Improved algorithms for length-minimal one-sided boundary labeling

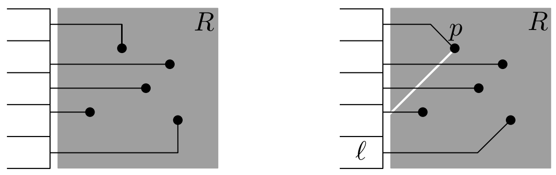
Marc Benkert* Martin Nöllenburg⋆

Abstract

We present algorithms for labeling n points that are contained in a rectangle R by labels that lie on one side of R. The points are connected to their labels by non-intersecting curves (leaders) that each have at most one bend. We consider two types of curves: rectilinear leaders, called *po-leaders*, and leaders that consist of a horizontal and a diagonal segment, called *do-leaders*. To obtain a good readability of the labeling we minimize the total leader length. For the *po*-leaders we give an $O(n \log n)$ and for *do*-leaders an $O(n^2)$-time algorithm. This type of labeling has applications for geographic maps or illustrations in medical atlases in which the labels should not be inserted directly because labels would obscure important information or if points lie too dense.

1 Introduction

Our work ties up to a work of Bekos et al. [2]: for n points contained in a rectangle R and n labels that lie either on one, two or all four sides of R they gave several algorithms for different types of polygonal lines that are allowed to connect the points with the labels. For a good readibility of the labeling they demand that the leaders should be non-intersecting. Further criteria that serve for a good quality are minimizing the total leader length and the total number of bends. One of the leader types that they introduced are the *po*-leaders: the leader starts with a (possibly empty) vertical segment followed by a horizontal segment that connects to the label, see Figure 1(a). For the case that the labels are located only on one side of R, they gave a quadratic algorithm that computes the length-minimum labeling with *po*-leaders. Ali et al. [1] propose several heuristic labeling methods using straight-line and rectilinear leaders. They first compute an initial labeling and then eliminate intersections between leaders. In Section 2 we improve the *po*-leader result of Bekos et al. by giving an $O(n \log n)$ algorithm. Furthermore, we introduce the notion of a *do*-leader. The only difference to a *po*-leader is that the leader starts with a diagonal segment of fixed angle oriented towards the label, see Figure 1(b). To our best knowledge, there is no literature yet that algorithmically deals with *do*-leaders. In practice the *do*-leaders seem to produce nicer labelings because their smoother shape makes the comprehension of the assignment from points to labels easier. In Section 3 we present a quadratic algorithm that computes the length-minimum *do*-labeling with labels on one side.



(a) *po*-leaders. (b) *do*-leaders.

Figure 1: Valid labelings for labels on the left side.

2 *po*-leaders

For simplicity we assume that the labels are uniform and located on the left side of R. We briefly sketch Bekos et al. quadratic algorithm [2] as we will prove the correctness of our algorithm by showing that it produces exactly the same labeling.

Their algorithm proceeds in two steps: first, they produce a non-crossing-free labeling that obviously minimizes the total leader length. This labeling is simply found by sorting the points according to their y-coordinate and then assigning the bottommost point to the bottommost label, the second bottommost point to the second bottommost label and so on. Then, in the second step, they purge all crossings by changing the assignment of two points at a time in an appropiate order. In this second step the total leader length does not change, see Figure 2. Hence, their output is a valid length-minimum labeling. However, in the second step their algorithm potentially has to deal with a quadratic number of crossings which is the bottleneck for the running time.

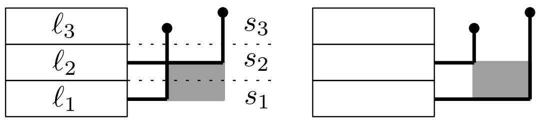


Figure 2: Exchanging the labels of two intersecting leaders without changing the total length.

The key idea for bringing the running time down to $O(n \log n)$ is to couple these two steps and to not

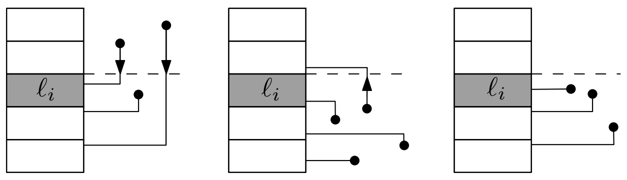
*Fakultät für Informatik, Universität Karlsruhe, P.O. Box 6980, D-76128 Karlsruhe, Germany, `http://i11www.iti.uni-karlsruhe.de/research/geonet/`, GeoNet is supported by grant WO 758/4-2 of the DFG.

make an assignment for a point until we know that this assignment will not produce any crossings in the remainder of the algorithm. We accomplish this by a sweep-line algorithm.

Let $\ell_1, \ldots, \ell_n$ be the numbering of the labels from bottom to top and let $s_1, \ldots, s_n$ denote the horizontal strips to the right of the according labels, see Figure 2. In a preprocessing step that takes $O(n \log n)$ time we determine point lists $P_1, \ldots, P_n$, where P_i contains exactly the points in s_i, and sort each list according to increasing y-coordinate. Throughout the algorithm we maintain a list L of points that are ordered according to increasing x-coordinate, L contains the points that, for some state of the algorithm, all have to be labeled by a label above or below the current sweep line, initially L is empty. For the sweep itself, there is one event point for every label, namely the horizontal line through the upper horizontal side of the label rectangle. We denote the event point for ℓ_i by $\hat{\imath}$ and the number of points in $s_1 \cup \cdots \cup s_i$ by $n_i^{\cup}$. We distinguish three possible states for an event point $\hat{\imath}$:

Need	if $i > n_i^{\cup}$,
Surplus	if $i < n_i^{\cup}$ and
Equilibrium	if $i = n_i^{\cup}$.

The three states are illustrated in Figure 3. *Need* means that there are too few points in $s_1 \cup \cdots \cup s_i$ for labeling $\ell_1, \ldots, \ell_i$. *Surplus* means that there are too many points in $s_1 \cup \cdots \cup s_i$ for labeling $\ell_1, \ldots, \ell_i$ and *Equilibrium* means that there are exactly i points in $s_1 \cup \cdots \cup s_i$. Note that then, in the length-minimum labeling, these points are assigned to the labels $\ell_1, \ldots, \ell_i$. We can see this by Bekos et al.' algorithm: in their first step the i points are obviously assigned to $\ell_1, \ldots, \ell_i$, meaning that they could have crossings amongst each other but never with other points above. Since Bekos et al. switch labels only of points whose leaders intersect, this means that in the end, these i points remain assigned to $\ell_1, \ldots, \ell_i$.



(a) Need (b) Surplus (c) Equilibrium

Figure 3: The three possible states at event point $\hat{\imath}$.

According to the state that we found at $\hat{\imath}$ we do the following: For *Need* we do nothing at all and proceed with the next event point. The label ℓ_i will get its assigned point during a *backtracking* later on.

For *Surplus* we check the state of the previous event point. If it was *Equilibrium* we assign the bottommost point in P_i to ℓ_i, insert the remaining points of P_i in the then empty list L, remove the first point of L and assign it to ℓ_{i+1}. If it was *Surplus*, label ℓ_i will already have its point assigned by the processing of $\hat{\imath} - 1$. We insert the points P_i into L. After that L must be non-empty because either there were points in P_i or L was still non-empty after processing $\hat{\imath} - 1$. Again, we remove the first point of L and assign it to ℓ_{i+1}. If it was *Need*, we check for which point p in s_i we have an 'artificial' *Equilibrium* and assign p to ℓ_i. We insert the points below p into L and *backtrack*: we set a counter c to $i - 1$, as long as L is not empty we do the following: we assign the first point of L to ℓ_c and delete it from L. Then we insert the points P_c into L and decrease c by one. After the backtrack we insert the points above p into L and proceed as above.

For *Equilibrium* we look at P_i. If $P_i = \emptyset$ the previous state was *Surplus* and ℓ_i will already have its point assigned. we have nothing to do. If $|P_i| = 1$, we assign the point in P_i to ℓ_i. If $|P_i| > 1$, the previous state was *Need*, we assign the topmost point of P_i to ℓ_i, insert the remaining points in P_i into the list L and *backtrack* as described above.

Theorem 1 *For labels on one side, the length-minimum labeling using po-leaders can be computed in $O(n \log n)$ time requiring $O(n)$ space.*

Proof. Obviously, each point is inserted in and deleted from L at most once, which establishs the running time since an insertion in the ordered list L is in $O(\log n)$. The linear space requirement is also obvious. It remains to prove that the algorithm finds a valid labeling and that this labeling has indeed minimum length. As mentioned earlier we do this by showing that our algorithm computes exactly the same labeling as Bekos et al.' algorithm. We omit the details but point out that taking the first point of L, i.e. the point with minimum x-coordinate in L, for assigning it to ℓ_c (backtracking) or to ℓ_{i+1} (treating a *Surplus* state) is necessary for the prevention of producing crossings in the further run of the algorithm. □

3 *do*-leaders

For simplicity we assume that the labels are uniform and located on the left side of R. We note that the algorithm will work for any fixed angle for the diagonal segments between 0° and 90° to the x-axis.

We cannot use the same approach as for the *po*-leaders simply as not every point can connect to any label by a *do*-leader, look e.g. back to Figure 1(b) where p cannot connect to ℓ. Roughly speaking we will use a generalization of Bekos et al.' algorithm for the *po*-leaders that takes these restrictions into account.

We start by introducing necessary conditions for the existence of a *do*-labeling and show how to algorithmically make use of them. In the end, we constructively get that the conditions are even sufficient.

Each label ℓ induces a funnel-shaped subregion R_ℓ in which all points that could be assigned to this label are located. The arrangement of all these regions defines $O(n^2)$ cells, see Figure 4. All points in the same cell of this arrangement can connect to the same set of labels and these sets are distinct for any two cells.

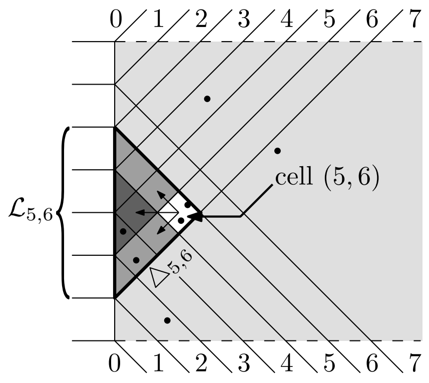


Figure 4: The cell arrangement.

A cell itself is the intersection of an ascending and a descending diagonal strip and after numbering these strips we can index each cell, e.g. the white cell in Figure 4 has index $(5,6)$, when we take the index of the descending strip as first coordinate. For a cell (i,j) we denote the label set that can be reached by $\mathcal{L}_{i,j}$ and the smallest triangle bordering to $\mathcal{L}_{i,j}$ and containing (i,j) by $\triangle_{i,j}$, see Figure 4. Now, a necessary condition for the existence of a *do*-labeling is obviously that the number $n_{i,j}$ of points in $\triangle_{i,j}$ does not exceed the number of labels in $\mathcal{L}_{i,j}$ which is $i+j-n$. Otherwise there will be unassigned points left over in $\triangle_{i,j}$ that cannot connect to any other labels beside $\mathcal{L}_{i,j}$. We say that $i+j-n$ is the *level* of cell (i,j).

Lemma 2 *There can only be a valid do-labeling if for each k-level cell (i,j) it holds that $n_{i,j} \leq k$.*

We can check these necessary conditions in $O(n^2)$ time. For this we have to compute all numbers $n_{i,j}$: initially we set each $n_{i,j}$ to zero. For each input point we determine its containing cell (i,j) and increment $n_{i,j}$ by one. Then, each $n_{i,j}$ gives the number of points in the cell (i,j) but we aim for the number of points in $\triangle_{i,j}$. We traverse the cells in increasing order of their levels. Apparently, all 1-level cells already contain the desired values, for all other cells $n_{i,j}$ is updated based on three predecessor values (see Figure 4):

$$n_{i,j} \leftarrow n_{i,j} + n_{i,j-1} + n_{i-1,j} - n_{i-1,j-1}.$$

This counts each point in $\triangle_{i,j}$ exactly once. The time complexity per cell is obviously constant.

Now, we present our algorithm to compute the length-minimum labeling. We assume that we computed the numbers $n_{i,j}$ in a preprocessing and neither of the necessary conditions has been violated. For a k-level cell (i,j) for which $n_{i,j}$ is k we say that $\triangle_{i,j}$ is *full*, meaning that in any valid *do*-labeling each of the $n_{i,j}$ points in $\triangle_{i,j}$ connects to a label in $\mathcal{L}_{i,j}$. For the algorithm we generalize the above definition: a triangle $\triangle_{i,j}$ is full if the numbers of points in $\triangle_{i,j}$ and labels in $\mathcal{L}_{i,j}$ that have not been assigned yet match. From now on we call such points and labels *open*.

The algorithm traverses the cells in increasing order of their levels and for each level from bottom to top. Whenever we find a k-level cell (i,j) for which $\triangle_{i,j}$ is full, we call the subroutine *complete*$(\triangle_{i,j})$ which computes a length-minimum valid labeling for the remaining open items in $\triangle_{i,j}$. Then, $\triangle_{i,j}$ is marked as completed. Eventually the traversal will examine the n-level cell (n,n) and if not all points have been assigned yet, an assignment for the remaining open points and labels will be found.

In the procedure *complete*$(\triangle_{i,j})$ we process the open labels from bottom to top. Basically for each open label ℓ the point that we assign to ℓ is the first open point that we find when we sweep $R_\ell \cap \triangle_{i,j}$ by a horizontal line from bottom to top. If the placement of the leader inserts any crossing with earlier drawn-in leaders we purge the crossings by flipping assigned labels without changing the total leader length.

However, we have to pay attention during the completion of a triangle $\triangle_{i,j}$: each time we assign a point that does not lie in a 1-level cell, we artificially shift this point into the 1-level cell adjacent to the assigned label, see Figure 7. This decreases the number of open labels for incompleted subtriangles of $\triangle_{i,j}$ while the number of open points in them stays the same, thus, these triangles can become full. If this happens we have to bring the completion of these recently filled subtriangles forward to the usual completion of $\triangle_{i,j}$.

For describing the full operation mode of *complete*$(\triangle_{i,j})$ we have to distinguish two cases:

complete$(\triangle_{i,j})$**:** First, we traverse the incompleted cells of $\triangle_{i,j}$ by a breadth-first search starting from (i,j). This yields lists of the remaining open points and labels in $\triangle_{i,j}$. If the lists of points and labels are empty we mark (i,j) as completed and are done, otherwise we sort both lists according to increasing y-coordinate.

Together with the BFS we purge redundant cells: due to already completed subtriangles of $\triangle_{i,j}$ cells can have become equivalent in the sense that they now can reach the same set of open labels. We merge these equivalent cells and assign the number and level of the topmost-level cell to the newly emerged cell. Obviously, this maintains the number of points and labels in the triangle associated with the new cell, see Figure 5. This step is indispensable for the maintenance of a quadratic running time as the update of the cell entries that we have to do when we make an assignment later on would cause the runtime to get super-quadratic if the number of cells was quadratic.

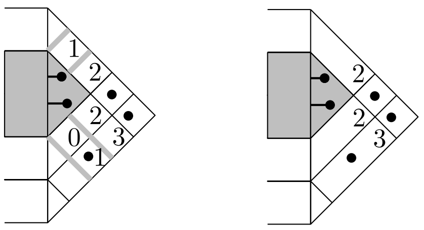


Figure 5: Merging redundant cells.

After finishing these initilizations we start with assigning points to the open labels. For this, we sweep the labels from bottom to top. For an open label ℓ we do the following: we traverse the list of open points and assign the first point p that we find and that is in R_ℓ to ℓ. We remove ℓ and p from the lists of open labels and points. If the leader from p to ℓ intersects earlier drawn-in leaders we take the leader of the topmost label among them and flip the assigned points with ℓ, we repeat this step until there are no crossings anymore, see Figure 6.

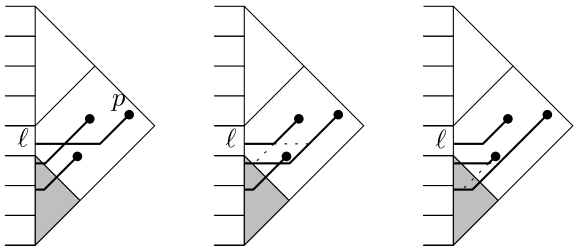


Figure 6: Purging crossings after assigning $p \rightarrow \ell$.

After making an assignment, we update the cell structure and data of $\triangle_{i,j}$. For cells that have become redundant by the assigment this works analogously as for the initilization. For the numbers $n_{i',j'}$ we have shifted the assigned point from its original cell to the cell c_ℓ adjacent to ℓ. We trace the leader from ℓ back to p and update the affected cells accordingly, see Figure 7. This update can cause subtriangles (i', j') to become full. If this happens we have to bring their completion forward. For this, we prepare the lists of open labels and points in $\triangle_{i',j'}$ and start the subroutine *subtriangle-complete*$(\triangle_{i',j'})$. Then, we mark (i', j') including the according points and labels as completed and proceed with the completion of $\triangle_{i,j}$.

Finally, after the last open label in $\triangle_{i,j}$ is assigned we mark $\triangle_{i,j}$ as completed.

subtriangle-complete$(\triangle_{i',j'})$**:** As before, only the lists of open points and labels are handed over by the overall procedure.

Theorem 3 *For labels on one side, a valid length-minimum labeling using do-leaders can be computed in $O(n^2)$ time requiring $O(n^2)$ space, if there is any.*

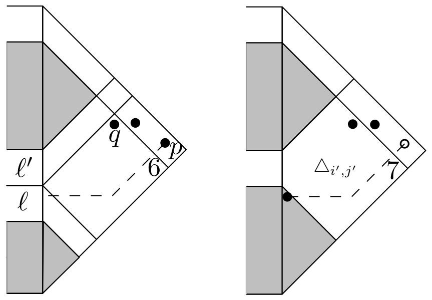


Figure 7: $\triangle_{i',j'}$ becomes full by assigning $p \rightarrow \ell$. The open point q in $\triangle_{i',j'}$ now has to be assigned to ℓ' in order to find a valid labeling.

Proof. We assume that the necessary conditions from Lemma 2 hold, otherwise we report infeasibility after the $O(n^2)$-time preprocessing.

For the correctness of the described algorithm it is obviously sufficient to show that the procedure *complete*$(\triangle_{i,j})$, not called within the completion of a supertriangle, computes a valid length-minimum labeling within $\triangle_{i,j}$. We do this in the following stages: after *complete*$(\triangle_{i,j})$ has finished it holds that ...

1. ... each of the labels $\mathcal{L}_{i,j}$ is assigned to a distinct point in $\triangle_{i,j}$.
2. ... the computed labeling is valid. Any flip that is performed to purge crossings leaves the total leader length unchanged.
3. ... the computed labeling is length-minimum.

For space reasons we have to omit the details of 1.–3. and conclude the proof by showing that the algorithm requires quadratic time and space. Storing the cell structure dominates the space consumption and is quadratic. A call to *complete*$(\triangle_{i,j})$, where $\triangle_{i,j}$ has κ open points, requires at most $O(\kappa^2)$ time: after the lists of open points in $\triangle_{i,j}$ have been generated and sorted in $O(\kappa \log \kappa)$ time, finding the point for an open label and updating the list of remaining items is in $O(\kappa)$. For the crossing purges we have to deal with at most $O(\kappa^2)$ crossings in total. Since each point appears as an open point for exactly one full triangle this settles the total running time to $O(n^2)$. □

References

[1] K. Ali, K. Hartmann, and T. Strothotte. Label layout for interactive 3D illustrations. *J. of WSCG*, 13:1–8, 2005.

[2] M. A. Bekos, M. Kaufmann, A. Symvonis, and A. Wolff. Boundary labeling: Models and efficient algorithms for rectangular maps. *Computational Geometry: Theory & Applications*, 36:215–236, 2007.