Conception and Implementation of Web Analysis for Web Navigation

Table of Contents

1. **Introduction** 5
1.1. Background 5
1.1.1. Graphs 5
1.1.2. Analyzing Web Structure 7
1.1.3. Context and Hyperstructures 9
1.2. Goals of the Project 10
1.3. Organization of the Document 10
2. **Fundamentals** 12
2.1. TheWorldWideWeb in the Large 12
2.2. Web Technologies 14
2.2.1. TCP/IP 14
2.2.2. DNS 14
2.2.3. Universal Resource Locators 16
2.2.4. Web Browsers 16
2.2.5. Cookies 18
2.2.6. HTML and Cascading Style Sheets 18
2.2.7. XML 20
2.2.8. DHTML 22
2.2.9. Server-Side Scripting 23
2.2.10. Perl, Python, and PHP 25
3. **Existing Tools** 26
3.1. Website Nightmares 26
3.2. Checking Explained 26
3.3. Available Free Link-Checking Software 27
3.3.1. HTML Interface Only 28

3.3.2. C ... 28
3.3.3. Java ... 29
3.3.4. Perl ... 29
4. PROPOSED TOOL ... 31
4.1. LINK EXTRACTION TOOL SPECIFICATION ... 31
4.1.1. Requirements ... 31
4.1.2. Features ... 31
4.1.3. Installation and Usage ... 31
4.2. DESIGN AND CODING ... 33
4.2.1. High-level Design Overview ... 34
4.2.2. Structure of Python Scripts ... 35
4.2.3. Lists and Dictionaries ... 36
4.2.4. Functions ... 37
4.2.5. Modules ... 37
4.2.6. Classes ... 38
4.2.7. Built-in Constants ... 38
4.2.8. System Modules ... 39
4.2.9. UML Component Diagram ... 40
4.3. ALGORITHMS ... 41
4.3.1. webcheck.py ... 41
4.3.2. crawler.py ... 42
4.3.3. Plugins ... 48
4.3.4. webcheckparse.py ... 50
4.4. CREATEGML ... 55

5. Conclusion and Future Works 56

5.1. Summary 56

5.2. Conclusion 56

5.3. Future Work 57

6. References/Bibliography 58

1. INTRODUCTION

This chapter introduces some of the basic notions and concepts in Web analysis for Web navigation. It presents graphical models of the dependencies between the documents on the Web and lays the theoretic framework for the research. Also, this chapter defines the goals of the project.

1.1. BACKGROUND

The WorldWideWeb is an enormous collection of completely uncontrolled heterogeneous documents. Documents on the web are constantly changing, which shifts the scope of possible links. This ultimately changes what we can infer *about* a document.

One of many goals in web site publishing is the effective communication of information. But in order to effectively communicate, one often has to understand the structure of the information to be communicated and develop a body of works with a similar structure. Moreover, the structure of a body of works often reveals information about the relative importance of its components. In this chapter, we shall develop a framework for effectively analyzing bodies of documents and their relationships.

1.1.1. Graphs

A *directed graph* `G` is an ordered pair `(V, E)` where `V` is a finite set of *vertices* and `E` is a set of ordered pairs whose elements are taken from `V`. Suppose `x` and `y` are elements of `V`. Elements of `E` are called *edges*, and if `(x, y)` is an element of `E`, we say that `x` *points to* `y`. The *degree* of a vertex is the number of edges pointing to it.

We call a vertex a *terminal vertex* if it does not point to anything. If `V'` is a subset of `V` and `E'` is a set of edges in `E` connecting points in `V'`, then `(V',E')` is a *subgraph* of the graph `G`. Figure 1-2 is a graphical representation of the graph `G` with `V = {a, b, c, d}` and `E ={(a, b), (a, d), (b, c), (b, d)}`. Note that `c` is a terminal vertex. The graph with `V' = {a, b, c}` and `E' = {(a, b), (b, c)}` is a subgraph of `G`. Undirected graphs are similar to directed graphs. A graph is *undirected* if for each ordered pair `(x, y)` in `E`, `(y, x)` is also an element of `E`. In an undirected graph, we say that `x` *is connected to* `y` if `(x, y)` is an element of `E`. This notion of connectedness is similar to the intuitive notion. For example, if

a rope is connected to a boat, the boat is connected to that rope. This is why we stipulate that if `(x, y)` is in `E`, then so is `(y, x)`.

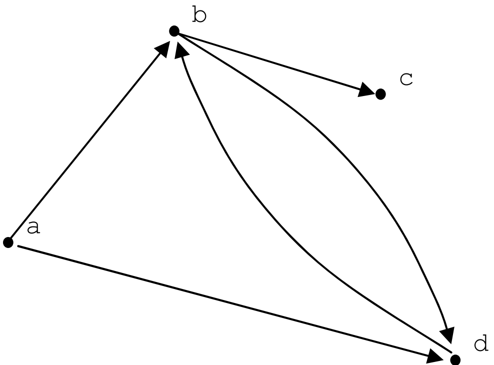

Figure 1-1 A directed graph

We can distinguish between two broad categories of directed graphs. The class of *linear* graphs consists of graphs which have a single terminal vertex and for which each other vertex points to only one other. Note that a graph with a single vertex is a linear graph. Figure 1-2 is an example of a linear graph.

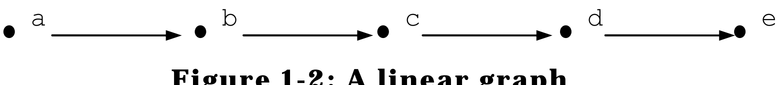

Figure 1-2: A linear graph

Branching graphs are those that are not linear graphs. Note that for a graph to be a branching graph, there must be a vertex that points to at least two others. A *path* between any two points in a graph is a linear subgraph containing the two points. Similar concepts can be defined for undirected graphs.

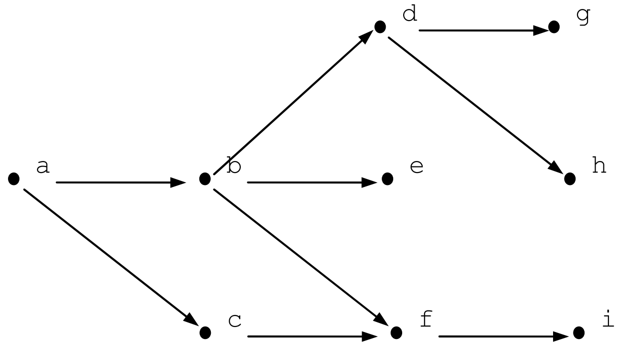

Figure 1-3: A branching graph

Directed graphs play a vital role in understanding the structure of a document or the body of documents. For example, we can view a book as a set of pages with a natural order determined by the content. The content printed on one page should follow the content printed on the page preceding it. Depending on the purposes of the model, a linear graph model might be a good approximation. Branched graphs can be used to model interactions between humans and computers in a graphical user interface. On screen, menus could be modeled by vertices, and each choice allowed to the user would correspond to an edge.

A *weighted* graph is a graph with a scalar value associated to each edge. These are useful to understand the structure of books and other similar documents. It is often the case that a book cites outside sources, but the reader is not expected to immediately go to those sources. The sources are only meant to enrich the reader's understanding of a book's material. A weighted graph is one way to model the relative importance of such choices. The double arrows in Figure 1-4 are to be interpreted as highly weighted arrows, whereas the single arrows are to be interpreted as lightly weighted arrows. Vertices `a` to `e` might be chapters in a book, and vertices `f` through `i` would be references. Since arbitrary scalars are allowed to be associated with each edge, we might wish to weigh one reference more than another.

One reason weighted graphs are important is because they allow us to simplify the analysis of multiply linked documents. Heavily weighted linear subgraphs of a particular graph can be taken to be a single document. Weighted graphs are often used by Internet search engines to efficiently rank the popularity and utility of web sites.

1.1.2. Analyzing Web Structure

Most methods for electronic communication, such as electronic mail and chat programs, extend the ability for its participants to interact with one another. In contrast, web-based communication is primarily an extension of the presentational mode of communication. Most web sites are modified only by their administrators. Relatively few web sites allow the user rich interaction.

Several web pages together form a web site. Hyperlinks can be categorized as either internal or external. A hyperlink is an *internal* link if it takes the user to a resource

within the web site, and is an *external* link otherwise. An internal link might even take the user to a different section of the same document.

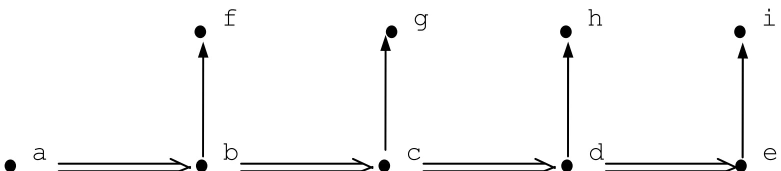

Figure 1-4: A weighted graph graph. The double arrows are to be interpreted as having more weight than the single arrows, as in a book with references.

There are only three ways for a user to move through web based information. Users can either type in an address they already know, or scroll through a document, or activate a hyperlink. The first two are under the direct control of the user, but the third is also under the control of the author. The user cannot control the placement of hyperlinks, nor the locations to which they point. The inclusion of a hyperlink in a web page is deliberate, and most often specifically created by the web designer. The presence of a link indicates a communicative choice by the web designer. This insight motivates much analysis of web structures. Differences in structure are representative of differences in communicative agendas. The relative importance of a hyperlink within the body of a document is indicative of the author's priorities and predispositions.

The World Wide Web can be analyzed on several levels. The document level considers the structural properties of documents on the World Wide Web. A clique level analysis considers the relationships between relatively small subgraphs of the WorldWideWeb. A systems level analysis considers the connections between all vertices in the World Wide Web.

The graph theoretic framework presented has little to say about the first level, but has many ramifications for the last two levels. As a graph grows, the methods used to analyze them change considerably. It is unfeasible to perform computationally intensive calculations on gargantuan graphs, but perfectly reasonable to do so on relatively small graphs. Search engines apply clique level analyses to the World Wide Web to determine the ranks of web pages relative to a set of key words. Statistical methods are often employed to analyze the structure of the World Wide Web in the large.

1.1.3. Context and Hyperstructures

If the pages of a book were to be in disorder, each page would lose its context. A similar consideration makes the analysis of multiply linked documents much more complicated. Web sites are often written as a single, long document and split into smaller web pages to aid in organizing the content. However, these web sites often refer to outside works via a mechanism known as *hyperlinking*. Modern web browsers default to replacing the content on screen with new content when a hyperlink is activated. When navigating a heavily weighted path, there are no unexpected effects. It is as if one turns the pages of a book. But when one activates a hyperlink to an external resource, all context is lost.

In the past few years, there has been work on designing models of communication that allow modular resource implementation to deliver more flexible ways to interact with information, and to maintain the context of that information. Some of these models are referred to as zzStructures, mSpaces, and polyarchies. We shall restrict ourselves to examining zzStructures. It turns out that zzStructures are general enough to subsume the other two.

A *directed multigraph* is a directed graph whose pairs of vertices can have multiple edges between them. Distinct edges between two vertices are identified by *color*. A *zzStructure* is a directed multigraph with colored edges such that every vertex can have at most one incoming edge of each color, and at most one outgoing edge of each color. The essential role of the colors in the definition is to sort edges into distinct incompatible contexts.

The goal is to eliminate redundancy. Let us direct our attention to Figure 1-5. The distinct arrows are meant to indicate differing colors. We may assume that each hyperlink preserves context. In the zzStructure in the figure, resources `c` and `d` are shared by the resources generated by the two colors. For example, we might wish to model the relationship between the sections in this document as a simple linear graph. However, some of the sections are independent from the rest and could be used again in another document without modification.

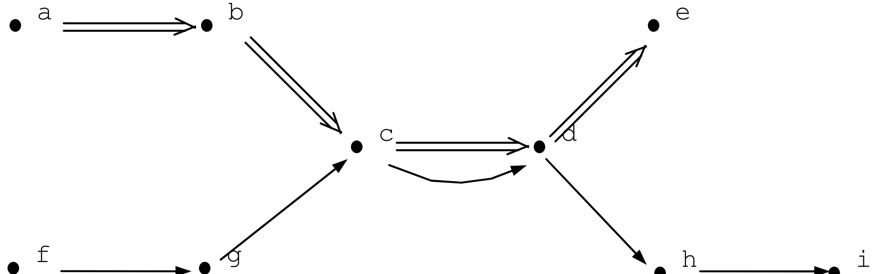

Figure 1-5: A simple zzStructure. Each vertex corresponds to a resource, and the different arrows represent colors. Resources c and d are shared between the two colors.

1.2. Goals of the thesis

The goals of the thesis can be identified as follows:

- Explanation of Web structures and the technologies that are necessary in order these structures to exist
- Description of the existing software for link checking
- Description of the proposed tool for link checking and extraction of the links
- Analysing the generated links and creating a GML document that can be given as an input for the VisuGraph tool.

1.3. Organization of the thesis

This thesis document is divided into five chapters, an abstract and a reference chapter and discusses the concepts and implementation of Web Analysis for Web Navigation. Each chapter addresses a particular aspect of the topic as follows:

Chapter 1

The first chapter introduces the theoretic framework, provides the motivation for the thesis and depicts the goals and the research methodology. It also explains the organization of the document.

Chapter 2

Chapter 2 sets the foundation of the thesis. This chapter further describes the theoretic framework that is important to understand the concepts involved in this document work.

Chapter 3

Chapter 3 explains what link checking is and deals with the existing tools for link checking. Several such tools, written in different programming languages are reviewed, together with their advantages and disadvantages.

Chapter 4

Chapter 4 discusses the proposed tool as a way to extract links from a website, and use the text file with the extracted links as an input to another tool – createGML to create a GML document with the structure of the site.

Chapter 5

The final chapter in the document provides conclusion with a brief summary, an outlook of related future research work aimed at improving the proposed tool.

2. FUNDAMENTALS

This chapter gives an overview of some of the technologies that are behind the Web. Among the fundamental concepts discussed in this chapter are the methodologies to estimate the size of the Web, basic Web technologies like protocols used (TCP/IP), naming system (DNS, URLs), Web browsers and their specifics, some of the markup (HTML, DHTML, XML) and programming languages used in Web development.

2.1. THE WORLDWIDEWEB IN THE LARGE

zzStructures, discussed in the previous chapter, are not the most suitable models of the current World Wide Web. Few web sites are designed to be modular in the sense described above, and even fewer are involved in a coordinated effort to create reusable resources with others. The Semantic Web by the World Wide Web Consortium is one such project.

We will study the World Wide Web from a graph theoretic point of view. There are several reasons to do so. First, doing so can help in designing algorithms to crawl the World Wide Web, can help our understanding of the sociology of content creation on the World Wide Web, and allows us to predict the emergence of new trends.

In a recent experiment, Barb´asi attempted to measure the diameter of the World Wide Web. He used the following approach: let us call a path a *short path* if it is the shortest path between two vertices. The *diameter* is the number of edges traversed along the longest short path on the World Wide Web. This definition holds for all graphs. Barb´asi used statistical methods to do so. They used a *crawler* to follow all the links on a starting page, then all the links on each paged reached from the last, and so on.

The crawler began on the Notre Dame web page and gathered information on 325, 729 documents and 1, 469, 680 links. The most important step in the analysis of these data was to calculate the probability that a page has degree n and the probability that a given page links to n pages. Barb´asi found that both probabilities obey a power law. The probability that a page links to n other pages is proportional to $n^{-2.45}$, and the probability that it has degree n is $n^{-2.1}$. Statistical calculations centered around these power laws revealed that the diameter of the

World Wide Web was probably 19 edges. Broder et al. confirmed the power law. A paper by Bornholdt indicates that this may be an instance of a larger phenomenon.

The power law implies that pages with few inwards and outwards links are the most numerous. Pages with relatively high degree are rare, but they tend to connect parts of the World Wide Web that would otherwise be very far apart.

Statistical methods can yield more information regarding large graphs. Large graphs tend to be sparse. A graph with *n* vertices can have at most

$$\frac{n(n-1)}{2}$$

edges, but large graphs in the real world have on the order of n edges. Web pages also tend to be clustered. Two pages linked to a third have a higher probability of linking to each other. Several mechanisms might explain this. These clusters are the basis for recent advances in search technologies. Large, real world graphs also tend to have a small diameter.

These graphs share many properties with the World Wide Web, and much recent work in the field has focused on generating random graphs with these properties in order to predict the behavior of the World Wide Web. A random graph is obtained by starting with a set of vertices and adding vertices between them at random. The most commonly studied random graph is called $G(n, p)$ and includes each possible edge independently with probability p. While these graphs tend to satisfy the three properties above, Kumar et al. designed models of stochastically evolving random graphs that agree with empirical data better than models using $G(n, p)$. Kumar et al. used a model where at each discrete timestep, a random number of new vertices and edges. These stochatic models required a lot of mathematical machinery to develop, and so are beyond the scope of this manual.

According to a study conducted in 2001, there were more than 550 billion documents on the web. Most of these are not directly accessible. As we shall soon see, many modern resources require a multitude of documents to implement. These documents constitute what is called the *Deep Web*. A 2002 survey of over two billion web pages determined that most web content was in English, with 56.4%. Next were pages in German with 7.7%, French with 5.6%, and Japanese with 4.9%. These numbers are no longer accurate because of recent growth of Chinese web sites.

2.2. Web Technologies

In this section we will explore various technologies that allow us to implement the web structures previously introduced.

2.2.1. TCP/IP

The fundamental basis underlying the Internet is the TCP/IP suite of communications protocols. A protocol is a set of rules determining the ways agents may interact. The TCP/IP protocol suite describes how communication is to occur at various levels of abstraction. TCP/IP is a suite of protocols enabling computers to communicate over physical networks. The TCP and IP protocols are among the most important, but there are many others.

In 1969, the Advanced Research Projects Agency created an experimental network known as *ARPAnet*. The experiment proved to be a success, and in 1975 the ARPAnet was converted into an operational network. In the early eighties, the TCP/IP protocols were adopted as US Military Standards, and all machines on the network were required to convert to the new protocols.

Because it is an open standard, TCP/IP is independent from any specific physical network hardware. This allows TCP/IP to integrate many different kinds of networks. TCP/IP also affords the Internet a common addressing scheme that allows any device to quickly find any other. This addressing scheme is implemented in the IPv4 protocol. Each device is given an *IP address* which uniquely identifies it.

2.2.2. DNS

IP addresses serve as referents for devices on a network, but remembering them is quite difficult. In early Internet history, few devices were networked. Users designed and implemented a system to associate IP addresses to simple mnemonics. It worked by translating *domain names* to IP addresses transparently. The hosts file held information regarding which host corresponded with which IP address. The Stanford Research Institute would periodically compile a table of all known machines and their domain names. Users would occasionally download a hosts file from the Stanford Research Institute to update their own hosts file.

However, as the Internet grew, it became apparent that this solution did not scale well. As more devices connected to the network, the hosts file grew, choking the

Stanford Research Institute's connection to the Internet. The hosts file was constantly out of date. A solution was urgently needed.

The Domain Name System is a distributed database that provides a physical location for each domain name on demand. Each user's computer runs a *DNS resolver*, which generates DNS requests on behalf of software programs. A DNS resolver generally queries *a recursive DNS server*. Recursive DNS servers search the Domain Name System on behalf of resolvers and if successful respond to each query with an IP address. *Authoritative DNS servers* are the source from which DNS information spreads.

The Domain Name System is quite complex. When a host attempts to resolve a domain name, it reads the domain name from right to left, querying name servers about the subdomain to the left. For example, if we try to resolve `www.example.com`, the host queries *a root server* regarding the `.com` domain. Domains like `.com` and `.net` are called top level domains. The root server points the client to the server to it, and the client queries for `example.com`. This information does not change often, so the modern DNS implementation includes caching so that the client does not have to visit a root server.

There are currently thirteen root servers world wide. *The Internet Assigned Numbers Authority* also oversees the creation of top level domains. The creation of top level domains is difficult. In particular, the creation of country specific top level domains such as `.de` or `.uk` is highly politicized.

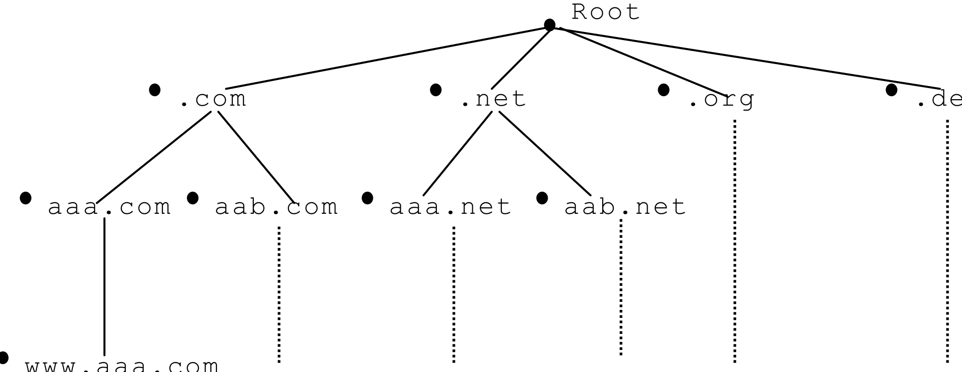

Figure 2-1: The organization of the Domain Name System. Each domain is associated with the namespace below it, in spotted lines. The vertices below a domain are elements in that domain's name space. Note that name spaces are related by inclusion. That is, every name space is contained by the ones above it.

The Internet Assigned Numbers Authority is under the control of the *Internet Corporation for Assigned Names and Numbers*, but is ultimately under the control of the United States Department of Commerce. There have been several proposals to sever the relationship between the Internet Assigned Numbers Authority and the Internet Corporation for Assigned Names and Numbers, even if the government of the United States does not co-operate.

2.2.3. Universal Resource Locators

Broadly speaking, we can define a *server* as any machine that accepts queries from other machines and replies with information. However, the software programs that implement the protocols necessary for communication are also called servers. For instance, a software program called *BIND* is responsible for responding to DNS requests on most machines running a Unix-like operating system, and so is often called a DNS server itself. We will soon encounter several other kinds of servers.

The domain name system, as we have seen, allows us to uniquely identify and find other machines on the Internet. But this is not the whole picture. A single machine might serve many resources, using many different protocols. A scheme was devised for uniquely identifying and locating resources. A *Uniform Resource Locator*, or *URL*, is a standardized identifier for a resource. Moreover, a URL's syntax encodes enough information to locate the resource. In general, a URL is formed with a protocol name, followed by a colon and two slashes, the name of the machine serving the resource, and a path to the resource. For example, `http://slashdot.org/` is the URL for a popular web site.

2.2.4. Web Browsers

A *user agent* is a software program used with a particular network protocol. The phrase is commonly used to refer to programs that access the World Wide Web. A common example is a *web browser*, but there are other kinds of user agents for accessing the World Wide Web.

Web browsers interact with web servers primarily through the *HyperText Transfer Protocol*, or *HTTP*. Browsers often support other protocols, such as the *File Transfer Protocol*, or *FTP*; and the *Secure HyperText Transfer Protocol*. Most web documents are written in HTML, which we shall examine shortly. Most browsers

also support other file formats, such as the *JPEG* and *GIF* image formats and XML document format.

We can now give a short account of some of the processes involved in navigating a web site. A user runs a web browser. Let us suppose that he wishes to view the resource at `http://slashdot.org/index`. So the user inputs the address into his browser. The browser connects to the machine whose name is slashdot.org and asks for a document locally named /index. The machine replies with the file and the web browser renders a formatted copy. From here, suppose that the user clicks on and activates a link to `http://en.wikipedia.org/wiki/Main Page/`. The process that occurs is similar: the user connects to the machine named `en.wikipedia.org` and requests a resource named `/wiki/Main Page`. The server responds with the file and the user agent renders a formatted copy.

Early web browsers only supported a simple version of HTML. Proprietary web browsers extended and developed non-standard dialects of HTML. This lead to issues with interoperability. Modern web browsers are designed to conform to the HTML 4.01 and XHTML 1.0 standards.

The first web browser was designed by Tim Berners-Lee in 1990, who also designed the first dialect of HTML. It was called *WorldWideWeb*. *NCSA Mosaic* was among the first browsers to enable the use of graphics embedded in web documents. It helped spur the popularity of the World Wide Web. Netscape Communications Corporation was founded by an NCSA Mosaic developer. Netscape released *Netscape Navigator* in 1994. Microsoft bought Spyglass Incorporated and released *Microsoft Internet Explorer* shortly after.

This initiated the *Browser Wars*. Both Microsoft and Netscape added proprietary extensions to HTML in order to attract new users, but by 1998, Netscape Navigator was generally seen as inferior. Its market share steadily decreasing, Netscape released the Navigator source code under a free software license. The popular free *Mozilla Firefox* is a Navigator derivative. America Online later bought the Netscape Communications Corporation.

There are several other browsers. *Lynx* is a popular text based browser used by people with old hardware and users with impaired vision. *Opera* was released in 1996 and is used in handheld devices. It is quite fast, but not very popular. Apple

released the *Safari* browser for its Macintosh line of hardware in 2003. It is based on KDE's *KHTML* rendering engine.

According to usage statistics compiled by Net Applications for PCWorld Magazine, Microsoft Internet Explorer was used 86.56% of the time, Mozilla Firefox 8.71%, America Online's Netscape 1.55%, Opera 0.59%, and Apple Safari 1.93% in June 2005.

Unfortunately no user agent conforms exactly to the World Wide Web Consortium's specifications for HTML or related languages. Even where browsers are compliant with the specification, each one will render a web page differently. Microsoft's Internet Explorer is particularly bad in both regards. In its zeal to win the browser wars, Microsoft released a version of Internet Explorer with faulty Cascading Style Sheet support. Because of its faults, pages that render correctly in standards compliant browsers may not render in Internet Explorer. Although there are ways to force Internet Explorer to render a layout, the modifications to the code make the document render poorly in more standards compliant browsers. If web designers wish to communicate to as large an audience as possible, they must make sure to test their design on as many browsers as possible to ensure that none are left out.

2.2.5. Cookies

An *HTTP magic cookie*, usually called a *cookie*, is a packet of information sent from a web server to a user agent and sent back to the server each time the user agent returns to that server. Cookies can contain arbitrary information. Cookies are often used by web sites to allow a user to remain logged in without requiring them to sign in again each time they access the site. Other uses include tracking desired products in an online store.

Cookies have important implications with regard to anonymity an privacy on the World Wide Web. Some web site surreptitiously place cookies that are tracked for marketing purposes. Companies using this mechanism justify it by claiming that it is an effective way to give consumers access to products they are likely to be interested in.

2.2.6. HTML and Cascading Style Sheets

HTML is an acronym for *HyperText Markup Language*. HTML is a markup language designed toward the construction of single page hyperlinked documents. Recall that

a hyperlink is a symbolic pointer from one resource to another. Documents written in HTML must be rendered by specialized rendering software called *HTML user agents*. A familiar example is *a web browser*, such as Microsoft's Internet Explorer or the Mozilla Foundation's Firefox. The HTML standard is codified by the *World Wide Web Consortium*, also called the *W3C*. Currently, the W3C endorses HTML version 4.01.

The purpose of a *markup language* is to allow a content author to easily encode metadata. Usually, this serves to separate content from its presentation. The author only must write the content and appropriately mark it with *HTML tags* which provide the HTML user agent information regarding how the document should be rendered. Early versions of HTML included tags that controlled exactly how a document should be rendered, but newer versions attempt to lift the burden of mixing content and presentation from the author's shoulders by including tags that provide the HTML user agent information regarding the semantics of elements.

The World Wide Web is quickly becoming a place where dozens of people might work on a single web site. It is unreasonable to expect each content author to understand the intricacies of page layout or the details of a web site's implementation, yet consistency in layout and style is often desired. The W3C designed *Cascading Style Sheets* to help further separate a content author from the details of implementation. Cascading Style Sheets take advantage of semantic and logical tagging to organize and group information and present a coherent layout uniformly. So long as content authors apply a consistent scheme for semantic tagging across their bodies of work, a web site will maintain a consistent visual layout on all its documents. Designing such an implementation requires meticulous planning to determine what sorts of structural elements require abstraction.

A well designed implementation of Cascading Style Sheets will obviously let content authors focus on writing content instead of worrying about the visual appearance of their work, but Cascading Style Sheets are designed to help web site designers easily change the visual appearance of a web site uniformly. Instead of including cascading style sheet information in each HTML file, a web site can store a single copy in a publicly accessible place and including the location instead. By modifying a single file, web designers can change the visual appearance of every document in their site. Before the invention of Cascading Style Sheets, web designers would have to go

through and edit each file individually. Clearly, such a project would become more time consuming as a web site grew.

2.2.7. XML

XML, also known as the *Extensible Markup Language*, is a schematic markup language for text documents. It defines syntax used to mark data with simple tags. Every datum in an XML document is a text string. The markup tags act as metadata and describe the data. XML markup is meant to describe the logical and semantic structure of a document. Because of its simplicity, XML offers the possibility of cross-platform, long-term data storage formats. It is often the case that a document written on one platform cannot be read on another. But XML files are encoded as plain text files, which is a nearly universal file format. XML documents can be read with any tool that can read a text file.

The XML specification describes the syntax exactly. The syntax is similar to that of HTML documents, but there are fundamental differences. XML is a schematic markup language. This means that XML does not have a fixed set of tags. Instead, XML allows developers to create the tags they need as they need them. A datum, together with surrounding markup, is called an *element*.

The XML specification precisely describes the syntax and grammar. Elements must fit a standard scheme. Elements can only be placed in certain places and orders. The grammar is precise to facilitate machine parsing. A document that satisfies the specification is called *well-formed*. XML parsers reject documents that are not well-formed.

It is often desirable to form an ontology, or a fixed set of common definitions, while working on a particular problem. In the parlance of XML, such an ontology is called an *application*. The markup permitted in an application can be codified in a schema. Documents can be compared against the schema. Those that satisfy the schema are called *valid*. Documents that do not satisfy the schema are called *invalid*. Note that this issue is independent of a document's well-foundedness. A document can be well-formed but still be invalid relative to a particular schema.

There are many different schematic languages. The most broadly supported language is the *document type definition* language. A document type definition includes all the supported elements and defines how and when they may be included

in a document. However, the document type definition has syntactic limitations that make it unsuitable for some applications.

XML is a derivative of the Standard Generalized Markup Language. The Standard Generalized Markup Language was designed to solve many of the same problems XML was intended to solve. It too is a semantic and structural markup language for text documents. The Standard Generalized Markup Language's biggest success was HTML, which is an application. The SGML is a very complicated language. Its official specification is over one hundred and fifty pages long, and covers many special and unlikely cases. Almost no software has ever implemented the specification fully. XML was intended to be a straightforward schematic markup language.

We shall present two relevant examples of XML applications. The first is *XHTML*. It is an XML application based on HTML 4.0, and serves the same purpose as HTML. XHTML was developed to allow web developers the use of powerful XML tools. The second application is GraphML. *GraphML* is a graph markup language. Recall that a graph is completely determined by a set of vertices `V` and a set of *edges* `E`. These edges, in fact, are ordered pairs, and are a relation between vertices. GraphML uses `<node>` tags to declare the vertices, and `<edge>` tags to declare the relationships between vertices. For example, the code to capture the graph in Figure 2-2 is given by:

```
<?xml version="1.0" encoding="UTF-8"?>
<graphml xmlns="http://graphml.graphdrawing.org/xmlns"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns
    http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd">
<graph id="G" edgedefault="undirected">
    <node id="a"/>
    <node id="b"/>
    <node id="c"/>
    <node id="d"/>
    <edge source="a" target="b"/>
```

```
    <edge source="a" target="d"/>
    <edge source="b" target="d"/>
    <edge source="b" target="c"/>
</graph>
</graphml>
```

The first section declares the version of the XML syntax the document will use, and the second gives information about the application. In particular, it gives a location where the user agent can find a specification of the schema. These sections are common to all XML documents. The `<graph>` section actually encodes the graph. Note that each vertex is declared and given a name, and that the `<edge>` tag encodes the relationships between vertices. Since this is an undirected graph, it does not matter if we use `<edge source="a" target="b"/>` over `<edge source="b" target="a"/>`.

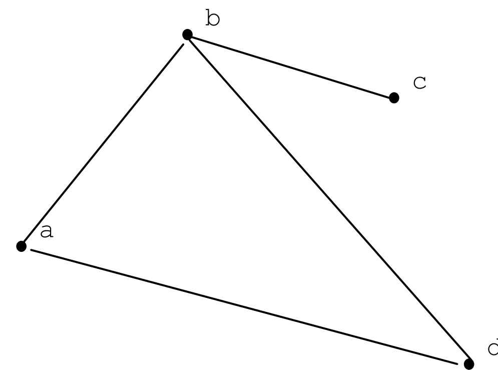

Figure 2-2: An undirected graph

GraphML syntax also accommodates the encoding of directed graphs, and includes syntax for encoding weighted and colored graphs

2.2.8. DHTML

Technologies such as *Java*, *JavaScript*, and *VBScript* allow *client-side scripting*. This refers to a class of programs executed by the user agent. Client-side scripts are usually embedded in an HTML file, as in the case of a Java program, can be contained in a separate file referenced by the documents that use it. Strictly speaking, Java executables are not scripts since they are pre-compiled. The distinction is unimportant in this context.

DHTML is not a language, but rather a combination of HTML or XHTML, JavaScript, Cascading Style Sheets, and the Document Object Model. The *Document Object Model* is an interface that exposes each element in an HTML or XHTML document to a scripting language. The Document Object Model allows a scripting language to manipulate nearly any part of an HTML or XHTML document, so that one can add, remove, or modify attributes or even entire elements.

DHTML is best suited for relatively trivial modifications to a document's structure. For example, one common use is to implement drop down menus in HTML and XHTML documents. A straightforward implementation would use JavaScript to modify the Document Object Model when a particular condition is satisfied. In particular, it would activate a previously inactive element when the mouse pointer crosses a certain threshold. Cascading Style Sheets would format and position the newly activated object. JavaScript would then modify the Document Object Model, removing the object when the mouse pointer was moved away from the menu.

While this mechanism can be used to dynamically change a document's visual layout and information content, the changes tend to be negligible from our perspective. In particular, changing a document's visual layout alone does not change the information content. We can treat a DHTML document with a drop down menu as we would treat a normal HTML or XHTML document containing the same links. This is simply an implementation issue. It does not change the page's relationship to other documents in any meaningful way. It is imprudent to use DHTML to change a document's information content significantly because the whole document must be downloaded in advance. Writing a document that can dynamically present the user with many large sections of information is wasteful because it is likely that the user will not be interested in most of it.

2.2.9. Server-Side Scripting

The technologies presented so far are only suited to present the user with static content. There is little scope for interaction. Resources only change when content authors publish modified versions of their documents. A client cannot directly cause the creation or modification of resources. Client-side scripting allows some interaction, but it is generally limited to changes in presentation. Newer technologies must be used to provide the user with a dynamic experience.

Modifications were made to web server software packages to allow to to interface with external programs. The *Common Gateway Interface*, also called *CGI*, was among the first to emerge as a way to present dynamically generated web documents at a user's request instead of being written in advance. CGI was invented by the United States' National Center for Supercomputing Applications for the NCSA HTTPd web server in 1993.

CGI codifies a standard for passing data between the user agent and the server. CGI allows a client user agent to request data from a program executed by the web server on the behalf of the user. The CGI standard abstracts many implementation details from the web server, so that CGI programs can be written in any computer language. In practice, most CGI programs are written in Perl.

From the point of view of a web server, CGI works by defining certain classes of URLs to be served by a CGI program. Whenever a request to a matching URL is received, the CGI program is executed. Any data sent by the client is taken as input by the CGI program. The web server takes any output from the CGI program and sends it to the client. The web server, in effect, acts as an intermediary between the client and an arbitrary executable.

The implementation of CGI leaves much to be desired. In particular, the web server starts a new copy a CGI program every time a user requests a resource controlled by that program. This is particularly taxing when working with scripting languages such as Perl. Scripted programming languages require interpreters to convert plain text scripts into machine readable code while the program is being run. Scripted languages tend to execute more slowly than compiled languages, because compilation occurs while the program is running. Pre-compiled programs also begin useful computations sooner than scripted programs, because it is necessary to load an interpreter into memory before starting a scripted program.

Because of this limitation, even a moderate workload can overwhelm a web server. More efficient server-side scripting mechanisms have been devised, however. *mod_perl*, for instance, embeds a Perl interpreter into the Apache web server. This allows Perl scripts to run in response to incoming requests without the overhead of starting a new instance of the Perl interpreter for each request. *mod_perl* was written to be backwards compatible with the CGI environment. Existing Perl CGI scripts can benefit from *mod_perl*'s significant improvements without being rewritten. Similar modules exist for other scripting languages.

2.2.10. Perl, Python, and PHP

Perl is an interpreted procedural scripting language designed by Larry Wall. Perl borrows features from C, scripting languages such as sh, awk, and sed, and many other programming languages. Perl is intended to be a practical language, and has grown to keep pace with programmers' needs. It supports procedural and object oriented programming, and has built-in support for efficient text processing. Perl's implementation for computing regular expressions is particularly powerful. These features make Perl ideally suited for server side scripting.

Perl has many varied applications. As mentioned before, it is often used in CGI applications, and is part of the *LAMP* platform for web development. LAMP consists of a computer running the Linux operating system running an Apache web server and a MySQL database server and using Perl, Python, or PHP to make them interoperate.

Like Linux and MySQL, Perl is *free software*, which means that it can be used, copied, modified, and redistributed freely. Moreover, they are available gratis on the Internet.

Python is an interpreted, interactive scripting language. It was created by Guido van Rossum in 1990. It is similar in spirit to Perl. Python is a multi-paradigm language. Like Perl, Python attempts to avoid tying a programmer to a particular style of programming by providing the ability to program using many different paradigms. Python shares many features with Perl, but Python's creators have tried to avoid the unnecessarily complex syntactic constructs often found in Perl. Python is also free software.

PHP was originally designed as a set of Perl scripts, and was later rewritten in C as a set of compiled CGI binaries by Rasmus Lerdorf in 1994. Lerdorf later combined it with an interpreter to ease the construction of web pages. The core of PHP was rewritten yet again, to produce the Zend Engine. The Zend Engine functions as an interpreter for the language and compiles PHP scripts as they run. PHP is now the most popular server-side scripting language on the World Wide Web. It is designed to process HTML out of the box, and a large body of libraries for extending PHP exists. It easily integrates with databases like MySQL and PostgreSQL.

3. EXISTING TOOLS

The purpose of this chapter is to reveal the issues in maintaining Web sites and the solution that link-checking software offers to these problems. Some of the basic terminology regarding link checking is explained as well. The main point of this chapter is the discussion of several existing link-checking tools (written in various programming languages).

3.1. WEBSITE NIGHTMARES

Ever since Tim Berners-Lee invented World Wide Web (WWW) in 1989 and HTML has become common markup language to exchange documents via WWW several issues appeared similar to real world handling with paper documents. As the pile of paper documents grows and becomes hard to manage (e.g. where a related paper is, is the information still valid, which information is internal and what is from external source, etc) same thing happens in electronic formats of documents, not only for HTML documents. As WWW is a special medium, allowing for so called hypertext references (now commonly known as URL or Uniform Resource Locator, subset of URI or Uniform Resource Identifier, or simply “links”), these references represent one of great challenges for webmasters (persons who maintain a web site, a collection of HTML and other documents and files like images, sound files, video etc) to keep them valid or even remove them if a resource, pointed to by reference, isn't available (“public”) anymore.

For helping maintain references, a special category of WWW software has born.

3.2. CHECKING EXPLAINED

Basic website checking in WWW terms describes a process of gathering a list of all hypertext references (“crawling links”) in HTML documents (web pages) of one website and then checking all list links if they are valid. Some common terms are used for various types and purpose of links:

- **sitemap** (or site map) – is a web page containing all or top level links of website
- **internal** – is a link which points to a web page on the same site
- **external** – is a link which points to a web page located on another web site
- **bad** – is a link which is no longer valid (e.g. the page has been renamed or removed)

- **old** – refers to a page which has not been modified for a long time
- **slow** – refers to a page that will probably take long time to display because of page size.

Note: Web analyzers are a different category of software. Its main purpose is to parse website access, transfer and error logs (usually access log and transfer log are combined to one) to find most visited pages, paths visitors have taken through the site, referrer sites (from which sites have visitors been redirected to our site), etc and can also produce an analysis of broken links (that do not link to an existing page) but usually only for internal, site links. Nevertheless, webmaster should have at least one application from both categories in his or her arsenal.

3.3. Available Link-Checking Software

For the purposes of this thesis, we shall concentrate only on link-checking software. The reason is that the tools (proposed tool and createGML), which will be discussed in the next chapter, are the implemented thesis work and because of this, it is fair to compare them to the other non-commercial software.

One aspect of link-checking software that has been skipped on deliberately because it goes out of the scope of this research paper is hardware requirements - for checking the same amount of links, different tools required more or less powerful hardware but since all of them can perform pretty well on an average machine (i.e. 2GHz CPU and 512 MB or more) hardware requirements are not a significant difference between them.

Also, since every link-checking software could put significant load on a Web server (especially a less powerful one) while crawling it to get the links and this can be interpreted as an attempt for a Denial of Service attack, we did not measure the exact speed at which links are collected. Even it is recommended, for those products that support it, for instance the proposed tool, to define a command line parameter (`-w n` or `--wait n`) to pause it for `n` seconds before it fetches another link.

Before getting to comparison of the features of the listed software and Linkchecker (the mentioned list resides on the website of this software) to the proposed tool, there are platform pros and cons for this category of software that also need to be clarified.

3.3.1. HTML Interface Only

The link-checking software that has a HTML interface only restricts the process of checking links to via HTML interface (browser) only, provided that the software is running on a server and additional scripts are deployed. No code is available for review etc.

Pros: simplicity, availability (if the Internet or local network connection is uninterrupted)

Cons: no code to review/modify, error correction

W3C checker (http://validator.w3.org/checklink/) is a website checker of this type. It is useful for a quick one web page check only.

3.3.2. C

The Web checking software in C seems most promising – every Unix version and Linux distribution is based on C thus networking and file handling libraries will be most efficient. Unfortunately, the open source community has not caught on for this type of software so products are still under development.

Pros: performance, efficient networking code, stable operating system libraries,

Cons: hard to maintain code, memory handling.

gURLChecker (http://labs.libre-entreprise.org/projects/gurlchecker/) is a C-based graphical website checker, written for the GNOME desktop environment. This software was not checked simply because it is not stable yet (as stated by three developers) and is at version 0.9.2, officially in beta (development) status. This is not necessary bad for open source software but if authors are fair it usually means that not all features are implemented so comparison is not fair with software already out of development phase, as the proposed tool is.

link-checker (http://ymettier.free.fr/link-checker/link-checker.html) is another C-based website checker. It has the same underdevelopment status as the previous one. Currently at version 0.1 with no future development roadmap - we do not recommend it.

3.3.3. Java

Java has been even more promising in the last year than the C community – libraries are free to use, mostly available from Sun corporation and features we seek (networking, file handling, arrays and objects manipulation) without doubt are optimized and stable. If there were not for the Java runtime "virtual machine" (JVM) needed and if we can cope with its memory requirements, this should be ideal.

Pros: performance, efficient networking code, stable operating system libraries.

Cons: JVM overhead and distribution requirement.

JCheckLinks (http://web.purplefrog.com/~thoth/jchecklinks/) development stopped in 1999. Its advantage is multithreading which is a feature of the Java libraries. But as we will see later, this actually is not nice for website owners. JCheckLinks unfortunately does not have a reports generator and thus only outputs two files: statuses and references. This is not practical information for most webmasters.

3.3.4. Perl

Perl comes with nearly every distribution of Unix and Linux. Its main value is the large set of libraries available on CPAN (http://www.cpan.org).

Pros: available code, mature and stable networking libraries, possible performance optimizations.

Cons: additional modules installation requirements, difficulties running Perl on non-Unix based platforms, speed, hard to maintain code.

checkbot (http://degraaff.org/checkbot/) is a regularly updated Perl script that could suffice for an experienced webmaster. Its main objection is that it only lists pages by HTTP error code returned, so one has to know HTTP protocol more than actually needed for web checking. Nevertheless, it can serve the purpose if the webmaster is not allowed to install Python. Also additional Perl modules must be installed to work correctly.

Checklinks (http://www.jmarshall.com/tools/cl/) has not been developed further after the year 2000. Just like checkbot, it is good if the webmaster only has Perl available to run scripts. While it is nice that no additional Perl modules have to be

installed for results to reside in a file, one has to redirect output to a file, which requires some Unix shell knowledge.

Dead Link Check (http://dlc.sourceforge.net/) or DLC is a simple Perl project that has not got past version 0.4 in development, with final version from year 2000. It does not check links of links (e.g. links in a linked page) so its value is low. DLC could be used for quick runs of checking one page, as it requires installing only one additional Perl module (`libwww-perl`) that is often already installed in the latest distribution of Linux for example.

linklint (http://www.linklint.org/) is a solid solution, another one of scripts that do not need additional Perl modules installed. This also means that it does not use optimized versions of available network and www libraries so top performance is not a feature of linklint. The same stands as for Dead Link Check – could be used for quick runs for limited checking and if Python is not available. Last linklint's version is 2.3.5 from 2001.

webgrep (http://cgi.linuxfocus.org/~guido/index.html - webgrep) was not analyzed because now it is an obsolete set of scripts. The webmaster has to combine calling of two of its scripts together to check for broken links not only for relative (site) links from base but also for other absolute (site and external) links.

4. PROPOSED TOOL

In Chapter 3 some of the existing tools for link-checking have been described. While some of them are still under development and are not suitable for real-life tasks, the proposed tool is stable and mature and is suitable for Web masters, who need to keep track of the links on their site. The architecture, together with some of the classes of the proposed tool, are the main focus of this chapter. After we discuss the proposed tool, we shall briefly review another tool – createGML – which uses the text file with the extracted links as an input to create a `GML` document with the structure of the site that can be used as an input for the VisuGraph tool.

4.1. LINK EXTRACTION TOOL SPECIFICATION

4.1.1. Requirements

- Python 2.3 or later version
- Operating system supported by Python (several brands of Unix, Linux, Windows, OS/2, Mac, Amiga and other platforms)

4.1.2. Features

The README file that accompanies the proposed link extraction tool lists the following features:

- view the structure of a site
- track down broken links
- find potentially outdated web pages
- list links pointing to external sites

4.1.3. Installation and Usage

Installation of the tool is simple. It is just downloading the distribution package (in a compressed GZIP TAR archive, e.g. the distribution file name is in form of `webcheck-major_version-minor_version-revision.tar.gz`) and unpacking files. One can optionally add a symbolic link to `webcheck.py` (Unix/Linux) or add a directory in the system path (Windows) so that the full path to the tool does not need to be specified for running the script. `Webcheck.py` is the

main script, written in Python, which calls all other subscripts to crawl a list of links for a specified website.

An example of a typical usage from the command line is:

```
webcheck.py  -o  /home/site/www/   http://www.example.com
```

where the destination directory for the report files is specified after the switch `-o` and followed by the URL of the website, which has to be analyzed. This produces several HTML reports; their number depends on the included plugins (parts of software code which enhance the functionality of the main software code). For the current version 1.9.4 the generated reports include the following:

- `index.html` – an overview of the crawled site (sitemap)
- `urllist.html` – the list of all URLs crawled during checking
- `images.html` – the list of all images found in website links
- `external.html` – the list of all external URLs found
- `notchkd.html` – the list of all not checked URLs (like `javascript:` and `mailto:` protocols)
- `badlinks.html` – the list of all unresolved links from crawled links gathered (includes external links)
- `old.html` – old or outdated pages (the number of days since last modification is set in `config.py`)
- `new.html` – recently updated pages (again, the number of days since last modification is set in `config.py`)
- `slow.html` – pages with long text and/or lots of images that display slowly (in `config.py`, specify the size of a page in kilibytes that is considered as the boundary between big and small pages)
- `notitles.html` – the list of all website pages that do not have a specified `<title>` tag
- `problems.html` – problems found and grouped by author (specified in the `<meta>` author tag)
- `about.html` – a more detailed view of used plugins that produced reports.

The command line options, displayed when running the tool with parameter –help are listed in the next table.

Table 1: linkMap ***Object Properties***

Parameter	Description
-x PATTERN	mark URLs matching PATTERN as external
-y PATTERN	do not check URLs matching PATTERN
-b	base URLs only: consider any URL not starting with the base URL to be external
-a	do not check external URLs
-q, --quiet, --silent	do not print out progress as the tool traverses a site
-d, --debug	set loglevel to LEVEL, for programmer-level debugging
-o DIRECTORY	the directory in which tool will generate the reports
-f, --force	overwrite files without asking
-r N	the amount of redirects the tool should follow when following a link, 0 implies follow all redirects.
-w, --wait=SECONDS	wait SECONDS between retrievals
-V, --version	output version information and exit
-h, --help	display this help and exit

4.2. DESIGN AND CODING

As mentioned above, the tool is written in Python. Therefore, in order to be able to understand its design and coding, some knowledge of Python is necessary. Although it should not be difficult to catch on Python for beginners to average programmers, concepts like indenting, functions, modules, classes and object-oriented programming should be overviewed before modifying/writing own Python scripts.

Below is a brief overview of the basic concepts in Python. Here is how Python FAQ introduces the language:

"Python is an interpreted, interactive, object-oriented programming language. It incorporates modules, exceptions, dynamic typing, very high level dynamic data types, and classes. ... Python is a high-level general-purpose programming language that can be applied to many different classes of problems. The language comes with a large standard library that covers areas such as string processing (regular expressions, Unicode, calculating differences between files), Internet protocols (HTTP, FTP, SMTP, XML-RPC, POP, IMAP, CGI programming), software engineering (unit testing, logging, profiling, parsing Python code), and operating system interfaces (system calls, file systems, TCP/IP sockets)."

Therefore, web checking software can be naturally built with Python as it has everything required in the distribution itself. While this paper is not a Python tutorial (it is recommended to read the **Python Tutorial**), the syntax of Python language statements used in the tool will be summarized as a quick introduction to next section covering algorithms.

4.2.1. High-level Design Overview

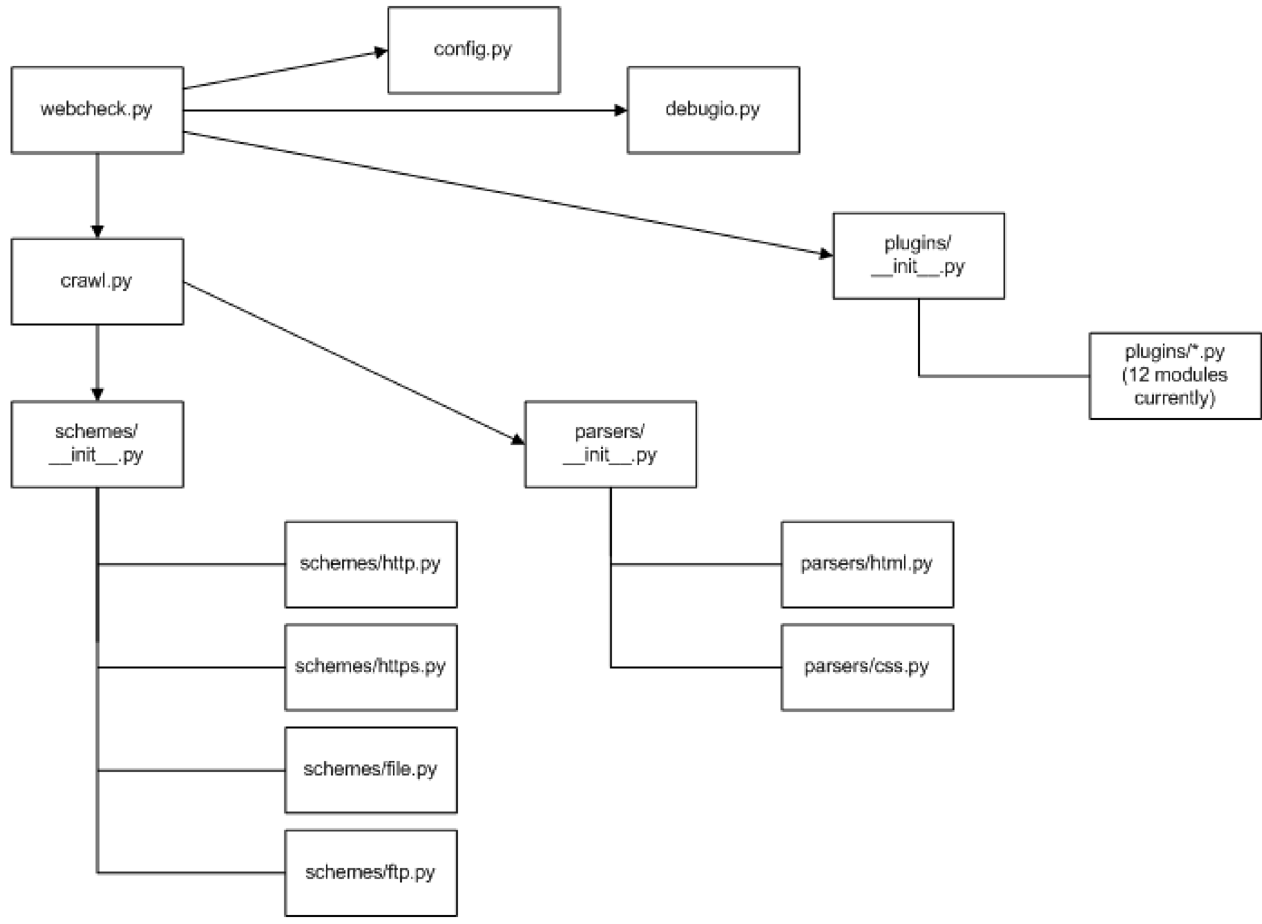

Figure 4-1: High level structure

The program's structure is modular and allows adding additional scheme, parsers or plugin modules by adding the appropriate module to one of subdirectories with general initialization code (schemes, parsers, plugins).

4.2.2. Structure of Python Scripts

Python syntax requires indenting of groups of statements and this is one aspect in which Python scripts are different from the scripts in similar languages that Python is compared with, like Perl and Java. For avoiding brackets usage (C, Java) and ambiguous placement of statements when there is wrong indentation (so it seems that one assignment statement belongs into a bracketed group having the same indentation – for example four space characters – but after closing bracket) Python developers decided to use indenting for grouping statements which define logical unit like executing statements if condition is met etc. Example:

```
if len(args)==0:
    print_usage()
    print_tryhelp()
    sys.exit(1)
mode='r'
```

The above script has two logical units – an `if` statement and a statement below it (mode ‘`r`’) that has the same indentation as the `if` statement etc.

Note: The above code is quite common in Python scripts and usually means that if no arguments were given when calling this script, the Python interpreter should call developer-defined functions `print_usage` and `print_tryhelp` and simply exit the script by calling the system *module's* (see later in text) function `exit`.

4.2.3. Lists and Dictionaries

Lists and dictionaries are two of the most extensively used data types and structures in the link extraction tool.

List is a compound data type that groups a list of comma-separated values or items between square brackets. Items do not need to have the same type. Example:

```
>>> PLUGINS = ["sitemap", "urllist", "images"]
```

The code above defines a list called `PLUGINS` with three items. Indices start at position `0` so `PLUGINS[0]` equals string `"sitemap"`.

One of the most often used functions in Python is the built-in function `len()` that returns the length of a string. For example,:

```
>>> len(PLUGINS[2])
```

outputs `6` (because the `"images"` string is six characters long).

Dictionary is another useful data type in Python. It represents an unordered set of key : value pairs where keys are unique (within one dictionary but can be the same in other dictionaries). An empty dictionary is created by a pair of braces like this:

```
self.linkMap = {}
```

but it's also possible to define initial key : value pairs by typing a comma-separated list of key : value pairs (this is meaningful for small amounts of data, larger would probably be read from file).

Mostly used operations on dictionary are setting or storing a value with some key and getting or extracting the value with the provided key. Once you have a dictionary object initialized, there are built-in methods to return a list of all keys used in the dictionary (`keys()`) and to check for the existence of single-key in the dictionary (`has_key()`).

4.2.4. Functions

Functions in Python are declared like this:

```
def <function_name>(parameter_1, parameter_2, …, parameter_n):
    """This function prints an average price of products."""
    <statements>
```

The keyword `def` is the start of the function definition. It must be followed by the function name, and a list of formal parameters in a parenthesized list. Statements start in the next line and must be indented. If the first statement is a string literal (*literal* is a notation for constant value of some built-in type), then it represents the function's documentation string or *docstring*.

Reminder: The statement following the function must be indented at the same level as `def` .

Note: If you are writing a function or any other statement interactively and want to end it (for evaluation of statements), you have to enter a blank line since Python interpreter will not know otherwise where the script or function ends.

4.2.5. Modules

By using Python interactively, all functions and variables are lost. So for any longer programs that we would like to maintain, we write *scripts* in files and run the Python interpreter by calling these scripts as a parameter in the command line. These files are called *modules*. They are useful for importing into other modules or into the *main* module (which is an entry point for executing your application).

Modules have an extension .py and within a module its name is available as the value of the global variable __name__. A module can contain executable statements and function definitions – these are executed only the first time the module is imported. It is common to import modules at the top of script:

```
import config

import crawler

import plugins
```

Modules are compiled when running and Python tries to write the compiled version to <module_name>.pyc. For optimized code one can run Python interpreter with -O switch which produces the <module_name>.pyo compiled script.

Note: An application does not run faster with compiled scripts (.pyc and/or .pyo), Python interpreter only loads such scripts faster.

4.2.6. Classes

Classes are named groups of statements. They usually wrap several functions but can have other statements (practically for private variables – which will not be visible to the main application). Definition:

```
class <Class_Name>:

    <statements>
```

The main usage of classes is to instantiate objects. We can write a code

```
x = OurClass()
```

which will create an instance of OurClass and assign this new object to the x variable. Now we can reference our object and all its functions (defined in class). Functions of the class instance objects are commonly called *methods* to distinct them from ordinary functions.

4.2.7. Built-in Constants

To understand the link extraction tool, we will need just three built-in constants. These constants are false, true and none and here are there definitions, taken from the Python Library Reference:

False

The false value of the `bool` type. New in version 2.3.

True

The true value of the `bool` type. New in version 2.3.

None

The sole value of `types.NoneType`. `None` **is frequently used to represent the absence of a value**, as when default arguments are not passed to a function.

4.2.8. System Modules

The tool uses various imported system modules for its working. These modules are as follows:

- **sys** – for getting command line arguments
- **urlparse** – parse URLs into components
- **urllib** – open resources by URL
- **robotparser** – parser for robots.txt website file
- **re** – for regular expression operations
- **time** – for time access and conversions.

We have seen before that modules are imported by the `import` statement. If you have many modules residing in the same directory, it is not necessary to import all by name - just import all directory modules with

```
import plugins
```

that (in the tool) imports all modules in the `plugins` subdirectory (so anybody can extend the tool by writing a module for the tool and placing additional ones into this subdirectory).

This short summary of Python definitions used in the tool's script should suffice for understanding the code even if you haven't seen any Python source code before. Following is the UML component diagram and in the next section the main algorithms (in two significant modules – main tool and crawler, the module for crawling web pages) of the tool are explained.

4.2.9. UML Component Diagram

Figure 4-2: A diagram of the components.

Note: Arrows represent dependent modules (e.g. `webcheck.py` **needs** `config.py` etc). Shaded squares represent modules residing in the same subdirectory of the parent webcheck directory and those libraries in Python core modules that are imported to the tool.

4.3. ALGORITHMS USED IN THE TOOL

Now when we have enough Python programming language background for the purpose of analyzing the tool's inner working, let's see the simple beauty of its main module, `webcheck.py`.

Note: The tool is well documented for nearly all lines or logical blocks of code though Python intermediate background is needed only for some advanced Python features. Therefore, pseudocode will be used in this paper for analysis of the most important methods in the tool but it is easy to go under the hood and examine actual code with pseudocode written here handy and comments inserted in the code itself.

4.3.1. webcheck.py

The algorithm is simple as it relies on other modules (actually, the `crawler.py` module is the hard worker in the distribution). Here is the main (as stated, the entry point or function of the module) pseudocode:

```
start script

import libraries

initialize site object as instance of crawler.Site class

parse command-line arguments

crawl site (adding link objects as instances of crawler.Link class) unless
interrupted by keyboard

for all plugins in plugin subdirectory

      generate plugin report in output directory

put stylesheet in output directory

end script
```

The proposed tool is very efficient due to the fact that Python language libraries itself are written in C language so native libraries are compiled and available on all

supported platforms (e.g. `DLLs` – Dynamically Linked Library – on Windows, `lib` files on Unix/Linux etc).

The time critical function is of course crawling of website entered for checking. We will see how crawling works in the next section.

The `Site` object has two properties: the first is `base` that is an URL that points to the base of the site, and the second is the `linkMap` object (dictionary data type), which includes a map of URLs to `link` objects. Let's examine the properties of `linkMap` by analyzing the `crawler.py` module where it is defined with its appropriate methods (object functions).

4.3.2. crawler.py

This module is the heart of the tool. Crawling websites must be efficient and internal memory representation must use the best available data type. We know from the `webcheck.py` module that the `site` object includes a `linkMap` object. Here are its properties from `crawler.py` with additional comments – we will need this information when examining the internal functions of this module.

Table 2: `linkMap` *Object Properties*

Property	Description from the author	Comment
`url`	the url this link represents	http://www.example.com/page.html?type=one
`scheme`	the scheme part of the url	http
`netloc`	the netloc part of the url	www.example.com
`path`	the path part of the url	/page.html
`query`	the query part of the url	type=one
`parents`	list of parent links (all the Links that link to this page)	commonly known as referrer pages
`children`	list of child links (the Links that this page links to)	or page references

Property	Description from the author	Comment
`pagechildren`	list of child pages, including children of embedded elements	explained in this section
`embedded`	list of links to embeded content	explained in this section
`depth`	the number of clicks from the base urls this page to find	e.g. number of links to visit to find this page from base
`isinternal`	whether the link is considered to be internal	
`isyanked`	whether the link should be checked at all	excluded with –y parameter at command line
`isfetched`	whether the link is fetched already	
`ispage`	whether the link represents a page	HTML or XHTML or server-parsed HTML
`mtime`	modification time (in seconds since the Epoch)	From January 1, 1970
`size`	the size of this document	
`mimetype`	the content-type of the document	type of the document
`title`	the title of this document	in title tag
`author`	the author of this document	in meta author tag
`status`	status of fetching this url (not `None` indicates a problem)	text like "url contains unescaped spaces: ..."
`redirectdepth`	the number of this redirect (`=0` not a redirect)	

So every link crawled will be represented by a `linkMap` **object that will include the URL itself plus all the properties above. Now let's see pseudocode of the crawler's module** `crawl()` **method for the** `site` **object (an instance of the** `Site` **class) which is called from webcheck module to get and parse all links found:**

```
initialize tocheck as empty list

for all elements of internal urls list    append url of element to tocheck
list, create Link object

while tocheck list is not empty

      select and remove one element from tocheck list

      if element's link is not to be crawled (is yanked) or link is already
fetched     do nothing

      else

            fetch element's link contents

            for every ordinary link found in element's link contents

                  if it's not yanked and not fetched and not already in
tocheck list

                        add it to tocheck list

            for every embedded link found in element's link contents

                  if it's not yanked and not fetched and not already in
tocheck list

                        add it to tocheck list

            wait for n seconds if parameter n (wait_between request) has been
defined

build the bases list (of base urls which have no parents), don't include base
url again

empty tocheck list, set link.depth in bases list to 0 and add bases links to
tocheck list

while tocheck list is not empty

      remove one element from tocheck list
```

append all children links (call _pagechildren method first to set depths of all children links) to tocheck list if they are not already included or have wrong depth

So the `crawl` method in the `crawler.py` module actually:

- first creates `Link` objects of all internal urls and adds them to site's `linkMap` dictionary (keys are urls, values are `Link` objects), adding urls to the `tocheck` list for checking
- second it loops until the `tocheck` list is empty and in the loop does this:
  - takes and removes one url from to the `tocheck` list and if it is not forbidden (yanked) or is already checked, fetches the page content of this url
  - it finds all links in the page content fetched and adds them to the `tocheck` list if they are not forbidden (yanked) or were already checked
- then it creates a bases list (of urls which don't have a parent so they form base of the site)
- and then again uses the `tocheck` list by emptying it first and inserting links of the bases' urls into it
- and again loops until the `tocheck` list is empty and in the loop does this:
  - takes and removes one url from the `tocheck` list
  - finds all children links at this url (`_pagechildren` method) and adds them to the `tocheck` list if they are not already included
  - the `_pagechildren` method also sets depth of link and finds all children links (ordinary, embedded and redirected), appending them to the `pagechildren` property of the `link` object

The end result of the `crawl()` method is the `site` object with a large `linkMap` object, filled with `Link` objects for every unique url of the site and each `Link` object has lists like parents and children (they are `Link` objects too), usual lists like `pagechildren` (urls) and properties like `title` and `author`, as shown in Table 2.

Now we know how links are crawled but not where and how every web page is fetched and parsed for its elements. We also do not know yet where links are found and what the difference between an ordinary and an embedded link is. This is the role of the `fetch()` method in the `Link` class.

The `fetch()` method's task is to make two things. First, it tries to find a suitable parser for the `MIME` type of the link with which the method was called. If it finds a suitable parser in the `parsers` subdirectory of the installation, it will use this parser to read the page and parse author, title and all page links.

Here is the pseudocode of the inner workings of the `fetch()` method in the `crawler.py` module:

```
if the link is not to be crawled (is yanked) - do nothing
else
      recognize scheme from link's scheme (http, https, file, ftp)
      if we don't have scheme module in schemes subdirectory -    do nothing
      else
            fetch content with appropriate scheme module
            if content is None (indicating redirect, error and HTTP code is
not 200) - do nothing
            else
                  find a parser for the content-type (MIME type)
                  if parser is found, parse content to Link object properties
```

The result of the `fetch()` method is thus completely parsed content parts of the current page. These are author, title, base URL, a list of embedded links and a list of children links.

Note: The tool has different protocols (schemes) supported so it is possible to check not only regular http-prefixed URLs but also https (SSL), file and ftp-prefixed urls. For all schemes there is a related scheme module in the `schemes` subdirectory of the tool's installation directory. Thus for example the `javascript:` scheme and the `mailto:` scheme links are found – and will be stated in "not checked" report - but will not be checked because there is no scheme module available for these schemes.

Now we know how links in `site` objects are populated, that this process works for several schemes supported and that pages are fetched and parsed to their elements. We will not go deeper for schemes because the `http.py` module is well documented – one useful piece of information here if you are analyzing logs with other software: for the tool's visits of pages you will see `User Agent: the tool in the` access log – but it is certainly interesting how parsing works e.g. what is considered as a link on a web page. This is a task of the `http.py` module, as seen from the pseudocode of its `parse()` method:

```
initialize parser as instance of _MyHTMLParser

parse url's content into parser, ignoring all errors encountered

if there are error messages for fetching content, add problem description to link object

if title was found by parsing, assign it to link's title

if author was found by parsing, assign it to link's author

assign link's url to temporary base variable

if base HTML tag was found by parsing, assign its link to base variable instead

for all embedded links found by parsing

      if embedded link is not empty

            clean its url by changing &#nnn character codes with their exact characters

            add it as link object in embedded list to current object

for all child links found by parsing

      if child link is not empty

            clean its url by changing &#nnn character codes with their exact characters

            add it as link object in children list to current object
```

This example deals with adding all links found to `link` objects, taking care of changes of decoding character codes to their exact character for example ";" is decoded to the character code ";".

The last part we have to see to globally understand the tool's code is the `_MyHTMLParser` class. This is simply a subclass of Python's internal `HTMLParser.HTMLParser` class with one change – overridden handling of exceptions that will occur only after 10 errors were found in parsing – and one addition of filling parser object properties with links found. Here is the `handle_starttag()` method's pseudocode:

```
if <title> tag is found      set parser.collect to empty string (it's None otherwise)

if <base href="url"> tag is found   set parser.base to href's url

if <link rel="type" href="url"> tag is found and type is stylesheet or icon

      add url to embedded list of links

if <meta name="author" content="name"> tag is found        set parser.author to name

if <meta http-equiv="refresh" content="0;url=address"> tag is found

      continue (it is not currently implemented)

if <img src="url"> tag is found     add url to embedded list of links

if <a href="url"> tag is found      add url to children list of links

if <frameset><frame src="url" …>…</frameset> tags are found

      add url to embedded list of links

if <map><area href="url"...>...</map> tags are found        add url to children list of links

if <applet code="url"...> tag is found    add url to embedded list of links

if <embed src="url"...> tag is found      add url to embedded list of links

if <embed><param name="movie" value="url"></embed> tags are found

      add url to embedded list of links
```

Subclassing an existing class works by overriding its methods. For overriding the `HTMLParser` **methods the most important method is how to analyze HTML tags from their start when doing the parsing process that is handled by the changed** `handle_starttag` **method described above. The other methods of the** `html.py` **module are not hard to understand.**

4.3.3. Plugins

`Webcheck.py` **calls various plugins, residing in the** `plugins` **subdirectory of the installation. When all links are parsed and residing in one large** `site` **object, it is easy to get reports. For example,** `sitemap.py` **is a simple script that writes unordered lists (** **HTML tag) of the links in the** `site` **object. Let's examine the**

recursive part of exploring and writing links to `sitemap.html` **(report), adding additional comments (two hash signs,** `##`**) to an existing code:**

```
if depth <= config.REPORT_SITEMAP_LEVEL: ## how many clicks followed
        # figure out the links to follow and ensure that they are only
        # explored from here
        children = []
        for child in link.pagechildren:
            # skip pages that have the wrong depth, are not internal or have
            # already been visited
            if child.depth != depth+1 or not child.isinternal or child in
      explored:
                continue ## e.g. continue for loop with next child
            # set child as explored and add to to explore list
            explored.append(child)
            children.append(child)
        # go over the children and present them as a list
        if len(children) > 0:
            fp.write(indent+' <ul>\n') ## start an unordered (HTML) list
            children.sort(lambda a, b: cmp(a.url, b.url))
            for child in children:
      ## do a recursive call increasing depth level plus writing
      ## additional spaces before link
                _explore(fp,child,explored,depth+1,indent+'  ')
            fp.write(indent+' </ul>\n') ## end an unordered list
```

The other plugins work in a similar fashion, using the `site.linkMap` **object to traverse through its** `link` **objects.** `Urllist.py` **is thus only few lines of code:**

```
def generate(fp,site):
```

```
    """Output a sorted list of urls to the specified file descriptor."""
    fp.write('   <ol>\n') ## start ordered (HTML) list as we need a count
     urls=site.linkMap.keys()  ## keys are actual link urls
    urls.sort()  ## sort them alphabetically
    for url in urls:  ## and write them, numbering is handled by browser
        fp.write('
<li>'+plugins.make_link(site.linkMap[url],url)+'</li>\n')
    fp.write('   </ol>\n')  ## end ordered list
```

As we have seen, the link extraction tool's architecture is quite expandable. Schemes could be added (for example it should be possible to analyze `javascript:window.open(url)` occurences by adding url to links), additional parsers could be written (for example for other file formats, like xml etc) and missing features can be inserted (for example `<iframe>` tag doesn't get handled but it is standard now).

4.3.4. webcheckparse.py

This is an additional script that crawls through the generated HTML document and write it to a text file. This text file is later used as an input for the createGML tool which inturn produces the necessary GML document. It'll be commented inline, in Python style:

```
from HTMLParser import HTMLParser  # import Python's default parser
import sys  # and system functions
import string  # and string handling functions

class simplestack:  # define stack class, Last In First Out (LIFO)
    def __init__(self):
        self.contents = list()
    def push(self, iobject):  # add one object to the end of list
        self.contents.append(iobject)
```

```
    def pop(self):  # remove one object at the end of list
        if len(self.contents) > 0:
            end = len(self.contents)-1
            obj = self.contents[end]
            self.contents.__delitem__(end)
            return obj
        return None

class site:
    def __init__(self, line):
        self.line = line # initalize line parameter of self to line (URL)
        self.children = list()  # and prepare an empty list of children

    def stream(self, file):
        file.write("\n")  # write a newline character to file
        file.write("# %s\n" % self.line) # write line (URL) and newline with
                                         # hash sign at first position
        for child in self.children:  # for every child in children list
            file.write("~ %s\n" % child)  # write it to new line with
                                         # tilda sign at first position


class WebCheckParser(HTMLParser):  # subclasses HTMLParser
    def __init__(self):  # initialize with empty strings or zero values
        HTMLParser.__init__(self)
```

```
        self.offs = 0
        self.ina = False
        self.url = ''
        self.line = ''

        self.parent = None
        self.parents = simplestack()  # parents will be our LIFO stack
        self.parentoffs = 0

        self.sites = list()  # and sites will be in standard list
        self.init = False

    def handle_starttag(self, tag, attrs):  # override handling of start tags
        if (tag == 'div') and (dict(attrs)['class'] == 'content'):
            self.init = True  # for tags <div class="content" …> set init to
                 # True value
        if not self.init:   # end handling if we are not in this <div> tag
            return

        if tag == 'ul':  # for <ul> tags found
            self.offs += 1  # increase offset for 1
            #print self.offs
            self.parents.push(self.parent)  # and put its parent on stack

            self.parent = site(self.line)  # initialize parent to line (URL)
            self.sites.append(self.parent)  # and add parent to sites
```

```
        if (self.offs > 0) and tag == 'a':  # for <a …> tags in offset
            self.url = dict(attrs)['href']  # set url to href’s value
            self.ina = True  # and remember that we are in a link

    def handle_data(self, tag):  # handling of data in tag
        if self.ina:  # if we are in <a> tag
            self.line     =     "%s     --     %s"     %     (string.strip(tag),
string.strip(self.url))  # write its title, separator and actual link (URL)

            if self.parent != None:  # if it has a parent
                self.parent.children.append(self.line)  # append self to
                                                                          #
parent’s children
    def handle_endtag(self, tag):  # override handling of end tags
        if not self.init:  # do nothing for tags we haven’t declared
            return      # self.init to True in handle_starttag (<div …>)

        if tag == 'ul':  # if unordered list is closed
            self.offs -= 1  # decrease offset
            #print self.offs
            self.parent = self.parents.pop()  # and set parent from stack
            #print self.parent



        if tag == 'a':  # if hypertext reference is closed
            self.ina = False  # remember that we are not in it anymore
```

```
mywcp = WebCheckParser()  # initialize my parser


try:
    input = file(sys.argv[1], 'r')  # let's read file in read-only mode
except:
    print "Bad input file specified. Please pass the name of the HTML file produced by webcheck as the first argument."
    sys.exit()  # if file was not found, exit with a message in previous line


mywcp.feed(file(sys.argv[1]).read())  # feed my parser with URLs in text file,
for i in mywcp.sites:  # which was given on command line and for all lines
    if (i == None) or (i.line == ''):  # if current line is empty
        continue  # do nothing
    if len(i.children) != 0:  # otherwise if it has children
        i.stream(sys.stdout)  # write them on standard output stream
```

The output of this class is a file that will parse the html page content in a `<div class="content" …>` **tag and write all links found (in unordered lists) one in line. Every line will have links content (e.g. the text between** `<a>` **and** `</a>`**), a separator of two dashes and the actual URL of the printed links. Regarding children links, there will be a # sign at the beginning of line for parents and standalone links and a ~ sign for all children's (links) at the beginning of line.**

4.4. createGML documentation

briefly about the tool

5. Conclusion and Future Works

The final chapter summarizes in brief what has been discussed throughout the paper and also states some possible directions for improving the proposed tools.

5.1. Summary

Web structures are a constantly changing entity. There have been many approaches to describing them and the basic theoretic framework in the beginning of the document gives an idea about the dependencies between the components of a Web site. The graphical representation of these dependencies is an easy way to show complex models of interaction.

The fundamentals that build in practice the Web, have been also discussed in this paper. The Web relies of important technologies like TCP/IP (this protocol is the very foundation of Web communiction), DNS (which makes it possible for the millions of servers and users to find each other and communicate), the history and present of Web browsers (which are the main software used to browse the Web), HTML (which is the fundmanetal language that all Web sites use), and DHTML, XML, Perl, Python, and Server-Side Scripting (which are advanced technologies to provide functional sites that enhance user experience).

Then we discussed some of the nightmares Web masters face in maintaining sites. One of the most time-consuming task is to check for bad links. If done manually, it could take ages, especially for large sites. But fortunately Web masters do not have to do it manually because a new category of Web software was born – link-checkers and Web analyzers. Among the freely available link-checkers, the link extraction tool stands out from the crowd with its functionality and stability.

The proposed link extraction tool has been thoroughly examined from architectural and programming point of view. The text file with links that the link extraction tool outputs are an input for another tool – createGML. Based on the structure of a site as described in the text file, createGML parses the site and creates a visual representation of the “nodes” (links) and “edges” (webpages) in the site as a GML document which can be used in the VisuGraph tool.

5.2. Conclusion

Since the technologies and tools described in this paper are a work in progress, it will never be possible to describe them in enough detail. Nevertheless, we believe

that their explanation in this paper has been sufficient in the context of Web analysis for Web navigation. We tried to describe as fully as possible the theoretical and fundamental concepts of the Web, as well as the practical implications. We believe that the detailed “dissection” of the link extraction and createGML tool is useful and necessary in order to show and understand its inner workings.

5.3. FUTURE WORK

As mentioned above, the technologies and tools described in this paper will continue to develop in the future. While it is more difficult to say in which direction Web technologies will shift, it is easier to do it for the tools described.

For instance, for link extraction tool the direction might be to add more features. Although it is now a mature and stable product, which is successfully competing with commercial alternatives and which should be in every webmaster arsenal, there are things that can be added. Maybe it does not need so desperately new features like some of its open source counterparts (for instance those that lack reports generation or do not have an option to set the interval of checks) but even improvement of existing ones (e.g. `<iframe>` HTML tag, possible URL searching in javascript methods etc) is a step ahead. It might be good to expand development to more programmers than just one currently, by establishing project on known open source project hosting sites like SourceForge or freshmeat.

6. REFERENCES/BIBLIOGRAPHY

Paul Albitz and Cricket Liu. DNS and BIND. O'Reilly, Beijing [China] ; Sebastopol, CA, 4th edition, 2001.

Ricardo Baeza-Yates, Carlos Castillo, and Felipe Saint-Jean. Web dynamics, structure, and page quality.

Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine.

Andrei Broder, Ravi Kumar, et al. Graph structure in the web. page 19, 2003.

Jason Chaffee and Susan Gauch. Personal ontologies for web navigation. CIKM, 2000, 2000.

Reinhard Diestel. Graph theory. Graduate texts in mathematics ; 173. Springer, New York, NY, 3rd edition, 2005. Reinhard Diestel.

David Flanagan. Java in a nutshell : a desktop quick reference. The Java series. O'Reilly, Cambridge ; Sebastopol, CA, 2nd edition, 1997.

Eric J. Glover, Kostas Tsioutsiouliklis, and et al. Using web structures for classifying and describing web pages. WWW2002, May 7-11, 2002.

Elliotte Rusty Harold and W. Scott Means. XML in a nutshell : a desktop quick reference. O'Reilly, Sebastopol, CA, 1st edition, 2001.

Craig Hunt. TCP/IP network administration. A Nutshell handbook. O'Reilly & Associates, Sebastopol, CA, 1st edition, 1992.

Ravi Kumar and Prabhakar et al. Raghavan. Stochastic models for the web graph.

Ben Laurie and Peter Laurie. Apache : the definitive guide. A Nutshell handbook. O'Reilly & Associates, Cambridge [Mass.], 1st edition, 1997.

Michael J. McGuffin and M. C. Schraefel. A comparison of hyperstructures: Zzstructures, mspaces, and polyarchies. HT'04, (August 9-13), 2004.

Jennifer Niederst. Web design in a nutshell : a desktop quick reference. O'Reilly, Beijing; Sebastopol [Calif.], 2nd edition, 2001.

Alexandros Ntoulas, Junghoo Cho, and Christopher Olston. What's new on the web? the evolution of the web from a search engine perspective. WWW, 2004 (May 17-22), 2004.

LarryWall, Tom Christiansen, and Jon Orwant. Programming Perl. O'Reilly, Beijing ; Cambridge, Mass., 3rd edition, 2000.

Bowlin, James B.. "Linklint - fast html link checker". 1997-2001. http://www.linklint.org/

Forsman, Robert. "JCheckLinks". 1996.

http://web.purplefrog.com/~thoth/jchecklinks/

FSF. "FSF - The Free Software Foundation". 2004, 2005.

http://www.fsf.org/

GNU Project. "GNU General Public License". Version 2, June 1991.

http://www.gnu.org/copyleft/gpl.html

Graaff, Hans de. "Checkbot". 2005.

http://degraaff.org/checkbot/

IANA. "MIME Media Types". 1999-2001.

http://www.iana.org/assignments/media-types/

ibiblio.org. Linbot 1.0 description. 1999.

http://www.ibiblio.org/pub/linux/apps/www/misc/linbot-1.0.lsm

Jong, Arthur de. "webcheck: website checker". 16 Aug. 2005.

http://ch.tudelft.nl/~arthur/webcheck/

Marshall, James. "Checklinks 1.01". 1998-2000.

http://www.jmarshall.com/tools/cl/

Mettier, Yves. "link-checker". 1999.

http://ymettier.free.fr/link-checker/link-checker.html

Michel, Martial. "DLC : Dead Link Check". 1999-2000.

http://dlc.sourceforge.net/

Network Working Group, Tim Berners-Lee eth. "Uniform Resource Identifiers (URI): Generic Syntax". Aug 1998.

http://www.ietf.org/rfc/rfc2396.txt

Python Software Foundation (PSF). “Python Programming Language”. 2005.

http://www.python.org/

Saracco, Emmanuel, eth. “GForge Project Info- gURLChecker”. 2003-2005.

http://labs.libre-entreprise.org/projects/gurlchecker/

Socher, Guido. “Guido's Linux® Home-page”. 2000

http://cgi.linuxfocus.org/~guido/index.html

The Internet Society. “HTTP/1.1: Status Code Definitions”. June 1999.

http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html

Watchfire Corporation. “Web Content Quality Testing”. 1996-2005.

http://www.watchfire.com/products/desktop/qualitytesting/default.aspx

W3C. “About the World Wide Web Consortium (W3C)”. 9 Aug. 2005. http://www.w3.org/Consortium/

W3C. “HyperText Markup Language (HTML) Home Page”. 1995-2005.

http://www.w3.org/MarkUp/

W3C. “W3C Link Checker”. 1999-2005.

http://validator.w3.org/checklink/