Balanced Aspect Ratio Trees and Their Use for Drawing Very Large Graphs

C. A. Duncan*, M. T. Goodrich*, and S. G. Kobourov*

Center for Geometric Computing
The Johns Hopkins University

Abstract. We describe a new approach for cluster-based drawing of very large graphs, which obtains clusters by using binary space partition (BSP) trees. We also introduce a novel BSP-type decomposition, called the *balanced aspect ratio* (BAR) tree, which guarantees that the cells produced are convex and have bounded aspect ratios. In addition, the tree depth is $O(\log n)$, and its construction takes $O(n \log n)$ time, where n is the number of points. We show that the BAR tree can be used to recursively divide a graph into subgraphs of roughly equal size, such that the drawing of each subgraph has a *balanced aspect ratio*. As a result, we obtain a representation of a graph as a collection of $O(\log n)$ layers, where each succeeding layer represents the graph in an increasing level of detail. The overall running time of the algorithm is $O(n \log n + m + D_0(G))$, where n and m are the number of vertices and edges of the graph G, and $D_0(G)$ is the time it takes to obtain an initial embedding of G. In particular, if the graph is planar each layer is a graph drawn with straight lines and without crossings on the $n \times n$ grid and the running time reduces to $O(n \log n)$.

1 Introduction

In the past decade hundreds of graph drawing algorithms have been developed (e.g., see [5]), and research in methods for visually representing graphical information is now a thriving area with several different emphases. One general emphasis in graph drawing research is directed at algorithms that display an entire graph, with each vertex and edge explicitly depicted. Such drawings have the advantage of showing the global structure of the graph. A disadvantage, however, is that they can be cluttered for drawings of large graphs, where details are typically hard to discern. For example, such drawings are inappropriate for display on a computer screen any time the number of vertices is more than the number of pixels on the screen. For this reason, there is a growing emphasis in graph drawing research on algorithms that do not draw an entire graph, but instead partially draw a graph, either by showing high-level structures and allowing users to "zoom in" on areas of interest, or by showing substructures of the graph and allowing users to "scroll" from one area of the graph to another. Such approaches would be more suitable, for instance, for displaying very large

* This research partially supported by ARO under grant DAAH04-96-1-0013.

graphs, such as significant portions of the world wide web graph, where every web page is a vertex and every hyper-link is an edge.

A common technique used in the scrolling viewpoint context is the *fish-eye view* [12, 13, 21], which shows an area of interest quite large and detailed (such as nodes representing a user's web pages) and shows other areas successively smaller and in less detail (such as nodes representing a user's department and organization web pages). Fish-eye view drawings allow for a user to understand the structure of a graph near a specific set of nodes, but they often do not display global structures.

An alternate paradigm is to display global structure present in a graph by clustering smaller subgraphs and drawing these subgraphs as single nodes or filled-in regions. By grouping vertices together into *clusters* in this way we can recursively divide a given graph into layers of increasing detail, which can then be viewed in a top-down fashion or even in fish-eye view by following a single path in a cluster-based recursion tree. If clusters of a graph can be given as input along with the graph itself, then several authors give various algorithms for displaying these clusters in two or three dimensions [6, 7, 9, 10, 18]. If, as will often be the case, clusters of a graph are not given *a priori*, then various heuristics can be applied for finding clusters, say, using properties such as connectivity, cluster size, geometric proximity, or statistical variation [17, 19, 24]. Once a clustering has been determined, we can generate the layers in a hierarchical drawing of the graph, with the layer depth (i.e., number of layers) being determined by the depth of the recursive clustering hierarchy. This approach allows the graph to be represented by a sequence of drawings of increasing detail. As illustrated by Eades and Feng [6], this hierarchical approach to drawing large graphs can be very effective. Thus, our interest in this paper is to further the study of methods for producing good graph clusterings that can be used for graph drawing purposes.

We feel that a good clustering algorithm and its associated drawing method should come as close as possible to achieving the following goals:

1. *Balanced clustering*: in each level of the hierarchy the size of the clusters should be about the same.
2. *Small cluster depth*: there should be a small number of layers in the recursive decomposition.
3. *Convex cluster drawings*: the drawing of each cluster should fit in a simple convex region, which we call the *cluster region* for that subgraph.
4. *Balanced aspect ratio*: cluster regions should not be too "skinny".
5. *Efficiency*: computing the clustering and its associated drawing should not take too long.

The goal of this paper is to study how well we can achieve these goals for very large graph drawings using clustering. Previous algorithms optimize one or more of the above criteria at the expense of some of the rest. Our goal is to try to optimize all of them. Our approach relies on creating the clusters using binary space partition (BSP) trees, defined by recursively cutting regions with straight lines.

1.1 BSP Tree Based Clustered Graph Drawing

The main idea behind the use of a BSP tree to define clusters is very simple. Given a graph $G = (V, E)$, where $n = |V|$ and $m = |E|$, we can use any existing method to embed it, provided that method places vertices at distinct points in the plane (e.g., see [5, 14, 25]). For example, if G is planar we can use any existing method for embedding G in the plane such that vertices are at grid points, and edges of the graph are straight lines that do not cross [4, 8, 22, 23, 26]. Once the graph drawing is defined, we build a binary space partition tree on the vertices of this drawing. Each node v in this tree corresponds to a convex region R of the plane, and associated with v is a line that separates R into two regions, each of which are associated with a child of v. Thus, any such BSP tree defined on the points corresponding to vertices of G naturally defines a hierarchical clustering of the nodes of G. Such a clustering could then be used, for example, with an algorithm like that of Eades and Feng [6], who present a technique for drawing a 3-dimensional representation of a clustered graph.

The main problem with using BSP trees to define clusters for a graph drawing algorithm is that previous methods for constructing BSP trees do not give rise to clustered drawings that achieve the design goals listed above. For example, the standard *k-d* tree and its derivates (e.g., see [11, 20]), which use axis-parallel lines to recursively divide the number of points in a region in half, maintain every criteria but the balanced aspect ratio. Likewise, quad-trees and fair-split trees (e.g., see [3, 20]), which always split by a line parallel to a coordinate axis to recursively divide the area of a region more or less in half, maintain balanced aspect ratio but can have a depth that is $\Theta(n)$. In our graph drawing application, aesthetics are extremely important, as "fat" regions appear rounder and a series of skinny regions can be distracting. But depth is also important, for a deep hierarchy of clusterings would be computationally expensive to traverse and would not provide very balanced clusters. The balanced box-decomposition tree of Arya et al [1, 2] has $O(\log n)$ depth and has regions with good aspect ratio, but it sacrifices convexity by introducing holes into the middle of regions, which makes this data structure less attractive for use in clustering for graph drawing applications. Indeed, to our knowledge, there is no previous BSP-type hierarchical decomposition tree that achieves all of the above design goals.

1.2 The Balanced Aspect Ratio (BAR) Tree

In this paper we present a new type of binary space partition tree that is better suited for the application of defining clusters in a large graph. Our data structure, which we call the *balanced aspect ratio* (BAR) tree, is a BSP-type of decomposition tree that has $O(\log n)$ depth and creates convex regions with bounded aspect ratio (i.e., so-called "fat" regions). The construction of the BAR tree is very similar to that of a *k-d* tree, except for two important differences:

1. The BAR tree allows for one additional cut direction: a $45°$-angled cut.
2. Rather than insisting that the number of points in a region be cut in half at every level, the BAR tree guarantees that the number of points is cut roughly

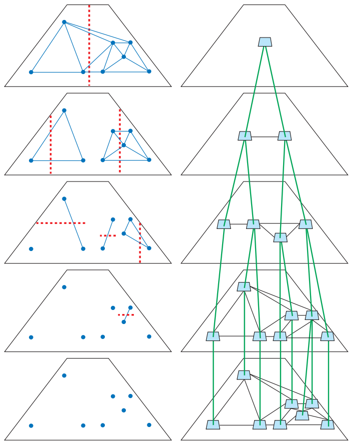

Fig. 1. The multi-level view of a graph with each cluster represented by a single node. Note the relationship between the cuts on the left and the clusters on the right.

in half every two levels, which is something that does not seem possible to do with either a *k-d* tree or a quadtree (or even a hybrid of the two) while guaranteeing regions with bounded aspect ratios.

In short, the BAR tree is an $O(\log n)$-depth BSP-type data structure that creates fat, convex regions. Thus, the BAR tree is "balanced" in two ways: on the one hand, clusters on the same level have roughly the same number of points, and, on the other hand, each cluster region has a bounded aspect ratio.

We show that a BAR tree achieves this combined set of goals by proving the existence of a cut, which we call a *two-cut*. A two-cut might not reduce the point size by any amount but maintains balanced aspect ratio and ensures the existence of a subsequent cut, which we call a *one-cut*, that both maintains good aspect ratio *and* reduces the point size by at least two-thirds. In Section 3, we formally define one- and two-cuts and describe how to construct a BAR tree.

1.3 Our Results for Cluster-Based Graph Drawing

In Section 4, we show how to use the BAR tree in a cluster-based graph drawing algorithm. The Very Large Graph Drawing (`VLGD`) algorithm runs in $O(n \log n + m + D_0(G))$ time, where n and m are the number of vertices and edges in the graph G and $D_0(G)$ is the time to embed G. If the graph is planar, the algorithm introduces no edge crossings and the running time reduces to $O(n \log n)$.

The algorithm creates a hierarchical cluster representation of a graph, with balanced clusters at each layer and with cluster depth $O(\log n)$. Each cluster region has a *balanced aspect ratio*, guaranteed by the BAR tree data structure. In the actual display of the clustered graph we represent the clusters either with their convex hulls, with a larger region defined by the BSP tree, or simply with a single node (see Figure 1).

```
create_clustered_graph(T, G)
    K ← G
    for i = k downto 0
        obtain G_i from K
        shrink K
    add the edges of T
    return C
```

Fig. 2. Creating the clustered graph.

2 Using a BSP Tree for Cluster Drawing

Let $G = (V, E)$ be the graph that we want to draw, where $|V| = n$ and $|E| = m$. The goal of our `VLGD` algorithm is to produce a 3-dimensional representation C of the embedded graph G given a BSP tree T. Similar to [6] we define the *clustered graph* $C = (G, T)$ to be the graph G, and the BSP tree T, such that the vertices of G coincide with the leaves of T. An internal node of T represents a cluster, which consists of all the vertices in its subtree. All the nodes of T at a given depth i represent the clusters of that level. A *view at level* i, $G_i = (V_i, E_i)$, consists of the nodes of depth i in T and a set of representative edges. The edge $(u, v) \in E_i$ if there is an edge between a and b in G, where a is in the subtree of u and b is in the subtree of v. In addition, each node $u \in T$ has an associated region, corresponding to the partition given by T.

We create the graphs H_i in a bottom-up fashion, starting with H_k and going all the way up to H_0, where $k = \texttt{depth}(T)$. Define the combinatorial graph $K = (V(K), E(K))$, where initially $V(K) = \{u \in T : \texttt{depth}(u) = k\}$ and $E(K) = E(G)$. Notice that K is well defined since the leaves of T are exactly the vertices of G.

At each new level i we perform a *shrinking* of K. Suppose $u, v \in V(K)$, and $\texttt{parent}(u) = \texttt{parent}(v)$. We replace the pair by their parent and remove the edge (u, v) if it exists. We also remove any multiple edges that this operation may have created and maintain for each surviving edge a pointer to the original edge in G. Thus a *shrinking* of K consists of all such operations, necessary to transform K into a representation of G at one higher level in the tree T.

At each level G_i is a subgraph of G with certain edges removed. In addition, the z-coordinate of a vertex $v \in V_i$ is equal to i, that is, all the vertices in G_i are embedded in the plane given by $z = i$. To obtain G_i from G_{i+1}, for $i = 0, \ldots, k-1$, we use the combinatorial graph K from level $i + 1$. Initially $E_i = E_{i+1}$. We then perform a shrinking of K and while removing an edge from K we remove its associated edge from E_i.

This algorithm (see Figure 2) runs in $O(n \cdot \texttt{depth}(T) + m)$ time. Depending on the type of BSP tree used, we can maintain most but never all of the desired properties. For example, if T is a k-d tree the cluster regions do not have balanced aspect ratios. We next describe how to construct a BSP tree which satisfies all of our goal criteria.

3 The BAR tree

Let us now discuss in detail the definition of our particular BSP-type decomposition tree, the BAR tree, and its construction. We begin with some general definitions. Let a line have a *canonical slope* if it forms an angle with the x-axis that is $0°$, $90°$, or $45°$ (referred to as the x, y, z directions). Define a *canonical cut* to be a cut with canonical slope, and a *canonical hexagon* to be a hexagon whose sides have canonical slope and possibly degenerate length. The *aspect ratio* of a canonical hexagon R is $\mathtt{ar}(R) = \max(\mathtt{diam}_i(R))/\min(\mathtt{diam}_j(R))$ $\forall i, j \in \{x, y, z\}$, where $\mathtt{diam}_x(R)$ is the distance between the x-slopes of R region in the L_m metric, and similarly for the others. Let the *maximum aspect ratio* be a constant, say $\alpha = 6$. A region R has *balanced aspect ratio* if $\mathtt{ar}(R) \leq \alpha$ and a *unbalanced aspect ratio* otherwise.

For simplicity of arguments and notations, we will use the L_∞ metric because all other metrics allow for "skinnier" rectangles to be produced. Using this metric, the length $||z|| = ||z||_\infty$ of a diagonal cut (with slope $45°$) is simply the vertical (or horizontal) distance between its endpoints. Also, the distance between two diagonal lines is one half of the vertical (or horizontal) distance between them. Throughout the paper we often refer to regions with balanced and unbalanced aspect ratios as *fat* and *skinny* regions, respectively.

Suppose we are given a point set $\mathcal{S}$ in the plane, $|\mathcal{S}| = n$, and an initially square region R containing $\mathcal{S}$. We now introduce the BAR tree data structure, which divides R into cells such that the following properties are guaranteed:

- Every cell is convex.
- Every cell has balanced aspect ratio.
- Every cell has no more than a constant number of points.
- The tree has $O(n)$ nodes.
- The depth of the tree is $O(\log n)$.

The structure is straightforward and reminiscent of the original *k-d* tree. Recall that in a *k-d* tree, every node i in the tree represents a cell region R_i and an axis-parallel cut partitioning R_i into two subregions. The leaves of the tree are cells with a constant number of points. In general, each cut divides the region into two roughly equal halves, and thus the tree has $O(\log n)$ depth and uses $O(n)$ space. However, if the vast majority of the points are concentrated close to any particular corner of the region, no constant number of axis-parallel cuts can effectively reduce the size of the point set and maintain good aspect ratio. This is a serious problem with many applications and with ours in particular. As a result, an extensive amount of research has been dedicated to improving and analyzing the *average* case performance of *k-d* trees and its derivatives often concentrating on trying to maintain some form of balanced aspect ratio.

3.1 Constructing the BAR tree

In this section we show how to construct a BAR tree T using the vertices of an embedded graph G, an aspect ratio parameter α and a balance parameter β.

```
create_BAR_tree(R, α, β)
    if an (α, β)-balanced one-cut l, exists in R
        (R_1, R_2) ← l(R)
    else let s be an α-balanced two-cut in R
        (R_1, R_2) ← s(R)
        let s' be an (α, β)-balanced one-cut in R_1
        (R_1, R_3) ← s'(R_1)
    recurse on R_i
```

Fig. 3. Creating the BAR tree.

Before we proceed any further with proving the existence of the necessary one- and two-cuts, we must first formally define these cuts.

Definition 1. Let R be a convex region with aspect ratio α or less and n points inside it and let β be the balance parameter. Define a *one-cut* to be a canonical cut which divides R into two subregions R_1 and R_2 such that for $i = 1, 2$:

1. R_i contains no more than βn points.
2. R_i has aspect ratio α or less.

We say that a region R is *one-cuttable* if it has balanced aspect ratio and there exists a one-cut for R.

Definition 2. Let R be a convex region with aspect ratio α or less and n points inside it, and let β be the balance parameter. Define a *two-cut* to be a canonical cut, s, which divides R into two subregions R_1 and R_2 such that:

1. R_1 contains no less than βn points.
2. R_2 has aspect ratio α or less.
3. R_1 is one-cuttable (call that cut s').

In other words, the sequence of two cuts, s and s', results in three regions each with balanced aspect ratio and each containing no more than βn points. A region R is *two-cuttable* if it has balanced aspect raion and there exists a two-cut for R.

We are now ready to give the pseudo-code for the construction of a BAR tree (see Figure 3). Here we use the notation $(R_1, R_2) \leftarrow l(R)$ as a shorthand for cutting the region R with a cut l which results in subregions R_1 and R_2. Given α and β, a cut is α-balanced if the subregions produced have aspect ratio less than or equal to α. Similarly, a cut is β-balanced if the subregions produced have less than or equal to a β fraction of the points in the original region. Finaly, a cut is (α, β)-balanced if satisfies both conditions.

3.2 Two-cut existence theorem

Here we consider the correctness and performance our algorithm for constructing a BAR tree. In the creation of the BAR tree we rely on the existence of either

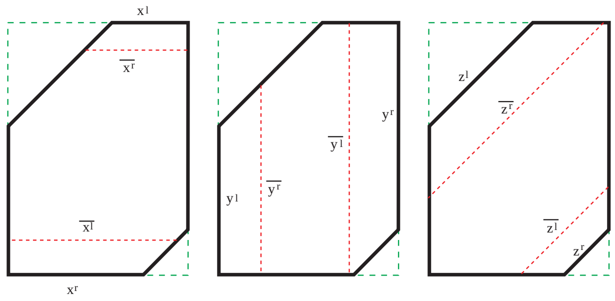

Fig. 4. The labels on the sides of a general canonical hexagon and the maximizing cuts in the respective directions.

a one-cut or a two-cut. Recall that we only make canonical cuts (cuts with x-angle $0°, 90°, 45°$, referred to as the x, y, z directional cuts of "canonical" slope). Thus the regions we create are canonical hexagons. We state the following trivial lemmas and postpone the detailed proofs for the final version of the paper.

Lemma 1. *Given a canonical hexagon R with balanced aspect ratio, consider a line l with canonical slope and coincident to a (degenerate) side of R. Let's sweep l upwards (inside R) until the region above l, P, has either maximum aspect ratio or is empty. Call the region P* maximized in the direction of l *and the defining cut, $\overline{l}$, a* maximal-cut. *If P is not empty and we continue sweeping, the region above $\overline{l}$ will have an unbalanced aspect ratio until it becomes empty.*

Corollary 1 *If the region P is maximized in the direction of l, $\texttt{ar}(P) = \alpha$ and* $\min(\texttt{diam}_x(P), \texttt{diam}_y(P), \texttt{diam}_z(P)) = \texttt{diam}_l(P)$.

Lemma 2. *If there exists a continuum of canonical cuts completely covering a region, R, in which each cut always yields two subregions of balanced aspect ratio, there exists a one-cut in R, for $\beta \geq 2/3$.*

Here we are extending a well-known geometric result which states that we can find a bisector in any direction. In our case, we have a continuum of cuts (in no more than three directions) which "cover" the entire region, while always maintaining balanced aspect ratios.

What are some specific regions guaranteed to be one-cuttable? We describe two such regions that are needed for guaranteeing the existence of two-cuts. Before we look at some specific regions, let us define the sides of the regions we will be dealing with. Let $x^l, x^r, y^l, y^r, z^l, z^r$ be the sides of a general canonical hexagon. Furthermore, let $\overline{x}^l, \overline{x}^r, \overline{y}^l, \overline{y}^r, \overline{z}^l, \overline{z}^r$ be the cuts that maximize the regions in the respective directions as shown in Figure 4.

Lemma 3. *Canonical isoceles trapezoidal* (CIT) *regions and canonical right-angle trapezoidal* (CRT) *regions are one-cuttable regions (see Figure 5).*

Proof. Let $d_i = \texttt{diam}_i(R)$. We analyze each region individually.

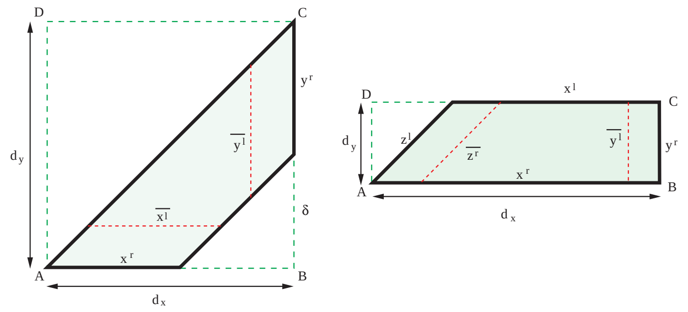

Fig. 5. CIT and CRT regions.

(a) *CIT regions:* In a CIT with aspect ratio α define $\delta = ||z^r|| = d_x - ||x^r||$. From the shape of the region, we know that $d_x = d_y$ and $x^r = y^r$. Recall that in the L_∞ metric, $d_z = x^r/2 = y^r/2$. Since the object has aspect ratio α, we have $d_x/d_z = \alpha$ and $2d_x/x^r = \alpha$, and so $2d_x = \alpha(d_x - \delta)$, which implies that $d_x = \alpha\delta/(\alpha - 2)$. We can sweep from $\overline{x}^l$ and from $\overline{y}^l$ covering the entire region while maintaining balanced aspect ratio for $\alpha \geq 4$. Thus, such an isoceles trapezoid has a one-cut.

(b) *CRT regions:* In a CRT with aspect ratio α, we have $d_x = \alpha d_y$ and so $x^l\alpha/(\alpha - 1) = x^r = d_x = \alpha d_y$, which implies $d_y = x^l/(\alpha - 1)$. We can sweep from $\overline{y}^l$ and from $\overline{z}^r$ in the same manner for $\alpha \geq 4$. Thus, such a right-angled trapezoid has a one-cut.

It is easy to construct examples where a region R cannot be divided into roughly equal portions by a simple single cut (be it axis-alligned or diagonal-type cut). However, the following theorem shows that using a two-cut followed by a one-cut we can in fact divide the cells into convex subregions with the desired properties (balanced aspect ratio and less than a constant fraction of the region's points). We omit the proof due to lack of space.

Theorem 1. (Two-Cut Existence Theorem) *A canonical hexagon R which does not have a one-cut must have a two-cut.*

Theorem 2. *Given a point set $\mathcal{S}$ in the plane, we can construct a BAR tree representing a decomposition of the plane into regions in $O(n \log n)$ time.*

Proof. Notice that a one-cut or a two-cut in any of the three canonical directions can be found in $O(n)$ time and that the depth of the tree is $O(\log n)$.

4 Using a BAR tree for Cluster Based Drawing

Let $G = (V, E)$ be the graph that we want to draw. Once we obtain the embedding of G, using whatever algorithm is most appropriate for the graph, we

```
VLGD(G, α, β)
    embed(G)
    T ← create_BAR_tree(G, α, β)
    H ← create_clustered_graph(T, G)
    display(H)
```

Fig. 6. Main algorithm.

associate with the graph the smallest bounding square, R, which we call G's *cluster region*. Using the embedding and its cluster region, we create the BAR tree T, as described above. Each node $u \in T$ maintains $\texttt{region}(u)$, $\texttt{cluster}(u)$, and $\texttt{depth}(u)$. Here $\texttt{cluster}(u)$ is the subgraph of G which is properly contained in $\texttt{region}(u)$. Recall that the depth of the tree T is $k = O(\log n)$. In our application of the tree structure to cluster-based graph drawing, we want every leaf to be at the same depth. Therefore, we propagate any leaf not at the maximum depth down the tree until the desired depth is reached. This is merely conceptual and does not require any additional storage space or change in the tree structure.

Using the tree T, we create the clustered graph C, which consists of k layers. Each layer is an embedded subgraph of G along with the regions and clusters obtained from T. The layers are connected with vertical edges which are simply the edges in T. The other inputs to VLGD are the aspect ratio parameter α and the balance parameter, β. Here, α determines the maximal aspect ratio of a cluster region in C, and β determines the cluster balance, the ratio of a cluster's size to its parent's. For a summary of the operations, see Figure 6.

Lemma 4. *A call to* $\texttt{VLGD}(G, \alpha, \beta)$ *for* $\alpha = 6$, $\beta = 2/3$ *results in* $2/3$*-balanced clustering with aspect ratio less than or equal to* 6 *and cluster depth* $O(\log n)$*.*

Proof. By construction, the clusters are β-balanced and the cluster depth is equivalent to the depth of T. For $\alpha \geq 6$ and $\beta \geq 2/3$ the depth is $O(\log_{1/\beta} n)$.

Theorem 3. *For* $\alpha \geq 6$, $\beta \geq 2/3$, *algorithm* VLGD *creates a* $2/3$*-balanced clustered graph* C *in* $O(n \log n + m + D_0(G))$ *time.*

Proof. The proof follows directly from the construction of the algorithm and previous statements about the running time of each component.

Once we obtain the clustered graph C, we can display it as a 3-dimensional multi-layer graph representing each cluster by either the the convex hull of its vertices or by its associated region in the BAR tree. Along with the clustered graph C we can display a particular cluster with more details. Thus we provide the global structure using the clustered graph and the local detail using the individual clusters.

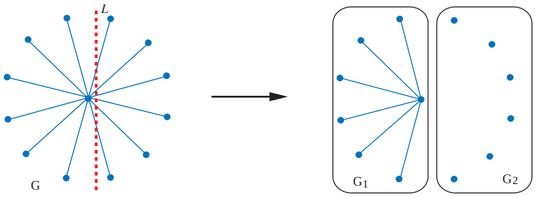

Fig. 7. Graph G with an inherently large cut. Any cut L which maintains an β-balance between the clusters, where $1/2 \leq \beta < 1$, cuts $O(n)$ edges.

4.1 Planar Graphs

When the graph G is planar, we are able to show a few special properties of our clustered drawings.

Theorem 4. *If G is planar, for $\alpha \geq 6$, $\beta \geq 2/3$, algorithm* `VLGD` *creates a $2/3$-balanced clustered graph C in $O(n \log n)$ time. Moreover, C is embedded with straight lines and no crossings on the $n \times n \times k$ grid, where $k = O(\log n)$.*

Proof. We begin with a planar grid embedding with straight-line edges [4, 8, 22] and then the original layer, G_k, is planar. Since each successive layer is a proper subgraph of the previous layer, it too must be planar and drawn without edge crossings.

It is possible to have an edge cross a region that is does not belong to. Moreover, it is possible to have an edge cross the convex hull of a cluster that it does not belong to. If we represent a cluster by the convex hulls of its connected components, however, there will be no such crossings. Thus, if we could guarantee that each cluster is connected or has a small number of connected components, the display of the graph can be improved even further. Alternatively, we can redefine the clusters at each level to be the connect components of vertices inside each cluster region of the BAR tree. With this definition of clusters we could then use the algorithm of Eades and Feng [6] to produce a new clustered embedding of the planar graph so as to have no edge or region crossings.

4.2 Extensions

Throughout this paper we do not discuss the cut sizes produced by our algorithm, that is the number of edges intersected by a cut line in the BAR tree. In some applications it is important that the number of such edges cut be as small as possible. There exist graphs, however, that do not allow for "nice" cuts of small size. Consider the *star* graph G on Figure 7. Any cut, which maintains a β-balance between the two subgraphs it produces, intersects $O(n)$ edges. If the balance parameter is $\beta = 1/2$, the cut contains $\lfloor \frac{n}{2} \rfloor$ edges. As this example shows, we cannot hope to guarantee cut sizes better than $O(n)$. Still, if the given graph has a small cut then we would like to find a small cut as well.

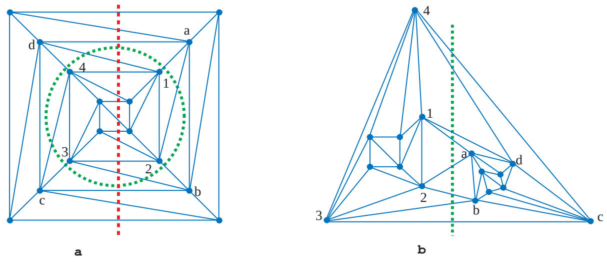

Fig. 8. The graph in part (a) has no β-balanced line cut of size better than $O(n)$ but it does have a cycle cut (the dotted circle) of size $O(1)$. We can transform the graph in (a) to the graph in (b) by taking one of the faces crossed by the cycle as the outer face. Note that in (b) the cycle cut has become a line and its size is $O(1)$.

Minimizing the cut size violates two of our five criteria, namely, speed and convexity. First of all, looking for the best β-balanced cut is a computationally expensive operation, and while it can be done in polynomial time, it is not hard to see that it cannot be done in linear time. In addition, the best β-balanced cut may not preserve the convex cluster drawing property that `VLGD` maintains, which may result in new edge crossings in our clustered graph.

Our algorithm does not guarantee that it will find the optimum β-balanced cut but we can modify the BAR tree construction so that we find locally optimal cuts. Here are some of the possible criteria that we can use in choosing among the potential cuts: minimize cut size, minimize connected components resulting from a given cut, minimize aspect ratio, maximize β-balance.

These criteria can also be combined in various ways to produce desired scoring methods. In finding such optimal cuts, it is important to note that a one-cut, if available, might not always be a better choice over a potential two-cut. Yet again, a two-cut that minimizes the cut size may have no subsequent one-cut that does not cut many more edges. Thus, it may be reasonable to go two levels in evaluating possible scores instead of choosing greedily.

5 Conclusion and Open Problems

In this paper we present a straightforward and efficient algorithm for displaying very large graphs. The `VLGD` algorithm optimizes cluster balance, cluster depth, aspect ratio and convexity. Our algorithm does not rely on any specific graph properties, although various properties can aide in performance, and produces the clustered graph in a very efficient $O(n \log n + m + D_0(G))$ time.

The embedding of the cluster graph is determined in the very first step of our algorithm. Unfortunately, it is possible that the initial embedding is not the best one. In fact, as shown on Figure 8, G may have a minimum β-balanced cut of size $O(n)$ or $O(1)$, depending on the embedding. While it is still true that some graphs may always have cuts of size $O(n)$ (for example, the star graph,

Figure 7), we would like to minimize the cut whenever we can. It is an open question whether it is possible to determine the optimal embedding, one that yields the minimum β-balanced cuts.

Another open question is related to the separator theorems of Lipton and Tarjan [15] and Miller [16]. Is it possible given a 2-connected planar graph G to always produce $O(\sqrt{dn})$ β-balanced cuts, where d is its maximum degree, and n is the number of vertices? If so, can we find an embedding for the resulting clustered graph which preserves efficiency, cluster balance, cluster depth, convexity, and guarantees good aspect ratio and straight-line drawings without crossings?

Acknowledgements

We would like to thank Rao Kosaraju and David Mount for their helpful comments regarding the balanced aspect ratio tree.

References

1. S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest neighbor searching. In *Proc. 5th ACM-SIAM Sympos. Discrete Algorithms*, pages 573–582, 1994.
2. Sunil Arya and David M. Mount. Approximate range searching. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, pages 172–181, 1995.
3. P. B. Callahan and S. R. Kosaraju. A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields. *J. ACM*, 42:67–90, 1995.
4. H. de Fraysseix, J. Pach, and R. Pollack. Small sets supporting Fary embeddings of planar graphs. In *Proc. 20th Annu. ACM Sympos. Theory Comput.*, pages 426–433, 1988.
5. G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. Algorithms for drawing graphs: an annotated bibliography. *Comput. Geom. Theory Appl.*, 4:235–282, 1994.
6. P. Eades and Q. W. Feng. Multilevel visualization of clustered graphs. *Lecture Notes in Computer Science*, 1190:101–??, 1997.
7. P. Eades, Q. W. Feng, and X. Lin. Straight-line drawing algorithms for hierarchical graphs and clustered graphs. *Lecture Notes in Computer Science*, 1190:113–??, 1997.
8. I. Fary. On straight lines representation of planar graphs. *Acta Sci. Math. Szeged.*, 11:229–233, 1948.
9. Q.-W. Feng, R. F. Cohen, and P. Eades. How to draw a planar clustered graph. *Lecture Notes in Computer Science*, 959:21–??, 1995.
10. Q.-W. Feng, R. F. Cohen, and P. Eades. Planarity for clustered graphs. *Lecture Notes in Computer Science*, 979:213–??, 1995.
11. J. H. Friedman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw.*, 3:209–226, 1977.
12. George W. Furnas. Generalized fisheye views. In *Proceedings of ACM CHI'86 Conference on Human Factors in Computing Systems*, Visualizing Complex Information Spaces, pages 16–23, 1986.

13. K. Kaugars, J. Reinfelds, and A. Brazma. A simple algorithm for drawing large graphs on small screens. In R. Tamassia and I. G. Tollis, editors, *Graph Drawing (Proc. GD '94)*, volume 894 of *Lecture Notes in Computer Science*, pages 278–281. Springer-Verlag, 1995.
14. R. J. Lipton, S. C. North, and J. S. Sandberg. A method for drawing graphs. In *Proc. 1st Annu. ACM Sympos. Comput. Geom.*, pages 153–160, 1985.
15. R. J. Lipton and R. E. Tarjan. Applications of a planar separator theorem. *SIAM J. Comput.*, 9:615–627, 1980.
16. G. L. Miller. Finding small simple cycle separators for 2-connected planar graphs. Report 85-336, Dept. Comput. Sci., Univ. Southern California, Los Angeles, CA, 1985.
17. Frances J. Newbery. Edge concentration: A method for clustering directed graphs. In *Proceedings of the 2nd International Workshop on Software Configuration Management*, pages 76–85, Princeton, New Jersey, October 1989.
18. S. C. North. Drawing ranked digraphs with recursive clusters. In *Graph Drawing '93, ALCOM International Workshop PARIS 1993 on Graph Drawing and Topological Graph Algorithms*, September 1993.
19. Sablowski and Frick. Automatic graph clustering. In *GDRAWING: Conference on Graph Drawing (GD)*, 1996.
20. H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, MA, 1990.
21. M. Sarkar and M. H. Brown. Graphical fisheye views. *Commun. ACM*, 37(12):73–84, 1994.
22. W. Schnyder. Embedding planar graphs on the grid. In *Proc. 1st ACM-SIAM Sympos. Discrete Algorithms*, pages 138–148, 1990.
23. S. K. Stein. Convex maps. *Proc. Amer. Math. Soc.*, 2:464–466, 1951.
24. K. Sugiyama and K. Misue. Visualization of structural information: Automatic drawing of compound digraphs. *IEEE Trans. Softw. Eng.*, 21(4):876–892, 1991.
25. W. T. Tutte. How to draw a graph. *Proceedings London Mathematical Society*, 13(3):743–768, 1963.
26. K. Wagner. Bemerkungen zum vierfarbenproblem. *Jahresbericht der Deutschen Mathematiker-Vereinigung*, 46:26–32, 1936.