Adaptable crawler specification generation system for leisure activity RSS feeds

Mart Lubbers
s4109053
Radboud University Nijmegen

Alessandro Paula[1]
Hyperleap, Nijmegen

Franc Grootjen[2]
Artificial Intelligence, Nijmegen
Radboud University Nijmegen

July 8, 2015

[1]External supervisor
[2]Internal supervisor

Contents

Abstract

When looking for an activity in a bar or trying to find a good movie it often seems difficult to find complete and correct information about the event. Hyperleap tries to solve this problem of bad information giving by bundling the information from various sources and invest in good quality checking. Currently information retrieval is performed using site-specific crawlers, when a crawler breaks the feedback loop for fixing it contains different steps and requires someone with a computer science background. A crawler generation system has been created that uses directed acyclic word graphs to assist solving the feedback loop problem. The system allows users with no particular computer science background to create, edit and test crawlers for *RSS* feeds. In this way the feedback loop for broken crawlers is shortened, new sources can be incorporated in the database quicker and, most importantly, the information about the latest movie show, theater production or conference will reach the people looking for it as fast as possible.

Chapter 1

Introduction

1.1 Introduction

What do people do when they want to grab a movie? Attend a concert? Find out which theater shows play in local theater?

In the early days of the internet, access to the web was not available for most of the people. Information about leisure activities was almost exclusively obtained from flyers, posters and other written media and radio/TV advertisements. People had to put effort in searching for information and it was easy to miss a show just because you did not cross paths with it. Today the internet is used on a daily basis by almost everyone in the western society and one would think that missing an event would be impossible because of the enormous loads of information you can receive every day using the internet. For leisure activities the opposite is true, complete and reliable information about events is still hard to find.

Nowadays information on the internet about entertainment is offered via two main channels: individual venues websites and information bundling websites.

Individual venues put a lot of effort and resources in building a beautiful, fast and most of all modern website that bundles their information with nice graphics, animations and other gimmicks. Information bundling websites are run by companies that try to provide an overview of multiple venues. Information bundling websites often have individual venue websites as their source for information. Individual venues assume, for example, that it is obvious what the address of their venue is, that their ticket price is always fixed to €5.− and that you need a membership to attend the events. Individual organizations usually put this non specific information in a disclaimer or a separate page. Because of the less structured way of providing information the information bundling websites have a hard time finding complete information. The event data can be crawled using automated crawlers but the miscellaneous information usually has to be crawled by hand.

Combining the information from the different data source turns out to be a complicated task for information bundling websites. The task is difficult because the companies behind these information bundling websites do not have the resources and time reserved for these tasks and therefore often serve incomplete information. Because of the complexity of getting complete information there are not many companies trying to bundle entertainment information into a complete and consistent database and website. Hyperleap[1] tries to achieve the goal of serving complete and consistent information and offers it via various information bundling websites.

[1] `http://hyperleap.nl`

1.2 Hyperleap & Infotainment

Hyperleap is an internet company that was founded in the time that internet was not widespread. Hyperleap, active since 1995, is specialized in producing, publishing and maintaining *infotainment*. *Infotainment* is a combination of the words *information* and *entertainment*. It represents a combination of factual information, the *information* part, and non factual information or subjectual information, the *entertainment* part, within a certain category or field. In the case of Hyperleap the category is the leisure industry, leisure industry encompasses all facets of entertainment ranging from cinemas, theaters, concerts to swimming pools, bridge competitions and conferences. Within the entertainment industry factual information includes, but is not limited to, starting time, location, host or venue and duration. Subjectual information includes, but is not limited to, reviews, previews, photos, background information and trivia.

Hyperleap says to manage the largest database containing *infotainment* about the leisure industry focussed on the Netherlands and surrounding regions. The database contains over 10.000 categorized events on average per week and their venue database contains over 54.000 venues delivering the leisure activities ranging from theaters and music venues to petting zoos and fast food restaurants. All the subjectual information is obtained or created by Hyperleap and all factual information is gathered from different sources, quality checked and therefore very reliable. Hyperleap is the only company in its kind that has such high quality information. The *infotainment* is presented via several websites specialized per genre or category and some sites attract over 500.000 visitors each month.

1.3 Information flow

Hyperleap can keep up the high quality data by investing a lot of time and resources in quality checking, cross comparing and consistency checking. By doing so the chances of incomplete or wrong data are much lower. To achieve this, the data will go through several different stages before it enters the database. These stages are visualized in Figure 1.1 as an information flow diagram. In this diagram the nodes are processing steps and the arrows denote information transfer or flow.

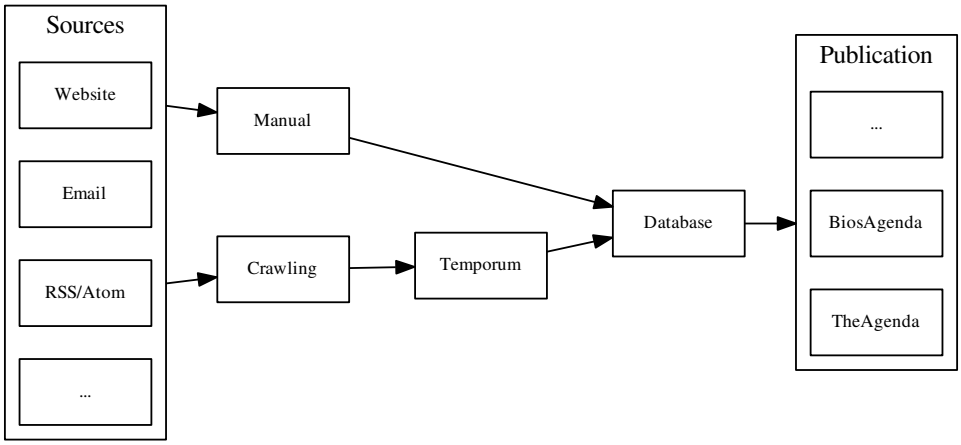

Figure 1.1: Information flow Hyperleap database

1.3.1 Sources

A source is a service, location or medium in which information about events is stored or published. A source can have different source shapes such as HTML, email, flyer, RSS and so on. All information gathered from a source has to be quality checked before it is even considered for automated crawling. There are several criteria to which the source has to comply before an automated crawler can be made. The prerequisites for a source are for example the fact that the source has to be reliable, consistent and free by licence. Event information from a source must have at least the *What, Where* and *When* information.

The *What* information is the information that describes the content, content is a very broad definition but in practice it can be describing the concert tour name, theater show title, movie title, festival title and many more.

The *Where* information is the location of the event. The location is often omitted because the organization presenting the information thinks it is obvious. This information can also include different sub locations. For example when a pop concert venue has their own building but in the summer they organize a festival in some park. This data is often assumed to be trivial and inherent but in practice this is not the case. In this example for an outsider only the name of the park is often not enough.

The *When* field is the time and date of the event. Hyperleap wants to have at minimum the date, start time and end time. In the field end times for example are often omitted because they are not fixed or the organization think it is obvious.

Venues often present incomplete data entries that do not comply to the requirements explained before. Within the information flow categorizing and grading the source is the first step. Hyperleap processes different sources and source types and every source has different characteristics. Sources can be modern sources like websites or social media but even today a lot of information arrives at Hyperleap via flyers, fax or email. As source types vary in content structure sources also vary in reliability. For example the following entry is an example of a very well structured and probably generated, and thus probably also reliable, event description. The entry can originate for example from the title of an entry in a RSS feed. The example has a clear structure and almost all information required is available directly from the entry.

`2015-05-20, 18:00-23:00 - Foobar presenting their new CD in combination with a show. Location: small salon.`

An example of a low quality item could be for example the following text that could originate from a flyer or social media post. This example lacks a precise date, time and location and is therefore hard for people to grasp at first, let alone for a machine. When someone wants to get the full information he has to tap in different resources which might not always be available.

`Foobar playing to celebrate their CD release in the park tomorrow evening.`

Information with such a low quality is often not suitable for automated crawling. In Figure 1.1 this manual route is shown by the arrow going straight from the source to the database. Non digital source types or very sudden changes such as surprise concerts or cancellations are also manually crawled.

1.3.2 Crawler

When the source has been determined and classified the next step is periodically crawling the source using an automated crawler. As said before sources have to be structured and reliable, when this is the case a programmer will create a program that will visit the website systematically and automatically to extract all the new information. The programmed crawlers are usually specifically created for one or more sources and when the source changes, the programmer has to adapt the crawler. Such a change is usually a change in structure. Since writing and adapting

requires a programmer the process is costly. Automatically crawled information is not inserted into the database directly because the information is not reliable enough. In case of a change in the source malformed data can pass through. As a safety net and final check the information first goes to the *Temporum* before it will be entered in the database.

1.3.3 Temporum

The *Temporum* is a big bin that contains raw data extracted from different sources using automated crawlers. Some of the information in the *Temporum* might not be suitable for the final database and therefore has to be post processed. The post-processing encompasses several different steps.

The first step is to check the validity of the event entries from a certain source. Validity checking is useful to detect outdated automated crawlers before the data can leak into the database. Crawlers become outdated when a source changes and the crawler can not crawl the website using the original method. Validity checking happens at random on certain event entries.

An event entry usually contains one occurrence of an event. In a lot of cases there is parent information that the event entry is part of. For example in the case of a concert tour the parent information is the concert tour and the event entry is a certain performance. The second step in post processing is matching the event entries to possible parent information. This parent information can be a venue, a tour, a showing, a tournament and much more.

Both of the post processing tasks are done by people with the aid of automated functionality. Within the two post processing steps malformed data can be spotted very fast and the *Temporum* thus acts as a safety net to keep the probability of malformed data leaking into the database as low as possible.

1.3.4 Database & Publication

Postprocessed data that leaves the *Temporum* will enter the final database. This database contains all the information about all the events that happened in the past and the events that will happen in the future. The database also contains the parent information such as information about venues. Several categorical websites use the database to offer the information to users and accompany it with the second part of *infotainment* namely the subjectual information. The *entertainment* part will usually be presented in the form of trivia, photos, interviews, reviews, previews and much more.

1.4 Goal & Research question

Maintaining the automated crawlers and the infrastructure that provides the *Temporum* and its matching aid automation are the parts within the data flow that require the most amount of resources. Both of these parts require a programmer to execute and therefore are costly. In the case of the automated crawlers it requires a programmer because the crawlers are scripts or programs are specifically designed for a particular website. Changing such a script or program requires knowledge about the source, the programming framework and about the *Temporum*. In practice both of the tasks mean changing code.

A large group of sources often change in structure. Because of such changes the task of reprogramming crawlers has to be repeated a lot. The detection of malfunctioning crawlers happens in the *Temporum* and not in an earlier stage. Late detection elongates the feedback loop because there is not always a tight communication between the programmers and the *Temporum* workers. In the case of a malfunction the source is first crawled. Most likely the malformed data will get processed and will produce rubbish that is sent to the *Temporum*. Within the *Temporum* after a while the error is detected and the programmers have to be contacted. Finally the crawler will be adapted to the new structure and will produce good data again. This feedback loop, shown in Figure 1.2, can take days and can be the reason for gaps and faulty information in the database. The

figure shows information flow with arrows. The solid and dotted lines form the current feedback loop.

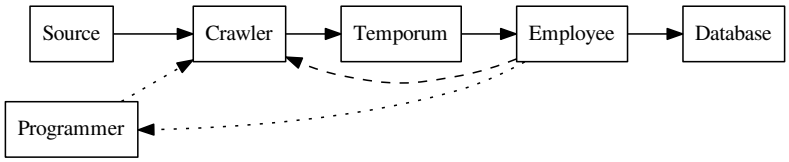

Figure 1.2: Feedback loop for malfunctioning crawlers

The specific goal of this project is to relieve the programmer of spending a lot of time repairing crawlers and make the task of adapting, editing and removing crawlers feasible for someone without programming experience. In practice this means shortening the feedback loop. The shorter feedback loop is also shown in Figure 1.2. The dashed line shows the shorter feedback loop that relieves the programmer.

For this project a system has been developed that provides an interface to a crawler generation system that is able to crawl RSS[2] and Atom[10] publishing feeds. The interface provides the user with point and click interfaces meaning that no computer science background is needed to use the interface and to create, modify, test and remove crawlers. The current Hyperleap backend system that handles the data can query XML feeds that contain the crawled data.

The actual research question can then be formulated as:

Is it possible to shorten the feedback loop for repairing and adding crawlers by making a system that can create, add and maintain crawlers for RSS feeds

1.5 RSS/Atom

RSS/Atom feeds, from now on called RSS feeds, are publishing feeds. Such feeds publish their data in a restricted XML format[3] consisting of entries. Every entry usually represents an event and consists of standardized data fields. The data fields we are interested in are the *title* and the *description* fields. Those fields store the raw data which describes the event. Besides the fields we are interested in there are there are several auxiliary fields that for example store the link to the full article, store the publishing data, store some media in the form of an image or video URL or store a *Globally Unique Identifier*(GUID)[2]. An example of a RSS feed can be found in Listing 1.1, this listing shows a, partly truncated, RSS feed of a well known venue in the Netherlands. Every RSS feed contains a `channel` field and within that field there is some metadata and a list op `item` fields. Every `item` field has a fixed number of different fields. The most important fields for RSS within the leisure industry are the `title` and the `description` field.

Listing 1.1: An example of a,partly truncated RSS feed of a well known dutch venue

```
1 <?xml version="1.0" ?>
2 <rss version="2.0">
3   <channel>
4     <title>Nieuw in de voorverkoop Paradiso</title>
5     <link>http://www.paradiso.nl/web/show/id=178182</link>
6     <description></description>
7       <item>
8         <title>donderdag 8 januari 2015 22:00 - Tee Pee Records Night - live:
```

[2]A GUID is a unique identifier that in most cases is the permalink of the article. A permalink is a link that will point to the article

```
9     Harsh Toke, Comet Control</title>
10          <link>http://www.paradiso.nl/web/Agenda-Item/Tee-Pee-Records-Night-
11    live-Harsh-Toke-Comet-Control.htm</link>
12          <description></description>
13          <pubDate>do, 27 nov 2014 11:34:00 GMT</pubDate>
14        </item>
15        <item>
16          <title>vrijdag 20 maart 2015 22:00 - Atanga Boom - cd release</title>
17          <link>
18    http://www.paradiso.nl/web/Agenda-Item/Atanga-Boom-cd-release.htm
19          </link>
20          <description></description>
21          <pubDate>do, 27 nov 2014 10:34:00 GMT</pubDate>
22        </item>
23        <item>
24          <title>zaterdag 21 maart 2015 20:00 - ...
25
26
27         ...
```

RSS feeds are mostly used by news sites to publish their articles. A RSS feed only contains the headlines of the entries. If the user, who reads the feed, is interested it can click the so called deeplink and will be sent to the full website containing the full entry or article. Users often use programs that bundle user specified RSS feeds into big combined feeds that can be used to sift through a lot of news feeds. The RSS feed reader uses the unique GUID to skip feeds that are already present in its internal database.

Generating RSS feeds by hand is a tedious task but almost all RSS feeds are generated by a Content Management Systems(CMS) on which the website runs. With this automatic method the RSS feeds are generated for the content published on the website. Because the process is automatic the RSS feeds are generally very structured and consistent in its structure. In the entertainment industry venues often use a CMS for their website to allow users with no programming or website background be able to post news items and event information and thus should often have RSS feeds.

1.6 Why RSS?

There are lots of different source formats like HTML, fax/email, RSS and XML. Because of the limited scope of the project and the time planned for it we had to remove some of the input formats because they all require different techniques and approaches to tackle. For example when the input source is in HTML format, most probably a website, then there can be a great deal of information extraction be automated using the structural information which is a characteristic for HTML. For fax/email however there is almost no structural information and most of the automation techniques require natural language processing and possibly OCR. We chose RSS feeds because RSS feeds lack inherent structural information but are still very structured. This structure is because, as said above, the RSS feeds are generated and therefore almost always look the same. Also, in RSS feeds most venues use particular structural identifiers that are characters. They separate fields with vertical bars, commas, whitespace and more non text characters. These field separators and keywords can be hard for a computer to detect but people are very good in detecting these. With one look they can identify the characters and keywords and build a pattern in their head. Another reason we chose RSS is their temporal consistency, RSS feeds are almost always generated and because of that the structure of the entries is very unlikely to change. Basically the RSS feeds only change structure when the CMS that generates it changes the generation algorithm. This property is useful because the crawlers then do not have to be retrained very often. To detect the underlying structures a technique is used that exploits subword matching with graphs.

1.7 Directed Acyclic Graphs

Directed graphs Directed graphs(DG) are mathematical structures that can describe relations between nodes. A directed graph G is be defined as the tuple (V, E) where V is a set of named nodes and E is the set of edges defined by $E \subseteq V \times V$. An edge $e \in E$ is defined as $(v1, v2)$ where $v1, v2 \in V$ and is show in the figure as an arrow between node $v1$ and node $v2$. Multiple connections between two nodes are possible if the directions differ. For example the graph visualized in Figure 1.3 can be mathematically described as:

$$G = (\{n1, n2, n3, n4\}, \{(n1, n2), (n2, n1), (n2, n3), (n3, n4), (n1, n4)\}$$

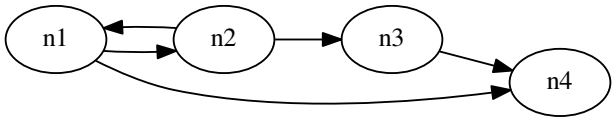

Figure 1.3: Example DG

Directed acyclic graphs Directed Acyclic Graphs(DAGs) are a special kind of directed graphs. DAGs are also defined as $G = (V, E)$ but with a restriction on E. Namely that cycles are not allowed. Figure 1.4 shows two graphs. The bottom graph contains a cycle and the right graph does not. Only the top graph is a valid DAG. A cycle can be defined as follows:

If $e \in E$ is defined as $(v_1, v_n) \in E$ or $v_1 \to v_n$ then $v_1 \xrightarrow{+} v_n$ which means $v_1 \to v_2 \to \ldots \to v_{n-1} \to v_n$ meaning that there is a connection with a length larger then 1 between v_1 and v_2. In a non cyclic graph the following holds $\nexists v \in V : v \xrightarrow{+} v$. In words this means that a node can not be reached again traveling the graph. Adding the property of non cyclicity to graphs lowers the computational complexity of path existence in the graph to $\mathcal{O}(L)$ where L is the length of the path.

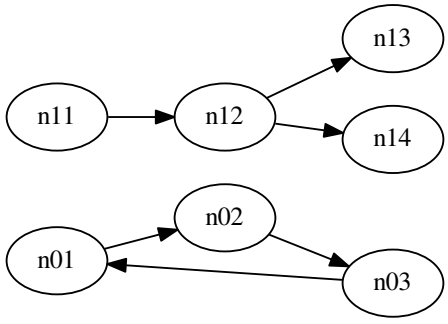

Figure 1.4: Example DAG

Directed Acyclic Word Graphs The type of graph used in the project is a special kind of DAG called Directed Acyclic Word Graphs(DAWGs). A DAWG can be defined by the tuple $G = (V, v_0, E, F)$. V is the same as in directed graphs, namely the set of nodes. E is the set of edges described by $E \subseteq V \times V \times L$ where $L \in A$ where A is the alphabet used as node labels. In words this means that an edge is a labeled arrow between two nodes and for example the edge (v_1, v_2, a) can be visualized as: $v_1 \xrightarrow{a} v_2$. In a standard DAWG the alphabet A contains all the characters used in natural language but in theory the alphabet A can contain anything. F describes the set of final nodes, final nodes are nodes that can be the end of a sequence even if there is an arrow leading out. In the example graph in Figure 1.5 the final nodes are visualized with a double circle as node shape. In this example it is purely cosmetic because $n6$ is a final node anyways because there are no arrows leading out. But this does not need to be the case, for example in $G = (\{n1, n2, n3\}, \{(n1, n2), (n2, n3)\}, \{n2, n3\})$ there is a distinct use for the final node marking. The only final node is the example is n_6, marked with a double circle. v_0 describes the initial node, this is visualized in figures as an incoming arrow. Because of the property of labeled edges, data can be stored in a DAWG. When traversing a DAWG and saving all the edge labels one can construct words. Using graph minimisation big sets of words can be stored using a small amount of storage because edges can be re-used to specify transitions. For example the graph in Figure 1.5 can describe the language L where all words w that are accepted $w \in \{abd, bad, bae\}$. Testing if a word is present in the DAWG is the same technique as testing if a node path is present in a normal DAG and therefore also falls in the computational complexity class of $\mathcal{O}(L)$. This means that it grows linearly with the length of the word.

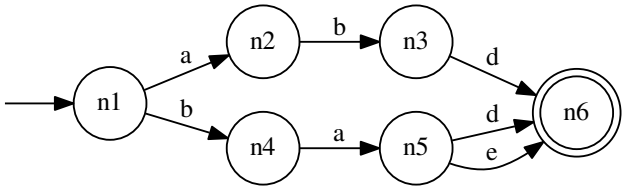

Figure 1.5: Example DAWG

1.8 Structure

The following chapters will describe the system that has been created and the used methods. Chapter 2 shows the requirements design for the program followed in Chapter 3 by the underlying methods used for the actual matching. Finally Chapter 4 concludes with the results, discussion and future research.

Chapter 2

Requirements & Application design

2.1 Requirements

2.1.1 Introduction

As almost every plan for an application starts with a set of requirements so will this application. Requirements are a set of goals within different categories that will define what the application has to be able to do and they are traditionally defined at the start of the project and not expected to change much. In the case of this project the requirements were a lot more flexible because there was only one person doing the programming and there was a weekly meeting to discuss the matters and most importantly discuss the required changes. Because of this a lot of initial requirements are removed and a some requirements were added in the process. The list below shows the definitive requirements and also the suspended requirements.

There are two types of requirements, functional and non-functional requirements. Functional requirements are requirements that describe a certain function in the technical sense. Non-functional requirements describe a property. Properties can be for example efficiency, portability or compatibility. To make us able to refer to them later we give the requirements unique codes. As for the definitive requirements a verbose explanation is also provided.

2.1.2 Functional requirements

Original functional requirements

I1: The system should be able to crawl several source types.

- I1a: Fax/email.
- I1b: XML feeds.
- I1c: RSS feeds.
- I1d: Websites.

I2: Apply low level matching techniques on isolated data.

I3: Insert data in the database.

I4: The system should have a user interface to train crawlers that is usable by someone without a particular computer science background.

I5: The system should be able to report to the employee when a source has been changed too much for successful crawling.

Definitive functional requirements

Requirement I2 is the one requirement that is dropped completely, this is due to time constraints. The time limitation is partly because we chose to implement certain other requirements like an interactive intuitive user interface around the core of the pattern extraction program. Below are all definitive requirements.

F1: The system should be able to crawl RSS feeds.

This requirement is an adapted version of the compound requirements I1a-I1d. We limited the source types to crawl to strict RSS because of the time constraints of the project. Most sources require an entirely different strategy and therefore we could not easily combine them. An explanation why we chose RSS feeds can be found in Section 1.6.

F2: Export the data to a strict XML feed.

This requirement is an adapted version of requirement I3, this is also done to limit the scope. We chose to not interact directly with the database or the *Temporum*. The application however is able to output XML data that is formatted following a string XSD scheme so that it is easy to import the data in the database or *Temporum* in a indirect way.

F3: The system should have a user interface to create crawlers that is usabl for someone without a particular computer science background.

This requirement is formed from I4. Initially the user interface for adding and training crawlers was done via a web interface that was user friendly and usable by someone without a particular computer science background as the requirement stated. However in the first prototypes the control center that could test, edit and remove crawlers was a command line application and thus not very usable for the general audience. This combined requirements asks for a single control center that can do all previously described tasks with an interface that is usable without prior knowledge in computer science.

F4: Report to the user or maintainer when a source has been changed too much for successful crawling.

This requirement was also present in the original requirements and has not changed. When the crawler fails to crawl a source, this can be due to any reason, a message is sent to the people using the program so that they can edit or remove the faulty crawler. Updating without the need of a programmer is essential in shortening the feedback loop explained in Figure 1.2.

2.1.3 Non-functional requirements

Original functional requirements

O1: Integrate in the existing system used by Hyperleap.

O2: The system should work in a modular fashion, thus be able to, in the future, extend the program.

Definitive functional requirements

N1: Work in a modular fashion, thus be able to, in the future, extend the program.

The modularity is very important so that the components can be easily extended and components can be added. Possible extensions are discussed in Section 4.2.

N2: Operate standalone on a server.

Non-functional requirement O1 is dropped because we want to keep the program as modular as possible and via an XML interface we still have a very intimate connection with the

database without having to maintain a direct connection. The downside of an indirect connection instead of a direct connection is that the specification is much more rigid. If the system changes the specification the backend program should also change.

2.2 Application overview

The workflow of the application can be divided into several components or steps. The overview of the application is visible in Figure 2.1. The nodes are applications or processing steps and the arrows denote information flow or movement between nodes.

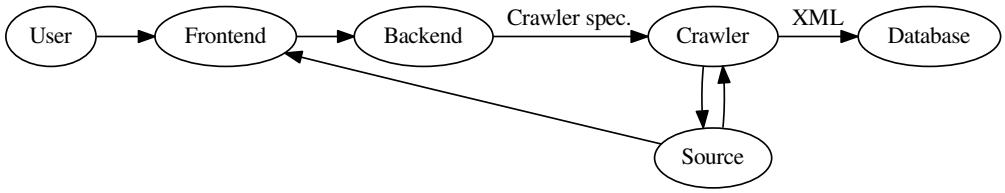

Figure 2.1: Overview of the application

2.2.1 Frontend

General description

The frontend is a web interface that is connected to the backend system which allows the user to interact with the backend. The frontend consists of a basic graphical user interface which is shown in Figure 2.3. As the interface shows, there are three main components that the user can use. There is also a button for downloading the XML. The *Get xml* button is a quick shortcut to make the backend to generate XML. The button for grabbing the XML data is only for diagnostic purposes located there. In the standard workflow the XML button is not used. In the standard workflow the server periodically calls the XML output option from the command line interface of the backend to process it.

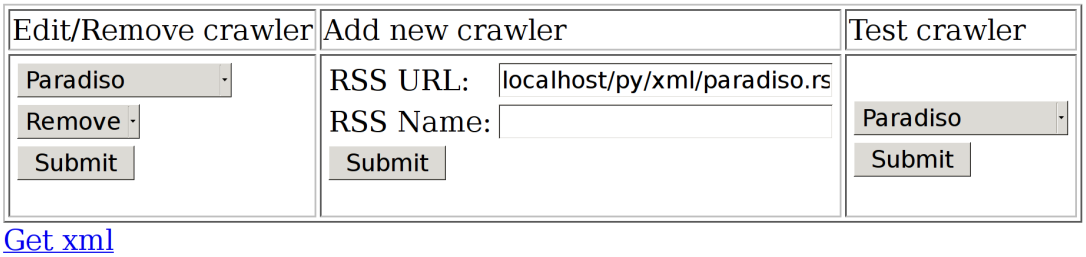

Figure 2.2: The landing page of the frontend

Repair/Remove crawler

This component lets the user view the crawlers and remove the crawlers from the crawler database. Doing one of these things with a crawler is as simple as selecting the crawler from the dropdown menu and selecting the operation from the other dropdown menu and pressing *Submit*.

Removing the crawler will remove the crawler completely from the crawler database and the crawler will be unrecoverable. Editing the crawler will open a similar screen as when adding the crawler. The details about that screen will be discussed in Section 2.2.1. The only difference is that the previous trained patterns are already made visible in the training interface and can thus be adapted to change the crawler for possible source changes for example.

Add new crawler

The addition or generation of crawlers is the key feature of the program and it is the intelligent part of the system since it includes the graph optimization algorithm to recognize user specified patterns in the new data. First, the user must fill in the static form that is visible on top of the page. This for example contains general information about the venue together with some crawler specific values such as crawling frequency. After that the user can mark certain points in the table as being of a category. Marking text is as easy as selecting the text and pressing the according button. The text visible in the table is a stripped down version of the original RSS feeds `title` and `summary` fields. When the text is marked it will be highlighted in the same color as the color of the button text. The entirety of the user interface with a few sample markings is shown in Figure 2.3. After the marking of the categories the user can preview the data or submit. Previewing will run the crawler on the RSS feed in memory and the user can revise the patterns if necessary. Submitting will send the page to the backend to be processed. The internals of what happens after submitting is explained in detail in Figure 3.1 together with the text.

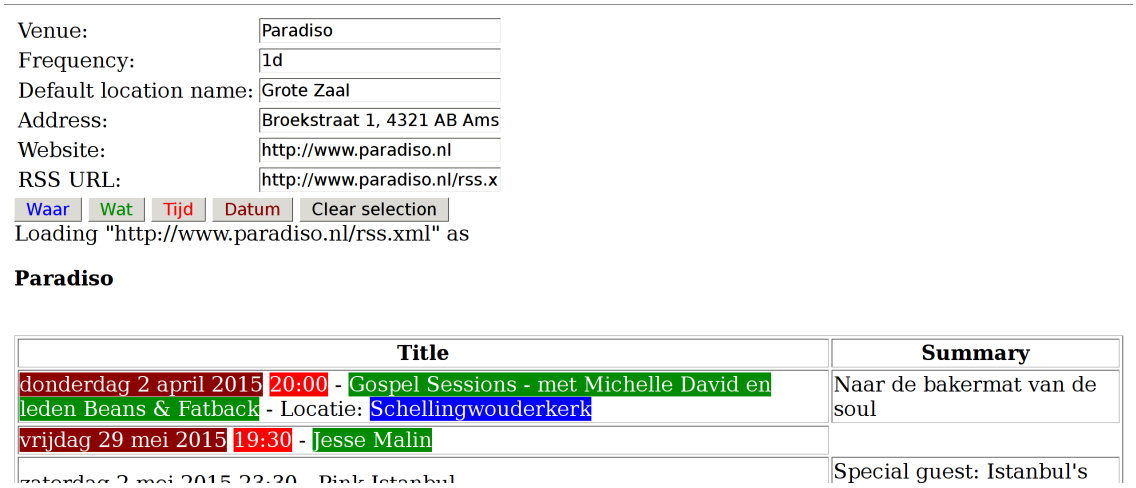

Figure 2.3: A view of the interface for specifying the pattern. Two entries are already marked.

Test crawler

The test crawler component is a very simple non interactive component that allows the user to verify if a crawler functions properly without having to access the database via the command line utilities. Via a dropdown menu the user selects the crawler and when submit is pressed the backend generates a results page that shows a small log of the crawler, a summary of the results and most importantly the results itself. In this way the user can see in a few gazes if the crawler functions properly. Humans are very fast in detecting patterns and therefore the error checking goes very fast. Because the log of the crawl operation is shown this page can also be used for diagnostic information about the backends crawling system. The logging is in depth and also shows possible exceptions and is therefore also usable for the developers to diagnose problems.

2.2.2 Backend

Program description

The backend consists of a main module and a set of libraries all written in *Python*[11]. The main module is embedded in an apache HTTP-server[6] via the *mod_python* apache module[12]. The module *mod_python* allows handling for python code via HTTP and this allows us to integrate neatly with the *Python* libraries. We chose *Python* because of the rich set of standard libraries and solid cross platform capabilities. We chose specifically for *Python* 2 because it is still the default *Python* version on all major operating systems and stays supported until at least the year 2020. This means that the program can function at least for 5 full years. The application consists of a main *Python* module that is embedded in the HTTP-server. Finally there are some libraries and there is a standalone program that does the periodic crawling.

Main module

The main module is the program that deals with the requests, controls the frontend, converts the data to patterns and sends the patterns to the crawler. The module serves the frontend in a modular fashion. For example the buttons and colors can be easily edited by a non programmer by just changing the appropriate values in a text file. In this way even when conventions change the program can still function without intervention of a programmer that needs to adapt the source.

Libraries

The libraries are called by the main program and take care of all the hard work. Basically the libraries are a group of python scripts that for example minimize the graphs, transform the user data into machine readable data, export the crawled data to XML and much more.

Standalone crawler

The crawler is a program, also written in Python, that is used by the main module and technically is part of the libraries. The property in which the crawler stands out is the fact that it also can run on its own. The crawler has to run periodically by a server to literally crawl the websites. The main module communicates with the crawler when it is queried for XML data, when a new crawler is added or when data is edited. The crawler also offers a command line interface that has the same functionality as the web interface of the control center.

The crawler saves all the data in a database. The database is a simple dictionary where all the entries are hashed so that the crawler knows which ones are already present in the database and which ones are new. In this way the crawler does not have to process all the old entries when they appear in the feed. The RSS' GUID could also have been used but since it is an optional value in the feed not every feed uses the GUID and therefore it is not reliable to use it. The crawler also has a function to export the database to XML format. The XML output format is specified in an XSD[1] file for minimal ambiguity.

XML & XSD

XML is a file format that can describe data structures. XML can be accompanied by an XSD file that describes the format. An XSD file is in fact just another XML file that describes the format of XML files. Almost all programming languages have an XML parser built in and therefore it is a very versatile format that makes the eventual import to the database very easy. The most used languages also include XSD validation to detect XML errors, validity and completeness of XML files. This makes interfacing with the database and possible future programs even more easy. The XSD scheme used for this programs output can be found in the appendices in Algorithm 5.1. The XML output can be queried via the HTTP interface that calls the crawler backend to crunch the latest crawled data into XML. It can also be acquired directly from the crawlers command line interface.

Chapter 3

Algorithm

3.1 Application overview

The backend consists of several processing steps that the input has go through before it is converted to a crawler specification. These steps are visualized in Figure 3.1. All the nodes are important milestones in the process of processing the user data. Arrows indicate information transfer between these steps. The Figure is a detailed explanation of the *Backend* node in Figure 2.1.

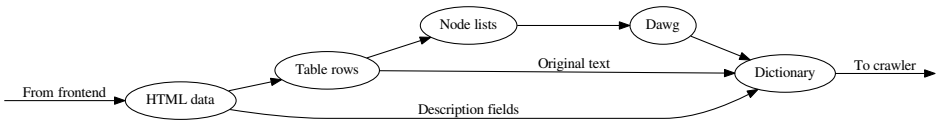

Figure 3.1: Main module internals

3.2 HTML data

The raw data from the frontend with the user markings enter the backend as a HTTP *POST* request. This *POST* request consists of several information data fields. These data fields are either fields from the static description boxes in the frontend or raw *HTML* data from the table showing the processed RSS feed entries which contain the markings made by the user. The table is sent in whole precisely at the time the user presses the submit button. Within the *HTML* data of the table markers are placed before sending. These markers make the parsing of the tables more easy and remove the need for an advanced *HTML* parser to extract the markers. The *POST* request is not send asynchronously. Synchronous sending means the user has to wait until the server has processed the request. In this way the user will be notified immediately when the processing has finished successfully and will be able to review and test the resulting crawler. All the data preparation for the *POST* request is done in *Javascript* and thus happens on the user side. All the processing afterwards is done in the backend and thus happens on the server side. All descriptive fields will also be put directly in the final aggregating dictionary. All the raw *HTML* data will be sent to the next step.

3.3 Table rows

The first conversion step is to extract individual table rows from the *HTML* data. In the previous step markers were placed to make the identification easy of table rows. Because of this,

extracting the table rows only requires rudimentary matching techniques that require very little computational power. User markings are highlights of certain text elements. The highlighting is done using `SPAN` elements and therefore all the `SPAN` elements have to be found and extracted. To achieve this for every row all `SPAN` elements are extracted to get the color and afterwards to remove the element to retrieve the original plain text of the *RSS* feed entry. When this step is done a data structure containing all the text of the entries together with the markings will go to the next step. All original data, namely the *HTML* data per row, will be transferred to the final aggregating dictionary.

3.4 Node lists

Every entry gotten from the previous step is going to be processing into so called node-lists. A node-list can be seen as a path graph where every character and marking has a node. A path graph G is defined as $G = (V, n_1, E, n_i)$ where $V = \{n_1, n_2, \ldots, n_{i-1}, n_i\}$ and $E = \{(n_1, n_2), (n_2, n_3), \ldots (n_{i-1}, n_i)\}$. A path graph is basically a graph that is a single linear path of nodes where every node is connected to the next node except for the last one. The last node is the only final node. The transitions between two nodes is either a character or a marking. As an example we take the entry `19:00, 2014--11--12 - Foobar` and create the corresponding node-lists and it is shown in Figure 3.2. Characters are denoted with single quotes, spaces with an underscore and markers with angle brackets. Node-lists are the basic elements from which the DAWG will be generated. These node-lists will also be available in the final aggregating dictionary to ensure consistency of data and possibility of regenerating the data.

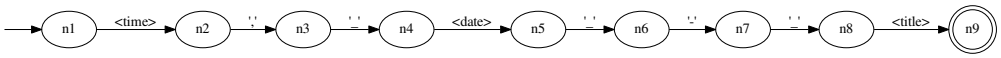

Figure 3.2: Node list example

3.5 DAWGs

3.5.1 Terminology

Parent nodes are nodes that have an arrow to the child.
Confluence nodes are nodes that have multiple parents.

3.5.2 Datastructure

We represent the user generated patterns as DAWGs by converting the node-lists to DAWGs. Normally DAWGs have single letters from an alphabet as edgelabel but in our implementation the DAWGs alphabet contains all letters, whitespace and punctuation but also the specified user markers which can be multiple characters of actual length but for the DAWGs sake they are one transition in the graph.

DAWGs are graphs but due to the constraints we can use a DAWG to check if a match occurs by checking if a path exists that creates the word by concatenating all the edge labels. The first algorithm to generate DAWGs from words was proposed by Hopcroft et al[7]. It is an incremental approach in generating the graph. Meaning that entry by entry the graph will be expanded. Hopcrofts algorithm has the constraint of lexicographical ordering. Later on Daciuk et al.[5][4] improved on the original algorithm and their algorithm is the algorithm we used to generate minimal DAWGs from the node lists.

3.5.3 Algorithm

The algorithm of building DAWGs is an iterative process that goes roughly in four steps. We start with the null graph that can be described by $G_0 = (\{q_0\}, \{q_0\}, \{\}\{\})$ and does not contain any edges, one node and $\mathcal{L}(G_0) = \emptyset$. The first word that is added to the graph will be added in a naive way and is basically replacing the graph by the node-list which is a path graph. We just create a new node for every transition of character and we mark the last node as final. From then on all words are added using a four step approach described below. Pseudocode for this algorithm can be found in Listing 1 named as the function `generate_dawg(words)`. A *Python* implementation can be found in Listing 5.2.

1. Say we add word w to the graph. Step one is finding the common prefix of the word already in the graph. The common prefix is defined as the longest subword w' for which there is a $\delta^*(q_0, w')$. When the common prefix is found we change the starting state to the last state of the common prefix and remain with suffix w'' where $w = w'w''$.

2. We add the suffix to the graph starting from the last node of the common prefix.

3. When there are confluence nodes present within the common prefix they are cloned starting from the last confluence node to avoid adding unwanted words.

4. From the last node of the suffix up to the first node we replace the nodes by checking if there are equivalent nodes present in the graph that we can merge with.

```
def generate_dawg(words):
    register := ∅;
    while there is another word do
        word := next word;
        commonprefix := CommonPrefix(word);
        laststate := delta*(q0, commonprefix);
        currentsuffix := word[length(commonprefix)...length(word)];
        if has_children(laststate) then
            replace_or_register(laststate);
        end
        add_suffix(laststate, currentsuffix);
    end
    replace_or_register(q0);
end
def replace_or_register_dawg(state):
    child := last_child(state);
    if has_children(child) then
        replace_or_register(child);
    end
    if there is an equivalent state q then
        last_child(state);
        delete(child);
    else
        register.add(child);
    end
end
```

Algorithm 1: Generating DAWGs pseudocode

3.5.4 Example

The size of the graphs that are generated from real world data from the leisure industry grows extremely fast. Therefore the example concists of short strings instead of real life event information. The algorith is visualized with an example shown in the Subgraphs in Figure 3.3 that builds a DAWG with the following entries: `abcd`, `aecd` and `aecf`.

- No words added yet
  Initially we begin with the null graph. This graph is show in the figure as SG0. This DAWG does not yet accept any words.

- Adding `abcd`
  Adding the first entry `abcd` is trivial because we can just create a single path which does not require any hard work. This is because the common prefix we find in Step 1 is empty and the suffix will thus be the entire word. Merging the suffix back into the graph is also not possible since there are no nodes except for the first node. The result of adding the first word is visible in subgraph SG1.

- Adding `aecd`
  For the second entry we will have to do some extra work. The common prefix found in Step 1 is `a` which we add to the graph. This leaves us in Step 2 with the suffix `ecd` which we add too. In Step 3 we see that there are no confluence nodes in our common prefix and therefore we do not have to clone nodes. In Step 4 we traverse from the last node back to the beginning of the suffix and find a common suffix `cd` and we can merge these nodes. In this way we can reuse the transition from q_3 to q_4. This leaves us with subgraph SG2.

- Adding `aecf`
  We now add the last entry which is the word `aecf`. When we do this without the confluence node checking we encounter an unwanted extra word. In Step 1 we find the common prefix `aec` and that leaves us with the suffix `f` which we will add. This creates subgraph SG3 and we notice there is an extra word present in the graph, namely the word `abcf`. This extra word appeared because of the confluence node q_3 which was present in the common prefix introducing an unwanted path. Therefore in Step 3 when we check for confluence node we would see that node q_3 is a confluence node and needs to be cloned off the original path, by cloning we take the added suffix with it to create an entirely new route. Tracking the route back again we do not encounter any other confluence nodes so we can start Step 4. For this word there is no common suffix and therefore no merging will be applied. This results in subgraph SG4 which is the final DAWG containing only the words added.

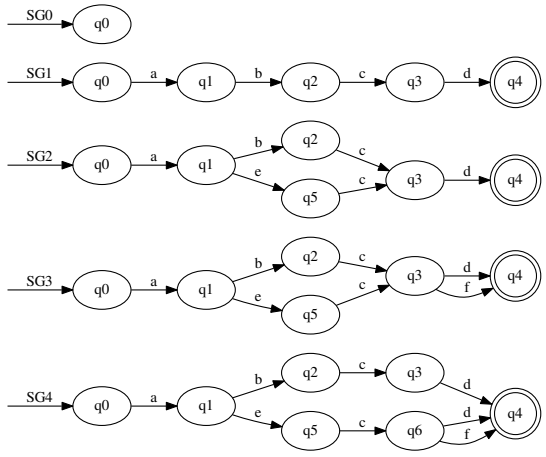

Figure 3.3: Incrementally constructing a DAWG

3.5.5 Appliance on extraction of patterns

The text data in combination with the user markings can not be converted automatically to a DAWG using the algorithm we described. This is because the user markings are not necessarily a single character or word. Currently user markings are subgraphs that accept any word of any length. When we add a user marking, we are inserting a kind of subgraph in the place of the node with the marking. By doing this we can introduce non determinism to the graph. Non determinism is the fact that a single node has multiple edges with the same transition, in practise this means it could happen that a word can be present in the graph in multiple paths. An example of non determinism in one of our DAWGs is shown in Figure 3.4. This figure represents a generated DAWG with the following entries: `ab<1>c`, `a<1>bc`.

In this graph the word `abdc` will be accepted and the user pattern `<1>` will be filled with the subword `d`. However if we try the word `abdddbc` both paths can be chosen. In the first case the user pattern `<1>` will be filled with `dddb` and in the second case with `bddd`. In such a case we need to choose the hopefully smartest choice. In the case of no paths matching the system will report a failed extraction. The crawling system can be made more forgiving in such a way that it will give partial information when no match is possible, however it still needs to report the error and the data should be handled with extra care.

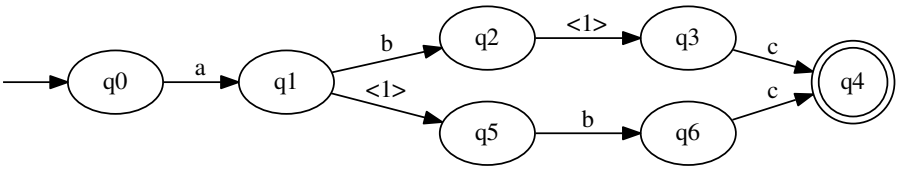

Figure 3.4: Example non determinism

3.5.6 Minimality & non-determinism

The Myhill-Nerode theorem [8] states that for every number of graphs accepting the same language there is a single graph with the least amount of states. Mihov[9] has proven that the algorithm for generating DAWGs is minimal in its original form. Our program converts the node-lists to DAWGs that can possibly contain non deterministic transitions from nodes and therefore one can argue about Myhill-Nerodes theorem and Mihovs proof holding. Due to the nature of the determinism this is not the case and both hold. In reality the graph itself is only non-deterministic when expanding the categories and thus only during matching.

Choosing the smartest path during matching the program has to choose deterministically between possibly multiple path with possibly multiple results. There are several possibilities or heuristics to choose from.

- Maximum fields heuristic

  This heuristic prefers the result that has the highest amount of categories filled with actual text. Using this method the highest amount of data fields will be getting filled at all times. The downside of this method is that because of this it might be that some data is not put in the right field because a suboptimal splitting occurred that has put the data in two separate fields whereas it should be in one field.

- Maximum path heuristic

  Maximum path heuristic tries to find a match with the highest amount of fixed path transitions. Fixed path transitions are transitions that occur not within a category. The philosophy behind is, is that because the path are hard coded in the graph they must be important. The downside of this method is when overlap occurs between hard coded paths and information within the categories. For example a band that is called `Location` could interfere greatly with a hard coded path that marks a location using the same words.

The more one knows about the contents of the categories the better the Maximum field heuristic performs. When, as in our current implementation, categories do not contain information both heuristics perform about the same.

Chapter 4

Conclusion & Discussion

4.1 Conclusion

Is it possible to shorten the feedback loop for repairing and adding crawlers by making a system that can create, add and maintain crawlers for RSS feeds

The short answer to the problem statement made in the introduction is yes. We can shorten the loop for repairing and adding crawlers which our system. The system can provide the necessary tools for a user with no particular programming skills to generate crawlers and thus the number of interventions where a programmer is needed is greatly reduced. Although we have solved the problem we stated the results are not strictly positive. This is because a if the problem space is not large the interest of solving the problem is also not large, this basically means that there is not much data to apply the solution on.

Although the research question is answered the underlying goal of the project has not been completely achieved. The application is an intuitive system that allows users to manage crawlers and for the specific domain, RSS feeds. By doing that it does shorten the feedback loop but only for RSS feeds. In the testing phase on real world data we stumbled on a small problem. Lack of RSS feeds and misuse of RSS feeds leads to a domain that is significantly smaller then first theorized and therefore the application solves only a very small portion.

Lack of RSS feeds is a problem because a lot of entertainment venues have no RSS feeds available for the public. Venues either using different techniques to publish their data or do not publish their data at all via a structured source besides their website. This shrinks the domain quite a lot. Taking pop music venues as an example. In a certain province of the Netherlands we can find about 25 venues that have a website and only 3 have a RSS feed. Extrapolating this information combined with information from other regions we can safely say that less then 10% of the venues even has a RSS feed.

The second problem is misuse of RSS feeds. RSS feeds are very structured due to their limitations on possible fields. We found that a lot of venues that are using a RSS feed seem not to be content with the limitations and try to bypass such limitations by misusing the protocol. A common misuse is to use the publication date field to put the date of the actual event in. When loading such a RSS feed into a general RSS feed reader the outcome is very strange because a lot of events will have a publishing date in the future and therefore messing up the order in your program. The misplacement of key information leads to lack of key information in the expected fields and by that lower overall extraction performance.

The second most occurring common misuse is to use HTML formatted text in the RSS feeds text fields. The algorithm is designed to detect and extract information via patterns in plain text and the performance on HTML is very bad compared to plain text. A text field with HTML is almost useless to gather information from because they usually include all kinds of information in other modalities then text. Via a small study on a selection of RSS feeds($N = 10$) we found that

about 50% of the RSS feeds misuse the protocol in such a way that extraction of data is almost impossible. This reduces the domain of good RSS feeds to less then 5% of the venues.

4.2 Discussion & Future Research

The application we created does not apply any techniques on the extracted data fields. The application is built only to extract and not to process the labeled data fields with text. When we would combine the information about the global structure and information about structure in a marked area we increase performance in two ways. A higher levels of performance are reached due to the structural information of marked areas. Hereby extra knowledge as extra constraint while matching the data in marked areas. The second increase in performance of the application is because the error detection happens more quickly. Faster error detection is possible because when the match is correct at a global level it can still contain wrong information at the lower marked field level. Applying matching techniques on the marked fields afterwards can generate feedback that could also be useful for the global level of data extraction.

Another use or improvement could be combining the forces of HTML and RSS. Some specifically structured HTML sources could be converted into a tidy RSS feed and still get processed by this application. In this way, with an extra intermediate step, the extraction techniques can still be used. HTML sources most likely have to be generated from a source by the venue because there has to be a very consistent structure in the data. Websites with such great structure are usually generated from a CMS. This will enlarge the domain for the application significantly since almost all websites use CMS to publish their data.

The interface of the program could also be re-used. When conversion between HTML and RSS feeds is not possible but one has a technique to extract patterns in a similar way then this application it is also possible to embed it in the current application. Due to the modularity of the application extending the application with other matching techniques is very easy.

Chapter 5

Appendices

5.1 XSD schema

Listing 5.1: XSD scheme for XMLoutput

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="xs3p.xsl"?>
<xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified"
   xmlns:xs="http://www.w3.org/2001/XMLSchema">
<!--This is the main element, required. It contains crawler and/data entries-->
  <xs:element name="crawler_output">
    <xs:complexType>
      <xs:sequence>
<!--
Crawler entries contain the information of the crawler, there can be multiple
-->
        <xs:element name="crawler" maxOccurs="unbounded" minOccurs="0">
          <xs:complexType>
            <xs:simpleContent>
              <xs:extension base="xs:string">
                <xs:attribute type="xs:string" name="name" use="optional"/>
                <xs:attribute type="xs:string" name="venue" use="optional"/>
                <xs:attribute type="xs:string" name="freq" use="optional"/>
                <xs:attribute type="xs:string" name="def_loc" use="optional"/>
                <xs:attribute type="xs:string" name="adress" use="optional"/>
                <xs:attribute type="xs:anyURI" name="website" use="optional"/>
                <xs:attribute type="xs:anyURI" name="url" use="optional"/>
              </xs:extension>
            </xs:simpleContent>
          </xs:complexType>
        </xs:element>
<!--
Data entries contain the information of a single crawled entry, there can be
multiple
-->
        <xs:element name="data" maxOccurs="unbounded" minOccurs="0">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="entry">
                <xs:complexType>
                  <xs:sequence>
<!--These four fields contain the user data-->
                    <xs:element type="xs:string" name="where"/>
                    <xs:element type="xs:string" name="what"/>
                    <xs:element type="xs:string" name="date"/>
                    <xs:element type="xs:string" name="time"/>
<!--These fields contain the raw original title and summary-->
                    <xs:element type="xs:string" name="full_title"/>
                    <xs:element type="xs:string" name="full_summary"/>
```

```
<!--These fields contain some other information from the rss-->
                      <xs:element type="xs:anyURI" name="link"
                        maxOccurs="unbounded" minOccurs="0"/>
                      <xs:element type="xs:string" name="pub_date"
                        maxOccurs="unbounded" minOccurs="0"/>
<!--Extracted URIs is a list of urls, this can be empty-->
                      <xs:element name="extracted_uris" maxOccurs="unbounded"
                        minOccurs="0">
                        <xs:complexType>
                          <xs:sequence>
                            <xs:element type="xs:anyURI" name="url"
                              maxOccurs="unbounded" minOccurs="0"/>
                          </xs:sequence>
                        </xs:complexType>
                      </xs:element>
                    </xs:sequence>
                  </xs:complexType>
                </xs:element>
              </xs:sequence>
<!--These fields specify the crawler name and the date crawled-->
              <xs:attribute type="xs:string" name="from"/>
              <xs:attribute type="xs:dateTime" name="date"/>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
</xs:schema>
```

5.2 Algorithm

Listing 5.2: DAWG generation algorithm in Python

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-


class DAWG:
    """Class representing a DAWG

    Variables:
    v     - Set of nodes
    v0    - Start node
    f     - Set of final nodes
    delta - Delta function
    """
    def __init__(self):
        """Initialize with null graph."""
        self.v = {'q0'}
        self.delta = {'q0': {}}
        self.f = set()
        self.v0 = 'q0'
        self.register = None

    def common_prefix(self, word, current):
        """Find the common prefix of word starting from current."""
        try:
            char, rest = word[0], word[1:]
            return char + self.common_prefix(rest, self.delta[current][char])
        except (KeyError, IndexError):
            return ''

    def delta_star(self, c, w):
        """Calculate the final node when traversing the graph from node c with
        word w"""
        return c if not w else self.delta_star(self.delta[c][w[0]], w[1:])

```

```
35      def equiv(self, a, b):
36          """Determine if two nodes are equivalent. This is the case when they
37          have the same final flag, the same number of children and their
38          children are equal."""
39          if (a in self.f) != (b in self.f) or\
40                  self.delta[a].keys() != self.delta[b].keys():
41              return False
42          return all(self.equiv(x, y) for x, y in
43                     zip(self.delta[a].values(), self.delta[b].values()))
44
45      def replace_or_register(self, suffix):
46          """Starting from the back try to merge nodes in the suffix."""
47          while suffix:
48              parent, char, state = suffix.pop()
49              for r in self.register:
50                  if self.equiv(state, r):
51                      self.delta[parent][char] = r
52                      if state in self.f:
53                          self.f.remove(state)
54                      self.v.remove(state)
55                      del(self.delta[state])
56                      break
57              else:
58                  self.register.add(state)
59
60      def add_suffix(self, state, current_suffix):
61          """Add the current_suffix to the graph from state and return it"""
62          nodenum = max(int(w[1:]) for w in self.v)+1
63          suffix = []
64          for c in current_suffix:
65              newnode = 'q{}'.format(nodenum)
66              self.v.add(newnode)
67              nodenum += 1
68              self.delta[state][c] = newnode
69              self.delta[newnode] = {}
70              suffix.append((state, c, newnode))
71              state = newnode
72          self.f.add(newnode)
73          return suffix
74
75      def add_words(self, words):
76          """Add words to the dawg"""
77          self.register = set()
78          words = sorted(words)
79          while words:
80              word = words.pop()
81              common_prefix = self.common_prefix(word, self.v0)
82              last_state = self.delta_star(self.v0, common_prefix)
83              current_suffix = word[len(common_prefix):]
84              if current_suffix:
85                  suffix = self.add_suffix(last_state, current_suffix)
86                  self.replace_or_register(suffix)
87              else:
88                  self.f.add(last_state)
89
90      def to_dot(self, options=''):
91          """Return the graphviz(dot) string representation of the graph"""
92          s = 'digraph {{\n{}\nn0 [style=invis]\n'.format(options)
93          for node in self.v:
94              s += '{} [shape={}circle]\n'.format(
95                  node, 'double' if node in self.f else '')
96          s += 'n0 -> {}\n'.format(self.v0)
97          for node, transitions in self.delta.items():
98              for letter, state in transitions.items():
99                  s += '{} -> {} [label="{}"]\n'.format(node, state, letter)
100         return s + '}\n'
```

Bibliography

[1] Sharon Adler, Alex Milowski, Jeremy Richman, Steve Zilles, et al. Extensible stylesheet language (xsl)-version 1.0. 2001.

[2] RSS Advisory Board. Rss 2.0 specification. *Web available*, 2007.

[3] Tim Bray, Jean Paoli, C Michael Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible markup language (xml). *World Wide Web Consortium Recommendation REC-xml-19980210. http://www. w3. org/TR/1998/REC-xml-19980210*, 1998.

[4] Jan Daciuk, Stoyan Mihov, Bruce W. Watson, and Richard E. Watson. Incremental Construction of Minimal Acyclic Finite-State Automata. *Computational Linguistics*, 26(1):3–16, March 2000.

[5] Jan Daciuk, Bruce W Watson, and Richard E Watson. Incremental construction of minimal acyclic finite state automata and transducers. In *Proceedings of the International Workshop on Finite State Methods in Natural Language Processing*, pages 48–56. Association for Computational Linguistics, 1998.

[6] Roy T Fielding and Gail Kaiser. The apache http server project. *Internet Computing, IEEE*, 1(4):88–90, 1997.

[7] John Hopcroft. An N log N algorithm for minimizing states in a finite automaton. Technical report, 1971.

[8] John E Hopcroft. *Introduction to automata theory, languages, and computation*. Pearson Education India, 1979.

[9] Stoyan Mihov. Direct Building of Minimal Automaton for Given List 2 Formal background and notations 3 Method description. pages 1–6, 1998.

[10] Mark Nottingham and Robert Sayre. The atom syndication format. 2005.

[11] G. Van Rossem and F.L. Drake (eds) Drake. *Python Reference Manual*. PythonLabs, Virginia, USA, 2001.

[12] G Trubetskoy. mod python: Apache/python integration.