Master Thesis

Improving the Readability of the ChordLink Model

Vasil Alistarov

Date of Submission:	02. June 2025
Revised:	01. Dezember 2025
Advisors:	Prof. Dr. Alexander Wolff
	Tim Hegemann, M. Sc.

Julius-Maximilians-Universität Würzburg
Lehrstuhl für Informatik I
Algorithmen und Komplexität

Zusammenfassung

Viele Graphen aus der echten Welt, wie etwa (soziale, logistische, usw.) Netzwerke, erweisen oft eine lokal hohe, jedoch global niedrige Dichte. Während dies sinnvolles Clustern ermöglicht, führt es auch zu Schwierigkeiten, beispielsweise wenn eine homogene Zeichnung besagter Netzwerke erzielt wird. Genauer scheitern traditionelle Zeichenalgorithmen daran, zeitgleich die interne Struktur der Cluster greifbar darzustellen und eine Übersicht des Netzwerks als Ganzes zu liefern.

Als Lösung dieser Problematik sind hybride Zeichenalgorithmen entstanden. Diese machen von unterschiedlichen Zeichenansätzen für die Cluster und die grobere Struktur des Graphen Gebrauch und heben dadurch die relevantesten Merkmale von beiden hervor. Berühmtes Beispiel hierfür ist ChordLink ([ADM$^+$19]), welches Chord-Diagramme für die Visualisierung der Cluster verwendet. Auch wenn ChordLink optisch ansprechende Zeichnungen liefert, erweist es unserer Meinung nach manche Nachteile. Man betrachte beispielsweise die *Knotenverdopplung*. Dabei erzeugt der ChordLink-Algorithmus mehrere unintuitive Kopien eines Knotens am Rande des Chord-Diagramms des Clusters, in welchem der Knoten liegt. Kanten, die zu dem Knoten führen, dürfen dann in eine beliebige Kopie münden. Des Weiteren ist ChordLink als interaktives Programm entworfen und erfordert demnach Nutzereingaben, um eine Zeichnung zu liefern.

In dieser Arbeit versuchen wir, ChordLinks Nachteile zu mitigieren. Wir stellen einen ChordLink-ähnlichen Zeichenalgorithmus vor, welcher Cluster ebenso als Chord-Diagramme darstellt. Unser Algorithmus verwendet jedoch eine radiale Zeichnung, um den Graphen im großen Maßstab wiederzugeben. Wir wenden Wegfindungsalgorithmen an, um in der radialen Zeichnung nicht benachbarte Cluster zu verbinden. Insbesondere ist unser Algorithmus vollkommen autonom und braucht lediglich einen Graphen als Eingabe; weitere Nutzereingaben sind nicht erforderlich. Cluster werden automatisch deduziert und eine radiale Zeichnung wird neben sorgfältig angepassten Chord-Diagrammen geliefert, die die Relevanz der einzelnen Knoten hervorheben, ohne die Zeichnung des Clusters unnötig zu überladen. Am Ende schlagen wir Alternativen zu manchen Schritten vor, wie etwa die Platzierung von Labeln, um Knoten zu kennzeichnen.

Abstract

Many graphs encountered in the real world, such as networks (social, logistic, or otherwise), often tend to exhibit a locally dense but globally sparse structure. While this allows for meaningful clustering results, it also imposes difficulties when attempting to visualise said networks in a homogeneous manner. In particular, most conventional visualisation approaches struggle to properly convey the internal structure of the network's communities and provide an overview of the network as a whole at the same time.

To this end, hybrid visualisation approaches have been proposed: they utilise different visualisation techniques for the large-scale overview and the communities, highlighting the important features of both. A notable example is given by ChordLink([ADM$^+$19]) which uses chord diagrams to represent the communities. While ChordLink arguably provides more optically pleasing drawings, it still has, in our opinion, some drawbacks – take, for instance, the *node duplication* feature which, for the sake of highlighting the connections between communities, creates unintuitive copies of the same vertex on the boundary of the chord diagram and allows intra-community-edges incident to the vertex to connect to any copy. Moreover, ChordLink is designed as an interactive tool for network visualisation, thus requiring user input to perform.

In this work, we strive to improve upon ChordLink's drawbacks. We introduce a hybrid graph visualisation model that, similarly to ChordLink, represents communities as chord diagrams. However, our model utilises a radial drawing for the network on a large scale instead of a simple node-link-diagram. We apply edge routing algorithms to connect communities which are not incident on the radial drawing. Most importantly, our algorithm is fully autonomous, taking only a graph (e.g., as a set of edges or as an adjacency matrix) and requiring otherwise no user input. Communities are deduced automatically and a radial drawing is produced alongside carefully tweaked chord diagrams emphasising on the importance of each vertex without losing any details inside a community. Finally, we propose some alternative approaches for some steps, such as the placement of the node labels.

Contents

1 Introduction

The importance of graph visualisation has long been recognised as drawings are a strong tool for both exploration and analysis. Still, there are various challenges faced by visualisation tools and researchers designing these alike, for instance when it comes to drawing social and other real-world networks. Consider, for example, the students of some university, connected by the property "attends the same lecture". Aside from the potentially immense size of this kind of graphs, the largest issue there lies in their structure. Their density is non-homogenous, exhibiting high density on a local scale (such as students with the same major) while at the same time being globally sparse (students of different faculties do not tend to have many interactions). The fine-grained dense substructures can be referred to as *communities*.

As discussed by Shneiderman [Shn96], a drawing of this kind of graphs should empower the user to perform two tasks efficiently: identify and analyse communities and get a good overview of the connections between the communities. Alas, with a classical, homogenous layouting algorithm, usually only one of these tasks can be executed well at a time. This has given a rise to the so-called *hybrid* algorithms that, akin to the graphs they are primarily made to draw, have a heterogenous approach: they setup the layout of the communities in one way and the layout of the graph as a whole – in another. The two layout mechanisms do not have to share any similarities, as long as they assist the user in performing the two tasks listed above. This has lead to some graph visualisation techniques that, while fulfilling that purpose, produce rather visually unappealing drawings, such as (in our opinion) NodeTrix [HFM07], which represents the communities as a labelled adjacency matrix and the entire graph – as a node-link diagram connecting such matrices.

An improvement is given by Angori et al. who present ChordLink [ADM$^+$19]. There, the authors also prefer node-link diagrams for the large-scale overview of the graph. However, the communities are represented as circles, and the nodes are drawn as circular arcs on the boundary of their respective communities. Inside a community, the node arcs are connected via chords. While this design is visually pleasing and also well-suited for analysing networks with communities, there are still some details that caught our attention and upon which we seek to improve.

Our contribution in this work consists of a graph drawing pipeline primarily targeting small graphs with a couple of hundred vertices at most; see Fig. 1.1 for a brief overview. The expected input for the pipeline is an undirected (multi-)graph. The basic concept stems from the drawing style of ChordLink: clusters are visualised as circles with the vertices being represented as circular arcs on the boundary of the circle they are part of. These circular arcs are connected via chords – parabolic curves protruding towards the interior of the cluster – representing the edges of the input graph.

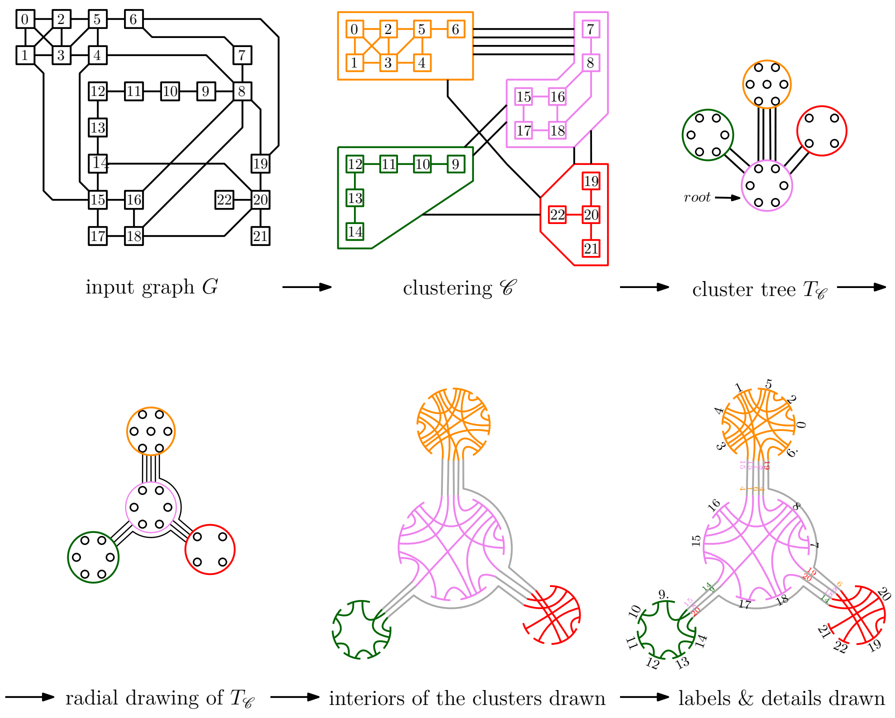

input graph G
clustering $\mathscr{C}$
cluster tree $T_{\mathscr{C}}$
root
radial drawing of $T_{\mathscr{C}}$
interiors of the clusters drawn
labels & details drawn

Fig. 1.1: Our pipeline with its main steps visualised.

However, our pipeline also exhibits some differences to ChordLink. To begin with, it is completely automated (save for some constants) and does not require any input from the user from beginning to end. It partitions the input graph into clusters, positions the clusters on the plane and connects them elegantly without cluttering the drawing with the edges between the clusters. Furthermore, unlike ChordLink, our pipeline does not perform the "node duplication" phase of ChordLink where, inside a cluster, a vertex connected to vertices of multiple other clusters is represented via *multiple* circular arcs. To avoid confusion, we assign exactly one arc to every node and route any edges between clusters through pre-defined, fixed points. The latter "dummy vertices" are also drawn in the form of circular arcs.

The rest of this work is structured as follows. In the subsequent Chapter 2, we define some terms and properties that would be used throughout the majority of our work. Then, we provide an overview of some existing drawing approaches, and hybrid visualisation algorithms in particular. Chapter 3 is the main part of this work. There, we describe step-by-step our pipeline, beginning with the input graph and building up to the final drawing. Each logical step is described in its own (sub)section. Finally, Chapter 4 summarises the model and provides an overview of some open questions or topics for future work.

2 Background

In this chapter, we provide information about various terms and concepts that are used in the remainder of this work. Furthermore, we investigate and provide an overview of previous works both in the areas of analysing networks and visualising graphs, particularly focusing on mixed drawing approaches. Many examples will stem from the category of social networks, as these have been the focus of both growing interest and, consequently, extensive research.

2.1 Concepts and Definitions

This work is concerned with the drawing of *graphs*. Given a graph G with vertices $V(G)$ (or simply V should the graph be unambiguously given via context) and edges $E(G)$ (or E) $\subseteq V \times V$, we define $n = |V|$ and $m = |E|$ as the number of *vertices* and *edges*, respectively. The neighbourhood of a vertex $v \in V$ is given by $\mathrm{adj}(v) = \{u \in V \mid \{u, v\} \in E\}$, and a vertex' degree is denoted as $\deg(v) = |\mathrm{adj}(v)|$.

A connected graph is a *multigraph* if $E(G)$ is a multiset, i.e., for a pair of vertices it may contain multiple edges between them. Note that unless stated otherwise, we will be working with simple graphs without such "multiedges" and will attempt to intercept and handle such cases beforehand. Additionally, the *degree* of a vertex refers to the number of edges incident to that vertex.

A graph is *connected* exactly when for each pair of vertices u, v there is a path $u = w_0, w_1, \ldots, w_n = v$ from u to v with $\{[w_i, w_{i+1}\} \in E$. If a graph is not connected, we refer to each separate part as *(connected) component*. A graph with $m = n - 1$ is called a *tree*. A tree may have a designated vertex known as the *root*.

A *clustering* $\mathscr{C}(G)$ is a partitioning of the graph's vertices into non-empty *clusters* $C_i \in \mathscr{C}$. Note that clusters containing a singular vertex are also allowed. We define $E(C_i) = \{\{u, v\} \in E(G) \mid u, v \in C_i\}$ as the *intra-cluster edges* of C_i. For a clustering $\mathscr{C}(G)$, we acquire a multigraph (in the general case) $G_{\mathscr{C}}$ by taking the clusters as vertices and adding an *inter-cluster edge* between two clusters C_i, C_j for every pair of vertices $u \in C_i, v \in C_j$ adjacent in G.

When working with real-world data, it is possible for it to contain additional information for some vertices or edges. We refer to such information as *attributes* and access it via $x.attribute$ for $x \in V$ or $x \in E$. Examples include vertex labels, edge weights, or even the direction of an edge. For simplicity, in our algorithms we assume that a graph containing the edge (u, v) also contains the edge (v, u), or, in common terms, is *undirected*. We discuss the applicability of our algorithm on directed graphs in Chapter 4.

Analogously to the concept of vertices and edges we define *nodes* and *links* as their respective counterparts on a drawing of a graph $\Gamma(G)$. Each node v has coordinates v_x and v_y on the plane.

2.2 Related Work

Rigorous work has already been done in the general field of visualising graphs. This is no wonder given the fact that humans tend to understand, memorise and analyse data (for instance, some logistic, electrical and social networks, a step-by-step description of a process, etc.) much better when said data is accompanied by some sort of visual representation [She67, CM84]. While the earliest graph drawing methods focused primarily, if not exclusively, on simple node-link diagrams designed per hand, more and more advanced and complex drawing techniques started to emerge once graphs became subject of mathematical study. Additionally, the emergence of computers gave rise to the idea of automatic generation of graph drawings, such as the work of Baecker[Bae67].

Still, a great number of drawing algorithms until the 21st century focused on force-generated node-link layouts using attraction and repulsion, akin to springs and electrically charged particles, respectively, as those simulate how a graph would behave "naturally", i.e. if it was a physical object. Some examples are given in [Ead84, KK89, FR91, FLM94, DH96]. While these approaches work acceptably well and are efficient enough on average, it should come to no surprise that in some cases they also exhibit various drawbacks in terms of readability. For instance, a study by Ghoniem et al. [GFC05] showed that on denser graphs with as few as 20 nodes, classic node-link diagrams make it difficult to perform some basic tasks such as finding a specific vertex or determining whether two vertices are adjacent. Many types of networks tend to be sparse on a global scale but rather dense locally. Therefore, designing layout algorithms tailored for these cases is clearly preferable. Examples include social (e.g. [WHPL10]), but also biological (e.g. [MMRR13]), information (e.g. [FLGC02]) networks and the like.

The local, dense parts of such graphs are often called *communities*. Given the interesting properties of networks with communities [WS98], those have been studied quite well; examples include the works of Horn et al. [HFB$^+$04] and Newman [New03]. Regarding their visualisation, different requirements have been proposed (compare, for instance, Shneiderman [Shn96] to Wasserman and Faust [WF94]). Still, it can be safely said that most, if not all, of those requirements or tasks can be reduced to the following points:

- Analysing the network on a large scale, thus identifying communities and their relations, including more "central" ones.
- Analysing the network on a small scale, i.e., getting insights into the internals of each community and how singular vertices are connected to each other.

Alas, achieving both tasks simultaneously is challenging as they tend to contradict in the case of a static layout: making the top-scale structure of the network easy to analyse by e.g. drawing communities close together means that the fine-grained structure will be more difficult to see, as its parts has been contracted so as to not obscure the overview of the top-scale structure. Similarly, putting an accent on the low-level structure such as singular vertices, vertices with high degree and their connections to each other, will often distort a graph's drawing in such a way that the large-scale structure – the communities – cannot be instantly recognised.

This, among other factors, like for instance scalability, has given rise to multi-layer and/or hybrid graph drawing approaches. The main idea of the former consists of recursively subdividing a (potentially large) graph into a set of smaller ones, and continuing doing so until the resulting graphs are small enough to be visualised in a clean and easily comprehensible manner. The subdivisions themselves are often drawn in a tree-like manner. However, one feature of the drawings of most multi-layered approaches is that they imply some sort of hierarchical structure in the input graph, defined via the order of subdivisions; in truth, this is a by-product of the exact method that is used for the subdivision, and often different partitions can be achieved by using different subdivision criteria. For example, Harel and Koren [HK00] subdivide the graph based on an estimation of the *k-centres* problem. The FM^3 algorithm [Hac05], which is not explicitly optimised for a specific use-case but works on general graphs instead, even has a proven worst-case runtime of $O(n \log n + m)$ by showing that a fixed number of nodes and edges is present in each subdivision. Worth mentioning is also the approach of Six and Tolis [ST99], which we will also refer to later in this work. It is unique in that it allows one to manually specify a custom size for the drawing of each community instead of automatically making subdivisions smaller; to the best of our knowledge, no other work has these properties. Newer works in the area also exist; consider, for instance, the radial algorithm PLANET [HLT$^+$20] which uses a list of angle assignment rules to distribute nodes evenly and minimise the number of edge crossings.

Hybrid drawing approaches have also begun to emerge, often for the purpose of drawing globally sparse but locally dense graphs. In particular, the main idea is to utilise a different drawing paradigm for the large-scale network and the small-scale communities, which are to some extent akin to clusters, thus waiving homogeneity but highlighting the features of both. Given that the great majority of graph visualisation approaches use node-link diagrams, it should come to no surprise that adjacency matrices, as the other "classic" graph representation, were the first to be included in hybrid approaches; take, for instance, Matrixexplorer, developed by Henry and Fekete [HF06], which provides both representations side-by-side, synchronised. This allows one to refer to either visualisation depending on the task at hand – as shown by Ghoniem et al. [GFC05], the node-link visualisation is better suited for path-related queries, while the matrix representation eases for instance the check whether two vertices are adjacent. The creators of Matrixexplorer would improve upon their work by first adding edges on the sides of the matrix, thus creating MatLink [HF07], and later introducing NodeTrix [HFM07], which is – we believe – currently one of the best-known hybrid visualisation tools. NodeTrix first represents the input graph as a node-link diagram and, on user input, groups the selected nodes into an adjacency matrix. Links incident to nodes encompassed by the matrix now connect the corresponding matrix row with its neighbour nodes (which can also be the rows of another matrix) instead. An example is given in Fig. 2.1.

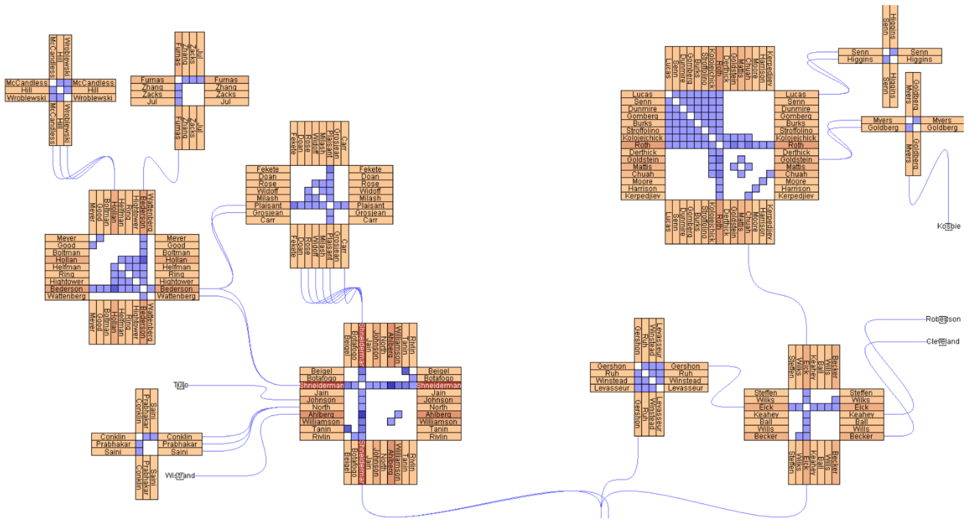
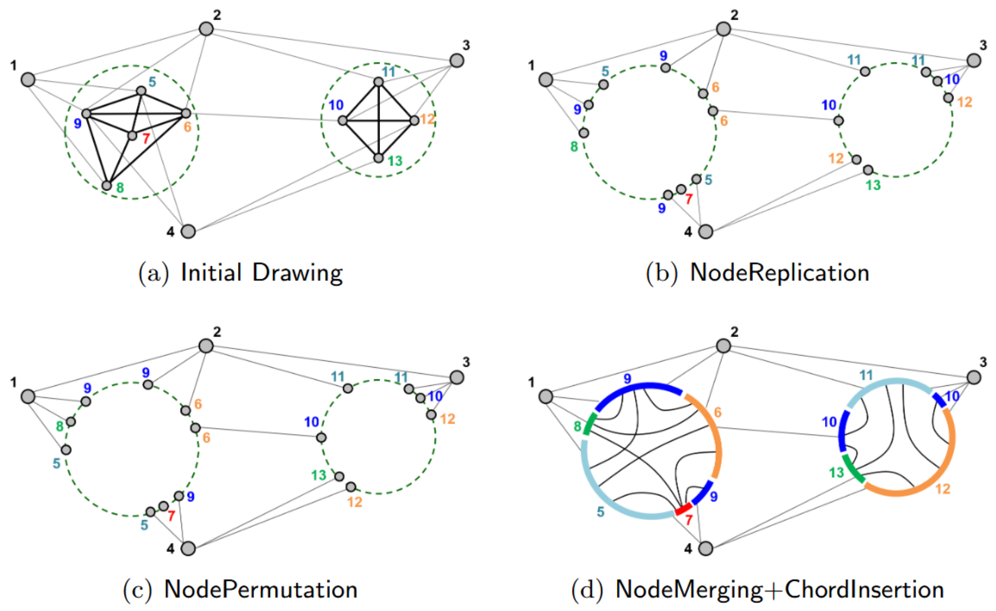


Fig. 2.1: An example of the output of NodeTrix, as given by Henry et al. [HFM07].

NodeTrix has not only sparked a vast array of papers presenting visualisation techniques based on it, but has also inspired the creation of new hybrid drawing algorithms. The most notable of those, and the main focus of this work, is ChordLink [ADM+19, ADM+22]. Designed by Angori et al., ChordLink opts to use not adjacency matrices for the representation of the communities, but rather circular chord diagrams. In each chord diagram, a node is represented by one or many circular arcs. In particular, the ChordLink algorithm creates an arc copy for each neighbour of a node that is not in the community. A subsequent step reorders those arc copies that share the same "external" neighbour in order to minimise the amount of copies of the same node that are not consecutive on the community's boundary. For instance, as seen on Fig. 2.2, the positions of the copies of nodes 5 and 9 that are adjacent to node 1 are swapped, thus yielding two consecutive copies of node 9. This is logically followed by a step where consequent arc copies of the same node are merged. Finally, within the community, a chord for each edge is placed. The authors employ a heuristic that minimises the number of edge crossings created this way, so as to decide between which copies of the two nodes should an edge chord be drawn. Note that ChordLink, similarly to NodeTrix, is an interactive tool: it does not compute any meaningful communities by itself, but instead relies on user input to group, ungroup and move communities around. Subsequent works revolving around ChordLink are also emerging. The authors of the original paper conducted an evaluation [ADM+22] of ChordLink based on some real-world networks and discussed in detail the algorithms used throughout the ChordLink toolchain. Kindermann et al. [KSW23] proved, among other points, the $\mathcal{NP}$-complexity of some of the algorithmic challenges of ChordLink. Finally, in this work we will build a hybrid visualisation toolchain for globally sparse but locally dense graphs that is to some extent inspired by the procedures of ChordLink.


(a) Initial Drawing

(b) NodeReplication

(c) NodePermutation

(d) NodeMerging+ChordInsertion

Fig. 2.2: An example of the steps of the ChordLink algorithm, as given by Angori et al. [ADM+19].

Finally, we mention the existence of hybrid graph drawers that use tree maps as a second component, next to node-link diagrams. Examples are given by the works of Fekete et al. [FWD+03], who extract a tree map structure from an input graph and subsequently add the remaining edges, and Zhao et al. [ZMC05] who construct a hierarchy of tree maps and connect them in a node-link diagram-like manner. A study on various hybrid drawing techniques is given by Giacomo [GDMT21]. Also, aside from Matrixexplorer, NodeTrix and ChordLink, many researchers and industries have opted for designing interactive tools which, while initially providing a large-scale overview of the network, allow the user to select and switch between different communities for closer inspection. Further examples for such tools are Treeplus [LPP+06] and Vizster [Hdb05].

3 The Model

We aim to design a hybrid visualisation toolchain focusing primarily on locally dense, but globally sparse networks. We focus on small graphs with a couple of hundred vertices at most – unlike most existing drawers that aim for scalability in order to be able to draw large networks with tens of thousands of vertices. This helps relax some requirements in terms of performance while also filling a niche and providing a framework for a more day-to-day, commonplace use where e.g. a researcher may want a quick overview of some small graph or network. Finally, to provide out-of-the-box functionality, we attempt to design our framework as automated as possible; user input in a few selected places is possible, but not required.

The framework can be described on a high level as follows:

1. Clustering the input graph G into $\mathscr{C}(G)$: Using only the graph's structure and the vertices' properties, we partition the graph into clusters for a high-level drawing.

2. Extracting a spanning tree $T_{\mathscr{C}}$ from $G_{\mathscr{C}}$: We compute a maximum spanning tree so that there are as few edges as possible that do not connect two adjacent clusters on the tree. Then, we define a root for the tree.

3. Routing the inter-cluster edges along the edges of $T_{\mathscr{C}}$: We bundle the edges entering/exiting a cluster. If the two clusters incident to an edge are not neighbours on $T_{\mathscr{C}}$, we route the edge in its bundle along the tree's structure, and around the boundary of other clusters until the target cluster is reached.

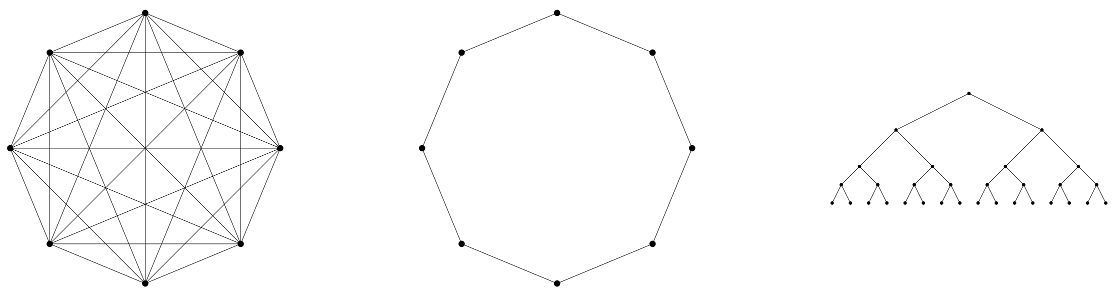
4. Drawing $T_{\mathscr{C}}$ as a radial tree on a large scale: We calculate space to dedicate to each subtree, then place the clusters at their initial coordinates on the plane.

5. Drawing each cluster in a ChordLink-like manner: We use circular arc nodes for the vertices and chord links for the intra-cluster edges. We also introduce *gate nodes* to connect clusters to one another.

6. Optionally placing vertex labels next to the respective node arcs, if the input graph contains such labels.

This exhibits several differences – and, in our opinion, improvements – as compared to the classic ChordLink. First, the drawn graph is structured (as a tree), whereas ChordLink places the drawing freely in the plane. Second, within a cluster, while the nodes are represented as circular arcs as is the case with ChordLink, our model completely eliminates the node duplication feature; there is exactly one arc per node[1].

[1]as well as additional arcs connecting the clusters; this will be elaborated further in Subsection 3.3.4.



(a) Complete graph with 8 vertices.

(b) Complete cycle with 8 nodes.

(c) Complete binary tree with 5 levels.

Fig. 3.1: Examples of graphs that do not exhibit any unambiguous clusters. The graphs in (a) and (b) cannot be clearly subdivided at all, while the graph in (c) can be subdivided in multiple ways that appear equally valid.

Finally, our algorithm is automated, requiring no user input from beginning to end (although one could argue that whether this is an improvement depends on the use-case).

In the following, we will go into detail about each step.

3.1 Clustering

A *clustering* of a set of elements consists of partitioning that set into some number of non-empty groups C_i. The members of each group are then expected to be similar in some way. In particular, clustering algorithms usually make use of a predefined *similarity (or distance) measure* between two points, $d(u, v)$ or, alternatively, of some measure of the "integrity" of a cluster. Clearly the quality of a clustering algorithm's result is heavily dependent on the chosen measure.

We emphasize that in general it is crucial to differentiate between clustering approaches that take a graph (as defined earlier) as an input on the one hand, and general data point clustering algorithms which do not concern themselves with the graph's structure, but work merely on n-dimensional data points on the other. Examples for the latter include the well-known k-means, DBSCAN [EKSX96] etc. Still, even in the former case, not all graphs admit a "meaningful" partition of their vertices into clusters, regardless of the chosen algorithm. Take, for instance, most trees, all complete graphs or even a circle, that is, a set of vertices, each with degree 2, connected in a chain (as seen on Fig. 3.1). Nevertheless, every clustering algorithm will yield a result for any input graph (such as a single cluster consisting of the entire graph); this result may simply be of no use for further reasoning. Visualisation often helps in reasoning whether a clustering algorithm has yielded a meaningful result for a given graph.

Similarly, many tools that cluster graphs either as a result or as an intermediate step rely on user input for doing so; examples include NodeTrix [HFM07] and ChordLink [ADM$^+$19]. This is convenient for experimentation and similar goals, but assumes that the user already has vague knowledge of what they aim for. As opposed to that, we wish to deliver an autonomous framework that produces a visually appealing drawing of a graph whilst requiring little to no input from the user.

To achieve this design, we will begin by examining the already existing graph clustering algorithms. Some of those operate based on algebraic concepts, rather than graph-theoretical ones. For instance, Markov Clustering [VD00] emulates random walks within the graph by modifying a transition probability matrix; intuitively, such a random walk is less probable to leave a cluster than to stay inside of it. Once a reoccurring state is reached, the modifications are halted. The final result, that is, the clusters, can be extrapolated by looking at the connected components of the graph induced by the matrix. Spectral Clustering [NJW01] uses the eigenvectors of the graph's Laplacian to essentially reduce the graph's vertices to datapoints in such a way that the edges are related to the distance between the datapoints. Afterwards, a clustering approach from the area of data analysis can be applied, such as the aforementioned k-means. Another useful concept is that of conductance – a measure of the quality of a graph partition, or, equivalently, of a cut through a graph. Intuitively, for any cut through a graph with high conductance there will be many edges connecting the two partitions. Iterative Conductance Clustering [KVV00], or ICC, uses this measure to (sub-)partition a graph recursively. However, ICC relies on an input parameter in order to recognise when to stop dividing the graph further; thus, it is not applicable for our use-case. Moreover, computing a cut with minimum conductance is $\mathcal{NP}$-hard [vS06] and so heuristics need to be used instead.

Hierarchical clustering algorithms like ICC, produce a structure where each cluster is comprised of further subdivisions, sometimes all the way down to clusters consisting of a single vertex. This hierarchy has a natural representation as an (often binary) tree rooted at the entire input graph; an edge directed away from the root indicates a further subdivision of the parent cluster. Such a tree is known as a *dendrogram*. Hierarchical clustering can be achieved in two ways:

- bottom-up, or *agglomerative* – starting with small clusters usually consisting of a single vertex, each step of those algorithms consists of either merging two clusters or adding a single vertex to a larger cluster. At the end, the entire input graph is reconstructed. To the best of our knowledge, not many algorithms in this category exist; as an example, we mention the work of Hopcroft et al. [HKKS03].
- top-down, or *divisive* – beginning with the input graph, a cut optimising some efficiently computable "connectivity measure" is found and then used to split the subdivision at hand. Aside from the selection of that connectivity measure, another important question for divisive algorithms is when to stop partitioning (if at all).

Various criteria can be used for determining the next partition in divisive clustering. As mentioned, conductance is an example for one. Another popular alternative consists in splitting the graph along its minimum cut, for instance with Karger's algorithm [Kar93]; equivalently, the maximum flow can be computed instead. Hartuv and Shamir [HS00] propose an approach based on this method; their criterium to stop dividing the graph is if its edge connectivity is over $\frac{n}{2}$. Well-known is also the betweenness centrality metric introduced by Girvan and Newman [NG04] based on the node betweenness given by Freeman [Fre77]. In essence, the betweenness centrality of an edge $\{u, v\}$ is the number of shortest paths between any two vertices of the graph that pass through that edge. While calculating this measure may seem computationally intensive, Brandes [Bra01] shows that it can be done in $O(n + m)$ time. In particular, Girvan and Newman's approach is to remove, step-by-step, the edge with the currently highest betweenness centrality. Note that unlike the minimum cut criterion, not every step will split the graph in this case.

3.1.1 Clustering with Modularity

In the aforementioned work, Girvan and Newman also introduce the notion of *modularity* in order to evaluate the quality of a division of a graph into clusters. Intuitively, the modularity of a clustering is a comparison between the number of intra-cluster-edges of the given graph and the expected number if the graph's edges were distributed uniformly. A positive modularity indicates a good division into clusters. An important feature of the modularity approach is given by the fact that a modularity-based algorithm needs not know the number or sizes of clusters beforehand. It can even automatically stop subdividing the graph if that would bring no improvements to the modularity at the moment.

While Newman [New06] shows that modularity can be optimised efficiently using some algebraic methods and heuristics, deciding on the existence of a clustering with a specific minimum modularity is $\mathcal{NP}$-hard [BDG+07]. As most approaches to find clusters with optimal modularity are computationally expensive, we decide against using modularity exclusively. Still, computing the modularity value of an already given clustering is easy. Therefore we opt in favour of applying modularity as a stop criterion when performing the clustering. As for the algorithm, we pick divisive clustering via betweenness centrality, as modularity and betweenness centrality are somewhat related.

Newman [New06] gives the following expression for the modularity of a division of a graph in two communities[2]:

$$Q = \frac{1}{4m} \sum_{i,j \in V} \left(A_{ij} - \frac{\deg(i)\deg(j)}{2m} \right) s_i s_j \tag{3.1}$$

where A is the graph's adjacency matrix and s_i is 1 or -1, depending on which community vertex i is placed in. However, we require a formulation applicable to more than two clusters. While Newman [New06] provides a formula for the additional contribution ΔQ upon further dividing a cluster, that formula is more suited for algebraic approaches such as spectral clustering that perform divisive clustering entirely via modularity. Instead, we are simply using this metric to evaluate an already given subdivision. As the s_i parameter is the only one to rely on the number of clusters, it needs a replacement. To retain its feature of distinguishing whether two vertices are part of the same cluster or not, we define

[2]note that the leading factor of $\frac{1}{4m}$ is only given for compatibility with [NG04].

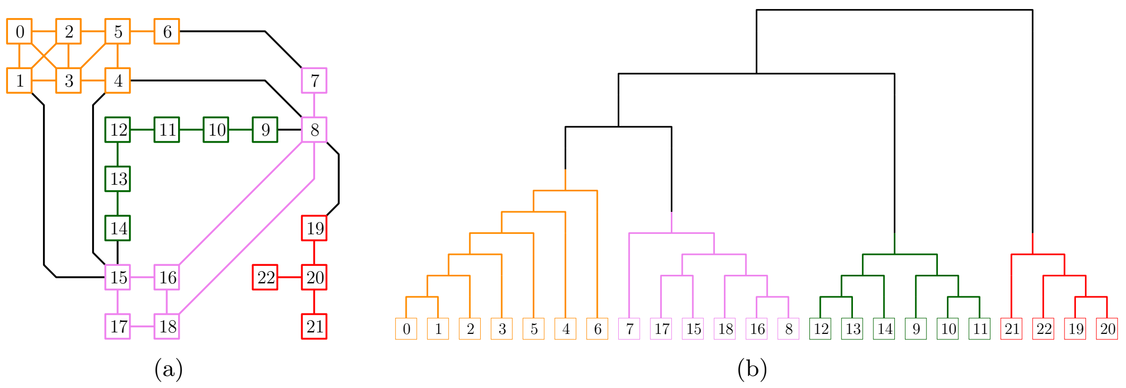


Fig. 3.2: An exemplary graph (a), and how it was clustered by our algorithm (b).

$$s_{i,j} = \begin{cases} 1 & \text{if } i \text{ and } j \text{ are in the same cluster} \\ -1 & \text{otherwise} \end{cases} \tag{3.2}$$

So in total, our approach looks as follows, with an example given in Fig. 3.2:

1. Calculate a (divisive) hierarchical clustering based on betweenness centrality.
2. Begin iterating the hierarchy (or dendrogram) from the root. As long as the modularity increases, accept a split and recurse on the two subdivisions.
3. Once there is no modularity increase on any path, take the clusters on the lowest levels reached.

Of course, that can somewhat be optimised: instead of computing the entire dendrogram, one could instead stop subdividing as soon as the modularity stops increasing. However, as we are dealing with small graphs, we expect no significant improvement from this.

3.2 Spanning Tree

Previously, we constructed a clustering $\mathscr{C}(G)$ for the given input graph G. In this brief step, we aim to derive a rooted tree whose structure will serve as a base for the residual drawing.

First, we note that since we ultimately wish to deliver a drawing for G, we will need to work with the multigraph of $\mathscr{C}(G)$, $G_{\mathscr{C}}$. Recall that this graph contains, for a pair of clusters C_i and C_j, a total of $|\{\{u,v\} \mid u \in C_i \wedge v \in C_j \wedge \{u,v\} \in E\}|$ many edges between C_i and C_j. Drawing $\mathscr{C}(G)$ as a tree means that many inter-cluster-edges – in fact, all those that connect two non-adjacent cluster vertices of the tree – will need to be routed in some way alongside the tree structure.

It is thus intuitive to try and minimise the number of such edges. We observe that this is equivalent to *maximising* the number of inter-cluster-edges that are also edges in the tree. From this follows trivially that the problem at hand can be solved via converting the multigraph of $\mathscr{C}(G)$ to a weighted graph with $w(C_i, C_j)$ equal to the number of edges between C_i and C_j, and then computing the maximum spanning tree of the resulting weighted simple graph.

Finding a maximum spanning tree, in turn, is also simple to achieve given the various algorithms that calculate a *minimum* spanning tree (MST). Indeed, for any weighted graph it is sufficient to negate the weights of the edges and execute a MST algorithm. After the negation, the previously heaviest edges will have the least weight and so will be prioritised by the MST algorithm. While algorithms for finding minimum spanning trees in as fast as $O(m)$ time are known to exist, such as the one of Pettie and Ramachandran [PR00], they are also rather elaborate and use complex data structures. Recall that the target graphs of our framework are small with a couple of hundred vertices at most. Therefore, we decide that taking a "standard" MST algorithm is sufficient. Due to the ease of implementation, we decide in favour of Kruskal's algorithm [Kru56] known to run in $O(m \log n)$ time [CLRS22].

In essence, the algorithm "grows" a forest of trees of minimal weight by iteratively taking light edges, as long as this would not cause a cycle. The other well-known algorithm we considered, the one of Prim [Pri57], has the same asymptotic running time and delivers the same result, but achieves it via computing shortest paths instead. As the implementation of Prim's algorithm is arguably more vexing and it provides no benefit as compared to Kruskal's algorithm, we opt against it. Note that instead of negating the edge weights, it is equally possible to modify e.g. Kruskal's algorithm to pick heavy edges first. However, with the weight negation, our pipeline remains flexible and able to accept any MST algorithm if different needs or use-cases arise.

3.2.1 Rooting the Tree

Finally, we wish to designate some cluster vertex as the root of the produced maximum spanning tree; this cluster vertex will be later placed in the middle of the final drawing. As previously mentioned, we explicitly try to compute a drawing which does not imply any hierarchical properties; therefore, in that regard, which cluster we choose as the root is irrelevant.

Still, the choice will ultimately have influence on the positions of the other clusters as well as the overall aesthetic of the final drawing. Roughly speaking, we want to choose a cluster that is somewhat "central" in the structure of the spanning tree, so that the drawing appears balanced. Intuitively, two possible metrics come to mind:

- a *centre* of the tree [HCH81] – one (or two) vertices that minimise the maximum distance to the rest of the vertices (i.e., the tree's *diameter*). It can be computed easily in $O(n + m)$ time by two graph searches: the first starting at an arbitrary node and resulting in one end of the diameter, and the second starting at the vertex that was just found and ending in the one that is farthest from it. Then, one (of the at most two) vertices in the middle is picked.

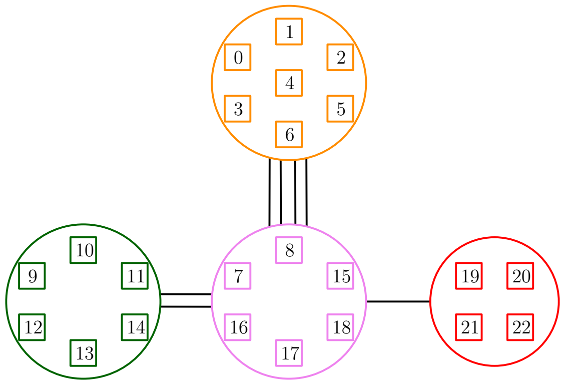


Fig. 3.3: The graph from Fig. 3.2 (a), reduced to the clustering tree $T_{\mathscr{C}}$.

- a *centroid* of the tree [Jor69] – a vertex that, if chosen as a root, will contain no more than half of the tree's vertices in each of its subtree children.

We believe that choosing the centroid will produce better results in our case – if the size of the root's subtree children is somewhat balanced, then the canvas space around the root will also be distributed and utilised in a more balanced manner, potentially also allowing for more compact drawings. To conclude this section, we provide an example of the produced tree on Fig. 3.3, as well as a simple algorithm for computing the centroid:

Algorithm 1: Our algorithm for computing the centroid of a tree.

```
fn ComputeCentroid($tree$)
    $start \leftarrow$ random vertex of $tree$
    return DFS($start, \varnothing$)
// Returns the centroid if it is in $u$'s subtree, and null otherwise
fn DFS($u, parent$)
    $u.size \leftarrow 1$
    for each $v \in \mathrm{adj}(u) \setminus parent$ do
        if DFS($v$, $parent$: $\{u\}$) is not null then
            return the found value
        $u.size \leftarrow u.size + v.size$
    $parentSize \leftarrow n - u.size$
    if all child sizes and $parentSize$ are at most $\frac{n}{2}$ then
        return $u$
    else
        return null
```

3.3 Drawing the Tree and Clusters

Having computed the cluster graph's spanning tree $T_{\mathscr{C}}$, the logical next step is to draw it. This task presented us with a bigger challenge than initially expected due to a variety of reasons. Indeed, as we will show later on (see Subsection 3.3.5), it is difficult to separate the large- and the small-scale drawing steps completely.

Before we proceed with the description of our approach, we summarize the main goals which guided us through our decision-making process.

- The large-scale drawing should *not* suggest any kind of hierarchy in the graph.
- The large-scale drawing should take different cluster sizes into consideration.
- In the cluster drawings, each vertex should be represented as a *single* circular arc.
- The size of those circular arcs should be relative to the "relevance" of the vertex, defined via some measure.
- At least within the clusters, the number of edge crossings should be minimised, as long as this does not impose an overwhelming computational overhead.

We will often use the term *cluster size* in this section, so we feel the need to elaborate on it in advance. In order to produce an uniform drawing in regard to the nodes, a node from cluster C_i which is as "relevant" as a node from cluster C_j must be drawn just as large. As we draw nodes as circular arcs on the boundary of the cluster they are part of, the size of the cluster's drawing directly depends on the sizes of the singular nodes. In particular, this influences the circumference of the cluster's drawing (and thus the circle's radius). This, in combination with some edges drawn directly around the cluster, is also what we will refer to as *cluster size*. We will quantify this later in Subsection 3.3.4.

The rest of this section is structured in a topological way: since many values depend on each other, we describe the derivation of these values so that at any given point all requirements for the step at hand have been fulfilled in previous steps. Though this order may appear counter-intuitive (as compared to, for instance, describing the drawing from large grained to small grained), we consider it important; otherwise, the separate steps would mention values that will be defined at some point in the future.

3.3.1 Routing the Inter-Cluster Edges

We begin with an unexpected part of our pipeline: The setup and positioning of the inter-cluster edges. This depends merely on the tree $T_{\mathscr{C}}$ which was constructed during the previous section. We will see shortly why it is important for this step to occur before drawing the interiors of the clusters. Note that this does *not* include the drawing of the edges as this requires the setup of the clusters, which will be performed in Subsection 3.3.4. Here, we are merely setting up the intermediate path each edge will take.

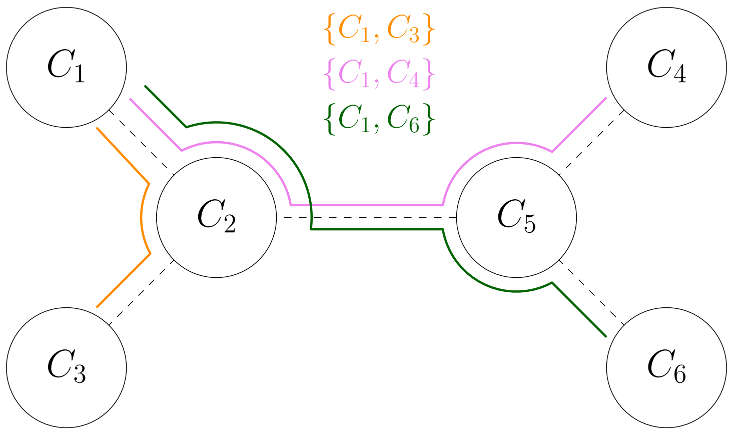


Fig. 3.4: Example for a cluster tree $T_{\mathscr{C}}$ with 6 clusters. Three edges are selected and displayed akin to the way they would be drawn with our pipeline.

Our general concept is as follows: the links corresponding to the inter-cluster edges connect to the clusters on specific locations only. Also, these links may not go directly from one cluster to another, should the two clusters not be adjacent in $T_{\mathscr{C}}$. Instead, bundles of inter-cluster links run alongside the edges of the spanning tree. If a link between clusters C_1 and C_2 would go through another cluster, C_3, it is instead routed alongside the boundary of C_3, as exemplified on Fig. 3.4. The approach we introduce in this section will see to it that the number of crossings within those link bundles is kept small, and that links do not take longer detours around clusters than necessary.

We begin by introducing *gate nodes* – additional nodes that each cluster contains and that serve as entry and exit locations for inter-cluster links. In particular, given the cluster tree $T_{\mathscr{C}}$, each cluster $C_i \in V(T_{\mathscr{C}})$ receives $\deg_{T_{\mathscr{C}}}(C_i)$ many gate nodes – one for each of its neighbour clusters. For uniqueness, we denote a gate node of cluster C_i "targeting" cluster C_j as $g_{i,j}$. A link $\{u, v\}$ such that $u \in C_i$ and $v \in C_j$ is now represented as a sequence of links $\{u, g_{i,j}\}, \{g_{i,j}, g_{j,i}\}, \{g_{j,i}, v\}$. Subsection 3.3.4 will handle the links of type $\{u, g_{i,j}\}$; this subsection's subject are the links of type $\{g_{i,j}, g_{j,i}\}$ only.

The problem of routing links to avoid node interiors has been well studied (see, e.g., Dwyer and Nachmanson [DN09]). Often, it is combined with link bundling – grouping links together as they traverse the same area simultaneously – and the problem of ordering the links within these bundles to minimise crossings as links enter or exit the bundle. Classic examples for general graphs are the works of Pupyrev et al. [PNBH11] and Holten et al. [HvW09]. A newer algorithm is given by Hegemann and Wolff [HW23] which build upon the work of Pupyrev et al. [PNBH11]. If the links are to traverse the structure of an embedded graph, this becomes an instance of the metro line crossing minimisation problem (MLCM) introduced by Benkert et al. [BNUW06].

A usual approach begins with the construction of an auxiliary *routing graph*. It defines the allowed routes on the plane for the links to run alongside of in order to connect the nodes. The links are then routed on top of the edges of the auxiliary graph, with one

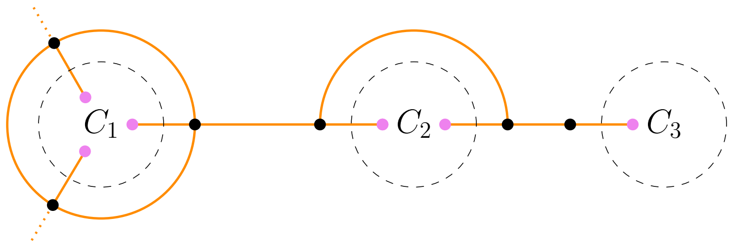


Fig. 3.5: Part of a routing graph. The purple vertices are terminal vertices corresponding to the cluster they are placed in. The black vertices are internal vertices.

link often covering multiple of its edges. Links routed on the same edge form a bundle. We define our routing graph H as follows.

$V(H)$ contains one *terminal vertex*, $term(g_{i,j})$, and one *internal vertex*, $int(g_{i,j})$, for each gate node of each cluster of $T_{\mathscr{C}}$. $E(H)$ is defined as follows. First, for each cluster and each of its gate ports, the corresponding terminal and internal vertex are connected with an edge. For clusters C_i, C_j that are adjacent in $T_{\mathscr{C}}$, we add an edge between $int(g_{i,j})$ and $int(g_{j,i})$. If a cluster C_i contains exactly two gate nodes, g_j and g_k, we connect $int(g_{i,j})$ and $int(g_{i,k})$ with a single edge. It is irrelevant on which side of the cluster this edge will be later on, as a cluster with degree two in $T_{\mathscr{C}}$ will be drawn in the middle of the straight line between its neighbours, and so routing edges on either side of it will be equally efficient. If C_i contains more than two gate nodes, we connect the corresponding internal vertices in such a way that a circle is formed. We do not impose any requirements on the order of that circle. An exemplary construction of the routing graph H is given on Fig. 3.5.

As our routing graph is sparse, routing is fairly trivial: for each link $\{u, v\}$ to be routed, it is sufficient to take the shortest path from u to v. Given that H is unweighted, the simplest algorithm for the task would be a breadth-first search. Indeed, as H was constructed on top of a tree, the only case where the routing algorithm has to actively take agency is when arriving at an internal node of a cluster C_i with three or more adjacent clusters. In this case, for a link $\{C_{start}, C_{end}\} \in E(T_{\mathscr{C}})$, there are two paths to the unique internal node propagating links further in the direction C_{end}.

The link ordering, however, cannot be declared trivial by the same premise. Observe that with a routing graph constructed as described above, minimising the link crossings during the link ordering step corresponds to Problem 5 from the work of Pupyrev et al. [PNBH11] (where "path" refers to a link that needs to be routed):

> Given an embedded graph H and a set of simple paths P so that no node of H is terminal for some path while being internal for some other path, compute an ordering of paths for all edges of H so that the number of crossings between pairs of paths is minimized.[3]

Our routing graph H fulfils the imposed condition by construction as all edges start and end in a terminal node. The authors show that in this case, minimising the crossings

[3]quoted directly from [PNBH11], with the definition of the *Path Terminal Property* being inlined.

can be done efficiently, and provide two algorithms[4]. While the first algorithm is less efficient, this does not present a significant issue in our case, given that $T_{\mathscr{C}}$ can be assumed to be comparatively small (recall that our assumed input graphs contain already a couple of hundred vertices at most). There are also two further degrees of freedom: first, at which internal nodes the crossings are to take place, and second, in what order a link bundle must arrive at a terminal node.

We apply this algorithm, with $T_{\mathscr{C}}$ as the main graph and H as the routing graph. The algorithm proceeds as follows. It iterates the edges of H in an arbitrary order and for each edge $\{u, v\}$, an ordering of the links routed across $\{u, v\}$ is computed. To this end, first an arbitrary direction is selected for $\{u, v\}$ and then the links are sorted by traversing the edges of H where they occur together, starting from u. For links e_1 and e_2, the ordering is determined as follows:

- If an edge $\{u', v'\}$ is reached that was already processed by the algorithm, then the ordering of e_1 and e_2 is reused from there.
- Otherwise, if a vertex u' is reached after which e_1 and e_2 diverge, the ordering is selected that does not induce a crossing after the divergence. Note that this only requires a fixed cyclic sequence of each vertex' neighbours in H, which we set during H's construction.
- Otherwise, if e_1 and e_2 end in a common vertex u', this operation is repeated starting from u but this time going in the opposite direction.

This yields an ordering of the edges of $T_{\mathscr{C}}$ alongside the edges of H and according to the authors [PNBH11], their crossings are minimised. There are two important co-products of this algorithm:

- Link order at the terminal nodes: As mentioned, the ordering algorithm of Pupyrev et al. [PNBH11] results in the edges of $T_{\mathscr{C}}$ ending in the gate nodes in a specific order. This offers a trade-of: either accept this fixed order for each gate node and work with it when we draw the interior of a cluster (see Subsection 3.3.4), or allow that step to draw the edges incident to the gate nodes freely. The latter could result in a different order of the inter-cluster edges at the entrance and the exit of the gate nodes, so an intermediate matching step would be necessary that induces more crossings. We handle this in Subsection 3.3.6.
- Number of "orbits" around each cluster: The internal vertices of H are designed to connect in a circle around a cluster. The algorithm of Pupyrev et al. [PNBH11] works with edge bundles (i.e., draws multiple edges in parallel next to each other alongside the same edge of H). Therefore, in general, a cluster will have a different number of edges around it as we walk on the aforementioned circle. For instance, on Fig. 3.4, cluster C_2 has two edges on one of its sides and one edge on the other,

[4]as a side remark, this is an extension of the MLCM-T1 problem first considered by Argyriou et al. [ABKS08] where all edges are incident to nodes of degree 1.

with the third side being free. This motivates us to induce the *orbit count* of a cluster as the largest number of edges routed parallel to each other next to some cluster. So, on Fig. 3.4, the orbit counts of C_2 and C_5 are 2 and 1, respectively.

As the orbit count is independent of the vertices in a cluster, but still contributes to the space we need to dedicate to a cluster in order to draw it without overlaps, it is important to define it before drawing the clusters themselves. Additionally, as we will see in Subsection 3.4.1, we wish to be able to place labels also as circular arcs next to the node arcs they describe (that is, should the input graph come with labels or specific identifiers for its vertices). For that purpose, we manually increment each cluster's orbit count by a constant amount. For the sake of readability, we dedicate more "vertical" space to a label than to a link; for instance, 2 port units of space can be used as compared to an orbit's 1.

3.3.2 Initial Cluster Sizes

As mentioned, defining the size of a cluster is a difficult task. It has dependencies on many other values: the positions of the clusters, the inter-cluster edges and even the drawing of the cluster's interior. However, the positions in particular depend on the cluster sizes themselves, as we will see in Subsection 3.3.3; so does the interior of a cluster (Subsection 3.3.4). To resolve those cyclic dependencies, we setup an initial size for each cluster that disregards some of those constraints; we will handle them in Subsection 3.3.5.

We aim for a cluster design not unlike the one of ChordLink, as will be seen in Subsection 3.3.4. In particular, we will represent the non-gate nodes of a cluster as radial arcs on the boundary of the cluster. Note that the gate nodes are only implicitly assigned an arc in order to simplify and generalise the used algorithms and formulas; these arcs are *not* visualised as such on the cluster's boundary. With this in mind, we clearly require enough space to position all arcs.

Recall that one of our goals, defined in the beginning of this section, is for the size of the radial arcs to represent the "relevance" of the node. As the graph does not necessarily come with node attributes of some sort, we opt in favour of using the node's degree as a measure for the size of the arc. This has the additional benefit of yielding cluster drawings where the links incident to a node arc are evenly spaced throughout the entire cluster. We thus define a *port* $p_{u,i}$ as the location where the i-th link incident to the node arc of u connects to that arc. Similarly, we can define a constant *port unit* to represent the space dedicated to a single port on the final drawing. This allows us to express the minimum circumference of the cluster needed to draw all ports of all arcs in terms of this unit:

$$p(C_i) \geq \sum_{u \in C_i} (\deg(u) + 1) + \sum_{j} \deg(g_{i,j}) \tag{3.3}$$

Or, in other words, the number of ports among all port and non-gate nodes. This guarantees that a cluster with many edges inside it will get more space assigned. For clarity, we additionally increase the circumference of the cluster by the number of nodes in it (the +1 in the first sum); these empty spaces will be placed between the node arcs to make them better distinguishable and to make the entire drawing clearer.

Regarding the orbits, we have already computed the orbit count of the cluster in Subsection 3.3.1. For convenience, we also set the "width" of one orbit to one port unit. So in total, since a cluster C_i needs to be large enough for both its orbits and the node arcs, the initial radius is given by

$$r_i = \frac{\sum_{u \in C_i}(\deg(u) + 1) + \sum_j \deg(g_{i,j})}{2\pi} + orbitCount(C_i) \tag{3.4}$$

3.3.3 Radial Layout

Note that for the large-scale drawing of $T_{\mathscr{C}}$, we need a layout algorithm of our own as ChordLink does not perform any layouting per se: It begins with an already existing drawing (without requiring it to have any specific properties or features) of the input graph and then replaces a set of selected vertices with a cluster. The remainder of the graph drawing is left untouched, aside from the edges incident to any of the clustered vertices.

In addition, recall that we strive for an "optically pleasing" drawing. While this statement is rather vague, we can still infer some criteria from it. For instance, we decided against using a force-based drawing algorithm relatively quickly, as this kind of algorithms do not consider any kind of spatial distribution. As we already require the coarse-grained layout algorithm to be able to handle different vertex sizes, it is also natural to require a drawing that dedicates space to a vertex (or a subtree) proportional to its size. Ideally, the drawing should also exhibit some amount of symmetry, although this is difficult as, in the general case, the graph $T_{\mathscr{C}}$ we are drawing is not symmetric.

We considered adapting various existing algorithms for this task. An early idea consisted of representing $T_{\mathscr{C}}$ as a cactus graph; however, this was swiftly dismissed since first, the transformation step would introduce an unnecessary level of complexity in the drawing process and second, hardly any layout algorithms dedicated to cactus graphs exist to the best of our knowledge, even less so ones catering to our specific needs.

A more promising idea involved the concept of *balloon drawings*, as given e.g. by Lin and Yen [LY05] (see Fig. 3.6 (a)). Notice that the algorithm presented in their work already complies with some of our desired properties: it is applicable on trees, takes varying vertex sizes into consideration and, in the drawing, distributes the space proportional to the subtree sizes. Alas, it also entails the implication of a hierarchical structure within the drawn graph, with the root of the tree being on top of the hierarchy and children being placed on the circumference of a circle centered at their parent – a property we explicitly desire to avoid. Applying similar reasoning, we decided against the work of Grivet et al. [GADM04] who introduce the notion of *bubble trees* (see Fig. 3.6 (b)). While they, too, consider trees with circular vertices of varying size, they draw the subtree rooted at a vertex implicitly enclosed in a circle large enough to fit it (thus suggesting a hierarchy).

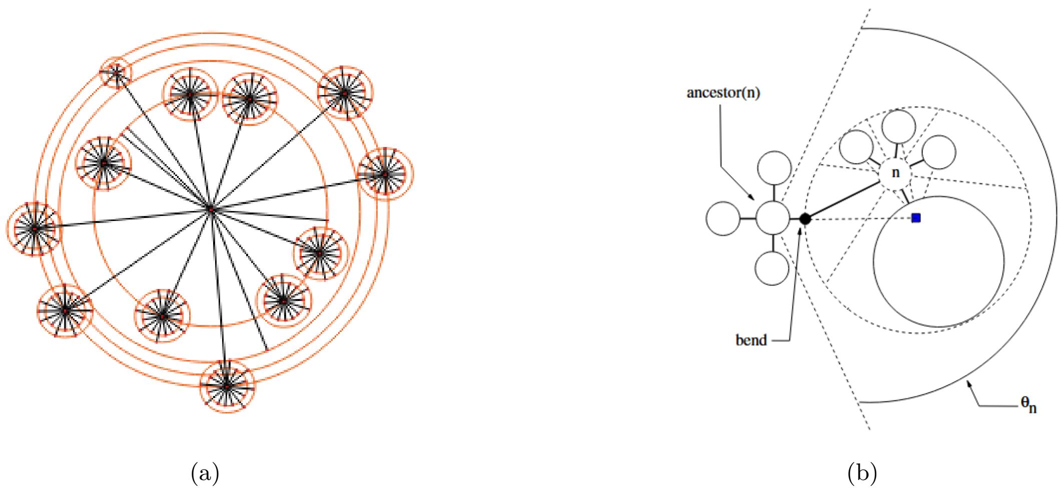


Fig. 3.6: Examples for a balloon-style drawing (a) and a bubble tree (b), as given by Lin and Yen [LY05] and Grivet et al. [GADM04], respectively.

Still, the idea of using a radial layout for the large-scale drawing appeared prospective and so we focused on it further. It must be emphasized here that our goal during this step is fixing the cluster coordinates for $T_{\mathscr{C}}$ using the initial cluster radii defined in the previous subsection. The locations of the inter-cluster edges were handled in Subsection 3.3.1; they will be drawn in detail later in Subsection 3.3.6.

While the "classical" radial drawing algorithm, presented by Eades [Ead91] and Battista et al. [BETT99], considers the drawn nodes to be of equal size (thus making the weight of a subtree equal to the number of nodes in it), this can be extended to allow nodes of different sizes to be included. This is also the approach taken by Six and Tollis [ST99] (see Fig. 3.7). Of particular interest for us is the algorithm "RADIAL – With Different Node Sizes" presented in that work. In regard of the large-scale drawing of $T_{\mathscr{C}}$, this algorithm covers all of our requirements:

- It is capable of handling vertices (in our case for $T_{\mathscr{C}}$ – the clusters) with predefined sizes.
- It distributes the space of the drawing canvas relatively to the sizes of the separate subtrees, taking not only the vertex count of the subtree into consideration, but also, once again, the vertex sizes.
- It does not induce any sort of "penalty" on vertices further away from the root, thus subverting the idea that the structure of the drawn graph is hierarchical. To the contrary, instead of constraining child vertices to a space defined by their parent, the algorithm allocates space to the parent strongly dependent on the children.

Similarly to other radial layout algorithms, "RADIAL – With Different Node Sizes" assigns polar coordinates (ρ, θ) to each node v with the root of the input tree always

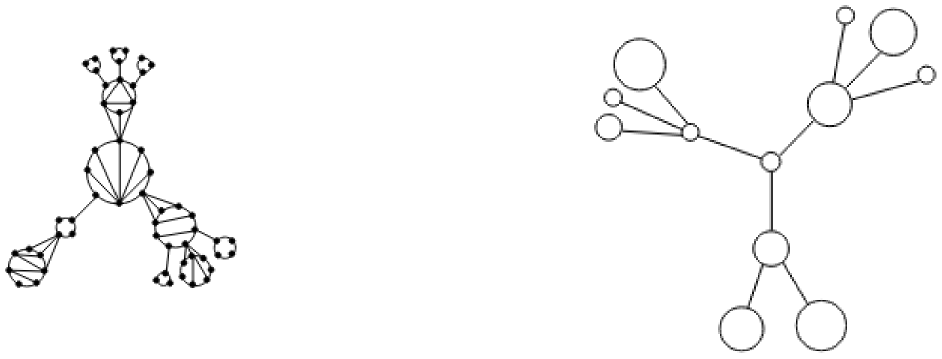


Fig. 3.7: Examples for drawings produced by the algorithm of Six and Tollis [ST99].

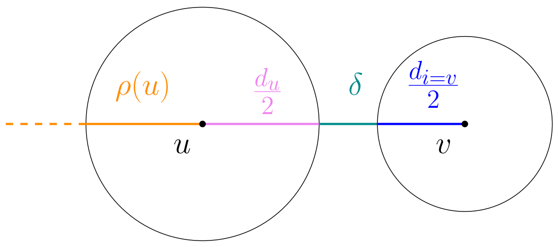


Fig. 3.8: The ρ coordinate of a node v, as defined by Six and Tollis [ST99]. The shown child of u – v – is the one with the largest diameter.

being placed at $(0, 0)$. The ρ coordinate is the distance from $(0, 0)$ to the centre of the node and in the work of Six and Tollis is defined, for a node v, as follows:

$$\rho(v) = \rho(u) + \delta + r_u + \max(r_1, r_2, \ldots, r_k) \tag{3.5}$$

Here, node u is the parent of v, r_u is its predefined radius, $r_1, \ldots, r_k$ are the radii of the children of u (including v) and δ is a predefined constant representing the smallest distance allowed between the boundaries of two nodes. This definition of ρ is intuitive: in theory, it leaves just enough space (δ) between u and v, as seen on Fig. 3.8. Moreover, a noteworthy feature is that it places all children of a node u at an equal distance from $(0, 0)$, *not* from u's centre.

Unfortunately, Six and Tollis' work does not provide an explicit formula or definition for the θ coordinate – the counter-clockwise angle between the x-axis and the line segment connecting $(0, 0)$ and the centre of the to-be-drawn node. It is merely mentioned that any such formula should incorporate the widths of that node's descendants and not merely their count, as is often the case with radial algorithms. In this spirit, we define

$$weight(u) = \begin{cases} r_u & \text{if } u \text{ is a leaf} \\ \max(r_u, \sum_{v \text{ child of } u} r_v) & \text{otherwise} \end{cases} \tag{3.6}$$

We then iterate $T_{\mathscr{C}}$ in a pre-order manner and split the plane into wedges (radial sectors) starting at $(0, 0)$. Each node's θ coordinate is given by the angle between the x-axis and the bisector of its assigned wedge. We begin with the root, whose "wedge" is the entire plane. For a node u and its wedge $wedge(u)$, we further split that wedge among u's children $v_1, \ldots, v_k$ proportionally to $weight(v_i)$. This way, a node is assigned a larger wedge if the subtree rooted at it is comprised of nodes with larger radii. Regarding the exact order in which the children v_i are drawn, we adopt the cyclic order that was used in Subsection 3.3.1 when routing the inter-cluster edges.

As we will discuss later, in our case the radii may change throughout the drawing process. This case is handled in Subsection 3.3.5.

3.3.4 Cluster Drawing

The design of the cluster layout exhibits large similarity to the one of ChordLink. However, it had to undergo some modifications to comply with the criteria we laid out at the beginning of this section, and also with the fact that the input graph is drawn as a tree $T_{\mathscr{C}}$ on a radial layout.

Recall that Subsection 3.3.1 introduced the notion of gate nodes. The main reasoning behind this is that visualising the inter-cluster links as simple straight lines would disrupt the otherwise clean tree structure we have established in Subsection 3.3.3, and thus lead to more poorly comprehensible drawings. Instead, the links should follow the tree structure and enter and exit the clusters in predefined points.

Both gate and non-gate nodes are represented as radial arcs (with the remark from Subsection 3.3.1 that the arcs of the gate nodes are not drawn in the end), as would be the case in the original ChordLink. However, with them, a cluster becomes a self-contained entity: as long as we control the positions of the gate nodes, we are free to draw the "normal" nodes and the rest of the interior of the cluster in an arbitrary manner without the cluster influencing the rest of the drawing.

Once the algorithm from the previous subsection has defined the initial positions of the clusters in the drawing of $T_{\mathscr{C}}$, we can fix the positions of the gate node arcs on the clusters' boundaries. Ideally, the gate node to gate node links between a cluster and its neighbours (in regard to $T_{\mathscr{C}}$) will be drawn without bends, as per the routing graph from Subsection 3.3.1. Therefore, we place the gate node arcs centered around the straight lines connecting the centres of the cluster and its neighbours, respectively, as seen on Fig. 3.9 (a).

In the case where the gate node arcs are large (as there are many links leaving a cluster towards another one) and the neighbouring clusters are positioned close to each other, the setting of the gate node arcs we described may lead to a situation where two or more gate node arcs overlap, as shown on Fig. 3.9 (b). However, both the sizes of the gate node arcs and the cluster positions are fixed at this point. This makes it possible to use angular geometry in order to define a new lower bound for the radius of the cluster that will allow the gate node arcs to be placed without overlapping with each other.

Consider cluster C_i placed at (ρ_i, θ_i). Assume the n neighbours of C_i are placed at polar coordinates $(\rho_k, \theta_k), 1 \leq k \leq n$ by the algorithm of the previous subsection, and

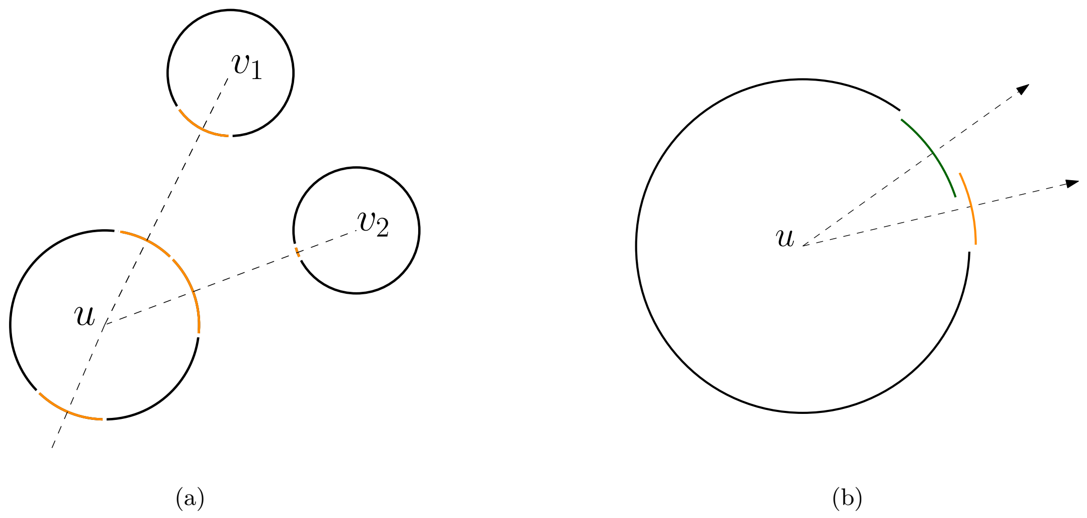


Fig. 3.9: Examples of naïve placement of gate node arcs on a cluster's boundary. In (a), the initial placement was successful. In (b), there would be an overlap if the arcs were placed on their initial positions.

the gate node arcs of C_i have sizes l_k. In the following calculations[5], r'_i will refer to the *necessary* radius of C_i to avoid overlaps.

1. First, we use the trivial conversion to Cartesian coordinates given by $x_k = \rho_k \cos \theta_k$ and $y_k = \rho_k \sin \theta_k$ to compute the angles of the arcs in the local coordinate system centered at C_i:
$$\varphi_k = \text{atan2}(x_k - x_i, y_k - y_i)$$
where atan2 refers to the 2-argument arctangent function yielding the angle between the positive x-axis and a ray from the origin to (x, y).

2. Then, sort the arcs ascending by that angle.

3. We calculate the gaps between the angles:
$$\Delta_k = \begin{cases} \varphi_{k+1} - \varphi_k & \text{for } 1 \leq k \leq n-1 \\ 2\pi - \varphi_n + \varphi_1 & \text{for } k = n \end{cases}$$

4. If arcs k and $k+1$ (taken modulo n) were to meet exactly, it must hold that
$$\Delta_k \geq \frac{l_k + l_{k+1}}{2r'_i}$$

This yields the following constraint: $r'_i \geq \max_k \frac{l_k + l_{k+1}}{2\Delta_k}$. Let $r^*_i = \max\{r_i, r'_i\}$; we will use this value going forward. Now, it is safe to place the gate node arcs of C_i on its boundary without causing overlaps.

[5]Some of the given formulas were derived with the help of AI assistants and then verified manually.

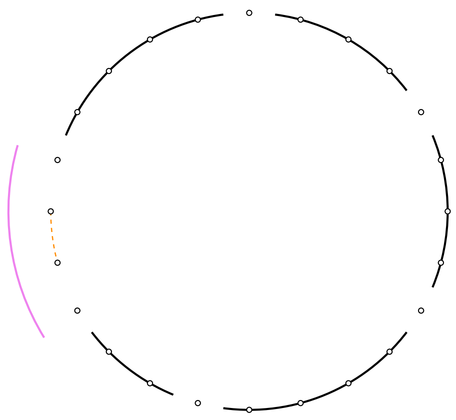


Fig. 3.10: An example where some arcs have already been inserted. There is sufficient space for one more arc (orange) with size at most 2 (as the ports next to the already placed arcs are buffers). However, the next arc (violet) has size 4, so a growth phase is necessary.

With this done, we now add the remaining, non-gate node arcs on the boundary of the cluster. A somewhat similar problem was tackled by Baur and Brandes [BB04] as they attempt to place nodes on a circular boundary while minimising the number of edge overlaps the placement would yield. As they show, this problem is $\mathcal{NP}$-complete. Nevertheless, the heuristics provided in their work can be of use to us as well. Note that we still have some tighter restrictions: for a given node, not every location is valid as it may not be large enough to host the arc without causing overlaps.

As proposed by Baur and Brandes [BB04], we process the non-gate nodes one-by-one. We also opt in favour of the *connectivity* rule for determining an insertion order, meaning: we select a node with the least number of unplaced neighbours. The rationale behind this is to keep the number of edges low for which one of the incident vertices is already placed while the other is not.

However, the decision *where* to place the node at hand is not as simple as in the work of Baur and Brandes [BB04]. In particular, a case can occur where due to the placement of the previous node arcs, there is no singular gap wide enough to fit the current one (see Fig. 3.10). Additionally, we wish to keep one port unit of buffer space between arcs for the sake of having a clearer drawing in the end. Unlike ChordLink, we also explicitly want to avoid splitting a node arc in multiple pieces. To mitigate this, we introduce the so-called *growth phase*.

In order to provide more space for the node arcs to be placed, we let (the drawing of) the cluster artificially "grow" (starting at r_i^* and updating this value after each expansion) until no overlaps occur. In each growth step, we expand the cluster size by a specific amount defined below. This is a feedback loop: once an expansion has occurred, we begin the drawing step anew, fixating the gate node arcs and attempting to insert the residual nodes. Note that this is a finite process as the sizes of the node arcs (both for gate nodes and otherwise) remain unchanged; we are merely "generating" more space on which to draw them without having to split them.

Going back to the stepwise insertion of non-gate nodes, we handle three different cases, for a node u:

1. There is no gap between already placed node arcs that is large enough to fit the node arc of u (plus a buffer on each side, if not present via neighbouring arcs): in this case, the cluster undergoes a growth phase, expanding by $\deg(u) - maxGap - 2$ (where $maxGap$ is the size of the largest gap currently available in port units), and the node placement step begins again.
2. There is a single gap between already placed node arcs that is large enough to fit the node arc of u (plus buffers): here, we simply put u in that gap and proceed with the next node.
3. There are multiple gaps between already placed node arcs that are large enough to fit the node arc of u (plus buffers): we place u in the gap that would result in the least amount of new edge crossings. These crossings are easy to find since for cyclically ordered nodes, two links $\{x, y\}$ and $\{z, w\}$ cross exactly when their nodes appear in an alternating sequence on the cyclic order (e.g. $x \dots z \dots y \dots w$).

After iterating through the node placement (and, if necessary, growth) steps, we are left with a final drawing of the node arcs of the cluster, and a minimum size constraint r_i^*. Note that since we expect some number of growth phases, the final size of the cluster is difficult to predict in advance. This collides with the layout algorithm of Six and Tollis described in Subsection 3.3.3. Should a cluster C_i's final size be larger than the initial one, C_i will extend beyond the boundaries of its wedge if it is placed on its original position. Our initial solution revolved around performing the tree layout procedure again with C's new size. However, doing so would require a rebalancing of the tree, including already drawn clusters, among other problems. We will show in the next Subsection 3.3.5 how we tackle this issue.

The final step needed to complete the cluster would be to select, for each link $\{u, v\}$, which ports of the arcs of u and v to connect in order to keep the number of crossings small. Recall that in the end of Subsection 3.3.1 we were left with a trade-off: The order of the links at each gate node is already defined by the routing algorithm of Pupyrev et al. [PNBH11]. So, we either use this order when inserting the edges inside the cluster, or insert all edges freely and, at the gate nodes, have a "crossing phase" to match the links on both sides of every gate node. We believe that the latter option will lead to more visual clutter and make the drawing less intuitive. So, the benefit of potentially less crossings of links in the interior of the cluster is deemed insufficient and we opt for the first option. The algorithm we designed is given below on Alg. 2. There, the "highest" port $p_{u,i}$ of a node u refers to the port with the largest index i (that satisfies some property).

Algorithm 2: Our algorithm for placing edges to connect the ports of an already drawn cluster.

```
fn PlaceEdges($arcs : RingBuffer$)
    for each arc $A \in arcs$.ordered() do
        if $A$ corresponds to a gate node then
            set the ports to $Open(\dots)$ as yielded by the algorithm in [PNBH11]
            continue
        for each arc $A' \in arcs$.nextOf($A$) to $arcs$.prevOf($A$) do
            if $A' \notin adj(A)$ then
                continue
            for each edge $e \in edges(A, A')$ do
                if $A'$ has port $p_{A'}$ that is $Open(A)$ then
                    $p_A \leftarrow$ highest port of $A$ that is $Free$ or $Open(A')$
                    $p_A.state \leftarrow Connected(p_{A'})$
                    $p_{A'}.state \leftarrow Connected(p_A)$
                else
                    $p_A \leftarrow$ highest port of $A$ that is $Free$
                    $p_A.state \leftarrow Open(A')$
```

The given algorithm takes into consideration not only the current node u's free ports, but also the fact that the node v that u is attempting to place a link to might have to be connected to other nodes after u as well (see Fig. 3.11). As for the links, they are drawn, in a ChordLink-manner, as chords in the inside of the cluster's boundaries. This completes the drawing step of the clusters.

3.3.5 Resolving Issues

Unfortunately, deeper analysis and experimentation of the approach described so far unveiled some issues that would prevent our model from working as expected. In this subsection, we go into detail about them and describe the steps we undertook in order to resolve them.

Node placement with Six and Tollis. The first problem was found with the algorithm of Six and Tollis [ST99]. As mentioned earlier, the authors do not provide an explicit way to compute $\theta(u)$ – the counter-clockwise angle between the x-axis and the line segment from $(0, 0)$ to the position of the node u – save for mentioning that this computation must take the sizes of the descendents of u into consideration. The function chosen by us satisfies this condition.

A graph was discovered during the concept phase which, when drawn with the specified ρ and θ coordinate functions, will still have some clusters extend beyond the boundaries of the wedges assigned to them (see Fig. 3.12).

Luckily, this issue was not difficult to resolve. It is sufficient to add a condition to the

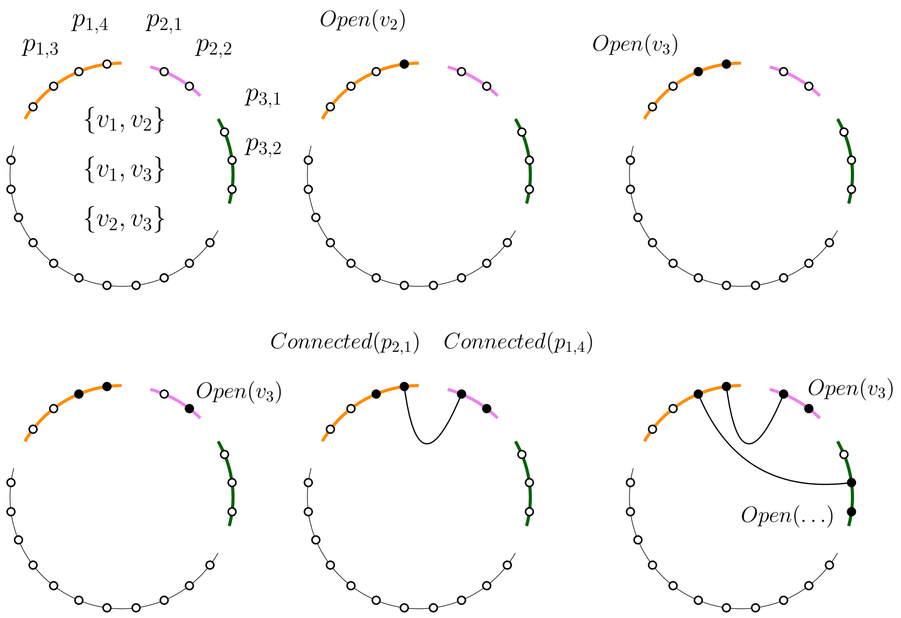


Fig. 3.11: An example of link placement between ports. First, ports become open towards a node they'll later connect to. In the 5. image, as v_2 attempts to place the $\{v_1, v_2\}$ link, it discovers v_1's open port $p_{1,4}$ towards v_2. The highest free port of v_2, $p_{2,1}$, then connects to $p_{1,4}$. On the 6. image, v_3 behaves similarly.

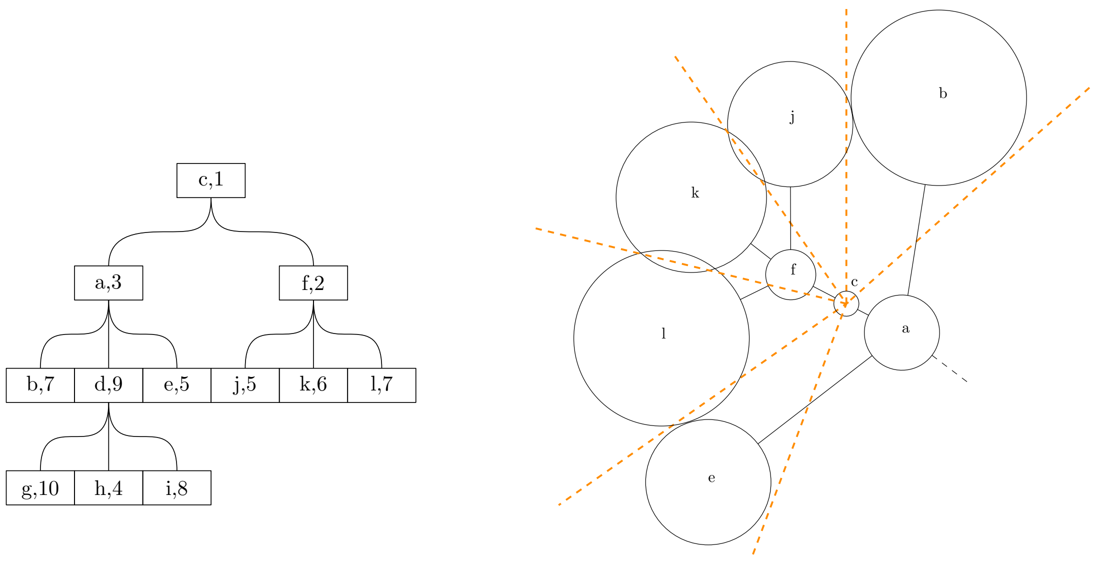


Fig. 3.12: Example for a graph that, when drawn with the algorithm of Six and Tollis [ST99], has overlaps of the clusters. Left: representation of the graph with the nodes given as pairs (name, size). Right: drawing of the problematic part with the orange lines showing the dedicated wedges.

ρ function that the distance from $(0,0)$ to the centre of the cluster must be large enough for the cluster's initially planned visualisation to fit in (we will elaborate the interactions with cluster growth shortly). Given the wedge assigned to node v, $wedge(v)$, the adjusted ρ coordinate is given by

$$\rho(v) = \delta + \max \begin{cases} \rho(u) + r_u + \max(r_1, r_2, \ldots, r_k) \\ \frac{r_v}{\sin \frac{wedge(v)}{2}} \end{cases}$$

where u, r_i, and δ are defined as before. Note that even so, this formula is unable to produce the results depicted in the work of Six and Tollis (Fig. 3.7) – a careful look into the drawings presented by the authors reveals that there, the centres of the children of a node u are equidistant from *the centre of u itself*, not from $(0,0)$, contradicting the original formulation of ρ given by the authors themselves. Still, this error does not present an issue for our model.

Clusters extending over their wedges after a growth phase. As noted earlier in Subsection 3.3.4, at the end of the drawing phase of a cluster C_i we have computed a new lower bound r_i^* for its size, as constrained by the node arcs. Since sufficient space to draw the orbits is also necessary, the final size of a cluster is $r_{fin} := r_i^* + orbitCount(C_i)$. In particular, this bound can be already larger than the value used in the algorithm of Six and Tollis in Subsection 3.3.3 (including the amendments from the first part of this subsection). Drawing C_i with the new radius will, in general, cause it to extend beyond the wedge defined for it by the layout algorithm. Alas, simply re-running the layout algorithm is not a plausible solution as it would create multiple issues:

- A changed balance of the clusters and the subtrees would yield new locations for the clusters in the general case. Thus the created drawing of C_i would no longer be valid as the gate node arcs would not point towards their respective clusters any more. This in turn leads to a repeated cluster drawing step for C_i, which could once again produce a new size for it, and so on. This process is not guaranteed to find an optimum and terminate.
- This changed balance would also disrupt previously drawn clusters; therefore, in the worst-case, all clusters would have to be recomputed as well.
- Even aside from terminating, such an iterative procedure would be very inefficient, regardless of us limiting our model to small graphs only.

It is clear that any solution that accepts a cluster's size to grow during the drawing phase is not allowed to invoke a new layout step completely from scratch, or to influence the entire cluster graph $T_{\mathscr{C}}$. In other words, we require a local solution enabling us to draw a larger cluster in the same wedge that was originally assigned to it.

What we ultimately opted for is a simple trade-off: we keep the drawing of the cluster C_i centered in its wedge (i.e., the θ coordinate remains unchanged), but we shift it further away (i.e., increase the ρ coordinate). With r_{init} and r_{fin} being the initial and the final size of C_i, the necessary shift is given by

$$dist = \frac{(r_{fin} - r_{init}) \cdot \rho(C_i)}{r_{init}} \tag{3.7}$$

or, in other words, C_i's final placement is at polar coordinates $(\rho(C) + dist, \theta(C))$.

This raises the question of how the subtree rooted at C_i is to be handled, as it is possible that pushing C_i further away from $(0, 0)$ yields an overlap with the drawing of one of its children. We solve this issue in two steps: first, we draw $T_{\mathscr{C}}$ in a post-order manner; this ensures that no cluster will be drawn after its parent's final placement, and that the parent can rely on the coordinates of its children being fixed in order to place its gate node arcs accordingly. Second, when we move an expanded cluster C alongside the $(0, 0) - (\rho(C_i), \theta(C_i))$ line, we also shift with it the subtree it hosts. Note that we do *not*, in any way, change the details of the drawing of the subtree, such as directions; we simply push it further away. This is guaranteed to cause no overlaps as the subtree is placed entirely in C_i's wedge, and that wedge is wider further away from the centre of the coordinate system.

The downside of the aforementioned trade-off influences the link(s) from C_i's parent cluster, C_j, to C_i. The position of $g_{i,j}$ has already been fixated, but it no longer directly points towards C_j. Note that rotating C_i and its subtree to compensate for this is not recommended as the subtree may thus enter the wedge of one of the sibling clusters of C_i. It is thus clear that a bend in the inter-cluster links between the two is necessary. We choose to place this bend near C_i. This minimises the possible interferences of the links in the wedges of the siblings of C_i. The positioning of $g_{j,i}$ on the other hand is no reason for concern. As C_j is drawn after C_i has been finished (post-order traversal), that gate node can be positioned directly towards C_i's new location. The gate node placement procedure described in Subsection 3.3.4 makes sure that the gate nodes placed towards sibling clusters will not overlap. An example of the steps just described is given on Fig. 3.13.

3.3.6 Finishing the Inter-Cluster Links

Recall that in Subsection 3.3.1, we performed a preliminary setup of the inter-cluster edges in order to determine the orbit counts and the edge orders. We achieved this by constructing a routing graph and then applying the algorithm of Pupyrev et al.[PNBH11]. Still, it remains to actually draw them like on Fig. 3.4, now that the setup and drawing phase of the clusters is complete as well.

Ultimately, we only have to handle the exact behaviour of the links at the boundaries of the internal and the terminal nodes. We handle an internal node $int(g_{i,j})$ of cluster C_i as an area in the form of an annular sector, as given on Fig. 3.14. These sectors have radial width equal to $orbitCount(C_i)$, and arc length equal to the number of links entering the $int(g_{i,j})$ from direction of $int(g_{j,i})$. This way, links that are "orbiting" the cluster at this internal node have enough space to pass by, and so do links arriving at

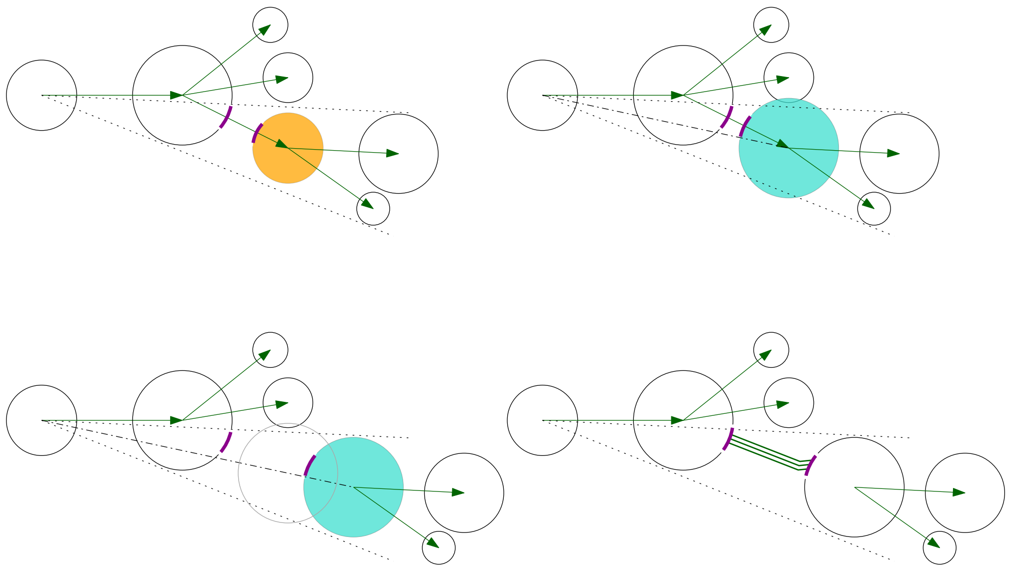


Fig. 3.13: An example of the cluster expansion process. The green arrows represent the cluster tree, and the purple arcs – the gate node arcs. The orange cluster is expanded to its new size (blue) which no longer fits in its dedicated wedge (between the dotted lines), so it is shifted further away. Edges between the cluster and its parent now have a bend.

or leaving the cluster. Recall that the orbit count was artificially inflated to allow us to draw the graph's labels at the end; to this, we dedicate the two innermost orbits.

Keeping the focus on C_i, let the clusters incident to it be C_j, C_k and C_l. We place ports on the sides of the internal nodes to affix the locations where links can enter. Once again, Fig. 3.14 visualises the described idea.

- For $int(g_{i,j})$, on the side directed towards $int(g_{j,i})$, we position as many ports as the number of links entering it from that side.
- For $int(g_{i,j})$, on the two sides directed towards $int(g_{i,k})$ and $int(g_{i,l})$, we position as many ports as is the orbit count of the cluster, starting from the inside. These are for links that simply go around C_i on the way between two clusters.
- As the radial width of $int(g_{i,j})$ is equal to the number of links entering it from $int(g_{j,i})$, on the opposite side – the one towards $term(g_{i,j})$ – we distribute $deg(g_{i,j})$ many ports not along the entire width, but starting from the middle. This way they can transit directly to $g_{i,j}$.

As for the small bends introduced at the end of Subsection 3.3.5, we mentioned that we decide to place them next to the cluster C_i that was pushed further away, like on Fig. 3.13. Since we do not want any crossings at the bends, we do not need to re-do the routing or ordering steps. Consider the inter-cluster links from C_j to C_i. Any location is suitable enough for the bend as long as first, it lies on the ray from the center of C_j

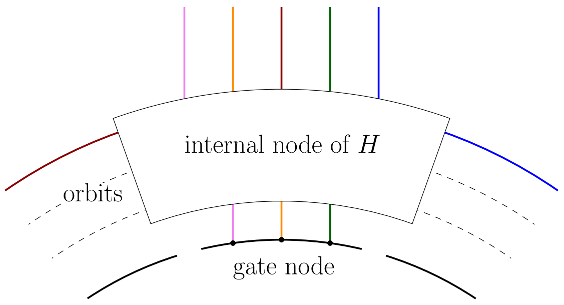


Fig. 3.14: An example of an internal node next to a cluster with orbit count of three. The cluster only has one valid orbit; we dedicate the lower two orbits to labels. The colours exemplify a possible way of how the links could be routed.

to the center of C_i and second, is sufficiently close to $g_{i,j}$ while allowing for all links to bend so that they can enter $g_{i,j}$ in their corresponding port.

Note that there are always at least as many links entering an internal node from the direction of another cluster as there are entering it from the connected terminal node, so taking the former as a measure for an internal node's size is sufficient. This can be proven simply: suppose there were k links entering from the terminal node, and $l < k$ many entering from the neighbouring cluster. Then, at least $k - l$ many of the former links would have to leave the internal node from one of its sides, travelling to another internal node of the same cluster. This is a contradiction to our design, as these links are thus taking an unnecessary detour; they would have been directly connected to that internal node instead.

Clearly there is sufficient space around the clusters to place the internal nodes as we have already constrained the radius of the cluster to include its orbit count (which is also the width of the internal nodes). In the provided area, the links can traverse their predefined paths, crossing if necessary. This concludes the considerations of the internal nodes.

Regarding the terminal nodes, there is not much left to do either. In fact, they exist merely for the purpose of having the links end in vertices of degree 1 in the routing graph. As we noted in Subsection 3.3.4, we decide against changing the order of the links that was defined by the routing algorithm of Pupyrev et al. [PNBH11] in an additional step. arissa With this, the inter-cluster edges are complete. Due to the way we have defined the orbits and the internal nodes, no nudging is necessary. Note that this also holds for the drawn links within a cluster, as there the node arcs already have fixed ports that leave sufficient space as well.

3.4 Labels and Other Details

Last, we would like to discuss some elements important for the readability of the produced drawing.

3.4.1 Labeling the Drawing

Vertex labels are of high significance when visualising real-world data. Without them a user is often left puzzled as to which vertices are, for instance, clustered together. In our particular case, all vertices of the original input graph G now lie on the boundary of their respective clusters in the form of circular arcs. This special case has also gained interest as of recent, considering for instance the work of Islam et al. [IHB+23], with the general use-case often being various round displays like smartwatches. Recently, Bonerath et al. [BNT+24] proposed a new variant of the classical boundary labeling problem called *orbital boundary labeling* and analysed some variants of it.

It is clear in our case that we wish to place labels outside of the cluster's drawing, as the inside is reserved for the chords. Also, the gate nodes do not need a label as they are artificially created; they can also be enhanced with additional information, but this will be discussed shortly. As for the labels of a cluster C_i's "normal" arc nodes, there are various possibilities for their placement.

The simplest option would be to draw them as freely placed axis-parallel rectangles next to the cluster; however, this is impractical as they could potentially cause overlaps by extending beyond C_i's dedicated wedge. We also consider shifting a cluster further away, as described in Subsection 3.3.5, possible but not beneficial enough to outweigh the distortions that would be caused in the drawing this way. Similar is the case with the labeling model utilised by ChordLink itself, which places labels perpendicular to the node arcs they correspond to. While this does not cause potential overlaps between labels of the same cluster, it also blocks a potentially large amount of space around each cluster. This is acceptable for ChordLink, given that it does not draw graphs in a structured way on the large scale, but even worse than the previously discussed option in our case.

This leaves as a viable option to draw the labels on the boundary of the cluster as circular arcs, similarly to our approach to the nodes. Recall however that in general, inter-cluster links are routed next to a cluster's boundary (except for clusters that are leaves in $T_{\mathscr{C}}$, but this is an edge case). We are therefore presented with a choice:

Opt. 1 Draw the labels between the cluster's boundary and the routed inter-cluster links

Opt. 2 Draw the labels outside of the routed inter-cluster links.

While both variants are similar in that they apply an additional "layer" around the cluster and thus increase the lower bound of its radius, they also offer a trade-off. The first option feels more natural as the labels are placed directly next to the node arcs they describe (i.e., without anything in between). Option 2 provides a bit more space as the labels are placed further away, but by separating them from the node arcs we make them less helpful. Using leaders – lines connecting a node arc to its corresponding label – undermines the readability of the drawing as the leaders will need to cross the inter-cluster arcs.

In addition, in both cases we do not have any way of guaranteeing that sufficient space will be available for the labels (in regard to their length) to be drawn. Initially, an idea was proposed where similarly to the orbital boundary labeling problem of Bonerath et al. [BNT$^+$24] the node arcs were drawn with size proportional to their label's length. This would indeed have trivialised the label placement problem; however, we abandoned this idea as it felt unnatural to regulate the length of a node arc by a feature outside of the graph structure. For instance, applying this approach could have produced drawings where the largest arc of a cluster corresponds to a node of degree 1, but with a long label attached to it. Opting for the arc length to be defined via the degree of the node was, in our opinion, a more logical choice.

As we could not arrive at a completely satisfying solution, we opt in favour of the first variant described above and implement it as follows. First, recall that in Subsection 3.3.1, we manually increased a cluster's orbit number by 2, thus leaving space for the labels. Since the gate node arcs are not labeled, they serve as delimiters that divide the space around a cluster in segments. For vertex u, let the length of the label describing u be $\lambda(u)$. We compute the following metric for each cluster C_i:

$$arcFit_i = \max_{u \in C_i} \frac{\lambda(u)}{deg(u)} \tag{3.8}$$

We wish to be able to fit every label directly over the arc it describes. The given metric provides a factor for each cluster that enables scaling of the labels so that this is possible. Should the down-scaled label be shorter than the node arc it describes, it is positioned over the middle of that arc. While this solution is admittedly suboptimal due to using this approach, we believe that it is sufficient as a compromise between viability and aesthetic.

3.4.2 Miscellaneous

Here, we focus on some other minor details of the drawing. We begin by examining the gate nodes closer. While our design enables the user to clearly see the connections between the nodes of the same cluster, this is not entirely the case with inter-cluster-edges. To the contrary, they simply exit from the cluster via a gate node and one must follow their curve to recognise what the target cluster (or node) is. We propose a solution to this. In the previous subsection, we mentioned that no label is placed over a gate node arc. Instead, we suggest using the space to apply small labels aligned with the departing links. Consider link $e = \{u, v\}$ with $u \in C_i, v \in C_j$. Then, as e departs C_i, it gets the label of v at the gate node. This is repeated symmetrically at C_j.

We also propose enhancing the produced image with the use of colours, via selecting a clearly distinguishable colour for each cluster. This is also a difference to ChordLink where each *node arc* gets assigned a dedicated colour; we put more emphasis on the clusters as entities. While keeping the labels of the non-gate nodes black is a solid

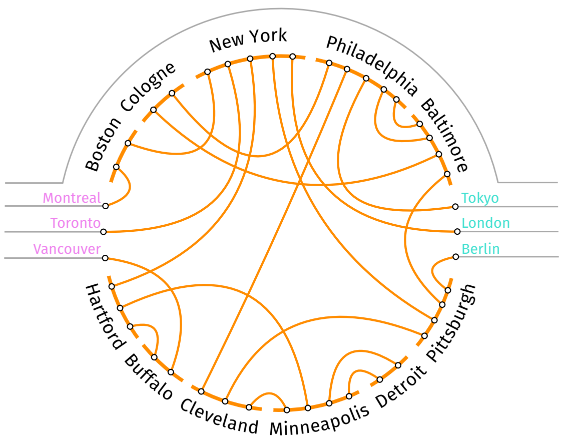


Fig. 3.15: An example of a cluster with labeled node arcs and inter-cluster links towards two other clusters, one of which is coloured in violet and the other one – in turquoise. The ports are displayed for clarity.

approach, we suggest colouring the labels over the inter-cluster links we just described. In particular, in the example from the previous paragraph, the label of v as $\{u, v\}$ departs from C_i gets the colour assigned to C_j, the cluster v lies in.

The ideas proposed in this section are visualised on Fig. 3.15.

4 Summary and Future Work

In this work, we presented a hybrid graph drawing pipeline inspired by ChordLink [ADM+19]. Our pipeline primarily targets small locally dense but globally sparse graphs, with the small size allowing for some simpler, but less efficient algorithms to be used. Similarly to ChordLink, our pipeline draws the communities of a graph as circles, and the nodes of the communities – as circular arcs on the boundary of their cluster. Intra-cluster edges are also still drawn as chords in the interior of the cluster containing the vertices they connect. However, unlike ChordLink, our pipeline does not require user interaction and is fully autonomous.

First, our pipeline extracts the communities from the input graph automatically by smartly cutting a dendrogram constructed via the measure of betweenness centrality. It proceeds by computing a rooted cluster tree $T_{\mathscr{C}}$ and then draws the tree using a radial layout inspired by the work of Six and Tollis [ST99]. This is another difference to ChordLink, which does not apply any particular structure to the large-scale drawing of the graph. Our model routes the edges of the graph which connect non-adjacent clusters in $T_{\mathscr{C}}$ alongside the tree structure and lets them enter their incident clusters via dedicated gate nodes, instead of simply connecting adjacent vertices to one another regardless of the cluster they are in. When these edges need to be routed around a cluster they are not incident to, they do so in dedicated layers we call orbits. To conclude the description of our model, we discussed some visual details such as labeling the drawn graph.

We believe the drawings produced by our pipeline to be more visually pleasing than the ones given by ChordLink, to no small part due to the explicit structure that is present. If necessary, our pipeline is also capable of some degree of flexibility, for instance in terms of the dendrogram construction and cutting mechanisms, although we believe to have selected good algorithms and metrics for the task.

4.1 Future Work

Here, we present some topics that, in our opinion, could probably be improved somewhat given additional analysis and research.

Directed graphs. As our work so far assumed that we are given an undirected graph as an input, we believe it important to discuss the opposite case as well. Recall that in the majority of our pipeline we use the cluster tree $T_{\mathscr{C}}$ which is undirected. Therefore the only real challenge is presented by the initial step of clustering the input graph (as the tree is then computed automatically). For the clustering, we utilise betweenness centrality and modularity, as described in Section 3.1. We have already extended the modularity formula of Newman [New06] to evaluate a division in more than two clusters. An additional consideration would be whether, if at all, the formula could be adapted further to accept directed graphs, as in general for an edge (u, v), the edge in the opposite direction is not always present. Same holds for the definition of betweenness centrality, which uses the concept of shortest paths; those would also function differently in the case of a directed graph. We believe that aside from that, no other major considerations are necessary when handling directed graphs, as directed edges can simply be drawn in the standard manner (with an arrow towards the target vertex) without changing the rest of our pipeline.

Ordering the children in the cluster tree. Subsection 3.3.1 constructed the routing graph H in order to route and order the inter-cluster edges. There, an arbitrary order was chosen to connect the internal nodes around a cluster C_i in a cycle. This order was later on used in Subsection 3.3.3 when assigning initial coordinates to the clusters. Consider a cluster C_i and its children (in regards to the tree $T_{\mathscr{C}}$). A potential research question would be whether it is possible, given $T_{\mathscr{C}}$ and the not-yet-routed inter-cluster edges, to define an order in which the cluster's children are drawn, so as to minimise the number of crossings later on. For this, the routing algorithm of Pupyrev et al. [PNBH11] can be taken into consideration, or another one entirely, given that a MLCM-T1-suitable routing graph will be constructed once the child order is fixed.

Cluster size prediction. As elaborated in Subsection 3.3.5, the growth phases introduced in Subsection 3.3.4 tamper in general with the cluster coordinates and wedges assigned in Subsection 3.3.3. This leads to an optically displeasing bend in the drawn link bundles since the violating cluster's subtree is pushed further away from $(0, 0)$. The reason for the growth phases is that occasionally, there will be no gap between the already placed node arcs that is large enough to host the arc of the next node to be inserted. This raises the following question: can one define an initial circumference or radius for each cluster beforehand so that no growth phase is necessary? In particular, what is the smallest such value to avoid making the clusters unreasonably big? The resulting value would have to take not only the cluster's vertices into consideration, but also the orbit counts and potentially the labels. After the node insertion phase, it could also be possible to *shrink* the clusters instead (while retaining the gate nodes' direction) to eliminate any gaps. This works as the non-gate node arcs do not have fixed directions and a shrinking would not have any influence on the number of link crossings in the end.

Labeling alternatives. In Subsection 3.4.1, we solved the issue of providing enough space for each label by scaling the labels for each cluster just enough for them to fit. While we already discussed some alternatives and their drawbacks, we believe it would be an intriguing research question to delve deeper into this topic. In the best-case, an approach is developed that allows drawing labels for each node arc without having to re-scale the labels themselves.

Acknowledgements.

This thesis would not have been possible without the continuous support of my friends and family. They stood with me, provided emotional support and encouragement, and also gave feedback as readers or listeners that were not deeply familiar with the topic, thus motivating me to explain some concepts in greater detail than initially planned.

I also thank those couple of individuals who took from their precious time to proofread this thesis. Their constructive criticism and insightful comments were crucial in ensuring its accuracy and clarity.

Last but not least, I extend my tremendous gratitude to my supervisors, Tim Hegemann and Prof. Dr. Alexander Wolff at Chair I for Computer Science at the University of Würzburg, for the exceptional guidance and support I received while writing this work. Their expertise and willingness to share their knowledge, as well as the many fruitful discussions, played a pivotal role in refining the ideas presented here and resulted in a more well-rounded analysis and design.

Bibliography

[ABKS08] Evmorfia N. Argyriou, Michael A. Bekos, Michael Kaufmann, and Antonios Symvonis: Two polynomial time algorithms for the metro-line crossing minimization problem. In *Graph Drawing, 16th International Symposium, Revised Papers*, volume 5417 of *Lecture Notes in Computer Science*, pages 336–347. Springer, 2008, 10.1007/978-3-642-00219-9_33.

[ADM+19] Lorenzo Angori, Walter Didimo, Fabrizio Montecchiani, Daniele Pagliuca, and Alessandra Tappini: ChordLink: A new hybrid visualization model. In *Graph Drawing and Network Visualization - 27th International Symposium, Proceedings*, volume 11904 of *LNCS*, pages 276–290. Springer, 2019, 10.1007/978-3-030-35802-0_22.

[ADM+22] Lorenzo Angori, Walter Didimo, Fabrizio Montecchiani, Daniele Pagliuca, and Alessandra Tappini: Hybrid graph visualizations with ChordLink: Algorithms, experiments, and applications. *IEEE Transactions on visualization and computer graphics*, 28(2):1288–1300, 2022, 10.1109/TVCG.2020.3016055.

[Bae67] Ronald M. Baecker: *Planar representations of complex graphs.* 1967, 10.21236/ad0648133.

[BB04] Michael Baur and Ulrik Brandes: Crossing reduction in circular layouts. In *Graph-Theoretic Concepts in Computer Science, 30th International Workshop, Revised Papers*, volume 3353 of *Lecture Notes in Computer Science*, pages 332–343. Springer, 2004, 10.1007/978-3-540-30559-0_28.

[BDG+07] Ulrik Brandes, Daniel Delling, Marco Gaertler, Robert Görke, Martin Hoefer, Zoran Nikoloski, and Dorothea Wagner: On finding graph clusterings with maximum modularity. In *Graph-Theoretic Concepts in Computer Science, 33rd International Workshop, Revised Papers*, volume 4769 of *Lecture Notes in Computer Science*, pages 121–132. Springer, 2007, 10.1007/978-3-540-74839-7_12.

[BETT99] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis: *Graph Drawing: Algorithms for the Visualization of Graphs.* Prentice-Hall, 1999, ISBN 0-13-301615-3.

[BNT+24] Annika Bonerath, Martin Nöllenburg, Soeren Terziadis, Markus Wallinger, and Jules Wulms: Boundary labeling in a circular orbit. In Stefan

Felsner and Karsten Klein (editors): *32nd International Symposium on Graph Drawing and Network Visualization*, volume 320 of *LIPIcs*, pages 22:1–22:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024, 10.4230/LIPICS.GD.2024.22.

[BNUW06] Marc Benkert, Martin Nöllenburg, Takeaki Uno, and Alexander Wolff: Minimizing intra-edge crossings in wiring diagrams and public transportation maps. In *Graph Drawing, 14th International Symposium, Revised Papers*, volume 4372 of *Lecture Notes in Computer Science*, pages 270–281. Springer, 2006, 10.1007/978-3-540-70904-6_27.

[Bra01] Ulrik Brandes: A faster algorithm for betweenness centrality. *Journal of mathematical sociology*, 25(2):163–177, 2001, 10.1080/0022250x.2001.9990249.

[CLRS22] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein: *Introduction to algorithms.* MIT press, 3rd edition, 2022.

[CM84] William S. Cleveland and Robert McGill: Graphical perception: Theory, experimentation, and application to the development of graphical methods. *Journal of the American statistical association*, 79(387):531–554, 1984, 10.2307/2288400.

[DH96] Ron Davidson and David Harel: Drawing graphs nicely using simulated annealing. *ACM Trans. Graph.*, 15(4):301–331, 1996, 10.1145/234535.234538.

[DN09] Tim Dwyer and Lev Nachmanson: Fast edge-routing for large graphs. In *Graph Drawing, 17th International Symposium, Revised Papers*, volume 5849 of *Lecture Notes in Computer Science*, pages 147–158. Springer, 2009, 10.1007/978-3-642-11805-0_15.

[Ead84] Peter Eades: A heuristic for graph drawing. *Congressus numerantium*, 42(11):149–160, 1984, 10.61091/cn.

[Ead91] Peter Eades: *Drawing free trees.* International Institute for Advanced Study of Social Information Science, 1st edition, 1991. `https://vca.informatik.uni-rostock.de/~hs162/treeposter/scans/Eades1992.pdf`.

[EKSX96] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu: A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD-96)*, pages 226–231. AAAI Press, 1996.

[FLGC02] Gary William Flake, Steve Lawrence, C. Lee Giles, and Frans Coetzee: Self-organization and identification of web communities. *Computer*, 35(3):66–71, 2002, 10.1109/2.989932.

[FLM94] Arne Frick, Andreas Ludwig, and Heiko Mehldau: A fast adaptive layout algorithm for undirected graphs. In *Graph Drawing, DIMACS International Workshop, Proceedings*, volume 894 of *Lecture Notes in Computer Science*, pages 388–403. Springer, 1994, 10.1007/3-540-58950-3_393.

[FR91] Thomas M. J. Fruchterman and Edward M. Reingold: Graph drawing by force-directed placement. *Software: Practice and Experience*, 21(11):1129–1164, 1991, 10.1002/SPE.4380211102.

[Fre77] Linton C. Freeman: A set of measures of centrality based on betweenness. *Sociometry*, 40(1):35, 1977, 10.2307/3033543.

[FWD$^+$03] Jean-Daniel Fekete, David Wang, Niem Dang, Aleks Aris, and Catherine Plaisant: Interactive poster: Overlaying graph links on treemaps. In *Proceedings of the IEEE Symposium on Information Visualization Conference Compendium (InfoVis 03)*, pages 82–83. IEEE, 2003.

[GADM04] Sébastien Grivet, David Auber, Jean-Philippe Domenger, and Guy Melançon: Bubble tree drawing algorithm. In *International Conference on Computer Vision and Graphics, Proceedings*, volume 32 of *Computational Imaging and Vision*, pages 633–641. Springer, 2004, 10.1007/1-4020-4179-9_91.

[GDMT21] Emilio Di Giacomo, Walter Didimo, Fabrizio Montecchiani, and Alessandra Tappini: A user study on hybrid graph visualizations. In *Graph Drawing and Network Visualization - 29th International Symposium, Revised Selected Papers*, volume 12868 of *Lecture Notes in Computer Science*, pages 21–38. Springer, 2021, 10.1007/978-3-030-92931-2_2.

[GFC05] Mohammad Ghoniem, Jean-Daniel Fekete, and Philippe Castagliola: On the readability of graphs using node-link and matrix-based representations: a controlled experiment and statistical analysis. *Information Visualisation*, 4(2):114–135, 2005, 10.1057/PALGRAVE.IVS.9500092.

[Hac05] Stefan Hachul: *A potential field based multilevel algorithm for drawing large graphs*. PhD thesis, University of Cologne, Germany, 2005. `http://kups.ub.uni-koeln.de/volltexte/2005/1409/index.html`.

[HCH81] S. Mitchell Hedetniemi, E. J. Cockayne, and S. T. Hedetniemi: Linear algorithms for finding the jordan center and path center of a tree. *Transportation Science*, 15(2):98–114, 1981, 10.1287/trsc.15.2.98.

[Hdb05] Jeffrey Heer and danah boyd: Vizster: Visualizing online social networks. In *IEEE Symposium on Information Visualization (InfoVis 2005)*, pages 32–39. IEEE, 2005, 10.1109/INFVIS.2005.1532126.

[HF06] Nathalie Henry and Jean-Daniel Fekete: Matrixexplorer: a dual-representation system to explore social networks. *IEEE Transactions on visualization and computer graphics*, 12(5):677–684, 2006, 10.1109/TVCG.2006.160.

[HF07] Nathalie Henry and Jean-Daniel Fekete: Matlink: Enhanced matrix visualization for analyzing social networks. In *Human-Computer Interaction - INTERACT 2007, 11th IFIP TC 13 International Conference, Proceedings, Part II*, volume 4663 of *Lecture Notes in Computer Science*, pages 288–302. Springer, 2007, 10.1007/978-3-540-74800-7_24.

[HFB$^+$04] Daniel B. Horn, Thomas A. Finholt, Jeremy P. Birnholtz, Dheeraj Motwani, and Swapnaa Jayaraman: Six degrees of Jonathan Grudin: A social network analysis of the evolution and impact of CSCW research. In *ACM Conference on Computer Supported Cooperative Work, Proceedings*, pages 582–591. ACM, 2004, 10.1145/1031607.1031707.

[HFM07] Nathalie Henry, Jean-Daniel Fekete, and Michael J. McGuffin: Nodetrix: a hybrid visualization of social networks. *IEEE Transactions on visualization and computer graphics*, 13(6):1302–1309, 2007, 10.1109/TVCG.2007.70582.

[HK00] David Harel and Yehuda Koren: A fast multi-scale method for drawing large graphs. In *Graph Drawing, 8th International Symposium, Proceedings*, volume 1984 of *Lecture Notes in Computer Science*, pages 183–196. Springer, 2000, 10.1007/3-540-44541-2_18.

[HKKS03] John E. Hopcroft, Omar Khan, Brian Kulis, and Bart Selman: Natural communities in large linked networks. In *Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Proceedings*, pages 541–546. ACM, 2003, 10.1145/956750.956816.

[HLT$^+$20] Ge Huang, Yong Li, Xu Tan, Yuejin Tan, and Xin Lu: PLANET: A radial layout algorithm for network visualization. *Physica A: Statistical Mechanics and its Applications*, 539:122948, 2020, 10.1016/j.physa.2019.122948.

[HS00] Erez Hartuv and Ron Shamir: A clustering algorithm based on graph connectivity. *Information processing letters*, 76(4-6):175–181, 2000, 10.1016/S0020-0190(00)00142-3.

[HvW09] Danny Holten and Jarke J. van Wijk: Force-directed edge bundling for graph visualization. *Computer Graphics Forum*, 28(3):983–990, 2009, 10.1111/J.1467-8659.2009.01450.X.

[HW23] Tim Hegemann and Alexander Wolff: A simple pipeline for orthogonal graph drawing. In *Graph Drawing and Network Visualization, 31st International Symposium, Revised Selected Papers, Part II*, volume 14466 of *Lecture Notes in Computer Science*, pages 170–186. Springer, 2023, 10.1007/978-3-031-49275-4_12.

[IHB$^+$23] Alaul Islam, Tingying He, Anastasia Bezerianos, Bongshin Lee, Tanja Blascheck, and Petra Isenberg: Visualizing information on smartwatch faces: A review and design space. *Computing Research Repository*, abs/2310.16185, 2023, 10.48550/ARXIV.2310.16185.

[Jor69] Camille Jordan: Sur les assemblages de lignes. *Journal fur die reine und angewandte Mathematik*, 70:185–190, 1869, 10.1515/crll.1869.70.185.

[Kar93] David R. Karger: Global min-cuts in rnc, and other ramifications of a simple min-cut algorithm. In *Fourth Annual ACM/SIGACT-SIAM Symposium on Discrete Algorithms, Proceedings*, volume 93, pages 21–30. ACM/SIAM, 1993. `http://dl.acm.org/citation.cfm?id=313559.313605`.

[KK89] Tomihisa Kamada and Satoru Kawai: An algorithm for drawing general undirected graphs. *Information processing letters*, 31(1):7–15, 1989, 10.1016/0020-0190(89)90102-6.

[Kru56] 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, 10.1090/S0002-9939-1956-0078686-7.

[KSW23] Philipp Kindermann, Jan Sauer, and Alexander Wolff: The computational complexity of the chordlink model. *Journal of Graph Algorithms and Applications*, 27(9):759–767, 2023, 10.7155/JGAA.00643.

[KVV00] Ravi Kannan, Santosh S. Vempala, and Adrian Vetta: On clusterings - good, bad and spectral. In *41st Annual Symposium on Foundations of Computer Science*, pages 367–377. IEEE, 2000, 10.1109/SFCS.2000.892125.

[LPP$^+$06] Bongshin Lee, Cynthia Sims Parr, Catherine Plaisant, Benjamin B. Bederson, Vladislav Daniel Veksler, Wayne D. Gray, and Christopher Kotfila: Treeplus: Interactive exploration of networks with enhanced tree layouts. *IEEE Transactions on visualization and computer graphics*, 12(6):1414–1426, 2006, 10.1109/TVCG.2006.106.

[LY05] Chun-Cheng Lin and Hsu-Chun Yen: On balloon drawings of rooted trees. In *Graph Drawing, 13th International Symposium, Revised Papers*, volume 3843 of *Lecture Notes in Computer Science*, pages 285–296. Springer, 2005, 10.1007/11618058_26.

[MMRR13] Hassan Mahmoud, Francesco Masulli, Stefano Rovetta, and Giuseppe Russo: Community detection in protein-protein interaction networks using spectral and graph approaches. In *Computational Intelligence Methods for Bioinformatics and Biostatistics - 10th International Meeting, Revised Selected Papers*, volume 8452 of *Lecture Notes in Computer Science*, pages 62–75. Springer, 2013, 10.1007/978-3-319-09042-9_5.

[New03] Mark E. J. Newman: The structure and function of complex networks. *SIAM Rev.*, 45(2):167–256, 2003, 10.1137/S003614450342480.

[New06] Mark E. J. Newman: Modularity and community structure in networks. *Proceedings of the national academy of sciences*, 103(23):8577–8582, 2006, 10.1073/pnas.0601602103.

[NG04] Mark E. J. Newman and Michelle Girvan: Finding and evaluating community structure in networks. *Physical review E*, 69(2):026113, 2004, 10.1103/physreve.69.026113.

[NJW01] Andrew Y. Ng, Michael I. Jordan, and Yair Weiss: On spectral clustering: Analysis and an algorithm. In *Advances in Neural Information Processing Systems 14 [Neural Information Processing Systems: Natural and Synthetic,]*, pages 849–856. MIT Press, 2001. `https://proceedings.neurips.cc/paper/2001/hash/801272ee79cfde7fa5960571fee36b9b-Abstract.html`.

[PNBH11] Sergey Pupyrev, Lev Nachmanson, Sergey Bereg, and Alexander E. Holroyd: Edge routing with ordered bundles. In *Graph Drawing, 19th International Symposium, Revised Selected Papers*, volume 7034 of *Lecture Notes in Computer Science*, pages 136–147. Springer, 2011, 10.1007/978-3-642-25878-7_14.

[PR00] Seth Pettie and Vijaya Ramachandran: An optimal minimum spanning tree algorithm. In *Automata, Languages and Programming, 27th International Colloquium, Proceedings*, volume 1853 of *Lecture Notes in Computer Science*, pages 49–60. Springer, 2000, 10.1007/3-540-45022-X_6.

[Pri57] Robert Clay Prim: Shortest connection networks and some generalizations. *The Bell System Technical Journal*, 36(6):1389–1401, 1957, 10.1002/j.1538-7305.1957.tb01515.x.

[She67] Roger N Shepard: Recognition memory for words, sentences, and pictures. *Journal of verbal Learning and verbal Behavior*, 6(1):156–163, 1967, 10.1016/S0022-5371(67)80067-7.

[Shn96] Ben Shneiderman: The eyes have it: A task by data type taxonomy for information visualizations. In *1996 IEEE Symposium on Visual Languages, Proceedings*, pages 336–343. IEEE, 1996, 10.1109/VL.1996.545307.

[ST99] Janet M. Six and Ioannis G. Tollis: A framework for circular drawings of networks. In *Graph Drawing, 7th International Symposium, Proceedings*, volume 1731 of *Lecture Notes in Computer Science*, pages 107–116. Springer, 1999, 10.1007/3-540-46648-7_11.

[VD00] Stijn Van Dongen: *Graph clustering by flow simulation.* PhD thesis, University of Utrecht, 2000. `https://dspace.library.uu.nl/handle/1874/848`.

[vS06] Jivr'i vS'ima and Satu Elisa Schaeffer: On the np-completeness of some graph cluster measures. In *SOFSEM 2006: Theory and Practice of Computer Science, 32nd Conference on Current Trends in Theory and Practice of Computer Science, Proceedings*, volume 3831 of *Lecture Notes in Computer Science*, pages 530–537. Springer, 2006, 10.1007/11611257_51.

[WF94] Stanley Wasserman and Katherine Faust: *Social Network Analysis: Methods and Applications.* Cambridge University Press, 1st edition, 1994, ISBN 9780511815478, 10.1017/CBO9780511815478.

[WHPL10] Hao Wu, Jun He, Yijian Pei, and Xin Long: Finding research community in collaboration network with expertise profiling. In *Advanced Intelligent Computing Theories and Applications, 6th International Conference on Intelligent Computing, Proceedings*, volume 6215 of *Lecture Notes in Computer Science*, pages 337–344. Springer, 2010, 10.1007/978-3-642-14922-1_42.

[WS98] Duncan J. Watts and Steven H. Strogatz: Collective dynamics of 'small-world' networks. *Nature*, 393(6684):440–442, 1998, 10.1038/30918.

[ZMC05] Shengdong Zhao, Michael J. McGuffin, and Mark H. Chignell: Elastic hierarchies: Combining treemaps and node-link diagrams. In *IEEE Symposium on Information Visualization (InfoVis 2005)*, pages 57–64. IEEE, 2005, 10.1109/INFVIS.2005.1532129.