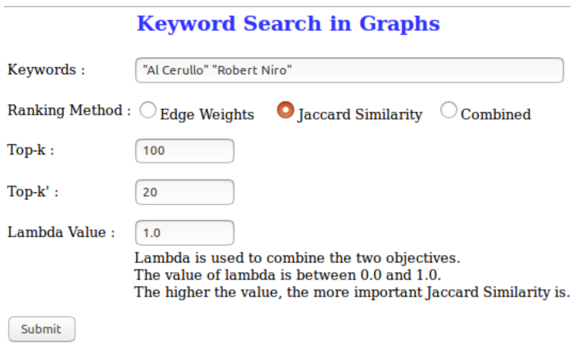


Figure 3.1: Resultant Query

combined objective ranking method. The top-k fields specify how many answers are presented by the system. The top-k' field determines how many answers of the top-k set are selected for presentation. The distinction between these sets is straightforward: top-k' answers are reranked according to the Jaccard similarity scores of nodes within the answer tree from the initial top-k set of answers selected by edge weight minimization. The lambda parameter determines the degree to which each ranking method's input is considered. Depending on the input keywords, this process can produce very diverse sets of answer trees that are suitable for different purposes. To demonstrate the possible variability in result sets, some of the top answers from each ranking method have been selected based on the input keywords "Al Cerullo" and "Robert [de] Niro".

The answer trees presented in Figure 3.3 are the first three answers presented from the previously-mentioned keyword search with the edge weights ranking method selected, with a $top - k$ parameter of 100, a $top - k'$ parameter of 20 and a $lambda$ parameter of 0. The root node of each of these trees are all films - titles that both Al Cerullo and Robert de Niro have acted in. Executing the keyword search using the same input keywords



Keyword Search in Graphs

Keywords : "Al Cerullo" "Robert Niro"

Ranking Method : Edge Weights [✓ Jaccard Similarity] Combined

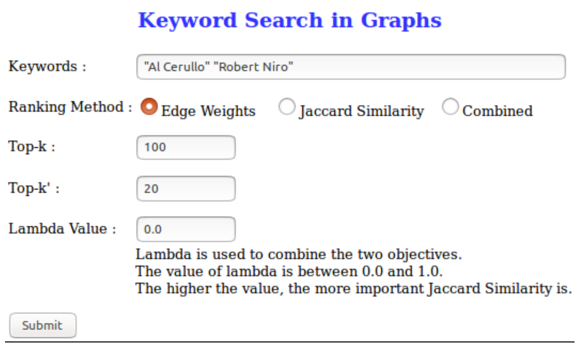
Top-k : 100

Top-k' : 20

Lambda Value : 1.0

Lambda is used to combine the two objectives.
The value of lambda is between 0.0 and 1.0.
The higher the value, the more important Jaccard Similarity is.

(a) Input parameters using Jaccard similarity.



Keyword Search in Graphs

Keywords : "Al Cerullo" "Robert Niro"

Ranking Method : [✓ Edge Weights] Jaccard Similarity Combined

Top-k : 100

Top-k' : 20

Lambda Value : 0.0

Lambda is used to combine the two objectives.
The value of lambda is between 0.0 and 1.0.
The higher the value, the more important Jaccard Similarity is.

(b) Input parameters using edge weights.

Figure 3.2: The eGraphSearch system.

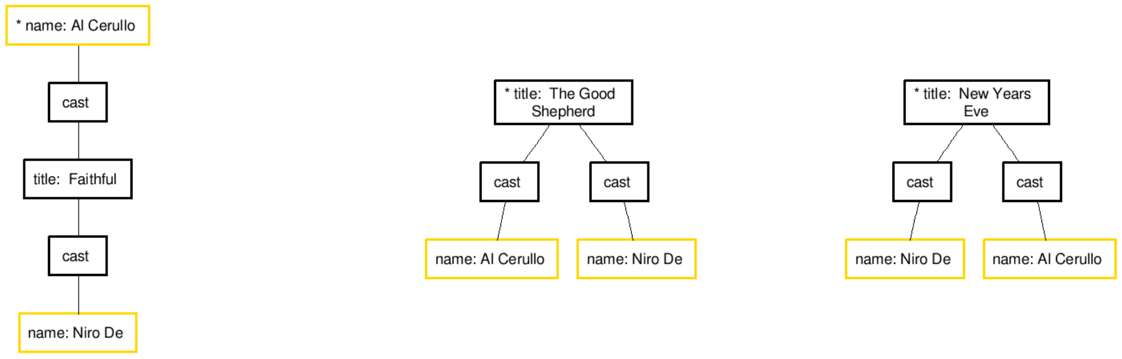


Figure 3.3: The top three answer trees selected based on minimized edge weight.

shows that the first eighteen answer trees all connect Al Cerullo and Robert De Niro based on common projects. These trees are a visual presentation of a subset of rows that would be retrieved if one were to run a query on a movie database searching for films that contain both Al Cerullo and Robert De Niro, and would be of use in the same situations where a user or developer requests project work histories of workers. For many cases, directly querying a database would provide similar results that are easier to work with.

In Figure 3.4, the top three answer trees reranked based on Jaccard similarity are shown based on a $top - k$ parameter of 100, a $top - k'$ parameter of 20 and a $lambda$ parameter of 1.0. Of the initial 100 selected answer trees, each tree is ranked based on their internally-determined Jaccard similarities. On evaluation, each root node is

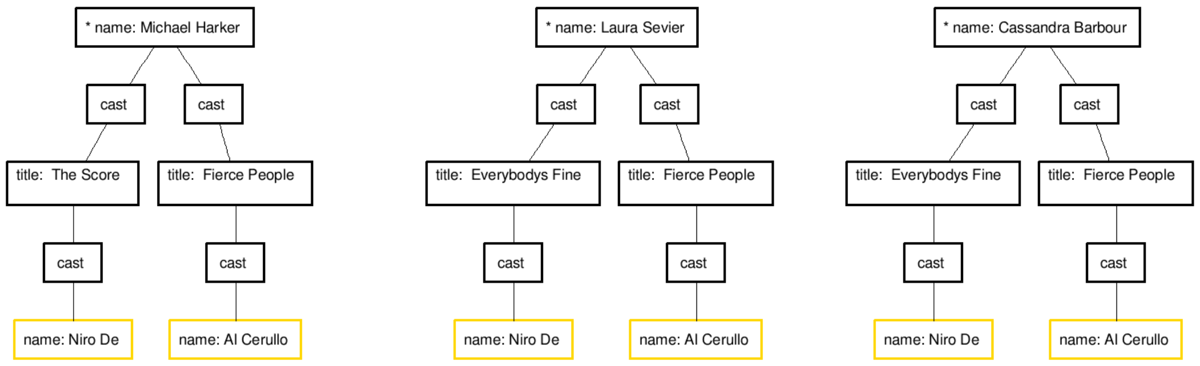


Figure 3.4: The top three answer trees selected based on Jaccard similarity.

shown to be the same type of entity as the input nodes - all the root nodes connecting the input nodes are fellow actors and actresses. Entering the same input keywords and parameters as described with the Jaccard Similarity ranking method selected shows that all top twenty answers have root nodes that are people working in the same industry as Al Cerullo and Robert De Niro. Initially, this may not seem as useful a response as the answers provided by the edge weight ranking method, but a set of applications have rapidly grown that utilize this type of result.

Social networks leverage these kinds of connections in many contexts: personal, professional, academic and more. These answer trees provide valuable data for showing relationships in a social network. With enough connections shown by the returned result, the Jaccard Similarity ranking method can be used to find data required to build domain-specific social networks and analyze the relationships between nodes in a particular domain based on their shared project involvement. In the context of movies, such a network can potentially show which movie industry professionals work together and point out possible commonalities in their genres of choice, or show preferences between professionals in different roles (i.e. which actors prefer to work with a particular director).

3.2 Visually Enriching Query Answers

Answer trees demonstrate a connection between input nodes through common nodes. As an illustrative example, the answer tree presented in Figure 3.1 demonstrates a connecting node Cassandra Barbour between actors Steve Martin and Meryl Streep, connected through a film that Cassandra Barbour has appeared in with Steve Martin (Jiminy Glick in Lalawood), and a work she has appeared in with Meryl Streep (Rendition). The set of films that Meryl Streep has participated in function as the *domain* of movies of Meryl Streep. Similarly, the films that Steve Martin and Cassandra Barbour have worked in are their domains. These domains are used to compare the similarity between actors based on common works.

Application: **Answer tree node relationship presentation.** The enrichment of a given answer tree T_i can be defined as follows. Given the nodes $N_1, ..., N_k$ of the tree that represent input keywords $K_1, ..., K_k$ as well as connecting node(s) $N_{c1}, ..., N_{ck'}$ of the same *type* as input and the nodes' domains $D_a, ..., D_k$, generate layout L using an indexing attribute. Augment the layout nodes in L with visual indicators that rank the top k layout nodes for N_x based on D_x.

The answer shown in Figure 3.1 is one result of a query that, given two keywords $K_1 = "Meryl\, Streep", K_2 = "Steve\, Martin"$ produces answer trees that include one or more leaves connecting the two input keywords. For the input dataset, consider the distribution of tuples according to attribute value *Genre* being laid out in the following image.

Recall the node positions shown by Figure 3.5 In this layout, each of the nodes' final positions is representative of a set of hexagonal cells. The hexagonal cells represent a distribution of tuples within each cell. Cells are coloured according to the colour associated with the node closest to the cell. If a cell lies within a node in the layout, the cell receives that node's colour value. Otherwise, a cell's colour value is equal to the colour value of the node with the shortest distance to the cell. In the layout pictured in Figure 3.5, labels

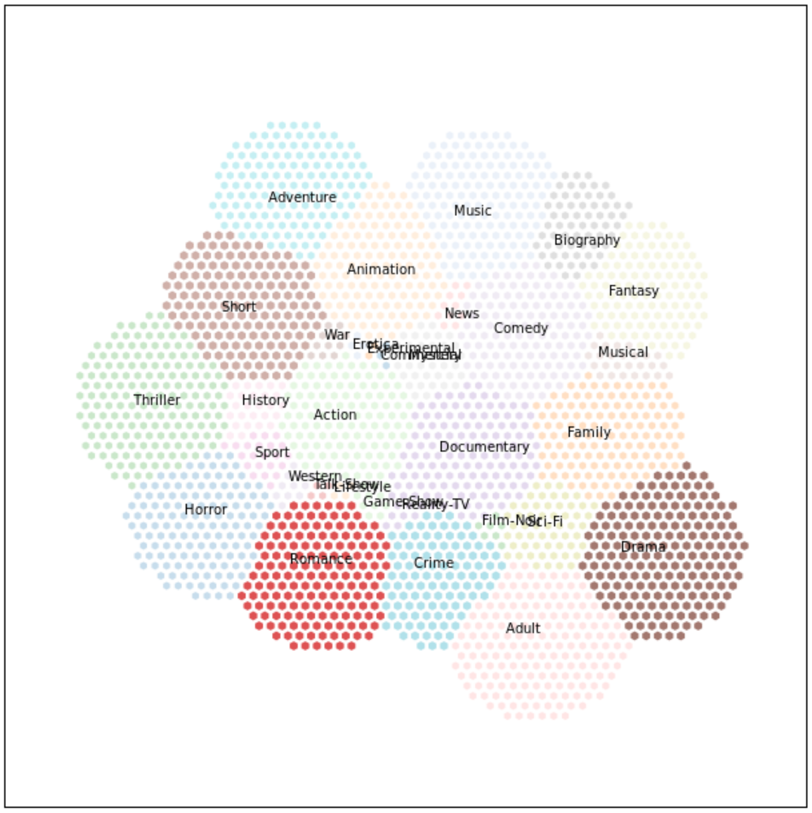
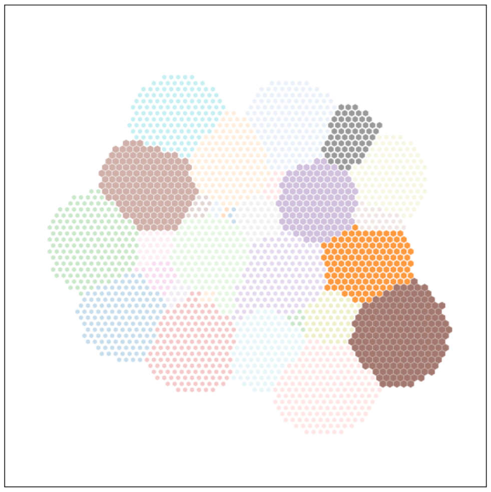


Figure 3.5: Tuple layout by Genre

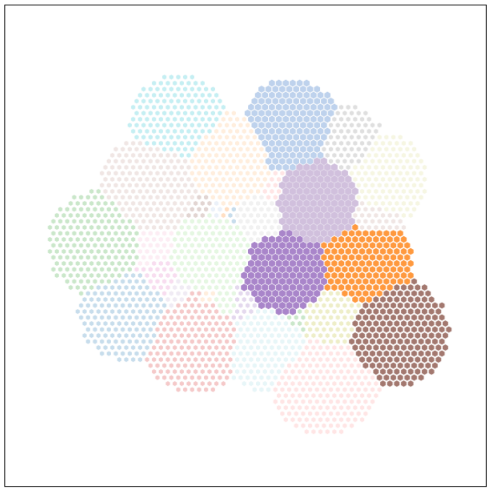
have been added to inform the viewer visually of the node at a given cluster.

This layout can be used to show the concentration of a set of nodes according to the characteristics of a tuple. As an illustration of the solution described previously in practice, the top k layout nodes for a given K are accented and distinguished from each other in the following layouts using a combination of node cell size, node cell border thinness, and node cell opacity. The following layouts are representative of the distribution of movies that the three actors *"Meryl Streep"*, *"Steve Martin"*, and *"Cassandra Barbour"* have participated in.

Figures 3.6a and 3.6b show the distribution of movies that the two input keywords have participated in. Within both layouts, some patterns can be shown - by the data, Steve Martin and Meryl Streep act in similar genres of movies. The border-thinness of a cluster is correlated with the ranking of that genre for a given actor in a *Top-k* list of



(a) Movies with Meryl Streep



(b) Movies with Steve Martin

genres. Given the rankings in D, the top 5 most populated genres will be distinguished visually. To accomplish this, as a node's position in D falls (i.e. is less populated), the borders of cells in that node increase in thickness. The border thickening corresponds to a reduction in the individual node size, visually identifying highly-populated nodes as containing extremely compact cells. Further, as a node's position in D decreases, its opacity also decreases. The effect being sought after is, through these visual indicators, for a given K the user's eyes will be drawn to the most populated nodes. This information augments the data from the answer tree, showing the user that the two input keywords have a degree of similarity of genre participation, if not also in ranking of genre.

In Figure 3.7, the movies that Cassandra Barbour has participated in are shown. In this layout, it is plain to see that there is not a high degree of overlap between the answer node of the result tree, and the two input keywords. This information provides richness to the result of the answer tree, showing the user that while Cassandra Barbour has connected with both Meryl Streep and Steve Martin, the genres that Cassandra Barbour participates in most do not have high overlap with Steve Martin or Meryl Streep.

To summarize, the layouts generated have provided a method by which query answers can be visually presented and augmented to provide the user with richer insight into the

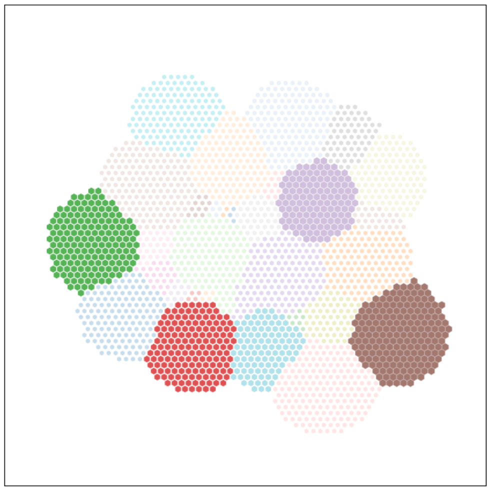


Figure 3.7: Movies involving Cassandra Barbour

answer. The layout generation pipeline has application in query answer presentation in the following ways:

- The base query answer shows node values. Layouts provide a visual representation of the distribution of a node's value in the source dataset. This gives a context for the nodes of an answer tree with respect to the data.
- Showing a node's distribution over the dataset by an attribute gives an at-a-glance summary of a node's presence in the dataset. With respect to the shown layouts, showing the distribution of an actor/actress gives an at-a-glance summary of what genres a given actor prefers to act in.
- In comparing multiple layouts, it is possible to visually identify the similarities and differences between the input nodes and root nodes of an answer tree. As shown, it is possible to see if two actors tend to work in similar or different genres.

These applications are proof of concept of a method by which the pipeline layout generation algorithm can be used to visually present and augment a given query answer. The efficacy of the particular tuning of visual indicators, as well as the effectiveness of various visual variables usable for annotating a layout, evaluated through user study, are designated as possible future work. As the layout provides positioning for clusters of points, visualizations using other techniques for aggregation than hexbins can be utilized. The most effective visualization for a given layout, evaluated through either user study or metric evaluation, is designated as possible future work.

Chapter 4

Experimental Evaluation

In order to experimentally evaluate the ability of the pipeline automatic layout generation process, the quality of the resultant layouts must be analyzed. The quality of a layout is defined as the degree to which all of the constraints in the pipeline are resolved. Different visualization and layout generation techniques that rely on different data structures than trees will not necessarily be able to apply the same metrics that we will be using, as not all quality metrics can be applied across all structures (i.e. a TreeMap visualization will not be able to use edge crossing reduction to evaluate its quality).

The pipeline layout generation system produces layouts using a maximal spanning tree. In order to determine the quality of layouts generated by the system, quality metrics for each phase of the pipeline must be identified. The pipeline works with three specific constraints: the generation of a node positioning where connected nodes are neighbouring nodes, the size of each node is visually representative of the tuples in the dataset that have that node's value, and the space occupied by the layout is minimized as much as possible without negatively impacting the visibility of nodes within the generated tree. To identify the quality of the layout with respect to each of these constraints, three quality evaluation metrics have been selected. The quality metrics of total edge crossings, layout compactness, and node size distribution will be discussed in the following sections.

4.1 Annealing Quality Evaluation

To determine the quality of the layout with respect to preserving the neighbouring spatial relationship between connected nodes, confounding factors that can impede the ability of two connected nodes to be placed next to each other must be identified. One of these factors is the crossing of unrelated pairs of nodes.

To understand this metric for quality evaluation, recall Definition 3. Edge crossings prevent connected nodes from positioning such that they are nearest-neighbours to each other. This positioning must be upheld in order to satisfy the objective of *Problem 2*, the minimization of unused space by positioning connected nodes next to each other.

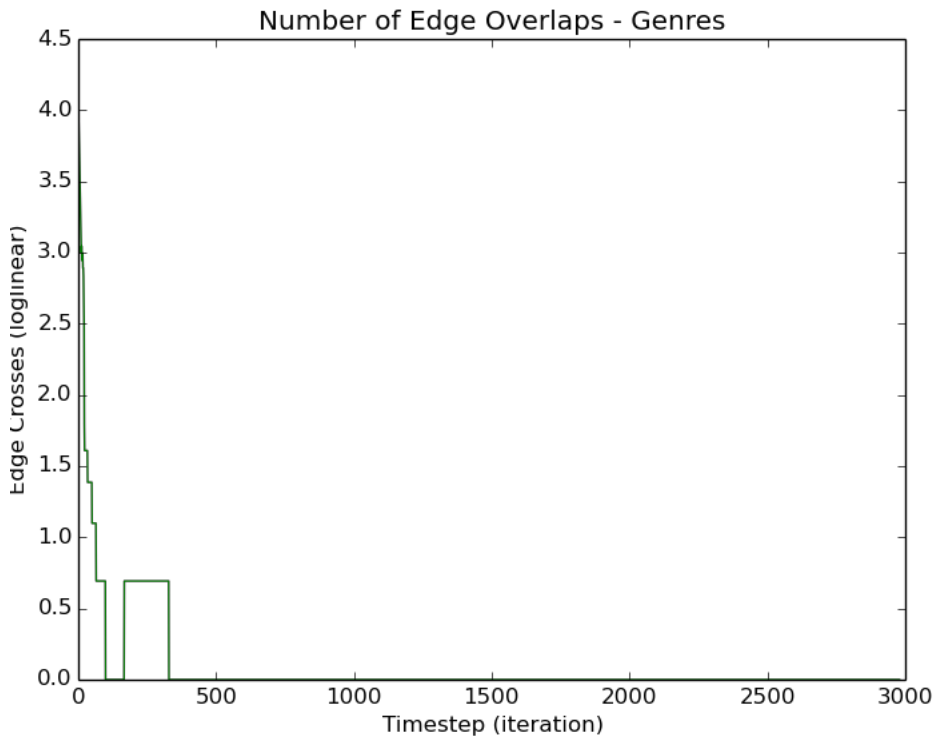


Figure 4.1: Edge Crossings Present In Genre Layout Over Time

In Figure 4.1, the number of edge crossings between branches is shown for the layout generated using the *Genre* indexing attribute with 32 nodes. With a lower number of distinct nodes present in the indexing attribute, the figure shows that edge crossings are drastically reduced, with few to no edge crossings reintroduced in future constraint resolution phases. The slope shows that, for attributes with a lower number of nodes, the quality of constraint resolution at all points in the pipeline does not suffer instability or degeneration.

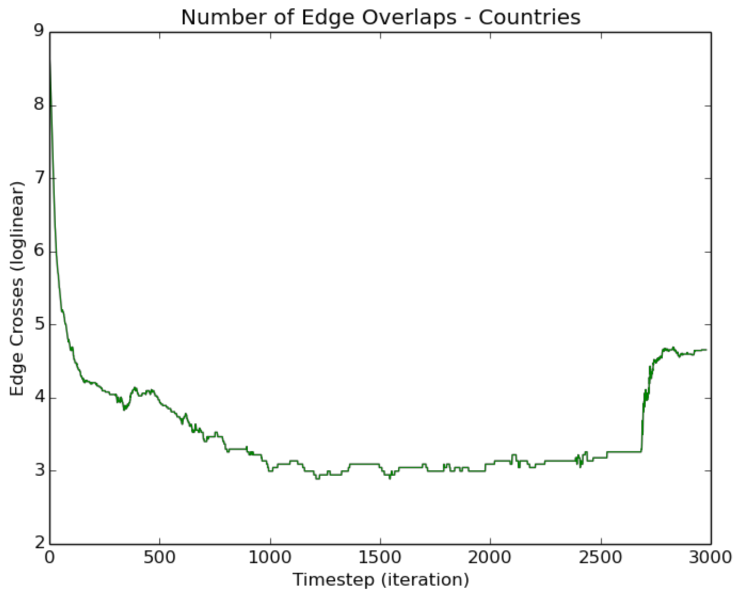


Figure 4.2: Edge Crossings Present In Country Layout Over Time

The above figure plots the number of edge crossings in the layout at all points in the pipeline for a layout generated using *Country* as an indexing attribute. This indexing attribute contains 235 nodes, a much higher number of nodes from the previous attribute. With 235 nodes, the generated tree has 234 edges. With a much higher number of edges, the number of possible edge crossings increases greatly. Further, the resolution of this constraint is unstable in later stages of the pipeline. This phenomenon occurs due to multiple factors. The first factor is the increased volume of edges. Due to edge volume being higher, the probability of edges intercrossing in the same canvas space is higher. The second factor involved in a higher chance of edges intercrossing is related to the population distribution of the data points. This attribute has a similar number of input data points as compared to Genre, with seven times as many nodes and edges. This distributes the points into many more nodes, reducing the average node size. Since larger nodes are less likely to reposition in the compacting process and smaller nodes are more likely to be repositioned due to gravity and repulsion, these nodes have a higher chance of being repositioned across a node.

4.2 Compactness

To determine the compactness of the layout, one method of evaluation involves identifying the dimensions of the bounding box of the layout and determining the area of the window that the bounding box covers. During the layout generation process, a bounding box of the layout initially will take up the majority of the

Definition 4. **Bounding Box:** Given a layout L consisting of nodes $N : \{N_1, N_2, ..., N_n\}$ connected by edges $E : \{E_1, E_2, ..., E_{n-1}\}$, the bounding box B of L is a rectangle that encompasses the absolute leftmost, rightmost, topmost and bottommost points contained by nodes. The leftmost point contained within a node is a point p_l with the lowest x value. The rightmost point p_r has the highest x value. The topmost point within a node is defined as the point p_t with the lowest y value, and the bottommost point p_b has the highest y value of all p within any N.

Definition 5. **Layout Compactness:** The compactness C of a layout L is the bounding box of L contained within the dimensions of the window W. It is the percentage of the window covered by the bounding box, or $(width_B * height_B)/(width_W * height_W)$.

The compactness of nodes in the genre attribute is shown to increase greatly in the associated pipeline step, the final stage in the graph in Figure 4.3. As the compacting phase iterates, the nodes collapse in to a minimized space. This effective compacting is partially due to the uneven distribution of points across nodes; a lower number of larger nodes will act as landmark nodes, shifting smller nodes around them. The smaller nodes become drawn in close to the larger nodes they are connected to, filling in the gaps of space between larger nodes. Due to there being a relatively uneven distribution of nodes

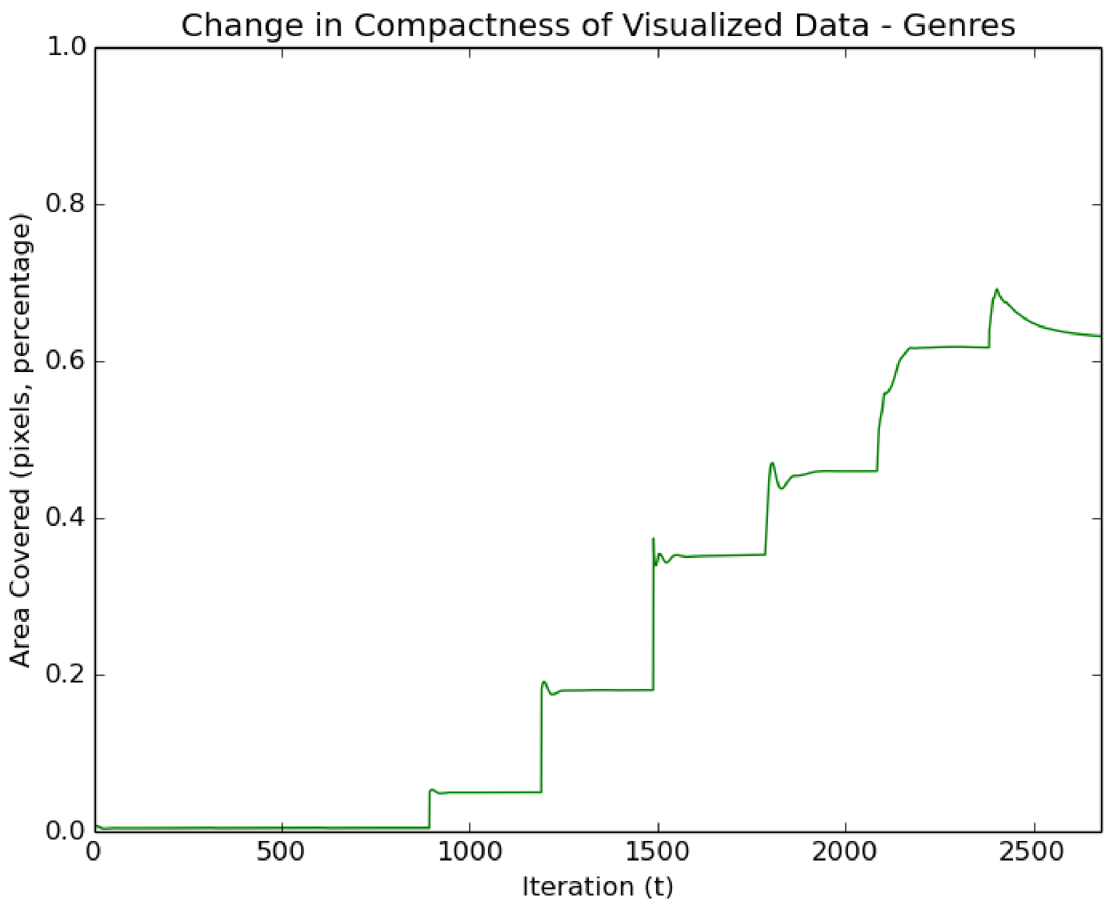


Figure 4.3: Genre Compactness

and a high variance in node size, pockets of whitespace remain in the final layout that cannot be reduced.

The characteristics of the country attribute that contribute to the increased number of edge overlaps present are relevant in the analysis of layout compactness. The distribution of entities to nodes is still fairly uneven, but the number of nodes present with fewer members is much higher, leading to a higher number of smaller nodes present in the layout. These smaller nodes will be pulled into pockets of whitespace in between larger nodes, filling more space due to their lower diameter and higher count. This contributes to a layout that maintains a high degree of compactness as node count scales up, showing stability in the satisfaction of compactness resolution.

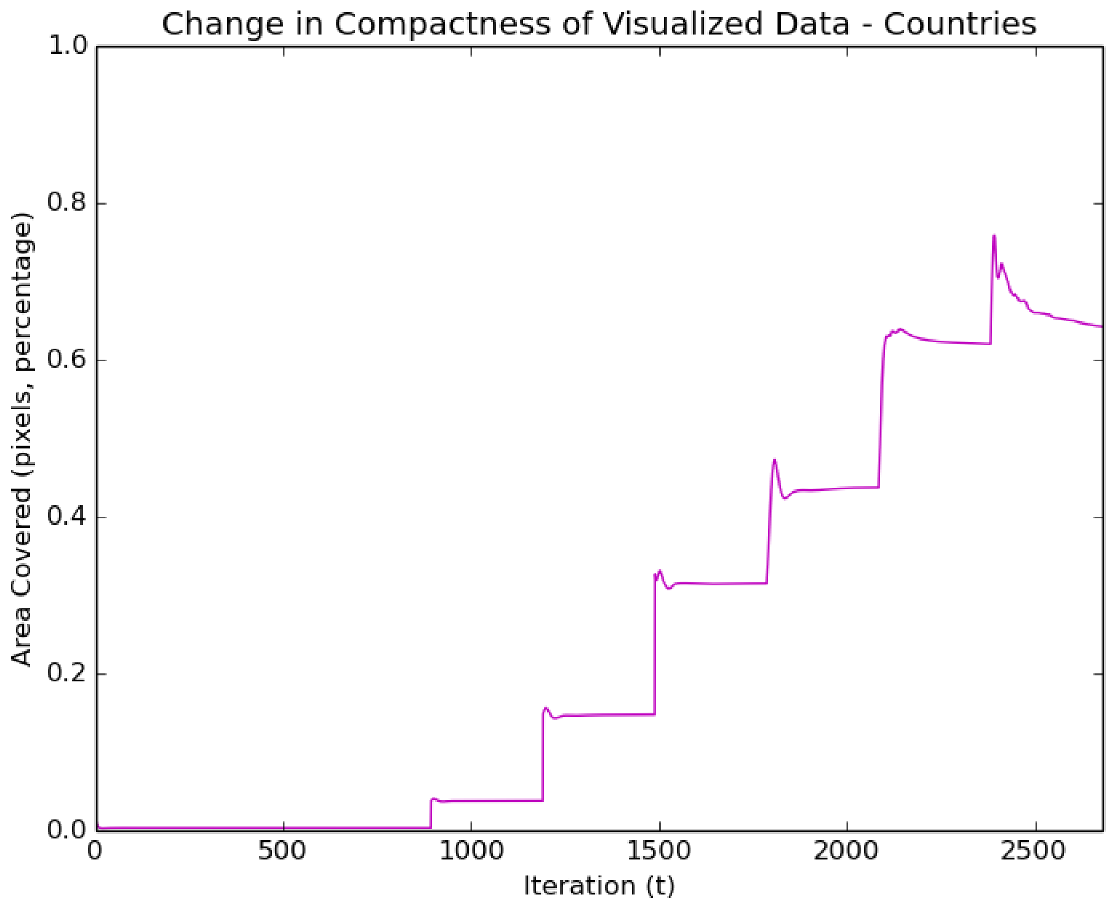


Figure 4.4: Country Compactness

4.3 Resizing Quality Evaluation

In order to provide context for the discussion surrounding the quality of the resizing constraint, layouts will be provided for both indexing attributes used. The layout for *Genre* is shown in Figure 4.5. This visual representation is a point of reference for discussion regarding the general mapping of tuples from the source dataset to their respective nodes in the resultant layout and is provided as a point of reference for further discussion.

Presented in Figure 4.6 is the distribution of points in the generated layout and the database for the indexing attribute *Genre*. The green points are related to the area covered by a given node with respect to the entire layout. The blue data points show the percentage of tuples that have a given attribute value (what will become the contents

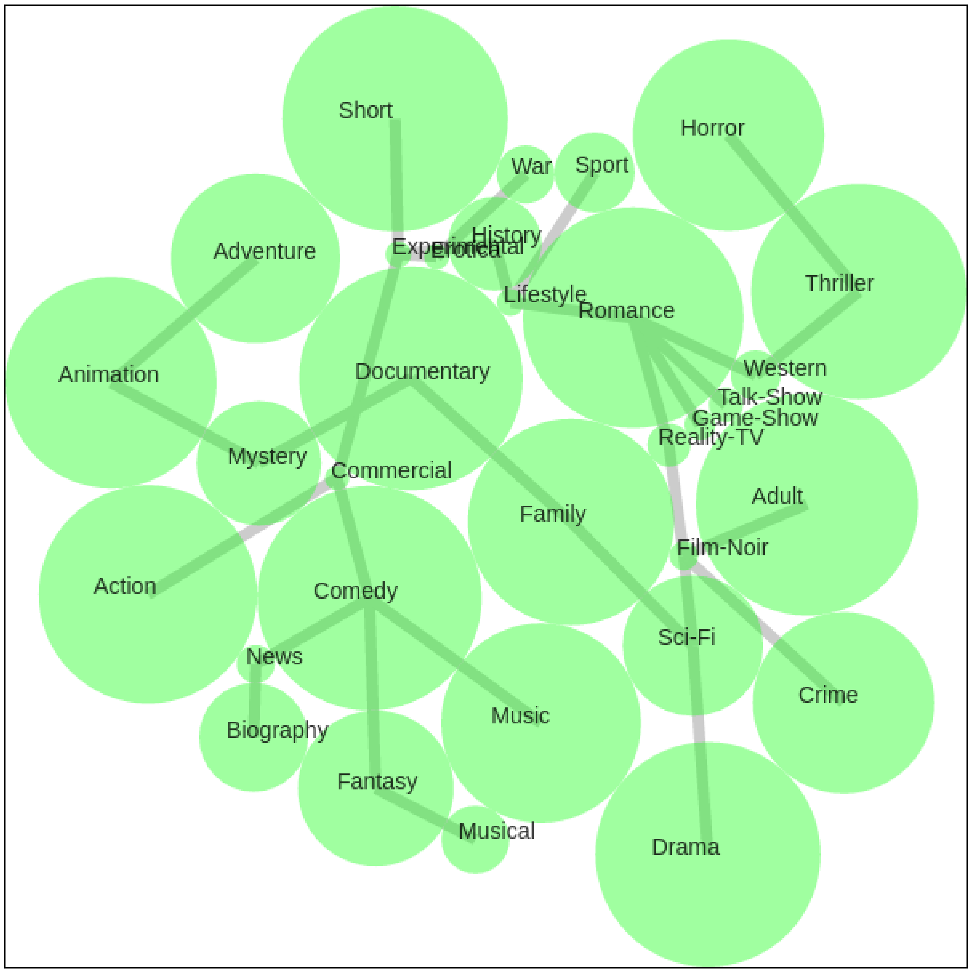


Figure 4.5: A layout of nodes for the Genre attribute.

of a node). This graph illustrates that the trend in tuple distribution in the database for an indexing attribute is maintained in direction and exaggerated in magnitude. This disparity occurs as a result of the percentile-based node scaling performed by the pipeline.

Pictured in Figure 4.7 is a layout of nodes indexed according to their country of production. This layout corresponds with the point distribution in Figure 4.8. Labels have been included for clarity of position for the larger nodes. This layout also demonstrates the factors that contribute to the degradation of quality of edge crossings.

Similar to Figure 4.6, the graph in Figure 4.8 shows the distribution of nodes according to the *Country* attribute. The red data points in this graph show the size of a given node

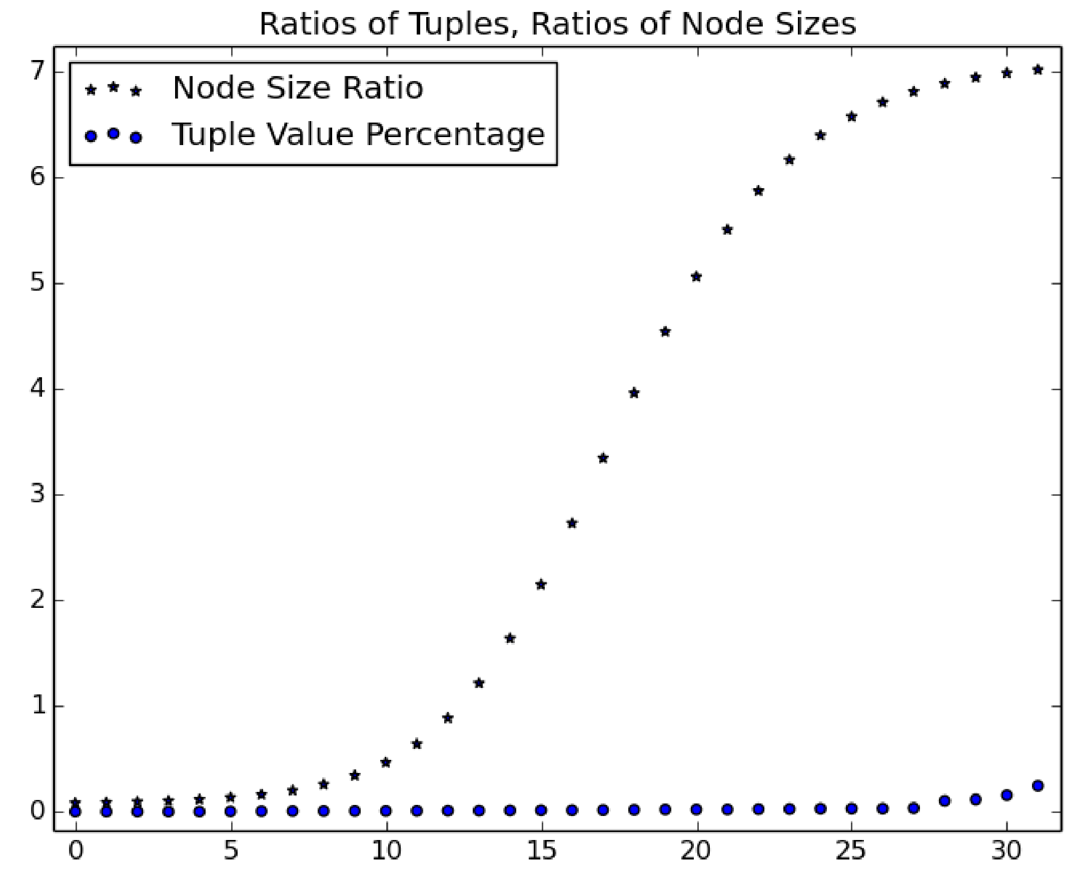


Figure 4.6: Genre Size Ratios

in a layout relative to all other nodes, as a percentage. Similarly to the case with Genre, the trend for individual nodes in the database is preserved in the layout.

One may wonder why there is a great disparity between the highest and lowest points in the displayed graphs. In order to maintain the ability to see and potentially interact with every node, the layout generation system incorporates a minimum size that, while quite small, allows the user of the layout to always see even nodes with extremely low membership. This is a safeguard in the case that particular nodes have a membership of far lower than one percent of the tuples. Similarly, the largest nodes reach a maximum size that ensures that they will always be visible within the layout, and does its best to ensure that all nodes are visible on any visualization using the layout.

In examining the figures corresponding to annealing over time, layout compacting

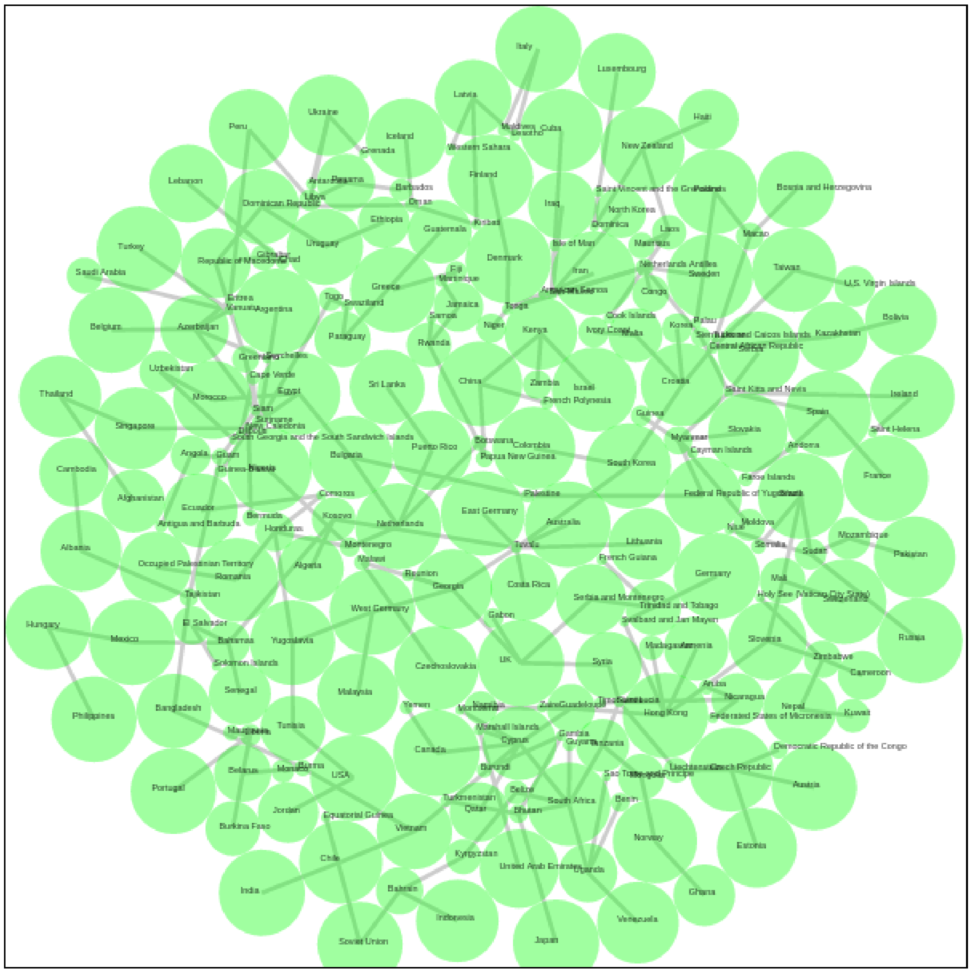


Figure 4.7: A layout of nodes for the Country attribute.

over time and tuple distribution, the impact of constraint ordering on constraint quality is present. Consider the trend of edge crossing resolution in the figures presented. The initial rate of edge crossing reduction is very high in the annealing stage, resolving that constraint first. With attributes that contain a higher number of edges, the edge overlap count increases during the final compacting step due to the factors previously discussed; the high number of small nodes and the more drastic repositioning of small node pairs linked together by the force directed simulation have a stronger impact in such an attribute. Ordering of constraints is relevant to the final degree of constraint resolution on a constraint-by-constraint basis; the constraints resolved earlier in the

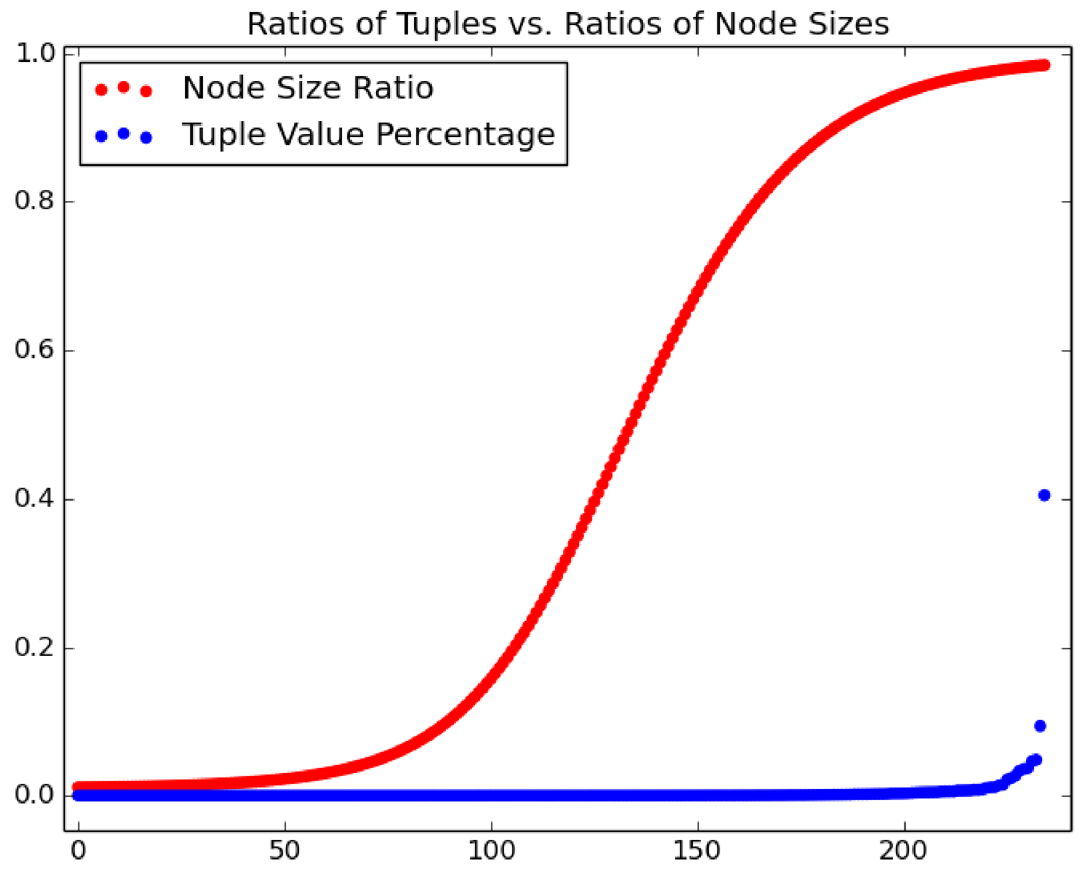


Figure 4.8: Country Size Ratios

pipeline experience a chance of quality due to future steps altering the layout. Using the presented cases as a baseline, the effect of quality degradation is correlated not only with constraint ordering but also with node count for a given attribute; a higher number of nodes in an attribute points to a more normally-distributed size of nodes in a given layout, producing more small nodes. Small nodes are more likely to be displaced in a step in the layout generation process which, depending on the constraint in question, can partially undo the quality improvement of a layout according to a constraint. Future experiments with different phases of a pipeline approach may seek to evaluate the optimal order for constraint resolution of a given set of constraints in order to maintain the highest average degree of stability in all constraints as layouts increase in node count, in order to determine whether a general pattern exists for different classes of constraints.

Chapter 5

Conclusion

The conclusion will be organized as follows. First, the summary of contributions made by the thesis will be presented. Following this, limitations of the system proposed to solve the layout generation process will be outlined. Lastly, possible future work related to the system, both in relation to overcoming limitations and in expanding the scope of the problem to be solved. Directions for further optimizations of the system, alongside possible problem spaces to explore, will be outlined. Next, the discussion of the thesis will be summarized. Contributions of the thesis will also be briefly summarized. Lastly, the thesis will be summarized.

5.1 Summary of Contribution

Within the thesis, the problem of creating layouts to visualize relational data on small screen devices was described. A method of layout generation of large volumes of relational data for small screen devices was presented and implemented. A system for performing meta-layout generation via sequencing of constraints, resolved by a multipass hyperparameter tuning of layout algorithms was created. This system provides a method for visualizing relational data using generated layouts. The system's multipass hyperparameter tuning and implementation were described with an illustrative walkthrough. The quality of the generated layouts with respect to each constraint was evaluated experimentally through defined metrics. Applications of the visual pipeline layout generation system were presented.

5.2 Limitations

While the proposed system does quickly produce layouts that efficiently organize similar nodes close to each other, the implementation and algorithm have some limitations that can be addressed going forward. Discussion of these unaddressed constraints will be divided into discussion regarding the layout resolution problem and concerns contained within the force-directed simulation's impact on the layouts generated.

Limitation Category I: **Layout Generation**. While the layout generation problem is solved using a force-directed simulation to arrange nodes of a tree, there are some restrictions. First, the pipeline approach to generating layouts is based upon an approximation of a set of similarities within an attribute. These similarities form the structure of the input maximum spanning tree, but beyond determining which nodes are connected, they do not contribute further to the generation of the layout (particularly as a system parameter). Another limiting factors involves the number of attributes being used. Many possibly interesting and possibly useful layouts are not producible as they combine, either directly (an IMDb Language layout corresponding equally to movie name and lead actor) or with a priority hierarchy (an IMDb layout corresponding first to organization by movie name, then by director) multiple keys for a given attribute. Such layouts can enable a wide variety of user-based applications, such as a recommendation layout that, given a particular tuple selection shows the most similar movies based on the aforementioned keys. Lastly, the generated layouts are specific to the type of dataset investigated within the thesis (the IMDb dataset), a dataset that does not contain an organic network which necessitates the inorganic inducing of a network. The ability of this approach to generate layouts for datasets with organic networks would need to be explored, to determine whether this approach is applicable.

Limitation Category II: **Force-directed Simulation**. With respect to the force-directed simulation used to drive the positioning of nodes in the final layout, there are some weaknesses. Chiefly, the parameters driving the simulation have not been experimentally evaluated to determine the optimal relative strengths of forces. The spring force is relatively weak compared to the initial values of gravity and charge, as a result the compacting force relies on a weak charge and a strong gravity to position the nodes instead of also incorporating spring force. It is possible that relying on spring force to facilitate compacting would result in a more compacted layout, but this has not been tested. Another significant limitation of the simulation is that it is time-based. Layout quality is, as shown in the experiments chapter, evaluated based on constraint resolution. The current implementation of the system steps through the pipeline process after a set number of iterations. The result is that a constraint's degree of resolution can only approach a bound, rather than meeting or approximating a threshold of resolution. The result is that, from attribute to attribute, the resultant layouts do not have an approximately consistent quality based on the previously-described factors, possibly leading to an expectation of quality being unmet. For two attributes both with node count N, layouts generated indexed on them are not guaranteed to achieve the same level of quality. One final limitation of the simulation is that it cannot be applied globally to every data structure - the layout node placement is exclusive to tree-based layouts. Based on the shape of the canvas and the shape of a given node, the fit of a layout may have very small, unremovable zones of unusable space between nodes.

5.3 Future Work

There are a number of directions that can the implemented system can be extended toward. These directions are informed by the limitations outlined in Section 5.2.

- *Determination of constraint sets for various platforms.* As an algorithm for automated layout generation, the algorithm can most likely be optimized in order to conform to different platform constraint sets. For example, extremely large displays may warrant a relaxation of the layout compactness constraint, which may warrant testing of appropriate whitespace presence. Constraint sets for different platforms can be determined, to produce a set of methods for meta-layout generation for each target platform.

- *Automatic attribute selection.* As an automatic meta-layout generation system, complete automation can be identified as a goal. To do this, implementation of an automatic attribute selection system to determine the indexing attribute(s) greatly strengthens the work, and provides additional possible applications to the system.

- *Identification of suitable and unsuitable attributes.* In implementing automatic attribute selection, criteria for identifying bad candidate attributes (attributes that are keys [fully unique], constant attributes) can be determined in order to automatically select good indexing attributes. These preferred indexing attribute would ideally tend to generate layouts that satisfy the set of constraints identified for a given platform, both based on experimental evaluation according to metrics and on visual inspection by users familiar with the data.

- *Investigation of suitability of other visualization algorithms for layout generation.* The D3.js layout library contains various additional layout generation methods, including chord, partition and circle-packing layouts. Additionally, different layout generation methods have been defined in graph layout literature. Implementation of and investigation into the suitability of these different layout generation methods can strengthen the system by enabling applicability to different domains. If suitable, incorporation of these additional layout generation algorithms can address limitations described in the force-directed layout generation.

- *Determination of optimal constraint-resolution sequencing.* As the previously-listed future works are implemented, the logical sequencing of constraint resolution stages in the pipeline being determined provides additional generality to the meta-layout generation. Different layout methods and different problem domains call for different sets of constraints. Investigation into a possible common pattern of constraint organization and resolution in order to minimize the quality degradation of earlier-resolved constraints (a problem that has shown itself to be present as node count increases) is a possible avenue of development. Results of this exploration would answer the question: are there classes of constraint resolution methods that, when specifically ordered, minimize the loss of quality for each constraint in the pipeline? An answer to this question could provide a general approach to implementing constraint resolution methods in particular order.

- *Crowdsourcing in applications for data management and traversal.* Crowdsourcing in applications like data cleaning and data curation provide a well-defined user role for the completely automatic system. Indexing attributes can be selected on a per-user basis, depending on the expertise of the user and their familiarity with the data. The resultant visualization of generated layouts may show the subset of the data relevant to a given expert to allow for interactive navigation of the data, for the purpose of curating or cleaning the data. Changes made to the data may trigger an automatic updated layout generation, providing a platform for continuous data cleaning of the dataset by expert users.

The defined future works would strengthen the system and generalize its applicability, both from the position of automatic graph layout, and from the position of an automatically updated visual interface as a tool.

Bibliography

[1] D3.js. `https://d3js.org/`.

[2] IMDbPy. `http://imdbpy.sourceforge.net/`.

[3] Amr Ahmed, Nino Shervashidze, Shravan Narayanamurthy, Vanja Josifovski, and Alexander J Smola. Distributed large-scale natural graph factorization. In *Proceedings of the 22nd international conference on World Wide Web*, pages 37–48. International World Wide Web Conferences Steering Committee, 2013.

[4] Alina Beygelzimer, Sham Kakade, and John Langford. Cover trees for nearest neighbor. In *Proceedings of the 23rd international conference on Machine learning*, pages 97–104. ACM, 2006.

[5] Stefano Boccaletti, Ginestra Bianconi, Regino Criado, Charo I Del Genio, Jesús Gómez-Gardeñes, Miguel Romance, Irene Sendiña-Nadal, Zhen Wang, and Massimiliano Zanin. The structure and dynamics of multilayer networks. *Physics Reports*, 544(1):1–122, 2014.

[6] Nicholas A Christakis and James H Fowler. Social network visualization in epidemiology. *Norsk epidemiologi= Norwegian journal of epidemiology*, 19(1):5, 2009.

[7] James Cook, Ilya Sutskever, Andriy Mnih, and Geoffrey E Hinton. Visualizing similarity data with a mixture of maps. In *International Conference on Artificial Intelligence and Statistics*, pages 67–74, 2007.

[8] Anton J Enright and Christos A Ouzounis. Biolayoutan automatic graph layout algorithm for similarity visualization. *Bioinformatics*, 17(9):853–854, 2001.

[9] Michael Ferron, Ken Q. Pu, and Jaroslaw Szlichta. ARC: A pipeline approach enabling large-scale graph visualization. In *2016 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining, ASONAM 2016, San Francisco, CA, USA, August 18-21, 2016*, pages 1397–1400, 2016.

[10] Thomas MJ Fruchterman and Edward M Reingold. Graph drawing by force-directed placement. *Software: Practice and experience*, 21(11):1129–1164, 1991.

[11] Hao He, Haixun Wang, Jun Yang, and Philip S Yu. BLINKS: ranked keyword searches on graphs. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 305–316. ACM, 2007.

[12] Jeffrey Heer and Danah Boyd. Vizster: Visualizing online social networks. In *IEEE Symposium on Information Visualization, 2005. INFOVIS 2005.*, pages 32–39. IEEE, 2005.

[13] Geoffrey E Hinton and Sam T Roweis. Stochastic neighbor embedding. In *Advances in neural information processing systems*, pages 833–840, 2002.

[14] Mathieu Jacomy, Tommaso Venturini, Sebastien Heymann, and Mathieu Bastian. Forceatlas2, a continuous graph layout algorithm for handy network visualization designed for the gephi software. *PloS one*, 9(6):e98679, 2014.

[15] Mehdi Kargar, Lukasz Golab, and Jaroslaw Szlichta. eGraphSearch: Effective keyword search in graphs. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management, CIKM 2016, Indianapolis, IN, USA, October 24-28, 2016*, pages 2461–2464, 2016.

[16] Yu-Jung Ko and Hsu-Chun Yen. Drawing clustered graphs using stress majorization and force-directed placements. In *Information Visualisation (IV), 2016 20th International Conference*, pages 69–74. IEEE, 2016.

[17] Robert Kosara, Helwig Hauser, and Donna L Gresh. An interaction view on information visualization. *State-of-the-Art Report. Proceedings of EUROGRAPHICS*, 2003.

[18] Joseph B Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society*, 7(1):48–50, 1956.

[19] Christopher McCarty, Jose Luis Molina, Claudia Aguilar, and Laura Rota. A comparison of social network mapping and personal network visualization. *Field Methods*, 19(2):145–162, 2007.

[20] Matej Novotny. Visually effective information visualization of large data. In *Proceedings of the 8th Central European Seminar on Computer Graphics (CESCG 2004)*, pages 41–48, 2004.

[21] Sam T Roweis and Lawrence K Saul. Nonlinear dimensionality reduction by locally linear embedding. *Science*, 290(5500):2323–2326, 2000.

[22] Georg Sander. Graph layout through the vcg tool. In *International Symposium on Graph Drawing*, pages 194–205. Springer, 1994.

[23] Manish Singh, Arnab Nandi, and HV Jagadish. Skimmer: rapid scrolling of relational query results. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 181–192. ACM, 2012.

[24] Magnus Sköld. *Social network visualization.* Skolan för datavetenskap och kommunikation, Kungliga Tekniska högskolan, 2008.

[25] Jian Tang, Jingzhou Liu, Ming Zhang, and Qiaozhu Mei. Visualizing Large-scale and High-dimensional data. In *Proceedings of the 25th International Conference on World Wide Web*, pages 287–297. International World Wide Web Conferences Steering Committee, 2016.

[26] Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. Line: Large-scale information network embedding. In *Proceedings of the 24th International Conference on World Wide Web*, pages 1067–1077. International World Wide Web Conferences Steering Committee, 2015.

[27] Laurens Van der Maaten and Geoffrey Hinton. Visualizing data using t-SNE. *Journal of Machine Learning Research*, 9(2579-2605):85, 2008.

[28] Shuicheng Yan, Dong Xu, Benyu Zhang, Hong-Jiang Zhang, Qiang Yang, and Stephen Lin. Graph embedding and extensions: a general framework for dimensionality reduction. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 29(1):40–51, 2007.