Tree-Based Distance Cartograms for Navigation

Jonathan Barron
CS270 + CS294-10 Project

Abstract

Cartograms are a popular method of using the geography of a map to convey some non-geographic piece of information. Though area cartograms are most popular, distance cartograms have also demonstrated some use in visualization. Unfortunately, distance cartograms are hard to automatically or manually generate, and their use has typically been constrained to point-to-point as-the-crow-flies problems, (such as airline travel), while many other uses (train navigation, for example) require more generic methods. We present a new type of distance cartograms, in which costs between (adjacent) points are specified using a tree, using the map of BART as an example of such a problem. We then present a method for automatically generating distance cartograms from such a tree. Our method consists of a number of non-trivial problems, each of which we present efficient solutions to.

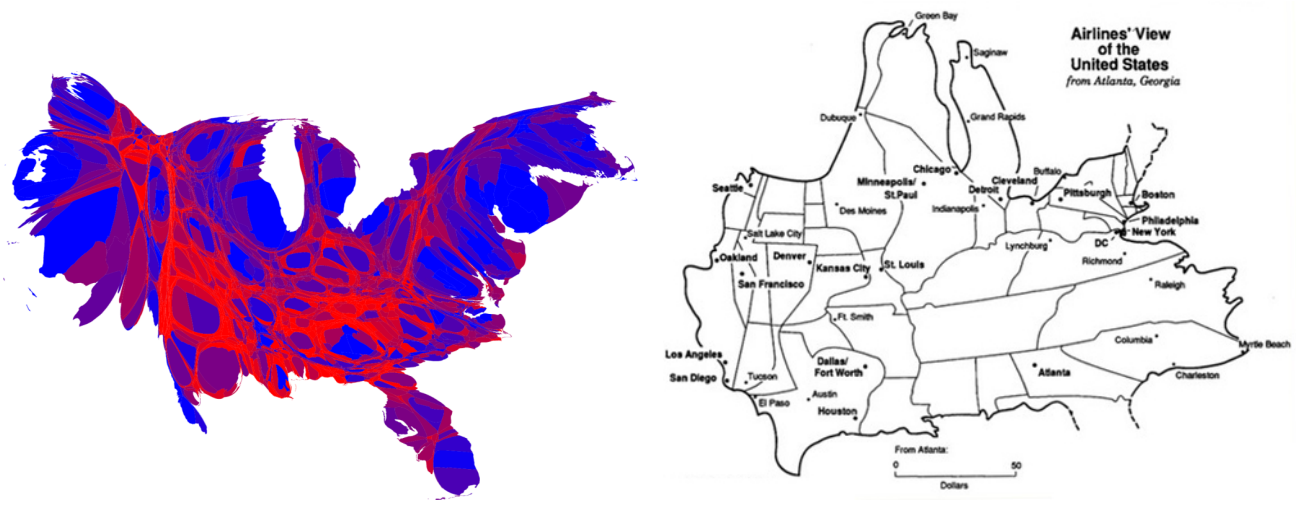
1. Introduction

A cartogram is a distortion of a map, for the purpose of displaying some non-geographic information. The most popular kind of cartogram is an "area" cartogram, in which areas of the map are enlarged or shrunk to reflect some property. For example, in Figure 1(a) we see a visualization of the 2004 US presidential election results, in which the area of the map has been distorted to reflect population density. Though this image does not accurately reflect the shape of the US or its states, it more accurately conveys non-geographic information, such as population. As opposed to a non-distorted map, which would seem heavily skewed to red (as red states tend to be larger and less dense), this area cartogram shows that red and blue are somewhat balanced, when the geographic size of states are factored out.

A "distance" cartogram is a distorted version of a map in which distances between points have been distorted to reflect some relationship. See Figure 1(b) for an example. A standard map of the United States has been distorted such that the distance between Atlanta and every other city is proportional to the cost of flying via airplane. These sorts of visualizations are powerful, as using distance to encode cost allows for effective comparison of costs ("Grand Rapids is twice as costly to visit as St. Louis") has been demonstrated to be effective by the information visualization community. However, the question of how to automatically generate such distance cartograms, especially when costs are more complicated than in Figure 1(b). In particular, we are interested in developing an algorithm that can tolerate not simply the costs from the origin to a set of points, but which can be used to visualize a *tree*, in which each edge has a cost, such that the total cost of visiting a node is the cost of the min-cost traversal of the tree from the origin. This framework encompasses a larger set of problems, such as air-travel with layovers, public transportation, etc.

Consider, for example, visualizing a tree-structured subway map, an example of which can be seen in Figure 6(a). Edges represent lines on the subway system, and nodes represent stops, with size (small = low, large = high) encoding the total "cost" of going from the start point (the root of the tree, indicated by a star) to that node, where "cost" is expected commute time. In this example, we calculated cost by first calculating the cost of traversing each edge (a function of the length of an edge, and the frequency at which trains run along that edge), and then finding the minimum-cost path from the origin (indicated by a star) to each point. We see that some points, though geographically close to the root, are substantially more "costly" to visit than other points that are geographically much more distant from the root. Using size (or color, shape, etc) to encode cost is unintuitive: it is difficult to estimate what points are more or less costly than others, which agrees with human perceptual research that indicates humans prefer using position to encode quantitative information.

In Figure 6(b), we see the output of our algorithm. The subway-tree in Figure 6(a) has been distorted such that Euclidian distance from the origin is exactly equal



(a) an area cartogram from (1), displaying presidential election results.

(b) a distance cartogram from (2), displaying airplane travel costs.

Figure 1. Two examples of cartograms.

to the cost of reaching that point from the origin. The image representing the map has accordingly been warped onto this transformed tree. An additional input/output example can be seen in Figure 3, which is a randomly generated graph on a colorized checkerboard image.

Because visualizations of this sort have never been made before, it's not clear how useful they will be. We hope that our results will provide insight into exactly how distance cartograms should be used, if at all.

2. Algorithm

Our algorithm is a series of operations, which we outline in Figures 8 and 9. First, we warp the tree (the subway line), by performing a traversal of the tree from the root, allocating "chucks" of space to each subtree, in a greedy fashion (Step 2). This algorithm has the nice property that distance from the origin is, by construction, equal to cost. We then tesselate the map-image into a mesh, which is parameterized by the nodes along the tree, and by a disjoint set of "seam" nodes (Step 3). We first subdivide the mesh by cutting along these seams, producing a set of triangles which attach to the tree (Step 4). These seam nodes are then adjusted to optimize a number of criteria, such as preventing overlap, minimizing distortion (perpendicular to the tree), and minimizing jaggedness in the warped mesh (Steps 5-7). We can "sew" up these seam nodes, to produce a coherent warped map-image (Steps 8-9). This mesh defines a transformation from the input image to the warped tree, which we use to produce the warped image seen in Figures 6(a) and 6(b).

2.1. Tree Warping

We lay out the tree according to the following critiera: The starting node (henceforth referred to as the origin) is located at the origin, and the Euclidian distance from all other tree-nodes to the origin must be exactly proportional to the cost from the origin to that node. We also require that edges in the warped tree do not overlap. Additionally, we want the tree to arranged such that no part of the map is much more densely or sparsely populated with tree nodes, and such that the overall shape of the warped tree is "simple", in that nodes tend to radiate outwards from the origin, with

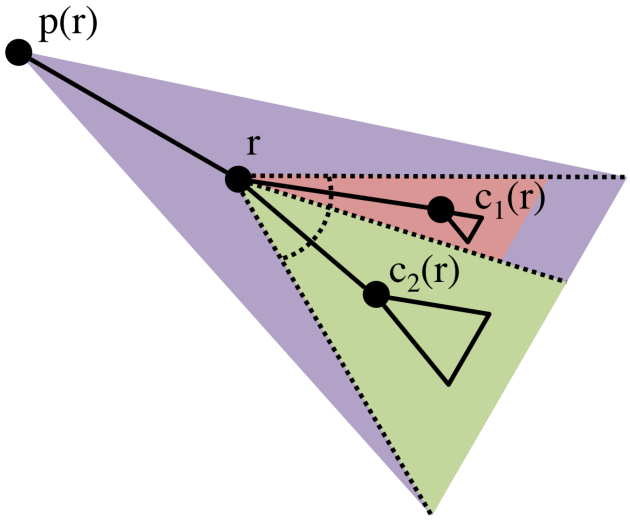


Figure 2. An example of our tree-warping algorithm.

no unnecessary zig-zagging. We were able to define a simple, greedy algorithm that traverses the tree in a breadth-first fashion, laying out the warped tree as it traverses.

Our algorithm, shown in Figure 2.1, is defined recursively on a subtree. Given the locations of node r and its parent $p(r)$, and an isoceles triangle rooted at $p(r)$ such that r is within that triangle, we wish to place $c(r)$, the children of r, and their enclosing triangles (note that r may have any number of children, though we just display 2). We do this by looking at the angle (shown as an arc) defined by r and the two corners of its enclosing triangle. This angle is subdivided for each sub-tree, with the size of the angle allocated to each subtree proportional to the number of nodes in that subtree (for the sake of having an evenly dense distribution of points in the final tree). Each angle is then subtended to produce the ray along which each $c(r)$ is placed. Each $c(r)$, is placed along its ray such that the Euclidian distance from $c(r)$ to the origin is equal to the cost of $c(r)$ (which requires using the quadratic equation). Assuming monotonically increasing costs, and assuming r was also placed such that its cost is proportional to its Euclidian distance from the origin, $c(r)$ will be able to be placed such that this requirement is satisfied, and such that $c(r)$ lies within the triangle enclosing r. Then, to place the enclosing triangle around $c(r)$, we find another point along that ray, such that the distance to the origin is equal to the cost of the most expensive node in the subtree of $c(r)$. The two corners of that triangle are placed at that distance, and the algorithm can then recurse. When placing the origin and its children, instead of considering an enclosing triangle, we use the same algorithm except with all angles being valid, so we subdivide all angles from 0-360.

One goal in placing the enclosing triangles is that the entire subtree rooted at the node placed within that triangle can be fit within it. As can be seen in Figures 8(b) and 9(b), this is not always the case, but the algorithm still produces pleasant-looking warpings that satisfy our criteria, so we consider this method of placing triangles a useful heuristic.

Note that our two hard criteria (that distance $\propto$ cost, and that edges do not intersect) are necessarily satisfied by this algorithm. A much simpler algorithm, which we experimented with earlier, is to simply convert the tree into polar coordinates, such that radius $\propto$ cost, and the angle is determined during a breadth-first traversal of the tree. This satisfies the first hard criterion, but sometimes produces overlapping edges.

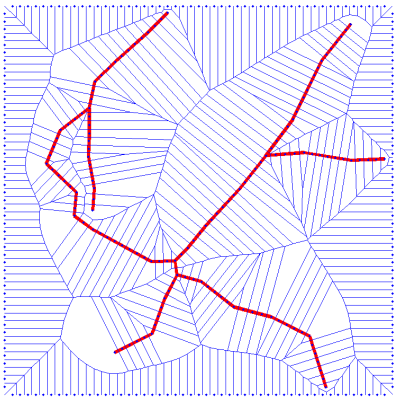


Figure 3. A Voronoi tesselation of the (subsampled points) on the initial tree. Blue dots are nodes in the voronoi tesselation, red lines indicate the initial tree, blue lines indicate edges in the voronoi tesselation.

2.2. Mesh Construction

Once we have an initial tree, and a warped tree, we need to construct a warping from every point in the initial map to the warped map. Our solution to this problem will be to tesselate the initial map into a triangular mesh, and then warp that mesh into the warped tree. This is a non-trivial operation, and requires a number of steps. The output of this mesh construction can be seen in Figures 8(b) and 9(b). We will first define the procedure we use, and then explain why we have followed this procedure.

First, we wish to construct the Voronoi tesselation of the tree, shown in Figure 2.2. Voronoi tesselations of line segments instead of points have been studied extensively, and are sometimes called "segment" Voronoi diagrams(4), or variations of the medial axis transformation(5). Instead of the points and polygons that the standard Voronoi tesselation produces, these techniques produce curves and line segments. To then produce a polygonal tesselation from these techniques requires sampling these curves and lines, which seems to defeat the purpose. We will therefore subsample our initial tree (adding nodes to each edge such that the distance between adjacent nodes is $\leq \epsilon$), and then take the Voronoi tesselation of those subsampled points (and additional points along the edge of the map). These "vanilla" Voronoi tesselations can be computed efficiently(3). These new points (which we will call "seam" points, as they compose the "seams" along which we will "cut" our final mesh) are equidistant from points on the tree, and are therefore

roughly equidistant from the line segments that the tree are composed of. First, we remove all edges in the voronoi tesselation that intersect with the initial tree (and those that connect to the added nodes along the border). We then add in all edges between each seam node and their closest nodes in the (subsampled) initial tree, and the edges of the subsampled initial tree. The resulting tesselation is what we see in Figures 8(b) and 9(b), where we have a series of triangles connecting the tree-nodes to the seam-nodes, such that each edge of the tree has a strip of triangles attached to it, bordering it on each side. Each strip will then be warped onto the warped tree, in Sections 2.3 and 2.4.

We must modify the mesh such that, when warped, it can be "cut" along the seam-nodes. This requires duplicating each seam-node, and then selectively merging those that belong to triangles that are on the same side of the seam. The details of this are tedious, but the overall process can be accomplished efficiently with the union-find datastructure.

2.3. Mesh Parameterization

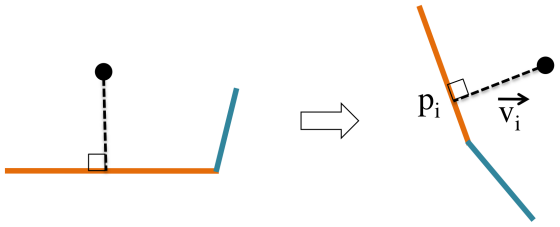
We can now parametrize the mesh constructed in the previous section, such that it can be warped onto the warped tree. We investigated using standard mesh warping techniques like Laplacian surface-editing(6), but these techniques tend to favor producing warpings in which distortion (by some metric like rigidity, congruence, similarity, etc) is minimized, and the primary goal of our algorithm is to produce a warping in which the map *is* heavily distorted when Euclidian distance on the map is not proportional to distance. We therefore had to design our own distortion technique.

We will parametrize each seam-node i by a point $\mathbf{p_i}$ and a unit vector $\mathbf{v_i}$. We will do this by looking at the nearest point on the tree to each seam-node. There are only three possible cases that we will need to address, each of which are shown in Figure 2.3.

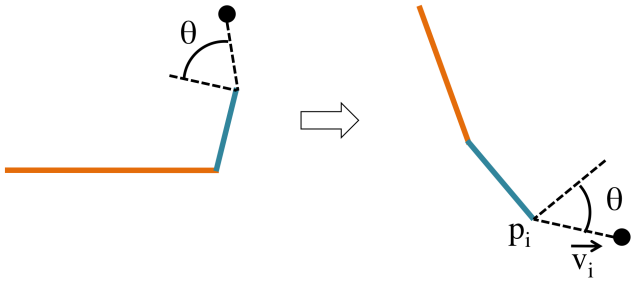
Case 1 (Figure 4(a)) is if the closest point on the tree is a point on an edge. In that case, we look at the distance along the edge of the projection of the point on that edge, and find a corresponding point on the warped tree ($\mathbf{p_i}$). We then take the perpendicular direction of that edge as $\mathbf{v_i}$, making sure that the vector is on the correct side of the edge.

Case 2 (Figure 4(b)) is if the closest point on the tree is a leaf node. In this case, we look at the angle defined by the vector from the seam-node to the tree-node, and the adjacent edge. We then find a vector on the warped tree with the same angle, and use that vector as $\mathbf{v_i}$, and that warped tree-node as $\mathbf{p_i}$.

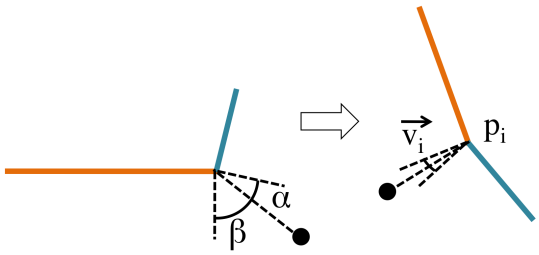
Case 3 (Figure 4(c)) is if the closest point is a non-leaf node. We use that non-leaf warped tree node as



(a) Case 1: closest point is an edge



(b) Case 2: closest point is a leaf node



(c) Case 3: closest point is a non-leaf node

Figure 4. The three cases we must address in parametrizing the seam-nodes

$\mathbf{p_i}$, and look at the angles between the adjacent edges and the seam-node. We then construct $\mathbf{v_i}$ such that the angle between it and the adjacent edges on the warped tree have the same ratio as they did in the initial tree.

This $\mathbf{p_i}$, $\mathbf{v_i}$ parametrization defines a family of warped meshes, in which the free parameters are the scalar values s_i that each $\mathbf{v_i}$ is multiplied with. That is, $\mathbf{n_i} = \mathbf{p_i} + s_i \times \mathbf{v_i}$, where $\mathbf{n_i}$ is the warped version of seam-node i. With this parametrization, we can optimize over s_i to produce an appropriate mesh.

2.4. Mesh Optimization

When optimizing over our parametrized mesh, we must have some criteria in mind.

1. We do not want triangles in the warped mesh to overlap.

2. We want the warped mesh to occupy as much of the screen as possible, for the sake of conveying as much information as possible

3. Because of how we parametrized our mesh in case 1, we will necessarily have significant distortion along the direction of the tree edges (as this is the goal of our algorithm), but we would like to minimize the amount of distortion perpendicular to each edge.

4. We do not want a jagged warped mesh, even though the initial mesh is very jagged, because it often difficult to interpret.

To satisfy the first goal, we first find the edges in the voronoi tesselation of the warped tree (shown in Figure 8(e) and 9(e)). By finding the first intersection of these edges with each ray $\mathbf{p_i} + s_i \times \mathbf{v_i}$, we find s_i^{max}, the maximum value of s_i. By keeping $s_i \leq s_i^{max}$, we constrain the seam-nodes such that the resulting warped mesh doesn't not self-intersect.

To satisfy the second goal, we can simply maximize each s_i, which has the effect of maximizing the area of each triangle, thereby maximizing the utilization of the screen.

To satisfy the third goal, for each seam-node we calculate d_i, the distance from that initial seam-node to the nearest point on the initial tree. In the warped mesh, we will encourage the distance from each seam-node to $\mathbf{p_i}$ (which corresponds to the nearest point of each initial seam-node on the initial tree) to be as close as possible to d_i. Because each v_i is a unit vector, this is equivalent to encouraging each s_i to be close to the corresponding d_i. Doing this minimizes distortion perpendicular to each tree-edge.

To satisfy the fourth goal, we simply constrain each s_i to be similar to the scales of its neighboring seam-nodes (that is, the other seam nodes which are involved in the same triangles as s_i). We will call each set of neighboring seam-nodes N_i.

These four criteria are satisfied by solving the following convex optimization problem:

$$\begin{aligned} \max \quad & \sum_i s_i - \lambda_1 (s_i - d_i)^2 - \lambda_2 \sum_{j \in N_i} (s_i - s_j)^2 \\ \text{s.t.} \quad & \forall_i s_i \leq s_i^{max} \end{aligned} \tag{1}$$

Where λ_1 and λ_2 are non-negative multipliers that control how heavily our smoothness and distortion penalties are weighted, and are set by hand. We solve this optimization problem using a generic convex optimization solver, though it certainly seems to have the form of a quadratic program, and could therefore be solved as such.

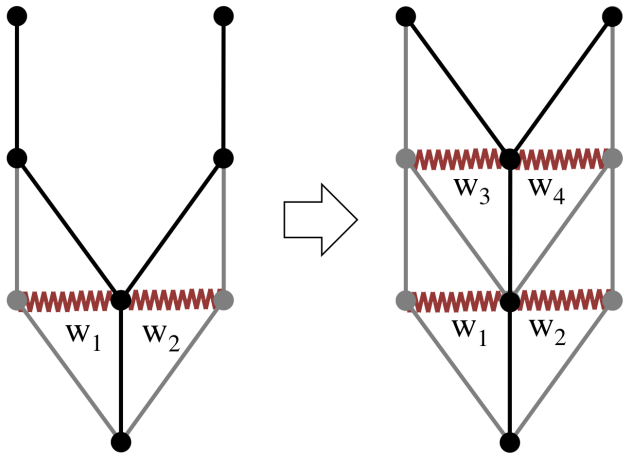


Figure 5. An example of a seam in our warped mesh being "sewn". Active edges are shown in black, removed edges are shown in gray. Distortion of the seam-nodes is shown as red springs.

Figures 8(d) and 8(d) show how the warped mesh looks when s_i are chosen arbitrarily. We see overlapping edges, and triangles with heavy perpendicular distortion. In Figures 8(f) and 9(f) show how the distorted mesh looks when we simply set each s_i to s_i^{max} (equivalent to solving the optimization problem with $\lambda_1 = \lambda_2 = 0$), in which there is no more overlapping, but still heavy distortion and jaggedness. In Figures 8(g) and 9(g) we see the final output of our optimization routine, with the λ multipliers set properly. The edges of the warped mesh are smooth, perpendicular distortion is minimized, overlaps do not occur, and the mesh still occupies a significant portion of the image.

2.5. Mesh Post-processing

The warped meshes have an unfortunate property that often, two warped seam-nodes that correspond to identical seam-nodes in the initial mesh are near each other, but not at the exact same location at each other, which makes small cracks in the warped mesh. We therefore experimented with post-processing the warped meshes, in which we "sew" some of the seams in the mesh back up, by merging their matching seam-nodes. We will do this according to the following criteria: we want to eliminate unnecessary edges in the mesh (that is, the edges that contain seam-nodes that could be merged), but not at the cost of excessively distorting the un-sewn warped mesh. This appears to be a difficult combinatorial optimization problem, but is made simpler by the fact that we only wish to merge a set of seam-nodes if the seam-nodes prior to that node

have already been sewn (imagine pulling a zipper up each seam). This requirement enforces a strict ordering over seam-nodes that can be merged, which makes this operation easy to implement efficiently.

First, we must construct the ordering in which nodes can be sewn. This ordering is shown in Figures 8(h) and 9(h), in which color encodes the different (disjoint) seams, and size encodes the order (smallest first). The disjoint seams can be found efficiently simply by finding the connected components of the seam-node edges. Finding the ordering is more complicated: for each disjoint seam, we keep track of the nodes that have been visited, and we repeatedly query the graph for a node which is adjacent to exactly 1 not-visited node. As nodes are visited in this fashion, they are added to the table of visited nodes, and to that seam's ordering. The resulting ordering is exactly to our specification: "leafs" of the seam nodes are added first (as they philosophically correspond to the starting point of the seams we are sewing), and then their neighbors are added, provided that the seam in all but one direction has already been sewn up. This is important for the three-way intersections in the seams, in which we must take care not to sew together a seam until all down-stream edges have already been sewn.

Given these orderings, we must now describe a cost function over seams in the warped tree. The motivation for our cost function is as follows: we want to reduce the number of duplicate edges (separate edges in our warped mesh that correspond to a single edge in the initial mesh), while reducing how much each warped seam-node is distorted from its starting position (the position returned in Section 2.4). Our cost function will therefore be $-|E| + \lambda_3 \sum_i w_i$, where $|E|$ is the number of seam-edges (edges that connect seam-nodes) in the warped mesh, and w_i is the Euclidian distance that seam-node i has been distorted. Therefore, in Figure 2.5, the cost of the tree to the left (before sewing an edge) is $cost_0 = -5 + \lambda_3(w_1 + w_2)$, while the cost of the tree to the right (after sewing an edge) is $cost_1 = -4 + \lambda_3(w_1 + w_2 + w_3 + w_4)$. The difference between the two is $\Delta_{cost} = -1 + \lambda_3(w_3 + w_4)$. We set λ_3 by hand, according to how appealing the resulting visualization looks.

Given how this cost function decomposes nicely as the sum of the cost of each node-sewing operation, and given that we have a pre-defined ordering in which edges must be sewn, it is easy to efficiently calculate the costs of all legal sewings: we simply iterate through the ordering, sewing nodes and recording the cost at each step, where we calculate the cost by calculating the Δ_{cost}, and adding that to the previously computed cost. This is effectively an extremely simple dynamic programming problem, in which we have a single vector for each disjoint seam, which we fill in (and can therefore be computed extremely efficiently). After the cost table for each seam has been constructed, we select the entry with the lowest cost, and use the set of merged nodes that corresponds to that entry (which we have already computed) as our new set of merged nodes. The resulting "sewn" meshes can be seen in Figures 8(i) and 9(i).

The warped mesh after this sewing operation has much fewer unnecessary seams, but this operation often produces overlapping triangles. This could presumably be remedied by checking for overlap during the optimization procedure and assigning overlaps a very high cost, though this seems significantly more computationally expensive than the current algorithm.

2.6. Image Warping

With our final warped mesh, warping the underlying image is fairly straightforward, and very similar to texture mapping. We will scan through each pixel in the warped mesh, and check which triangle it lies in. We then compute an affine transformation from that warped triangle to the initial triangle, and compute the unwarped coordinates of the pixel being queried. We then interpolate the value of that unwarped coordinate in the initial image, as use that as the pixel value of the point being queried in the warped mesh. This operation can be done efficiently using standard graphics techniques and hardware, by simply texture-mapping the initial mesh, and then rendering the warped mesh. The final warped images can be seen in Figures 6(b) and 7(b). Additionally, in Figure 3 we labeled some point on the warped and unwarped map, such that they can more easily be correlated with each other.

3. Conclusions

There are many shortcomings to this method of visualization:

- The map-image being warped can be heavily distorted — in fact, for distance cartograms to be useful, this *must* be the case: we would not want to produce a visualization such as this for a map in which euclidian distance is proportional to cost, which are the only maps in which distortion would not occur.
- Distances between points not on the tree are effectively meaningless. Preserving distances between such points seems as though it would overconstrain our algorithm, though its not even clear *what* we would want to preserve.

- It is difficult to relate the warped map to the unwarped map. In Figure 6(b), for example, it is difficult to see that the map on the far left is in fact the eastern coast of the San Francisco peninsula. This is necessary artefact from our warping requirements, so it is not clear how to resolve this issue.

- This algorithm does not address cyclic trees (general graphs). The fundamental ideas of this method of visualization does not have a clear analog when the graph is not a strict tree.

- There is no real notion of what it means to travel *off* the tree. For example, most users would not travel the length of Bart and Caltrain to visit Giant's stadium, when they could get off the Bart earlier and simply walk the remaining distance. Exactly how to resolve this is unclear.

Thankfully, there seem to be some benefits to this visualization method:

- The warped maps are *compelling* and *informative*. Viewers seem to enjoy interpreting these visualizations, and are often surprised by what they learn regarding relative distances. Such effects when viewing the un-distorted maps are less common, because this cost information must be encoded with a less effective cue than position.

- Labeling known points on the map seems to be very helpful for orienting the viewer, and for comparing distances.

A simplification to this visualization technique, in which we do not render the warped image, but instead heavily annotate the warped tree, may be more useful in practice. Or perhaps a compromise between displaying and not displaying the warped image, in which the warped image becomes partially translucent as it gets further from the tree, may be most effective.

Many of the negative aspects of this method of visualization might be addressed by radically rethinking what it means to warp a map. Our method, because it relies on this mesh warping framework, implicitly has the assumption that every point in the initial map must be present exactly once in the warped mesh. However, it may be reasonable to render the same point on the initial map more than once on the warped map if it can be reached in multiple ways (for example, one can walk to Giant's stadium from Bart, or reach it by Caltrain). This idea should probably be addressed in conjunction with rethinking what it means to travel "off" the tree, as this is presumably the means by which one could reach the same destination through two separate routes.

References

[1] `http://www-personal.umich.edu/~mejn/election/2008/`.

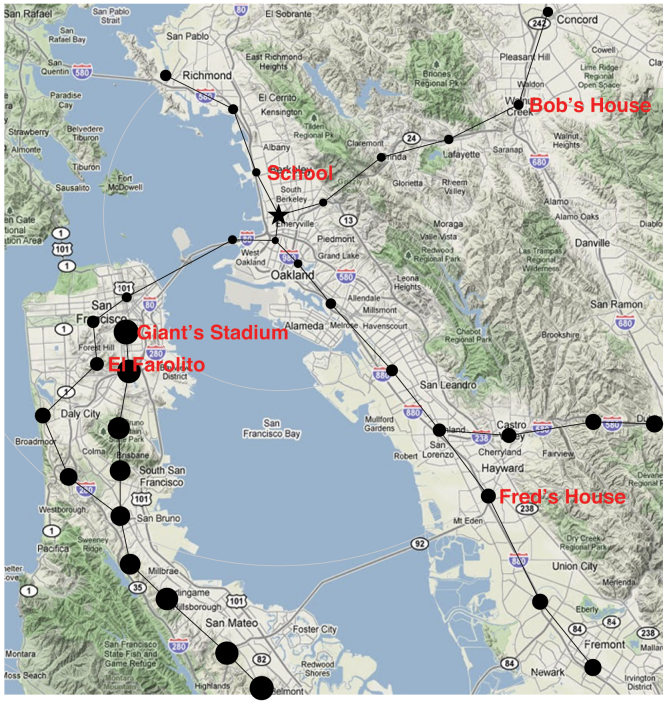
[2] B. Dent. *Cartography: Thematic Map Design*. Wm. C. Brown, Dubuque, IA, 1996.

[3] S. Fortune. A sweepline algorithm for voronoi diagrams. In *SCG '86: Proceedings of the second annual symposium on Computational geometry*, pages 313–322, New York, NY, USA, 1986. ACM.

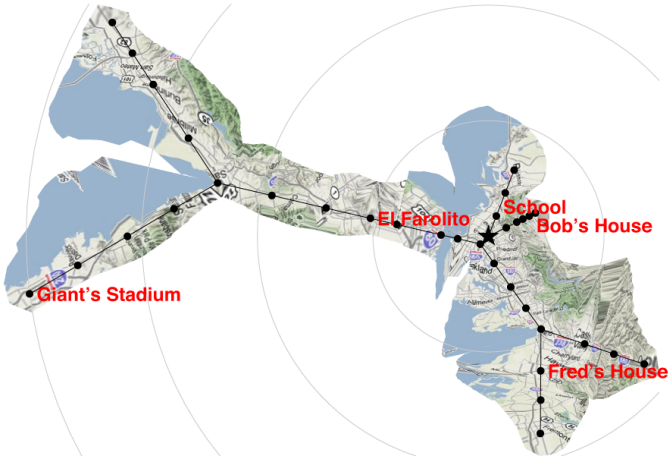
[4] M. I. Karavelas. A robust and efficient implementation for the segment voronoi diagram. In *Proc. 1 st Int. Symp. on Voronoi Diagrams in Science and Engineering*, pages 51–62, 2004.

[5] D. Lee. Medial axis transformation of a planar shape. 4(4):363–369, July 1982.

[6] O. Sorkine, D. Cohen-Or, Y. Lipman, M. Alexa, C. Rössl, and H.-P. Seidel. Laplacian surface editing. In *SGP '04: Proceedings of the 2004 Eurographics/ACM SIGGRAPH symposium on Geometry processing*, pages 175–184, New York, NY, USA, 2004. ACM.

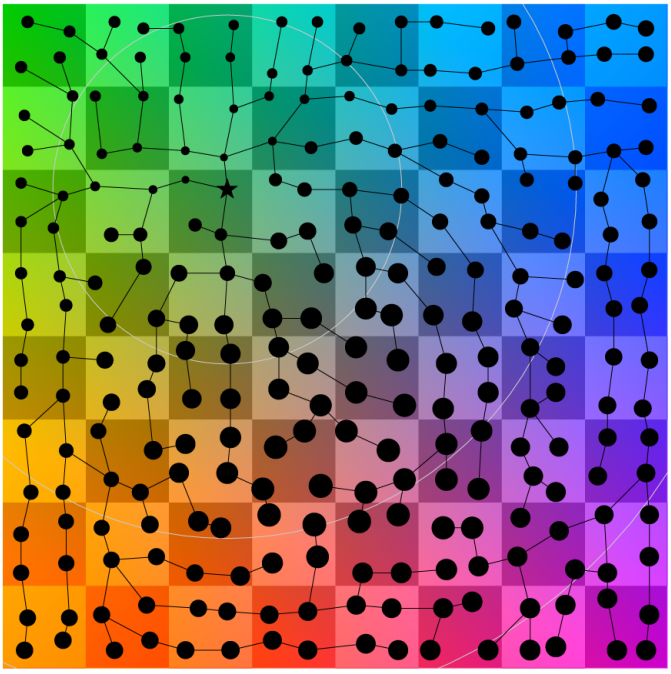


(a) Input

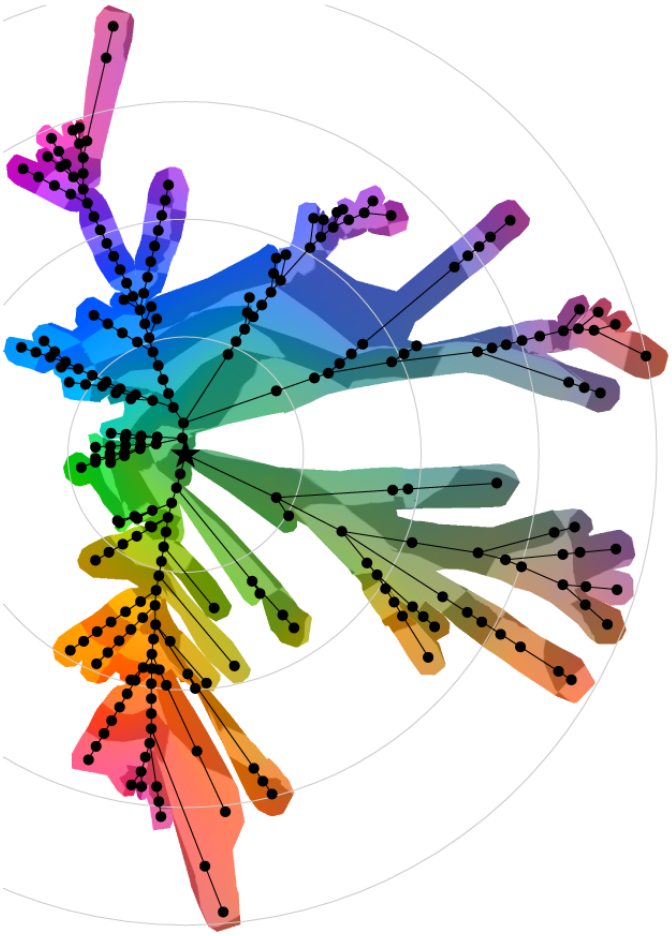


(b) Output

Figure 6. An input and output map, of the Bay Area Rapid Transit system.

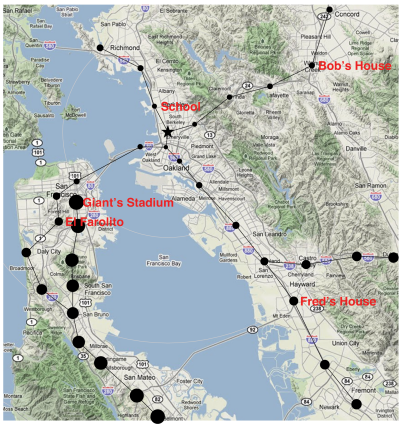


(a) Input

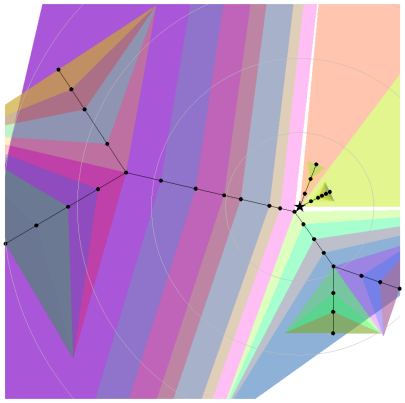


(b) Output

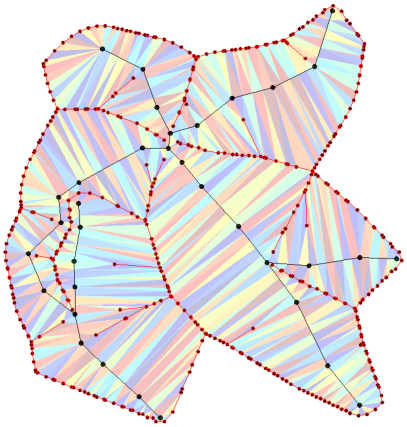
Figure 7. A synthetic input and output map.



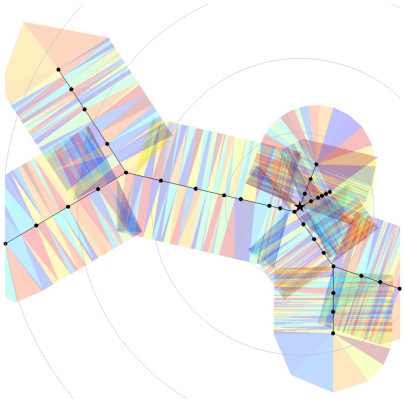
(a) Step 1: The input, a tree with costs from the origin to each point, which satisfy the triangle inequality, and an accompanying map.



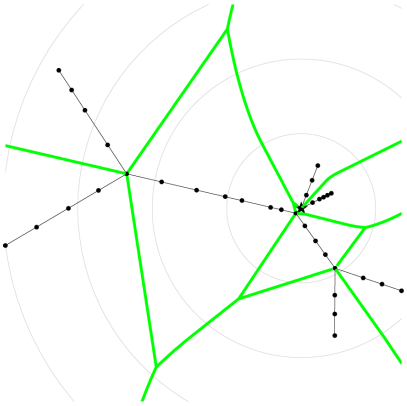
(b) Step 2: The input tree, warped such that distance from the origin is proportional to cost from the origin. The translucent triangles are used in warping the tree.



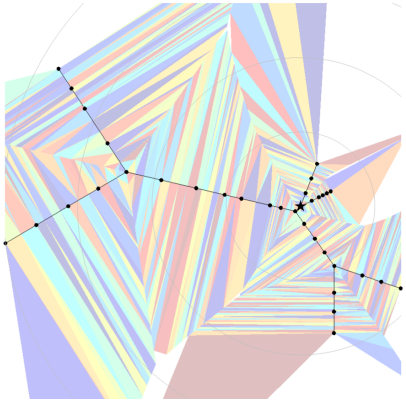
(c) Step 3: The mesh resulting from the voronoi tesselation of the tree. Red edges indicate "seams" along which we separate the mesh.



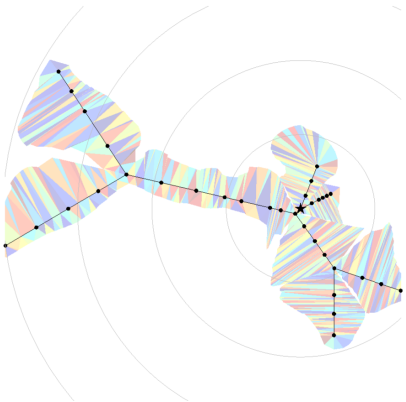
(d) Step 4: That mesh re-oriented with respect to the warped tree in step 2.



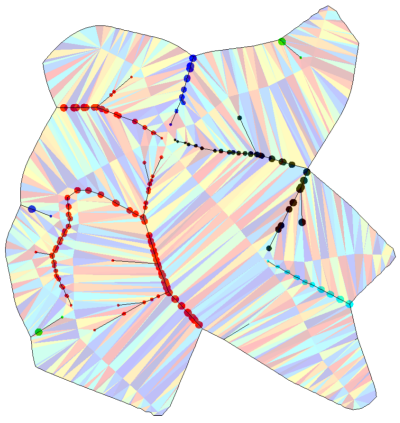
(e) Step 5: The voronoi tesselation of the warped tree, indicating sensible boundaries for the warped mesh.



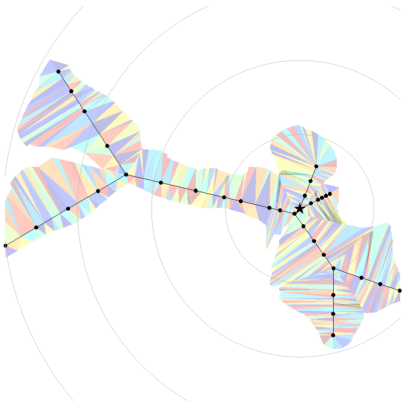
(f) Step 6: The mesh in step 4, fit within the tesselation boundaries in step 5.



(g) Step 7: The mesh in step 5, optimized to minimize perpendicular distortion and jaggedness

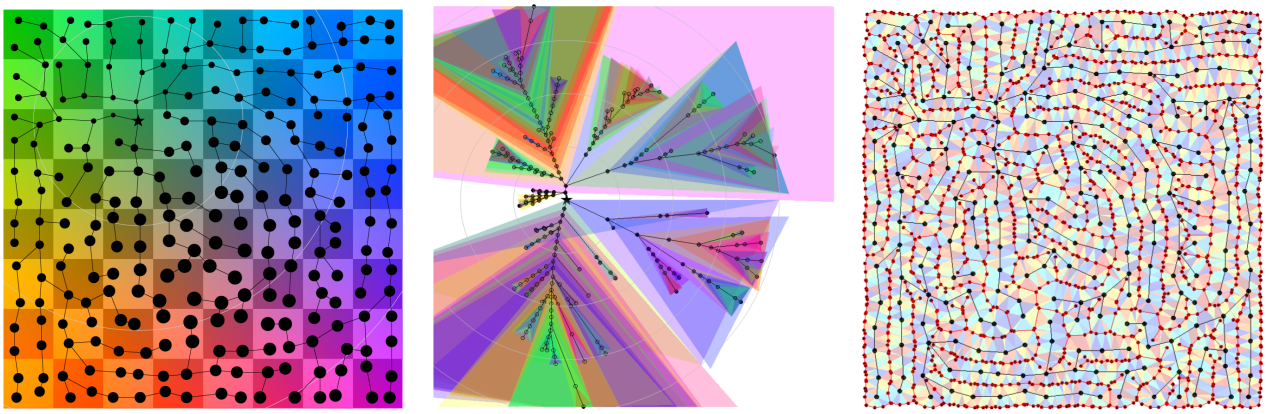


(h) Step 8: The different seams along which we may wish to "sew" up the warped mesh. Color indicates which seam, size indicates order in which we must sew (smallest first)

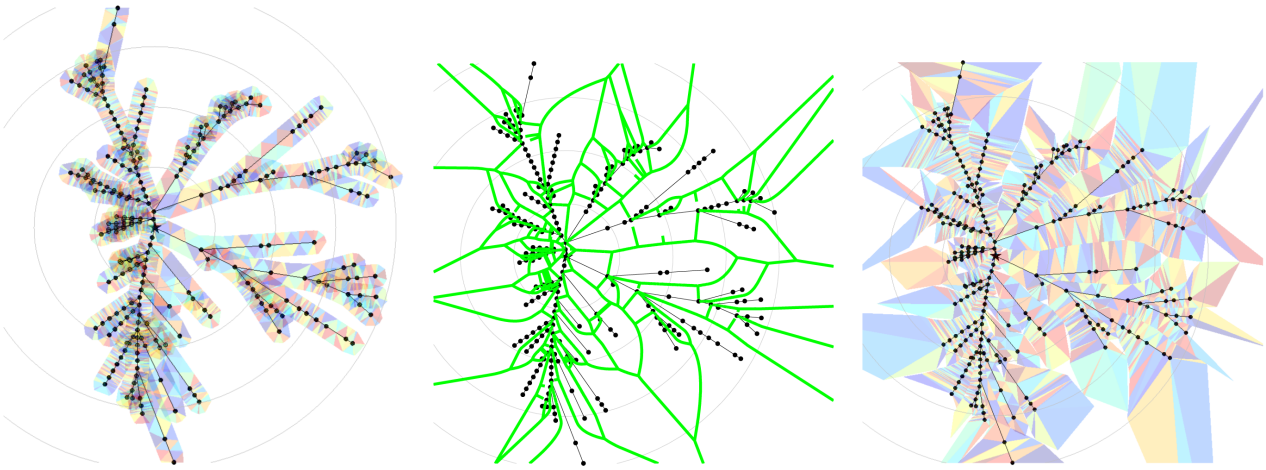


(i) Step 9: The mesh in step 7, with some seams sewn.

Figure 8. A walkthrough of our algorithm.



(a) Step 1 (b) Step 2 (c) Step 3



(d) Step 4 (e) Step 5 (f) Step 6

(g) Step 7 (h) Step 8 (i) Step 9

Figure 9. The same algorithm as show in Figure 3, but with a different input.