Fast Edge-Routing for Large Graphs

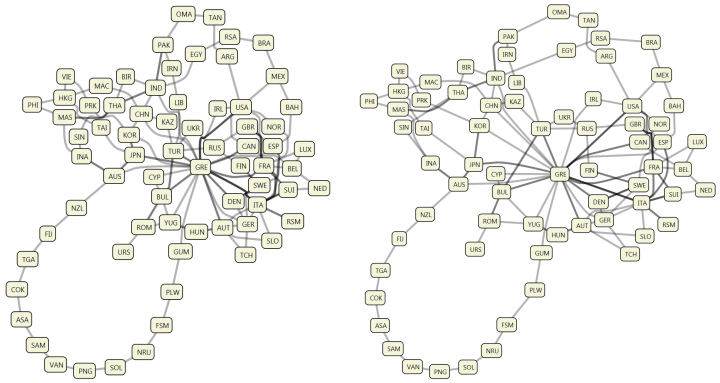
Tim Dwyer and Lev Nachmanson

Microsoft Research,
Redmond, USA,
{t-tdwyer,levnach}@microsoft.com

Abstract. To produce high quality drawings of graphs with nodes drawn as shapes it is important to find routes for the edges which do not intersect node boundaries. Recent work in this area involves finding shortest paths in a tangent-visibility graph. However, construction of the full tangent-visibility graph is expensive, at least quadratic time in the number of nodes. In this paper we explore two ideas for achieving faster edge routing using approximate shortest-path techniques.

1 Introduction

Most graphs that people need to visualize have nodes with associated textual or graphical content. For example, in UML class diagrams the nodes are drawn as boxes with textual content describing the class attributes or methods. In metabolic pathway diagrams nodes representing chemical compounds may have long textual labels or a graphic representation of the molecular structure. If edges that are not directly connected to a particular node are drawn over that node then the label may be obscured. Alternately, if edges are drawn behind a node then the reader may erroneously assume a connection to the node. Routing edges *around* nodes can avoid this ambiguity.



(a) Spanner graph routes with 20° cones.

(b) KD-Tree routing

Fig. 1. The "Olympic Torch Relay" graph from the GD'08 competition.

Some layout algorithms, such as the level scheme for directed graphs or the topology-shape-metrics approach for orthogonal graph drawing (see [1]) consider edge routing as an integral step in the layout process. However, the popular force-directed family of layout algorithms for general undirected graphs do not usually consider routing edges around node hulls; except perhaps as a post-processing step (e.g. Gansner and North [9]). Recent work such as [6, 7] has proposed force-directed methods which are able to preserve the topology of a given edge routing, but a feasible initial routing must still be found using a standard routing algorithm. As described in Section 2, for graphs with hundreds of nodes, the quadratic (in the number of nodes) or worse cost of constructing the visibility graph can be too slow, especially for interactive applications where the layout is changing significantly from iteration to iteration.

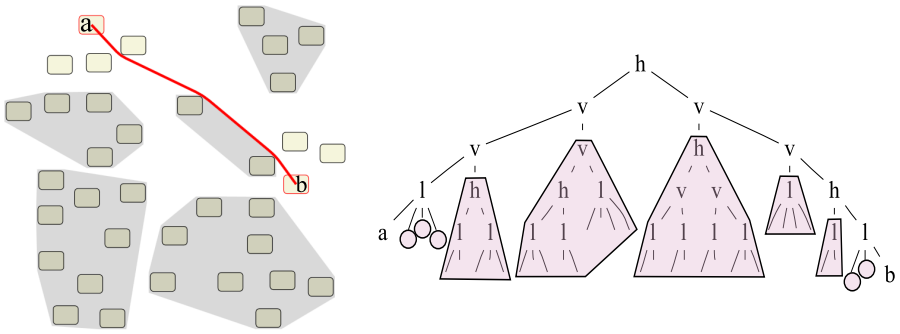
In this paper we present two approaches to achieve faster routing using *approximate* shortest paths. The first approach uses a spatial decomposition of the nodes, moving them slightly to obtain strictly disjoint convex hulls around groups of nodes, and then computing visibility graphs over these composite hulls rather than individual nodes. The second approach generates a sparse visibility-graph *spanner*. The two techniques are complementary, that is they can be used together to obtain even faster routing.

2 Related Work

Dobkin et al. [4] introduced visibility-graph methods for shortest-path edge routing into graph-drawing applications. They also considered the problem of fitting splines to the piecewise-linear path to obtain smooth curves.

Freivalds [8] gives a novel approach which treats edge routing as a problem of finding a low-cost path across a continuous cost function defined over the drawing area. A grid simplification is used so that the cost of routing one edge is $O(L^2 \log L)$ where L is the length of the path in grid units. The method is slow but is noteworthy in that adding additional routing criteria, such as perpendicular crossings between edges and slight offsets between collinear edges, is very easy.

Wybrow et al. [13] explored an efficient incremental implementation of a tangent-visibility graph for interactive graph manipulation or editing, for example, adding or removing a single node. However, their method is still $O(n^2 \log n)$ running time for n nodes in the static case. Faster algorithms for static construction of a visibility graph exist, but they are intricate and the asymptotic complexity improvement is not clear cut. For example, Ghosh and Mount [10] give an $O(E + V \log V)$ time algorithm for constructing a visibility graph with E edges over a set of obstacles with V vertices. Note that the usual tangent-visibility graph construction is not sensitive to the number of vertices V but rather to the number of obstacles n, and that the visibility graph may contain $O(n^2)$ edges.



(a) The edge routing

(b) The corresponding KD-tree with compound obstacles enclosed.

Fig. 2. An edge between two nodes a and b, routed using a tangent-visibility graph over a simplified set of obstacles and the corresponding KD-tree. The labels indicate: 'h'-a horizontal split internal node; 'v'-a vertical split internal node; and 'l'-a KD-tree leaf-node.

3 A spatial-decomposition routing scheme

The most expensive part of the routing schemes described above is the $O(n^2 \log n)$ construction of the tangent-visibility graph over n nodes with convex boundaries. Therefore, the first new idea we explore in this paper is a scheme for routing over simpler visibility graphs using a spatial partitioning scheme. The intuition is to replace groups of nodes (especially those that are far from the end nodes of the edge being routed) with their convex hulls, thus reducing the number of obstacles to consider in construction of the visibility graph. For example, see Figure 2.

To achieve this we need to obtain a recursive spatial partitioning of the nodes such that the convex hulls of the nodes in each partition are not overlapping with their siblings in the partition hierarchy. To be precise, recursive application of a spatial partitioning to nodes positioned in the plane gives a tree structure where each tree node at level k in the tree has children on level $k+1$. We use $desc(T)$ to denote the set of all leaf nodes (the original nodes in our graph) that are contained in a particular tree node T. We require that for any tree node U at level k in the tree, the convex hull of $desc(U)$ must not overlap with the convex hull of $desc(V)$ for any other tree node V also at level k, i.e. a sibling of U.

In order to achieve a reasonable asymptotic complexity, we also require that the tree be balanced. Obtaining such a tree for a given arrangement of nodes may be difficult or impossible. However, if we are willing to allow a little adjustment of node positions then we can enforce separation of siblings in a balanced KD-tree partitioning [2].

3.1 KD-tree Partitioning

Our spatial-decomposition routing scheme begins with a starting configuration of nodes obtained with any layout algorithm, the examples in the paper were

arranged using a fast-force directed approach. The bounding boxes of nodes can be initially overlapping as overlaps are removed by the first step, see Section 3.2. We build a KD-tree structure for these initial node positions as follows.

Fig. 2 shows an example of routing around simplified convex hulls and the KD-tree used to generate this routing. The KD-tree has *internal* nodes and *leaf* nodes, where an internal node has two child KD-tree nodes and a leaf node has two copies of the list of nodes from our original graph, one sorted by x-position, the other sorted by y-position. The KD-tree is built by initially constructing a single leaf node with lists containing all original graph nodes. We then recursively split the leaves either horizontally or vertically across the median element in the appropriate sorted list, and insert a new internal node as parent of these new leaves in the emerging hierarchy. We follow Lauther [11] in choosing to do a horizontal split if the bounding box of the elements in the leaf node is wider than tall, and vice versa otherwise, in order to keep the aspect ratio of leaf bounding boxes roughly square. We continue splitting until leaves are all smaller than some arbitrary bucket-size B. Initially sorting the n graph nodes by x- and y-position takes $O(n \log n)$ time. The sortedness of lists for new leaf nodes can be maintained by copying them in order from their parents. The tree is balanced since we always split across the median element, so $O(\log n)$ splits are performed. Thus, construction of the KD-tree requires $O(n \log n)$ time.

3.2 Removing overlaps

The routing scheme that follows requires that nodes in the KD-tree do not overlap their siblings. We first remove overlaps between the B children of each leaf node in the KD-tree, i.e. the original graph nodes. A number of methods for effectively resolving overlaps between rectangular bounding boxes exist. We use the quadratic-programming based method of [5] since we find that it leads to relatively little displacement of nodes from their starting positions.

Next we must remove overlaps between the bounding boxes of the children of internal nodes. Each internal node i has two children as the result of a split. If the split was horizontal then we resolve overlap horizontally. That is, if the amount of horizontal overlap $o_h = rightSide(leftChild(i)) - leftSide(rightChild(i)) > 0$, then we translate $leftChild(i)$ by $-o_h/2$ and $rightChild(i)$ by $o_h/2$. We resolve overlap in the same manner vertically if i was constructed with a vertical split. All internal nodes are processed in this way, proceeding bottom-up.

Since we move each (graph) node up to $\log n$ times, the running time of this overlap removal step is $O(n \log n)$.

3.3 Computing convex hulls

The next step is to compute convex hulls around the descendents of each internal node. Again, this is computed bottom up. It is possible to compute the convex hulls of all internal nodes in the KD-Tree in $O(n \log n)$ total time using the linear time hull merging method of Preparata and Hong [12]. However, we use a naïve application of Graham Scan to calculate internal node hulls of the points in

child hulls in $O(n \log^2 n)$ total time since the overall complexity of edge routing is dominated by the computation of visibility graphs anyway. In the sequel, $hull(i)$ refers to the precomputed convex hull of internal node i.

3.4 Simplified Visibility Graphs

Using the KD-tree of non-intersecting convex hulls described above we are able to construct a simplified visibility graph for all edges between a particular pair of leaves. Procedure *leaf-obstacles* returns a list of obstacles for any two leaves u and v in the KD-Tree T.

leaf-obstacles(u, v, T)
  $U \leftarrow$ the set of nodes of the shortest path between u and v in the KD-tree
  $w \leftarrow$ the lowest common ancestor of u and v
  $H \leftarrow \{hull(sibling(i)) | i \in U \setminus \{u, v, w\}\}$
return $H \cup \{hull(c) | c \in children(u) \cup children(v)\}$

Where $sibling(i)$ returns the sibling of internal node i and $children(u)$ returns the original graph nodes that are children of leaf node u.

Lemma 1. *Procedure* leaf-obstacles *returns* $O(\log n)$ *obstacle hulls.*

Proof. If the maximum bucket size $B = 1$ *then the height of the balanced KD-tree* T *is* $\log n$ *in which case the hulls returned by* leaf-obstacles *are just the siblings of the ancestors of* u *and* v *up to the lowest common ancestor. The worst case is that the lowest common ancestor of* u *and* v *is the root of* T*, resulting in* $2 \log n$ *obstacles. In practice we use* $B \approx 10$ *which results in up to* $2(\log n - \log B + B)$*, i.e. also* $O(\log n)$ *for* $B << n$. □

3.5 Routing edges

First we group edges by the (unordered) pair of leaves in KD-Tree T of their end nodes. For each group of edges between KD-Tree leaves u and v we generate the visibility graph over $leaf\text{-}obstacles(u, v, T)$. This takes time $O(\log^2 n \log \log n)$.

4 A sparse visibility-graph spanner

An alternative approach for edge routing in a large graph that we explore is the one suggested by [3]. This approach uses so called Yao graphs which are built by using fans of cones. A fan of cones is constructed at each vertex of an obstacle and only one edge of the visibility graph is chosen per cone. The resulting spanner graph contains only $O(\frac{\pi n}{\alpha})$ edges where n is the number of vertices of the graph and α is the cone angle. In spite of the graph sparseness for every $\epsilon > 0$ one can choose angle α such that for each shortest path in the full visibility graph the length of the corresponding shortest path in the spanner is at most $(1 + \epsilon)$ of the length of the former [3]. We construct the spanner graph by a direct method rather than by following the suggestion of [3] to build the

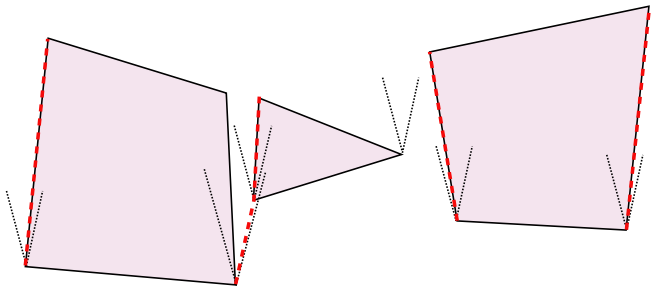


Fig. 3. Bold-dashed segments depict the edges created by one sweep.

conic Voronoi diagram first. We have not found the details of this algorithm for building such a spanner graph in [3] or in literature.

The technique presented here is a sweepline algorithm. One sweep finds edges inside cones constructed with the inner angle α and the bisector pointing to the direction of the sweep. By performing $\frac{\pi}{\alpha}$ sweeps we cover all possible edge directions from 0 to π. The input for the algorithm is the set P of convex mutually disjoint polygonal obstacles, the angle α and a vector representing the cone bisector direction (sweep direction). We say that a vertex u is visible from vertex v if $u \neq v$ and the line segment uv does not intersect the interior of a polygon in P. This way the neighbor vertices of an obstacle are visible to each other, and a side of an obstacle can be taken as an edge of the visibility graph. For the sake of simplicity the following explanation assumes further that the bisector coincides with the vector $(0, 1)$ and therefore the sweepline is horizontal and processing is from bottom to top. For points $a = (a_x, a_y)$ and $b = (b_x, b_y)$ we say the *cone distance* between them is $|b_y - a_y|$. Let V be the set of the vertices of P. For $v \in V$ we denote by C_v the cone with the apex at v, bisector $(0, 1)$, and angle α and by Vis_v the set of vertices from $V \cap C_v$ which are *visible* from v. For each $v \in V$ with $Vis_v \neq \emptyset$ the algorithm finds $u \in Vis_v$ which is closest to v in the cone distance, and adds the edge (v, u) to the spanner graph, see Fig. 3.

The algorithm works by processing events as the sweepline moves up. There are the following types of events (see Fig. 4(a)):

lowest vertex at the leftmost lowest vertex of an obstacle;

left vertex at a vertex that can be reached by a clockwise walk on obstacle edges starting from the vertex of a *lowest vertex* event and stopping at the rightmost highest vertex;

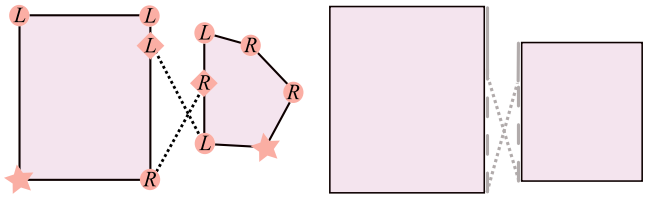
right vertex at a vertex that can be reached by a counterclockwise walk starting at the vertex of a *lowest vertex* event and stopping at the vertex before the rightmost highest vertex;

left intersection at the lowest intersection of a left cone side and an obstacle;

right intersection at the lowest intersection of a right cone side and an obstacle;

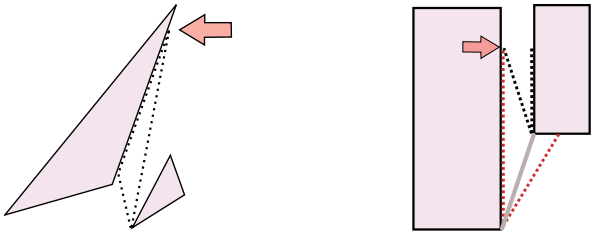
cone closure see below.

Events are kept in a priority queue Q with those sited at the lowest y-coordinate taking highest priority. For each cone participating in a sweep we keep pointers to its left and right sides. A cone side can be a default cone side,



(a) Events (b) Cone sides

Fig. 4. (a) Stars are *lowest vertex* events; circles labeled L and R are *left* and *right vertex* events; diamonds labeled L and R are *left* and *right intersection* events (b) Dotted lines show the default cone sides, Dashed lines - broken cone sides that are created at vertex events, solid lines - broken cone sides that are created at intersection events.



(a) Cone closure event (b) Missing broken side

Fig. 5. (a) The arrow points to a *cone closure* event. After such an event the cone is discarded. (b) The lower cone is discarded after discovering the vertex that completes the grey edge. The cone's left broken side is removed from LCS and the intersection marked by the arrow is not detected.

i.e. a ray starting at the cone apex at angle $\pm\frac{\alpha}{2}$ to the y-axis, or it could be a ray along an obstacle side if the cone is partially obscured by the obstacle as demonstrated by Fig. 4(b). We call such a cone side a *broken side*. Now we can define a *cone closure* event as an event happening when a broken side intersects a default cone side of the same cone, see Fig. 5(a). For a broken side we keep a pointer to its default cone side.

4.1 Balanced trees of active cone sides and obstacle segments

During the sweep we maintain a set of active cones. An active cone with its apex at a vertex is constructed at the vertex event. It is discarded when completely obstructed by an obstacle or when a visible vertex is discovered inside of the cone. We keep left cone sides of the active cones in a balanced tree LCS, and their right cone sides in a balanced tree RCS. The processing order guarantees that no two default active left (right) sides intersect. We also know that the cone side that we search for, insert into or remove from the tree must intersect the sweepline. This allows us to define the following order between cone sides a and b where x is the intersection of a with the sweepline. If x is to the left

of b then $a < b$, else if x is to the right of b then $a > b$. Otherwise, if a and b are both broken sides, then compare the default cone sides they point to, else we are comparing a broken side and a cone side; in LCS the default left cone side is less than the broken cone side, and in RCS the default right cone side is greater than the broken side. When looking for element a in a tree, inserting, or removing the element, we only need to compare a with elements of the tree and no other comparison is done. This observation will help us to overcome a difficulty arising later.

Trees LCS and RCS serve to find active cones "seeing" a vertex. Another function of the trees is calculation of intersection events. However, as shown at Fig. 5(b), a broken side containing an intersection event site can be removed before the intersection is found. To work around this we maintain two additional balanced trees called LS and RS. The members of these trees are called active obstacle segments; they are line segments connecting two adjacent vertices of an obstacle which are intersected by the current sweepline. Members of LS are segments traversed on the clockwise walk from the lowest vertex to the top of the obstacle and the remaining obstacle segments are members of RS. An active segment is added to the tree when its low vertex is processed and removed when its top is processed. Elements of LS (RS) are called left (right) active segments. The order of the segments in LS and RS is defined by the x-coordinates of the intersections of the segments and the sweepline. We call a segment *almost horizontal* if the absolute value of the difference between the y-coordinates of its start and end point is less than some small positive ϵ set in advance. Almost horizontal segments are not included in LS and RS since their intersections with the sweepline are not well defined, since we assume that the sweepline is horizontal. The order is well defined since the obstacles are disjoint.

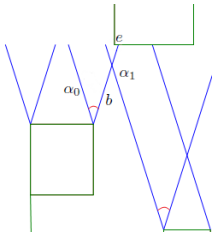


Fig. 6. FindConesSeeingEvent: Cone side b is the first in RCS not to the left of e. Side a first takes the value of a_0 then a_1. The two cones marked by an arc-segment "see" e.

4.2 Algorithm description

The main loop of the algorithm is described below.

Sweep (P, α, bisector)
  initialize queue Q by all *lowest vertex* events
  while Q is not empty
    $e \leftarrow$ pop event from Q
    ProcessEvent(e)

Routine **ProcessEvent** proceeds according to the event type. If the event is a *cone closure* then we discard the cone. However, it can happen that the cone of a *cone closure* event has been discarded earlier when a visible vertex was found

inside the cone. To handle this we keep a Boolean flag associated with a cone and set it to true when the cone is removed. In the case of a *cone closure* event if flag is not set we remove the cone; that is, its left and right sides are removed from LCS and RCS respectively.

We describe in detail only the event handlers **LeftVertexEvent** and **LeftIntersectionEvent** since other events are symmetric, with the exception of *lowest vertex*. We explain this exception below.

LeftVertexEvent(e)
  move the sweepline to the site of e
  FindConesSeeingEvent(e)
  CloseConesByHorizontalSegment(e)
  remove from LS the segment incoming into e clockwise
  AddConeAtLeftVertex(e)

The procedure **FindConesSeeingEvent** finds all cones that "see" the site of e, creates the corresponding edges and discards the cones, see Fig. 6.

FindConesSeeingEvent(e)
  $b \leftarrow$ the first right cone side in RCS which is not to left of e
  if b exists
    $a \leftarrow$ the left side of the cone of b
    while a is defined and a is not to the right of e
      create the edge from the apex of the cone of a to the vertex and remove the cone
      $a \leftarrow$ the successor of a in LCS

Procedure **CloseConesByHorizontalSegment** handles the case when the obstacle side going clockwise and ending at the vertex of event v is almost horizontal. It finds all cones obstructed by the segment. Since by this time we have removed all cones "seeing" v or "seeing" the start of the segment, every cone with a side intersecting the segment is completely obstructed by it. Tree RCS, for example, can be used to find all such cones in an efficient manner.

In **AddConeAtLeftVertex** we try to create a cone, enqueue events, and add a segment to LS. Let v be the vertex of e, and u is the next one on the obstacle in the clockwise order. We enqueue a left vertex event for u if it is not below v. The cone is created at v only when it is not completely obscured by the obstacle. The left side of the cone is a default side: for this side we look for the intersection with the last segment of RS to the left of v. If the right cone side is a default cone side we look for the intersection of it with the first segment of LS to the right of v. If the right side of the cone is an obstacle side we check for its intersection with the last segment of RCS which is to left of v. If the cone is created we add new cone left (right) side to LCS (RCS). If the segment $[v, u]$ is not almost horizontal and points to the left of the default right cone side starting from v then the segment is added to LS as a left active segment. An obstacle segment which does not point to the left of the default right cone side is not inserted into LS since no default right cone with the apex different from v can intersect it without first intersecting the obstacle at some other segment.

At a lowest vertex event we do almost all the work of a left vertex event and a right vertex event. However, since the lowest vertex is the first one examined on the obstacle we do not try to close cones by the horizontal obstacle segments adjacent to the vertex. When processing a right vertex event we enqueue the next right vertex event only in the case when the segment from the event vertex to

the next vertex going counterclockwise on the obstacle is not almost horizontal; this way we avoid processing a top vertex of an obstacle twice.

Let us describe the way **LeftIntersectionEvent** works. This procedure deals with the intersection of a default left cone side and a right obstacle segment.

```
LeftIntersectionEvent(e)
    c ← the cone side of e
    x ← the intersection point of e
    s ← the obstacle segment of e
    u ← the top point of s
    if the cone of c is not removed
        if segment [x, u] is almost horizontal
            remove the cone of c
            move the sweepline to the event site
        else
            RemoveFromTree(c, LCS)
            move the sweepline to the event site
            t ← new broken side [x, u]
            replace c by t in the cone and insert t into LCS
            m ← the successor of t in LCS
            if m exists and intersects t
                enqueue the new left intersection event
            if t intersects the cone right side and the intersection point is below u
                enqueue a new cone closure event at the intersection point
```

RemoveFromTree, in most cases, trivially removes the cone side from the tree. However, it can happen that the sweepline passes through the intersection point of an intersection event before we start processing it. In this case the removal might fail since the tree is not ordered correctly. The remedy here is to lower the sweepline temporarily by some value (but not lower than the start of the cone side being removed), remove the cone side and then restore the sweepline. The comparison of the cone sides still works since the lowered sweepline intersects the first cone side under comparison, see Fig.7.

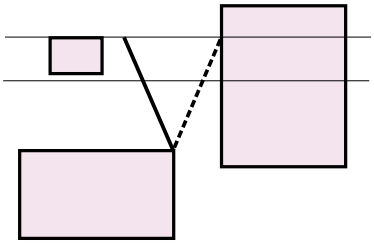


Fig. 7. Lowering the sweepline temporarily when removing the cone side at an intersection event. The right cone side (dashed) is being removed.

4.3 Performance of the sweep

Let n be the number of vertices of obstacles from P. The number of events is $O(n)$ since there are n vertices producing not more than n cones and each cone creates at most three events: two intersection events and one closure event. The trees never have more than n elements each. Each search on the trees takes $O(\log n)$ steps. Operations "successor" or "predecessor" on the trees also take $O(\log n)$ steps. In **FindConesSeeingEvent** and in **CloseConesByHorizontalSegment** we walk the tree by moving to the cone side successor until some condition holds. We can potentially make $O(n \log n)$ steps per call. However, for each processed cone side we remove the corresponding cone, so the routines cannot make more then $O(n \log n)$ steps during the whole algorithm run. Therefore the overall number of steps of the algorithm is $O(n \log n)$.

5 Spline refinement

At the final stage of our routing algorithm we "beautify" the spline. The detailed discussion of this stage is beyond the scope and space limitations of this paper, but on a very high level we do the following steps; *shortcutting*, *relaxation* and *fitting*. In shortcutting we try to skip each internal vertex of the shortest path by removing it and checking that the path still does not intersect the interior of an obstacle. Intersections are checked efficiently using a binary space partitioning. In relaxation we modify the path in such a way that it does not touch the obstacles anymore. In fitting we inscribe cubic Bezier segments into the corners of the shortest path. We have not carefully proven asymptotic complexity of these steps but in practice we find only a fraction of the full routing time is spent in refinement.

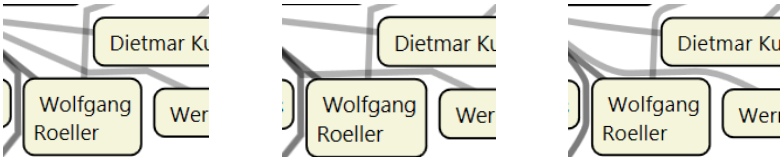
6 Experimental results

We tested routing over various combinations of spanner visibility graphs and KD-tree partitioning for several different graphs of very different sizes. Please see our appendix for more detailed results, but in summary, we find that the two methods proposed in this paper are complementary or can be used in isolation to achieve significant speed-up. For a large graph with 1138 nodes and 1458 edges shortest-path edge routing over the standard tangent-visibility graph took around 95 seconds. Routing over a spanner visibility graph with 10° cones reduced this time to 43 seconds, including time spent in spline refinement. With 45° cones, this was further reduced to 34 seconds. Increasing cone size was found to increase the longest edge length - by up to 7% for 45° cones, however the short-cutting step in our spline refinement phase was very effective at keeping average edge lengths relatively short. At a cursory glance the quality of the spanner-visibility graph routing together with refinement is close to the optimal shortest path routing. Routes that are slightly longer than necessary (for example following the side of an obstacle when a more direct route is possible) are only noticeable with careful inspection, e.g. see Fig. 8.

Adding the KD-tree routing scheme was found to add a further, very significant, speed-up. Using a 45° cone spanner as well as KD-tree, routing the 1458 edges of our largest graph took only 5 seconds (compared to 95 seconds optimal routing). The extra "spreading-out" of nodes due to the spatial partitioning scheme, and the resultant increase in edge length (around 20% on average), was noticeable (e.g. see Fig. 1), but less so for the very large graph.

7 Conclusion and Further Work

This paper represents the first attempt of which we are aware of using a spanner visibility graph scheme in routing of graph drawings. We achieve very significant speed-up with only marginal degradation in route quality so in future we intend to use it by default with a largish cone-size of 30° for all routing. The only



(a) Optimal (b) 30° cone spanner (c) Spline smoothing

Fig. 8. Detail from routing over the GD'08 "Companies" contest graph. (a) shows the optimal shortest path routing (b) an edge that follows the side of a shape rather than taking the optimal shortest path when routed using a 30° cone spanner (c) spline smoothing makes this path seem less bad. Even so, such non-optimal routes are relatively rare thanks to short-cutting (see Sec. 5).

disadvantage of the spanner visibility graph scheme is that it is quite complicated to implement. However, in this paper we have given more implementation details than we have found in the literature.

The KD-tree routing scheme is novel as far as we are aware. This gave us very significant speed improvement and was found to be particularly fast when used in combination with the spanner visibility graph scheme. The only disadvantage is that additional adjustment of nodes is required which may make it impractical (for example) in interactive scenarios where too much layout adjustment would spoil the user's mental map.

We were also pleased with the results of our spline refinement strategy when applied to spanner visibility graph routing. In the future we intend to do further analysis and improvement of algorithmic complexity of this step which currently could be high in the worst case, especially our short-cutting strategy.

References

1. Battista, G.D., Eades, P., Tamassia, R., Tollis, I.G.: Graph Drawing: Algorithms for the Visualization of Graphs. Prentice Hall (1999)
2. Bentley, J.L.: Multidimensional divide and conquer. Communications of the ACM 23(4), 214–229 (1980)
3. Clarkson, K.L.: Approximation algorithms for shortest path motion planning. In: STOC '87: Nineteenth. New York, New York (May 1987)
4. Dobkin, D.P., Gansner, E.R., Koutsofios, E., North, S.C.: Implementing a general-purpose edge router. In: Proc. 3^{rd} Int. Symp. Graph Drawing (GD'97). LNCS, vol. 1353, pp. 262–271. Springer (1998)
5. Dwyer, T., Marriott, K., Stuckey, P.J.: Fast node overlap removal. In: Proc. 13th Intl. Symp. Graph Drawing (GD '05). Lecture Notes in Computer Science, vol. 3843, pp. 153–164. Springer (2006)
6. Dwyer, T., Marriott, K., Wybrow, M.: Integrating edge routing into force-directed layout. In: Proc. 14th Intl. Symp. Graph Drawing (GD '06). Lecture Notes in Computer Science, vol. 4372, pp. 8–19. Springer (2007)
7. Dwyer, T., Marriott, K., Wybrow, M.: Topology preserving constrained graph layout. In: Proc. 16th Intl. Symp. Graph Drawing (GD'08). Lecture Notes in Computer Science, vol. 5417, pp. 230–241. Springer (2009), URL http://www.csse.monash.edu.au/ tdwyer/topology.pdf

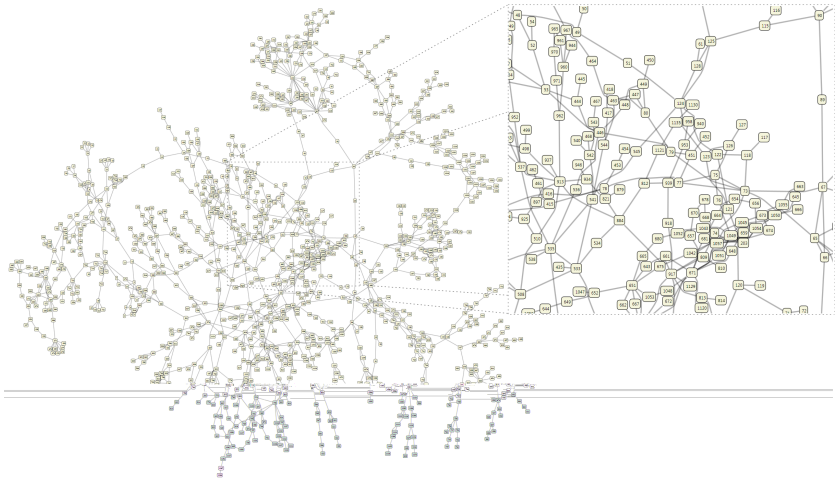
8. Freivalds, K.: Curved edge routing. In: Proc. 13^{th} Int. Symp. Fundamentals of Computation Theory. LNCS, vol. 2138, pp. 126–137. Springer (2001)
9. Gansner, E.R., North, S.C.: Improved force-directed layouts. In: Proc. 6^{th} Int. Symp. Graph Drawing (GD'98). LNCS, vol. 1547, pp. 364–373. Springer (1998)
10. Ghosh, S.K., Mount, D.M.: An output-sensitive algorithm for computing visibility. SIAM Journal on Computing 20(5), 888–910 (1991)
11. Lauther, U.: Multipole-based force approximation revisited - a simple but fast implementation using a dynamized enclosing-circle-enhanced k-d-tree. In: Proc. 14^{th} Intl. Symp. on Graph Drawing (GD'06). LNCS, vol. 4372, pp. 20–29 (2007)
12. Preparata, F.P., Hong, S.J.: Convex hulls of finite sets of points in two and three dimensions. Communications of the ACM 20(2), 87–92 (1977)
13. Wybrow, M., Marriott, K., Stuckey, P.J.: Incremental connector routing. In: Proc. 13^{th} Int. Symp. Graph Drawing (GD'05). LNCS, vol. 3843, pp. 446–457. Springer (2006)

8 Appendix: Detailed Results

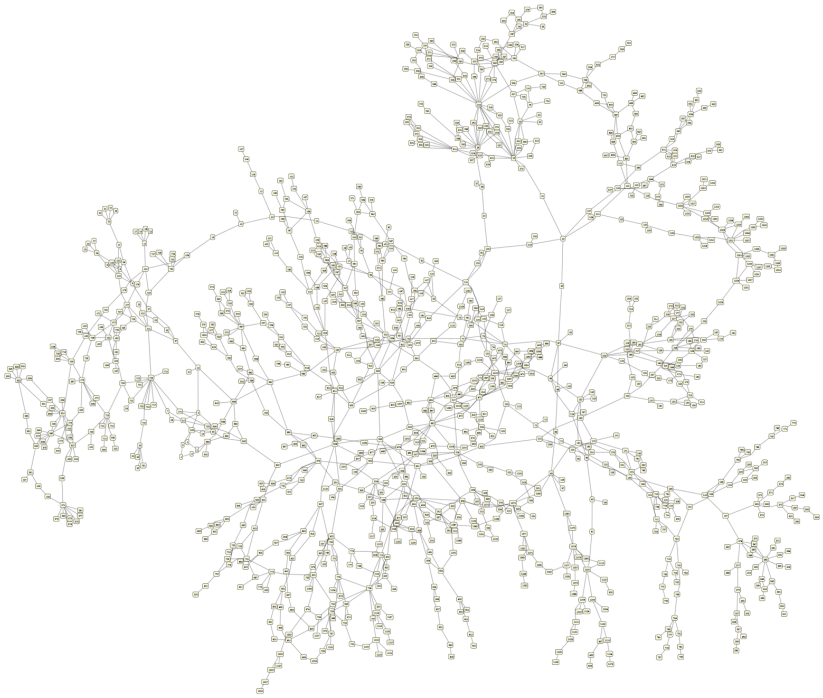
		Path type	Time	Max. Increase	Mean. Increase
1138 Bus Graph	1138 Nodes 1458 Edges	Optimal	94.73	0%	0
		10° cones	42.74	0.2%	0
		20° cones	38.23	0.5%	0
		30° cones	36.23	3.5%	0
		45° cones	33.87	7.0%	0
		Optimal + KD-Tree	9.76	201%	16%
		10° cones + KD-Tree	8.22	201%	16%
		20° cones + KD-Tree	6.40	201%	16%
		30° cones + KD-Tree	5.51	201%	16%
		45° cones + KD-Tree	5.07	201%	16%
GD'08 Companies Graph	130 Nodes 269 Edges	Optimal	1.59	0%	0
		10° cones	1.12	0.8%	0
		20° cones	1.04	2.6%	0
		30° cones	0.98	2.6%	0
		45° cones	0.97	6.3%	0
		Optimal + KD-Tree	1.22	179%	23%
		10° cones + KD-Tree	1.16	179%	23%
		20° cones + KD-Tree	0.89	179%	23%
		30° cones + KD-Tree	0.78	179%	23%
		45° cones + KD-Tree	0.76	179%	23%
GD'08 Torch Relay Graph	71 Nodes 145 Edges	Optimal	0.38	0%	0
		10° cones	0.26	4.2%	0
		20° cones	0.23	4.2%	0
		30° cones	0.23	3.5%	0
		45° cones	0.23	7.0%	0
		Optimal + KD-Tree	0.31	231%	20%
		10° cones + KD-Tree	0.21	231%	20%
		20° cones + KD-Tree	0.19	231%	20%
		30° cones + KD-Tree	0.19	231%	20%
		45° cones + KD-Tree	0.18	231%	20%

Table 1. Routing performance results under various conditions for different graphs. Time is in seconds. Percentage edge length increases are calculated by $100 \times (Approximate - Optimal)/Optimal$).

The results were obtained by using C# on a standard PC with a 2.1 GHz processor and 4 GB of memory.



(a) Edges routed with a spanner visibility graph using a cone angle of 20 degrees. Routing quality is virtually indistinguishable from the optimal shortest path routing.



(b) Using the spatial decomposition method much faster routing is possible but some adjustment of the layout occurs and edge routes are potentially more circuitous.

Fig. 9. A large *Matrix Market* graph (1138Bus) with 1138 nodes and 1458 edges routed using the methods described.