

UNIVERSITY OF CALIFORNIA
RIVERSIDE

Artur's Auto Annotator

A Project submitted in partial satisfaction
of the requirements for the degree of

Master of Science
in
Computer Science

by
Artur Kedzierski

June, 2002

Project Committee:

Dr. Marek Chrobak

Dr. Dimitrios Gunopulos

Copyright by
Artur Kedzierski
2002

The Project of Artur Kedzierski is approved:

University of California, Riverside

ABSTRACT OF THE THESIS

Summarizing systems are essential in the management of search engines and other web page collection systems. Although there are several approaches used to generate summaries, each has unique limitations. What is explored by this research project is a summarizing approach that recognizes a variety of web pages and uses an appropriate technique to create a summary page. This is done by parsing and converting the pages into a set of various objects, identifying the type of the web page, and then using the objects and the appropriate technique to create a summary. The summary could be a meta description, a sample paragraph, or sample sentences coming from a web page given by a user. If appropriate or necessary the system will crawl the links in that page to obtain better content for the summary.

The work generated by this thesis is intended for use by the INFOMINE Project developed by the University Library at the University of California at Riverside. It will serve as a tool in creating annotations for the INFOMINE-a job currently performed manually by project personnel. Its integration in the upcoming INFOMINE crawler will automatize the annotating process.

Contents

1	Introduction	1
1.1	Historical Perspective	1
1.2	Problems That Emerged in the Growth of the Internet	2
1.3	Automatic Summarizing Systems	3
1.4	Problem	4
1.5	Description of Thesis	4
1.6	Overview	5
1.7	Thesis Organization	5
2	Background and Related Work	7
2.1	Search Engines	7
2.2	Annotation using anchors	9
2.3	Ocelot	11
2.3.1	Bag of words gisting	11
2.3.2	Expanded-lexicon gisting	12
2.3.3	Readable gisting	13
2.3.4	Training a Ocelot	14

2.4	Summarization by Paragraph Extraction	17
2.5	Summarization by Sentence Extraction	20
3	Methods	25
3.1	Time Line	25
3.2	Scope	26
3.3	The Process	30
3.3.1	Parsing and Storing	31
3.3.2	Type of the Web-Page	41
3.3.3	Creating Annotation	43
3.3.4	Returning the Results	45
4	Experiments and Results	53
5	Conclusion and Future Work	62
5.1	Summary	62
5.2	Future Work	63
A	Source Code	66

List of Figures

3.1	Tag Parsing	32
3.2	Dividing Units	34
3.3	Sentence representation of 'This is an example'	50
3.4	The Four Storages	51
3.5	Creating Annotation	52

List of Tables

3.1	Terminating Properties of HTML Tags	37
3.2	Available Parser Actions	38
3.3	Sentence, Paragraph, and Text Division Types.	39
3.4	Command options for the 'autoadder' program.	46

Chapter 1

Introduction

1.1 Historical Perspective

In 1962, Paul Baran of the RAND corporation came up with the notion of a packet switched network that would allow the military to "maintain its command and control over its missiles and bombers, after a nuclear strike..." [1]. This system of communication eventually evolved into the Internet. Although this system of communication connected only four sites in 1972—the University of California at Los Angeles, the SRI in Stanford, the University of California at Santa Barbara, and the University of Utah—it is now a widely used method of communication with millions of sites worldwide.

Although in today's world use of the Internet is second nature, it was a fairly new system only a decade ago. Prior to 1990, use of the Internet was primarily restricted to the military, scientific research, and technical use. However, in the nineties, as the personal computer became a standard item in the average home, the public and the business world discovered the "net". Kristula [1] tells us that, "many new networks were added to the NFA backbone. Hundreds of thousands of new hosts were added to the Internet during this time period." By 1994 it was possible to order pizza or bank on the net. Internet traffic increased so rapidly that the system had to be

continually upgraded. The incredibly rapid growth of this system created many new challenges. One of these being the ability to "navigate the web." The ability to catalog and classify the hundreds of web sites that were popping up on a daily basis was definitely a challenge for the various search engines. As the number of web-pages on the Internet keeps growing, the search engines and web directories keep trying to keep up with indexing them.

1.2 Problems That Emerged in the Growth of the Internet

To manually create summaries for most web pages, it requires hundreds, if not thousands of people. For today's average search engine it is not expedient, cost effective or productive to manually create pages. A case in point that serves as an excellent example, is the Google search engine. As of February 17, 2002, Google had indexed 2,073,418,204 web pages. For the sake of argument, if one allows an average of five minutes for a person to write an annotation about a given page, it would take 172,784,850 man hours to have summaries for all of the pages. Although, Google keeps it a secret as to how often the pages are indexed, it is a process that continually should be updated. If they were to do it once a month, a staff of approximately 4,319,621 would be required for this task. Even if the staff worked for minimum wages (figured as \$6.75 per hour in California), the cost of the labor could be estimated to be \$29,157,443 per month. Keeping in mind that in any business a company has many other operating costs beyond labor, it would be next to impossible to stay in business without automation. Clearly, it is not practical, cost effective or sound business to manually summarize web pages.

1.3 Automatic Summarizing Systems

An automatic summarizing system can help with the labor intensive problem. Two distinct methods of automation help by showing the personnel who add records a candidate for annotation or by replacing the personnel altogether and writing the summary for them. In the first method, a person types in a URL, and he or she is presented with a summary for the given web-page. From there, he/she can accept it, modify it, or write it from scratch. Of course, the more optimal summarizer would be the one that would provides summaries that require a minimum of editing. The second method, doesn't involve any human interaction. A computer feeds the system with a URL, the summarizer returns generated summary, and the two are paired in a database This is the method that the search engines employ to update their systems. The reason for this is that the first method, cuts down on the time that is required to write such an annotation, but it doesn't eliminate it. If one goes back to the Google example, and one cuts the time it takes to create an an annotation to 20 seconds, the whole process would still require approximately 11,518,990 man hours and a staff of 287,974 people. Although it is better, it is still not an acceptable business option.

Since the automation of a summarizing system is essential to the management of search engines and or other web page collection systems, there is growing research in this field. The automatic summary problem is being discussed by research that addresses the problem from various dimensions. From what is currently known, the summaries can be generated using or combining the following approaches: a) query-based, b) statistic, c) linguistic, and d) anchor-based.

A Query-based approach is commonly used by search engines. A summary is generated using sentences that contain words that a user searched for. In the statistic approach, chunks of text (words, sentences, or paragraphs) are assigned weight based on various factors. Some of these factors include position in the text, frequency of their occurrence in the document, or whether or not they are stop words. Then, the annotation is generated using text chunks with the highest weights. A linguistic ap-

proach recognizes the part-of-speech in the sentences to create the sentences. Finally, the last approach, the anchor based one, relies on the web pages that point to the document. From those documents, one extracts text that is in the proximity of the anchor that points to our document.

1.4 Problem

The problem is that each of the current approaches has unique limitations. Where as a query-based approach is good with generating annotations for search engines, these approaches are useless for directories where annotation is generated without knowing what a user will search for. The statistic and linguistic approaches work well with pages that contain a lot of static content but not on pages containing scattered links or with pages that contain dynamic content (for example news-sites). The anchor-based approach can handle such pages. It can even create annotation about a page from which we cannot extract any text– like a flash-only site (a site containing Macromedia’s Flash navigation). However, if the page is not indexed by any of the search engine, there is no way of obtaining other pages that would point to the site. Without that required information, one is not able to use the anchor-based approach.

1.5 Description of Thesis

What is needed is a system that recognizes the variety of web pages and uses an appropriate technique to create a summary. The intent of this project is to develop a model that recognizes a variety of web pages and uses the appropriate technique to create a summary page. This will be done by parsing and converting the pages into a set of various objects, identifying the type of the webpage, and then using the objects and the appropriate technique to create a summary. The work generated by this thesis is intended for use by the INFOMINE[2] project.

1.6 Overview

The INFOMINE is a computer project developed by the University Library at the University of California, Riverside. It is intended for: "the introduction and use of Internet resources of relevance to faculty, students, and research staff at the university level. It is being offered as a virtual library and reference tool containing highly useful Internet resources including databases, electronic journals, electronic books, bulletin boards, listservs, online library card catalogs, articles and directories of researchers, among many other types of information. " [2]. The research described in this thesis will be used in two ways: as a helping tool in creating annotations for the INFOMINE adders, and as a replacement of the adder by integrating it in the upcoming INFOMINE crawler. Within this project the term adder refers to an individual who adds records to the INFOMINE database.

To add a record, an adder opens a web interface page called a record editor where among other information, he/she has to type in a URL, author of the page, keywords, subjects, and an annotation. The intent of this research project is to facilitate this process. It is necessary because having an available annotation not only makes adding records more enjoyable, but it also shortens the time needed to enter a new record. This in turn reduces the cost of maintaining the database. The second aim of this project is to incorporate this summarizer in the INFOMINE crawler. Currently the crawler is under development. Since the design plans contained automatic generation of various information about a web-page but did not include automatic annotation, this summarizer will fill the gap.

1.7 Thesis Organization

In fulfilling the requirements of the thesis project, there will be an extended review of the existing related work in this field of study. Chapter two will entail a description of how some of the major search engines generate summaries. There will also be an

elaboration on how to generate a summary using anchors. Additionally, there will be an explanation of how a system called "Ocelot", that uses the statistic approach for creating annotations, functions, and how summarizations, using paragraph and sentence extractions, work.

Chapter three will describe in detail the methods used within this project. It will delineate how parsing and converting the page into a set of objects is done. It will also describe how to determine the type of the web-page, and how to use the objects, the type of the page, and the appropriate technique to create a summary. Finally, there will be an explanation of the method used to return the result to the user.

Chapter four will focus on experiments and result that presented themselves in the application of the research work. Part of this chapter will describe the mixing of the methods, the process used to get an idea of what a page is about, and how doing it in this manner, obtained a better annotation. It will also describe the outcome of the project. The final and last part of this project, chapter five, will report my conclusions and recommendations for future work. It is my intent to improve the existing methods of creating annotations and add to the current research in the field with the completion of this project.

Chapter 2

Background and Related Work

This chapter will review some of the related literature in this field of study. It will include a description of the methodology used by popular search engines to summarize web-pages. It will also review the use of anchors in the annotation process. There will be an examination of the work related to the Ocelot system that uses probabilistic models to create a summary. Additionally there will be a study of summarization by paragraph extraction and summarization by sentence extraction. The intent is to understand the existing work in the field and to use this body of knowledge in the design of this project.

2.1 Search Engines

This section describes various methods that most popular search engines use to summarize a web-page. Finding out the details of how their summarizers work is difficult since most search engines use proprietary technologies. Sites like Excite and Lycos use such technology. However, Infoseek and AltaVista use META description as a summary [3].

Excite[4] does not use META tags for description. It pays attention to the punctuation and extracts complete sentences. Also, the beginning of the page is important.

From it, Excite, tries to determine the theme of the page. If the theme cannot be determined, its robot uses the rest of the page and even goes deeper into the web-site.

HotBot / Inktomi[5] uses META tags. It incorporates both keywords and description.

Infoseek[6] uses META tags for description and keywords. It uses first 1000 bytes from meta keywords and 200 bytes of meta description. If the tags are not found on the web-page, its robot obtains first 200 bytes after the <BODY> tag. Infoseek also uses ALT description of tag for the description of the page. This is helpful with sites that contain a lot of graphics.

Lycos doesn't use META tags. It processes the entire page to pick the topic of the page. As a description, it uses the most informative part.

2.2 Annotation using anchors

Einat Amitay and Cécile Paris[7] propose a new solution for generating summary about a web-page. The idea is to use anchors from pages pointing to the site that we want to write an annotation about. Using this idea, the authors created an experimental system called InCommonSense.

The approach is very useful for pages that constantly change or it is hard to determine the page's theme, i.e. news portal web-site. It is able to create a summary of any kind of document from journal articles to lists of links. It can be a video or audio clip.

The summarizer assumes that there exists a pattern in the way people annotate other pages and that this pattern can be used in conjunction of knowledge of HTML to automatically extract a summary. There are four patterns that people use to annotate a link. The first one is an anchor followed by the description. No other anchors follow in the block of text. This is the most widely used pattern on the web and it is used by the InCommonSense. The second pattern is a description with an anchor in the middle of it. The third pattern is a description followed by an anchor. In the second and third pattern, there exists only one anchor in the paragraph. The fourth pattern is a description with multiple anchors embedded in the text.

The InCommonSense system works by obtaining URLs of web-pages that point to the document. This is done by connecting to the major search engines and executing 'link:' query. The system obtains up to 220 URLs. After downloading the pages, the system goes through these documents looking for the pattern previously described. The paragraph is determined not only by spaces but also by HTML tags that create a visual break when the page is viewed on the web browser. Next, the system has to pick the best description from the list that matched the criteria. The filter that would perform this task has been designed based on an experiment using real people.

In the experiment that helped design the filter for picking up the best description,

the subjects were given a web-page and then a list of descriptions that had been picked in the previously defined way. Then, they had to rank each description using a 1 to 5 scale where 1 was bad and 5 was good. Once all the pages were rated, all the descriptions has been divided into three groups. One group consisted of annotations with scores between 4 to 5 which indicated the 'good example' group. The second group, with scores from 4 to 2, meant that the examples were not distinctively good or bad. The third group, with scores 2 to 1, contained the bad descriptions. The purpose of dividing the descriptions into groups was to determine what language features had been used in bad examples. This way, the system would never return a description that would be considered bad. After further experiments, it was determined that 15 features should be used in the system. The features include "...length, punctuation (commas, dashes, exclamation marks), use of personal pronouns, use of acronyms, use of terms expressing opinion (e.g. best, comprehensive), use of terms indicating content (e.g., about, information), position of punctuation (beginning, end), position of verbs (beginning), text beginning with capital letter, and term repetition ration (in %)."

The flaws in this system is that it relies on search engines. The web-page must be already submitted and indexed. Also, there must be other sites linking to it.

2.3 Ocelot

Adam L. Berger and Vibhu O. Mittal[8] describe a system called Ocelot. This system uses probabilistic models to create a summary of a web-page. It will synthesize a summary rather than extract. To create the statistical models, the system takes previously summarized web-pages from Open Directory Project. The summaries at that project were created by volunteers and contain approximately 13 words.

Ocelot consists of three parts:

- Content selection; Ocelot uses two different methods in selecting content from a web-page. One method is using word frequencies. A word appears in the summary (a “gist” as author calls it) with the same frequency as it appears in the document. This method contains an extension that allows words that do not appear in the document to be in the summary.
- Word ordering; An order at which a word appears in the summary depends on words before it. For a example, if a word `platypus` is already in the summary, it is not likely to appear in it again. However, a common word like `the` may be in the summary in a few places, but it will not be at the position k , if it already is at position $k-1$.
- Search; After a few candidates for a summary are selected, the system searches them for one that is the best in terms of readability and content selection.

The authors of Ocelot describe three increasingly sophisticated algorithms that they use in creating the summaries. These are “Bag of words gisting”, “Extended-lexicon gisting”, and “Readable gisting”.

2.3.1 Bag of words gisting

One starts by picking a length n of the summary according to some probability distribution ϕ . Then, for each word position in the summary, one picks a word from

the document using frequency function $\lambda(\cdot|d)$. This is the algorithm:

Input: Document \mathbf{d} with word distribution

$\lambda(\cdot|d)$; Distribution ϕ over summary lengths.

Output: Summary \mathbf{g} of \mathbf{d}

1. Select a length n for the gist: $n \sim \phi$
2. Do for $i = 1$ to n
3. Pick a word from the document: $w \sim \lambda(\cdot|d)$
4. Set $g_i \leftarrow w$

According to this algorithm, the probability of a gist $g = g_1, g_2, \dots, g_n$ is:

$$p(g|d) = \phi(n) \prod_n^{i=1} \lambda(g_i|d).$$

In algorithm, words that appear most frequent will be most likely to appear in the summary.

2.3.2 Expanded-lexicon gisting

This scheme expands the previous algorithm by allowing words that do not appear in the document. If a word is very similar to another one, the word is replaced. To determine which words should be replaced, we use a probability distribution $\sigma(\cdot|w)$. If a word w is similar to u , value of $\sigma(v|w)$ will be high. The values of σ can be represented by $W \times W$ size table, where W is the number of words available to us.

Input: Document \mathbf{d} with word distribution

$\lambda(\cdot|d)$; Distribution ϕ over summary lengths;

Word-similarity model $\sigma(\cdot|u)$

Output: Summary \mathbf{g} of \mathbf{d}

1. Select a length n for the gist: $n \sim \phi$
2. Do for $i = 1$ to n
3. Pick a word from the document: $u \sim \lambda(\cdot|d)$
4. Pick a word replacement for that word: $v \sim \sigma(\cdot|w)$
5. Set $g_i \leftarrow w$

Using this algorithm a probability of a n -words gist $g = g_1, g_2, g_3, \dots, g_n$ of a document d containing m words, we have the following probability of a summary:

$$p(g|d) = \phi(n) \prod_{i=1}^n p(g_i|d) = \phi(n) \prod_{i=1}^n \left(\frac{1}{m}\right) \sum_{j=1}^m \sigma(g|d_j) \quad (2.1)$$

Both of the algorithm considers a summary as a loose collection of words. The third algorithm increases it coherence.

2.3.3 Readable gisting

This algorithm extends the second algorithm by scoring the gist by not only how well it corresponds to the document but also how readable it is. The readability or coherence of an n -word string $g = \{g_1, g_2, \dots, g_n\}$ is a probability ($p(g)$) of seeing that gist in text. This probability can be factored into a product of conditional probabilities as:

$$p(g) = \prod_{i=1}^n p(g_i|g_1, g_2, \dots, g_{i-1})$$

However, in practice, Ocelot uses a trigram model for $p(g)$:

$$p(g_i|g_1, g_2, \dots, g_{i-1}) \approx p(g_i|g_{i-2}, g_{i-1}) \quad (2.2)$$

To pick the best gist, use pick $g^* = \operatorname{argmax}_g p(g|d)$. One can apply Bayes' Rule to it to obtain:

$$g^* = \operatorname{argmax}_g p(g|d) = \operatorname{argmax}_g p(d|g)p(g)$$

. Here one has two terms $p(d|g)$ and $p(g)$. The $p(d|g)$ tells us how close document d and the gist g match in content and the $p(g)$ indicates how readable the gist is.

For the $p(d|g)$, one can reverse the direction of (2.1):

$$p(d|g) = \hat{\phi}(m) \prod_{i=1}^n p(d_i|g) = \hat{\phi}(m) \prod_{i=1}^n \sum_{j=1}^m \left(\frac{1}{n}\right) \sigma(d|g_j) \quad (2.3)$$

A $\hat{\phi}$ is a length distribution on documents. However, since it contributes equally to every candidate, it can be ignored. The algorithm for Readable gisting follows:

Input: Document \mathbf{d} with word distribution $\lambda(\cdot|d)$;

Distribution ϕ over summary lengths;

Word-similarity model $\sigma(\cdot|w)$

Trigram language model $p(g)$ for gists

Output: Summary \mathbf{g} of \mathbf{d}

1. Select a length n for the summary: $n \sim \phi$
2. Find, by searching, the sequence $g = \{g_1, g_2, \dots, g_n\}$ which maximizes $p(d|g)p(g)$.

The $p(d|g)$ can be thought of as a probability that document d would arise from the gist g .

2.3.4 Training a Ocelot

The source of training the statistical model of summarizing, the authors used summaries from Open Directory Project (<http://dmoz.org>) and then downloaded the corresponding documents. Open Directory Project is an effort of a large group of volunteers to create short (approximately 13 words) summaries of web-pages. The database, as of August 2001, contained 2,813,011 records. After downloading the corresponding web-pages, the authors followed these steps: ”

- Normalize text: remove punctuation, convert all text to lowercase; replace numbers by the symbol NUM; remove each occurrence of the 100 most common overall words (stopword-filtering).
- Remove all links, images, and meta-information from the web-pages.
- Remove pages containing adult-oriented content;
- Remove HTML markup information from the pages;
- Remove pairs whose pages contained frames;
- Remove pairs whose pages that had been moved since they had been included in the list; in other words, pages which were just “Page not found errors”;
- Remove pairs whose web-pages or gists were too short—less than 400 or 60 characters, respectively.
- Remove duplicate web-pages.
- Partition the remaining set of pairs into a training set (99%) and a test set (1%). (Traditionally when evaluating a machine learning algorithm, one reserves more than this fraction of the data for testing. But one percent of the Open Directory dataset comprises over a thousand web-pages, which was sufficient for the evaluations we performed.)

“[8],pg.4

Word Relatedness

The probability $p(d|g)$, which is a probability of obtaining a document given a gist g , can be factored out into a product of sum of $\sigma(d|g)$ (see 2.3). $\sigma(d|g)$ is a probability that a word g in the summary of a document will translate itself into a word d from that document.

To construct the probabilistic model for $\sigma(d|g)$, we create a stochastic matrix the size of $W_g \times W_p$. W_g is a set of words that can be used in the gists and W_p is a set of words that can be found in the web-pages. To calculate the values of entries in this matrix, a notion of *alignment* \mathbf{a} between sequences of words. This alignment indicates how the words in the summary produce words in the document. Also, to connect gist words that are not relative to words in the document, a NULL is used. It is inserted at the zeroth position of the gist.

We can rewrite $p(d|g)$ using *alignment* \mathbf{a} :

$$p(d|g) = \sum_a p(d, a|g) = \sum_a p(d|a, g)p(a|g) \quad (2.4)$$

Assuming that each word in the d corresponds to exactly one word in g (including NULL), we can simplify the equation:

$$p(d|a, g) = \prod_{i=1}^m \sigma(d_i|g_{a_i}) \quad (2.5)$$

Where, d_i is the i th word in the document and g_{a_i} is the gist word alignment with that word.

There are $(n + 1)^m$ alignments between a summary g containing n + 1 words and document d containing m words. We say n + 1 because of the NULL word. Assuming that the alignments are equally likely, we obtain:

$$p(d|g) = \frac{p(m|g)}{(n + 1)^m} \sum_A \prod_{i=1}^m \sigma(d_i|g_{a_i}) \quad (2.6)$$

To train the model, one takes the collection of summaries and their corresponding pages (form the Open Directory project) and maximize the value of 2.3 keeping in mind that $\sum_d \sigma(d|g) = 1$ for all words g. Using Lagrange multipliers, we obtain:

$$\sigma(d|g) = Z \sum_a p(d, a|g) \sum_{j=1}^m \delta(d, d_j) \delta(g, g_{a_j}) \quad (2.7)$$

Z is the normalizing factor and δ is the Kronecker delta function.

The $\sigma(d|g)$ appears in the equation on the left hand side explicitly and on the left hand side implicitly. By solving the equation repeatedly for all pairs d,g, we will reach eventually reach a stationary point of likelihood.

Language Model

Ocelot use a trigram model (2.2) to ensure that the summaries are readable: $p(w|u, v)$. This is the probability that a word w follows the bigram u,v . For training, the Open Directory is used and the process takes the following steps: “

1. Construct a vocabulary of active words from those words appearing at least twice within the collection of summaries...
2. Build a trigram word model from this data using maximum-likelihood estimation.
3. “Smooth” this model (by assigning some probability mass to unseen trigrams) using the Good-Turing estimate [9].

“[8],pg.5-6

For the final two steps, Ocelot uses the publicly-available CMU-Cambridge Language Modeling Toolkit [10].

2.4 Summarization by Paragraph Extraction

Another method of summarization is one by paragraph extraction. One paper that discusses this method is “Automatic Text Summarization by Paragraph Extraction.” by Mandar Mitra, [11].

The benefit of this method is that the summary tends to be more coherent than using a sentence based extraction.

One can represent a document D as a vector of weighted terms: $D_i = (d_{i_1}, d_{i_2}, \dots, \dots, d_{i_t})$. Each term d_{k_i} corresponds to an importance weight of a specified unit in document D_i . This unit can be a word or a phrase. The weight is calculated based on occurrence of the word, total number of words in a document, and total number of words in the set of documents.

The drawback of this method is that it doesn't work well with most of the web-pages. A lot of web-pages are not structured into a document-like format. They tend to have various pieces of information scattered around the page.

The program starts by extracting pieces of text using indexing procedure based on word frequencies in the document and in a collection of documents to which the document belongs. Each piece is then represented by a vector as a set of weighted terms. With such representation, one is able to compute similarities between two pieces D_i and D_j based on what they are composed of, i.e. if D_i and D_j are each a paragraph, one can find the similarity between them based on the words that each of the paragraphs is composed of. The similarity can be written as:

$$Sim(D_i, D_j) = \sum_{k=1}^t d_{i_k} d_{j_k}$$

and Sim can be normalized to range between 0 and 1, where 1 indicates that D_i and D_j are identical. Next a *text relationship map* is created.

To pick the best paragraph, a *text relationship map* is composed. "A textual relationship map is a graphical representation of textual structure, in which paragraphs (in general pieces of text) are represented by nodes on a graph and related paragraphs are linked by edges" [11], page 2. The map basically shows how various paragraphs are similar to each other. To create the map, first one picks a threshold that will be used for the minimum similarity. If a similarity between two paragraphs exceeds that threshold, one creates an edge between them. Such two paragraphs one considers to be "semantically related". One continues mapping the relationship for each paragraph in the document. This gives us the *text relationship map* and we are ready to create a summary. An important thing to notice is that in some maps, one may find that some paragraphs clustered themselves into segments. A segment is a group of paragraphs that are linked to each other but there exists very little connection to other paragraphs.

Now, one starts creating the gist of the document. There are four increasingly sophisticated algorithms that one can use in conjunction with *text relationship map*

to create a summary. These are: Bushy Path, Depth-first Path, Segmented Bushy Path, and the most advanced, Augmented Bushy Path.

To describe a “Bushy path”, one needs to define a “bushy node”. A node is bushy if it contains a lot of links to other nodes. Bushy nodes indicate that the paragraph has a lot of common vocabularies with other paragraphs. A “Bushy path” is constructed by picking n most bushy paragraphs. This path is then our summary of the document.

The problem with Bushy Path is that paragraphs in the summary are not necessarily connected. That means that the summary can be incoherent. The “Depth-first Path” fixes this problem. To create the “Depth-first Path”, one starts at an important node. The important node can either be the most bushy paragraph or first paragraph in the document. Then, one selects the most similar paragraph. From there, one follows the path of selecting the most similar nodes until one has n of them. This gives us the “Depth-first Path”. The problem with this approach is that not all aspects of the document may be covered in the summary. This can occur when there exist more than one clusters since the “Depth-first Path” will create summary that consists of nodes in the cluster that the initial node was from.

If a document deals with multiple subjects, and they tend to cluster into individual groups, some of the topics will not be included in the summary using “bushy path.” To avoid the problem one constructs a Bushy path for each segment. Then one concatenates these paths in the order that the subject occurs in the document. The resulting path one calls a “Segmented Bushy Path”. This algorithm ensures that one has at least one paragraph that is selected for each segment. If a segment is long, we select more paragraphs for the summary.

Although one now has all the aspects of the document in the summary and the summary is more coherent (it doesn’t switch from one subject to another and back to the first subject), the most informative paragraphs may not be included in it. The problem is that a document usually starts with paragraph that identifies a topic

of the document. If one had a good way of separating the topics, one could easily create a good summary by selecting first paragraphs from each segment. The last method that was discussed, the “Segmented Bushy Path”, tends to pick a paragraph that is located in the center of a segment because that paragraph links to all the others (it is the most bushy). This behavior causes the summary not to contain the introductory paragraph. One of the problems of not having this paragraph is that readability of the gist is decreased. Therefore, we augment the “Segmented Bushy Path” by always picking up the introductory paragraph, and then if needed, picking other bushy paragraphs. The resulting path is the “Augmented Bushy Path.”

The “Augmented Bushy Path” is the best method from the above described four algorithms. It tends to create a summary that is coherent and that covers most of the aspects in the document. However, we may not find it applicable for many web-pages due to their scattered and badly formed content.

2.5 Summarization by Sentence Extraction

Since many web-pages do not contain a multi-paragraph articles, a summary by paragraph selection may not be possible so one has to create a summary by sentence extraction. One of the papers that discusses this approach is “Summarizing text documents: Sentence selection and evaluation metrics” [12]. The paper discusses the use of combination of statistical and linguistic approach to information retrieval. The project allows also a query-relevant ‘ summaries. A query-relevant summary is a summary generated based on a word you are looking for. The formula for scoring each sentence is the following:”

$$Score(S_i) = \lambda \sum_{s \in S} w_s * (Q_s \cdot S_i) + (1 - \lambda) * \sum_{l \in L} w_l * (L_l \cdot S_i)$$

where S is the set of statistical features, L is the set of linguistic features, Q is the query, and w is the weights of the features in that set.” [12], page 2.

The authors made an important observation that the length of the summary should be independent of the length of the document. For example, the compression ratio is smaller for large documents. Therefore, the size of the summary should be fixed length. Other observations relate to what is important in the sentence extraction.

There are some words that tell us whether a sentence is a good or bad candidate for extraction into a summary. A word “Reuters” is a good indicator. It is placed at the beginning of the sentence that is an introduction to the rest of the article. Such sentences are good candidates for extraction. Also, “Reuters” is placed at the end of the last sentence and last sentences are not good candidates. The names of the days of the week (“Monday,” “Tuesday”, etc.) tend to be in summary sentences. Also, direct or indirect quotations appear more often in non-summary sentences. For example in such sentences, these words appear at least 75% more frequently: “according,” “adding,” “said.” Similarly, the word “adding” if followed by “that,” “he,” “she,” “there,” comma, or a colon appear in the non-summary sentences. Sentences with quotations are not good summary sentences, e.g. “analyst,” “sources,” “studies.” Words “adding to” does not indicate a quotation but “according to” does. Also, passive pronouns (“us,” “our,” “we,” etc.) are indicators of a non-summary sentences since they occur in quoted statements. Informal or imprecise words (“came,” “got,” “really,” “use”) are frequently seen in non-summary sentences.

“Other classes of words that appeared more frequently in non-summary sentences in our dataset included:

- Anaphoric references, such as “these”, “this”, and “those”, possibly because such sentences cannot introduce a topic.
- Honorifics such as “Dr.”, “Mr.” and “Mrs.”, presumably because news articles often introduce people by name, (e.g., “John Smith”) and subsequently refer to them more formally (e.g. Mr. Smith) (if not by pronominal references).

- Negations, such as “no”, “don’t”, and “never.”
- Auxiliary verbs, such as “was”, “could”, and “did”.
- Integers, whether written using digits (e.g. 1, 2) or words (e.g., “one”, “two”) or representing recent years (e.g., 1991, 1995, 1998).
- Evaluative and vague words that do not convey anything definite or that qualify a statement, such as “often”, “about”, “significant”, “some”, and “several”.
- Conjunctions, such as “and”, “or”, “but”, “so”, “although”, and “however”.
- Prepositions, such as “at”, “by”, “for”, “of”, “in”, “to” and “with”.

In addition to these indicators, “Selecting Text Spans for Document Summaries: Heuristics and Metrics” [13] gives additional information about the sentence selection. There are three levels at which one can analyze a document and assign weights to the sentences: “

- Sub-document Level: Different regions in a document often have very different levels of significance for summarization. These sub-documents become especially important for text genres that contain either (i) articles on multiple, equally important topics, or (ii) multiple subsection, as often occurs in longer, scientific articles, and books. All sentences within a sub-document are assigned an initial score based on the whole sub-document. Sub-document scores depend both on various properties independent of the content in the sub-document (e.g. length and position), as well as the lexical and syntactic relations that hold between the sub-documents (e.g. discourse relations, co-references, etc.)
- Sentence Level: Within a sub-document, individual sentences can be ranked by using both features that are independent of the actual content, such as the length and position, as well as content specific features that are based on number of anaphoric references, function words, punctuation, named-entities, etc.

- **Phrase/Word Level:** Within a sentence, phrases or words can be ranked by using features such as length, focus information, part of speech (POS), co-reference information, definiteness, tense, commonality, etc.

[13], pg. 2.

The paper also gives us additional indicators of summary-sentence. One is punctuation. Punctuation (not counting periods) tend to appear more often in non-summary sentences. Also, articles (“A,” “An,” “The”) appear more often at the beginning of summary sentence. Additionally, the length of a word can be used. Summary sentences have a greater average word length. Phrases such as “finally,” “in conclusion,” etc. also appear more often in summary sentences.

From “A trainable Document Summarizer” [14] we can also learn about location heuristic. It assumes that the important information is located and the beginning and the end of a document, at the beginning and end of a paragraph, and right after a heading. For their summarizer, they use the following feature set: “

- **Sentence Length Cut-off Feature:** Short sentences tend not to be included in summaries (section headings generally count as short sentences). Given a threshold (e.g., 5 words), the feature is true for all sentences longer than the threshold, and false otherwise.
- **Fixed-Phrase Feature:** Sentences containing any of a list of fixed phrases, mostly two words long (e.g., “this letter...”, “In conclusion...”, etc.), or occurring immediately after a section heading containing a keyword such as “conclusions”, “results”, “summary”, and “discussion” are more likely to be in summaries. This features is true for sentences that contain any of 26 indicator phrases, or that follow section heads that contain specific keywords.
- **Paragraph Feature:** This discrete feature records information for the first ten paragraphs and last five paragraphs in the a document. Sentences in a paragraph are distinguished according to whether they are paragraph-initial, paragraph-

final (for paragraphs longer than one sentence) and paragraph-medial (in paragraphs greater than two sentences long).

- Thematic Word Feature: The most frequent content words are defined as thematic words (ties for words with the same frequency are resolved on the basis of word length). A small number of thematic words is selected and each sentence is scored as a function of frequency. This feature is binary, depending on whether a sentence is present in the set of highest scoring sentences. Experiments were performed in which scaled sentence scores were used as pseudo-probabilities, however this gave inferior performance.
- Uppercase Word Feature: Proper names are often important, as is explanatory text for acronyms e.g. ,”... by the ASTM (American Society for Testing and Materials)”. This feature is computed similarly to the previous one, with the constraints that an uppercase thematic word is not sentence-initial and begins with a capital letter. Additionally, it must occur several times and must not be an abbreviated unit of measurement (e.g. , F, C, Kg, etc.). Sentences in which such words appear first score twice as much as later occurrences.

”[14],p.69

Chapter 3

Methods

3.1 Time Line

This section will delineate the steps taken in completion of this project. There will be a brief review of the time lines encountered within the work, the scope of the the programming, and the exact method used in conducting the project. This information is provided to ensure all important elements are annotated and made available to the reader. Although there are many common terms within the field of computer science, a brief definition of all terms is included within this section. The reason for this inclusion is to ensure that there are no misunderstandings regarding this work and the methods used in completing this project. The expectation and desire is to provide enough detail so that the project may be easily replicated in future research.

Time lines and General Procedures of the Project: Prior to beginning the project, an extensive review of the literature and related work was conducted. The literature review, started in the winter quarter of 2001 and finished at the end of the Spring quarter of 2001. There was extensive use of the Internet in the collection of research and background information. An especially helpful site was NEC Research Index[15]. The front page of the site explains that the ResearchIndex “is a scientific literature digital library that aims to improve the dissemination and feedback of

scientific literature, and to provide improvements in functionality, usability, availability, cost, comprehensiveness, efficiency, and timeliness.” The site’s database contains hundreds if not thousands of technical documents. The site gives the ability to the user to search the documents not only by the content of the documents but also by various other fields.

The fields that were especially important within this project were the citation fields. After a paper was found, it was possible to find other papers that cited this document. This enabled the researcher to find the improvements and weaknesses of the ideas expressed in the document. A comprehensive review of the literature related to this project is found in chapter 2.

The programming phase of the project began once the literature review was completed. It started in Spring quarter of 2001 and continued to the Spring quarter of 2002. The write-up of information began in Spring 2002 and continued throughout the duration of the work. The writing of the project occurred concurrently to the last programming phase. Modifications and clarifications were continually made as needed to ensure as accurate information as possible.

3.2 Scope

History of INFOMINE: The objective of the research within this thesis is to support and interface with the INFOMINE[2] project. The work described in this thesis was intended to help adders create new records and create annotations for the INFOMINE crawler. Within this document the term adders refers to a person who adds records, web pages and annotations, to INFOMINE. To understand this work, it is also important to understand the scope, intent and use of ‘the INFOMINE. Keeping in mind that this thesis is connected to a much broader research project, a brief description of INFOMINE is included in this section. The description is intended to help the reader understand and frame the context of the work within this thesis.

The history and a description of INFOMINE can be found on <http://infomine.ucr.edu/welcome>. It certainly is a project that had many changes and challenges as it evolved into its present existence. The project makes the following claim:

“INFOMINE began in January of 1994 as a project of the Library of the University of California, Riverside. It was one of the first Web resources of any type offered by a Library. It was also one of the first Web-based, academic virtual libraries as well as one of the first to develop a system combining the advantages of the hypertext and multimedia capabilities of the Web with those of the organizational and retrieval functions of a database manager. We now include focused, automatic Internet crawling and text extraction functions. Many of INFOMINE’s important features and services, described below, remain unique “

INFOMINE, as mentioned, provides a great number of access points, through both BROWSE (Date-What’s New, Title, Table of Contents, Subject, KeyWord, Title, Author, hyper-linked indexing) and SEARCH (title, subject, key word, author, description, full text) modes. Searching in fielded and full-text modes allows the user to quickly find high quality resources on the chosen subject(s). Nested, boolean searching capabilities are featured as is exact searching. Search results come back in the form of dynamically created Web pages. Results within these can be ranked by relevance to the search or alphabetically by title. Displays can be set to title only or full citation (with annotation). In INFOMINE each results set record, in addition, features indexing terms that are viewable and in hyper-link form and, when clicked upon, allow further broadening or narrowing of the search as desired.” [2]

The INFOMINE project was originally written in C program and services, described above remain unique to this project. However, after the original programmer (who did all the coding) left the project, the new programmers were not able to maintain it. At that point, the new programmers decided to re-write everything from scratch. This time they chose MiniSQL database and the programming language ‘Lite’ that comes with the database[16]. Unlike before, where all the query genera-

tion and all the computation were done on the server side, it was now done on the client side. The reason for this decision was the speed issue. The work was now distributed among the clients so that the load on the server would be as little as possible. The server would only be limited to serving HTML pages and obtaining the CGI arguments containing the SQL query. That query would then be passed to a database server and the results would then be returned to the user in form of a web-page.

When the project was finally re-written, it started showing its limitations. To have the web-browsers do most of the computation, a heavy use of Javascript was needed. The scripts on the web-pages contained hundreds of lines. In fact, the HTML that created the page was only a small percentage of the whole source for that page. The reason for the extreme size of the script was that the web browsers were incompatible with each other, and there was lack of object oriented programming in Javascript. Another limitation was found in the MiniSQL and in the Lite language.

The first reason for the problem was that the Javascript support on different browsers varied greatly. Some functions were either not implemented or they did not work as the specification indicated. The Microsoft Internet Explorer is especially known for this problem. To accommodate the variety of limitations, numerous functions were written using basic Javascript commands. Also, various pieces of code were activated based on the type of browser that was accessing the page.

The second reason for the big size of Javascript used in INFOMINE web-pages was the lack of object oriented programming. All the functions were written in two or three files. These files also included lengthy comments containing unnecessary information such as the names of authors of individual functions and the dates that the functions were created. The files were then appended to the HTML pages and sent over to the end-users. For users who have access to the Internet through the dial-up accounts, it meant waiting a few seconds before a page appeared on their screen instead of a fraction of a second.

The MiniSQL supported only a limited set of SQL commands. There were certain things that programmers wanted to do in SQL but were unable to do because of the limitations. Also, programming in Lite language required a constant rewrite of code for each page. There were no objects like classes or modules that could be used to re-use the code. Another issue was related to cost. At the time the previous programmers finished re-writing the project, the prices on the hardware dropped so much that it did not make sense to continue having to deal with the Javascript just to keep the load on the server low. Again, it was time for another re-write. However, this rewriting task was a more gradual one.

Since moving to the server-side for the search query generation would require much more work, the work started on a different side. Because of the limitations, the project moved to a more advanced DataBase Manage System (DBMS) called MySQL[17]. The next natural course of action was to dump Lite. The simple reason for it was that Lite did not support any other DBMS than MiniSQL. The language of choice for the replacement of Lite was Perl.

The command set of Perl is almost a superset of programming language to Lite. It is not a superset because there are a few commands that are different. It was this similarity that made the project decide to choose Perl over other programming languages. The porting to the language was simple but time consuming. Most of the work was replacing one command with the other. The most frequently replaced command was 'echo' that was substituted with 'print'. Also, programmers created their own Perl programming library that allowed them to reuse the existing code and create extra functionality. At this point it was time to remove most of the Javascript. At the same time that these changes were taking place, a new lead programmer became part of the project.

One of the major decisions made by the lead programmer was a complete re-write of all of the Perl scripts into C++. As time went by, more functionality and search choices were added, new interfaces to extradite adding new records were created, and

the whole project is now written in C++. The next additions to the INFOMINE is the new crawler and this automatic annotation system. The automatic annotation system is the focus of this thesis.

3.3 The Process

Before the process begins, a document must be downloaded and saved as a file. To download a document, the cURL and Libcurl[18] is used. The tool is described on its home-page in the following way: “Curl is a tool for transferring files with URL syntax, supporting FTP, FTPS, HTTP, HTTPS, GOPHER, TELNET, DICT, FILE and LDAP. Curl supports HTTPS certificates, HTTP POST, HTTP PUT, FTP uploading, kerberos, HTTP form based upload, proxies, cookies, user+password authentication, file transfer resume, http proxy tunneling and a busload of other useful tricks.” [18] Originally, it was only a command-line tool. However, recently, it has been re-written to become a C library. Using the library, the authors of cURL wrote the command-line tool. This change allows a programmer to link a program to Curl without the need of calling an external program.

Another thing that is done before the process begins and after the document is downloaded, is the insertion of important phrases. The tool used in obtaining phrases from a web-page is Keith Humphrey’s Phrase Rater. The Phrase Rater was written for the INFOMINE project as tool in aiding adders in selecting keywords for a given web-site. The first design was written in 2000, and it was able to extract individual words from a web-page based on assigning weight to individual words. For example, words in the title (except for the stop-words) would weigh more than the words in the rest of the text. Same for the words in the headers, text in bigger font, or in bold casing also were assigned a greater weight. Over the time, the program continued to evolve. Currently it is able to extract phrases as opposed to the individual words.

The automatic annotation system can be divided into four parts. This section

will present a comprehensive description of each of the components. The first part is parsing a web page and storing the elements in various objects, the second part is the process of determining the type of the page, the third part is creating an annotation using the objects, and the last part is returning the results back to the user.

3.3.1 Parsing and Storing

The first phase of this thesis involves parsing the page and storing the elements in various objects. For the sake of clarity, the term parsing within this project is defined as taking the file containing the HTML code, and then proceeding word by word and tag by tag to extract all the necessary information. During the parsing, the extracted words, sentences, paragraphs, and text-divisions had to be stored in such a way that they would be easily accessible later.

To create a parser, the fast lexical analyzer generator called Flex was used. The home-page of this generator describes it in the following way: “It is a tool for generating programs that perform pattern-matching on text. There are many applications for Flex, including writing compilers in conjunction with GNU Bison. Flex is a free implementation of the well known Lex program. It features a Lex compatibility mode, and also provides several new features such as exclusive start conditions”[19].

The parsing of the document is not case-sensitive. A simple reason for this is that the HTML standard does require tags to be in any particular casing. The parser ignores Java and Javascript tags and commands, Cascading Style Sheets definitions, 'noframe' tags, HTML comments, 'doctype' tags, and 'select' statements. If the parser encounters the 'frameset' tag, it inserts URLs for each frame into a list. On the closing 'frameset' tag (i.e. </frameset<), the parser exits and downloads each URL. Since a frameset can be re-written using a HTML table, the parser places the HTML source for each frame into an individual cell of one table. Then, the whole document containing this big table is passed back for parsing.

Parsing HTML tags is not trivial. The standard does not specify that every tag

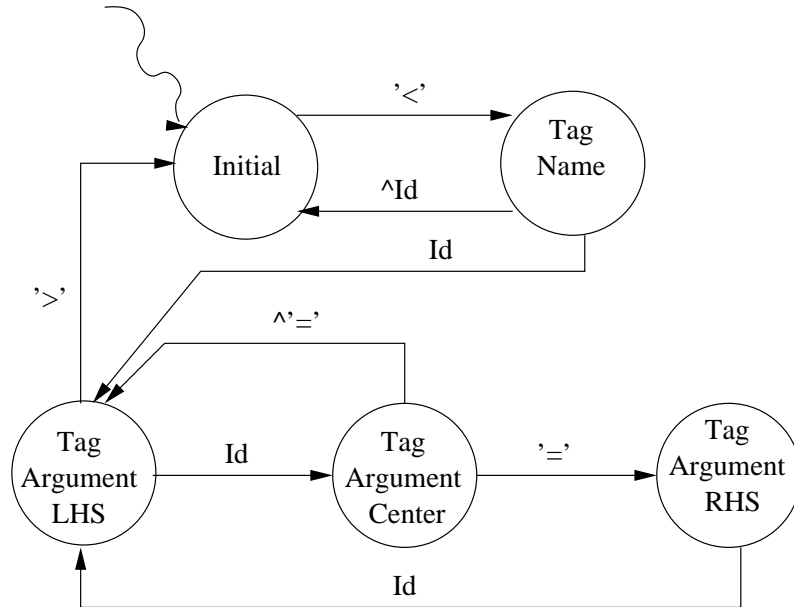


Figure 3.1: Tag Parsing

should have a closing tag, that each field in a tag should have some value, or that the value should or should not be in quotes. In addition to this complexity, most browsers are very forgiving so most of the web-developers ignore the standard and create pages that work with a specific browser. For example, even though the `<pp>` should have the closing `</p>` tag, it is frequently omitted. The researcher within this thesis project dealt with the problem of parsing tags as illustrated in figure 3.1. A tag looks like this: `<tag_name tag_argument_name=tag_argument_value>`. For example, the tag `` is parsed as `tag_name='font'`, `tag_argument_name='size'`, `tag_argument_value='3'`. When during the parsing (i.e. staying in initial state), an opening of an HTML tag (`'<'` character) is encountered, the parser enters the 'Tag Name' state. At this point, if the parser does not encounter 'Id' (which is a set of characters from 'a' to 'z' and from '0' to '9'), it goes back to the initial state. However, if the parser encounters the 'Id', it stores it as the tag name and goes to the 'Tag Argument LHS' (LHS = Left Hand Side).

Basically, at this point, the parser is looking for the name of the argument. If the tag closes (in example, `'>'` is encountered), it goes back to the initial state.

This can be the case of tag `<p>`. Otherwise, the parser stores 'Id' as the name of the argument, and proceeds to the 'Tag Argument Center'. Here the parser wants to see if this argument has a value. The parser does that by checking if the '=' character follows. If the tag is something like `<textarea nowrap>` (in other words '=' is missing), it assigns a value of '1' to this tag argument and process to the 'Tag Argument LHS'. Otherwise, it stores the 'Id' that follows the '=' as the value of the argument and also goes back to the 'Tag Argument LHS'. From that state, the parser either collects more name=value pairs or goes back to the initial state.

During the parsing of an HTML tag, the parser emits 'signals' at various states. These 'signals' are basically function calls. For example, when a pair of name=value is parsed, `ExtractorHelper::SignalTagArgumentValue` is called. The arguments for that function are the value of the tag argument and the enumerated type that tells which name of the argument it was. The current set of tag argument names is: 'class', 'content', 'href', 'http-equiv', 'name', 'rel', 'type', 'src', 'size', 'style', and 'dontcare'. The 'dontcare' tells us that the name of the argument is something that one doesn't care about. For example, one is not interested in tags such as 'align', 'color', or 'bgcolor'. The name of the tag is also an enumerated type. The reason for this is that the parsing is already case insensitive so converting the tags from string to enumerated type gives us a performance boost by eliminating any further string comparison. Once the closing of the tag is reached (in example, the '>' character is encountered), the parser calls 'ProcessTag'. The 'ProcessTag' takes two arguments: the enumerated type of the HTML tag name, and the value that specifies whether the tag is opening (for example, `<p>`) or closing (for example, `</p>`).

The purpose of the `ExtractorHelper::SignalTagArgumentValue` is simple. It keeps appending tag name and value pairs to the list of arguments. Once the end of tag is reached, `ProcessTag` will clear this list. `ProcessTag` performs various operations based on the different tags passed to it. Before these operations are performed, the function creates a variable 'actions' that contains various flags that indicate what various actions are to be taken. These actions are: `ACT_NONE`, `ACT_END_UNIT`,

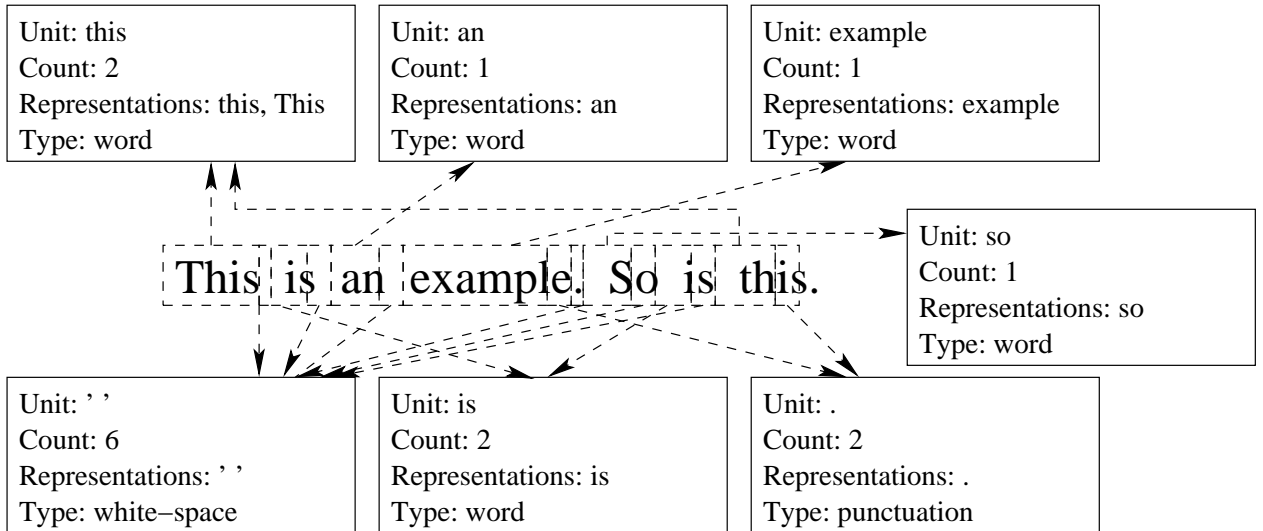


Figure 3.2: Dividing Units

ACT_END_SENTENCE, ACT_END_PARAGRAPH, ACT_END_TEXT_DIVISION, ACT_FEED_OUT_OF_TAGS, ACT_ADD_TO_WORD, ACT_TREAT_AS_SINGLE_UNIT. To explained the meaning of these actions, one has to understand how the parsed information is being stored.

Originally, each word, sentence, paragraph, and text-division (region of text), was stored in its own linked list. For example, there was the list of words, list of sentences, list of paragraphs, and the list of text-divisions. However, that lead to the problems that had to be addressed by the re-writing of the storage part. The main problem was that the program started looking extremely messy because various arrays for keeping track of the number of words, sentences, etc. had to be added. In the new design, there is a structure 'unit' that holds a chunk of text. The unit can be a white-space, a word, or punctuation. The structure contain various parameters. One of the parameters is 'representation' which holds other ways the unit can be represented. Let's take this case in point, 'Bacteria' is also 'bacteria' (note the capitalization of the words). The nice thing about this situation is that it allows stemming to be plugged in easily if it is necessary. For the same example, 'bacterium' would just be another representation for the 'bacteria'. Figure 3.2 gives an example how a sentence

is divided in units. It shows that a sentence “This is an example. So is this” is stored in seven units: this, is, an, example, so, ' ', ' '. The unit 'this' has two representations: 'this' and 'This' (notice capitalization). Also, its occurrence count is 2. As for the unit containing a space (' '), the occurrence count is 6.

A sentence is a list that contains pointers to units along with the index to the representation. With these two values, the actual string can easily be reconstructed. Additionally, 'sentence' has other attributes like the `number_of_words` which holds the number of actual words in the sentence. It also contains a boolean value that tells us whether the sentence is a link (in other words a text within `<a>` and `` tags). This structure also allows adding other parameters that can help assign weights to the sentences. Figure 3.3 illustrates how the sentence 'This is an example' is stored using this scheme. The linked list inside the sentence points to the appropriate units and holds an index to the representation of the words. The 'sentence' also keeps the number words (as in units of type 'word'), and may easily hold any additional information. Once a sentence is ended by a period or by “sentence terminating tag”, the sentence storage is added to the paragraph storage.

Then, there are structures 'paragraph' and 'text_division' as seen in figure 3.4. Again, the 'paragraph' is a linked list of pointers to the sentences and 'text_division' is a linked list of pointers to the paragraphs. Along the linked lists, each storage contains various additional information. For example, both of them contain the word count.

To determine whether a word, a sentence, a paragraph, or a text division has ended, various properties has been assigned to the HTML tags. For example, not every HTML tag terminates a word. An HTML source to display “H₂O” would be “H< *sub* >2< /*sub* >O”. In addition to tags, there are other things that cause terminations. A word is terminated when a white-space is encountered and a sentence ends when a period is reached.

To keep track of which HTML tag terminates what, the researchers keep a linked

of properties for each tag along with various helper functions in a `tag_util.cc` and `tag_util.h` files. Each item in the list is a 'struct' that holds information about an individual tag. The entries are: `which_tag_name`, `name`, `is_word_terminating`, `is_sentence_terminating`, and `is_paragraph_terminating`. The "which_tag_name" is an enumerated type that spans across all of the HTML tags. For example, the type for the `< a >` is `A_TAG` and for the `< base >` it is `BASE_TAG`. This eliminates string comparison when two tags have to be compared. The next field in the structure is "name". It is a string containing the actual name of the tag. For the `< base >` tag, it is "base". Then there are three boolean fields: "is_word_terminating", "is_sentence_terminating" and "is_paragraph_terminating". These fields specify that when a given tag is encountered during parsing, a word, a sentence or a paragraph should be terminated. Please note the lack of "is_text_division_terminating". Only a `< table >` tag terminates a tag division so there was no need of adding that flag to the structure. There are only a handful of HTML tags that do not terminate a word. These are: `form`, `link`, `style`, `sub`, and `sup`. There are more HTML tags that do not terminate the sentence. These tags are: `a`, `base`, `b`, `center`, `font`, `form`, `i`, `s`, `small`, `strike`, `style`, `sub`, `sup`, and `u`. The number increases when it comes to paragraph terminating tags. There are less of them. The tags that do not terminate it are: `a`, `base`, `b`, `big`, `blink`, `br`, `font`, `i`, `image`, `s`, `small`, `strike`, `style`, `sub`, `sup`, and `u`. Table 3.1 shows which HTML tags terminate words, sentences, and paragraphs.

It is worth noting that a combination of tags is equivalent to another tag. For example, two `< br >` tags are equivalent one `< p >` tag. This also means that the two `< br >` in a text will break that text into two paragraphs.

While parsing, each unit, sentence, paragraph, and text division is flagged as containing certain property. Unit type is pre-determined in the parsing table. Each time some characters are extracted from the text, the parser 'sends signal' by calling one of the appropriate functions. Some of these functions are `SignalWord`, `SignalPunctuation`, or `SignalWhiteSpace`. These functions, mark various bits in a variable that is later used to determine what actions should be performed. Table 3.2 gives names

	<i>HTML Tags</i>									
Ends	a	base	b	blink	br	center	dd	dir	div	dl
Word	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Sentence	N	N	N	Y	Y	N	Y	Y	Y	Y
Paragraph	N	N	N	N	N	Y	Y	Y	Y	Y
Ends	dt	font	form	frame	frameset	h1	h2	h3	h4	h5
Word	Y	Y	N	Y	Y	Y	Y	Y	Y	Y
Sentence	Y	N	N	Y	Y	Y	Y	Y	Y	Y
Paragraph	Y	N	Y	Y	Y	Y	Y	Y	Y	Y
Ends	h6	hr	i	image	input	li	link	meta	ol	p
Word	Y	Y	Y	Y	Y	Y	N	Y	Y	Y
Sentence	Y	Y	N	Y	Y	Y	Y	Y	Y	Y
Paragraph	Y	Y	N	N	Y	Y	Y	Y	Y	Y
Ends	s	small	span	strike	style	sub	sup	table	td	th
Word	Y	Y	Y	Y	N	N	N	Y	Y	Y
Sentence	N	N	Y	N	N	N	N	Y	Y	Y
Paragraph	N	N	Y	N	N	N	N	Y	Y	Y
Ends	title	tr	tt	u	ul	“dontcare”				
Word	Y	Y	Y	Y	Y	N				
Sentence	Y	Y	Y	N	Y	N				
Paragraph	Y	Y	Y	N	Y	N				

Table 3.1: Terminating Properties of HTML Tags

of the actions and provides their short description. When no actions is needed to be performed, ACT_NONE is selected. Throughout the parsing, various characters are appended to the unit. The ACT_END_UNIT, adds given characters to the unit and passes the unit for processing. This is done by calling ProcessUnit function that takes as arguments value of the current unit and the enumerated type of the unit. This

Name	Value	Description
ACT_NONE	0000000	No actions are performed.
ACT_END_UNIT	0000001	A new unit is completed.
ACT_END_SENTENCE	0000010	A new sentence is completed.
ACT_END_PARAGRAPH	0000100	A new paragraph is completed.
ACT_END_TEXT_DIVISION	0001000	A new text division is completed.
ACT_FEED_OUT_OF_TAGS	0010000	These characters are outside of any HTML tag.
ACT_ADD_TO_WORD	0100000	These characters are part of the unit.
ACT_TREAT_AS_SINGLE_UNIT	1000000	These characters create one unit.

Table 3.2: Available Parser Actions

enumerated type will be explained later. The `ACT_TREAT_AS_SINGLE_UNIT` action is almost the same as the `ACT_END_UNIT` except for that it processes the given text as a separate, new unit. As the units are being processed, they are also added to a temporary sentence, temporary paragraph, and a temporary text-division called `current_sentence`, `current_paragraph`, and `current_text_division` respectively. Action `ACT_END_SENTENCE`, `ACT_END_PARAGRAPH`, and `ACT_END_TEXT_DIVISION` terminate appropriate collection, call a processing function (eg. `ProcessCurrentSentence()`, etc.), and start a new collection. The remaining two actions `ACT_FEED_OUT_OF_TAGS` and `ACT_ADD_TO_WORD` are simple. The `ACT_FEED_OUT_OF_TAGS` performs anything that has to be done with text outside of HTML tags. For example, it adds characters to the title string. The `ACT_ADD_TO_WORD` tells program to add the characters to the current word.

The previously mentioned function 'ProcessUnit' takes two parameters: a string value of the unit, and an enumerated type of the unit. This enumerated type will be assigned to the unit and kept in the unit storage. It can take one of the values: `UNIT_IS_WORD`, `UNIT_IS_POSSESSION`, `UNIT_IS_PUNCTUATION`, `UNIT_IS_WHITE_SPACE` or `UNIT_IS_NOTHING`. The `UNIT_IS_WORD` simply means that the given unit is

a word. The `UNIT_IS_POSSESSION` is equivalent to the `UNIT_IS_WORD`. It tells the unit storage that this is a word ended with “s” so that it can be associate with the word without the “s”. For example, Artur’s is the same as Artur. When a punctuation or white space is encountered in text of the web-page, it is stored as `UNIT_IS_PUNCTUATION` and `UNIT_IS_WHITE_SPACE` respectively. The last type, `UNIT_IS_NOTHING`, is a default value. When a unit of this type is passed to storage, the storage will ignore it.

Type	Description
<code>SENTENCE_IS_INFORMATIVE</code>	has enough information
<code>SENTENCE_HAS_LOW_WORD_COUNT</code>	not enough words
<code>SENTENCE_IS_MARKED_AS_LINK</code>	it lies between $\langle a \rangle$ and $\langle /a \rangle$
<code>PARAGRAPH_IS_INFORMATIVE</code>	has enough information
<code>PARAGRAPH_HAS_LOW_WORD_COUNT</code>	not enough words
<code>PARAGRAPH_HAS_LOW_SENTENCE_COUNT</code>	not enough sentences
<code>PARAGRAPH_IS_MARKED_AS_LINK</code>	it lies between $\langle a \rangle$ and $\langle /a \rangle$
<code>PARAGRAPH_IS_IMPORTANT_HEADING</code>	is an important heading
<code>PARAGRAPH_IS_HEADING</code>	is a heading
<code>TEXT_DIVISION_IS_INFORMATIVE</code>	has enough information
<code>TEXT_DIVISION_HAS_LOW_WORD_COUNT</code>	not enough words
<code>TEXT_DIVISION_HAS_LOW_SENTENCE_COUNT</code>	not enough sentences
<code>TEXT_DIVISION_HAS_LOW_PARAGRAPH_COUNT</code>	not enough paragraphs

Table 3.3: Sentence, Paragraph, and Text Division Types.

Once a sentence is ended, action `ACT_END_SENTENCE` executes main storage’s function `ProcessCurrentSentence`. Before the sentence is added to the sentence storage, it is assigned a type. It can be one of the three values: `SENTENCE_HAS_LOW_WORD_COUNT`, `SENTENCE_IS_MARKED_AS_LINK`, or `SENTENCE_IS_INFORMATIVE`. When a sentence does not exceed certain amount of words, it is marked as the type `SENTENCE_HAS_LOW_WORD_COUNT`. The amount of words is specified in `MINI-`

MUM_WORDS_IN_SENTENCE variable and the value used in this project is 3. When a sentence is contained within opening and closing tags (the `< a >` and `< /a >`), it is marked as SENTENCE_IS_MARKED_AS_LINK. If it is not marked by neither of these two values, it is marked as SENTENCE_IS_INFORMATIVE. Having this classification simplifies classification of the paragraphs and text_divisions, which in turn leads to a better annotation work.

Because the annotator attempts to create summary using a sample paragraph, a paragraph has more types than a sentence. Also, a paragraph can have more than one type (i.e. it can be a heading and a link). These are all possible types for a paragraph: PARAGRAPH_HAS_LOW_WORD_COUNT, PARAGRAPH_HAS_LOW_SENTENCE_COUNT, PARAGRAPH_IS_MARKED_AS_LINK, PARAGRAPH_IS_HEADING, PARAGRAPH_IS_IMPORTANT_HEADING, PARAGRAPH_IS_INFORMATIVE. A paragraph is PARAGRAPH_HAS_LOW_WORD_COUNT if it has less than 6 words (specified in MINIMUM_WORDS_IN_PARAGRAPH variable). It may seem like a small number for a paragraph, but if a paragraph has only sentence, six words could be enough to describe a web-site. Note, that it does not mean that this paragraph will be returned as annotation. The PARAGRAPH_HAS_LOW_SENTENCE_COUNT is used when the sentence contains less than desired number of sentences. The value has been set to 1 after testing showed that paragraphs with one sentence can be quite good. When a paragraph is within '`< a >`' and '`< /a >`', it is marked as link: PARAGRAPH_IS_MARKED_AS_LINK. When it is within header tags (`< h1 >` and `< /h1 >`, or `< h2 >` and `< /h2 >`, ..., `< h7 >` and `< /h7 >`), it is marked as PARAGRAPH_IS_HEADING. A paragraph with one sentence with at most 5 words (MAX_WORDS_IN_HEADING variable), can also become a heading if it is in font size 5, 6, or 7, or if it is in font size 3 or 4, and it is in bold casing or in all capital letters. Once the paragraph is marked as heading, it can become an important heading (PARAGRAPH_IS_IMPORTANT_HEADING), if it contains important words such as "Introduction", "About", "Description", etc. To satisfy a requirement of being informative (PARAGRAPH_IS_INFORMATIVE), the paragraph cannot have low word

count, low sentence count, nor it cannot be marked as link. This concludes the paragraph type description.

When it comes to the types of 'text divisions', it is simpler again. A text division can be marked as one of these types: `TEXT_DIVISION_HAS_LOW_WORD_COUNT`, `TEXT_DIVISION_HAS_LOW_SENTENCE_COUNT`, `TEXT_DIVISION_HAS_LOW_PARAGRAPHS`, or `TEXT_DIVISION_IS_INFORMATIVE`. Text division has low word count (`TEXT_DIVISION_HAS_LOW_WORD_COUNT`) when it has less than 18 words (defined in `MINIMUM_WORDS_IN_TEXT_DIVISION`). When it has less than the desired number of sentences or paragraphs, it is marked as `TEXT_DIVISION_HAS_LOW_SENTENCE_COUNT` or `TEXT_DIVISION_HAS_LOW_PARAGRAPHS` respectively. Both of these values are set to 1 because there can be a good single sentence that describes a web-page.

One may question why we keep sentences, paragraphs, or text-divisions that are not informative. They seem not to be used anywhere. It turns out that they are used in the last part of summarization. They are used to recreate the original paragraph before it is returned to the user.

3.3.2 Type of the Web-Page

To obtain better results in extracting the summary, one cannot approach each web-page in the same way. One web-page can have static content perfectly suitable for creating summary about the page. However, another web-page can have a dynamic content that is not related to what the page is about. The type of a web-page that I am referring to is a news-site such as `www.cnn.com`. If one were to use a statistical method to create a summary of a news-site, one would obtain something that relates to the events that happened on the day the annotation was created. To determine that the page is a news-site, one can use the cache control for that page.

A cache control specifies the amount of time that the page is allowed to be cached by the browser. This can be specified by either in server's HTTP response header or in the HTML of the document that is being downloaded. A web-server can specify the

cache control in the HTTP headers by using 'Cache-Control', 'Pragma' or 'Expires'. Just to be sure the browser understands it, some servers put more than one of those statements in the same header. The browsers may not understand all of the statements due to the fact that not all of the statements have been introduced at the same time. For example, 'Pragma' has been defined in the HTTP/1.0 and 'Cache-Control' in HTTP/1.1. An HTML page may contain 'meta http-equiv' tag that is equivalent to a line in the HTTP header. Also, some sites use 'meta http-equiv="refresh"' which tells a browser to refresh the page after specified number of seconds. This is frequently used by news-sites (for example, cnn.com or my.netscape.com use those).

The values for the 'Cache-Control' are: "public", "private", "no-cache", "no-store", "no-transform", "must-revalidate", "proxy-revalidate", "max-age=delta-seconds", "s-maxage=delta-seconds", and cache-extensions. The first three values control what is cache-able. The "public" tells that the cache can be accessible to anybody (for example, proxy server). The "private" tells that only the person who requested it can cache it. The "no-cache" indicates that the page should not be cached. From these three, only "no-cache" is of interest to this project. The other values except for "max-age", are also not important to this research. The "no-stores" prevents sensitive information from leaking out (for example, being backup and stored on backup tapes). The "no-transform" tells that the content cannot be converted to another type for caching purposes (for example, an image cannot be converted to another format in order to save on the storage requirements). The "must-revalidate" tells that a page must be revalidated before its cache is used. The "proxy-revalidate" works similar to "must-revalidate" except that it doesn't include end user agents (for example, it applies to the proxy). The "max-age" specifies the period of time that the cache is allowed to be valid. The "s-maxage" is similar to "max-age" except it is destined for the shared caches. The cache-extensions can be any set of 'field=value' pairs. As pointed out before, only "no-cache", and "max-age" are important to this work. These two allow us to tell that the information on the page is very dynamic (as in the case of a news-site).

Although originally designed to hold a list of values, currently the only valid value for “pragma” is “no-cache”. It tells us that the page should not be cached. Neither proxy or the user agent is allowed to do that [20]. The last field in the HTTP header that controls the cache is the ‘Expires’ field. According to the HTTP protocol, the value for “expires” is the date after which the cache becomes stale[21]. However, in experiments, it was noticed that some broken server give a value of “0” or “now”. Although these values are not valid, they were used for the cache control.

Once the web-site has been determined to be a news-site, all the anchor links are checked for words that would indicate a page with a description page. Some of these words are: “About”, “Introduction”, or “Abstract”. If the links contain those words, a web-page for them is fetched and annotation system parses it. The process is described in more details in the next section.

3.3.3 Creating Annotation

This section describes how annotation is created. The process is illustrated in figure 3.5.

The process starts with obtaining a URL. A page for the URL is fetched and parsed. If during the parsing the page has been found to be using frames, each frame is downloaded, and all the HTML sources are combined into one page. The pages are ‘glued’ together using the HTML table construct where the individual frames are turned into table cells. The document source is then parsed and the various storages (unit storage, sentence storage, etc.) are filled with data.

The storages make the data access easy. Meta description can be obtained with one function call. If it is found, the meta description is returned as the annotation. Otherwise, the page is checked if it is cache-able. If it is not cache-able, all the text in the links from the web-page is checked for keywords that would indicate a better description page. Some of these words are “About”, “Abstract”, “Introduction”. If such a link exists, the page for it is fetched and the processed is restarted using this

page. However, if such a link does not exist or the link has been crawled already, a sample paragraph is extracted. The process of extracting sample paragraph is explained next.

A sample paragraph is extracted from the best fitting text division. The text division that fits best is picked based on the number of points that the division received. The points are given to the divisions based on various factors. One of them is for having phrases from Keith's Phrase Rater. Another factors are the properties of the text-division. All the text divisions that are not informative, get score of 0. Then the division obtain a point "boost" based on the position in the web-page. Text divisions on top, get more points than the ones on the bottom. The boost is computed based on the word offset for the first word in the text-division rather the text-division count. This prevents a situation where in a web-page with five text-divisions, the last text-division get low score even though it has two thousand words where the first four text-division had only 5 words each. The text-division also obtains extra points for having important headings. The important headings are the ones described in the previous section. Once the points are calculated for all of the text divisions, the one with the highest amount of points is returned for further processing. The next step is to pick the sample paragraph from that text division.

A sample paragraph is extracted from the text division in a similar manner as the text-division was selected. However, more processing is done here. Once again phrases from Keith's Phrase Rater are used for part of the scoring. Note that in both cases the scores are pre-calculated at the sentence level. Every time a word is added to the sentence, it is checked if it matches any of the phrases. If it does, the sentence gets extra points. The ranking begins with paragraph having nothing but the points from the sentences. Then, it receives points for the position in the web-page, loses all points for being a link, and gets extra points for being right after an important headings. Also points are given to paragraphs that meet minimum requirements. These requirements are minimum word count and minimum sentence count. The minimum word count is set to 10 (MIN_WORDS) or to the document

word average, which ever is greater. Similar case is for the minimum sentence count. It is 1 (MIN_SENTENCES) or the average number of sentences in all paragraphs. After all the points are calculated, a paragraph with the highest number of points is returned.

If a sample paragraph cannot be extracted, the system will attempt to extract individual sentences. However, if the page is cache-able, it will attempt to crawl introduction links just like it would when the page is not cache-able. If that fails or is not applicable (i.e. crawling has already been done), the sample sentences are extracted.

The sample sentences are the last resort in the annotation extraction. They are extracted from the document by taking two sentences that have the highest score. The score is based on the Phrase Rater's phrases. Before they are returned to user, they are checked for duplication. There might be two or more the same sentences in a document and due to their similarity, they get similar amount of points. Also there might be two sentences that are the same except for an extra word at front or at the end. To eliminate that, a similarity is calculated on the two sentences. If it exceeds 90%, another sentence is extracted.

3.3.4 Returning the Results

There are two ways to start up the Automatic Annotation program to create and obtain back the summary of a web-page. One way to do so is through invocation of a command-line program called 'autoadder'. The second is through a C++ library Application Programming Interface (API).

The 'autoadder' command line is an already existing program in the INFOMINE. It takes a URL, filename, and a command. The command can be one of the commands specified in table 3.4. So to obtain the annotation, the user can simply specify the -d option along with the filename and the URL.

Short Option	Long Option	Description
-t	-title	title should be extracted
-u	-urls	urls should be extracted
-d	-description	the description should be extracted
-a	-authors	author should be extracted
-g	-get-text	text without HTML should be extracted
-w	-wordcount	wordcount should be extracted
-i	-images	images should be extracted
-s	-strip	HTML references should be stripped from the returned URLs
-n	-no-tags	specifies that the output should not contain tags
-h	-help	displays help.

Table 3.4: Command options for the 'autoadder' program.

The C++ library API allows a programmer to link directly to this thesis project. In addition to extracting the summary, it is able to obtain various other informations. For example, it can get a list of all URLs embedded in a HTML page. This is the full list of available functions:

brief: Parses a given string.

param: root_url the URL of the document

param: html the HTML source of the document

void ParseFromBuffer(const std::string &root_url, const std::string &html);

brief: Parses a give file.

param: root_url the URL of the document

param: parser_input a file handle to the document

void Parse(const std::string &root_url, FILE *parser_input);

brief: Parses a give file.

param: root_url the URL of the document

param: filename a file-name ofthe document

**void Parse(const std::string &root_url, const std::string &filename) throw
(std::exception);**

brief: Activates collecting text without html

void ActivateCollectingText(void);

brief: Obtains the title of the document

return: the extraced title

std::string GetTitle(void);

brief: Obtains a desired meta tag.

param: meta_name the name of the meta tag

return: Empty string if such meta-tag wasn't found otherwise the value of the tag.

std::string GetMeta(const std::string &meta_name);

brief: Obtains all the URLs embedded in the document

param: url_list A reference to a std::list of urls.

void GetURLs(std::list<std::string> * const url_list);

brief: Returns a pointer to a reference to a list of urls.

std::list<std::string> * GetImages(void);

brief: Returns a list of sentences in the passed reference 'sentence_list'

void GetSentences(std::list<std::string> * const sentence_list);

brief: Returns a list of paragraphs in the passed reference 'paragraph_list'

void GetParagraphs(std::list<std::string> * const paragraph_list);

brief: Returns a list of text_divisions

param: text_divisions_list a reference to a list used as a return value

void GetTextDivisions(std::list<std::string> * const text_division_list);

brief: Returns an annotation of a web-page.

param: annotation A reference to a string that will be set with annotation.

param: annotation_method the method in which the annotation was created.

```
void GetAnnotation(std::string * const annotation, AnnotationMethod-  
Type * const annotation_method = NULL);
```

brief: Returns a list of words in reference 'word_list'

```
void GetWords(std::list<std::string> * const word_list);
```

brief: Returns a given http-equiv. If not found, empty string is returned.

```
std::string GetHttpEquiv(const std::string &equiv);
```

brief: Returns true if a style sheet has been found in the document

```
bool IsStyleSheetFound(void);
```

brief: Returns a a string containing text without HTML tags

```
std::string *GetTextWithoutHtml(void);
```

brief: Returns the number of words that have been found in the document

```
unsigned GetTotalWordCount(void);
```

brief: Inserts important words into the system

```
void InsertImportantWords(const std::list<std::string> &words_list);
```

brief: Tells whether according to HTTP headers the page should be cached.

```
bool CanPageBeCached(void);
```

Using this API, this is how a programmer would uses the library:

```
#include < extractor.h >  
  
int main() {  
    std::string url="http://some.url.com";  
    std::string filename="some_url_file.txt";  
    Extractor::Extractor extractor;  
    extractor.Parse(url, filename);  
    std::string annotation, method_as_string;  
    Extractor::AnnotationMethodType annotation_method;  
    extractor.GetAnnotation(&annotation, &annotation_method);  
    std::cout << annotation;
```

```
}
```

If the program is called `test.cc`, this is all is needed to compiled the program:

```
g++ test.cc -lHtmlParser -o test
```

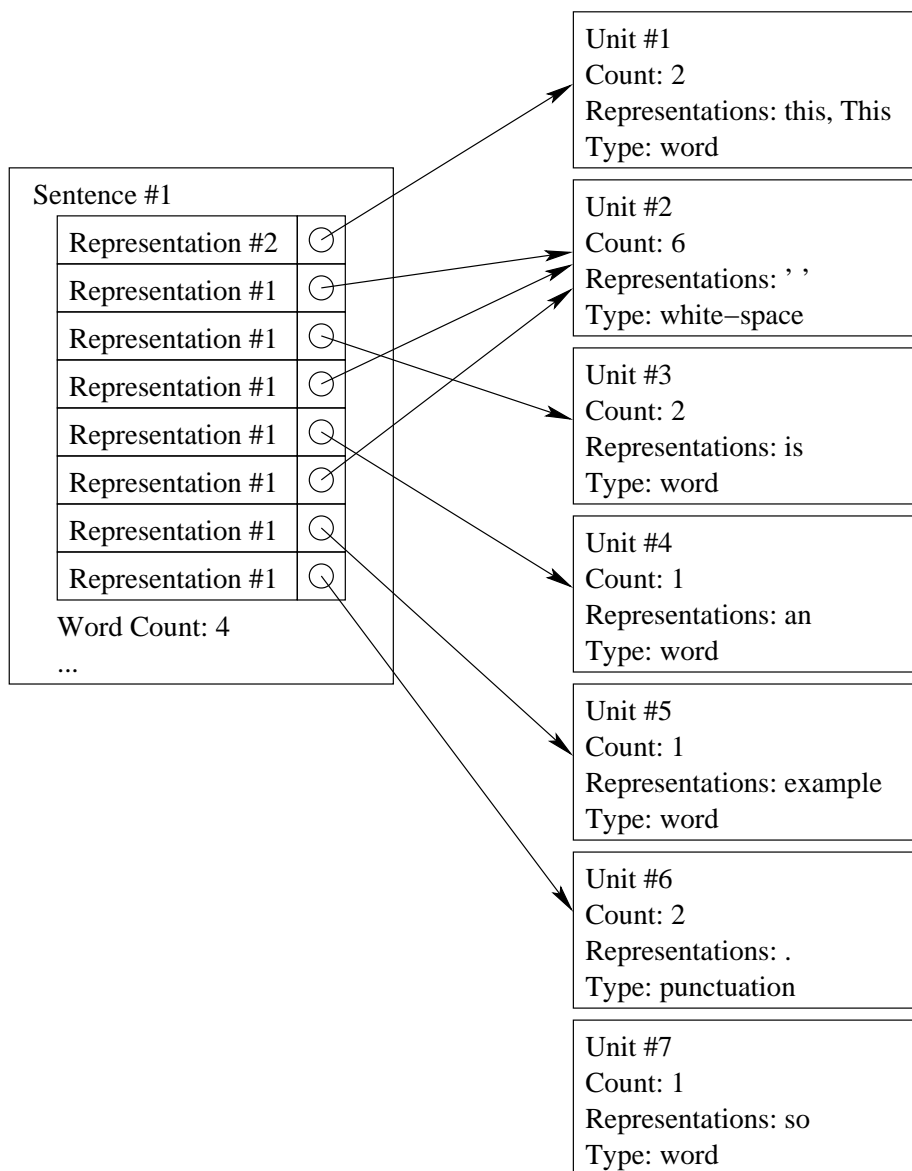


Figure 3.3: Sentence representation of 'This is an example'

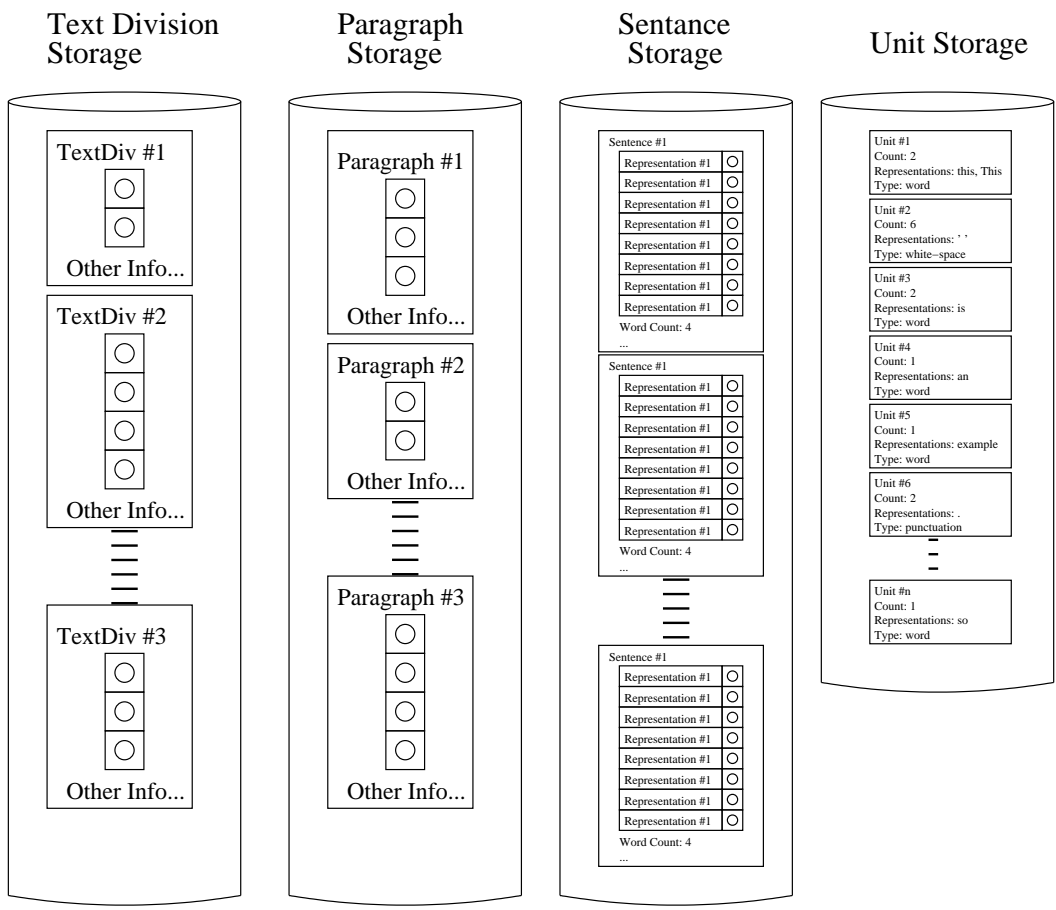
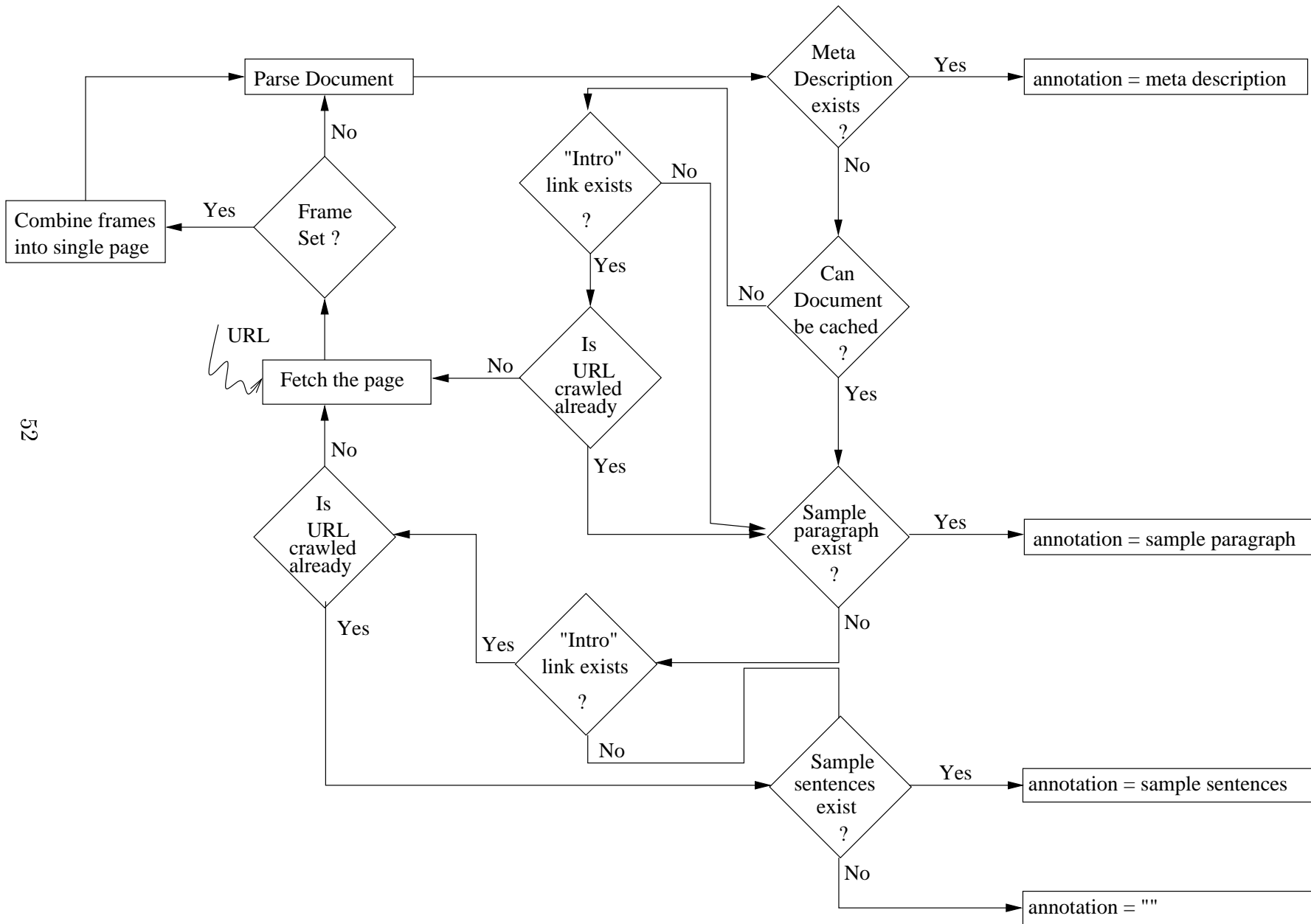


Figure 3.4: The Four Storages



Chapter 4

Experiments and Results

Originally, this thesis project used GNU Bison[22] for parsing web-pages. The Bison's home-page tells us about the project. "Bison is a general-purpose parser generator that converts a grammar description for an LALR context-free grammar into a C program to parse that grammar. Once you are proficient with Bison, you can use it to develop a wide range of language parsers, from those used in simple desk calculators to complex programming languages.[22]"

Bison worked together with Flex. Flex would pass tokens to Bison every time a HTML tag or a word was encountered. Bison would then check if the grammar was correct and perform desired actions. The grammar for Bison was constructed using the HTML 4.01 standard. Once done, the grammar table contained hundreds of lines. Despite the size, however, the table was not complex. It was easy to maintain it and perform any additional changes. When it came to start testing it on the real-life web-pages, the situation got worse. One of the first changes to the grammar was removing the requirement of having the `<html>`, `<head>`, and `<body>` tags. Although, this increased the complexity of the grammar, the change was not major. However, as more pages were parsed, more changes to the grammar were needed. It became clear that the web-masters were not interested in following the standard. Judging by the web-pages that they have written, they have done the minimal amount of work that

was required to get a page displayed on one type of browsers. For example, a lot of closing tags would be missing, opening and closing tags would be interpolated (eg. `some text< /font>< /b>`), or values of HTML tags would be missing quotes. It became an endless cycle of finding broken web-pages and patching the grammar to handle them. In the end, the grammar became too complex to be usable. Also, there was still a lot of pages that the grammar would not recognize. As a result, this approach has been completely abandoned. The next approach (which is the current approach) has been written from scratch. It no longer assumes that the tags will be in any particular order. It treats them on individual basis. Once the new parser was in place, another part had to be re-written. It was the storage part of the project.

During the execution of the program, the parser would place various pieces of text in numerous arrays of string type. For example, there was an array that contained all words, or an array that contained all the sentences where an individual sentence was just one string. The processing of the data was complex. To display a sentence, there had to be string processing done to check if the last character was a period. Since extra white-space could follow the period, more processing was needed. There had to be a loop that would check if the last character is a period. If instead period, we encountered a white-space character, that last character was deleted and the loop would continue. If anything else was encountered, a period had to be added. In the new approach a sentence is a class that along other information contains a linked-list of 'units' that represent the sentence. It is now trivial to test if a sentence is terminated by a period. White-space characters can no longer follow the period so just the last unit's type have to be checked to see if it is a punctuation.

In the previous design, it was also difficult to do anything with the extracted data. There were arrays that would keep the count of the number of words in sentences, in paragraphs, and in text-divisions. All these arrays had to be iterated to extract any desired information. Also, there was no easy way to extract individual paragraphs from a text-division. To do that, a string processing was needed to divide into para-

graphs, the string containing an entire text-division. From there, the program had to match each paragraph with the paragraphs contained in the array that stored all of the paragraphs. That matching was needed to obtain other information that was already available (eg. word count) without dividing the paragraph into smaller pieces (i.e. into sentences and then into words). At this point, it was clear that a replacement was needed. The current approach of the storage was re-written from scratch. Although it took me some time, it helped make the code more clear and maintainable. It drastically decreased the amount of space needed for storage since sentences, paragraphs, and text-divisions were represented as linked-lists of pointers to smaller units instead of being represented as long strings. Also, the revamp made any further changes easy to accomplish.

To improve the results, I have considered using an Optical Character Recognition. Some web-sites use images to display various information. For example, the images on a web-site may contain a title, company name, or section headings. The OCR that I have considered using was GNU Optical Character Recognition (GOOCR)[23].

Since GOOCR is not able to download documents and it can recognize only few image formats, other programs were needed. To download the document, I used previously described cURL and Libcurl [18] and to convert an image to another format, I used ImageMagick[24]. ImageMagick is described on its home-page in the following way: “ImageMagickTM is a robust collection of tools and libraries to read, write, and manipulate an image in many image formats (over 68 major formats) including popular formats like TIFF, JPEG, PNG, PDF, PhotoCD, and GIF. With ImageMagick you can create images dynamically, making it suitable for Web applications. You can also resize, rotate, sharpen, color reduce, or add special effects to an image and save your completed work in the same or differing image format. Image processing operations are available from the command line, as well as through C, C++, Perl, or Java programming interfaces. “ [24].

The process of optical recognition was designed to be a set of system pipes that

was executed and read by a C++ program. These commands would communicate with the C++ through `popen(3)` call. That `system(3)` command was:

```
“curl -L -f -s -m 30 image_URL 2> error_log | convert - pnm:- 2> error_log | gocr  
- 2> error_log”
```

This told `cURL` to download an image whose URL was “`image_URL`” and dump it to standard output. The options told `cURL` to fail without any output at all on server errors (`-f`), to follow HTTP relocations (`-L`), not to display a progress meter (`-s`), and to time-out after 30 seconds (`-m 30`). The `cURL`'s output, which was the raw data, was then piped to `ImageMagick`.

The ‘`convert`’ command is one of the command-line tools that the `ImageMagick` comes with. It allows converting images from one format to the other. It is able to determine the format of an incoming image without being told what it is (through an option or the filename). This ability greatly simplifies this piping process. It eliminates the need to download JPEG files first, then PNGs, then GIFs, and so forth. In other words, all the images can be processed in one iteration. There are two options to ‘`convert`’ in this `system(3)` call: the ‘`-`’ and ‘`pnm:-`’. The first option, ‘`-`’, tells “`convert`” to accept the input from the standard input. The second option specifies that the output should be a PNM file dumped to standard output. The PNM is one of the few image formats accepted by the `GOOCR`. Although explanation of the PNM format is beyond the scope of the paper it is worth to mention that there are three PNM (or “Portable anymap”) file formats: PBM for bilevel images, PGM for gray level images, and PPM for color images[25].

The last command in the pipe is the `GOOCR` itself. The only option for it is the ‘`-`’ which tells it to read the image from standard input. By default, `GOOCR` returns the extracted text to the standard output as ASCII characters. This is a desired behavior. It allows the C++ program to read it as an input through the `pipe(3)`. `GOOCR` has also various other options but they tend to be specific to an image that is about to be converted to text.

Once the processing of the image is done, the program checks the error log for any messages. The standard error redirections (the '2>') in system call write any error messages to the "error_log" so if something went wrong, the error log would not be empty. If there was no errors, the extracted text from an image, would be paired with that image, and then displayed on a web-browser. The testing environment consisted of a simple web-page that allows for typing in a URL of page containing images. The URL was then passed to the previously described C++ program, and then the image was displayed together with the recognized text.

The use of GOCR was not successful in this thesis project. The number of successfully extracted text from images was extremely small to the point of not being useful. Mainly, it was due to immaturity of the software. The GOCR has been in development since June 2000 so by the time of the experiment, it was in development for only a year. However, if it continues to be actively developed so it may become a useful tool in generating summaries from web-pages.

During the testing, the need for a dynamic minimum size of the extracted paragraph became apparent when the system created an annotation for

<http://www.hort.purdue.edu/newcrop/proceedings1990/V1-331.html>.

This is a long paper containing large paragraphs. Because it was at the beginning of the document and it contained a couple sentences, the string: "Ferguson, L. and M. Arpaia. 1990. New subtropical tree crops in California. p. 331-337. In: J. Janick and J.E. Simon (eds.), Advances in new crops. Timber Press, Portland, OR." became an annotation. This led to realization that the minimum should be based on the average of words in the documents and on the average of sentences in a paragraph. Once this has been accomplished, the annotation extraction has improved.

Now that the minimum size of annotation has been set, another problem came out while summarizing

<http://www.gbf-braunschweig.de/DSMZ/bactnom/bactname.htm>

Before the main text there are a few paragraphs explaining availability of additional

data. These paragraphs match the size, contain keywords, and are ahead of other paragraphs so one of them became an annotation. To fix the problem, the system started checking all the headings in the documents and marking some of them as 'important headings'. In this page, "Introduction" is a heading because it resides in between `< h4 >` and `< /h4 >` tags. Then, because it contains a keyword "introduction", it became an important keyword. With the important heading, the immediate paragraph got more points. Which in turned, created correct annotation.

One thing that was missed in the in the previous problem, was assigning extra points to the text-divisions with the important heading. The web-site <http://geopubs.wr.usgs.gov/prof-paper/pp1623/> made the problem visible. The page has pretty much two tables. One small table contains an abstract and the other contains everything else. Because of the size, the second table has been picked as a better text-division and the abstract was missed. Giving points to the text-division based on the important headings, fixed this problem.

The site <http://yosemite.epa.gov/r9/sfund/overview.nsf/ef81e03b0f6bcdb28825650f005dc4c1/68b4c563033a942f8825660b007ee675?opendocument> called for another improvement. The page has a heading "SITE DESCRIPTION AND HISTORY" that should be marked as important heading. However, the system didn't pick it up because no header tags (eg. `< h1 >`, `< h2 >`, etc.) was used. Instead, the header has been created using the 'font' tags. To take care of the problem, a subsystem has been created that keeps track of font size throughout the parsing. Once the font size exceeded a certain size in a small sentence, that sentence was tested for being a heading.

At this point, the parser still did not find all the important headings. On the site http://www.genome.ou.edu/protocol_book/protocol_index.html there is an "introduction" heading. However, it neither created by heading tags nor font tags. Instead, it was created using bold tags (`< b >` and `< /b >`). The problem

was solved after the system started recognizing sentences with limited number of words but in bold text as the headings. As a side not, this site also proved that `< pre >` tags should be paragraph terminating.

There were also sites that showed how badly HTML can be and how careful the annotator should be. The page

<http://www.cr.nps.gov/seac/outline/index.htm>

would crash the system because it contains empty heading. In example, the code `< h1 >< /h1 >`.

There was also a problem with links that would be combined to form a paragraph. On a site,

<http://www.kidsdomain.com/holiday/earthday/>

both of the columns of links were combined into a paragraph that would become an annotation. That happened because a paragraph would be marked as link only if it would reside between a single “`¡a¡`” and “`¡/a¡`”. A small fix where all the non-whitespace had to be embeded in links in order for the paragraph to be a link, has been implemented. That eliminated the problem.

A site that was used for a major idea was <http://www.cnn.com/>. It allowed testing of cache-able data and crawling the links. Once the program was able to find <http://www.cnn.com/INDEX/about.us/>, another problem surfaced. This page uses `< span >` tags extensively. Due to misunderstanding, that tag did not terminate a sentence, paragraph, or the text-division. Once that was changed, appropriate divisions has been created. Because of phrase 'CNN.com is' in a paragraph, that paragraph is picked as the winner.

When the property of 'span' tag has been changed, another problem surfaced. Site <http://www.consrv.ca.gov/dog/>, contains ` ` `` in the middle of the best paragraph. The span tag would break the paragraph in two and return improper result. As a result, the span tag breaks sentences only if it contains words in between.

For the testing the annotator on pages where no description nor sample paragraph were available, the following URLs has been used:

<http://www.ansi.okstate.edu/library/>

<http://www.ncgia.ucsb.edu/>

<http://www.im.ac.cn/>

<http://www.nws.noaa.gov/oh/hic/>

<http://www.os.dhhs.gov/>

<http://www.megago.com/l/>

<http://ws4.niai.affrc.go.jp/dbsearch2/pmap/plmap/plmap.html>

<http://www-bio.llnl.gov/bbrp/bbrp.homepage.html>

<http://www.ces.ncsu.edu/hil/>

<http://www.ie.embnet.org/embnet.html>

<http://www.mcp.com/>

<http://wwwicic.nci.nih.gov/trials/>

Since the meta description nor sample paragraph was available, the system uses the title sentence as one of the possible sentences that can be extracted. It works out really well. For example, in <http://www.ansi.okstate.edu/library/> the entire page is using Macromedia Flash. There is no text on it so the system uses the only available sentence – the title sentence. Similar case is in <http://www.ncgia.ucsb.edu/> This time, the entire page is composed of images. Once again, the title sentence comes to the rescue. In both of the cases, the sentence is informative and describes the page well. This is also a case for most of these URLs. The title sentence would be used to describe the web-page.

As for testing of sample sentences extraction, these are the URLs used:

<http://omni.ac.uk/>

<http://www.ohsu.edu/clinweb/wwwv1/>

<http://megasun.bch.umontreal.ca/protists/protists.html>

<http://ortho84-13.ucsd.edu/MusIntro/Jump.html>

<http://www.nmnh.si.edu/gopher-menus/IndexoftheUSNMBirdsCollection.html>

<http://molbio.info.nih.gov/molbio/desk.html>

It was a difficult task to find URLs that would force the annotation system to use the sample sentences. Most of the web-pages that have been tested had enough information for paragraph extraction or they would have small amount of data that they would classify to the category described above (i.e. title sentence would be extracted).

In general, the annotation system proved to be working well. It generated appropriate summaries for most of the pages. However, there are improvements that would make the system work even better. They will be listed in the “Future Work” section.

Chapter 5

Conclusion and Future Work

5.1 Summary

In summary, summarizing systems are essential in the management of search engines and other web page collection systems. Although there are several approaches used to generate summaries, each has unique limitations. This research project explored a summarizing approach that recognizes the content of web pages and uses an appropriate technique to create a summary page. The process included parsing and converting the pages into a set of various objects, identifying the type of the web page, and then using the objects and the appropriate technique to create a summary. The summary could be a meta description, a sample paragraph, or sample sentences coming from a web-page given by a user. The research also showed that if appropriate or necessary the system will crawl the links in that page to obtain better content for the summary.

The work generated by this thesis is intended for use by the INFOMINE Project developed by the University Library at the University of California at Riverside. It will serve as a tool in creating annotations for the INFOMINE—a job currently performed manually by project personnel. Its integration in the upcoming INFOMINE crawler will automatize the annotating process.

5.2 Future Work

There are several things that can be done to improve the annotator:

- Use a limited JavaScript interpreter. Using the interpreter could enable crawler part of the annotator to obtain JavaScript generated redirections, or get a source of the document that was generated by the JavaScript.
- Use OCR. The GOCR is in active development so it is more likely to be working fine in the future. Using it could provide annotator with keywords or sentences that would strengthen the already available phrases.
- Convert Cascading Style Sheets (CSS) into pure HTML. The CSS separates what is being displayed on a page from how it is being displayed. For example, XHTML completely eliminated the “font” tag. The CSS pre-processor could re-create such tags. This would make the rest of the process more accurate.

These items are the main improvements that could be done to the annotator. Although these enhancements would increase the time needed for entire process, the resulting annotation would be more accurate.

Bibliography

- [1] The history of the internet.
- [2] Infomine, scholarly internet resource collections.
- [3] How do search engines work?
- [4] Excite @home.
- [5] Hotbot / inktomi.
- [6] www.infoseek.go.com.
- [7] Cecile Paris Einat Amitay. Automatically summarising web sites - is there a way around it?, 2000.
- [8] Adam L. Berger and Vibhu O. Mittal. OCELOT: a system for summarizing web pages. In *Research and Development in Information Retrieval*, pages 144–151, 2000.
- [9] I. Good. The population frequencies of species and the estimation of population parameters., 1953.
- [10] P. Clarkson and R. Rosenfeld. Statistical language modeling using the cmu-cambridge toolkit., 1997.
- [11] Chris Buckley Mandar Mitra, Amit Singhal. Automatic text summarization by paragraph extraction.

- [12] Jade Goldstein, Mark Kantrowitz, Vibhu O. Mittal, and Jaime G. Carbonell. Summarizing text documents: Sentence selection and evaluation metrics. In *Research and Development in Information Retrieval*, pages 121–128, 1999.
- [13] Vibhu O. Mittal, Mark Kantrowitz, Jade Goldstein, and Jaime G. Carbonell. Selecting text spans for document summaries: Heuristics and metrics. In *AAAI/IAAI*, pages 467–473, 1999.
- [14] Julian Kupiec, Jan O. Pedersen, and Francine Chen. A trainable document summarizer. In *Research and Development in Information Retrieval*, pages 68–73, 1995.
- [15] Nec research index.
- [16] Minisql and lite by hughes technologies pty ltd.
- [17] Mysql database management system.
- [18] Curl and libcurl.
- [19] Flex the fast lexical analyzer generator.
- [20] Http request fields: pragma.
- [21] Object metainformation: 'expires'.
- [22] Bison - gnu project - free software foundation (fsf).
- [23] Gocr open-source character recognition.
- [24] Imagemagick - convert, edit, and compose images.
- [25] Pnm and rle file formats.

Appendix A

Source Code