DIMACS Technical Report 2005-43

Interactive Navigation of Power-Law Graphs

by

James Abello
Teoma - ASK Jeeves Research,
Piscataway, NJ 08854
`jabello@ask.com`

Frank van Ham
Technische Universiteit Eindhoven,
P.O. Box 513, 5600 MB Eindhoven,
The Netherlands,
`fvham@win.tue.nl`

DIMACS is a collaborative project of Rutgers University, Princeton University, AT&T Labs–Research, Bell Labs, NEC Laboratories America and Telcordia Technologies, as well as affiliate members Avaya Labs, HP Labs, IBM Research, Microsoft Research, Stevens Institute of Technology, Georgia Institute of Technology and Rensselaer Polytechnic Institute. DIMACS was founded as an NSF Science and Technology Center.

ABSTRACT

We illustrate how iterated deletion of vertices of degree one, followed by biconnected graph decomposition constitute simple but powerful preprocessing steps that we can use to create suitable hierarchies on a graph. These hierarchies can then aid us substantially in the process of graph layout and navigation. The benefits of this approach are more apparent on sparse power-law graphs when a hierarchy tree is computed for each biconnected component via recursive clustering. Our sample data sets include an Autonomous System-level Internet topology with 11461 vertices and a 43,433 vertex graph detailing the static call dependencies of the Linux core.

1 Introduction

The average size of graphs under analysis in industry and science has grown well beyond the limits of what traditional graph drawing techniques can handle. In many cases simply laying out all nodes of the graph in a single image is not very effective, because of limits to human information processing abilities, nor very efficient, due to complexity of the algorithms involved.

Instead of confronting the user with a single layout of graphs on more than a thousand vertices, we propose the use of a hierarchical navigation scheme. That is, we construct a hierarchy of nodes on the original vertex set and allow the user to interactively expand different nodes in the hierarchy. Although this approach is not new (see for example [7, 10]) most previous approaches have largely avoided the problem of actually creating such a hierarchy when the graph is moderately large (say more than 10,000 vertices). Efficiently constructing such a hierarchy and using it to navigate moderately large graphs is the main focus of this paper. The methods described here are particularly suited for graphs in which the number of vertices with degree k is proportional to $k^{-\beta}$ with $\beta > 0$, referred in the literature as power-law graphs. Examples of power-law graphs include Internet topologies [8], communication networks [1] and software structures. When these graphs are sparse, have a low diameter and high clustering coefficient it pays off to use specialized hierarchy trees and to use targeted force-directed layout methods only at certain levels of granularity.

2 Clustering

Below we give a condensed definition of a hierarchy tree and views on a hierarchy tree. For a more exhaustive description see [2].

Given a connected graph $G(V, E)$, a hierarchy tree $T(V_T, E_T)$ is a tree where $Leaves(T) = V$. Two nodes $u, v \in V_T$ are called incomparable, denoted as $u \bowtie v$, if neither one is an ancestor of the other. We denote the set of leaves of the subtree rooted at $x \in V_T$ by T_x and define the multiplicity $m(u, v)$ of a node pair $u, v \in V_T$ as $\#\{(x, y) \in E | x \in Leaves(T_u) \wedge y \in Leaves(T_v)\}$.

A multilevel digraph then, is a graph $G_M(V_T, E_M)$ with edge set $E_M = \{(u, v) | u, v \in V_T \wedge u \bowtie v \wedge m(u, v) > 0\}$. A *macro-view* on G_M is a subgraph of G_M induced by a maximal set of incomparable nodes. We can use macro views as high level representations of G, assuming we can find a good hierarchy tree for G. In the next sections we refer to the internal nodes of T as clusters and present two efficient procedures for computing a suitable hierarchy tree T, given input graph G.

2.1 Peel Grouping

Because power-law graphs contain a relatively large number of vertices with degree one, a large portion of their local structure is tree-like. From a graph drawing perspective then, it does not make much sense to employ expensive algorithms such as force directed algorithms

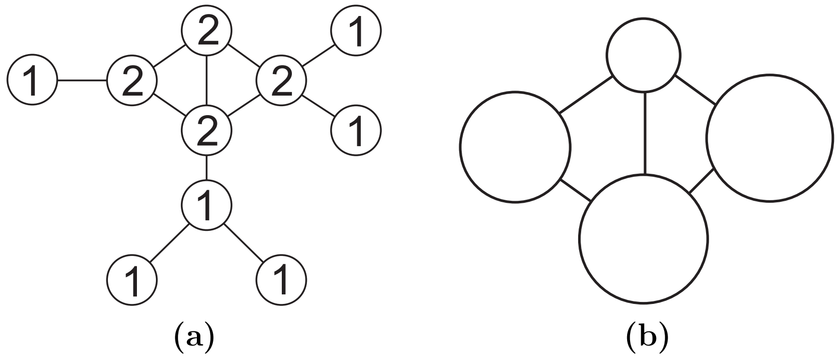


Figure 1: Grouping the peripheral forest of a graph. **(a)** Sample graph with peeling numbers indicated; **(b)** Peripheral trees contracted with their rootnodes

to compute a layout for these tree-like parts. We propose to use a linear time process known as *peeling* to identify these peripheral tree-like sections and group them together. If the user decides to expand such a tree group, we can employ much more efficient tree specific algorithms [9, 4] to provide a layout.

Peeling [11] is a time ordered process that maintains a vertex list of the graph sorted by the vertices' peel degree. Initially the peel degree of a node is equal to its degree. The peeling process visits the nodes in non-decreasing order of peel degrees exactly once, and for each node decreases the peel degrees of its direct unvisited neighbors, resulting in a permutation of the node list. The process keeps track of the maximal peel degree M encountered so far and assigns *peeling number* M to a vertex when it is visited. The peeling number of a graph is the maximum peeling number of its vertices.

For an arbitrary graph G, its peeling number is less than or equal to its maximum degree. Moreover, it follows from the above definitions that the peeling number of a graph is equal to one iff the graph is a forest. We call the forest induced by the set of vertices with peeling number one the peripheral forest of G. Since the peeling process can be implemented in time linear on the number of edges of G [3], we can identify the peripheral forest of G efficiently. If G is connected and if its peel number is strictly greater than one, then the subtrees in the peripheral forest of G must have at most one vertex adjacent to a vertex with peeling number higher than one. We place all the subtrees in the peripheral forest that are adjacent to the same higher peel numbered vertex r in the same subgroup along with r. Simply put, we delete all the vertices in the peripheral forest F of G and mark those vertices r in $V(G) \setminus F$ that have connections to F. We refer to this process as the removal of the peripheral forest.

2.2 A Biconnected Component Hierarchy Tree

A biconnected component of a graph is a maximal connected subgraph that cannot be broken into multiple components by removing a single node or edge. The biconnected components of a graph can be computed efficiently in $O(|V|+|E|)$. The collection of biconnected components have a standard natural tree associated with it that is called the biconnected decomposition tree [12]. Those vertices or edges whose removal disconnect the graph are called articulation points and bridges respectively.

Biconnected components (BCC's) provide a high level view of the structure of a graph. Articulation points and bridges are important to assess the structure of communication networks since their removal/failure disconnect the network. Unfortunately, large induced subtrees in a graph correspond to a large number of BCC's (one for each node in the tree). This makes it difficult to use BCC's information to aid the graph layout process. Power-law graphs are at a greater disadvantage here since they contain large peripheral forests. However, as in the previous section we first group all the subtrees in the peripheral forest and only then obtain the biconnected decomposition of the resulting graph. In this manner we obtain a substantial smaller number of BCCs without losing essential biconnectivity information.

The next step is to obtain a vertex partition from the edge partition provided by the biconnectivity decomposition. The trick is to replace each *articulation* vertex a that is shared among k biconnected components B_1, B_2, B_k, by a star centered at a new vertex a' with k spokes, where the i^{th} spoke connects a' to a new 'cloned' vertex named a_i and where the neighborhood of a_i is $\{a'\} \bigcup \{v : v \in B_i \wedge (a, v) \in E\}$. We refer to the a_is as clones of a and to a' as the central clone of a, see fig 2.

Notice that by introducing clones for each articulation point we obtain an enlarged graph with more vertices than the original one, but with disjoint BCCs. Deleting the central clones and identifying their associated clone vertices gives us back the original graph. The biconnected decomposition tree of this enlarged graph contains essentially the same biconnected information as the original graph with some redundancy added by the incorporation of the clone vertices. This provides us with a topological abstraction that is directly encodable by a hierarchy tree without committing to a particular global layout.

2.3 Graph Clustering

An embedding of the BCC's hierarchy tree introduced in the previous section can be used for high level navigation. In many cases however, this tree contains some nodes that represent moderately large BCCs with more than a thousand nodes. In this case we use a clustering method to recursively break down these large BCCs into smaller subgraphs. Our clustering algorithm of choice is MCL [6], because of its ability to compute relatively good clusters in short time for sparse graphs with high local density. Another advantage of MCL is that it does not require us to specify a desired number of clusters in advance. When the number of clusters computed by the MCL algorithm was larger than a small constant (say 20) we reapplied MCL recursively to a suitable weighted version of the clustered graph . Iterating this process results in a hierarchy of clusters based on the inherent cluster structure of the graph.

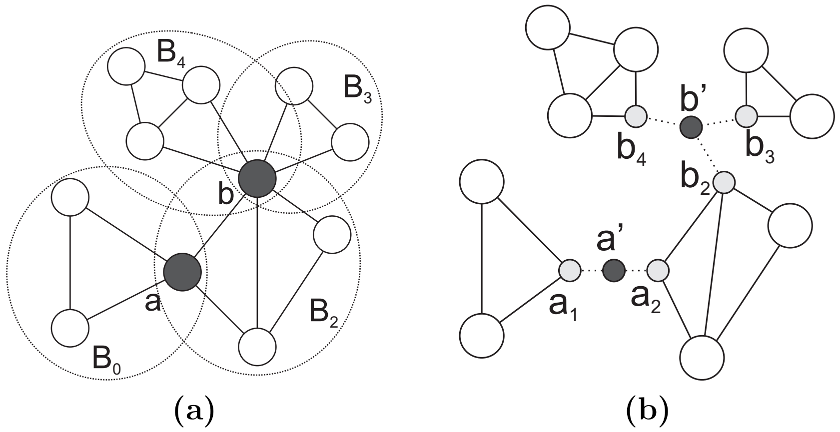


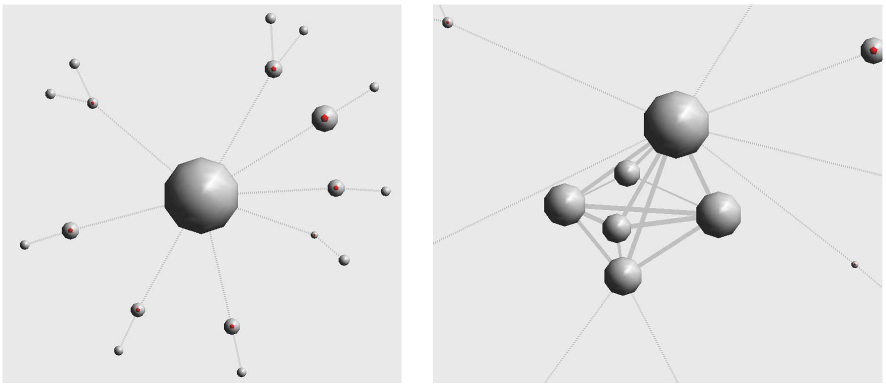
Figure 2: Effect of separating articulation points: **(a)** input graph with 4 biconnected components and 2 articulation points **(b)** cloned version, articulation points are split over their respective BCC's.

3 Application

We implemented the above procedures and applied them to a number of different graphs that are known to have power-law characteristics. The first one was a part of an Autonomous System-level Internet topology containing 11,461 nodes and 32,730 edges obtained from [5]. Grouping the peripheral forest as indicated in section 2 reduced this to a graph with 8110 nodes and 29,379 edges. Applying the biconnected components algorithm to this new graph resulted in a tree decomposition with 19 biconnected components and 18 macro edges. Figure 3 shows a layout of this clustered graph.

Articulation points are marked by a dot inside the cluster. Edges connecting articulation points to their cloned vertices in the BCC are indicated by dashed lines. The size of the glyphs is proportional to the size of the cluster or group and the width of the edges is proportional to their multiplicity. This image was created in just a few seconds of computation time on a Pentium IV 2.2 GHz.

From the picture we can see that the majority of nodes are contained in a central biconnected component. We split this BCC, by recursively applying MCL, resulting in 5 high level clusters. We then applied a force-directed algorithm to compute a layout for these clusters and position them inside the high level layout that we computed in the previous step, similar to [7]. A user can interactively navigate this graph by selecting any visible cluster for expansion. Our current interface computes a layout of the children of the selected cluster (either by a force directed layout or a radial layout in the case of the peripheral forest), animates the transition between the collapsed and the expanded view and zooms in to a close up of the expanded cluster.



a **b**

Figure 3: 11K node AS-level internet topology: **(a)** top-level BCC view, showing articulation points and their connections; **(b)** Largest central component was recursively split using MCL resulting in 5 medium sized clusters.

A second sample we applied this clustering scheme to was a 43,433 node graph detailing the static call dependencies of the Linux core. Applying peel grouping to this graph reduced it to 35,726 nodes. Both the peel and BCC processes took just under two minutes to complete, resulting in the image displayed in figure 4. The largest articulation points (one of them is circled) are heavily called functions such as `printk`.

Although we limited this exposition to power-law graphs, there are no theoretical objections as to why we should not apply the proposed scheme to non-regular sparse graphs. Provided we group the peripheral forest first, the biconnected component hierarchy tree gives us the essential macro connectivity structure of the graph. Applying recursive clustering afterwards helps us identify highly connected subgraphs.

4 Conclusions

We have presented two linear time preprocessing techniques that serve as a good starting point for creating a multilevel clustering of any graph. The first one uses peeling to identify peripheral tree structures and groups them at the bottom of the hierarchy. This first step singles out some sections of the graph that are suitable for much more efficient layout algorithms and reduces in turn the size of the graph that needs to be clustered.

The second one creates clusters at the top level of the hierarchy by computing biconnected components on the reduced graph. Just as computing the connected components of a graph should be the first step when analyzing it, we believe that grouping peripheral hierarchies followed by a biconnected component decomposition should be the second step to be applied before any further processing.

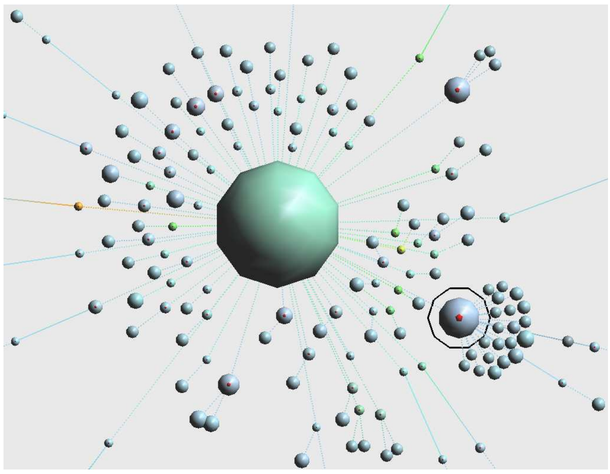


Figure 4: Clustering procedure applied to 43K node graph of the linux kernel.

References

[1] Abello, J., Buschbaum A. and Westbrook J. A Functional Approach to External Graph Algorithms. in Proc. 6th European Symp. on Algorithms (1998) pp 332–343

[2] Abello, J. and van Ham, F. Matrix zoom: A Visual Interface to Semi-External Graphs. In Proc. IEEE InfoVis (2004) pp 183–290

[3] Batagelj, V. and Mrvar, A. Pajek - Analysis and Visualization of Large Networks. In Jünger, M., Mutzel, P. (Eds.): Graph Drawing Software. Springer (series Mathematics and Visualization), (2003) pp 77-103

[4] Bernard, M.A. On the Automated Drawing of Graphs. In Proc. 3rd Carribean Conference on Combinatorics and Computing (1981) pp 43-55

[5] Chang, H. et al. Towards Capturing Representative AS-Level Internet Topologies. Computer Networks Journal, Volume 44, Issue 6, (2004) pp 737–755

[6] van Dongen, S. Graph Clustering by Flow Simulation, PhD thesis, Universiteit Utrecht (2000).

[7] Eades, P. and Huang, M.L. Navigating Clustered Graphs using Force-Directed Methods. in Journal Graph Algorithms and Applications vol 4 no 3 (2000) pp 157–181

[8] Kleinberg, J. et al. The Web as a Graph: Measurements, Models and Methods. in Proc. of the Int. Conf. on Combinatorics and Computing (1999) pp 1–18

[9] Reingold, E.M. and Tilford, J.S. Tidier drawing of Trees. IEEE Transactions on Software Engineering, SE-7(2) (1981) pp 223-228

[10] Schaffer, D. et al. Navigating Hierarchically Clustered Networks Through Fisheye and Full-zoom Methods. ACM Transactions on Computer-Human Interaction vol. 3 no. 2 (1996) pp 162–188

[11] Seidman, S.B. Network Structure and Minimum Degree. Social Networks **5** (1983) pp 269–287

[12] Tarjan, R.E. Depth first search and linear graph algorithms. SIAM Journal on Computing, 1(2) (1972) pp 146-160