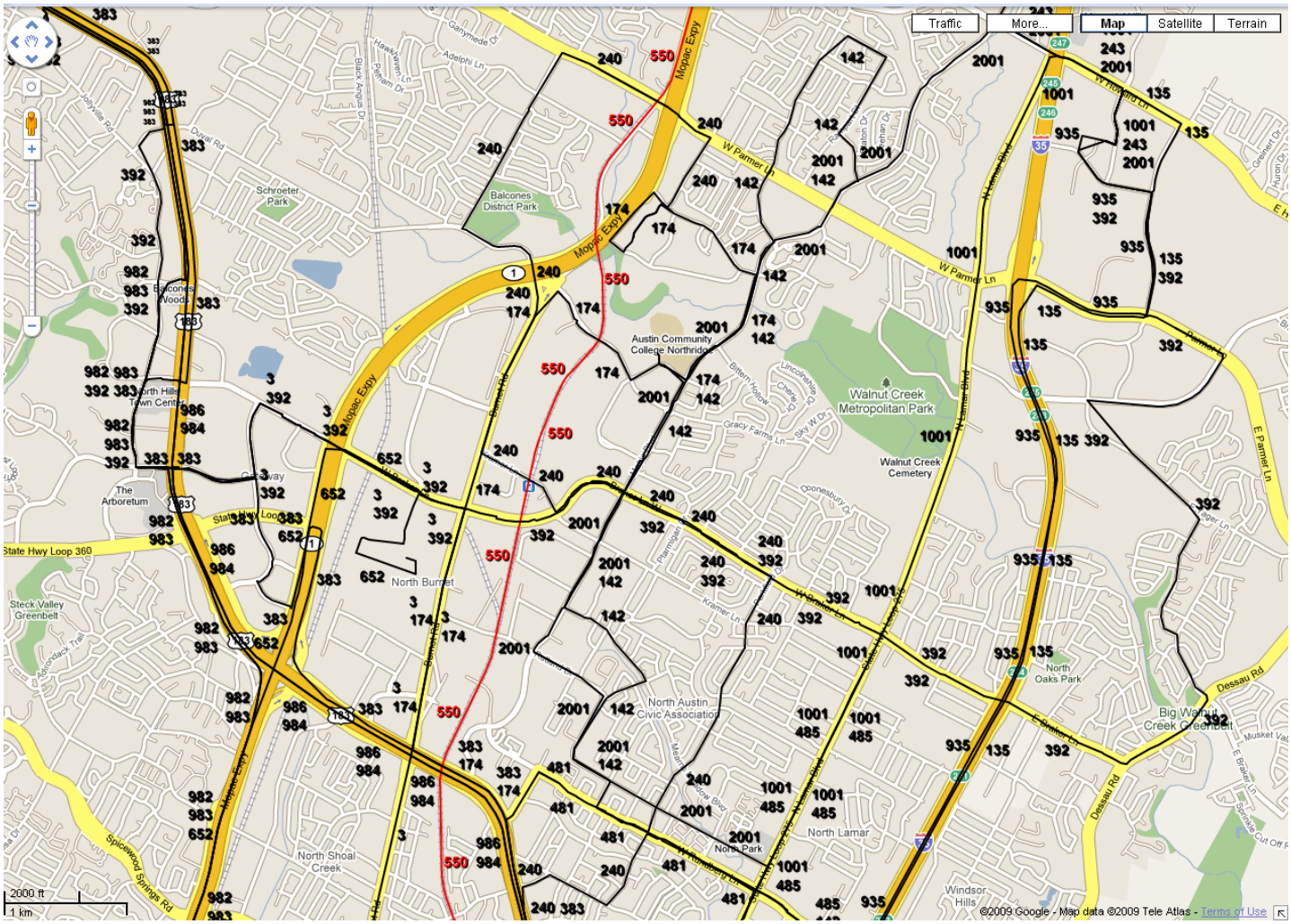
Periodic Multi-Labeling of Public Transit Lines

Valentin Polishchuk and Arto Vihavainen

CS Department, University of Helsinki
{firstname.lastname}@cs.helsinki.fi

Abstract. We designed and implemented a simple and fast heuristic for placing multiple labels along edges of a planar network. As a testbed, real-world data from Google Transit is taken: our implementation outputs an overlay onto Google Maps, adding route numbers to public transit lines.



1 Introduction

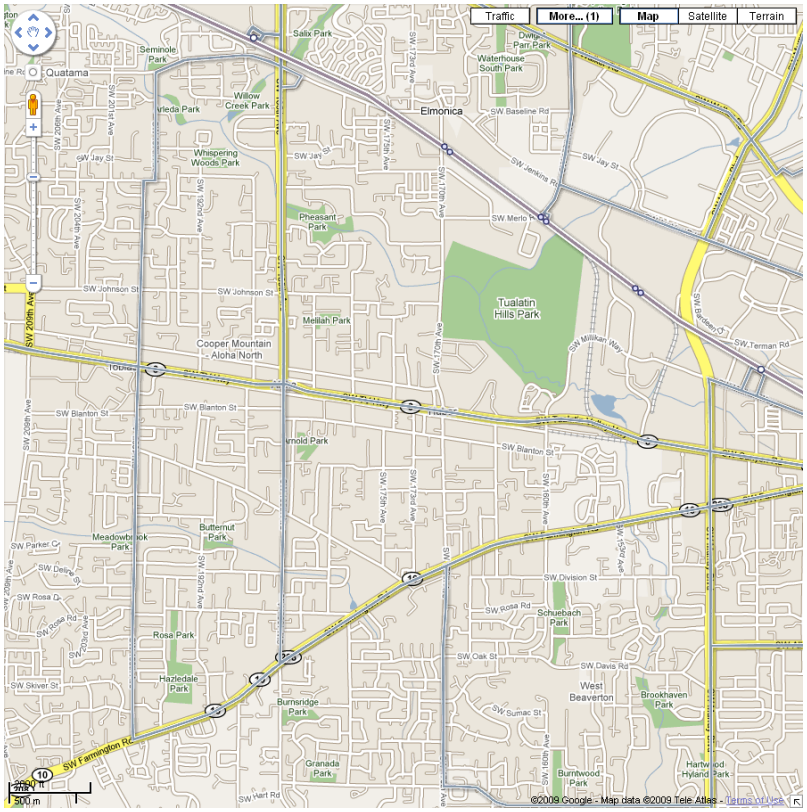
Map labeling comes in many flavors depending, in particular, on the "dimensionality" of the objects to be labeled. Cities, buildings, and other places of interest on a map are "zero-dimensional" points; the requirement to add labels close to them highly restricts the label placement options. Streets, rivers, etc. are one-dimensional; their labels may be placed anywhere on or along them. Countries, states, districts are two-dimensional, and bear labels inside them. A common (often, implicit) assumption in map labeling applications is that the *objects* that have to be labeled are *disjoint*. (Of course, the labels must be disjoint too, and this is the essence of the labeling challenge.)

In this paper, we consider labeling linear features that may *partially overlap*. Hence, parts of the features have to be labeled with multiple labels. Moreover, the features are *long*, which requires that every feature is labeled multiple times, on a periodic basis.

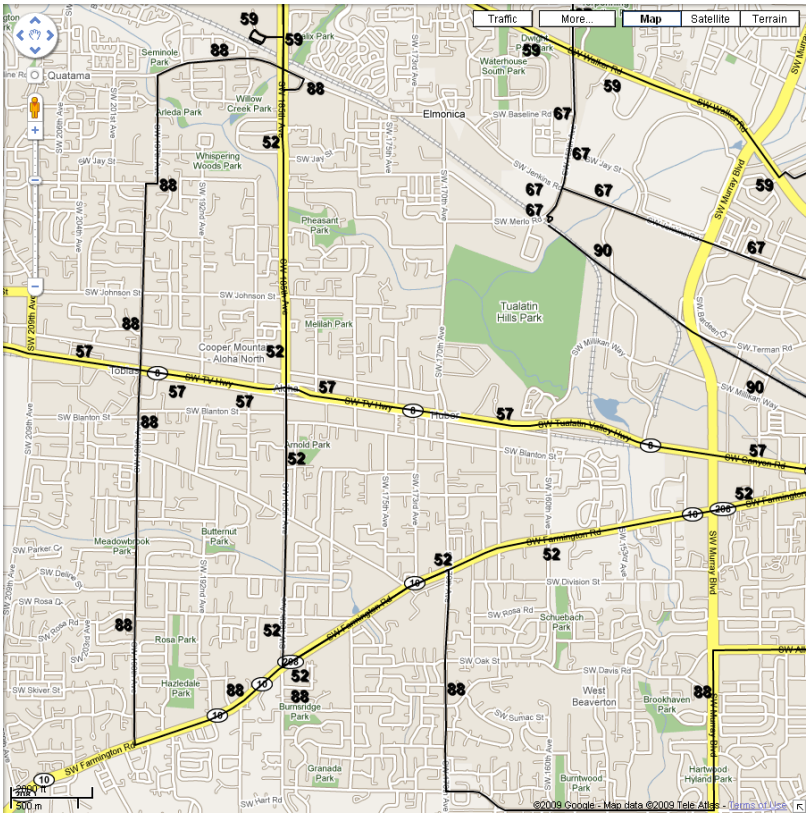
Motivation Public transportation maps have two formats. *Static* maps are printed in booklets or hung at the stops. To the best of our knowledge, the current practice is to place route labels on these maps manually, which is a rewarding but tedious job. We set out to ease it by automating the label placement. Our implementation allows one to customize the parameters of the label placement procedure, as well as to manually fine-tune the output by adding/moving/deleting/resizing the placed labels.

Recent years have seen a shift towards *online, interactive* maps for journey planning. The possibly leading system here is Google Transit™; many cities employ planners of their own. While these systems work very well for suggesting a path *from A to B*, the possibility of seeing a map of a location with *all* nearby public transport routes labeled, is often missing (the only available option is usually just to zoom on a static map). In many of the systems, one can only see which streets have public transport on them, but unfortunately not the line numbers (Figs. 1 and 2). Even a brute-force solution of putting the labels, say, at the stops would be an improvement; a more intelligent route labeling could be even more helpful.

Related work Two variations of linear-features labeling have been considered in the literature: *street labeling* and *river labeling* [15–17, 20, 21]; see also Chapter 58.3.1 in the book [12]. Our problem is different from these because different rivers do not share parts of streambeds, as well as different streets very rarely share the same road (and when they do, it does create confusion for the map reader). On the contrary, in our setting exactly the opposite is the case – many lines may run along the same road (and hence, for instance, a simple solution such as coloring the routes differently may not easily help). The other, more subtle difference is that street or river names are usually long (words) while our labels are generally short (numbers).



A snapshot of Google Maps with the Transit overlay



Our output overlayed on the same location

Fig. 1: Seeing just the routes is not very helpful without knowing which numbers they are.

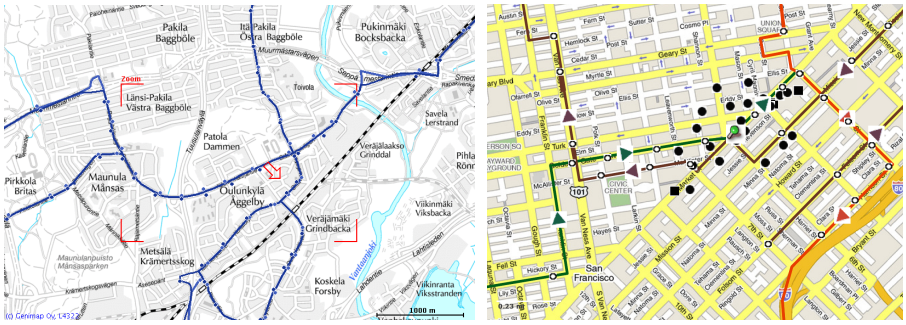


Fig. 2: Cities' journey planners have a functionality to show public transport routes near a location; displaying route labels here could be a huge plus. Left: Helsinki (reittiopas.fi). Right: San Franciso (transit.511.org).

In *boundary labeling* the goal is to connect point features inside a region to the labels placed on the region's boundary [4–8, 14]. While in standard point-feature labeling the label for a feature has only few candidate positions, in the boundary labeling the labels can be slid arbitrarily along the boundary. Sliding the labels was studied in [11, 19, 21], and is also relevant for us, since route labels can be slid along the roads.

For labeling *points*, simple heuristics are known to work well in practice [12, Ch. 58.3.1]. Our heuristic is a practical one for labeling (possibly overlapping) *linear features* with a large number of regularly spaced small-size labels. An automated system for generating an attractions map is presented in [13].

Our contribution We developed a simple and efficient label generator for routes in a transportation network. To the best of our knowledge, it has no competitors — we are not aware of any automatic system for periodic multi-labeling of overlapping linear features. The latest version of our software is publicly available from `http://www.cs.helsinki.fi/group/compgeom/maplab/`.

As a test case we use real-world public transport route data from Google Transit Data Feed [1]. Sample outputs of our algorithm are presented in figures throughout the paper; Table 1 provides links for online versions of the maps in the figures.

Naturally, the (quality of the) algorithm's output is greatly influenced by the parameter values with which the algorithm is run. The overall number of parameters is small, and a user can familiarize himself with them quickly. However, for the purposes of fully automated label generation (say, when the algorithm is deployed in an interactive online system), no parameter adjustment should be expected from the user. That is, the final, good-looking output must be produced with *default* parameters values. We were able to set the default values so that satisfactory results are obtained without any finetuning. Figures 3 and 6 compare our output with "official" maps produced "once-and-forever" with human oversight.

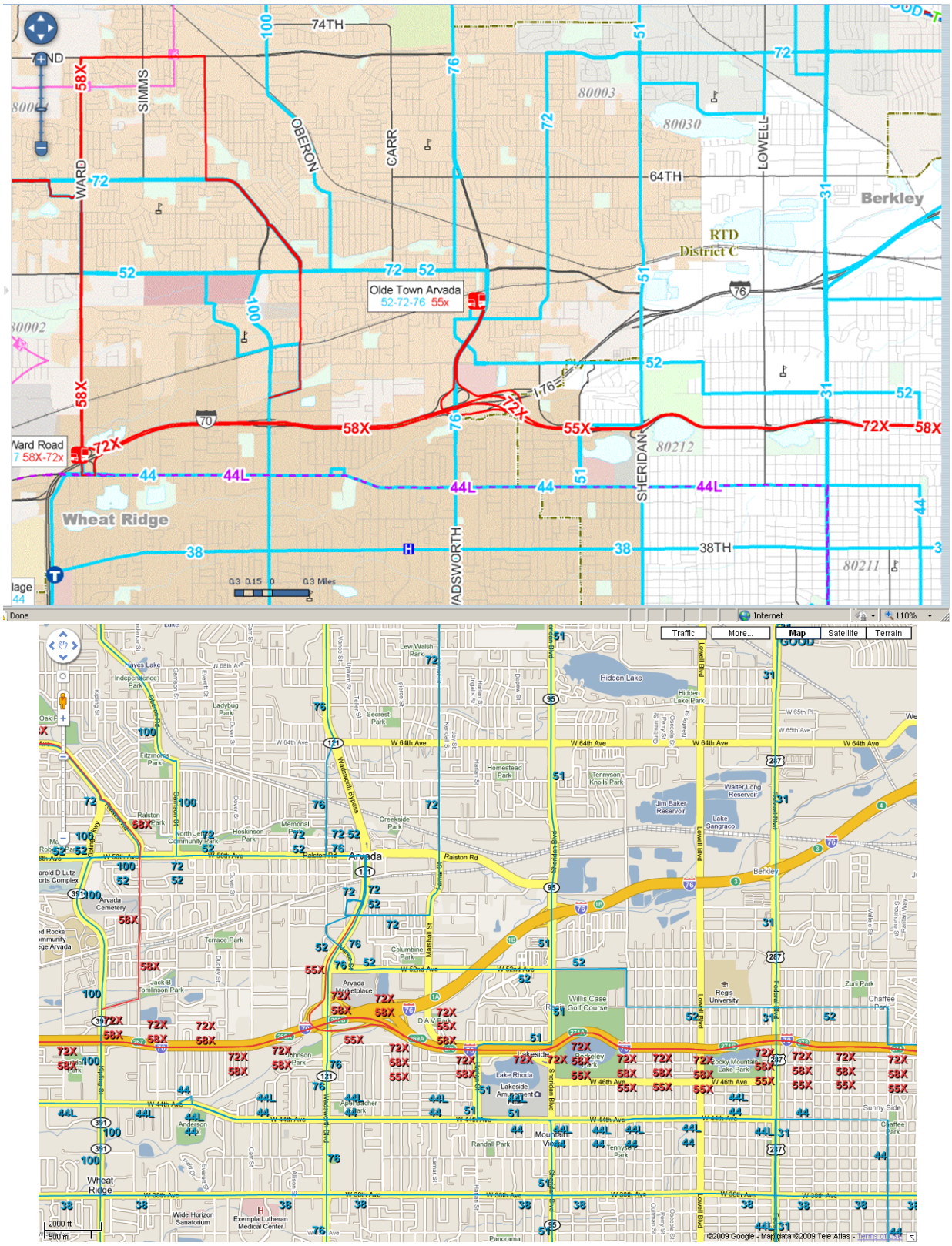


Fig. 3: Official map from rtd-denver.com (top) against our output (bottom).

1.1 Overview of the Approach

The input to our algorithm is a route network – a collection of polygonal paths, each representing a public transit line (Fig. 4, left). The algorithm places route labels along each path, based on a set of user-defined parameters.

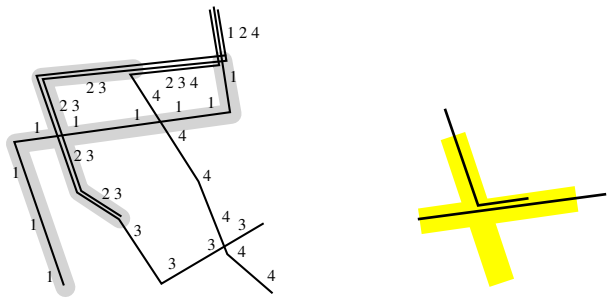


Fig. 4: Left: A network; two of the stretches are enclosed in shaded tubes. Right: Routes merge at an intersection of roads, but the corresponding paths do not cross.

Our first step is to preprocess the routes, breaking them into (maximal) stretches such that along each stretch, the set of lines using the stretch is the same. This implies, of course, that along each stretch, the route labels stay the same. So for each stretch, we group the labels of its routes into a "multi-label" – a box that contains the labels of all the lines using the stretch. (Such label grouping is very common in the existing public transport maps of large cities.)

The stretches are further broken, by the intersection points between them, into *subpaths*. This reduces our original labeling problem to the one in which we are given a set of subpaths (pairwise-disjoint other than at endpoints), each with its own multi-label. Our goal now is to place the multi-labels along the subpaths so that the routes are "easy to follow", while satisfying the constraints that the multi-labels stay pairwise-disjoint and do not overlap with any of the subpaths. That is, the boxes corresponding to different multi-labels must not intersect between themselves and, in addition, no box should intersect any of the routes. Satisfying the constraints is necessary and sufficient to guarantee overall clarity of the labeling (we add a small margin to each box in order to enforce some minimum separation between different multi-labels, as well as between a multi-label and a route).

Key Idea: "Propagation" from intersections

The crux of our approach is the observation that for routes to be easily tracked, it is most important to place route labels near points of routes intersection. The intuition is that in a subpath, along which the routes do not change, any particular route can be traced easily. It is only at an intersection point, where

the routes diverge and merge, that the route may be lost. Thus, our general approach is as follows:

Step 1: *Place multi-labels near the endpoints of the subpaths*, and then,
Step 2: *For each subpath, long enough to hold more multi-labels, place them evenly along the subpath.*

Inside each of the steps, the subpaths are processed in the order of increasing "capacity", which is the maximum number of multi-labels that can be placed along the subpath. The rationale is simple: if subpaths "compete" for a candidate multi-label location, less-capacitated subpaths should be given priority, for if a smaller-capacity subpath is not labeled at the location, it may not get a chance to be labeled at all, while a higher-capacitated subpath is more likely to get its multi-label somewhere else along its stretch.

2 Algorithmic Details

We now elaborate on the preprocessing and the label placement steps, and discuss the choice of default values for the parameters.

2.1 Preprocessing

Real-world routes are stored in Google Transit Data Feeds with too high a precision, making the data files unnecessarily large. We simplified the input using Douglas-Peucker algorithm [10].

Another issue is that in the raw data downloadable from Google Transit, different routes going along the same road are often represented by slightly different vertex sequences. This may be due to the fact that the data for different routes comes from different authorities or just because different routes were "sampled" differently. In addition, when two routes merge at an intersection of two roads, the paths that represent the routes may not necessarily intersect because the data is "too precise" (Fig. 4, right).

To address the above issue, we introduce a *mergeThreshold* parameter. Whenever there is a vertex of a route that is closer than the threshold to another route, we snap the vertex to the latter route. Such snapping is implemented as a part of the sweepline algorithm [9, Ch. 2] for detecting the intersections between the routes; this way the merging does not (asymptotically) increase the running time of the algorithm.

Overall, on the sweep completion we have a set of subpath, pairwise-disjoint other than at endpoints; each subpath has a unique set of routes that use it. The last step of the preprocessing is creating the multi-label box for each subpath. In case the multi-label contains more than one number (i.e., the subpath is shared by several routes), we arrange the labels into one of the pre-defined rectangluar grid-like patterns, and choose the pattern with the smallest area (in case of ties – one with the smallest aspect ratio).

Note that although our algorithm works with the merged routes, when overlaying the algorithm's output onto Google Maps, we show the original, unmodified routes. This does not hurt the visual effect of label placement as the labels are placed close also to the original routes (and our modification does not distort the routes by much). At the same time, it allows to output all features present in Google Transit data. For instance, for some cities the routes are colored, and we show them such on the map.

2.2 Label Placement

In what follows, where it causes no confusion, we will call the multi-labels just *labels* (even though a multi-label may contain labels for several lines). We will call the subpaths just *paths*.

For every path we estimate its capacity as follows:

$$capacity \quad \leftarrow \quad 1 + \left\lfloor \frac{pathLength - labelLength - 2 \cdot offset}{labelLength + gap} \right\rfloor$$

where *pathLength* is the length of the path, *labelLength* is the length of the box of the path's multi-label, *offset* is the distance between the label and path endpoint, and *gap* is the gap between the consecutive labels – the last two are user-defined parameters. Of course, the formula is exact only if the path is a horizontal straight-line segment; however, in practice, it gives a good estimate of the number of labels that can be placed along any path.[1]

We first place labels near paths endpoints, processing paths in order of increasing capacity; during the placement, some paths are marked with an 'X' indicating that we give up placing more labels for them:

- Capacity-0 paths are not labeled and are marked with X. Such paths are very short (shorter than the label length), so the user cannot loose a route while following such a path.
- For every capacity-1 path, we try to place the label in its midpoint, at distance *roadDistance* from the path where *roadDistance* is a user-specified parameter. If it is not possible to place the label on one side of the path, the other side is tried. After that, whether we succeeded or not, we mark the path with an X: even if the labeling did not succeed, we will not attempt to label the path in the future because capacity-1 paths are still short.
- For $c = 2, 3, \ldots$, for every capacity-c path we try to place the labels near each endpoint of the path, more specifically – at distance *offset* from the endpoint

[1] This may have several plausible explanations: (1) The majority of the subpaths are straight, and even if not, they "curve" only gently. (2) The subpaths never "zigzag back and force" so the length of the path reflects well the number of labels that have to be evenly spaced along the path. (3) The boxes have small aspect ratio, so the estimate is not hurt much when a path is not horizontal. (4) Often, the paths are almost perfectly North-South or East-West. (This is common at least in the US cities.)

and at distance *roadDistance* from the path. If the placement is not possible, the label is dragged away from the endpoint until a feasible placement is found. Then, another side of the path is tried, and the label remains only on the side for which the dragged distance was smaller.
If no placement is found on either side, the path is marked with X. If the labels dragged from different endpoints come closer than $labelLength + 2{*}gap$, only one label is kept, and the path is marked with X.

Now, for each path not marked with X, we reestimate its "residual" capacity taking into account the locations of its two placed labels (note that only the paths with two labels remain not marked with X):

$$capacity \quad \leftarrow \quad 1+\left\lfloor \frac{|pathLength(label2) - pathLength(label1)| - labelLength - gap}{labelLength + gap} \right\rfloor$$

where $|pathLength(label2) - pathLength(label1)|$ gives the distance between the placed labels. We place additional labels, evenly spaced on each path. The paths are, again, processed in order of increasing capacity. As before, we try both sides of the path when placing a label, and we drag the label until a feasible placement is found (this time though we do not drag further then for a distance *maxDrag* – another user-specified parameter).

Size matters In a dense part of the network there may not be enough space to place even a single label of desired size, without intersecting the routes. In such situation, it makes sense to reduce labels size by using a smaller font for label text. We use a local network complexity metric to determine the areas where the label font size is halved. The complexity metric is based on the total length and the number of routes that intersect a given cell of a regular grid. The complexity of the cell is calculated as the weighted sum of the total length of the routes in the cell and the number of routes crossing the cell boundary (the weights are user-defined parameters).

We shifted the grid by half the cell length in $\pm x$ and $\pm y$ directions, thus obtaining 4 grids. The label size was halved in an area whenever the area was voted to have complexity higher than 10 in 3 or more of the 4 grids (Fig. 5).

2.3 Default parameters values

There are three types of parameters:

Map-, or input-related These are used during the preprocessing (Section 2.1). *DouglasPeuckerEpsilon* is the distance parameter of the Douglas-Peucker line simplification algorithm. Based on the level of details, visible when viewing the routes at a reasonable zooming (we consider zoom level 12-14 in Google Maps to be "reasonable"), we set $DouglasPeuckerEpsilon = 2$ meters. In most cases, the difference between the original and the simplified routes cannot be noticed by eye while the labeling speeds up substantially.

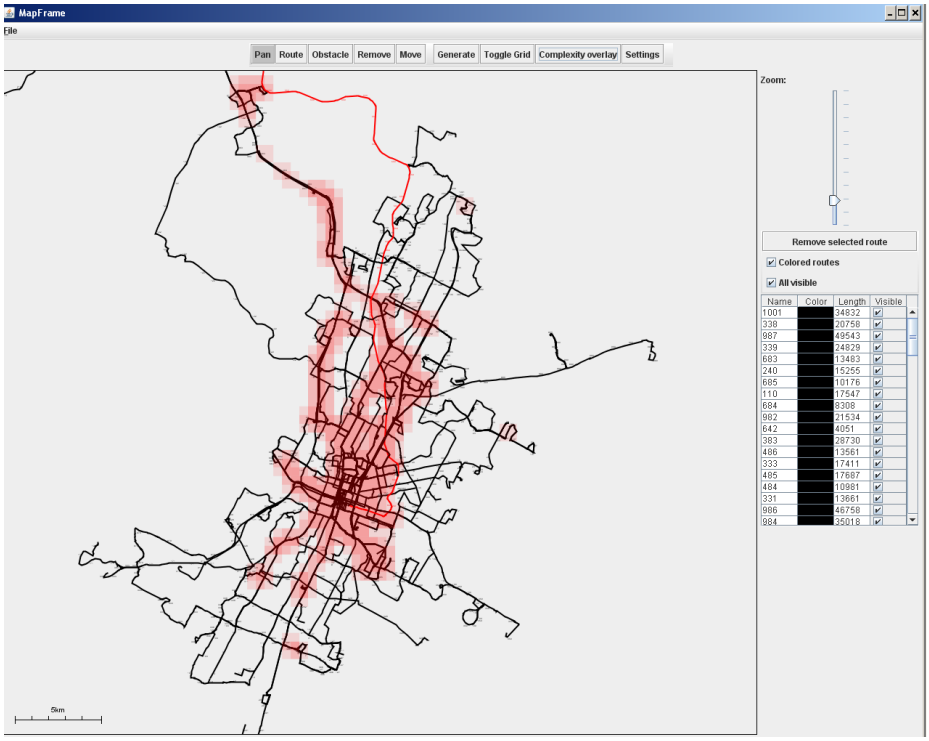


Fig. 5: Complexity overlay over Austin network.

The *mergeThreshold* defines when two routes are assumed to belong to the same road (see Section 2.1). We set the default value of the parameter to 8 meters based on a typical width of a road.

Complexity-related The default values $lengthMultiplier = 1.35$, $routeCountMultiplier = .45$ (Section 2.2) were chosen empirically; the default grid size is 1 km.

Output-related The parameters *labelFontSize*, *roadDistance*, *offset*, *maxDrag*, *gap* are used in the labeling itself (Section 2.2). We choose *labelFontSize* so that it looks approximately as 12pt font when projected on Google Maps at zoom level 14; the height of a number at that scale is about $h = 90$ meters. The rest of the parameters are set based on h: $roadDistance = h/3 \approx 30$m, $offset = 3h \approx 270$m, $gap = 10h \approx 900$m, $maxDrag = 5h \approx 450$m.

All maps in figures in this paper, as well as those at linked URLs were generated by the algorithm running with the default parameters values and without any postprocessing.

3 Results: Overlaying onto Google Maps

The results of our implementation can be overlayed on Google Maps. The overlay is toggled on by entering the location of a .kml file, produced by the implementation, into Google Maps' search box (the box, into which the address is usually typed). Links to the .kml files from the figures in this paper are given in Table 1.[2]

Figure/ City	Link to the kml file (may be input into Google Maps search box)
Title page	`http://www.cs.helsinki.fi/valentin.polishchuk/pages/map/austinPark.kml`
Fig. 1	`http://www.cs.helsinki.fi/valentin.polishchuk/pages/map/portlandSmall.kml`
Fig. 3	`http://www.cs.helsinki.fi/valentin.polishchuk/pages/map/denverSmall.kml`
Fig. 6	`http://www.cs.helsinki.fi/valentin.polishchuk/pages/map/portlandSideBySide.kml`
Austin	`http://www.cs.helsinki.fi/valentin.polishchuk/pages/map/austin.kml`
Dallas	`http://www.cs.helsinki.fi/valentin.polishchuk/pages/map/dallas.kml`
Denver	`http://www.cs.helsinki.fi/valentin.polishchuk/pages/map/denver.kml`
Portland	`http://www.cs.helsinki.fi/valentin.polishchuk/pages/map/portland.kml`

Table 1: To see the map, click on the link or copy and paste it into Google Maps search box. For better contrast, remove the satellite image in Google Maps (click the Map button).

We also generated .kml files with labeled routes for whole cities. Unfortunately, Google Maps limits the size and the complexity of displayed .kml files [2]. Due to the restrictions, not all labels produced by our implementation for the cities are actually rendered on Google Maps. Anyway, in Table 1 we offer links to the cities' .kml files, but note that our output cannot be fairly judged for them because of the many missing labels. (For the maps, presented in the figures, we ran our algorithm on small subsets of routes visible for each figure; thus, all our labels should be seen for them.)

Running times Loading Google Transit data for a city takes 10-50sec. Generating labels takes from 2 sec to tens of seconds.

4 Extensions

It is easy to add to our implementation the functionalities like using different colors, thickness, style, fonts, etc. for different routes (in fact, the implementation has the possibility to manually change route colors, or to use the colors that

[2] One and the same city has several data files in Google Transit Data Feed, which differ a lot one from another. We suspect in certain cases none of the files is actually close to reality; in fact, Google Transit directions sometimes use routes that are not in the database.

come from Google Transit Data Feeds). Our goal though was to keep the output "minimalistic" w.r.t. the number of ways used to distinguish between the routes. Even with that, our map can compete in readability with the ones using the power of coloring, different label styles, and human supervision (see Figs. 3 and 6).

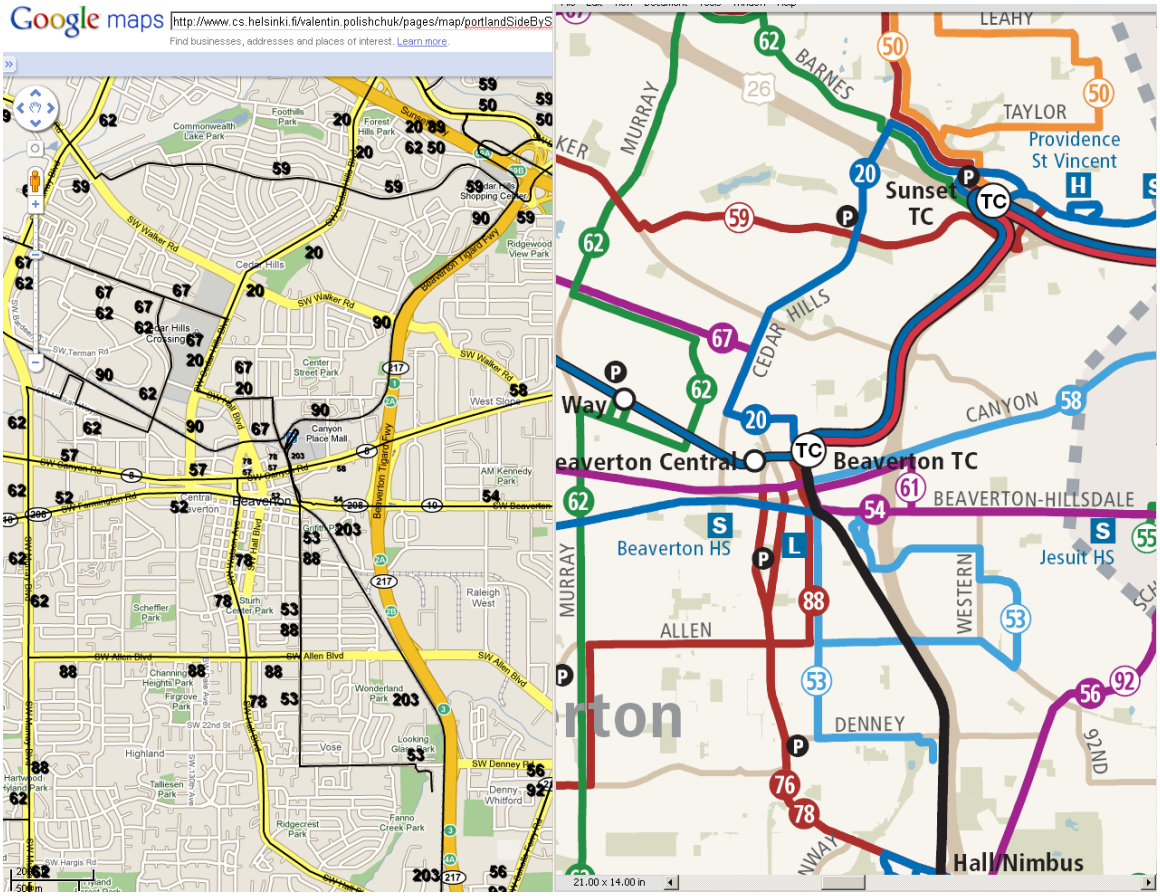


Fig. 6: Somewhere in Portland: our output side-by-side with official map (trimet.org). Note that the latter is a static map, produced with human oversight once-and-forever, while ours is an interactive online overlay.

City centers As expected, our output is worst in the city centers where the routes form a complicated network, often with almost all routes having a terminal point. To help that, one may introduce more grades of label font size, and/or

change the definition of local complexity, e.g., base the font size on the depth of a region in some tree-like subdivision of the input.

Scaling with zooming Our labels are images; the implementation can also output text labels. Because text labels do not scale with zooming, deciding active ranges [3] for them is a separate research problem. (In our case, one may ask that the labels near intersections have larger ranges.)

Obstacles Our implementation allows the user to specify regions where labels are not allowed to be placed, due to, say, another label residing there (a district label, a point of interest, etc.)

Future work We are experimenting with other ways of defining the local network complexity, e.g., based on a quadtree subdivision. Also, it would be interesting to evaluate the quality of our labeling by running a user study or by using the framework of [18].

Acknowledgments We acknowledge discussions with members of Algorithms groups at Stony Brook University, TU Eindhoven and Karlsruhe Institute of Technology. Initial coding was done by David Consuegra, Vesa Hautsalo, Niko Himanen, Anttijuhani Lantto and Mikko Sysikaski during a class project at CS Department, the University of Helsinki. Google Maps™and Google Transit™are Google Brand Features.

References

1. `http://code.google.com/p/googletransitdatafeed/wiki/PublicFeeds`.
2. `http://code.google.com/apis/kml/documentation/mapsSupport.html`.
3. K. Been, M. Nöllenburg, S.-H. Poon, and A. Wolff. Optimizing active ranges for consistent dynamic map labeling. In *SCG '08: Proceedings of the twenty-fourth annual symposium on Computational geometry*, pages 10–19, New York, NY, USA, 2008. ACM.
4. M. A. Bekos, M. Kaufmann, K. Potika, and A. Symvonis. Multi-stack boundary labeling problems. In S. Arun-Kumar and N. Garg, editors, *FSTTCS 2006: Foundations of Software Technology and Theoretical Computer Science, 26th International Conference, Kolkata, India, December 13-15, 2006, Proceedings*, pages 81–92, 2006.
5. M. A. Bekos, M. Kaufmann, A. Symvonis, and A. Wolff. Boundary labeling: Models and efficient algorithms for rectangular maps. *Comput. Geom.*, 36(3):215–236, 2007.
6. M. Benkert, H. Haverkort, M. Kroll, and M. Nöllenburg. Algorithms for multi-criteria boundary labeling. *Journal of Graph Algorithms and Applications*, 2009.
7. M. Benkert, H. J. Haverkort, M. Kroll, and M. Nöllenburg. Algorithms for multi-criteria one-sided boundary labeling. In *Graph Drawing, 15th International Symposium, GD 2007, Sydney, Australia, September 24-26, 2007. Revised Papers*, pages 243–254, 2007.

8. M. Benkert and M. Nöllenburg. Improved algorithms for length-minimal one-sided boundary labeling. In *23rd European Workshop on Computational Geometry (EuroCG'07)*, pages 190–193, 2007.
9. M. d. Berg, O. Cheong, M. v. Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*. Springer, 2008.
10. D. H. Douglas and T. K. Peucker. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Canadian Cartographer*, 10(2):112–122, Dec. 1973.
11. M. A. Garrido, C. Iturriaga, A. Márquez, J. R. Portillo, P. Reyes, and A. Wolff. Labeling subway lines. In *ISAAC '01: Proceedings of the 12th International Symposium on Algorithms and Computation*, pages 649–659, Berlin, Heidelberg, 2001. Springer-Verlag.
12. J. E. Goodman and J. O'Rourke, editors. *Handbook of discrete and computational geometry*. CRC Press, Inc., Boca Raton, FL, USA, 1997.
13. F. Grabler, M. Agrawala, R. W. Sumner, and M. Pauly. Automatic generation of tourist maps. *ACM Trans. Graph.*, 27(3):1–11, 2008.
14. C. Iturriaga and A. Lubiw. Elastic labels around the perimeter of a map. *J. Algorithms*, 47(1):14–39, 2003.
15. G. Neyer and F. Wagner. Labeling downtown. In *CIAC '00: Proceedings of the 4th Italian Conference on Algorithms and Complexity*, pages 113–124, London, UK, 2000. Springer-Verlag.
16. S. Seibert and W. Unger. The hardness of placing street names in a manhattan type map. In *CIAC '00: Proceedings of the 4th Italian Conference on Algorithms and Complexity*, pages 102–112, London, UK, 2000. Springer-Verlag.
17. T. Strijk. *Geometric algorithms for cartographic label placement*. PhD thesis, 2001.
18. S. van Dijk, M. van Kreveld, T. Strijk, and A. Wolff. Towards an evaluation of quality for label placement methods. In *Proceedings of the 19th International Cartographic Conference, Ottawa, International Cartographic Association*, pages 905–913, 1999.
19. M. van Kreveld, T. Strijk, and A. Wolff. Point labeling with sliding labels. *Comput. Geom. Theory Appl.*, 13(1):21–47, 1999.
20. A. Wolff, L. Knipping, M. van Kreveld, T. Strijk, and P. K. Agarwal. A simple and efficient algorithm for high-quality line labeling. In P. M. Atkinson and D. J. Martin, editors, *Innovations in GIS VII: GeoComputation, chapter 11*, pages 147–159. 2000.
21. K.-L. Yu, C.-S. Liao, and D.-T. Lee. Maximizing the number of independent labels in the plane. In F. P. Preparata and Q. Fang, editors, *Frontiers in Algorithmics, First Annual International Workshop, FAW 2007, Lanzhou, China, August 1-3, 2007, Proceedings*, pages 136–147, 2007.