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
TODO: ADDRESS!

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

Annotating a document is usually solved by adding footnotes in an appropriate position and adding a simple reference in the text, leaving the reader to find the referenced content by themselves. If a more obvious connection between the text and the referenced content is required, the reference placed to the side of the document and visibly connected to the text by drawing a straight line or a more complex path between them.

In this paper, we will look at ways to use Boundary Labeling, 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 [2])

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.

While there are many papers discussing Boundary Labeling in general, only very few exist that apply this concept to written text. Generally, this approach isn't used very often, and tends to use simplistic algorithms which produce mediocre results. However, the papers that do discuss boundary labeling in text offer interesting contributions.

In the paper about the Luatodonotes-Package[3] illustrates some of the different styles of drawing these connecting paths, and came to the conclusion that paths without bends are easier to follow. However, most solutions proposed in that paper do not consider whether a path overlaps with text or not, which results in a decrease in readability.

The paper by Loose[4] on the other hand is based around only using the free space between lines and words, which produces longer paths, and forces curves, but doesn't obscure any part of the text.

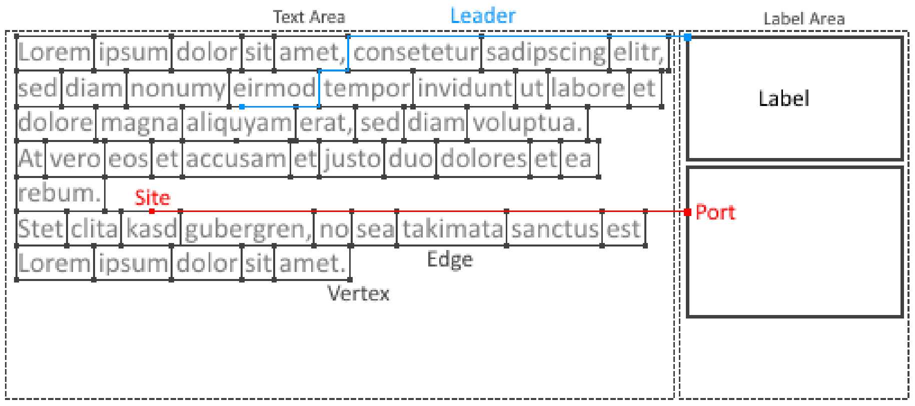

Figure 1.1: Illustrated guide to the labeling terminology

1.1 Terminology and Fundamentals

While Boundary Labeling (or an equivalent concept) can be applied to a space with different geometry or more dimensions, this paper will only concern itself with two-dimensonal, 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.

Polylines are a connected series of *line segments*. Line segments are straight lines that contain each point between their starting and end point. *Labels* hold additional information and are represented as boxes containing this information. They are usually placed in the *Label area* which is a rectangular area designated to hold labels. It is located next to of the bigger *text area*, which contains the document's text and all *sites*, the points or objects that a label's information refers to. 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 einding 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.

CHAPTER 2

The Algorithm

In our algorithm, the focus was put on keeping the text as readable as possible. This means that leaders aren't allowed to pass through words, which was implemented by enclosing each word in a bounding rectangle that leaders aren't allowed to pass through. The leaders should also be kept as short as possible, so we restricted them to always be moving toward the label, even if the direct route is unavailable.We also wanted to use the space available in the labeling area as efficiently as possible, so labels are placed as far up as possible in order to maximize the space available to future labels at the cost of increased leader length.

To create leaders that exclusively use the space not taken up by a word's bounding rectangle, we decided to use a graph similar to the one Loose [4] used. As each vertex represents a physical location, they will have co-ordinates associated with them, and the vertices representing the sites will hold additional information regarding its leader and label. The graph is constructed by placing vertices between the lines, located next to each corner of a word's bounding rectangle, with two consecutive words in a line sharing the two vertices associated with their adjacent corners. For the sites, we inserted an additional vertex above the center of the word, which will serve as the leader's starting point. After placing all vertices, we created edges between each vertex and the closest horizontal neighbour to both sides, and between nodes that are located exactly above or below each other, and exactly one line apart. (For a representation in Pseudocode, see Alg. 1)

In our labeling algorithm, we decided to work through the labels in the order they appear in the text, placing each as far up as possible, and skipping any label that would've required us place the label below its leader's source node. We also opted to use fixed ports located in the top left corner of each label, as this allowed us to unabiguously place annotations by only knowing their port location, which is equal to their leader's ending point. To not restrict the label's placement by the line spacing, we left a buffer zone between the text area and the label area, which allows us to place labels even further above to optimally use the available space. The algorithm's input consists of the Graph

```
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
            startNewLine()
            connectBasedOnPosition(UpperVerticesList)
            UpperVerticesList ←LowerVerticesList
            emptyList(LowerVerticesList)
        end
        v1 ←new Vertex(w.getTopLeft())
        v2 ←new Vertex(w.getTopRight())
        v3 ←new Vertex(w.getBottomLeft())
        v4 ←new Vertex(w.getBottomRight())

        Graph.addAll(v1,v2,v3,v4)
        UpperVerticesList.addAll(v1,v2)
        LowerVerticesList.addAll(v3,v4)
        Graph.createEdgeBetween(v1,v3)
        Graph.createEdgeBetween(v2,v4)
    end
end
```

Algorithm 1: Representation of the Graph-creation algorithm in pseudocode

G and a set of sites $V_{ann} \subset V$ which will be routed using a depth-first search algorithm, restricted to only selecting vertices located above or to the right of the current vertex, prioritizing moving up whenever possible. After the algorithm terminates, it returns a set with routing information for each vertex, which contains a Path $P = \{v_1, \cdots, v_n\}$, leading from the site to the text area's border, along with some additional information on how to draw the OPO-Segment that connects to the Label's port. If the routing for any given site failed, the path consists of a single vertex - the site. (For an illustration in Pseudocode see Alg. 2.)

```
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) $\neq$ null) $\wedge$ $\neg$backtracking
     then
        Path.addVertex(currentVertex);
        currentVertex $\leftarrow$ Graph.getTopNeighbourOf(currentVertex);
        ;
    else if Graph.getRightNeighbourOf(currentVertex) $\neq$ null then
        Path.addVertex(currentVertex);
        currentVertex $\leftarrow$ Graph.getTopNeighbourOf(currentVertex);
        backtracking $\leftarrow$ False;
    else
        backtracking $\leftarrow$ True;
        repeat
            oldVertex $\leftarrow$ currentVertex;
            currentVertex $\leftarrow$ 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 2: The Depth-First-Search algorithm used in the program.

2.1 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.

Bibliography

[1] Jgrapht - a free java graph library.

[2] Michael A. Bekos, Michael Kaufmann, Antonios Symvonis, and Alexander Wolff. Boundary labeling: Models and efficient algorithms for rectangular maps. *Computational Geometry*, 36(3):215 – 236, 2007.

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

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