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 needs 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 [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.

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-dimensional,

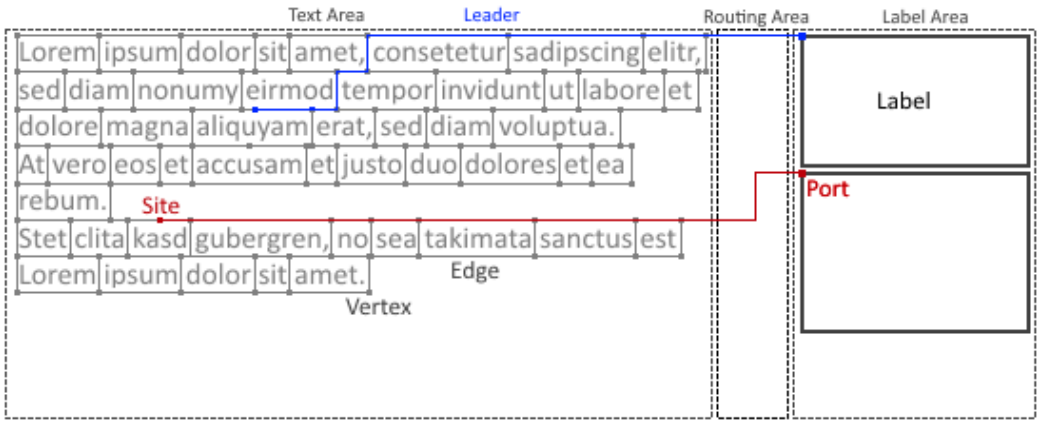


Figure 1.1: Illustrated guide to the labeling terminology

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 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 O *monotonically increasing*, if any two vertices $p_i, p_j \in O$ 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.

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 [2]), 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 [1], 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 [3] 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[4] uses several styles of drawing leaders, and came to the conclusion that leaders without bends are easier to follow, which fits with [1]'s observations, which ranks OPO- and PO-Leaders lower than other variants. However, most solutions proposed in [4] 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[6] 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.[5] 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 $T = \{W, H\}$ into separate lines $L = \{l_1, l_2, ..., l_n\}$ which are in turn made up of words $w \in W$ and whitespace $h \in H$, which alternate, starting and ending with a word, which creates a sequence $l = (w_1, h_1, w_2, ..., h_{m-1}, w_m)$, where m is the number of words in l. The width of each line is equal to the text area's total width in characters ($width$), which means that $m \leq \frac{width}{2}$. 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. The $s \in S$ and $l \in L$ are arranged in alternating order in the text $T = \{s_1, l_1, s_2, l_2, ..., s_n, l_n, s_{n+1}\}$. 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

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

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 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 $P = \{s, v_1, v_2, ..., v_h, v_b\}; P \subset V; P \cap B = v_b$

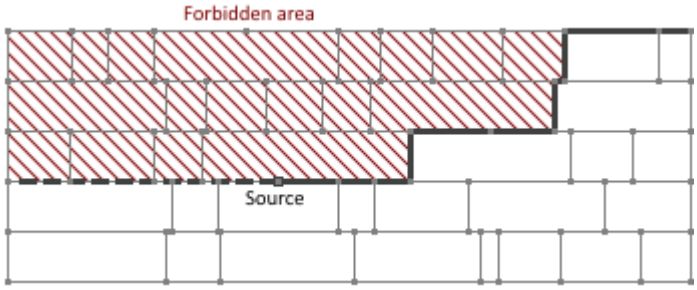


Figure 2.2: Illustration of the search-space limitation by previous leaders

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, the most recently successfully routed leader P_{last} is used as a ceiling to the available search space. This means that any vertex above the polyline created by the vertices in P_{last} and the horizontal connection of P_{last}'s source with the text area's border opposite of the label area is forbidden to use. (See also Fig. 2.2) 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.

```
   Data: A single annotation's source and its Graph
   Result: A List of vertices describing the leader's path
 1 initialization
 2 while currentVertex not at right text border do
 3 |   if (Graph.getTopNeighbourOf(currentVertex) ≠ null) ∧ ¬backtracking and
   |    previous Vertex not part of other leaders then
 4 |   |   Path.addVertex(currentVertex)
 5 |   |   currentVertex ← Graph.getTopNeighbourOf(currentVertex)
 6 |   |
 7 |   else if Graph.getRightNeighbourOf(currentVertex) ≠ null then
 8 |   |   Path.addVertex(currentVertex)
 9 |   |   currentVertex ← Graph.getTopNeighbourOf(currentVertex)
10 |   |   backtracking ← False
11 |   else
12 |   |   backtracking ← True
13 |   |   repeat
14 |   |   |   oldVertex ← currentVertex
15 |   |   |   currentVertex ← Path.getLastEntry()
16 |   |   |   Path.RemoveVertex(currentVertex)
17 |   |   until currentVertex's Position is below oldVertex or Path is Empty
18 |   |
19 |   |   if currentVertex not below oldVertex then //No path found
20 |   |   |   break
21 |   |   end
22 |   end
23 end
```

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

2.3 Analysis and Proof

In this section we will first analyze the computation time required to create the graph. Afterwards, we will prove the correctness of our routing algorithm, and finally we will determine the algorithm's runtime complexity.

Lemma 1. *The creation of the routing graph is in O(n).*

Proof. As discussed in Section 2.1, each line holds at most $\frac{width}{2}$ words, with an appropriate amount of whitespace in between. Since *width* is a constant value, the amount of vertices placed is only dependent on the number of lines in the text, therefore $|V| = O(n)$. Similarly, each vertex has at most 4 incident edges, representing the two possible placements of P- or O-Segments starting from that vertex, respectively. Therefore, $|E| = O(n)$. As both the creation of new vertices and edges can be done in a constant amount of time, the overall computation time is in $O(n)$. □

For a representation of the graph-generation algorithm in pseudocode, see Alg. 2.

```
   Data: A text with annotations,stored as a String-Array
   Result: A Graph (as described above)
 1 initialization
 2 foreach w in words do
 3     if w is annotation then
 4         v ←new Vertex(previousWord.getCenter())
 5         v.setAnnotation(w)
 6         Graph.addVertex(v)
 7         UpperVerticesList.addVertex(v)
 8     else
 9         if w is too big for the line then
10             //Create last pair of vertices in current line
11             v1 ←new Vertex(previousWord.getTopRight())
12             v2 ←new Vertex(previousWord.getBottomRight())
13             Graph.addAll(v1,v2)
14             UpperVerticesList.addVertex(v1)
15             LowerVerticesList.addVertex(v2)
16             Graph.createEdgeBetween(v1,v2)
17
18             //Start new line
19             startNewLine()
20             connectBasedOnPosition(UpperVerticesList)
21             UpperVerticesList ←LowerVerticesList
22             emptyList(LowerVerticesList)
23         end
24         v1 ←new Vertex(w.getTopLeft())
25         v2 ←new Vertex(w.getBottomLeft())
26
27         Graph.addAll(v1,v2)
28         UpperVerticesList.addVertex(v1)
29         LowerVerticesList.addVertex(v2)
30         Graph.createEdgeBetween(v1,v2)
31     end
32 end
```

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

Lemma 2. *The algorithm only creates crossing-free routings.*

Proof. As stated in Section 2.2, we are forbidden to use any vertices above the polyline created by the last successfully routed leader (P_{last}) and the direct connection of P_{last}'s source and the left text area's border. As this makes it impossible to construct a route

that incorporates any vertex above or to the left of P_{last}, those two leaders cannot possibly cross each other. This is true for any two consecutive leaders, therefore each leader also cannot cross any of the leaders routed before P_{last}, as those are located on the other side of P_{last}, which contains only forbidden vertices. Therefore, our algorithm produces crossing-free routings. □

For the next part, we first need to define what we consider a *highest legal path*: A *legal path* is one that satisfies both the constraints of monotonicity and being crossing-free, whereas a *highest path* reaches the highest possible border vertex reachable from a given source.

Lemma 3. *Our algorithm only returns a highest legal path.*

Proof. The legality of any generated path is given, due to the algorithm only returning monotonically increasing paths, and Lemma 2. To show that there are no reachable border vertices above the path's highest reachable border vertex b_{high}, we will 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_{top} to reach higher vertices than b_{high} 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 conclude that such a path cannot exist. □

Theorem 1. *The routing problem can be solved with a worst-case time complexity of $O(n^2)$ for all sources.*

Proof. As part of the proof for Lemma 1, we have shown that the total number of vertices $|V|$ is in $O(n)$. Since $S \subset V$, $|S|$ must too be in $O(n)$. We have proven our algorithm's solutions to be correct in the Lemmas 2 and 3. The performance of our routing algorithm is easily calculated, as depth-first-search has a worst-case-performance of $|V|+|E|$, which is in $O(n)$ for our problem, as $|E|$ is in $O(n)$ as well. (See Lemma 1) Therefore, our overall computation time comes out to $|S| \times O(n)$, or $O(n^2)$. □

CHAPTER 3

Implementation

The program was written in Java 1.8.0u40, using JGraphT 1.0.1[1] as graph library. No other external libraries were used. The program itself can be found at GitHub [2].

3.1 Architecture

The program is divided into three parts. The GUI and program window only propagate user interaction to other parts of the program. It will be discussed further in the next section. The main drawing area handles the visualization of the results, calculates where to place the text on-screen, and also generates the routing graph, as this can be done alongside the text placement. The routing is done in a separate class, and can easily be exchanged with another algorithm because of this.

3.2 GUI

The GUI of the program is kept simple, as the main goal was to demonstrate the routing algorithm. All customization options are located on the control panel above the drawing area, which contains the text, leaders and annotations. (See also Fig. 3.1)

The control panel's first three options determine the font the program uses for the main text, whether it is bold or italicized and its size. The annotations' text currently uses a smaller version of this font. The fourth and last option allows the selection of different routing algorithms, although the current options are mostly variations on the routing algorithm described in Section 2.2. After changing any value, the program will automatically be notified of the changes and react accordingly.

[1] http://jgrapht.org/
[2] https://github.com/e1125755/AnnotationRouting

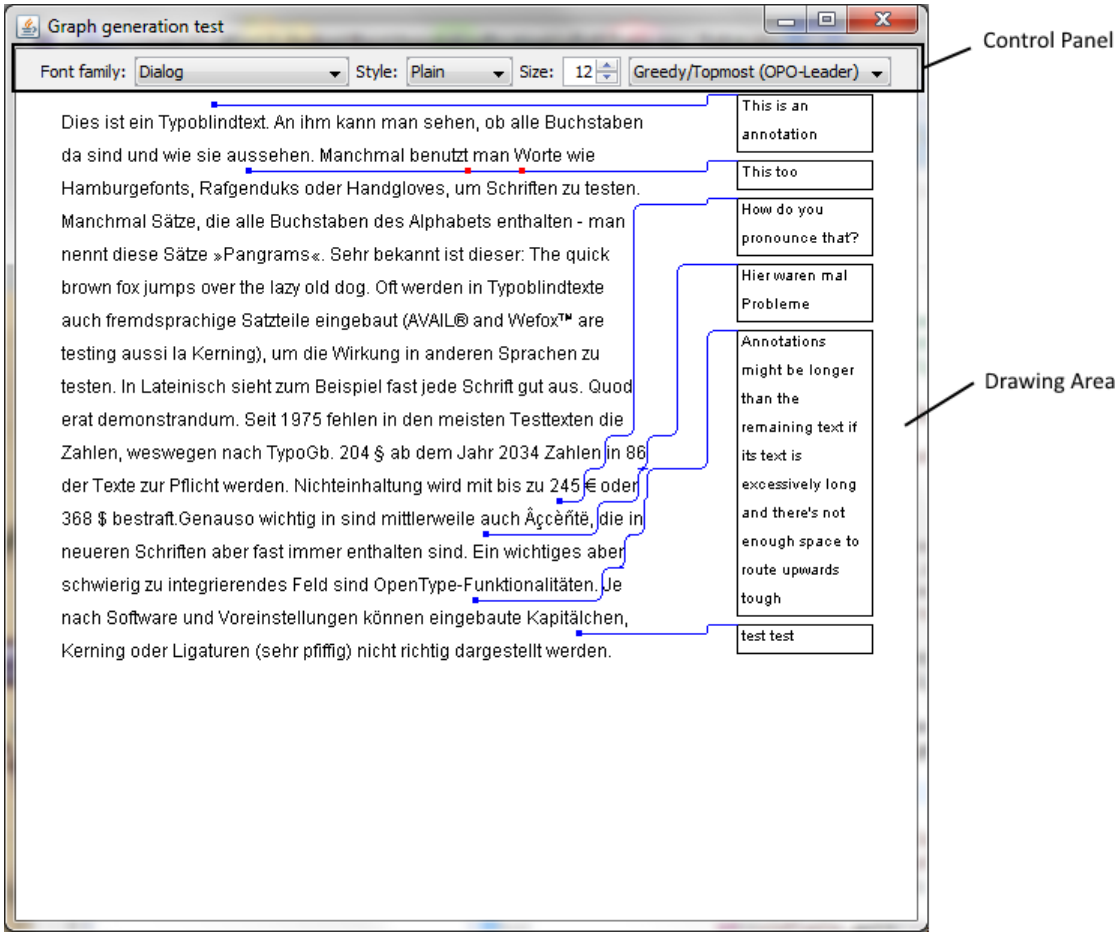


Figure 3.1: Picture of the routing program

3.3 Drawing Area

As this program is a proof of concept, the drawing area itself can't be interacted with, it only serves to display the algorithm's results. If we were to adapt this program for regular use, the user should be able to manually change routings or move labels around here. Due to reasons described below in Section 3.5, each word has to be placed individually, which allows us to create the routing graph alongside the darwing process. After the text has been drawn completely, the routing algorithm is invoked from here, and its results are visualized.

3.4 Routing Algorithm

The Routing algorithm has already been described thoroughly in Section 2.2, that process is implemented by creating ordered lists of the vertices traversed by the SP-Path,

along with some additional information on where to draw the P-segment located in the Routing Area. While the algorithm offers methods to route each annotation individually, we currently route all annotations at the same time, which vastly decreases the required communication between the algorithm and the rest of the program.

3.5 Challenges

Displaying the text on-screen without losing the position of the whitespace characters turned out to be harder than expected. While Java offers a class that divides any string of characters into separate lines, the resulting line object does not allow for it's text to be extracted, which makes it unusable for our purposes. This problem was solved by placing each word on the screen individually. For simplicity's sake, we only allowed line breaks between words, whenever a whitespace character was encountered.

Another issue was the visualization of the routing graph and the leaders. While JGraphT offers ways to display graphs, it wasn't possible to influence the placement of individual vertices. Our solution was forgoing these options and drawing the graph unassisted, assigning coordinates to each vertex on creation. Vertices are represented by small rectangles, whereas edges are represented with straight lines. To avoid confusion between crossing leaders and leaders that incorporate the same vertex in their path, but curve away from each other, all leaders are drawn with rounded corners, which can be seen in Fig. 3.1.

3.6 Workflow

When the program starts, it first scans the text word by word in reading order, placing each word $w \in W$ on-screen, and saving the content of each encountered annotation for later. As the words are placed, vertices are added to the graph after each word, and at the start and end of each line. Source vertices are inserted whenever an annotation is encountered, and contain a reference to their annotation in addition to the information contained in regular vertices.

After the text has been processed, additional vertices are inserted into each line between the last word of that line and the right text border, located directly above or below vertices in the neighboring lines. Whenever a vertex is inserted exactly above or below another vertex in a neighboring line, they are immediately connected via an edge, whereas all vertices in the same space between lines $s \in S$ are connected in order of their x-coordinates after all vertices have been placed.

Once the graph has been generated, a list of all source vertices in reading order is passed to the selected routing algorithm, which then determines leader routing and annotation placement for each of them and returns the results to the main program, which then draws the results. (For details on the routing algorithm, see section 2.2.)

If any of the parameters on the control panel (see section 3.2) are changed, the process re-starts from the beginning.

CHAPTER 4

Evaluation and Testing

4.1 Data generation

To generate data, we created our own texts using word length statistics from Table III in "Length-frequency statistics for written English" by Miller et al. [7], which lists word frequencies by length. We opted to use the data from the "Content Words" column, as the combined statistics almost exclusively produced words with three or less letters in our tests. Using this data, we compiled our own fixed-length testing texts by creating words containing randomized alphabetic characters, taking care that the word length distribution matched our data. A fixed number of annotation sites is inserted into the text either by distributing their locations uniformly across the text, or by clustering them regionally.

For the clustering approach, we divided the text vertically and horizontally into three subsections each - top/center/bottom and left/center/right respectively. To determine the site's location, we then determined the line the site will be in and its exact word by generating numbers from two normal distributions, each with a standard deviation of 1/6th of the text's width or height, and a mean located at 25%, 50% or 75% of that parameter, depending on the chosen region.

The uniform distribution is achieved by choosing each annotated word in advance, regardless of its relative location within the text.

To enable the inspection of individual texts, each text was generated from a randomized seed that is saved at time of creation. To enable the repetition of full test batches, each of them was also generated using a seed, which is recorded as well.

4.2 Experimental Setup

All tests were run under 64-bit Windows 7 Professional Version 6.1.7601 Service Pack 1 Build 7601 on a consumer-grade laptop using Java 1.8.0u40. The laptop is a Dell

Latitude E6400 with an Intel Core2 Duo dual-core processor running at 2.53 GHz and 4 GB DDR2 RAM. There's 4047 MB virtual memory available on the SSD as well.

To generate many datasets at once, each testing run consists of 100 randomly generated texts that are routed one after the other without drawing them on-screen. During this process, we measure the time taken to complete each text, as well as the amount of successful routings, the total space used by all labels and the amount of P-Segments within the text area. The results are saved to a CSV-style text file, and a R script is generated to visualize the data. Additional information regarding test parameters are logged to a separate file. Each test run can be repeated, its seed is part of the testing parameters.

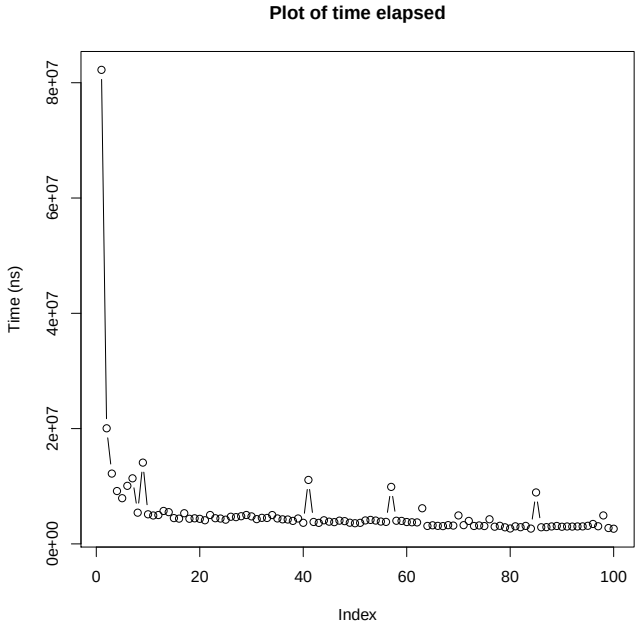
4.3 Results

Across every test run, the first two instances always took significantly longer to complete than subsequent ones. As this was replicated even when using the same text for all 100 passes, we believe this to be an issue caused by the implementation rather than the algorithm itself. (This effect can be seen in Fig. 4.1) Similarly, the biggest fluctuations in computation time for the remaining instances were caused by the computer's scheduling, as Fig. 4.1a and Fig. 4.1b both depict time statistics for the same run, executed twice in a row.

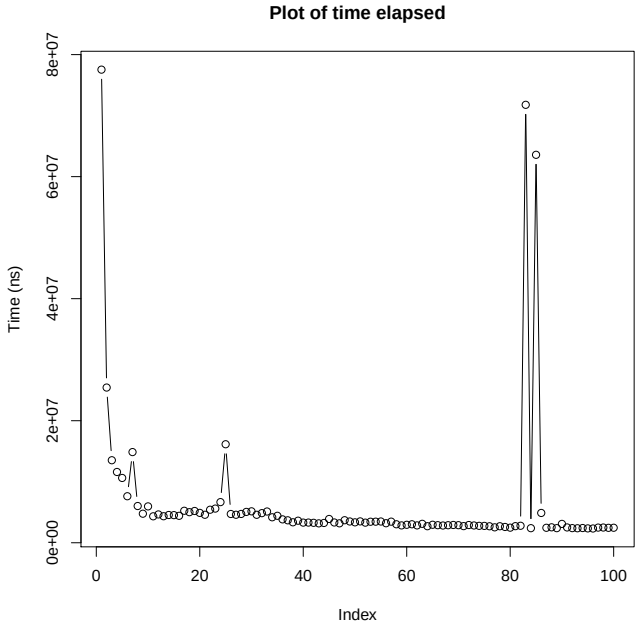
Adding more sources to the text increases the both the space usage of the labeling area and the number of successfully routes, but there seems to be a limit, as increasing the amount of sources only brings marginal improvements after a certain point. For our experiments, this was reached at 15 sources in a 300-word text. Doubling their amount improved both the space usage and the amount of routed leaders for several test instances, but the overall highest amount of successful routings increased only by 1, whereas the increase from 10 to 15 sources impacted the results much stronger. (See Fig. 4.2 for details.)

While the placement of sources definitely had an impact on the amount of successful routings, the influence of horizontal placement is far smaller than that of the vertical placement. This is likely due to the annotations being placed vertically, which means that the more annotations are placed, the less likely it becomes for sources close to the beginning of the text to be routed. Together with the Routing Area, which allows for labels to be placed further up, this means that placements in the first few lines are disadvantaged compared to those further down.

The effects of clustering are currently minor, and mostly outweighed by the influence of vertical placement, as the average word length is short enough that there is enough whitespace in each line to avoid leaders from blocking each other. Increasing the word length would make this effect more severe, and it would be interesting to see the influence on the results.

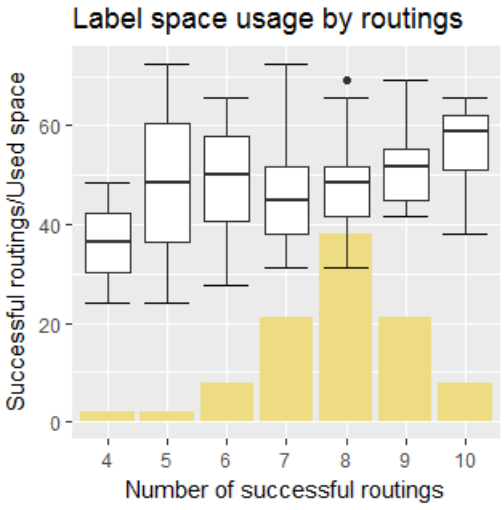


(a)

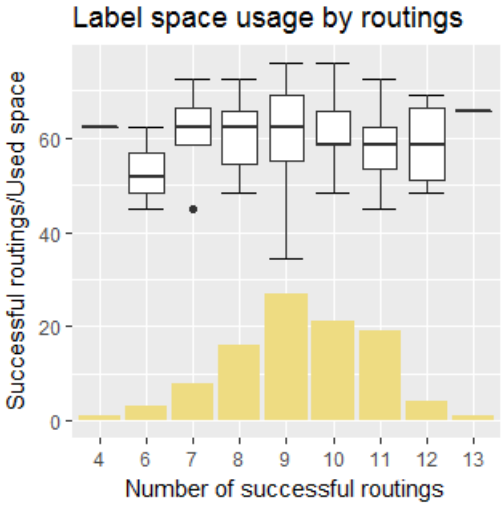


(b)

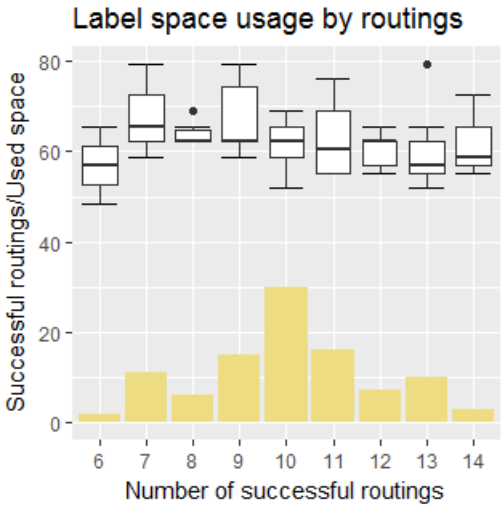
Figure 4.1: Two plots of the same test seed's time taken per text. Note the massive differences caused by the computer's scheduling.



(a)



(b)



(c)

Figure 4.2: Three plots of test batches with identical seeds. The only difference is the number of sources in the generated texts: 10, 15 and 30 respectively.

CHAPTER 5

Conclusion

5.1 Further notes

Bibliography

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

[2] 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.

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

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

[5] 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.

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

[7] G.A. Miller, E.B. Newman, and E.A. Friedman. Length-frequency statistics for written english. *Information and Control*, 1(4):370 – 389, 1958.

[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.