FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

# Boundary Labeling for annotated documents

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

**Bachelor of Science**

in

**Bachelor's programme Software & Information Engineering**

by

**Jakob Klinger**
Registration Number 1125755

to the Faculty of Informatics
at the TU Wien

Advisor: Ass.Prof. Dipl.-Inform. Dr.rer.nat Martin Nöllenburg
Assistance: Univ.Ass. Fabian Klute, M.Sc., B.Sc.

Vienna, 25th September, 2017

Jakob Klinger    Martin Nöllenburg

# Erklärung zur Verfassung der Arbeit

Jakob Klinger
Scherzergasse 10/8, 1020 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 25. September 2017

Jakob Klinger

# Contents

CHAPTER 1

# Introduction

Whenever additional information neets to be inserted into an existing document without altering the original text, we can make use of annotations. They usually take the form of footnotes, which require only a minimal reference in the main text, and are used for a variety of reasons - for example, to provide additional information that would hinder the text's flow if inserted directly, or as a result of a commenting tool that is used for communication between an author and their editor. If a more obvious connection between the text and the referenced content is required, for example when lengthy comments are added, or if a change-tracking tool is used, the reference is often placed to the side of the document and visibly connected to the part of the text it is referring to. This style of annotation is easily implemented on virtual documents, since they can be hidden on demand, however if the annotations need to be included in a printed version, there are several issues that arise regarding readability of the final product and ambiguity of text-annotation assignments.

In this thesis, we will look at ways to use Boundary Labeling for this problem, which means that all annotations will be placed somewhere outside of the text they are referencing and will be visually connected to the feature they are referencing. (See also [3])

The guidelines on how to create suitable labelings are as follows: the connections should be as direct as possible, no important information should be obscured, and it should be easily discernable which Label belongs where. These three criteria easily come into conflict with one another, as the text usually is very dense and leaves little space for lines in between, yet one shouldn't allow them to pass through the text, as this makes the text harder to read.

## 1.1 Terminology and Fundamentals

Boundary Labeling (or equivalent concepts) can be applied to a space with different geometry or more dimensions, but this thesis will only concern itself with two-dimensonal,

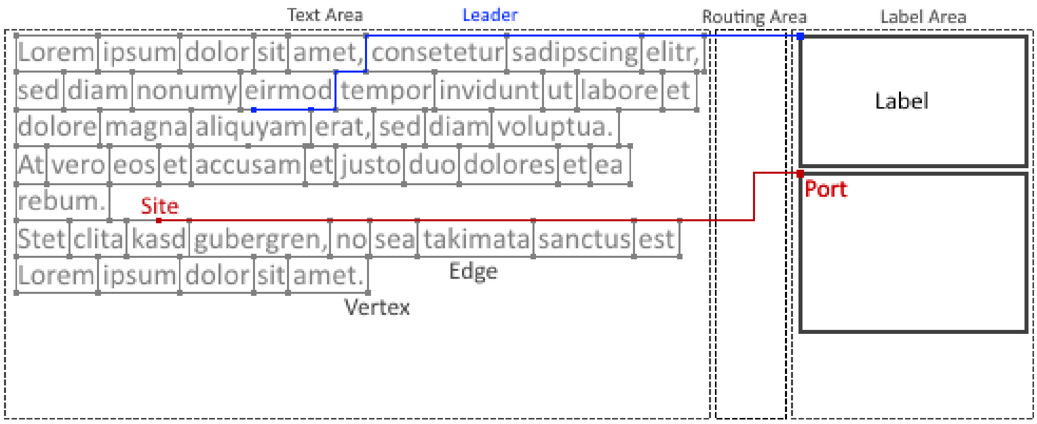



Figure 1.1: Illustrated guide to the labeling terminology

Euclidean space. To easily reference important concepts, some additional terminology will be introduced as well. (See Fig. 1.1 for a visual explanation)

A *graph* $G = \langle V, E \rangle$ is a tuple of *vertices* $V = \{v_1, v_2, ..., v_n\}$ and *edges* $E = \{e_1, e_2, ..., e_m\}$. A vertex $v$ is a featureless object. Each edge $e$ is a relation between two vertices $E \subseteq V \times V$. We call two vertices $u, v \in V$ *adjacent*, if the edge $e = (u, v) \in E$. A *path* $P = v_1, ..., v_h$ is an ordered sequence of vertices, where each vertex must have an edge connecting it to the subesquent one. *Depth-first search* is a searching algorithm on a graph G that starts at a given vertex $v \in V$ and explores the graph by traversing its edges as far as possible before backtracking, and continues to do so until a pre-defined goal is met.

A *Polyline* is a curve defined by a sequence of points $O = p_1, p_2, ..., p_n$, connecting each point to its successor with straight lines. Each point $p \in O$ cannot be equal to any other point in $O$. We will call a polyline *monotone*, if it satisfies certain conditions. For *monotonically increasing* polylines, each two vertices must satisfy the following: $\forall p_i, p_j \in O : i < j \iff (X(p_i) \leq X(p_j) \wedge Y(p_i) \leq Y(p_j) \wedge p_i \neq p_j)$, where $X(.)$ returns a point's X-coordinate, and $Y(.)$ returns its Y-coordinate. For *monotonically decreasing* polylines, the inverse must be true - each point must have at least one coordinate with a value smaller than its predecessor. *Labels* hold additional information and are represented as boxes containing this information. They always have a *port*, a special point on the label's border which will be defined later. Labels are usually placed in the *label area* which is a rectangular area designated to hold labels. It is located next to of the bigger, rectangular *text area*, which contains the document's text and all *sites*, the points or objects that a label's information refers to. If multiple label areas exist on different sides of the text area, we speak of *multi-sided labeling*, otherwise we speak of *one-sided labeling*. We will be using one-sided labeling in our implementation. Some space was left in between the text and label area, to make connecting sites to their labels easier, which we will call the *routing area*. The site and the label are connected via a *leader*, a polyline that can be further classified by looking at the orientation of its segments: *O-Segments* run orthogonally to the border of the label area. *P-Segments* run parallel to the border of the label area, and as such must be combined with other segments for the leader to reach its destination. *S-Segments* are not required to have any particular orientation, and simply connect their start and ending points in a straight line. The leader's name is created by combining the name of the segments - for example, the blue leader from Fig 1.1 would be classified as an OPOPO-Leader. The location where a leader connects to the label is called the *port*. It can be restricted to pre-defined positions.

## 1.2 Related Work

Boundary labeling was first introduced by Bekos et al. in 2004 (see [3]), where both one-sided and multi-sided labelings with different leader types are looked into. They also showed that the optimal placement of arbitrarily-sized labels on two sides of the text area can be NP-hard by drawing comparisons to the Partition-Problem. However, a pseudo-polynomial solution exists for this problem, which was adapted to this variation of the problem.

Since then, several papers have been written about boundary labeling. One of these is [2], which looks into the readability of different leader styles. Interestingly, some leader styles perform quite well, despite the study's participants preferring others over them, with OPO-Leaders being both least preferred and the hardest to follow.

Another article using boundary labeling is [4] by Göetzelmann et al., which creates boundary labeling-style annotations along other methods to label different parts of three-dimensional figures, resulting in pictures similar to what could be found in a textbook. As this algorithm works in real-time, it is suitable for labeling interactive models and allows for user interaction.

Boundary labeling in text documents however, is rarely discussed, and only few papers exist about this topic. The programs that employ this style of annotation often also use rather simple algorithms, to mediocre results or make extensive use of the interactivity of a digital medium, showing annotations only on demand. However, the few papers that approach this topic add interesting information to the discussion.

The paper about the Luatodonotes-Package[5] uses several styles of drawing leaders, and came to the conclusion that leaders without bends are easier to follow, which fits with [2]'s observations, which ranks OPO- and PO-Leaders lower than other variants. However, most solutions proposed in [5] do not consider whether a path overlaps with text or not, which results in a decrease in readability. While we do not use the routing and leader styles introduced in this paper, the results can be used in comparisons regarding readability of the main text and ease of use of the different leader styles.

The thesis by Loose[7] on the other hand is based around only using the free space between lines and words, which produces longer leaders, and forces curves, but doesn't obscure any part of the text. The two different approaches in this paper were a clustering-based algorithm, which was previously described in [8], and a flow network-based approach. Several concepts of this paper, such as the graph-based strategy and the usage of a routing area will be adopted in our thesis and it is by far the biggest influence on our approach to the problem.

Lin et al.[6] use only OPO-Leaders that have their P-Segment located outside of the text area in their paper, but allow the leaders to use the text area's border on the opposite side of the label area to route upwards or down. This allows for more labels to be placed as close as possible to their leader's source, at the cost of increasing select leaders' length and placing some labels out of order. While this is an interesting way to avoid longer leaders in general, it is hard to combine with the graph-based routing that happens inside the text area, so we won't make use of it.

CHAPTER 2

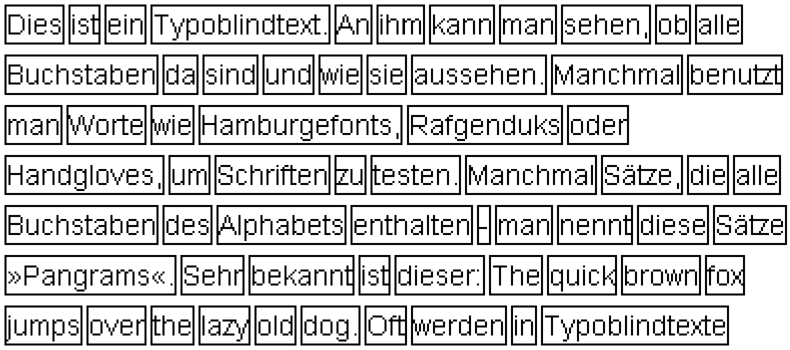
# The Algorithm

## 2.1 Problem Specification

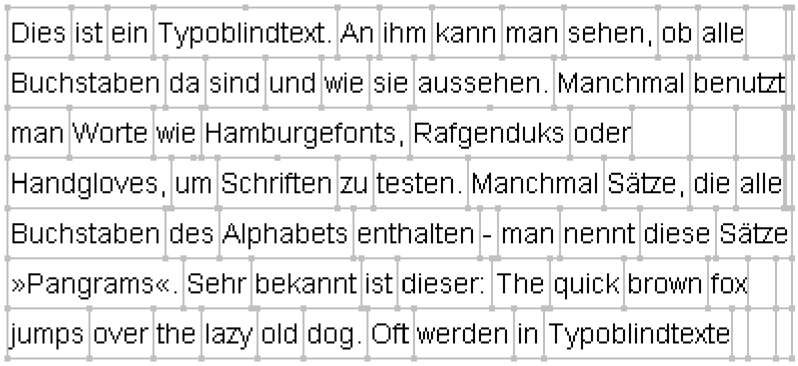
We limited the leaders to use exclusively O- and P-Segments, and banned them from passing through words. We also place labels as far up as possible to maximize the space remaining for remaining placements. Additionally, a leader isn't allowed to be any longer as is necessary to connect a given Site with its label's port.

These restrictions are implemented as follows: We divided the text $T$ into separate lines $L = \{W, H\}$ which are in turn made up of words $W$ and whitespace $H$. The remaining space between lines is called $S = \{s_1, ...s_{n+1}\}$, with $n$ being the number of lines in $L$. For each $l_i \in L$, $s_i$ is the space directly above that line. All $s \in S$ are the same height and width, even $s_1$ and $s_{n+1}$, which only have lines on one of their sides. For each word $w \in W$, we define $R(w)$ as its bounding rectangle, which marks the space leaders aren't allowed to cross. For a graphical representation see Fig. 2.1a.

Since we only allowed the Usage of O- and P-Segments in leaders, the only way for a leader to cross through a line of text is with a P-Segment through whitespace, whereas O-Segments are only usable between lines. Therefore, we can create a routing graph $G = \langle V, E \rangle$ whose Edges $E$ reflect the legal paths a leader can take within $T$. For each line $l_i \in L$ and each whitespace character $h \in l_i$, one vertex is placed in $s_i$ and $s_{i+1}$ each, located above or below the whitespace character, and with a maximum distance from each of its neighbouring lines. The start and end of each line are also assigned a pair of vertices each. Sites will be represented by the insertion of additional vertices in $S_i$ on a similar height as those next to whitespace of the same line and located directly above the center point of its words $w_j$ bounding rectangle $R(w_j)$. The edges between the vertices are all perfectly horizontal or vertical, and do neither intersect with any bounding rectangle, nor any vertices other than their starting and ending vertex. The resulting graph looks similar to Fig. 2.1b. In the following, we define vertex $v_a \in s_i$ to be *above* another vertex $v_b \in s_j$, if $i < j$. If $v_a$ and $v_b$ are adjacent to each other, $v_a$ is *directly above* $v_b$. We will also define a vertex $v_b$ to be *between* two vertices $v_a$




(a)




(b)

Figure 2.1: Visualization of the space reserved for the text, and the resulting graph.

and $v_c$, if $v_a, v_b, v_c \in s$ ($s \in S$) and it is impossible to create a path $P \subseteq s$ using only vertices from $s$ that leads from $v_a$ to $v_c$ without including $v_b$. Furthermore,we will call the vertices closest to the text area's border to the routing area *Border Vertices* $B \subset V$. They serve as a goal for the depth-first search, and are located at the end of each line in our implementation.

Each edge has a capacity of 1, meaning that no more than one leader is allowed to pass through it. Leaders also aren't allowed to cross one another, as it is hard to discern between intersections and two leaders curving away from each other. Furthermore, they are monotonically increasing, and go as far up as possible, limited only by the previous label's placement.

The Routing Area will be used to connect each site's path with its source, using OPO-Leaders. Combined with the path $P = \{s, v_1, v_2, ..., v_b\}; P \subset V$ leading from the source $s \in S$ to the graph's border ($v_b \in B$) it creates an unbroken connection between a source and its port, which shall be called a *Source-Port-Path*, or *SP-Path*.

## 2.2 Description

The routing algorithm works through the annotations $a \in A$ in the order they appear in the text, placing each as far up as possible, skipping any annotation that cannot be placed above its leader's source or is impossible to route. It will use fixed ports located in the top left corner of each label.

We use a depth-first-search algorithm with the Graph $G$ and the set of sites $V_{ann} \subset V$ as input. The Algorithm prioritizes routing to not yet visited vertices located above the current vertex and terminates either when reaching the right text border or if all possible paths failed to reach the text border, and the resulting SP-path will be monotonically increasing. If the routing for any site $v_{ann} \in V_{ann}$ failed, the algorithm returns $\perp$ for that site. To ensure that routings remain crossing-free, leaders are forbidden to incorporate any vertices that are either located directly above any vertices of the last successfully routed leader, or not between that leader's site and the border vertices. This path is split into two parts: The path within the graph ($P \subset V$), leading from the source to the text area's border, and the OPO-Leader that connects the text area's border with the Label's port. The former is implemented as a list of vertices $V_{Path} \subset V$ the leader travels through, whereas the latter is determined by the rightmost vertex of $V_{Path}$, the Port, and the location of the P-Segment. (For an illustration in Pseudocode see Alg. 1.)

Each SP-path's P-Segment in the Routing Area is only placed after all SP-Paths have been generated, and will be placed with equal spacing between the borders of the Routing Area and each P-segment, to ensure readability of the resulting leaders.

## 2.3 Analysis and Proof

In this section we will analyze the following:

- Computation time of graph-creation
- Computation time of the routing algorithm for single leaders
- Overall computation time of the routing algorithm

Furthermore, we will prove that:

- The algorithm generates only crossing-free routings
- The algorithm always returns a highest (define!) legal path

  NOTE: The algorithm keeps the annnotations in reading order

To determine the program's computation time, we will first take a look at the creation of the graph. As discussed in section 2.1, we will be placing a pair of vertices per whitespace character, as well as one additional pair at both the start and end of each line, which amounts to $2 \times |H| + 4 \times |L|$ vertices total. Together with the sites $I$,

```
Data: A single annotation's source and its Graph
Result: A List of vertices describing the leader's path
initialization
while currentVertex not at right text border do
    if (Graph.getTopNeighbourOf(currentVertex) ≠ null) ∧ ¬backtracking and
     previous Vertex not part of other leaders then
        Path.addVertex(currentVertex)
        currentVertex ← Graph.getTopNeighbourOf(currentVertex)

    else if Graph.getRightNeighbourOf(currentVertex) ≠ null then
        Path.addVertex(currentVertex)
        currentVertex ← Graph.getTopNeighbourOf(currentVertex)
        backtracking ← False
    else
        backtracking ← True
        repeat
            oldVertex ← currentVertex
            currentVertex ← Path.getLastEntry()
            Path.RemoveVertex(currentVertex)
        until currentVertex's Position is below oldVertex or Path is Empty

        if currentVertex not below oldVertex then //No path found
            break
        end
    end
end
```

**Algorithm 1:** The Depth-First-Search algorithm used in the program.

which are also represented as vertices, this results in the total number of vertices $|V| = |I| + 2 \times |H| + 4 \times |L|$. Additionally, each vertex in a pair is connected to its counterpart, which creates $(|V| - |I|)/2$ edges, representing all possible P-Segments. Finally, each edge placed in a space $s \in S$ above or below a line will be connected to each other, which is solved by connecting each vertex to the next unconnected vertex remaining in $s$. This leads to the creation of $|s| - 1$ edges representing possible P-segments, which leads to a total of $|V| - |S|$ additional edges, making the total number of edges $|E| = (3 \times |V| - 2 \times |S| - |I|)/2$.If we assume that the creation of edges and vertices both take a similar amount of time, we are looking at roughly $5/2 \times |V|$ operations, which means that the algorithm scales linearly with the amount of vertices placed, which in turn is directly proportional to the number of words $|W|$ in the text. For a representation of the graph-generation algorithm in pseudocode, see Alg. 2.

Next, we shall look at the routing algorithm - since we use depth-first search, which is a well-known algorithm, the worst-case computation time is known to be $|V| + |E|$. To reach this time, we'd have to try and use every single edge in the graph, reaching each of the graph's vertices in the process. However, as our algorithm can't use any edges leading away from its target, our search space is reduced to only the vertices either in lower-numbered $s \in S$ (those "above" the current position) or those in the same $s$, but located between the source and the border vertices. If we also take into account that leaders aren't allowed to cross one another, this further limits our search space, removing all vertices above the previous leader's vertices.

To prove that the algorithm only creates crossing-free routings, we will examine the rules given in section 2.2: All nodes that are either located directly above any nodes of the last successfully routed leader (whose path is represented by $P_{old}$) or not between the site $s_{old} \in P_{old}$ and the border vertices. For a crossing to occur, both the currently routed path ($P_{new}$) and $P_{old}$ must have one or several consecutive vertices in common, before which the nodes of one were below the other, and afterwardsthe opposite is true. This leaves us with two options:

- $P_{new}$ crosses from below to above $P_{old}$: For this to happen, $P_{new}$ must at some point start to include vertices that were also used in $P_{old}$. While this is allowed, the routing algorithm was explicitely forbidden to use any vertices located directly above any vertices of $P_{old}$, which makes crossings impossible.

- $P_{new}$ crosses from above to below $P_{old}$: While this type of crossing would be possible if $P_{new}$ managed to incorporate vertices that are above and not between $P_{old}$ and the border vertices, this requires the inclusion of nodes that are not between $s_{old}$ and the border vertices, as the currently routed leader's site $s_{new}$ is either located between $s_{old}$ and the border vertices, which wouldn't make this routing possible without violating the monotonicity constraint, or below $s_{old}$. Since the inclusion of these nodes was also explicitely forbidden, this type of crossing is impossible as well.

Therefore, the algorithm only produces crossing-free routings.

Finally, we want to prove that the algorithm only returns a highest legal path - a path that is both monotonically increasing and crossing-free, ends at the highest possible border vertex $b_{high} \in B$ reachable from a given source $s$, and contains no other border vertices than $b_{high}$. The path must also contain exactly one border vertex. The first two criteria are given, as the algorithm only returns monotonically increasing paths, and the existence of crossing was disproven above. To show that there are no reachable border vertices above $b_{high}$, we can assume there exists a vertex $b_{top} \in B$ that is located above $b_{high}$ and legally reachable via the path $P_{top}$. As the algorithm prioritizes routing to vertices directly above, the only way for the path to $b_{high}$ to reach higher vertices than $P_{top}$ would be that either $P_{top}$ would generate crossings with previously routed leaders or violate the the monotonicity constraint, which both render $P_{top}$ illegal. Since we also cannot include more than one border vertex in our path, we must conclude that $b_{top}$ cannot exist.

```
Data: A text with annotations,stored as a String-Array
Result: A Graph (as described above)
initialization
foreach w in words do
    if w is annotation then
        v ←new Vertex(previousWord.getCenter())
        v.setAnnotation(w)
        Graph.addVertex(v)
        UpperVerticesList.addVertex(v)
    else
        if w is too big for the line then
            //Create last pair of vertices in current line
            v1 ←new Vertex(previousWord.getTopRight())
            v2 ←new Vertex(previousWord.getBottomRight())
            Graph.addAll(v1,v2)
            UpperVerticesList.addVertex(v1)
            LowerVerticesList.addVertex(v2)
            Graph.createEdgeBetween(v1,v2)

            //Start new line
            startNewLine()
            connectBasedOnPosition(UpperVerticesList)
            UpperVerticesList ←LowerVerticesList
            emptyList(LowerVerticesList)
        end
        v1 ←new Vertex(w.getTopLeft())
        v2 ←new Vertex(w.getBottomLeft())

        Graph.addAll(v1,v2)
        UpperVerticesList.addVertex(v1)
        LowerVerticesList.addVertex(v2)
        Graph.createEdgeBetween(v1,v2)
    end
end
```

**Algorithm 2:** Representation of the Graph-creation algorithm in pseudocode

CHAPTER 3

# Implementation

The program was written in Java 1.8.0u40, using JGraphT1.0.1[1] as graph library. Since we only want to create leaders that don't intersect with the text, the graph was created alongside the placement of the words on the canvas.

## 3.1 Challenges

CHAPTER 4

# Evaluation and Testing

## 4.1 Data generation

## 4.2 Testing methods

## 4.3 Results

CHAPTER 5

# Conclusion

## 5.1 Further notes

# Bibliography


[1] Jgrapht - a free java graph library.

[2] Lukas Barth, Andreas Gemsa, Benjamin Niedermann, and Martin Nöllenburg. *On the Readability of Boundary Labeling*, pages 515–527. Springer International Publishing, Cham, 2015.

[3] Michael A. Bekos, Michael Kaufmann, Antonios Symvonis, and Alexander Wolff. *Boundary Labeling: Models and Efficient Algorithms for Rectangular Maps*, pages 49–59. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[4] Timo Götzelmann, Knut Hartmann, and Thomas Strothotte. *Agent-Based Annotation of Interactive 3D Visualizations*, pages 24–35. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[5] Philipp Kindermann, Fabian Lipp, and Alexander Wolff. *Luatodonotes: Boundary Labeling for Annotations in Texts*, pages 76–88. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[6] C. C. Lin, H. Y. Wu, and H. C. Yen. Boundary labeling in text annotation. In *2009 13th International Conference Information Visualisation*, pages 110–115, July 2009.

[7] Amrei Loose. Annotation von texten unter berücksichtigung von textzwischenräumen. Master's thesis, Karlsruhe Institute of Technology, 2015.

[8] Martin Nöllenburg, Valentin Polishchuk, and Mikko Sysikaski. Dynamic one-sided boundary labeling. In *Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems*, GIS '10, pages 310–319, New York, NY, USA, 2010. ACM.