

Undergraduate Thesis

pdf2gtfs: Timetable Extraction from PDF Files

Julius Heinzinger

Examiner: Prof. Dr. Hannah Bast

Advisers: Patrick Brosi

University of Freiburg

Faculty of Engineering

Department of Computer Science

Chair of Algorithms and Data Structures

July 14th, 2023

Writing Period

14. 04. 2023 – 14. 07. 2023

Examiner

Prof. Dr. Hannah Bast

Advisers

Patrick Brosi

Declaration

I hereby declare that I am the sole author and composer of my thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work.

I hereby also declare that my Thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof.

Place, Date

Signature

Abstract

We present *pdf2gtfs*, which enables us to extract timetable data from PDF schedules and export it in the machine-readable GTFS format. This format requires the coordinates of the location of each stop. We use OpenStreetMap to search for the location of a stop, using only the stops name and the other stops of a route. We also introduce a new table extraction algorithm to *pdf2gtfs* that is row- and column-independent, meaning that we can process timetables regardless of their orientation. As further improvement to *pdf2gtfs*, we include additional information from OpenStreetMap about stops, such as the internationally unique IFOPT, or whether a stop is wheelchair-accessible. Finally, we create *p2g-eval* to evaluate the GTFS feed generated by *pdf2gtfs* based on a ground truth. This is to quantify the quality of the extracted information, and to help discover those parts that can still be improved. We show that the new table extraction algorithm achieves more accurate results, when compared to the previously used algorithm, and another, state-of-the-art table extraction tool. We also find that the detected locations are, in general, close to the true locations.

Zusammenfassung

Wir präsentieren *pdf2gtfs*, mit dem man die Fahrplaninformation aus PDF-Fahrplänen extrahieren und in dem maschinenlesbaren Format GTFS exportieren kann. Dafür brauchen wir die Koordinaten der Haltestellen. Wir nutzen OpenStreetMap, um die Haltestellenkoordinaten nur anhand des Haltestellennamens und der anderen Haltestellen einer Route zu finden. Wir stellen einen neuen Tabellenextraktionsalgorithmus für *pdf2gtfs* vor. Dieser ist Spalten- und Zeilenunabhängig, was bedeutet, dass wir in der Lage sind, Fahrpläne unabhängig von ihrer Orientierung zu extrahieren. Als weitere Verbesserung für *pdf2gtfs* inkludieren wir zusätzliche Information von OpenStreetMap über Haltestellen. Wie zum Beispiel die international eindeutige IFOPT einer Haltestelle, oder ob eine Haltestelle barrierefrei ist. Außerdem erstellen wir das Tool *p2g-eval*, mit dem wir die erkannten Positionen von Haltestellen, anhand der wahren Daten, evaluieren können. Mithilfe dieser Evaluation, versuchen wir die Teile von *pdf2gtfs* aufzudecken, welche Verbesserungspotenzial besitzen. Wir zeigen, dass der neue Tabellenextraktionsalgorithmus verglichen mit dem alten Tabellenextraktionsalgorithmus, sowie einem anderem, aktuellem Tabellenextraktionstool, präzisere Ergebnisse erzielt. Ferner zeigen wir, dass sich die gefundenen Haltestellenpositionen in der Regel nahe der wahren Position befinden.

Contents

1	Introduction	1
2	Related Work	4
2.1	Algorithmic Extraction of Data in Tables in PDF Documents	4
2.2	Camelot and Tabula-Java	4
2.3	Solutions Based on Machine Learning	5
2.4	PDFTables	5
3	Background	7
3.1	Bounding Box	7
3.2	PDF	8
3.3	Tables	9
3.3.1	Caveats	12
3.4	PDF Extraction	13
3.4.1	Caveats	15
3.5	GTFS	15
3.6	OpenStreetMap	18
3.7	Graph Search using Dijkstra’s Algorithm	20
4	Approach	23
4.1	Extraction Tool <i>pdf2gtfs</i>	23
4.1.1	Old Table Extraction	24
4.1.2	GTFS Creation	27
4.1.3	Location Detection	27

4.1.4	Export	32
4.2	New Table Extraction	32
4.2.1	Data Structures	33
4.2.2	Character Extraction	34
4.2.3	Basic Type Inference	35
4.2.4	Table Creation and Splitting	36
4.2.5	Table Expansion	38
4.2.6	Advanced Type Detection	39
4.2.7	Cleanup and TimeTable Creation	41
4.3	Evaluation Tool <i>p2g-eval</i>	42
5	Experiments	44
5.1	Location Detection Evaluation	44
5.1.1	Dataset Preparation	44
5.1.2	Results	46
5.2	Table Extraction Evaluation	47
5.2.1	Dataset Preparation	48
5.2.2	Configuration	50
5.2.3	Evaluation Measures	51
5.2.4	Results	53
5.3	Performance	56
6	Conclusion	59
7	Future work	61
7.1	Improvements for pdf2gtfs	61
7.2	Improvements for p2g-eval	63
8	Acknowledgments	64
	Bibliography	65

List of Figures

1	A timetable from a PDF schedule of the tram ‘Linie 1’ of the VAG Freiburg [1].	1
2	The true celltypes of the cells in a table. The legend in the last two row shows the cell type of the cells with the same color and number.	12
3	A timetable of the railway line ‘RB90’ of the RMV [2]	13
4	The hierarchical output of <i>pdfminer.sizes</i> ’ layout analysis [3].	14
5	An example of a directed weighted graph.	21
6	The possible celltypes of different cells in a table. The legend in the last two rows shows the cell type of the cells with the same color and number.	35
7	The possible celltypes of different cells in a table. The legend in the last two rows shows the cell type of the cells with the same color and number.	40
8	The number of detected and missing stops closer to the ground truth than the specified distance.	47
9	An excerpt of the ‘ART 42’ route from the TTT dataset.	49

List of Tables

1	Excerpts of the <code>stops.txt</code> , <code>calendar.txt</code> , and <code>routes.txt</code> of an example GTFS feed.	16
2	Excerpts of the <code>trips.txt</code> and <code>stop_times.txt</code> of an example GTFS feed.	17
3	The min, max, mean, and standard deviation for detected and missing locations for all datasets. All distances in meters and rounded to the nearest integer.	46
4	The precision recall and F1-score for the different datasets and table-extraction methods	53
5	The precision recall and F1-score for the different datasets and table-extraction methods	55
6	The mean and weighted (by table-count) mean of the precision recall and F1-score for all datasets and table extraction methods	56

List of Listings

1	An example-query for QLever using SPARQL.	19
2	We show how we create <code>Fields</code> from a line of characters.	25
3	A rough overview on how the location search works.	30

1 Introduction

There exists an abundance of schedule information in PDF timetables, like the one shown in Figure 1. These timetables are made publicly available by their respective transit agencies. However, the raw data contained within the tables, often is not.

	Montag - Freitag																Samstag			
VERKEHRSHINWEIS		V		V		V														
Moosweiher &	ab	20.13	20.16	20.28	20.33	20.43	20.52	20.58	21.13	21.28	21.43	21.58	22.13	22.22	22.43	23.13	23.43	0.13	0.43	4.13
Diakoniekrankenhaus &		20.14	20.17	20.29	20.34	20.44	20.53	20.59	21.14	21.29	21.44	21.59	22.14	22.23	22.44	23.14	23.44	0.14	0.44	4.14
Moosgrund &		20.15	20.18	20.30	20.35	20.45	20.54	21.00	21.15	21.30	21.45	22.00	22.15	22.24	22.45	23.15	23.45	0.15	0.45	4.15
Paduaallee &		20.17	20.20	20.32	20.37	20.47	20.56	21.02	21.17	21.32	21.47	22.02	22.17	22.26	22.47	23.17	23.47	0.17	0.47	4.17
Betzenhauser Torplatz &		20.18	20.21	20.33	20.38	20.48	20.57	21.03	21.18	21.33	21.48	22.03	22.18	22.27	22.48	23.18	23.48	0.18	0.48	4.18
Am Bischofskreuz &		20.20	20.23	20.35	20.40	20.50	20.59	21.05	21.20	21.35	21.50	22.05	22.20	22.29	22.50	23.20	23.50	0.20	0.50	4.20
Runzmattenweg &		20.22	20.25	20.37	20.42	20.52	21.01	21.07	21.22	21.37	21.52	22.07	22.22	22.31	22.52	23.22	23.52	0.22	0.52	4.22
Rathaus im Stühlinger &		20.23	20.26	20.38	20.43	20.53	21.02	21.08	21.23	21.38	21.53	22.08	22.23	22.32	22.53	23.23	23.53	0.23	0.53	4.23
Eschholzstraße &		20.25	20.28	20.40	20.45	20.55	21.04	21.10	21.25	21.40	21.55	22.10	22.25	22.34	22.55	23.25	23.55	0.25	0.55	4.25
Hauptbahnhof &		20.26	20.29	20.41	20.46	20.56	21.05	21.11	21.26	21.41	21.56	22.11	22.26	22.35	22.56	23.26	23.56	0.26	0.56	4.26
Stadttheater &		20.28	20.31	20.43	20.48	20.57	21.07	21.12	21.27	21.42	21.57	22.12	22.27	22.37	22.57	23.27	23.57	0.27	0.57	alle 4.27
Bertoldsbrunnen	an	20.30	20.33	20.45	20.50	20.59	21.09	21.14	21.29	21.44	21.59	22.14	22.29	22.39	22.59	23.29	23.59	0.29	0.59	30 4.29
Bertoldsbrunnen	ab	20.31	—	20.46	—	21.01	—	21.16	21.31	21.46	22.01	22.16	22.31	—	23.01	23.31	0.01	0.31	1.01	Min. 4.31
Oberlinden		20.32	—	20.47	—	21.02	—	21.17	21.32	21.47	22.02	22.17	22.32	—	23.02	23.32	0.02	0.32	1.02	4.32
Schwabentorbrücke &		20.34	—	20.49	—	21.04	—	21.19	21.34	21.49	22.04	22.19	22.34	—	23.04	23.34	0.04	0.34	1.04	4.34
Brauerei Ganter &		20.35	—	20.50	—	21.05	—	21.20	21.35	21.50	22.05	22.20	22.35	—	23.05	23.35	0.05	0.35	1.05	4.35
Maria-Hilf-Kirche &		20.36	—	20.51	—	21.06	—	21.21	21.36	21.51	22.06	22.21	22.36	—	23.06	23.36	0.06	0.36	1.06	4.36
Alter Messplatz &		20.37	—	20.52	—	21.07	—	21.22	21.37	21.52	22.07	22.22	22.37	—	23.07	23.37	0.07	0.37	1.07	4.37
Musikhochschule &		20.39	—	20.54	—	21.09	—	21.24	21.39	21.54	22.09	22.24	22.39	—	23.09	23.39	0.09	0.39	1.09	4.39
Emil-Gott-Strasse &		20.40	—	20.55	—	21.10	—	21.25	21.40	21.55	22.10	22.25	22.40	—	23.10	23.40	0.10	0.40	1.10	4.40
Hasemannstraße &		20.41	—	20.56	—	21.11	—	21.26	21.41	21.56	22.11	22.26	22.41	—	23.11	23.41	0.11	0.41	1.11	4.41
Römerhof &		20.42	—	20.57	—	21.12	—	21.27	21.42	21.57	22.12	22.27	22.42	—	23.12	23.42	0.12	0.42	1.12	4.42
Laßbergstraße &	an	20.44	—	20.59	—	21.14	—	21.29	21.44	21.59	22.14	22.29	22.44	—	23.14	23.44	0.14	0.44	1.14	4.44

Figure 1: A timetable from a PDF schedule of the tram ‘Linie 1’ of the VAG Freiburg [1].

Due to the way a PDF is constructed, extraction of this information from the PDF is not trivial. This is because text in a PDF is stored in text objects (Section 3.2). Each text object contains the text, the font the text should be displayed in, as well as the position at which the text should be displayed. For the position, PDFs use a coordinate system with the bottom-left corner of the page as the origin. The coordinate system uses points as the unit, where one point equals one seventy-second of an inch. This way of storing text makes it easy to display in a compatible and portable manner. However, it also makes it harder to extract it.

There does not exist a one-size-fits-all solution to easily extract tables from PDF documents, at least not with equally good results. This is largely due to the different formats of tables. Though the PDF standard allows it, most PDFs do not contain any additional metadata about the structure of their tables. While there are solutions that aim to provide PDF table extraction in general, specialized solutions can often prove to be more accurate. Thus, we created *pdf2gtfs*, which can detect and extract the schedule information contained in timetables [4]. The output of *pdf2gtfs* is a GTFS feed, which is a machine-readable data format for schedule data.

We may understand the problem of table extraction as a classification problem. We need to classify whether a word on a page of the PDF is part of a table (and also, which one, if there are multiple tables). We define a table cell as a collection of words within the table that are related to each other by their content, position, or both. We can derive a cell type from the cells' content, its absolute position, and its position relative to other cells. In the case of *pdf2gtfs*, we need to classify the cell type of the different table cells, as well. Otherwise, we would not be able to differentiate between cells that contain a time and cells that contain a stop, for example.

The largest part of the timetable is the body, which contains cells of type 'time'. That is, cells that contain text that represents a time (e.g., '09:42'). These cells are usually easiest to detect, because of the simple yet restrictive format of their text. Hence, we create a table extraction algorithm that uses these cells to detect the body of the timetable. We then incrementally add more cells to the timetable, as long as the cells can be aligned to it. A previous table extraction algorithm in our tool *pdf2gtfs* used only the vertical and horizontal position of the cells to detect the timetables.

Before we export the data from the timetables using *pdf2gtfs*, we use the names of the stops together with the information from the open map-service OpenStreetMap (OSM) [5] to detect the location of each stop. We use QLever [6], a query engine able to work on OpenStreetMaps' dataset, to retrieve that information.

We compare our algorithm against a previous algorithm of *pdf2gtfs*, as well as another, state-of-the-art table extraction method, namely *PDFTables* [8]. We may upload a PDF to *PDFTables*, which will extract the tables from within the PDF and offer us different file-formats, like `.csv`, to download the contained data. To compare the table extraction methods, we create datasets for different public-transit agencies that we use to evaluate the accuracy of the table extraction. We show that our algorithm achieves higher accuracy on these datasets than the other solutions.

We also evaluate the location detection of *pdf2gtfs*. For this purpose, we created *p2g-eval* [9]. It enables automatic evaluation of the stop locations of one GTFS feed using another GTFS feed, given a mapping between the two feeds.

This thesis is structured as follows. In Chapter 2 we discuss previous work on the problem of table extraction. Then, we introduce some of the terminology and tools we use (Chapter 3). Next, we explain how *pdf2gtfs* and both its table extraction algorithms work, as well as how we designed *p2g-eval* (Chapter 4). In Chapter 5 we evaluate the location detection and table extraction of *pdf2gtfs*. We discuss these results in Chapter 6. In the last chapter, Chapter 7, we illustrate some promising ideas to improve *pdf2gtfs* in various ways, based on the evaluation results.

2 Related Work

In this chapter, we show some existing solutions and research results for the problem of table extraction.

2.1 Algorithmic Extraction of Data in Tables in PDF Documents

This Master’s thesis by Anssi Nurminen provides a method that aims to solve the more general problem of table extraction [10]. It works by using the alignment of text in rows and columns to detect the tables. They also use the vertical and horizontal lines present in some tables, to improve the extraction accuracy. To use these graphical elements, they convert each page into a gray-scale image and use changes in the pixel brightness to detect these lines. They conclude, that their algorithm performs “very well in defining table structure in correctly defined table areas” [10].

2.2 Camelot and Tabula-Java

Both *Camelot* [11] and *tabula-java* [12] are open-source libraries that allow table extraction from PDF documents. *Tabula-java* is, as the name implies, written in Java, while *Camelot* is written in Python. *Camelot* is based on the work of Anssi

Nurminen (2.1) and aims to provide more adjustment possibilities to the user than other solutions. Both *Camelot* and *tabula-java* support two modes. The first, called ‘stream’, uses the text position and the text alignment for the table detection, while the other mode, called ‘lattice’, uses the horizontal and vertical lines. When we tried *tabula-java* and *Camelot* on timetables, both had similar issues. When using the lattice-mode, the main problem was that only some cells of the timetables were completely enclosed by lines. Other times, the lines grouped several rows or columns together. The stream-mode on the other hand, often did not properly split text of different table cells. *Camelot* also provides the neat feature to display an image of the bounding box of the different detected elements, like the text, lines, and table.

2.3 Solutions Based on Machine Learning

In recent years, more and more effort is spent to create and improve table extraction tools that use machine learning. For example, the Master’s thesis of Muhammad Moez Malik [13] employs deep-learning based object-detectors, to detect the table regions of solar-panel datasheets. Then, they provide the detected table coordinates to the aforementioned *tabula-java* or *Camelot*, to extract the table. This seems to yield good results for these types of tables.

2.4 PDFTables

There also exist (closed-source) web services, such as *PDFTables* that provide table extraction from PDF documents, as well. Being proprietary, it is difficult to tell how exactly their table extraction works. It does seem to work using an approach like the one used by *Camelot* and *tabula-java*; at least, *PDFTables* makes similar mistakes to these. Sadly, there is no way to fine-tune *PDFTables*. As noted, our initial experiments suggest they seem to provide similar results to *Camelot* and

tabula-java. That, combined with its ease-of-use, were the deciding factors why we use *PDFTables* for the evaluation.

3 Background

In this chapter, we introduce some of the terms we use. We begin by explaining in Section 3.1 what a bounding box is. We briefly describe how the portable document format (PDF) stores text and why table extraction is not trivial (Section 3.2). In Section 3.3, we explain the building blocks of a table. The next section, Section 3.4, is about the tools and terms we use when extracting the text of a PDF. Then, in Section 3.5, we introduce the GTFS-format and show some examples. We also explain how OpenStreetMap (OSM) stores its values and how to retrieve them (Section 3.6). Finally, in Section 3.7 we explain what a directed, weighted graph is, and how we can use Dijkstra’s algorithm to find a shortest path in that graph.

3.1 Bounding Box

We can use a bounding box (also known as bbox) to represent the space an object occupies. For example, `[bounding]` has a different bounding box than `[box]`. To define the bounding box of an object, we use its lowest and highest x - and y -coordinates. We define these as x_0 and x_1 for the lowest and highest x -coordinate, respectively; and y_0 and y_1 for the lowest and highest y -coordinate, respectively. Two bounding boxes may overlap vertically or horizontally. In summary, a bounding box is a simple way to define the position and size of an object.

3.2 PDF

Here we give a broad overview of the portable document format (PDF). We will focus on specific, basic aspects and terminology necessary for understanding this thesis. The full format is a lot more detailed and versatile.

We cannot extract the text of PDF as easily as we can display it. This is largely because the PDF uses a layout-based system to store its content. There are many different kinds of objects in a PDF document. However, only the following are relevant to us. The text objects, which contain the text of the document; the vector graphics, used for drawing shapes, often generated by some other program; and images, usually used for photographs, though this also includes scans of text.

Each page of a PDF uses a coordinate system, with the bottom-left corner of the page being the origin. The unit of measurement used in the coordinate system is points, where one point equals one seventy-second of an inch.

The text in a PDF is not stored as plain text. The position of each letter on the page is defined using its bounding box. The letter of the character, and its font properties are stored as well. The font properties include the font family, font size in points, and font effects (such as boldness, italicization, and color).

The coordinate system also makes it easy to draw vector-graphics. To draw a straight line, we simply specify its start and end points in the coordinate system. Then, we define the thickness and color of the line. The PDF viewer takes care of actually drawing the character at the right position.

This way of defining objects makes it easy to always display a PDFs' content the same way, independently of the used operating system. At the same time, it makes text extraction difficult, because the context of each character is lost. Only the absolute position is stored, so we cannot trivially tell, which characters are next to another. The same applies to word boundaries; a space is often only emulated, by a

greater distance between two characters. Therefore, it is difficult to decide whether two characters belong to the same word. The bounding box is also defined in a way that it contains just the character. As such, two different characters (e.g., ‘q’ and ‘W’) may have slight differences in their y-position, even if they are on the same line, which further complicates the previously mentioned problems.

Another difficulty is to decide whether a character (or multiple characters) are relevant, when only some of the PDFs content should be extracted. For instance, when extracting a table we cannot easily decide which characters are part of the table. In this case, the vertical and horizontal lines, as well as text alignment may prove to be indicators for this boundary. Though the PDF specification offers some ways to specify which text is part of a table, these are seldom used.

3.3 Tables

In this section, we define the terminology we use when referring to specific parts of a timetable. For a more general introduction about tables, see the Master’s thesis of Ansii Nurminen [10].

We define the characters in a similar manner as the text objects in Section 3.2. Each character has a bounding box, a letter, and font properties. We use the following four coordinates to define the bounding box: the left, x_0 ; the right, x_1 ; the top, y_0 ; and the bottom, y_1 . Note that, compared to the bounding box used in a PDF, we use the *upper*-left corner as origin. This is only an implementation detail, because it feels more intuitive to us. Further, if we talk about directions in this thesis, we mean the four cardinal directions: north, east, south, and west. Similarly, an orientation means either vertical or horizontal.

The basic building block of a table is the cell. A cell consists of zero or more characters. If a cell has no characters, we call it an empty-cell. Each cell also has a row and column. The row contains all cells that are vertically aligned. That is, every cell of a

row has the same (or very similar) y-coordinates. The column contains all cells that are horizontally aligned, with each cell having equal or similar x_0 -coordinates. While there do exist multi-column or multi-row cells, in the case of timetables they are rarely used. Also, when we encountered timetables with multi-column or multi-row cells, only one of the rows or columns contains non-empty cells. These multi-row or multi-column cells can easily be merged into one, without changing the overall structure of the timetable.

Every cell also has up to four neighbors. A cell is the horizontal neighbor of another cell, if the cells are consecutive entries of the same row. Similarly, a cell is the vertical neighbor of another cell, if the cells are consecutive entries of the same column. This also means that the neighbor-relation is symmetric, i.e., if one cell is neighbor of another cell the reverse is also true. If a cell is the first cell in a specific direction, it has no neighbor in that direction. We do not consider the case of single-column or single-row tables. Therefore, the cells in the corners only have two neighbors, while the cells in the first and last row and column have three. Every other cell has four neighbors; one in each direction.

Finally, each cell of a timetable has a cell type. We can use the cell type to describe the meaning of a cells' content. For instance, we can define a cell type 'stop' and call a cell with this type a stop cell. We differentiate between 10 different normal cell types. These cell types contain text of the table.

Time	These cells contain the time of arrival and departure of the public transit vehicle. For a single PDF, they all use the same format.
Time annotation	These are rarely used, but sometimes there are annotations directly adjacent to time cells.
Stop	Cells of this type contain the names of the stops.
Stop annotation	These cells have a text like 'arr.' or 'dep.', for example. They indicate, whether the public transit vehicle arrives at or departs from the stop, defined by an adjacent stop cell.

Days	These cells tell us on which weekdays service occurs. They indicate this either for the whole timetable, or for only some entries.
Repeat identifier	These are the cells that usually exist on both sides of a repeat interval. At the same time, they can only be found between two columns or rows of time cells. They contain the surrounding text, like “every ... min.”, where ‘...’ is the interval.
Repeat interval	These cells contain the interval(s) of the service repetition.
Route identifier	These have a text like ‘Route’ or ‘Line’. They indicate that either their row or column contains route annotations.
Route annotation	These usually contain the (short) name of a route.
Entry identifier	They have text like ‘Verkehrshinweis’ (= traffic info). They indicate that either their row or column contains entry annotations.
Entry annotation	These are usually single characters that indicate, for example, that a route is not serviced on some days, like the 31st of December.

There are 2 additional special types, namely **Other** and **Empty**. We use **Empty** to denote table cells that do not contain and **Other** to denote all cells that are not part of the table or that cannot be classified using the other types.

An example for stop cells are the cells with the green (3) box in Figure 2. Each stop cell defines a stop and requires that either its row or column contain the times when a public transit vehicle arrives at or departs from that stop. Similarly, the route annotation (the red (2) box) can usually be found in the same row or column as timecells, as well. It contains extra information (here, the short route name) about a route. The first cell of the table, the route identifier, can be used to define that its row or column contains route annotations. The cells with orange (4) box are time cells, which contain the actual times. Also note the two empty cells between the time cells. As seen in these examples, the cell type may depend on the contents of a

cell, or its neighbors. It may also change based on more distant rows or columns. We can use the cell type when extracting the timetable, to narrow down the number of possible cells (Section 4.2).

<div>1</div>	Route	<div>2</div>	S1	<div>2</div>	S2	<div>2</div>	S3
<div>3</div>	Central	<div>4</div>	13:36	<div>5</div>		<div>4</div>	15:46
<div>3</div>	Station	<div>4</div>	13:38	<div>4</div>	14:33	<div>5</div>	
<div>3</div>	Airport	<div>4</div>	13:42	<div>4</div>	14:42	<div>4</div>	16:00

<div>1</div>	Route Identifier	<div>2</div>	Route Annotation		
<div>3</div>	Stop	<div>4</div>	Time	<div>5</div>	Empty

Figure 2: The true celltypes of the cells in a table. The legend in the last two row shows the cell type of the cells with the same color and number.

3.3.1 Caveats

Some timetable formats use some special notation, as the ones we describe here. We need to detect these, if possible at all, to ensure proper table extraction.

For example, the timetable in Figure 3 contains “connections”. That is, stops and stop times that are not part of the route, but those of different, frequently used, or otherwise important routes. In the figure, the first three stops are part of a connection from ‘Frankfurt’ to ‘Limburg’. However, these are serviced by a different public transport vehicle on a different route (‘G 20’).

Montag - Freitag				Samstag														
G 20 Frankfurt Hbf	ab	19.29	20.29	21.29	6.29	8.29	10.29	12.29	14.29	16.29	18.29	20.29	21.29	22.29	0.29			
G 20 Niedernhausen	ab	20.01	21.01	22.01	7.01	9.01	11.01	13.01	15.01	17.01	19.01	21.01	22.01	23.01	1.01			
G 20 Limburg	an	20.41	21.41	22.41	7.39	9.41	11.41	13.41	15.41	17.41	19.41	21.41	22.41	23.41	1.42			
Hinweise																BUS	BUS	
Limburg	ab	20.45	21.58	22.47	7.45	9.45	11.45	13.45	15.45	17.45	19.45	21.58	22.47					
- ZOB Nord	ab															23.50	1.50	
- Diez Ost		20.49	22.02	22.51	7.49	9.49	11.49	13.49	15.49	17.49	19.49	22.02	22.51					
Staffel		20.53	22.10	22.55	7.53	9.53	11.53	13.53	15.53	17.53	19.54	22.10	22.55					
- Ost																0.01	2.01	
Elz Mitte																0.06	2.06	
- Elz		20.56	22.13	22.58	7.56	9.56	11.56	13.56	15.56	17.56	19.57	22.13	22.58					
Niederhadamar		20.59	22.16	23.01	7.59	9.59	11.59	13.59	15.59	17.59	20.00	22.16	23.01					
- Süd																0.11	2.11	
Hadamar Stadtmittel																0.15	2.15	
- Hadamar		21.02	22.19	23.04	8.06	10.06	12.06	14.06	16.06	18.06	20.06	22.19	23.04					
Niederzeuzheim		21.06	22.23	23.08	8.10	10.10	12.10	14.10	16.10	18.10	20.10	22.23	23.08					
- Ort																0.20	2.20	
Frickhofen		21.12	22.29	23.14	8.16	10.16	12.16	14.16	16.16	18.16	20.16	22.29	23.14			0.32	2.32	
Wilsenroth		21.16	22.33	23.18	8.20	10.20	12.20	14.20	16.20	18.20	20.21	22.33	23.18					2.36
Berzhahn #		21.20	22.37	23.22	8.24	10.24	12.24	14.24	16.24	18.24	20.24	22.37	23.22					
- Ortsmitte																		2.40
Willmenrod Ortsmitte																		2.44
- Willmenrod #		21.22	22.39	23.24	8.26	10.26	12.26	14.26	16.26	18.26	20.26	22.39	23.24					
Westerburg #	an	21.26	22.43	23.28	8.30	10.30	12.30	14.30	16.30	18.30	20.30	22.43	23.28					2.50

In Niedernhausen bestehen zusätzliche Anschlüsse von der aus Frankfurt.

Figure 3: A timetable of the railway line ‘RB90’ of the RMV [2]

Another special format, visible in the same figure, that is sometimes used by transit agencies is what we call “split stop names”. These are the stops that start with a hyphen (or in other cases, are indented). In the example, the stop S_2 with text “-ZOB Nord” is a split stop; it starts with a hyphen. The stop S_1 above it with text “Limburg” contains the city (here “Limburg”) that S_2 exists in, as well. On the other hand, S_1 may have been a stop with multiple words as well, where we would have found it more difficult to detect the correct city-name. For the location detection, it is often better to include the city name.

These are just two examples, however. In reality, there exist numerous ways to convey information in a timetable, which we can detect with varying levels of accuracy.

3.4 PDF Extraction

We use *pdfminer.six* [14] to extract the individual characters from the PDF. In the following, we will (briefly) explain how *pdfminer.six* works, as is written in its documentation [3]. In Subsection 3.4.1 we show some caveats regarding the use of *pdfminer.six*.

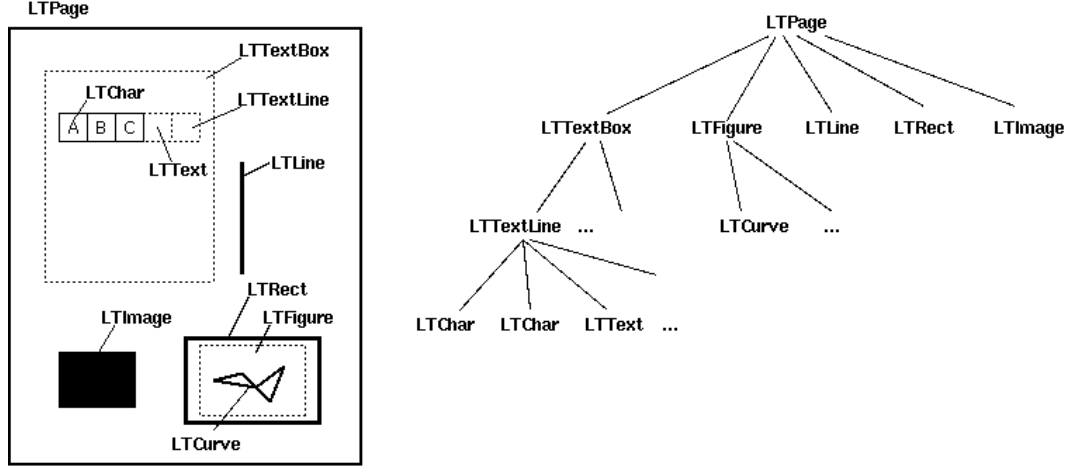


Figure 4: The hierarchical output of *pdfminer.sixes*' layout analysis [3].

In Figure 4 the different data structures are shown that *pdfminer.six* uses. Some of these represent a PDF object, while others are (more abstract) helpers. For us, only the `LTTTextLine`, `LTChar` and `LTAnno` (not shown in the figure) are of importance. The `LTTTextLine` contains `LTChars` that are on the same line, and `LTAnnos`. Two `LTChars` are on the same line, if their x - and y -distance is less than some values. The specific values depend on the size of the object, as well as the `line_overlap` and `char_margin` parameters of *pdfminer.six*. Each `LTChar` contains a single character, its bounding box, and its font, among other things; it is essentially a direct representation of a single character in the PDF. A `LTAnno` does not exist in the PDF, but is created by *pdfminer.six* to represent word and line boundaries. Therefore, a `LTAnno` has no bounding box. Instead, a `LTAnno` with text `'\n'` (a newline) is added to the end of every `LTTTextLine`. Similarly, a `LTAnno` with text `' '` (a single space) is added between two consecutive `LTChars` of a `LTTTextLine`, if the distance between them is greater than the parameter `word_margin`.

3.4.1 Caveats

Sometimes, a character can not be extracted properly. In that case, *pdfminer.six* uses “(cid:X)” as the text for the `LTChar`, where X is some number. The reason is that each character is mapped to both a glyph and a Unicode value. The glyph is used by a PDF viewer to display the character, while the Unicode value is used when extracting or copy-pasting the character [15]. For us, this has only occurred with the German umlauts. Interestingly, in these cases `chr(x)`, which is used to get “the Unicode string of one character with ordinal x” [16], gave the correct umlauts.

There is an “advanced layout analysis” *pdfminer.six* runs by default. It is used to order the `LTTextBoxes` in a non-trivial way. However, this can lead to some of the problems other tools exhibited, like *Camelot* (which uses *pdfminer.six* with this default). Also, depending on the size of the timetables and the number of pages, this analysis can be quite time-consuming. For these reasons, we disable the advanced layout analysis. This tells *pdfminer.six* to use the bottom-left corner of each `LTTextBox` [17] to order them, instead.

3.5 GTFS

The General Transit Feed Specification (GTFS) defines a file format for public transit data, created by Google [7]. A GTFS feed is a `.zip`-archive of multiple comma-separated-values files. However, these files use the `.txt` file extension, instead of `.csv`. Each `.txt` file has a different purpose. For instance, the file `stops.txt` contains the definition of the different stops used in the feed. At least the names and locations, as well as a user-selected ID need to be provided for each stop. The ID defined in one file of a feed may be referenced by entries in other files of the same feed. The `stop_times.txt`, for example, uses the `stop_id` defined in the `stops.txt`, to specify the times a vehicle arrives at and departs from the referenced stop.

stops.txt			
stop_id	stop_name	stop_lat	stop_lon
Central	A	50.0395	8.9503
Station	B	50.1339	8.3913
Airport	C	50.4224	8.2442
...

calendar.txt				
service_id	monday	tuesday	...	sunday
Mondays	1	0	...	0
...

routes.txt		
route_id	route_short_name	route_type
S1	Line 1	1
...

Table 1: Excerpts of the `stops.txt`, `calendar.txt`, and `routes.txt` of an example GTFS feed.

In Figure 1 excerpts of multiple `.txt` files of a single GTFS feed are shown. We omitted some required columns and files for brevity’s sake. As visible in the first table, we define some stops using their IDs, names, and locations. Then, we add a service `Mondays` that is only active on Mondays, as indicated by the 1. Next, we define a route, by giving it an ID and a name `S1`, and specify its route-type (here, 1 for ‘Subway’).

trips.txt				
trip_id	route_id	service_id		
Trip1	S1	Mondays		
...		

stop_times.txt				
trip_id	arrival_time	departure_time	stop_id	stop_sequence
Trip1	13:36:00	13:36:00	Central	0
Trip1	13:38:00	13:40:00	Station	1
Trip1	13:42:00	13:42:00	Airport	2
...

Table 2: Excerpts of the `trips.txt` and `stop_times.txt` of an example GTFS feed.

In Figure 2 excerpts from two more GTFS files of the same feed are shown. The trips are used to map a route to a service. The line shown basically means: “There is a trip with ID `Trip1` that serves the route with ID `S1` on the days specified by the service `Mondays`.”

In the last table, we see the different times, at which the subway will arrive at and depart from the different stops of that same trip. The `stop_sequence` is used to define the order of the stops of a specific trip.

There are two aspects of the specification, that make it difficult to compare two GTFS feeds. First, the IDs used in a feed must only be unique for that feed. On one hand, this makes sense, because feeds are created locally. On the other, this means that if we have two feeds that are equal, apart from the IDs used in the `stops.txt`, it is not immediately obvious. Secondly, a feed is not unique, in the way that there is only a single way to define its routes. For example, we could add a single entry to the `calendar.txt`, to enable a route to be serviced every Monday, as in the previous

figure. Then, we add dates to the `calendar_dates.txt`, at which this route should not be serviced. We could define the same route equivalently, if we only specify the dates where the route is serviced in the `calendar_dates.txt`.

Nowadays, more and more GTFS feeds are being made publicly available by transit agencies. In Germany this is often done as part of the open data movement. There are also public databases of public transit data, like the “OpenData ÖPNV” [18] in Germany. On the other hand, some transit agencies do not provide a way to retrieve their information. Others add a password to their PDF timetables (on accident or on purpose), making even the extraction impossible.

3.6 OpenStreetMap

OpenStreetMap (OSM) [5] is a free and open project that provides geographic data. This information is often manually entered by users.

There exist three fundamental elements to describe objects on the map in OSM:

- Node** A node is a single position on the map. It can be used, for instance, to define the location of a stop [19].
- Way** A way is defined by at least two nodes. It can be used to define map-features with a shape, like the path of street, or the area of a station [20].
- Relation** A relation can be used to define that some nodes or ways are part of a “logical or geometric relationship” [21]. For example, the different stop positions of a station.

Each of these elements can have a number of different tags. A tag consists of a key and a value, and is used to add more information about an object. For instance, we could add the tag with the key ‘highway’ and the value ‘bus_stop’ to a node, to define that the node describes a bus stop.

We can query the information contained in OSM, using a tool like QLever [6]. We use the query language SPARQL [22] in QLever, to specify what information we want to receive. Listing 1 shows an example QLever query. The `PREFIX`s are used to enable use of the shorthand form (e.g., ‘osm’) instead of the full URL. Any word that starts with a question mark is a variable. The query returns the names and locations (Line 6) of all nodes (Line 7) that use the `public_transport` tag (Line 8). The `public_transport` tag is used to define the purpose of a node in public transport. For example, a node with the `public_transport` tag set to `stop_position` defines that a public transport vehicle (like a bus, or tram) stops at the nodes’ location.

Listing 1 An example-query for QLever using SPARQL.

```

1 PREFIX geo: <http://www.opengis.net/ont/geosparql#>
2 PREFIX osm: <https://www.openstreetmap.org/>
3 PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
4 PREFIX osmkey: <https://www.openstreetmap.org/wiki/Key:>
5
6 SELECT ?name ?location WHERE {
7   ?stop rdf:type osm:node .
8   ?stop osmkey:public_transport ?public_transport .
9   ?stop osmkey:name ?name .
10  ?stop geo:hasGeometry ?location .
11 }
```

There also exist specific tags, that can be used in a GTFS feed. For example, the `ref:IFOPT`-tag can be used as `stop_id`. IFOPT stands for Identification of Fixed Objects in Public Transport [23]. As the name implies, the IFOPT is used to (uniquely) identify fixed objects (e.g., stops, stations, and the like) in public transport. The different IFOPTs are also defined in a hierarchical way. For example, the central station in Freiburg, Germany has the IFOPT `de:08311:6508`, while the IFOPT `de:08311:6508:8:1` is used for the first platform of that same station. Another

tag we will use in the Subsection 4.1.4 is the `wheelchair_boarding` tag. This tag is used to define whether a stop is wheelchair accessible.

3.7 Graph Search using Dijkstra's Algorithm

Graph

A graph consists of vertices (or nodes) V and edges E . An edge between two nodes $v_1, v_2 \in V$ can be denoted as (v_1, v_2) . We use a directed, (positively-)weighted graph in our location detection. This type of graph has edges with a direction, that is $(v_1, v_2) \neq (v_2, v_1)$, unless $v_1 = v_2$. The weighted bit means, that each edge has a cost assigned to it. For example, we use $|(v_1, v_2)| = 3$ to denote that the cost of the edge between v_1 and v_2 is 3. We also do not allow edges where the start node is the end node, like (v_1, v_1) .

A path of a directed graph is a series of nodes, such that there is an edge (in the correct direction) between every two consecutive nodes of the path. We use $P = \langle v_1, v_2, v_3 \rangle$, with $v_1, v_2, v_3 \in V$ to specify a path from v_1 to v_2 to v_3 . For a weighted graph, we can also define the cost or length of a path, as the sum of the cost of all edges between the nodes of the path. A path between two nodes is called a “shortest path” between the two, if there is no other path between the same nodes in the graph that has lower costs (though there may exist some with equal costs).

For example, Figure 5 shows a directed, weighted graph with the nodes A , B , C , and D . A path P could then be $P = \langle A, B, C \rangle$, because there is an edge between A and B , and one between A and C . The length of P would be 14. At the same time, there is another path $R = \langle A, D, C \rangle$ with length 7, and a third path $Q = \langle A, B, D, C \rangle$ with length 6.

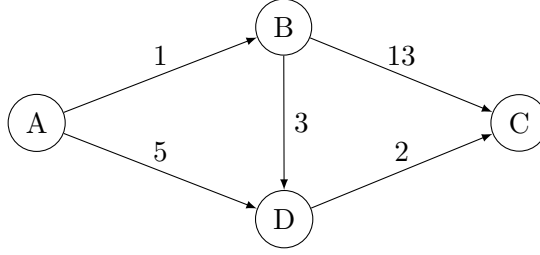


Figure 5: An example of a directed weighted graph.

Finally, a graph contains loops if we can find a path that contains the same node more than once. Then, we call the graph cyclic, otherwise acyclic. In the previous example, the graph has no loops. If we added an edge (B, A) with some cost, we could find a path like $\langle A, B, A \rangle$ that contains A twice. The graphs we use in the location detection do not contain loops either.

Dijkstra's Algorithm

Dijkstra's algorithm [24] can be used to search for the shortest path in a directed, acyclic graph.

In principle, it works like this: First, we add a cost C_v to each node v , which we define as the lowest cost of a path from the start node s to v . For all nodes except s , we initially set this cost to infinity; for s we set it to 0. Similarly, we add a parent node P_v to each v . That is, the origin of the edge we used to assign a specific cost to a node. Thus, no node has a parent at the beginning. Each node can also be visited and unvisited. In the beginning, all nodes are unvisited.

Now, to find the shortest path between two nodes, 'start' and 'end', in a graph with nodes V using Dijkstra's algorithm, we do the following:

1. We select an unvisited node $v \in V$ with the lowest cost.
2. If v is the same node as the end node, we are done.

3. For all unvisited nodes w where an edge (v, w) exists in the graph, we update the cost and parent of w , if this condition applies: $C_v + |(v, w)| < C_w$, where C_w is the current cost of w . That is, the cost when using v as parent is lower than the current cost. If the condition applies, we set C_w to $C_v + |(v, w)|$ and $P_w = v$.
4. We mark v as visited and continue with 1.

Note that, because all nodes have a cost of infinity except the start node, the first iteration will naturally use the start node. Also, there is no need to store the current shortest path. Once we assign a cost and parent to the end node, we can recursively get the parent of each node of the path, starting at the end node.

4 Approach

In this chapter, we give a high-level overview on how *pdf2gtfs* works (Section 4.1). Here, we also include a short introduction to how the old table extraction algorithm works. In Section 4.2 we explain the new table extraction in detail. Lastly, we show how we designed the evaluation tool *p2g-eval* in Section 4.3.

Because *pdf2gtfs* currently cannot extract tables spanning multiple pages, in the following sections we will always talk about a single page. When we run *pdf2gtfs* with a multi-page PDF, it will iteratively do the same steps for each page.

4.1 Extraction Tool *pdf2gtfs*

Here, we will briefly explain the different parts of *pdf2gtfs*. We created a blogpost where we go into more detail, though some parts may be slightly outdated [25].

In Subsection 4.1.1 we show how the old table extraction works. Then, we introduce the `TimeTable` and `GTFShandler` that we use to create the GTFS feed in memory (Subsection 4.1.2). In Subsection 4.1.3 we explain how we use QLever and OSM to detect the location of the stops. Finally, in Subsection 4.1.4, we export the feed.

4.1.1 Old Table Extraction

We preprocess the input file using *ghostscript* [26], to remove unnecessary content. This removes the images and vector graphics from the PDF. We do not use these in our table extraction, thus they can be safely removed.

Next, we use *pdfminer.six* to extract the individual **LTChars** from the PDF. As stated in Section 3.4, each **LTChar** stores its letter, bounding box and rotation.

Data Structures

Here, we give some definitions about the data structures we use in the old table extraction. We use **Rows** to represent the rows in a table, **Columns** to represent the columns, and **Fields** to represent non-empty cells. A **Field** contains characters of the same row and has a bounding box, which is based on the bounding boxes of its characters. Similarly, **Rows** and **Columns** contain all **Fields** in the same row or column, respectively, and have a bounding box based on their **Fields**' bounding boxes. The **Fields**, **Rows**, and **Columns** also have a type. For the **Fields**, this is basically equivalent to the cell type, described in Section 3.3, with the difference that a **Field** can only have a subset of these types.

The type of the **Rows** and **Columns** depends on the **Fields** they contain. For instance, if a **Row** contains a **HeaderField** it is a **HeaderRow**, and if a **Column** contains a **StopField** it is a **StopColumn**. Finally, a **PDFTable** consists of **Rows** and **Columns**. It represents a complete timetable of the input file.

PDFTable Creation

Here, we explain how we create the **PDFTables** for a single page.

We first group the `LTChars` into lines. Each line is simply a list of `LTChars`. For this, we sort the `LTChars` by their y_0 -coordinate. Then, we add a new list to the lines containing the first `LTChar` and iterate over the rest. We compare the distance between the current `LTChar` and the first entry of the current line. If the distance is smaller than the user-defined `max_char_dist`, we add it to the current line. Otherwise, we add a new line to the list of lines. Simply sorting the `LTChars` by their y_0 -coordinate would not work, because of the tolerance described in Section 3.2.

Then, we create a `Row` for each line, as described in Listing 2. As can be seen, we iterate over the sorted line and create a new `Field`, if the distance between two consecutive `LTChars` is greater than `max_char_dist`. Once this is done, we simply use all `Fields` of a line, to create a `Row`.

Listing 2 We show how we create `Fields` from a line of characters.

```
def line_to_row(line: list[Character]) -> list[Row]:
    # We want to iterate over the chars from left to right.
    line.sort(key=attrgetter("bbox.x_0"))
    fields: list[Field] = []
    # Create a Field from the first char, to make the loop cleaner.
    field: Field = Field.from_char(line[0])
    # Thus, we start the loop at the second char.
    for char in line[1:]:
        # The x-Distance between two consecutive characters.
        # That is, the space between the right side of the first
        # and left side of the second character
        if abs(field.last_char.bbox.x_1 - char.bbox.x_0) > max_distance:
            fields.append(field)
            field = Field()
        field.add_char(char)
    return Row(fields)
```

We do this for all lines and group the resulting **Rows** into **PDFTables**, similar to how we create the **Fields**. We iterate over the **Rows**. If the distance between two consecutive **Rows** is greater than the user-defined **max_row_dist**, we create a new **PDFTable**. The lines were sorted when we created the **Rows**, hence, the **Rows** are sorted in the same way.

Next, we detect the type of each **Field** and **Row**. We split **PDFTables** with multiple **HeaderRows** horizontally, such that each **PDFTable** contains only a single one. Then, we create the **Columns** of the **PDFTable**. For this, we create a **Column** for each **Field** of the first **Row**. We iterate over the **Fields** of the other **Rows**. If a **Fields**' bounding box overlaps with an existing **Column**, we add it to that **Column**. Otherwise, we create a new **Column** for that **Field**. We sort the **Columns** based on their bounding boxes x_0 -coordinate, which is the lowest x_0 -coordinate of their respective **Fields**.

Once we have created all **Columns**, we detect the type of each **Field**, **Row**, and **Column**. The reason we detect the type for the **Fields** and **Rows** again is that their type may have changed.

We split any **PDFTables** vertically that contain more than one **StopColumn**, such that each **PDFTable** contains only a single one. Then, we run the type detection one last time, in case the splitting has altered the type of any **Fields**, **Rows**, or **Columns**.

We then run some cleanup steps. In particular, to fix the split stop names described in Subsection 3.3.1.

Lastly, we create a **TimeTable** from each **PDFTable**. Whereas a **PDFTable** contains the bounding box and other low-level information of each **Field**, a **TimeTable** contains a list of stops and a list of **TimeTableEntries**. In simple terms, each **TimeTableEntry** is a **TimeColumn** of a **PDFTable**. However, instead of containing a list of **Fields**, it contains mapping between the stops and the times. In abstract terms, a **TimeTable** is closer to the GTFS, while the **PDFTable** is closer to the PDF.

4.1.2 GTFS Creation

At this point, we have extracted all necessary data from the PDF and have created a `TimeTable` for each timetable in the PDF. Next, we want to create the GTFS feed in memory. The reason we create the GTFS feed before we do the location detection, is because this makes some of the information we need more accessible to us. For instance, we require the stops of each different route, because we detect the locations of each route separately.

To properly store the GTFS feed in memory we create two types of data structures. The first one represents a single entry in a single file of the GTFS feed. For example, the `StopTimesEntry` has a variable for each column in the `stop_times.txt`, like the `trip_id` or the `arrival_time`. In a way, these data structures mirror the structure of the file they represent. The other type of data structures are the ones that represent a single file of the GTFS feed. For example, the `StopTimes` contains functions to create new `StopTimesEntry` objects from `TimeTableEntries`. This makes it easy, to create the `stop_times.txt`. We simply have to iterate over all `StopTimesEntry` objects and add their values to the same file.

The user also has the option, to include their own GTFS files. In that case, we read the input files using these data structures first. For example, we use the `Stops` and `StopEntries`, to read the `stops.txt`. These can then be used in the location detection.

4.1.3 Location Detection

In this subsection, we will describe how we search for the locations of the stops.

First, we use QLever to fetch the names, locations and other, optional OSM-values we may need of OSM-nodes that represent public transport locations.

The **names** of an OSM-node consists of a string of different names of each location, separated by the pipe-character (`|`). We normalize each of the names to improve the search for the stop names. In this step, we remove all text in parentheses, lower all characters, and sort the words of a name in alphabetical order. We also expand common abbreviations to their full form, like ‘Hbf’ to ‘Hauptbahnhof’ (= central station). We use a user-provided dictionary for this. The normalization of the names is the step that takes by far the longest, with regard to retrieving the OSM-nodes. Thus, we cache these values in a `.csv` file.

As stated in Subsection 4.1.2, we detect the stop locations for each route separately. We do this in case the timetables contain stops that are not all serviced in sequence. If we only detected the stops for the longest route, we would not be able to properly detect stops that are serviced by one route and not the other.

We read the cached OSM-nodes into a pandas **DataFrame** and pre-filter the nodes, based on the names of all stops of the routes. Then, for each stop we create a **DataFrame** that only contains the possible locations for this stop. If a `stops.txt` file was given as input, we search this file beforehand. If any stop was found, the **DataFrame** for that stop will only contain that single entry.

The search for the correct locations can be specified as a shortest-path search in a directed, weighted graph. Each stop location is a node in the graph. Each possible node of one stop has an edge to all possible nodes of the next stop. The cost of an edge between a node of a stop and a node of the next stop can be defined as the sum of three costs. The name cost, node cost and travel cost. The name cost represents the difference between the name of the node and its stop. Because of the way we filter the **DataFrame** beforehand, we can simply use the difference in length for this, instead of a more performance-intense accurate calculation like the edit-distance. (We require a stop-node name to contain all words of a stop name.) The node cost depends on the type of node. For example, a bus stop has a lower node cost than a railway stop, if the user-defined route type is ‘Bus’. The travel cost

depends on the distance between the two nodes. Depending on the selected settings, this may simply mean that higher distance equates higher travel cost. On the other hand, we may also use an estimation of the average speed and the travel time. The average speed is different based on the route type, while we get the travel time from the `GTFSHandler`. Then, we simply multiply the travel time with the average speed, to get the expected distance. In this case, the travel cost is calculated using the difference between the actual distance and the expected distance. The nodes of the first stop have a travel cost of 0. We normalize each of the costs, such that neither is the single deciding factor. Otherwise, the travel distance would most likely dominate the node selection.

The filter step may sometimes filter out the true stop location, or the true stop location may simply not exist on OSM. Thus, we also define `MissingNodes`. We need these, because each node is only connected to nodes of the next stop. Thus, if one stop has no nodes at all, no shortest path can be found. A `MissingNode` has a high node-cost and the travel cost is defined using the parent of the `MissingNode`. We create these `MissingNodes` on the fly, for stops that do not have any nodes.

Similarly, for existing locations, we define `ExistingNodes`. These are used for stops that were found in the input GTFS files. If an `ExistingNode` exists for a stop, we simply remove any other node for this stop. We also do not create `MissingNodes` for stops that have an `ExistingNode`. Thus, regardless of the settings, these nodes will always be part of the locations returned by the location detection.

In Listing 3, we show how we use Dijkstra’s algorithm (described in Section 3.7), to solve the shortest-path search. The unvisited nodes are stored in a min heap, sorted by each nodes’ overall cost.

Listing 3 A rough overview on how the location search works.

```
1 def shortest_path(nodes: MinHeap[Node], last_stop: Stop) -> Node:
2     while True:
3         # Get the node with the lowest cost.
4         current_node = nodes.pop()
5         # We are done once we have reached the last stop.
6         if current_node.stop == last_stop:
7             break
8         for node in current_node.get_neighbors():
9             # Any parent is trivially better than no parent (in this case).
10            new_parent = not node.has_parent()
11            # Missing nodes are worse than normal nodes.
12            new_parent |= (not isinstance(MNode, current_node)
13                           and isinstance(MNode, node.parent))
14            if not new_parent:
15                continue
16            new_cost = node.cost_with_parent(current_node)
17            if not new_parent or new_cost >= node.cost:
18                # The current parent is better than the current_node could be.
19                continue
20            node.set_cost(new_cost)
21            node.set_parent(current_node)
22            # We need to update the nodes' position in the heap.
23            nodes.update(node)
24            current_node.visited = True
25            # We can recreate the route from the last node,
26            # by using its parents' parents, and so on.
27            return current_node
```

Location Interpolation of Missing Nodes

Next, we interpolate the location for each **MissingNode** using the surrounding nodes. We denote the nodes of the route with N_1, \dots, N_n with location vectors $\mathbf{L}_1, \dots, \mathbf{L}_n$, respectively, where n is the number of stops in the route. The coordinates of the location vectors are the latitude and longitude of the location.

Next, we explain how we interpolate locations for **MissingNodes** between other nodes. Assume there are consecutive **MissingNodes** N_{i+1}, \dots, N_{k-1} with $1 \leq i < k - 1$ and $i + 1 < k \leq n$, such that N_i and N_k are nodes that already have a valid location. We can interpolate the locations of each N_j with $i < j < k$ using the following formula.

$$L_j = \frac{j - i}{k - i} \cdot (\mathbf{L}_k - \mathbf{L}_i) + \mathbf{L}_i$$

Loosely speaking, this means that we place the **MissingNodes** between two nodes in equidistant steps.

We interpolate the locations for **MissingNodes** at the start in a similar way. Assume there are consecutive **MissingNodes** N_1, \dots, N_{i-1} with $1 < i < n$, where N_i and N_{i+1} are nodes that already have a valid location. We can interpolate the locations of each N_j with $1 \leq j < i$ using the following formula, instead.

$$L_j = (i - j) \cdot (\mathbf{L}_i - \mathbf{L}_{i+1}) + \mathbf{L}_i$$

For **MissingNodes** at the end, we simply reverse the order of all nodes, temporarily. Then, we can apply the same steps as for the **MissingNodes** at the start.

Final Node Selection

It may have happened that we found different nodes for a single stop, depending on the route. Thus, we need to select one of these nodes for each stop as the one we

use in the GTFS feed. For this, we simply count the number of routes that use a specific node for a stop. Then we select the node we most frequently detected for that stop.

4.1.4 Export

As the final step, we update the location of each stop. Here, we also add the additional information from OSM about the locations to the feed. In particular, we add whether the detected stop locations are wheelchair-accessible, as well as the IFOPT, if it exists. For the missing locations, we also add a note in the `stop_description`, to show that they were interpolated.

To create the GTFS feed, we simply write all GTFS files that currently exist as data structures in memory to a temporary directory. Then, archive the contents of this directory in a `.zip` file, to create the GTFS feed.

4.2 New Table Extraction

The new table extraction algorithm works in multiple stages. First, in Subsection 4.2.1, we define the data structures we use. In Subsection 4.2.2, we create new `Cells` from the words of a single page of the input file. Then, we run the basic type detection, described in Subsection 4.2.3, to determine which `Cells` contain time data and are therefore part of a tables' body. We create a preliminary `Table` from all `TimeCells` of a page (Subsection 4.2.4). If necessary, we split this `Table` and expand each resulting `Table` using the remaining `Cells` as described in Subsection 4.2.5. We run the advanced type detection on each `Cell`(4.2.6). In the last stage, we fix some common issues each `Table` may have (Subsection 4.2.7). Here, we also create a `TimeTable` from each `Table`, using only those `Cells` we could detect a proper type for.

4.2.1 Data Structures

Cell

A **Cell** represents a single non-empty table cell that contains at least one **LTChar**. As such, each **Cell** has a bounding box, defined by the bounding boxes of its **LTChars**. A **Cell** also has a **CellType** that needs to be inferred first, and neighbors, as described in Section 3.3.

We define the row of a **Cell** as the ordered list of all **Cells** that can be reached by recursively using only the **previous** and **next** neighbors. In the same way, a column starts with a **Cell** that has no neighbor **above** and ends with a **Cell** that has no neighbor **below** and can be reached using only **below**.

EmptyCell

An **EmptyCell** is a subclass of **Cell**, but does not contain any **LTChars**. The **CellType** of an **EmptyCell** is always **Empty**. When we want to check if a **Cell** overlaps horizontally with an **EmptyCell**, we use the combined bounding box of the **Cells** in the column of the **EmptyCell**. We use its row, if we want to check, for vertical overlap.

Table

A **Table** contains a variable number of **Cells**. However, instead of actually storing a reference to each one, only the first **Cell** (the one in the top-left corner) and the last **Cell** (the one in the bottom-right corner) are stored. That way, when adding more rows and columns to the table, we only need to update either of these **Cells**, as well as the neighbors of the **Cells** in the row or column.

We define the bounding box of a **Table** as the bounding box containing all of its **Cells**. We calculate it using the first and last row and column.

The **potential_cells** of a **Table** are **Cells** on the same page of the **Table**, that may be part of it. Using the **potential_cells**, we can reduce the number of **Cells** we need to check for each **Table**. For instance, if there are exactly two tables T_1 and T_2 on a page, with T_1 being above T_2 , we can add all **Cells** that are above T_2 as **potential_cells** of T_1 . Any **Cell** that is next to or below T_2 can not be part of T_1 . As the **Cells** between T_1 and T_2 might be part of either **Table**, we simply duplicate them and add them to both tables' **potential_cells**.

4.2.2 Character Extraction

Just like in the old table extraction, we preprocess the input file using *ghostscript*, and read the preprocessed PDF using *pdfminer.six*. However, instead of the **LTChars**, we use the **LTTextLines** (Section 3.4). We do this, because the basic layout analysis of *pdfminer.six* is run either way. Additionally, it yields results comparable to the word detection we use in Subsection 4.1.1.

We split the **LTTextLines** of the page into words and create a **Cell** from each word. For this, we iterate over the objects of a **LTTextLine**. We create a new **Cell**, whenever one of two conditions applies: Either the current object is a **LTAnno**, or the current and previous objects are both **LTChars** and have different font properties. Otherwise, we add the **LTChar** to the current **Cell**. The font properties we check are the used font and font size of the **LTChars**. Because we create a new **Cell** for the first **LTChar** of each **LTTextLine** a **Cell** can only contain **LTChars** that are on the same line.

At this point, we have created a **Cell** for each word-like string of characters on the page.

4.2.3 Basic Type Inference

We do not know the structure of the table yet, so we detect the type of each **Cell** based only on its text, hence ‘basic’. When we infer the type of a **Cell**, we assign probabilities for each **CellType** for that **Cell**. For example, in Figure 6, the same table as in Section 3.3 is shown. However, all possible types are shown for each **Cell**, instead of only the true types. Note that **EmptyCells** can only have **Empty** as possible type, while the **Other** type is a possible type for all **Cells** regardless of the **Cells**’ text, (except **EmptyCells**, of course).

6 1 Route	6 2 3 4 S1	6 2 3 4 S2	6 2 3 4 S3
6 2 3 4 Central	6 5 13:36	7	6 5 15:46
6 2 3 4 Station	6 5 13:38	6 5 14:33	7
6 2 3 4 Airport	6 5 13:42	6 5 14:42	6 5 16:00

1 Route Identifier	2 Route Annotation	3 Entry Annotation	
4 Stop	5 Time	6 Other	7 Empty

Figure 6: The possible celltypes of different cells in a table. The legend in the last two rows shows the cell type of the cells with the same color and number.

The basic type detection works best for **Cells** that contain text with an easy-to-check and restrictive format, like times. It also works well, if a type requires the **Cell** to have a specific text, we call keyword, like route identifier. This can also be seen in the figure. We might still get false positives; however, these are exceedingly rare (especially for the **TimeCells**). Therefore, we assume that all **Cells** that were identified as **TimeCells**, were correctly identified. Similarly, a **Cell** that contains a keyword, can either be of the corresponding type, or **Other**. We can provide the exact keywords for different types using the configuration of *pdf2gtfs*.

For **CellTypes** that have neither a strict format they adhere to, nor use specific keywords, we cannot be certain about a **Cells**’ type, using the text alone. For

instance, when encountering a `Cell` with the text ‘Central’ as in Figure 6, we can not make a reliable decision about what type it is, or whether it is even part of the table.

We run the type detection iteratively over all `Cells`, as follows. For a given `Cell`, we check if its text contains any of the keywords that directly indicate some `CellType`. If it contains any, the corresponding `CellType` will have an increased probability, while all others will be zero, and we continue with the next `Cell`. If it does not, we can be certain that it is not of any type that uses these keywords (with the exception that the correct keyword may not have been specified). In that case, we set the probability of each type that does not require a specific text to the same value. This means, that these types will all be treated equally by the advanced type detection.

At this point, we ran the basic type inference on every `Cell`.

4.2.4 Table Creation and Splitting

Now, we create a `Table` from all `TimeCells`. To do this, we first group the `TimeCells` into rows and columns. We consider two `TimeCells` to be in the same row, if they overlap more than `min_cell_overlap` vertically. Similarly, we consider two `TimeCells` to be in the same column, if they overlap more than `min_cell_overlap` horizontally. By default, `min_cell_overlap` is 0.8. That is, we check if the absolute overlap is greater than 80% of the smaller `Cells`’ size.

We update the neighbors of each `Cell`. To do that, we iterate over the `Cells` of each row and set the horizontal neighbors, `previous` and `next`. Similarly, we iterate over the `Cells` of each column and set the vertical neighbors, `above` and `below`.

Then, we iterate over the rows and columns and insert `EmptyCells`, into any gaps. We need these, to ensure some nice-to-have properties. For example, if we did not

add `EmptyCells`, the following would not hold, in general (for simplicity, we assume that each of the neighbors exist).

```
node.next.below.prev.above == node
```

That is, the `Table` is structured in a grid-like way.

To figure out, where to add the `EmptyCells`, we do the following. We first store the first `Cell` of each row in a list, which we will call `all_rows`. We iterate over both `all_rows` and the first column of the `Table`, `first_column`, to ensure that the first column has the correct number of `Cells` (= number of rows). We start at `i = j = 0` and run the following algorithm:

1. If `j >= len(all_rows)`, we are done.
2. If `first_column[i].row == all_rows[j]`, increment both `i` and `j` and at continue at 1.
3. Otherwise, insert an `EmptyCell` at `first_column[i]` and then link it to the `Cells` at `i-1` and `i+1`, if they exist. Increment `i` and continue at 1.

Next, we proceed with the same algorithm for the rest of the columns. Lastly, we iterate over all `Cells` of adjacent columns and link the respective `Cells` using the `previous` and `next` neighbors, to fix the rows. The `previous` and `next` neighbors may not be correct after we inserted the `EmptyCells`.

Once we have done this, every row contains the same number of cells, and every column contains the same number of cells. Also, at this point the `Table` already looks similar to the combined bodies of all timetables on the page.

We add the potential `RepeatIdentifierCells` and `RepeatIntervalCells` to the table. These types of `Cells` are used in timetables to define that some of the times get repeated at some interval. Basically, we search for all `RepeatIntervalCells` and `RepeatIdentifierCells` that are contained within the bounding box of the

`Table`, and add them to the `Table`. In practice, we run some additional checks to ensure that the `Cells` we found are truly of these types.

Then, we recursively split the `Table`, if necessary. To split the `Table`, we look for `Cells` that are within its bounding box, and that either overlap with a row or a column. We split the `Table`, by simply unlinking the `Cells` next to the splitting `Cells` and creating a new `Table` for each split. After the splitting, we remove any rows and columns that only contain `EmptyCells`.

Then, we set the `potential_cells` of each `Table`. These are those `Cells` that may be part of the `Table`, but we do not know yet if they are. For example, given two `Tables` next to each other, we can be sure, that the left table only contains `Cells` that are either above, below or left of it, or are between the two `Tables`. That is, if a `Cell` is right of the right `Table`, it can not be part of the left `Table`. `Cells` that are between two `Tables` are simply duplicated and added to both `Tables`' potential `Cells`.

At this point, we have a list of `Tables`, where each `Table` contains `TimeCells`, and possibly `RepeatIdentifierCells` and `RepeatIntervalCells`, and has a list of `Cells` that might be part of it. If we were to print any of these `Tables`, it would look exactly like the body of one of the timetables, as long as the type inference worked properly. Except, that it would contain only times and repeat cells.

4.2.5 Table Expansion

We now add more rows and columns to each `Table`, using its `potential_cells`. We call this process table expansion. By default, we only expand the `Tables` in the directions `West` and `North`, because it is unusual for a timetable to contain significant amounts of information in the other directions. However, if a specific timetable contains additional information in the other directions, we can use the configuration

of *pdf2gtfs* to change this. The expansion works similar in all directions. Hence, we will only explain it in one direction: **West**.

To expand the table in direction **West**, we select those **Cells** of the **potential_cells**, that are adjacent to the last column in that direction, that is, the first column of the **Table**. For instance, a **Cell** c of **potential_cells** C is adjacent to a **Table** T in direction **West**, if all of the following conditions apply. We denote R as the reference **Cells**, in this case the first column of T .

- c is **West** of T , that is, the x_1 -coordinate of c is less than or equal to the x_0 -coordinate of the right-most **Cell** of R .
- c is overlapping vertically with any row of T by more than `min_cell_overlap`.
- c is either the right-most **Cell** of the adjacent **Cells** of C , or it overlaps horizontally with the right-most adjacent **Cell**.

We link the adjacent **Cells** using the neighbors **above** and **below**. We insert **EmptyCells**, to ensure the numbers of **Cells** is equal to the number of rows of the **Table**, just like we did in Subsection 4.2.4. Finally, we link the adjacent **Cells** to the first column of the **Table** and remove them from the **potential_cells**.

We expand the **Table** in the directions **West** until no **Cells** of the **potential_cells** can be added to the **Table**.

At this point, we expand each **Table** maximally in the directions **North** and **West**, such that each **Table** either has no more **potential_cells**, or no **Cell** of the **potential_cells** can be added to the **Tables'** rows or columns in this way.

4.2.6 Advanced Type Detection

Once a **Table** was fully expanded, we can run the advanced type detection, which uses other **Cells** of the **Table** to improve the type detection accuracy. With the basic type detection, we estimated the probabilities of the potential types of a **Cell**.

We now apply weights to each of these probabilities, based on other **Cells**. The weight represents the confidence we have, that a **Cell** is a specific type. We run this advanced type detection on all **Cells**. For many **Cells** this weight will simply be zero for all but one type (or two if we count **Other**).

<div>6</div> <div>1</div> Route	<div>6</div> <div>2</div> <div>3</div> <div>4</div> S1	<div>6</div> <div>2</div> <div>3</div> <div>4</div> S2	<div>6</div> <div>2</div> <div>3</div> <div>4</div> S3
<div>6</div> <div>2</div> <div>3</div> <div>4</div> Central	<div>6</div> <div>5</div> 13:36	<div>7</div>	<div>6</div> <div>5</div> 15:46
<div>6</div> <div>2</div> <div>3</div> <div>4</div> Station	<div>6</div> <div>5</div> 13:38	<div>6</div> <div>5</div> 14:33	<div>7</div>
<div>6</div> <div>2</div> <div>3</div> <div>4</div> Airport	<div>6</div> <div>5</div> 13:42	<div>6</div> <div>5</div> 14:42	<div>6</div> <div>5</div> 16:00

<div>1</div> Route Identifier	<div>2</div> Route Annotation	<div>3</div> Entry Annotation	
<div>4</div> Stop	<div>5</div> Time	<div>6</div> Other	<div>7</div> Empty

Figure 7: The possible celltypes of different cells in a table. The legend in the last two rows shows the cell type of the cells with the same color and number.

Figure 7 shows our previous example, after the basic type detection was run. We will explain how the advanced type detection works using the two light-blue colored **Cells**. First, we look at the **Cell** with the text “Station”. The basic type detection was unable to provide a specific type for that **Cell**. Thus, for each of the 3 types (we do not do this for **Other**), we need to calculate the additional weights. There is no possible **EntryIdentifierCell** in the row or column of the **Cell**. Thus, we multiply the probability that this **Cell** is an **EntryAnnotationCell** with zero. This is, because we are not confident at all, that the type of the **Cell** is an **EntryAnnotationCell**. On the other hand, there exists a possible **RouteIdentifierCell** in the column of the **Cell**, as well as **TimeCells** in the row of the **Cell**. At the same time, all **Cells** in the **Cells**’ column that are in **TimeCell**-rows, are possibly **StopCells**. Therefore, we have a higher confidence that the **Cell** is one of these two types. In the end, the deciding factor, in this case, is the text of the **Cell**. For the **StopCells**, we increase the weight depending on its texts’ length and the texts’ relative amount of numbers versus letters. That is, the text of a **StopCell** should not be too short

and should not contain too many numbers. On the other hand, we expect the `RouteAnnotationCells` to generally have shorter text, and they might even consist of only numbers. Consequently, our confidence that this specific `Cell` is a `StopCell` increases, while our confidence that the `Cell` is a `RouteAnnotationCell` decreases. If we run the advanced type detection on the “S2” `Cell`, the opposite happens. The text is short and 50% of its characters are numbers. Hence, it is more likely a `RouteAnnotationCell` than a `StopCell`.

4.2.7 Cleanup and TimeTable Creation

As the final step of the `Table` creation, we need to fix some potential issues our approach has. For example, each `Cell` contains only a single word. This means, that stops that contain spaces are not properly detected as one `Cell`, but as multiple. Therefore, we need to merge these `StopCells`. If the type detection was successful, this simply means that we need to merge consecutive `StopCells` of the same row (though in practice, we only merge the text). Otherwise, the table extraction, or at least the location detection will likely fail.

Similarly, consecutive `DayCells` of the same row also need to be merged, sometimes. Here as well, the type detection already did most of the heavy lifting. When multiple adjacent `Cells` in the same row are part of the same days keyword, this is detected by the advanced type detection, and it infers the `CellType` “Days” for them. For instance, suppose there is a days keyword “Monday – Friday” and we have a `Cell` C_1 with the text “–”. If C_1 has a previous neighbor C_0 with text “Monday” and a next neighbor C_2 with text “Friday”, we merge the text of C_1 and C_2 with the one of C_0 . This approach prevents us from accidentally merging two `DayCells` with the texts “Saturday” and “Sunday”, respectively, if only the individual keywords “Saturday” and “Sunday” were specified.

As the final step of the table extraction, we transform each `Table` into a `TimeTable`.

This is done very similarly to how the old table extraction algorithm does it. The `Cells` are added to the `TimeTable` based on their type. Any `Cells` that do not have a proper type (i.e., `OtherCells` and `EmptyCells`) are skipped. Thus, even if a `Cell` was added to the `Table`, if we could not correctly infer its type, we will not add the contained information to the output. This is to safeguard our overly-greedy table expansion against the (incidental) alignment of irrelevant text to the `Table`.

To create a `TimeTable` from a `Table`, we first search the `Table` for the `StopCells` and create the list of stops from them. If there are multiple rows or columns with `StopCells`, we only use the first one we encounter to create the stops. Assuming the `StopCells` are in a column, we create a `TimeTableEntry` for each column of the `Table`. We iterate over the columns of the `Table` and add each `Cell` with a proper type, to the respective `TimeTableEntry`. We add the stops and the `TimeTableEntries` to the `TimeTable`. If the `StopCells` are in a row instead, we iterate over the `Cells` of the rows, instead.

At this point, the new table extraction is finished and *pdf2gtfs* will continue with the GTFS creation (Subsection 4.1.2).

4.3 Evaluation Tool *p2g-eval*

To evaluate the locations detected by *pdf2gtfs*, we created *p2g-eval*. This tool reads two GTFS feeds, the first being the ground truth or true feed, while the second is the test feed; the one that should be evaluated. The true feed we use, is usually provided by the different transit agencies. We create the test feed using *pdf2gtfs*. We use *p2g-eval* to calculate the distance between the locations of the stops in the ground truth and the corresponding locations in the test feed.

Because neither the name of a specific stop nor its ID can be assumed to be equal in both feeds, we need to map the feeds, somehow. We currently only provide a

manual approach to this. That is, the user has to provide a `.csv`-file, where each line contains the IDs of both the true feed and the test feed for one stop.

The evaluation of the test feed is straight forward. We use *pandas* [27], to read the `.txt` files of each feed, which stores each in a `DataFrame`. Then, we filter the `DataFrames` using the mapping. We iterate over the stops based on the mapping, and calculate the distance between the stop location of the true feed and that of the test feed. The output is printed in human-readable form, where the lowest, highest and mean distances, as well as the standard deviation are printed, as well.

The `batched` script of *p2g-eval*, allows us to supply one feed and a directory. The script will run *p2g-eval* to evaluate all feeds in the directory. It still requires each mapping, which must have the same name as the respective test feed. Compared to the normal operation, we output the distances of all feeds to a single file, instead of printing them, which makes evaluation of multiple feeds easier.

5 Experiments

In Section 5.1, we evaluated the accuracy of the locations detected by *pdf2gtfs*. We evaluated the accuracy of the table extraction in Section 5.2. Finally, we evaluated the runtime- and memory-requirements of *pdf2gtfs* in Section 5.3.

5.1 Location Detection Evaluation

We evaluated the location detection of *pdf2gtfs* on different timetable PDFs using the ground truth data, which is provided by their respective transit agencies. We first created datasets in Subsection 5.1.1. Here, we also created the mapping between the stops of the GTFS feeds. In Subsection 5.1.2 we show the results of the evaluation.

5.1.1 Dataset Preparation

We created datasets for three different transit agencies from Germany:

VAG the “Verkehrs AG Freiburg”, the transit agency in Freiburg im Breisgau.

RMV the “Rhein-Main-Verkehrsverbund”, the transit agency around Frankfurt; its service is covering most of Hesse

VGN the “Verkehrsverbund Großraum Nürnberg GmbH”, the transit agency in and around Nuremberg

The RMV did not provide the full GTFS feed of its transit data on its website. It did however, provide a `.csv`-file containing information about its stops [28]. Thus, we created a GTFS feed using only the stop names and locations of the stops, and used this for the evaluation. Though this feed is not valid, as per the specification, it contains all data required by *p2g-eval*. To create the feed, we had to change the latitude and longitude strings in the `.txt` file, because it used commas as floating-point delimiter. An alternative solution to the `.csv`-file would have been, to use the API provided by the RMV. We opted for our approach instead, because we designed *p2g-eval* to work with two input GTFS feeds. The other two agencies provide the full GTFS feed on their respective websites [29, 30].

The locations provided by the RMV only contained a single location for each stop. The feeds provided by the VAG and VGN contained multiple locations for each stop. The additional stop locations are used for the different stop positions and platforms of the stop. This makes a difference for large stops such as a cities' central station. Depending on the size of the station, the different stops may have up to a few hundred meters between them.

Due to the single locations provided by the RMV, we decided to use the location of the station of a stop for the other datasets, as well. If no parent station, but multiple locations existed for a stop we simply selected the first one. That way, we expected we would be able to compare the results of the datasets in a fairer manner. Note that *pdf2gtfs* uses the type of the location (e.g., station or stop position) to modify the node cost. However, because the stop position is preferred to the station, the impact of the decision to use the stations, should even out.

We selected the timetables for the evaluation based on whether we could detect the timetables using *pdf2gtfs*, and whether there was at least one location detected. We did not include the other cases, because we evaluated the timetable extraction separately.

As stated in Section 4.3, a mapping between the two feeds is required to use *p2g-eval*.

We created this mapping manually. We searched the ground-truth feed (or, true feed), provided by the transit agency, for the stop name of each stop in the test feed; the one created by *pdf2gtfs*. We used the new table detection algorithm to extract the timetables. Next, we simply created the mapping: A *.csv*-file that contains the *stop_id* of the true feed in the first column and the ones of the test feed in the second. We did not try to create this mapping automatically the same way we do manually, using the stop names. That would have been an almost identical problem to the one that we are trying to solve using *pdf2gtfs*: Finding the locations of the stops based solely on their names and order.

5.1.2 Results

Once we had created the mappings for each feed generated by *pdf2gtfs*, we simply ran *p2g-eval* using the mapping, the true feed, and the test feed. We list the minimum, maximum, and mean distance, as well as the standard deviation, of each detected stop location to its true location in Table 3. From these results, we can infer that the location accuracy for detected locations was very good. On the other hand, the results for the accuracy of missing locations, that is, those that have been interpolated, were mixed.

	VAG			VGN			RMV		
	both	detected	missing	both	det.	miss.	both	det.	miss.
count	100	98	2	61	40	21	27	18	9
min	2	2	129	4	5	107	6	6	40
max	175	123	175	87 317	260	87 317	1 012	83	1 012
mean	34	32	152	3 743	44	10 788	231	39	616
std	30	25	32	14 043	49	22 630	319	24	282

Table 3: The min, max, mean, and standard deviation for detected and missing locations for all datasets. All distances in meters and rounded to the nearest integer.

The bar chart in Figure 8 shows the number of stops that were closer than the specified distance over all datasets. The results show that if a position was found, it was within 250m of the actual location, in most cases. On the other hand, the interpolated locations were generally farther than 500 meters away. This is most likely due to how simple the interpolation is done.

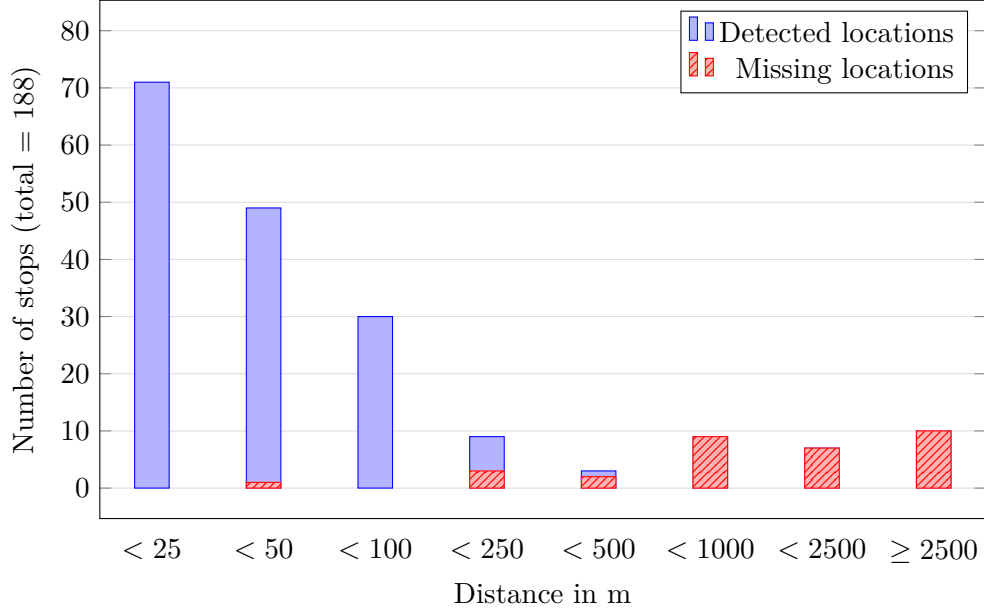


Figure 8: The number of detected and missing stops closer to the ground truth than the specified distance.

5.2 Table Extraction Evaluation

We evaluated both the new and the old extraction algorithms of *pdf2gtfs* against each other. We also evaluated them against the output of the online pdf-extraction service *PDFTables* [8]. For this, we first prepared three different datasets in Subsection 5.2.1. These contain a diverse set of PDFs that we used for the evaluation. We describe how we set up each of the tools for the different datasets in Subsection 5.2.2. We explain the metrics we used for the evaluation in Subsection 5.2.3. Finally, in Subsection 5.2.4, we compared the results for each extracted timetable.

5.2.1 Dataset Preparation

To our knowledge, no dataset exists that contains only PDF timetables. Therefore, we created a ground truth using a sample of timetables provided by the VAG and RMV. We also evaluated the quality of the table extraction of transposed tables. That is, these tables contain the stops in a single row, as opposed to a single column. For this, we used timetables from different transit agencies in the USA, where this format is more frequently used.

We created the ground truth manually, by copy-pasting a timetables' content into LibreOffice Calc, the spreadsheet component of LibreOffice. Then, we saved the extracted timetable data in the `.csv`-format. The copy-pasting proved more difficult than initially imagined. When we selected and copied text from the table, it was difficult to predict how it would be pasted. At times, we had to copy-paste single cell-values, as the order of cells was mangled up. We also used the macro functionality of the text-editor Vim, to speed up repetitive tasks; to quickly join consecutive lines, or to replace every second space by a comma, in order to make use of LibreOffices' text import dialog. In hindsight, if the copy-pasting had worked seamlessly, table extraction from PDFs would not be such an issue.

The way we selected the timetables for the VAG and RMV datasets was done as follows. We looked at the PDF files, or schedules, of the different transit agencies and tried to select a diverse set of different timetable formats. We considered two timetable formats different, if not only the content of their cells was different, but also the overall structure of the timetables. Then, for each of the PDF files, we selected two timetables to use for the evaluation, based on their features. For instance, we generally selected the table with the most cell types.

For the VAG and the rail-based public transport of the RMV, we simply looked through all available schedules. Because of the sheer volume of bus-line schedules of the RMV—they list almost 800 bus-line schedules on their website—we did not look

at all of these schedules, individually. Instead, we used the name of the bus lines to narrow down candidates that are most likely to use different timetable formats. For instance, given three bus lines with the names ‘1’, ‘10’, and ‘MR-97’, respectively, the latter is most likely to use a different format.

We call the additional dataset that we use for the evaluation of transposed timetables ‘Transposed Time Tables’, or ‘TTT’, for short. The overall selection process of the timetables was the same, with the difference that we selected only one timetable per PDF file. As these are seldom used in Germany and more common in the USA, this dataset contains timetables exclusively from US-based transit-agencies. One of the PDF files we initially selected for this dataset, was password-protected, so we removed it from the dataset.

In total, we selected ten timetables for the VAG dataset, six timetables for the RMV dataset, and four timetables for the TTT dataset. Theoretically, we could have also used *pdf2gtfs* to generate more data for us, which we would only have needed to check for errors. We decided against this, as this would have made it harder to catch errors made by all three extraction methods. Also, it would have been difficult not to make decisions based on the output, rather than the input. What we mean by this can be best explained using the example in Figure 9.






	 Ballston-MU Metro	 Clarendon Metro	Sequoia DHS/2nd St. S	Columbia Pike & Orme	 Pentagon Metro
Bus 42	6:00 A	6:08 A	6:14 A	6:20 A	6:30 A
42	6:15 A	6:23 A	6:29 A	6:35 A	6:45 A
42	6:30 A	6:38 A	6:44 A	6:50 A	7:00 A
42	6:45 A	6:53 A	6:59 A	7:05 A	7:15 A
42	7:00 A	7:08 A	7:14 A	7:20 A	7:30 A
42	7:15 A	7:23 A	7:29 A	7:35 A	7:45 A

Figure 9: An excerpt of the ‘ART 42’ route from the TTT dataset.

The ‘A’s and ‘P’s denote, whether the preceding time is AM or PM. The new extraction algorithm of *pdf2gtfs* would have (and did) output the ‘A’s and ‘P’s as separate cells from the times. While this is a possible interpretation of cells, it is more intuitive, as well as closer to reality, to have cells that contain both the time and the ‘A’ or ‘P’.

5.2.2 Configuration

We mostly used the default *pdf2gtfs* configuration for all PDFs. We only set the time format (e.g., "HH:MM" or "HH.MM") and other ‘obvious’ settings. For example, one timetable used the word ‘Bus’ as keyword for the route identifier. Thus, we added the keyword to the corresponding config-key. We did so, because this setting would be commonly used during normal operation of *pdf2gtfs*. Only when a table was not detected at all, did we change the more intricate settings. This occurred once. The old algorithm was initially unable to extract either of the two tables from the bus-line ‘N46’ of the VAG dataset. We inspected the log and found that the distance between the rows was too high. Thus, we increased the `max_row_distance`, to enable extraction. Apart from these adjustments, we did not change other settings. Our reasoning was, that we can evaluate the quality of the table extraction based on (basically) the default settings. Finding and evaluating on the optimal settings would not only have been time-consuming, it would also not reflect the expected real-world outcome.

There was no option to configure *PDFTables* in any way.

We ran all table extraction methods on each of the PDFs we selected. We exported the extracted table(s) in the `.csv`-format.

5.2.3 Evaluation Measures

We calculated the precision, recall and the resulting F_1 -score. These are common measures to evaluate binary classifiers. We can understand the table detection as a classification problem, where we decide whether a cell is part of the table or not. This means, we did not directly evaluate the exact cell type, which also would have made the evaluation with *PDFTables* difficult.

To use the above-mentioned measures, we first had to define the following disjoint sets of table cells.

True Positive (TP)

All cells with the correct content, and position relative to other cells.

False Positive (FP)

All extracted cells that do not exist in the ground truth, and extracted cells with different content to the corresponding cell in the ground truth.

True Negative (TN)

All empty cells that were correctly detected.

False Negative (FN)

All cells that exist in the ground truth, but were not extracted.

The precision describes the relative amount of relevant cells that were extracted.

$$P = \frac{TP}{TP + FP}$$

The recall describes the relative amount of correct cells of all extracted cells.

$$R = \frac{TP}{TP + FN}$$

The F_1 -score can be used to balance the other two measures, in a way that both are

relevant. It is the harmonic mean between the two and is defined as:

$$F_1 = \frac{2 \cdot P \cdot R}{P + R}$$

The F_1 -score is a more meaningful measure, because both the precision and the recall can be increased in a (trivial) manner. The recall can be trivially increased by simply extracting all cells. On the other hand, we can easily increase the precision by extracting only the cell with the highest probability of being part of the table. For instance, if we only extracted all **TimeCells**, the precision would be 1, provided that we correctly extract those. Note that maximizing either of these measures is, generally, more difficult, because the **Cells** need to have the correct content as well, to count as true positives. The F_1 -score is maximized only, if both of the other measures are maximized. One of the down-sides is that the F_1 -score ignores the true negatives.

Because *PDFTables* was unable to split the page into multiple tables, we assumed the best case. That is, we removed any text that was not between the top and bottom rows of each table. Otherwise, we felt a comparison between *PDFTables* and the other methods would not be meaningful in any way.

We evaluated rows and columns that only contained empty cells favorably. That is, when the ground truth did not contain an empty row or column, we only counted the true negatives, if there were any. Otherwise, we evaluated the cells as we normally would. The aforementioned manual selection of rows for *PDFTables* was the main reason we did this, though the algorithms of *pdf2gtfs* sometimes had this issue, as well. We also ignored leading and trailing whitespace of a cell. For example, we consider two cells with the texts ‘Bahnhof ’, and ‘Bahnhof’, to have equal content. However, we do not consider ‘Am Bahnhof’ and ‘AmBahnhof’ to have equal content. This whitespace has no impact on the meaning of a cell and can be safely removed.

We also manually disabled some functionality of *pdf2gtfs* that alters the extracted

table in preparation of the location detection. Specifically, we skipped the fixing of split stop names, described in Subsection 3.3.1. By default, we do this in the `Table` or `PDFTable` as opposed to the `TimeTable`, where it would technically fit better, because we need the bounding boxes of the cell. We use the bounding boxes to determine if a stop cell is indented compared to the other stop cells in its column. It is ok to disable these, because it only alters the cells content based on cells that were already detected and does not change the overall detection process.

5.2.4 Results

Table 4 shows the results of the evaluation for the VAG and RMV datasets. On the VAG dataset, *PDFTables* performed significantly worse than the other two extraction methods. On the RMV dataset, the results of *PDFTables* are more comparable to the other two.

VAG	Precision	Recall	F1-score
pdftables	86.84%	57.63%	69.28%
pdf2gtfs-old	99.83%	88.84%	94.01%
pdf2gtfs-new	93.40%	97.78%	95.54%
RMV	Precision	Recall	F1-score
pdftables	94.03%	85.34%	89.78%
pdf2gtfs-old	98.82%	95.94%	97.36%
pdf2gtfs-new	98.97%	91.05%	94.84%

Table 4: The precision recall and F1-score for the different datasets and table-extraction methods

The biggest problem that occurred with *PDFTables* was that it often merged cells in the same row or column using whitespace (either newline or space). Because *PDFTables* is proprietary, we do not know how its table extraction works. However,

similar problems occurred, when we created the ground truth, or when attempting to use *pdfminer.six*'s advanced layout algorithm. Though the exact reason is unclear to us, it seemed this happened more frequently when columns or rows were close to each other.

Comparing the old extraction of *pdf2gtfs* to the new one in a definitive manner is difficult, from these results alone. Overall, their performance can be considered similar for these datasets.

When we manually compared what kind of cells were detected we noticed that no `TimeCell` was falsely detected by the new method, while the old method missed a few `TimeCells`. Note that we only did this for *pdf2gtfs*'s extraction methods and the VAG and RMV datasets. This was somewhat expected, because the algorithm works by using the `TimeCells` as base. However, it shows that the splitting of the `LTTextLines` works as intended.

As stated in 5.2.2, we had to adjust the maximum distance between lines for the old algorithm, in order to extract the tables of the bus-line 'N46' of the VAG dataset.

In Table 5 the precision, recall, and F_1 -score of the extraction methods using the TTT dataset are displayed. The results are clearly worse than the ones for the VAG or RMV datasets. This is especially true for the old algorithm. The high recall and low precision of the new extraction algorithm of *pdf2gtfs*, suggests that too few cells were extracted. That is, the algorithm appears too restrictive when deciding, which cells to extract. However, this dataset included the table, where the times could not be extracted at all.

TTV	Precision	Recall	F1-score
pdftables	61.36%	43.12%	50.65%
pdf2gtfs-old	22.87%	8.48%	12.37%
pdf2gtfs-new	49.83%	96.76%	65.79%

Table 5: The precision recall and F1-score for the different datasets and table-extraction methods

We calculated the mean and weighted mean of the precision, recall and F_1 -score over all datasets. We weighted each of the measures using the number of tables in the respective dataset. The results, shown in Table 6, suggest that the old extraction algorithm of *pdf2gtfs* and *PDFTables* have similar accuracies. However, the low F_1 -score of the old algorithm appears to mostly be a coming from the low results of the TTT dataset. Overall, the new extraction algorithm of *pdf2gtfs* performed best. What can be observed, is that the precision of all extraction methods is almost equal. At the same time, the recall of the new extraction algorithm is significantly higher than for the other extraction methods. Therefore, we can conclude that the new extraction method seems to extract fewer incorrect cells, while keeping an equally high level of precision.

mean	Precision	Recall	F1-score
pdftables	80.74%	62.03%	70.16%
pdf2gtfs-old	73.84%	64.42%	68.81%
pdf2gtfs-new	80.74%	95.20%	87.37%
weighted mean	Precision	Recall	F1-score
pdftables	83.90%	63.04%	71.99%
pdf2gtfs-old	84.14%	74.90%	79.25%
pdf2gtfs-new	86.36%	95.56%	90.73%

Table 6: The mean and weighted (by table-count) mean of the precision recall and F1-score for all datasets and table extraction methods

One final observation: For many of the input files, *pdf2gtfs* failed to generate GTFS feeds. This was the case, regardless of which table extraction algorithm we used, and happened, because *pdf2gtfs* failed to convert the detected tables into **TimeTables**. Currently, if a table does not have a valid DaysCell, no **TimeTable** can be created from it, because we need the days to create the **GTFSHandler**.

5.3 Performance

In this section, we will estimate the performance of individual parts of *pdf2gtfs*, as well as the performance, in general.

Overall, *PDFTables* was the fastest at extracting the tables, closely followed by the old table extraction algorithm of *pdf2gtfs*. The new table extraction algorithm was comparably fast, but still the slowest of the three. We will look at the performance of the new table extraction in the case of a single table T . In the following R denotes the total number of rows in T , C denotes the total number of columns in T , and $N = RC$ denotes the total number of cells in T .

For the table expansion, the worst-possible runtime case would occur, for example, when there is a single time cell and all other cells are in the same row of the time cell. Further, every cell is in its own column. Then, in each expansion step we have to check one less node for adjacency than in the previous step. This can be calculated using the Gaussian Sum Formula, $0.5 \cdot (N^2 + N)$. Finding the neighbor of a cell can be done in $\mathcal{O}(1)$, because the direct neighbor is stored for every cell. For each expansion step, we need to check for overlaps, link the adjacent cells and insert empty cells. This can be done in linear time as it only depends on the number of adjacent cells. As such, the whole expansion can be done in $\mathcal{O}(N^2)$.

Next, we will estimate the runtime of the basic type inference. We need to check the content of every cell and compare it to the keywords for the different times. These checks can be done in $\mathcal{O}(k)$ for a single cell, where k is the number of keywords, because each cell only contains one word. Thus, because usually $k \ll N$, the overall basic type inference takes $\mathcal{O}(N)$.

The complexity of the advanced type inference is more difficult to evaluate. Here, every cell-type check has a different complexity, and we only check the possible types detected by the basic type inference. Hence, we will simply assume that we need to check each cell type for each cell. The cell-type check with the highest complexity is the one for the stop. When checking if a cell is a stop, we need to check the types of the cells' row and column. This can be done in $\mathcal{O}(C + R)$. The other checks, like the one to find the relative amount of letters and numbers in the cells' text, can be done in $\mathcal{O}(k)$, where k is the length of the cells' text. As with the keyword check from before, because generally $k \ll N$, this can be done in $\mathcal{O}(1)$. Thus, the complexity of the advanced type inference for a single cell is $\mathcal{O}(N(C + R))$. In general, $R \sim C$; that is, the R and C are similar. It follows that $R + C \approx R + R = 2R \ll R^2 \approx R \cdot C = N$ for large R . Hence, we can simplify the complexity to $\mathcal{O}(N)$. We need to run the advanced type inference on all cells, which results in $\mathcal{O}(N^2)$ to run the advanced type inference on all cells of T .

During the cleanup step of each table, we retrieve all cells of a table with a type. Given that we do not store the cells by their type in any way, this can only be done in $\mathcal{O}(N)$. We need to iterate over all cells, in order to find out which cell has that type.

Overall, the table extraction takes at most $\mathcal{O}(N^2)$.

In general, Dijkstra’s algorithm that we use for the location detection, takes in the worst case, $\mathcal{O}(E + V \cdot \log V)$, where, E is the number of edges and V is the number of vertices, [31]. Because our implementation does not use a Fibonacci heap as is used in the cited source, the complexity of our implementation can be expected to be worse, though it should still be comparable.

However, the general runtime should be considerably better for our graph. This stems from the fact that the nodes of our graph are sparsely connected. That is, there are comparably few nodes for each stop and each node of a stop is only connected to the nodes of the next stop. The worst case occurs, when multiple consecutive stops have names that are widely-used terms, like “Bahnhof” (= “station”). In this case, the overall runtime is considerably and noticeably worse.

We did not see much of a difference in memory usage between the two table extraction algorithms of *pdf2gtfs*. Generally it can be said, that the largest impact on the memory usage was the Python interpreter itself and the reading of the location cache into memory. On the other hand, when the cache needed to be rebuilt, the memory usage roughly doubled to 500 MB. This is most probably because of the normalization of the stop names.

We cannot compare the performance of *PDFTables* qualitatively with that of *pdf2gtfs*, because *PDFTables* is proprietary. We also have no knowledge about the system the web service is running on. However, *PDFTables* was faster than *pdf2gtfs* when extracting the tables. This includes the time it took to upload the PDF.

6 Conclusion

The results suggest that the new table extraction algorithm is performing better than the old one. However, more work needs to be done to ensure it works in all cases at least as good as the old one does.

What we noticed, is that the overall robustness and error-responsiveness of *pdf2gtfs* has room for improvement. Many times it was not immediately clear, why a specific table could not be read. Another problem that occurred was that both table extraction algorithms were unable to detect times that contain spaces; these times were recognized as two different cells. This only occurred on the TTT dataset, and results from the fact that they use a 12-hour clock in the US (e.g., ‘13:42 A’).

Until the aforementioned issues have been addressed, we should hold off the removal of the previous table detection. Instead, we suggest to use the new algorithm by default, while improving it, and the old one as a fallback, in case the detection fails, when using the new algorithm.

Compared to *PDFTables*, the results of the new table extraction algorithm can be considered very good. This is expected, because we built *pdf2gtfs* specifically for detecting timetables, while *PDFTables* is able to detect tables in PDFs, in general. At the same time, the errors made by the new extraction algorithm occurred only in the cells surrounding a tables’ body, except for the case with split times. These errors are relatively easy to recover from. Compared to that, the errors made by *PDFTables* were a lot more grave, for example the merging of multiple rows or columns.

The location detection worked well, when a location was found. If a location was detected for a stop, in most cases it was within 100 meters of the true location, while it was never farther than 500 meters away, at least for our datasets. Here, the biggest difficulty was detecting the correct stop position for a given stop. In rare cases, the wrong OSM-node was selected, when it had a similar name and was close to the true node location. For missing locations, the results are spread out far more. In some cases the interpolated locations had similar accuracy to the detected locations. However, most interpolated missing locations were within 1000 to 5000 meters from the real location. In rare cases, they were too far away to be helpful.

One problem that occurred was that sometimes the table detection did not recognize connections as such. Connections are generally farther away than other stops. Thus, they were not properly detected, which resulted in more missing locations. In fact, the highest distances between a missing location and the true location were from these connections.

In consequence, we can consider the overall results of *pdf2gtfs* good. At the same time, there are many improvements we can implement, to improve the overall usability, as well as the accuracy of the table extraction and location detection.

7 Future work

In this chapter, we will provide some ways in which could change *pdf2gtfs* and *p2g-eval*, in order to ensure more accurate results. In Section 7.1 we discuss how the table extraction and the location detection of *pdf2gtfs* can be improved. We include some ideas on how we could develop *p2g-eval* to increase both its usability and usefulness in Section 7.2.

7.1 Improvements for pdf2gtfs

We can improve the location detection in two ways. First, we can reduce the number of missing locations, that is, stops where no location was found. And second, we can decrease the distance to the true locations.

To begin with, the main reason locations were not detected was that the stop names used in OSM and in the PDF differ. Otherwise, we would have detected the wrong location for a stop, instead of no location, at all. One simple solution then, would be to search a smaller area (< 10 km) around interpolated locations. That way, we could use more expensive comparisons for the names, without impacting the performance too much. On the other hand, we could also change the way the nodes of a specific stop are searched for, in general. For example, we could implement a fuzzy search, which should be less susceptible to differences in the names than our current approach. Finally, instead of using QLever, we could allow using a `.csv` file containing the stop names and locations, instead. This could improve the results, if

the coverage of OSM is suboptimal, or if there is a special naming-scheme of the stops that is not used by OSM. Overall, these are all comparably simple-to-implement, or at least straightforward, solutions that should significantly improve the overall location detection accuracy.

The other way to improve the location detection is to decrease the distance of detected stop locations to the true stop location. In theory, we could solve this, by using additional programs to improve the location detection. For example, we could use a tool like *pfaedle* [32] or *TransitRouter* [33]. These tools search for the actual path a public transport vehicle takes on the map. We could do a rough location matching using *pdf2gtfs* first, where we get multiple locations for each stop. Then, we could use either of these tools to generate the shape for different combinations of locations. Finally, we would select the locations of the route shape that is best, by some measure. That measure could be, for example, the length of the route or its “curviness”. The effectiveness of this approach depends on the measure we use. Overall, this is more difficult to implement, as we need to completely change the way we do the location detection.

One way to improve the table detection is to use the graphical hints provided, such as lines or background color. This would allow us to use the work by Nurminen [10] as a starting point. We could also use these hints to only complement the existing table detection, for example, to only detect the outer bounds of the table. On the other hand, if the detected table has only minor issues, like a typo or a missing cell, we could also provide the user interface with an option to manually edit detected tables. This could be done in a way that allows the user to edit, add, or remove cells; merge and split tables; or change the type of a cell. The downside with this approach is that it relies on manual intervention. Both of these improvements are non-trivial to implement, so it might be worthwhile, to look for ways to improve the type detection and table expansion, first.

As stated in Section 6, the error-responsiveness of *pdf2gtfs* can be improved. This

should at least include the error handling, to show more descriptive error messages, instead of simply failing the table extraction or the location detection. At the same time, we could change the `TimeTables` and `GTFSHandler` in a way that they work even if some cells were not detected. For example, if the days on which the service occurs were not extracted, we could have the user input the missing values, instead. On the other hand, we could also simply add a single service in the `calendar.txt` for each route. While not exactly space-efficient, the GTFS allows this, and it would enable the user to supply the correct service days.

7.2 Improvements for *p2g-eval*

The current bottleneck of *p2g-eval* is the need to create the mapping between the feeds. If we could map the feeds automatically, we could evaluate *pdf2gtfs* using more feeds. The automatic mapping could be done using the stop times. This would only work, if both feeds contain at least some of the same stop times. However, unless the stops are given as a `.csv` file instead of a GTFS feed, as was the case with the RMV dataset we used, that should always be the case.

We could also evaluate the `stop_times.txt`, as well as the other parts of the GTFS feed. This was not possible, due to time constraints.

Another possibility, is to use *p2g-eval* during development of *pdf2gtfs*, as part of an automated test-suite. This could ensure that regressions in the location detection become apparent immediately. However, extra care needs to be taken to prevent overfitting.

8 Acknowledgments

I would like to thank Patrick Brosi, for getting me into the rabbit hole that is public transit data and PDF extraction. Additionally, he gave me pointers in the right direction, whenever I was stuck, and also proofread this thesis.

I would like to thank Prof. Dr. Hannah Bast, for being a great example of a person I can strive to be like.

Next, I would like to thank my parents, for their support throughout my studies, regardless of my non-existent time-management skills.

Many thanks to my two sisters, who helped me immensely by proofreading this thesis.

Finally, I would like to thank the rest of my family and friends, whom I can count on to be there for me whenever I need them.

Bibliography

- [1] (2023) VAG Freiburg. Accessed: 2023-05-05. [Online].
Available: <https://www.vag-freiburg.de/>
- [2] (2023) RMV Service Timetables. Accessed: 2023-06-29. [Online].
Available: <https://www.rmv.de/c/de/fahrplan/fahrplaene/linienfahrplaene/fahrplantabellen>
- [3] (2023) pdfminer.six - converting a pdf file to text. [Online].
Available: https://pdfminersix.readthedocs.io/en/latest/topic/converting_pdf_to_text.html
- [4] J. Heinzinger. (2023) pdf2gtfs. [Online].
Available: <https://github.com/heijul/pdf2gtfs>
- [5] (2023) OpenStreetMap. Accessed: 2023-05-05. [Online].
Available: <https://www.openstreetmap.org>
- [6] (2023) QLever. Accessed: 2023-06-04. [Online].
Available: <https://github.com/ad-freiburg/qllever>
- [7] GTFS reference. Accessed: 2023-05-07. [Online].
Available: <https://developers.google.com/transit/gtfs/reference>
- [8] (2023) PDFTables. Accessed: 2023-06-29. [Online].
Available: <https://pdftables.com/>

- [9] J. Heinzinger. (2023) p2g-eval. [Online].
Available: <https://github.com/heijul/p2g-eval>
- [10] A. Nurminen, “Algorithmic Extraction of Data in Tables in PDF Documents,” Master’s thesis, Tampere University of Technology, Apr. 2013, accessed: 2023-07-03. [Online].
Available: <https://trepo.tuni.fi/bitstream/handle/123456789/21520/Nurminen.pdf>
- [11] (2023) Camelot. Accessed: 2023-06-04. [Online].
Available: <https://github.com/camelot-dev/camelot>
- [12] (2023) Tabula-java. Accessed: 2023-06-04. [Online].
Available: <https://github.com/tabulapdf/tabula-java>
- [13] M. M. Malik. (2023) Extraction of solar cell data from pdf datasheets. [Online].
Available: https://ad-publications.informatik.uni-freiburg.de/theses/Master_Moez_Malik_2023.pdf
- [14] (2023) pdfminer.six. [Online].
Available: <https://github.com/pdfminer/pdfminer.six>
- [15] (2023) pdfminer.six faq. [Online].
Available: <https://pdfminersix.readthedocs.io/en/latest/faq.html#why-are-there-cid-x-values-in-the-textual-output>
- [16] (2023) python chr() builtin. [Online].
Available: <https://docs.python.org/3/library/functions.html#chr>
- [17] (2023) pdfminer.six laparams. [Online].
Available: <https://pdfminersix.readthedocs.io/en/latest/reference/composable.html#laparams>
- [18] OpenData ÖPNV. Accessed: 2023-05-07. [Online].
Available: <https://www.opendata-oepnv.de/>

- [19] (2023) OSM Node wiki article. Accessed: 2023-05-05. [Online].
Available: <https://wiki.openstreetmap.org/wiki/Node>
- [20] (2023) OSM Way wiki article. Accessed: 2023-05-05. [Online].
Available: <https://wiki.openstreetmap.org/wiki/Way>
- [21] (2023) OSM Relation wiki article. Accessed: 2023-05-05. [Online].
Available: <https://wiki.openstreetmap.org/wiki/Relation>
- [22] (2023) Sparql. Accessed: 2023-06-04. [Online].
Available: <https://www.w3.org/TR/sparql11-query/>
- [23] (2023) IFOPT standard. Accessed: 2023-05-05. [Online].
Available: <https://www.transmodel-cen.eu/ifopt-standard/>
- [24] E. W. Dijkstra, “A note on two problems in connexion with graphs,” in *Edsger Wybe Dijkstra: His Life, Work, and Legacy*, 2022, pp. 287–290.
- [25] J. Heinzinger. (2023) pdf2gtfs. [Online].
Available: <https://ad-blog.informatik.uni-freiburg.de/post/transform-pdf-timetables-into-gtfs/>
- [26] Ghostscript. Accessed: 2023-07-02. [Online].
Available: <https://www.ghostscript.com/>
- [27] (2023) pandas. Accessed: 2023-05-11. [Online].
Available: <https://www.pandas.pydata.org>
- [28] (2023) RMV .csv-file containing information about stops. Accessed: 2023-06-29. [Online].
Available: <https://opendata.rmv.de/site/start.html>
- [29] (2023) Datensatz der VAG Freiburg. Accessed: 2023-06-29. [Online].
Available: <https://www.vag-freiburg.de/service-infos/downloads/gtfs-daten>

- [30] (2023) VGN GTFS-feed. Accessed: 2023-06-29. [Online].
Available: <https://www.vgn.de/web-entwickler/open-data/>
- [31] M. L. Fredman and R. E. Tarjan, “Fibonacci heaps and their uses in improved network optimization algorithms,” *Journal of the ACM (JACM)*, vol. 34, no. 3, pp. 596–615, 1987.
- [32] pfaedle. Accessed: 2023-07-05. [Online].
Available: <https://github.com/ad-freiburg/pfaedle>
- [33] Public Transit Map-Matching with GraphHopper. Accessed: 2023-07-02. [Online].
Available: https://ad-publications.cs.uni-freiburg.de/theses/Bachelor_Michael_Fleig_2021.pdf

