Luatodonotes: Boundary Labeling for Annotations in Texts

Philipp Kindermann, Fabian Lipp, and Alexander Wolff

Lehrstuhl für Informatik I, Universität Würzburg, Germany
http://www1.informatik.uni-wuerzburg.de/en/staff

Abstract. We present a tool for annotating Latex documents with comments. Our annotations are placed in the left, right, or both margins, and connected to the corresponding positions in the text with arrows (so-called *leaders*). Problems of this type have been studied under the name *boundary labeling*. We consider various leader types (straight-line, rectilinear, and Bézier) and modify existing algorithms to allow for annotations of varying height. We have implemented our algorithms in Lua; they are available for download as an easy-to-use Luatex package.

1 Introduction

Many word processing systems support annotations for the text. The most common case for this annotations are comments, which can be inserted in arbitrary positions inside the text. The comments themselves are placed as *labels* in the margin next to the text and connected to the corresponding position, called *site*, by a line called *leader*. The endpoint of a leader at a label is called a *port*. Such comments are available, for example, in LibreOffice (see Fig. 1) and Microsoft Word. This task can be expressed in the boundary labeling notion introduced by Bekos et al. [5]: the sites to be annotated lie inside the text area and the labels are to be placed outside the text area. They describe several types of leaders, such as straight-line leaders (s-leaders), rectilinear leaders with one bend (*po*-leaders) and rectilinear leaders with two bends (*opo*-leaders).

Previous work. Boundary labeling has been extensively investigated in the last few years, see a survey on the interaction between cartography and graph drawing [17]. For labels of uniform size, the problem is well-studied. Most algorithms try to minimize the total leader length. For s-leaders, it suffices to compute a minimum-weight perfect matching, which can be done in $O(n^{2+\varepsilon})$ time [1]. For *opo*-leaders, Bekos et al. [5] gave three different algorithms for the number of sides used by the labels, with running times $O(n \log n)$ (one-sided), $O(n^2)$ (two-sided), and $O(n^2 \log^3 n)$ (four-sided). Further, they presented an $O(n^2)$-time algorithm for *po*-leaders that lie on one side or on two opposite sides of the text. The result for *po*-leaders was improved by Benkert et al. [6] for the one-sided case. They gave an $O(n \log n)$-time algorithm for length minimization and an $O(n^3)$-time algorithm for a very general class of objective functions, including, for example,

C. Duncan and A. Symvonis (Eds.): GD 2014, LNCS 8871, pp. 76–88, 2014.
© Springer-Verlag Berlin Heidelberg 2014

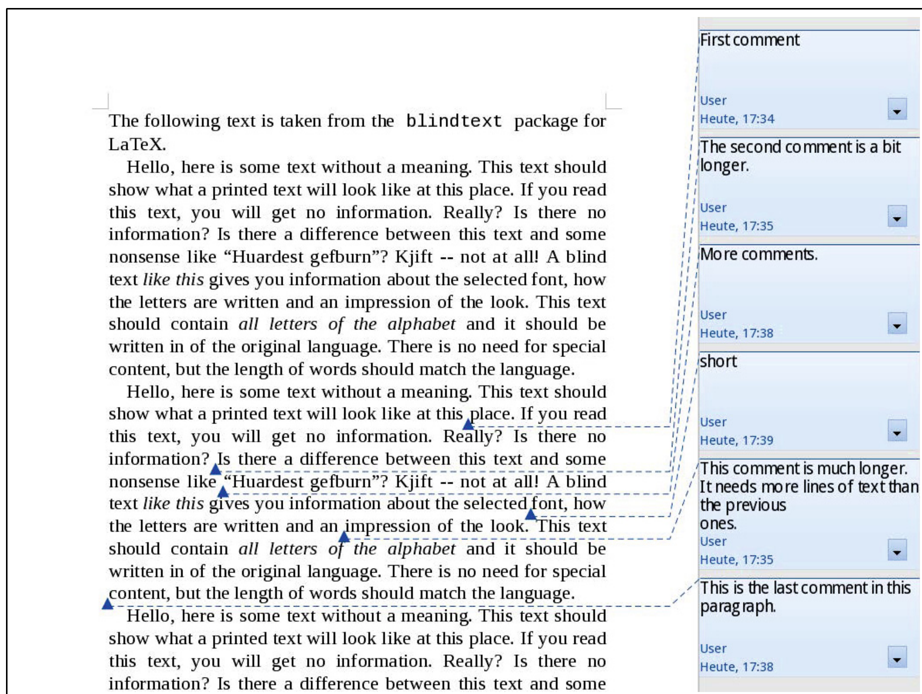

Fig. 1. Screenshot of comments in a document in LibreOffice 4.1.5

bend minimization. They also studied leaders that contain a diagonal part and gave an $O(n^2)$-time algorithm for the one-sided case. This result was extended by Bekos et al. [3] to more than one side. Recently, Kindermann et al. [10] gave the first efficient algorithms for *po*-leaders that decide whether an instance with labels on two adjacent, three, or four sides has a crossing-free solution (and, if yes, compute one).

Boundary labeling for non-uniform labels is still largely unexplored. Bekos et al. [4] showed that it is NP-hard to find a crossing-free labeling if the labels have to be placed on two sides (or two stacks on the same side). Huang et al. [9] considered a version of the problem that is always feasible: labels are placed into the right margin or into both margins, which are not bounded from below or above. For this model, *opo*-leaders, and labels of non-uniform size, they gave an $O(n^3)$-time algorithm that minimizes the total leader length in the one-sided case. For the two-sided case, they showed NP-hardness.

In this paper, we focus on comments for Latex documents. There are some packages that support the placement of textual comments in the margin, namely `todonotes` [12], `fixme` [16] and `fixmetodonotes` [2]. They have in common that they use Latex's `\marginpar` command to print the note as soon as the corresponding command is encountered in the source of the document. The drawback of this approach is that the positions of the following comments are not known

and cannot be considered when placing a note. The first label is placed beside the first site, and the following ones are placed below. Often it happens that a lot of free space is wasted above the topmost label, while the bottommost label is only partially visible (if at all), see Fig. 4a. Another disadvantage is that the `\marginpar` method cannot be used inside floating environments such as tables or algorithms. While the packages `fixme` and `fixmetodonotes` do not draw any leaders, `todonotes` uses *opo*-leaders. With this leader style it is hard to match a note to its corresponding site in the text when there are many comments in a short piece of text. A similar problem occurs with the leader style used by LibreOffice; see Fig. 1.

Other Latex packages support annotations as metadata for PDF documents, for example, `pdfcomment` [11]. The drawback of this package is that the user needs a compatible PDF viewer and that the annotations cannot be printed with the text. Packages such as `easy-todo` [14] don't place annotations in the margins, but insert a marker into the text and list all comments at the end of the document.

Our contribution. Our approach is different from all those listed above in that we collect the comments for a whole page and then compute a good placement for the labels. Of course, this computation needs more resources than the ad-hoc placement of the existing packages. Additionally, our Latex package supports different leader types, which the user can select when loading the package; see Section 2. We give several algorithms for non-uniform labels, most of which are extensions of existing algorithms for the one-sided case; see Section 3. We improve upon these basic algorithms by considering label clustering and the two-sided case; see Section 4. We have implemented all of our algorithms and have evaluated them experimentally; see Section 5. We conclude with some open problems; see Section 6. The package is available on CTAN:
`http://ctan.org/pkg/luatodonotes`

2 Implementation

We have implemented the algorithms in Lua and have bundled them into a Luatex package, which we call `luatodonotes`. The package requires the modern Tex-processor Luatex [8], which allows us to embed Lua code inside our Tex sources. This gives us access to a high-level programming language for implementing our label-placement algorithms. From the user's point of view, this does not change much. Luatex is part of every modern Tex installation, for example, Tex Live. Assuming such an installation, the difference in usage is simply that instead of calling (`pdf`)`latex`, the user calls `lualatex`.

Our package is based on the `todonotes` package (see Section 1). It is downward compatible as it provides the same commands to the user as the original package. Usage is quite simple: the user loads the package with the command `\usepackage{luatodonotes}` and inserts a comment into the text with the command `\todo{comment text}`.

Now, we describe how our package works. Wherever the user inserts a `\todo` command in the text, we store its position and its argument (that is, the comment) in a Lua list, but we do not print anything at this moment. When a page is finished ("shipped out" in Tex terminology), we compute the position of the labels and draw them. Before calling our label-placement algorithm, we have Tex determine the label heights. To determine the absolute positions of the sites, we use PGF/Ti*k*Z [15], a widely used Tex package for producing vector graphics. This package can locate the position of a site on the page where the `\todo` command was inserted, even when the command occurs inside a floating environment (such as a figure or a table).

For each label, the placement algorithm computes the absolute coordinates on the page on which the label is to be placed. Then, we use Ti*k*Z to draw the labels and the leaders that connect the labels with their corresponding sites in text. Finally, a mark is placed at each site. This modular design simplifies the implementation of new algorithms and makes the package extensible.

The size and position of the rectangles that contain the label texts depend on the current page layout. We provide options to control the distances between the labels and the text (`distanceNotesText`) and between the labels and the border of the page (`distanceNotesPageBorder`). The algorithms can place labels in the left and in the right margin (see Section 4), but a margin is used only if it is wide enough to accomodate a label, that is, if the label can be at least of width `minNoteWidth`.

When loading the package with `\usepackage{luatodonotes}`, optional arguments can be specified in square brackets. The most relevant options are (a) the algorithm for label placement (`positioning`) and (b) the leader type (`leadertype`). Other options control the layout: the minimum vertical distance of the labels (`interNoteSpace`), the distance from the contents of the label to its border (`noteInnerSep`) and the color of the leaders (`linecolor`).

3 Algorithms for Label Placement

In the following, the algorithms are categorized by the leader type that they support. In principle, our package allows the user to combine any label-placement algorithm with any leader type. Still, some algorithms have been designed with certain leader types in mind. Other combinations will probably yield unwanted results, such as label overlap or crossing leaders.

In the descriptions of our algorithms below, we assume that labels are placed on the left side of the text, but this is not a restriction of our actual implementations. Additionally, we try to place the labels without gaps between them, while in reality we want to preserve a certain minimum distance between them. Clearly, this is easy to achieve.

3.1 *s*-Leaders

Our algorithms designed for *s*-leaders have a common property: they draw the leaders without crossing each other. Their common objective is to place the labels

one below the other on the boundary while avoiding gaps between them. They differ in the position of the ports, that is, the position on the label boundary to which the leader is attached. A pleasant position for the port would be the center of the right side of the label. Unfortunately, we don't have an algorithm that can place the labels without gaps using this port position. We don't even know whether every instance of site positions and label heights is feasible w.r.t. these criteria; see Section 6.

We don't give algorithms that minimize the total leader length here, but concentrate on drawings without crossings. The clustering approach described in Section 4 can decrease the leader length as labels are placed closer to their corresponding sites.

NorthEast. We use an algorithm of Bekos et al. [5] for fixed labels, which can easily be adopted to our problem with labels of non-uniform heights: The upper right corner of each label is used as its port. The labels are placed consecutively from the top of the page to the bottom. In each step, we emit a ray from the port of the next label vertically to the top and rotate it clockwise until the first unlabeled site is hit. Obviously, by connecting this site to a label at the current position, we don't hide any other sites and can label the remaining sites without crossings.

NorthEastBelow. This algorithm is based on the preceding one. The difference is that we lower the port from the corner by a constant offset. In our opinion the result looks better when the leader is not attached directly at the corner. A good value for this offset is half of the height of the smallest label. As we know the position for each port while placing the label, we can still use the ray construction of the preceding algorithm to place the labels without spaces between them.

East. In this algorithm the port of every label is located at the center of its right side. When we try to find the next unlabeled site to be labeled, we do not know the port position as it depends on the height of the label. Therefore, we cannot use the ray construction from the previous algorithms. Algorithm 1 is a heuristic that guarantees crossing-free leaders while trying to avoid gaps between the labels. It can usually handle real-world inputs without additional gaps.

An instance that is not handled optimally by the heuristic is depicted in Fig. 2. The sites can be labeled without gaps when placing the labels in the order 2, 1, 3. As mentioned above it is an open question if this is possible for all instances.

3.2 Bézier Curves as Leaders

We base our Bézier curves on s-leaders using a force-directed algorithm described by Fink et al. [7]. We use cubic Bézier curves that are required to enter the port at the label horizontally. This means that the first control point has to stay on the same horizontal line as the port and can only be moved to the left or the

Algorithm 1. Placing labels using east anchors

```
   Input: p_1, ..., p_n are the sites in the text
   Output: y-coordinate y_1, ..., y_n of the top edge of each label
1  P ← {p_1, ..., p_n}
2  L ← [ ] // list contains labels in the order in which they are
   placed
3  lastY ← 0
4  while P ≠ ∅ do
5      H ← {height(p_j) | j = 1, ..., n} // H is in ascending order
6      foreach h ∈ H do
7          place a label of height h directly below the last label
8          emit a ray from the port of the newly placed label
9          i ← index of first point in P that is hit by the ray (rotated clockwise)
10         if height(p_i) ≤ h then
11             break
12     y_i ← lastY − (h − height(p_i))/2
13     L.add(p_i)
14     P ← P − {p_i}
15     lastY ← y_i − height(p_i)
   // Postprocessing: try to shrink gaps
16 foreach l ∈ L do
17     if there is a gap above l then
18         move l up as far as possible without creating any new intersection
           between leaders
```

right. The second control point is always placed in the center between the first control point and the site.

In the first iteration of the algorithm, the control points are placed on the endpoints of the leader, that is, it starts as a straight line. Later, the first control point of each curve is moved by applying forces to it. We use a force that pulls the control point to its optimal point, which is computed beforehand and usually yields a good-looking curve. Other forces try to increase the distance between curves. In every iteration the forces on every point are limited by the distance to the nearest curve to inhibit new intersections between leaders. Therefore, the algorithm guarantees crossing-free Bézier curves when starting with straight-line leaders without intersections.

The runtime of this algorithm is dominated by the calculation of the distances between each pair of curves. This calculation is done by an approximation of the curves. We need the distances to update the forces in every iteration.

3.3 *opo*-Leaders and *os*-Leaders

Positioning the labels for crossing-free *opo*-leaders is simple as Bekos et al. [5] show: we place the labels in the order given by the y-coordinates of their sites.

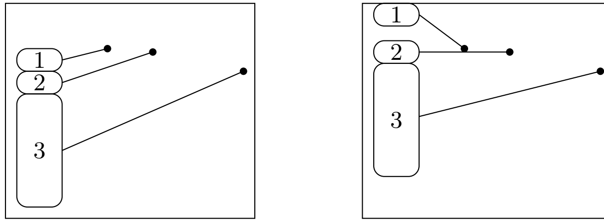

Fig. 2. An instance where the East algorithm does not yield a drawing without gaps. Left: label positions before postprocessing; Right: after postprocessing.

Sites with identical y-coordinates are processed from left to right. The vertical parts of the leaders are drawn in the *track routing area*, that is, the vertical strip between text and labels. The width of this track routing area is specified using the option `routingAreaWidth` of the package. We split the labels into groups, with labels sharing a common vertical segment being put in the same group. This can be done by a simple linear-time algorithm. Thus the vertical segments of the leaders in each group must be placed side by side. We draw the vertical segments in one group with equal distances between them, using the whole width of the track routing area.

The algorithm is even easier for *os*-leaders, a leader style that was not discussed until now. We list it here because this is the style that, for example, LibreOffice uses (see Fig. 1). Labels are placed in the same order as for *opo*-leaders. For the leaders, we connect the site with a horizontal line segment that extends to a fixed x-coordinate inside the margin. Then we connect the end of the horizontal segment to the label's port with a straight-line segment.

3.4 *po*-Leaders

Benkert et al. [6] developed an algorithm to compute an optimal crossing-free labeling using *po*-leaders with respect to an arbitrary badness function. This algorithm, which uses a dynamic programming approach, is designed for uniform labels only. It needs $O(n^3)$ running time and $O(n^2)$ space.

For our application, we extend the algorithm of Benkert et al. to non-uniform labels. To be able to work with the arbitrary heights of the labels, we need to raster the page, that is, we define the y-coordinates on which labels may be placed. Our algorithm yields a labeling respecting this raster with minimum total leader length. The height of the raster can be chosen using the parameter `rasterHeight` of the Latex package. The port for each label can be chosen arbitrarily. In the following, the ports are fixed to the center of the right side of the labels.

Let $p_1, \dots, p_n$ denote the sites from top to bottom and let $r_1, \dots, r_m$ be the slots obtained by rasterizing the page from top to bottom. We use a 5-dimensional table in our dynamic program. The entry $T[t, b, \tau, \beta, k]$ represents the minimum length of a labeling of the k leftmost sites in $\{p_t, \dots, p_b\}$ using

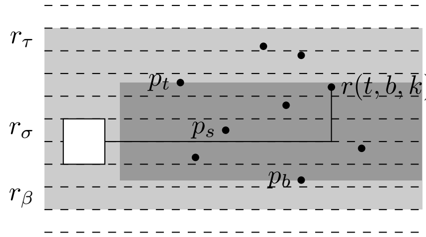

Fig. 3. The labeling problem for $T[t, b, \tau, \beta, k]$ is split into two independent subproblems by fixing the label position of $r(t, b, k)$. The dashed lines show the raster slots. The light gray area indicates the slots from r_τ to r_β. The dark gray area shows the sites between p_t and p_b.

only the raster slots $r_\tau, \dots, r_\beta$. The labels must lie completely inside the given slots.

Let $r(t, b, k)$ the k-th point from the left in the set $\{p_t, \dots, p_b\}$. The length of the shortest *po*-leader from the site p to its corresponding label beginning in slot r_σ is denoted by $l^*(p, \sigma)$. The entries of the table are computed using the following decomposition (illustrated in Fig. 3):

$$T[t, b, \tau, \beta, k] = \min_{\text{feasible } \sigma \in \{\tau, \dots, \beta\}} l^*(r(t, b, k), \sigma) + T[t, s, \tau, \sigma - 1, k_1] + T[s + 1, b, \sigma + h, \beta, k_2]$$

In this formula p_s is the lowest point that lies above the leader arm (the horizontal part of the leader), when the label for $r(t, b, k)$ is placed at slot r_σ. Let h the height of this label. The number of sites from $\{p_t, \dots, p_b\}$ lying left of $r(t, b, k)$ and above resp. below the leader arm is denoted by k_1 resp. k_2.

A position for the label is feasible, if both partial solutions (above and below the leader arm) are feasible, that is, there are enough slots to label the contained sites.

Clearly, $T[1, n, 1, m, n]$ is the optimal labeling of the whole instance. With this algorithm we can compute an optimal solution in $O(n^4 m^3)$ time with $O(n^3 m^2)$ space, where n is the number of sites to be labeled and m is the number of slots in the raster on the page.

Avoid overlappings with text lines. The algorithm described above does not take the position of the text lines of the document into account. Thus it can happen that a line gets striked out by the horizontal segment of a leader. We modified the algorithm to move the port up or down by a small offset to avoid such overlappings and place the leader into the gap between the lines.

It is quite hard to determine the positions of the lines in Tex because they are not fixed until the document is written to the output file. But in Luatex we can modify the linebreaking algorithm such that it inserts special nodes into the data structures of Tex that write the position of every line into a text file when

Algorithm 2. Clustering labels

```
Input: p_1, ..., p_n are the sites in the text ordered by their y-coordinate from
       top to bottom
Output: list of clusters S
1  S ← [{p_1}, {p_2}, ..., {p_n}]
2  i ← 1
3  while i ≤ #S − 1 do
4      if clustersIntersect(S[i], S[i + 1]) then
5          S[i] ← S[i] ∪ S[i + 1]
6          S.delete(i + 1)
           // as the size of stack i has increased we check again for
              intersection with the previous stack in next iteration
7          i ← max{1, i − 1}
8      else
9          i ← i + 1
10 return S
```

typesetting the page. In a second Tex run we can read the line positions from this file and use them for our algorithm.

4 Improvements

In this section we discuss some general improvements implemented in our package that can be used by every algorithm described in the previous section.

Label clustering. Most of the algorithms described in the previous sections place labels in a single stack (that is, without gaps between them) beginning at the upper margin of the page. This can produce unnecessarily long leaders, for example when the text contains a single site near the end of the page. We split the labels into separate clusters and place each of them near the corresponding sites in the text. An algorithm for clustered labeling is also described by Nöllenburg et al. [13]. Our approach is simpler but slower.

To group the labels into clusters we use Algorithm 2. It repeatedly joins adjacent clusters as long as they intersect each other. To test if two clusters intersect we place the contained labels as a stack each beneath the arithmetic mean of the sites in the cluster. The clusters intersect if their corresponding stacks overlap.

The positioning algorithm is executed independently for each of the identified clusters. The intended position is passed to the algorithm as a parameter.

Two-sided label placement. On some page layouts there is enough space to place labels in the margins on the left *and* the right side of the text. We have to decide for each label on which side of the text it should be placed. Our approach is to split the sites by a vertical line through the text. The sites which are left of this

split line are labeled on the left side, those right of the split line are labeled in the right margin. There are several ways to determine the position of this split line. We use a weighted median to split the sites such that the sum of the label heights on the left side is approximately equal to that of the right side. With this algorithm it is not an issue if the widths of the two margins are different (which means that the height of a label depends on the side on which it is placed).

5 Experimental Results

We compare the leader styles presented in the previous sections on an example document with nine comments in it. This document stays the same, only the options of our package are modified to switch between the available algorithms. We used the label clustering approach described in the previous section for all examples except for that of the *po*-leader algorithm. For comparison, we also processed the document with the `todonotes` package (see Fig. 4a).

The NorthEastBelow algorithm for *s*-leaders (Fig. 4b) is straight-forward and fast. It is easy for the reader to match the sites to their corresponding label. Using Bézier curves (Fig. 4c) instead of the straight-line leaders yields a more aesthetic result with the disadvantage of a significantly higher runtime caused by the iterations of the force-directed algorithm. Using two-sided label placement with the same leader type produces shorter leaders because the labels can be placed closer to their site. Especially in text segments with a lot of comments this makes the relationship between sites and their labels clearer.

Our algorithm for *po*-leaders (Fig. 4d) has a high asymptotic runtime and space consumption. But in practice when there are only few comments per page this is not an issue. Among the algorithms we implemented, this is the only algorithm minimizing the total leader length.

The *opo*-leaders and *os*-leaders are available mainly for comparison. Clearly, it gets hard for the reader to match sites to their labels on pages with many comments. In particular, if several sites are in the same line it is hard to tell the matching between sites and labels. On the other hand the leaders only run between the lines and in the track routing area and thus don't disturb the text.

The running times of Luatex with the different leader types for some example documents are shown in Table 1. Note that Documents 2 and 3 with 15 resp. 25 comments on one page are quite unrealistic. When using two-sided label placement both sides are processed independently and thus the algorithm for *po*-leaders becomes feasible again. The measured times are for a single run of Tex only. When the absolute position of a site of a label changes, a second run is needed. When we deactivate our package, processing still needs 1.4 seconds. This means that *s*- and *opo*-leaders cause only small extra cost compared to a standard Latex run. With the classical `todonotes` package processing needs about 1.8 seconds, too.

We would have liked to give a numerical comparison of the drawing quality of the different algorithms, but it is not obvious how to find an appropriate indicator for the quality that is suitable for all of the available leader types. So we ask the reader to inspect Fig. 4 visually.

(a) `todonotes` with *opo*-leaders

(b) `luatodonotes` with *s*-leaders

(c) `luatodonotes` with Bézier leaders

(d) `luatodonotes` with *po*-leaders

Fig. 4. An example document with notes produced by the `todonotes` package (a) and the `luatodonotes` package (b–d) with different leader styles

Table 1. Running times of the different label styles on three one-page documents D1, D2, and D3 (in CPU seconds). The times were measured using a Intel Core 2 Duo E8400 with 3.0 GHz. D1 is the instance with 9 comments shown in the figures above. D2 has 15 comments, D3 has 25. For each document, we report two running times; for label placement into one margin vs. both margins. We use a raster height of 1 cm for *po*-leaders, resulting in 28 horizontal strips. We couldn't use *po*-leaders for D3 with one margin because the algorithm needed too much memory. For comparison we also give the running times for the classical `todonotes` package (which does not support placing labels in both margins) and the running times for the document without loading the `luatodonotes` package.

Document	D1		D2		D3	
Number of margins	1	2	1	2	1	2
s-leaders	1.8	1.7	1.9	1.9	2.2	2.2
Bézier leaders	5.7	5.4	33.2	11.1	322.9	116.3
po-leaders	4.8	3.0	17.7	6.2	—	27.6
po-leaders avoiding text lines	7.0	4.0	26.8	9.5	—	42.4
opo-leaders	1.8	1.7	1.9	1.9	2.2	2.2
classical `todonotes`	1.9		2.2		2.6	
without `luatodonotes`	1.4		1.4		1.3	

6 Conclusion and Open Problems

All our algorithms turned out to work well in practice—some of them cannot process too many labels on a single page. Using both margins helps in terms of speed. By visual inspection we reached the conclusion that *s*-leaders or Bézier leaders work better than the *os*-leaders used by other type-setting programs. The reason may be that the reader's eye can follow leaders without bends more easily. It would be interesting to verify this in a user study. With the modular design of our Latex package it is easy to improve the label-placement algorithms or add additional ones.

An interesting theoretical problem remains open: Given an instance with non-uniform label heights, is it always possible to place the labels without gaps so that *s*-leaders do not cross each other even if we insist that the ports are centered vertically at each label?

We have some ideas for further improvements of our package. The force-directed Bézier curve algorithm is quite slow at the moment. We think that we could speed up the computation of the distances between curves by doing a rough estimate first and computing the fine approximation only when needed. It would be interesting to transform the *po*-leaders into Bézier curves. As our algorithm yields a length-minimal *po*-labeling this could produce a shorter leader length than our approach with *s*-leaders. But it is not clear how to inhibit intersections between the curves.

Admittedly, our dynamic program for *po*-leaders is quite slow. Can we save time by computing labelings that are just feasible rather than length-minimal? For the other leader types, on the contrary, it would be interesting to minimize

the total leader length. Such algorithms are known only for the case of uniform labels. When minimizing the leader length in the two-sided case, one could also try to improve the approach for partitioning the labels.

References

1. Agarwal, P.K., Efrat, A., Sharir, M.: Vertical decomposition of shallow levels in 3-dimensional arrangements and its applications. SIAM J. Comput. 29(3), 912–953 (1999)
2. Barabucci, G.: fixmetodonotes (2013), `http://www.ctan.org/pkg/fixmetodonotes`
3. Bekos, M.A., Kaufmann, M., Nöllenburg, M., Symvonis, A.: Boundary labeling with octilinear leaders. Algorithmica 57(3), 436–461 (2010)
4. Bekos, M.A., Kaufmann, M., Potika, K., Symvonis, A.: Multi-stack boundary labeling problems. In: Arun-Kumar, S., Garg, N. (eds.) FSTTCS 2006. LNCS, vol. 4337, pp. 81–92. Springer, Heidelberg (2006)
5. Bekos, M.A., Kaufmann, M., Symvonis, A., Wolff, A.: Boundary labeling: Models and efficient algorithms for rectangular maps. Comput. Geom. Theory Appl. 36(3), 215–236 (2007)
6. Benkert, M., Haverkort, H.J., Kroll, M., Nöllenburg, M.: Algorithms for multi-criteria boundary labeling. J. Graph Algorithms Appl. 13(3), 289–317 (2009)
7. Fink, M., Haunert, J.H., Schulz, A., Spoerhase, J., Wolff, A.: Algorithms for labeling focus regions. IEEE Trans. Vis. Comput. Graphics 18(12), 2583–2592 (2012)
8. Hagen, H., Henkel, H., Hoekwater, T.: Luatex (2007), `http://www.luatex.org`
9. Huang, Z.-D., Poon, S.-H., Lin, C.-C.: Boundary labeling with flexible label positions. In: Pal, S.P., Sadakane, K. (eds.) WALCOM 2014. LNCS, vol. 8344, pp. 44–55. Springer, Heidelberg (2014)
10. Kindermann, P., Niedermann, B., Rutter, I., Schaefer, M., Schulz, A., Wolff, A.: Two-sided boundary labeling with adjacent sides. In: Dehne, F., Solis-Oba, R., Sack, J.-R. (eds.) WADS 2013. LNCS, vol. 8037, pp. 463–474. Springer, Heidelberg (2013)
11. Kleber, J.: pdfcomment (2012), `http://www.ctan.org/pkg/pdfcomment`
12. Midtiby, H.S.: todonotes (2012), `http://www.ctan.org/pkg/todonotes`
13. Nöllenburg, M., Polishchuk, V., Sysikaski, M.: Dynamic one-sided boundary labeling. In: Proc. 18th SIGSPATIAL Int. Conf. Adv. Geogr. Inform. Syst. (ACM-GIS), pp. 310–319. ACM (2010)
14. Rada-Vilela, J.: easy-todo (2014), `http://www.ctan.org/pkg/easy-todo`
15. Tantau, T.: PGF and Ti*k*Z – Graphic systems for TeX, `http://www.sourceforge.net/projects/pgf` (accessed April 2, 2014)
16. Verna, D.: Fixme (2013), `http://www.ctan.org/pkg/fixme`
17. Wolff, A.: Graph drawing and cartography. In: Tamassia, R. (ed.) Handbook of Graph Drawing and Visualization, ch. 23. CRC Press (2013)