Do Not Crawl in the DUST: Different URLs with Similar Text

Uri Schonfeld

Do Not Crawl in the DUST:
Different URLs with Similar Text

Research Thesis

Submitted in Partial Fulfillment of the Requirements
for the Degree of Master of Science
in Electrical Engineering

Uri Schonfeld

Submitted to the Senate of the Technion - Israel Institute of Technology SH'VAT 5766
HAIFA FEBRUARY 2006

Acknowledgments

I would like to thank my supervisors, Dr. Ziv Bar-Yossef and Dr. Idit Keidar, for their patience, for believing in me, and for bringing me down to earth when needed. I could not have asked for better supervisors, and I truly feel lucky to have had them both. I would also like to thank Prof. Israel Cidon, Prof. Yoram Moses, and Dr. Avigdor Gal for their insightful and helpful input. Finally, I would like to thank Tal Cohen and the forum site team, and Greg Pendler and the `ee.technion.ac.il` admins for providing us with access to web logs and for technical assistance.

The generous financial help of the Technion is gratefully acknowledged

Contents

List of Figures

List of Tables

Abstract

We consider the problem of DUST: Different URLs with Similar Text. Such duplicate URLs are prevalent in web sites, as web server software often uses aliases and redirections, translates URLs to some canonical form, and dynamically generates the same page from various different URL requests. We present a novel algorithm, *DustBuster*, for uncovering DUST; that is, for discovering rules for transforming a given URL to others that are likely to have similar content. DustBuster is able to mine DUST effectively from previous crawl logs or web server logs, *without* examining page contents. Verifying these rules via sampling requires fetching few actual web pages. Search engines can benefit from this information to increase the effectiveness of crawling, reduce indexing overhead as well as improve the quality of popularity statistics such as PageRank.

List of Symbols

Abbreviations and Acronyms

- DUST – Different URLs with Similar Text.
- DUST rules – A transformation rule that when applied to one URL results in a different URL likely to have similar text.
- **DustBuster** – The algorithm presented in this thesis for detecting DUST rules.
- URL – Uniform Resource Locator.

Symbols

- $D_{\alpha,\beta}$ – The set of disqualified envelopes for a pair (α, β).
- $E_{\mathcal{L}}(\alpha)$ – the set of envelopes of α in URLs that: (1) appear in the URL list $\mathcal{L}$; and (2) have α as a substring.
- I_u – Interval between the minimum and the maximum size values encountered for URL u in a URL list.
- $\mathcal{L}$ – The URL list.
- MAD – The maximum absolute deficiency used by the "Eliminating Redundant Rules" phase to compare the support of two rules.
- MRD – The maximum relative deficiency used by the "Eliminating Redundant Rules" phase to compare the support of two rules.
- MS – The minimum support size. Any rules below this support are not considered.
- MW – The maximum window size considered during the "Eliminating Redundant Rules" phase.
- O – Buckets bigger than some threshold T are called *overflowing*, and all envelopes pertaining to them are denoted collectively by O.

- S – The maximum substring length.
- SUPPORT$_{\mathcal{L}}(\phi)$ – The set of instances (u_1, u_2) of the DUST rule ϕ, for which both u_1 and u_2 appear in the URL list $\mathcal{L}$.
- T – The maximum bucket size.

Chapter 1

Introduction

The DUST problem. The web is abundant with DUST, Different URLs with Similar Text [18, 12]. For example, the URLs `http://google.com/news` and `http://news.google.-com` return similar content. Adding a trailing slash or `/index.html` to either will still return the same result. Many web sites define links, redirections, or aliases, such as allowing the tilde symbol ˜ to replace a string like `/people`, `/users`, or `homepages`. Some web servers are case insensitive, and can be accessed with capitalized URL names as well as lower case ones. Some sites allow different conventions for file extensions– `.htm` and `.html`; others allow for multiple default index file names – `index.html` and `default.html`. A single web server often has multiple DNS names, and any can be typed in the URL. As the above examples illustrate, DUST is typically not random, but rather stems from some general rules, which we call DUST *rules*, such as "˜" $\rightarrow$ "`/people`", or omitting "`/default.html`" at the end of the URL. Additionally, our findings show that these rules remain valid for months and even years and are valid for new pages as well as old ones.

Moreover, DUST rules are typically not universal. Many are artifacts of a particular web server implementation. For example, URLs of dynamically generated pages often include parameters; which parameters impact the page's content is up to the software that generates the pages. Some sites use their own conventions; for example, a forum site we studied allows accessing story number "num" both via the URL `http://domain/story?id=num` and via `http://domain/story_num`. Our study of the CNN web site has discovered that URLs of the form `http://cnn.com/money/whatever` get redirected to `http://money.cnn.com/whatever`. In this thesis, we focus on mining DUST rules within a given web site. We are not aware of any previous work tackling this problem.

Knowledge about DUST rules can be very valuable for search engines as well as for large scale investigation of the web, web caching, and web site traffic analysis. The fact that pages do not have unique identifiers creates problems in almost every large scale software that deals with the web. Knowledge about DUST rules allows for a *canonical* URL representation, where each page has a single canonical name. Thus, canonization using DUST rules can reduce crawling overhead, reduce indexing overhead, reduce caching overhead [18, 12], and increase the accuracy of any statistical study of web pages, e.g., computing their popularity [8, 17]. For example, in search engines, using knowledge about DUST rules can increase the effectiveness of web crawling by eliminating redundant accesses to the same page via multiple URLs. In one crawl we examined, the number of URLs fetched would have been reduced by 26%. We focus on URLs with *similar* contents rather than identical ones, since different versions of the same document are not always identical; they tend to differ in insignificant ways, e.g., counters, dates, and advertisements. Likewise, some URL parameters impact only the way pages are displayed (fonts, image sizes, etc.) without altering their contents.

Standard techniques for avoiding DUST are not based on discovering site-specific DUST rules. Universal rules, such as adding `http://` or removing a trailing slash are used, in order to obtain some level of canonization. Additional DUST is found by comparing document sketches such as shingles, which are hash-based summaries of page content. However, this is conducted on a page by page basis, and all the pages must be fetched in order to employ this technique. By knowing DUST rules, one can dramatically reduce the overhead of this process. But how can one learn about site-specific DUST rules?

Detecting DUST from a URL list. Contrary to initial intuition, we show that it is possible to discover likely DUST rules without fetching a single web page. We present an algorithm, *DustBuster*, which discovers such likely rules from a list of URLs. Such a *URL list* can be obtained from many sources including a previous crawl or the web server logs. The rules are then verified (or refuted) by sampling a small number of actual web pages.

At first glance, it is not clear that a URL list can provide reliable information regarding DUST, as it does not include actual page contents. We show, however, how to use a URL list to discover two types of DUST rules: *substring substitutions* and *parameter substitutions*. A substring substitution rule $\alpha \to \beta$ replaces an occurrence of the string α in a URL by the string β. A parameter substitution rule replaces the value of a parameter in a URL by some default value or removes the parameter altogether. Thanks to the standard syntax of parameter usage in URLs, detecting parameter substitution rules is fairly straightforward. Most of our work therefore focuses on substring substitution rules, which are similar to the "replace" function in many editors.

DustBuster uses three heuristics, which together are very effective at detecting likely DUST rules and distinguishing them from false rules. The first heuristic is based on the observation that if a rule $\alpha \to \beta$ is common in a web site, then we can expect to find in the URL list multiple examples of pages accessed both ways. For example, in the site where `story?id=` can be replaced by `story_`, we are likely to see in the URL list many different pairs of URLs that differ only in this substring; we say that such a pair of URLs is an *instance* of the rule "`story?id=`" $\to$ "`story_`". The set of all instances of a rule is called the rule's *support*. Our first attempt to uncover DUST is therefore to seek rules that have large support.

Nevertheless, some of the rules that have large support are not DUST rules. For example, when examining one site we found instances such as `http://movie-forum.com/story_100` and `http://politics-forum.com/story_100` which support the false rule "`movie-forum`" $\to$ "`politics-forum`". Another example of a false rule with large support is "1" $\to$ "2", which emanates from instances like `pic-1.jpg` and `pic-2.jpg`, `story_1` and `story_2`, and `lecture.1` and `lecture.2`, none of which are DUST. Our second and third heuristics address the challenge of eliminating such false rules.

The second heuristic is based on the observation that false rules tend to flock together. For example in most instances of "1" → "2", one could also replace the "1" by other digits. We therefore ignore rules that come in large groups.

Further eliminating false rules requires calculating the fraction of DUST in the support of each rule. How could this be done without inspecting page content? Our third heuristic uses cues from the URL list to guess which instances are likely to be DUST and which are not. In case the URL list is produced from a previous crawl, we typically have document sketches available for each URL in the list. Document sketches are short summaries that can be used to estimate document similarity [9]. We can thus use the sketches to estimate the likelihood that pairs of URLs are DUST and eliminate rules whose support does not contain sufficiently many DUST pairs.

In case the URL list is produced from web server logs, document sketches are not available. The only cue about the contents of URLs in these logs is the sizes of these contents. We thus use the size field from the log to filter out instances (URL pairs) that have "mismatching" sizes. The difficulty with size-based filtering is that the size of a dynamic page can vary dramatically, e.g., when many users comment on an interesting story or when a web page is personalized. To account for such variability, we compare the ranges of sizes seen in all accesses to each page. When the size ranges of two URLs do not overlap, they are unlikely to be DUST.

Having discovered likely DUST rules, another challenge that needs to be addressed is eliminating redundant ones. For example, the rule "`http://site-name/story?id=`" → "`http://site-name/story_`" will be discovered, along with many consisting of substrings thereof, e.g., "`?id=`" → "`_`". However, before performing validations, it is not obvious which rule should be kept in such situations– the latter could be either valid in all cases, or invalid outside the *context* of the former. Nevertheless, we are able to use support information from the URL list to remove many redundant likely DUST rules. We remove additional redundancies after performing some validations, and thus compile a succinct list of rules.

Canonization. Once the correct DUST rules are discovered, we exploit them for URL canonization. In the general case, the canonization problem is NP-hard. Nevertheless, we have devised an efficient *canonization algorithm* that *typically* succeeds in transforming URLs to a site-specific canonical form.

Experimental results. We experiment with DustBuster on four web sites with very different characteristics. Two of our experiments use web server logs, and two use crawl outputs. We find that DustBuster can discover rules very effectively from moderate sized URL lists, with as little as 20,000 entries. Limited sampling is then used in order to validate or refute each rule.

Our experiments show that up to 90% of the top ten rules discovered by DustBuster *prior to the validation phase* are found to be valid, and in most sites 70% of the top 100 rules are found to be valid. Furthermore, we show that DUST rules discovered by DustBuster may account for 47% of the DUST in a web site and that using DustBuster can reduce a crawl by 26%.

The rest of this thesis is organized as follows. In Chapter 2, we review some related work. We formally define what DUST rules are and state the DUST detection problem in Chapter 3. Chapter 4 presents the basic heuristics our algorithm uses. DustBuster and the canonization algorithm appear in Chapter 5. Chapter 6 presents experimental results. We end with discussion and some concluding remarks in Chapter 7.

Chapter 2

Related Work

2.1 Canonization rules

2.1.1 Global canonization rules

The most obvious and naive way DUST is being dealt with today is through standard canonization. URLs have a very standard structure [4]. The hostname may have many different aliases. Different hostnames may return the exact same site. For example adding a "`www`" to the base hostname often returns the same content. Choosing one hostname to identify each site is a standard way to canonize a URL. Other standard canonization techniques include replacing a "`//`" with a single "`/`" and removing the index.html suffix. However, site specific DUST rules cannot be detected using these simple rules.

2.1.2 Site-specific canonization rules

Another method for discovering DUST rules is to examine the web server configuration file and file system. In the configuration, file alias rules are defined. Each such rule allows a directory in the web server to be accessed using a different name, an alias. By parsing the web server configuration file [1] one could easily learn these rules. Further inspection of the file system may uncover symbolic links. These too have the exact same effect.

There are three main problems with using this technique. The main problem is that symbolic links and aliases are by no means the sole source of DUST rules. Other sources include parameters that do not affect the content, different dynamic files that produce the same content and many others. Our technique, therefore, can discover a wider range of DUST rules, regardless of cause. The second problem is that each configuration file is different according to the type and version of the web server. The third and final problem is that once the files are parsed and processed the rules discovered would have to be transfered to the search engine. Transferring these rules from the web server to the search engine may be possible in the future if new protocols are defined and adopted. The Google Sitemaps architecture [14] defines a protocol for web site managers to supply information about their sites to search engines. This type of protocol can be extended to enable the web server to send canonization rules to any party that wants such information. However, this protocol is not widely adopted yet.

2.2 Detecting similarity between documents

The standard way of detecting similarity is by using document sketches, also known as fingerprints, at the search engine's site after information has been collected. Our approach cannot replace the use of document sketches, since it does not find similar documents across sites or similar documents whose associated URLs are not related to each other by some rule. However, it is desirable to use our approach to complement the use of document sketches in order to reduce the overhead of collecting redundant data, and increase the percentage of unique pages found by crawling a given site. Moreover, since using document sketches does not find rules, it cannot be used for URL canonization, which is important in order to improve the accuracy of page popularity metrics.

Shingles [9] are perhaps the most well known technique for document sketches. A shingle is a short string that represents the document. The document is viewed as a sequence of words. A hash is calculated for every m consecutive word sequence. The minimal hash is the documents shingle. The probability that the shingles are equal, is equal to the documents similarity. We can use several hash functions and calculate several shingles to increase the accuracy of the results.

In [22] and [13] a general framework for the detection of plagiarism and copy detection is introduced: break down the document to chunks, retain a small number of chunks, create digests of each chunk and store these chunks. The digests can then be used using any of the possible resemblance measures to compare the two documents. This paper is interesting because it builds a generalized sketching framework that highlights the choices you have to make. They consider several chunking strategies: sentence, overlapping and hash based breakpoints. They also examined several fingerprint selection based on the chunk size distribution. One method, for example, retains the square root of the number of chunks, closest to the median.

In [16] two techniques are combined to compare similar documents: Content-defined Chunking (CDC) and bloomfilters. The CDC technique is used to determine how to split a document into chunks while the bloomfilters are used to estimate the overlap between two documents. Chunk boundaries in this technique are determined by computing the Rabin fingerprints of consecutive overlapping 48-byte sequences. If the lower bits of the fingerprint match some predetermined value a chunk boundary is set. This makes the chunks insensitive to small changes in the text while allowing you to set the average chunk size. The chunks are then hashed.

A bloom filter consists of m bits and k hash functions. Each hash function returns a number between 1 and m. Each element is hashed using the k hash functions and the appropriate bits are set to 1. Bloomfilters are usually used to test membership in a set efficiently. They are used here to estimate the similarity between the documents.

In [28] an interesting attempt is made to put several well known similarity measures into a single framework. Using the framework one can then specify different similarity measures using a *Q-expression* of an eight-position string. This string specifies a whole space of possible similarity measures. This space, however, could not be explored in any meaningful way.

2.3 Detecting mirror sites

One common source of near-duplicate content is mirroring. A number of previous works have dealt with automatic detection of mirror sites on the web. There are two basic approaches to mirror detection. The first method is based on the content itself. The documents are downloaded and processed. Mirrors are detected by processing the documents to detect the similarity between hosts. We will call these techniques *"bottom-up"*.

The second method uses meta information that may already be available such as a URL list, the IP number associated with the host names and other techniques. We will call these techniques *"top-down"*. These techniques are closer to what we are doing in this thesis.

In contrast to mirror detection, we deal with the complementary problem of detecting DUST within one site. Mirror detection may exploit syntactic analysis of URLs and limited sampling as we do. However, a major challenge that site-specific DUST detection must address is efficiently *discovering* prospective rules out of a daunting number of possibilities (all possible substring substitutions). In contrast, mirror detection focuses on comparing a given pair of sites, and only needs to determine *whether* they are mirrors.

Bottom-up: based on content

In [10] Cho et al. deal with the replicated collections problem. They start by discussing the difficulty of defining a replicated web collection: different versions, partial crawls and different formats. A document collection is modeled as a graph; vertices are documents and edges are hyperlinks between them. Two such collections, C_1 and C_2 are similar if a mapping exists M such that for every $p \in C_1$, p is similar to $M(p)$. Similarity between pages can be defined using any of the methods in the previous section. It is interesting to note that they assume similarity is transitive. The mapping between collections is restricted to being one-to-one. The collections compared are restricted to being of equal size. Furthermore, they require that pages deemed similar have at least a pair of incoming links from pages that are also similar. Finally they define a similar cluster as a set of collections each of which is similar to the other.

The algorithm initially finds trivial clusters by using a document similarity technique. Each collection is of a single document. A trivial cluster graph is formed. A vertex represents a trivial cluster. An edge exists between two trivial clusters if they can be merged. Trivial clusters can be merged if each page in one trivial cluster has a link to a unique page in the other. Finally the connected components of these graphs are merged into similar clusters. This means that the algorithm prefers a large set of mirror sites with few pages in each mirror to few mirror sites with many pages in each. They use the information collected to reduce the next crawl by 48%.

The algorithm is very different from ours. They operate on multiple sites and require extensive analysis of the entire document collection. Our method operates on a single site and requires very little processing and very few document fetches. Their technique requires that the collection be equally sized. Our technique unifies both document similarity and "missing" documents through the validation threshold. Their technique puts emphasis on the link structure while ours puts the main emphasis on the documents existence.

[18] attempts to quantify the problem of URL aliases and its performance impact on URL-indexed cache management. They show that aliased payloads, data requested using different URLs, account for 54% of the traces they examined and 36% of the actual bytes transfered. They also show that the 10% most popular payloads account for over 85% of http transactions. They go over the causes of aliasing and categorize them according to whether or not they reside on the same host and in the same absolute path. They find that 10% of payload transfers to clients are redundant and 20% to a shared proxy.

Top-down: based on structure similarity

The top-down methods utilize common paths between mirror sites and common IP numbers between host names to detect mirror candidate host pairs.

In [5] Bharat and Broder define mirror sites by saying that the two sites share a high percentage of the paths and that these path reference similar documents. They detect candidate host pairs by coordinately sampling the URLs from both sites and testing for similarity. At the next stage they process the candidate host pairs, test if each pair is a mirror and the extent of their overlap. To detect candidates they produce features for each host and try to find a large overlap in features. The features are produced from the URL, from its hostname and path segments. Non-alphanumeric characters are considered word breaks. Each two consecutive word sequence is a feature. A position number is added to the features produced from the path in order to make them more distinguishing. Other optimizations used include converting the URLs to lowercase, eliminating frequent words such as txt and html and tossing infrequent words as well. Features are weighted in inverse proportion to the number of hosts they appear in. Another optimization they use is ignoring features that appear within more than 20 hosts.

Each feature has a weight that is inversely proportional to the number of hosts which it occurs in, giving host features four times the weight of path features. The similarity score is simply the sum of shared feature weights. Calculating this score is straightforward. The score is then normalized. Host pairs which share only one feature or share no path features are eliminated. Finally each host pair is classified according to its content and structure similarity.

As an optimization they do not consider hosts with few URLs as they are probably not worth the added processing time. Further reduction is achieved by sampling the URLs by accepting URLs only if the hash of their path segment yields 0 mod m. The value of m is doubled every 20 URLs sampled. This is done to achieve a sub-linear relationship between site size and sample, large sites have more URLs but will not dominate the list. [20] implements and evaluates the above algorithm.

The work [6] by Bharat, Broder et. al compares several top-down techniques. These are all categorized by using the URLs, the IP addresses and even the hyperlinks but without using the content itself. The first method lists hosts which have identical or highly similar IP addresses. The second method detects host pairs that have URLs that are very similar. This is done by extracting terms from the URLs and comparing them using their term vectors. The third method examined seeks host pairs with common paths, and with documents with similar paths sharing outlinks. The final method examined considers each host as one big document and examines the links between these "super-documents", mirror sites are expected to link to common hosts.

The first observation made is that if two hosts are mapped to the same IP they are likely to be mirrors. However, the more hosts being mapped to the same IP the more likely it is that these are not mirrors but rather something like virtual hosting servers. In the URL string matching scheme they considered several alternatives: terms from the hostname, full path, path prefixes and word bigrams from the path segment. The third method, extends the path based method by examining outlinks. If two common paths share a fraction f of their outlinks, above a certain threshold, they pass the test. A threshold is used because different mirrors have different update frequencies. The outlinks are turned relative to increase the accuracy of this technique.

Finally when examining the hosts as one big document they use term vector matching as well. Each host represents a document. The terms this time are the hosts that the outlinks point at. Each document is represented as a vertex on a graph. A link from document A to document B indicates an outlink exists from host A to host B. Document frequency, the number of documents in which the terms appear is exactly the in-degree of the host-term. The term frequency is not used in the term weight but rather only in term selection, choosing the terms with the highest frequency.

By combining all methods they achieve a precision of 0.57 with a recall of 0.86 when considering the top 100,000 results.

2.4 Analysis of web logs

Various commercial tools as well as papers are available on analyzing web logs. We are not aware of any previous algorithm for automatically detecting DUST rules nor of any algorithm for harvesting information for search engines from web access logs. Some companies (e.g., [27, 25, 3]) have tools for analyzing web logs, but their goals are very different from ours. This type of software usually provides such statistics as: popular keyword terms, entry pages (first page users hit), exit pages (pages users use to move to another site), information about visitor paths and many more.

In [7] Borges et al. model the users browsing session using probabilistic regular grammars. The goal is to build a grammar which generates strings that represent the users navigational trails. They define a user navigation session as a sequence of page requests such that no two consecutive requests are separated by more than X seconds. They define a subclass of probabilistic regular grammar which they call hypertext probabilistic grammar. The user navigation sessions are then modeled as text in the hypertext probabilistic language defined by hypertext probabilistic grammar. The hypertext probabilistic grammar consists of a state for each page plus two more states S for start and F for finish. The probability of production from one state to another is modeled according to the number of times these two pages appeared consecutively in user navigation sessions. The probability of production from the start state is a linear combination of the probability that this page appeared first in a user navigation session and the probability that a random page in the log would be this page.

The results we are looking for are the trails that best characterize the user's behavior. The results are found by traversing the state graph with a special case of directed-graph Depth First Search. The results are then pruned using two methods. The probability of switching from the start state to the first state in the sequence has to be above the *support threshold* θ. The derivation probability of a sequence is calculated by multiplying the probabilities along the links between the states, not including the link from the start state. The derivation probability has to be above the cut-point λ. The navigational paths with support above θ and confidence above λ are the rules we are looking for.

2.5 Grammatical inference

The problem we consider resembles the problems of grammatical inference [11] and learning one variable pattern languages, but fundamentally differs from them since URLs are typically not generated as context-free grammars or one variable languages. Moreover, these problems learn from a collection of positive examples, a sequence of strings all of which are in the language learned. Our data, the URL list, usually does not necessarily follow any such language (apart from basic structure of the URL), and contains many URLs that have no relation to the rules we are trying to discover.

In [23] Reischuk et al. learn a one variable language from positive examples. They define the general problem as follows. Let L be any language, a *text* of L is an infinite sequence of string in L which should contain all of L eventually. The learner is presented with the text incrementally and outputs hypothesis on L which should converge to the language L. A *one-variable pattern* $(\Sigma \cup x)^+$ consists of a non empty string above the alphabet Σ and a variable x. *Pat* denotes the set of all such patterns. Let $\pi \in Pat$ and let $\pi[x/u]$ denote the string resulting in substituting all occurrences of x in π with u. The language generated by pattern π $L(\pi) := \{y \in \Sigma^+ | \exists u \in \Sigma^+, y = \pi[x/u]\}$. The goal of this paper is to learn such languages with sub-quadratic bounds on the expected total learning time.

Given a sequence of example strings $X_1, X_2, ...$ the algorithm should output a series of one-variable patterns $\psi_1, \psi_2, ...$ such that the ψ_i converges to a pattern ψ such that $L(\pi) = L(\psi)$. Unlike the general problem defined above, they do not assume that $\{X_i | i \in N^+\} = L$, meaning they do not assume that every string in the language is eventually represented. Additional assumptions are made about the distribution of the substitution strings used to generate the text. These assumptions are necessary to ensure the boundaries placed on learning time as well as correctness. Infinite length example strings are not allowed of course, otherwise we could not guarantee the algorithm would stop. Another assumption prevents the substitution strings themselves following a pattern and thus creating two languages that cannot be distinguished.

The algorithm itself starts by assuming the first string is the pattern. The prefix and suffix of the pattern are learned from the strings by choosing the largest prefix and suffix of all seen so far. It is shown that once the prefix and suffix strings of the pattern are correct the hypothesis pattern is only refined with every new string that is presented.

2.6 Mining association rules

Although our problem also resembles the problem of mining association rules [2], the two problems differ substantially. Whereas the input of data mining algorithms consists of a sample of complete list of items that belong together, our input includes individual items from different lists. The absence of complete lists renders techniques used therein inapplicable to our problem. Another difference is that rules cannot be deduced solely from the input URL list, since the content of the corresponding documents is not available from this list.

2.7 Unsupervised learning of synonyms

Learning synonyms and name aliases has been researched in the past. The DustBuster algorithm finds words that in some sense "mean" the same thing. DustBuster does not, however, learn synonyms. The structure of sentences is different from the structure of URLs. Thus, DustBuster algorithm solves the problem of detecting likely substring substitution rules in different way and does not use the following methods.

In [26] Turney compares two methods of unsupervised learning of synonyms. Latent Semantic Analysis (LSA) is a method first introduced in [19]. LSA is based on Singular Value Decomposition (SVD). The idea is that SVD naturally breaks down document space into concept vectors.

The basic idea is as follows. We start off with a matrix X. The rows represent words, the columns represent text fragments. This could be sentences, paragraphs or documents. Landauer and Dumais [19] used X directly measuring the cosine of inner product of the word vectors, this lead to a 36.8% score in the TOEFL. Applying SVD results in a matrix that can be viewed as a words by concepts matrix. Using the exact same method with this matrix lead to a 64.4% score.

In PMI the basic idea is again that of co-occurrence. A word is characterized by the words it appears with. So given a problem word and several possible synonyms marked $choice_i$ we score the choices using $score(choice_i) = \frac{p(problem,choice_i)}{p(choice_i)} = p(problem|choice_i)$. PMI-IR attempts to estimate these scores using the Web through search engines.

[21] deals with the problem of automatic identification of similar words given a corpus. Initially dependency triples are extracted from each sentence. A dependency triple consists of two words and the grammatical relationship between them. For example "I have a brown fox" yields the triple (have subject I). The similarity between the words w_1 and w_2 is defined

as the amount of information contained in the words they share a relationship with divided by the sum of the total information contained in all their relationships. The similarity between words is used to build a thesaurus by searching word pairs that are most similar to each other.

Chapter 3

Problem Definition

URLs. We view URLs as strings over an alphabet Σ of tokens. Tokens are either alphanumeric strings or non-alphanumeric characters. In addition, we require every URL to start with the special token `^` and to end with the special token `$` (`^` and `$` are not included in Σ). For example, the URL `http://www.site.com/index.html` is represented by the following sequence of 15 tokens: ˆ,http,:,/,/,www,.,site,.,com,/,index,.,html,\$. We denote by U the space of all possible URLs.

A URL u is *valid*, if its domain name resolves to a valid IP address and its contents can be fetched by accessing the corresponding web server (the http return code is not in the 4xx or 5xx series). If u is valid, we denote by DOC(u) the returned document[1].

DUST. Two valid URLs u_1, u_2 are called DUST, if their corresponding documents, DOC(u_1) and DOC(u_2), are "similar". To this end, any method of measuring the similarity between two documents can be used. For our implementation and experiments, we use the popular *resemblance* measure due to Broder *et al.* [9].

DUST **rules.** In this thesis, we seek general *rules* for detecting when two URLs are DUST. A DUST rule ϕ is a relation over the space of URLs. ϕ may be a many-to-many relation. Every pair of URLs belonging to ϕ is called an *instance* of ϕ. The *support* of a ϕ, denoted SUPPORT(ϕ), is the collection of all its instances.

We discuss two types of DUST rules: substring substitutions and parameter substitutions. *Parameter substitution rules* either replace the value of a certain parameter appearing in the URL with a default value, or omit this parameter from the URL altogether. Thanks to the standard syntax of parameter usage in URLs, detecting parameter substitution rules is fairly straightforward. Most of our work therefore focuses on substring substitution rules, which are similar to the "replace" function in many editors. A *substring substitution rule* $\alpha \to \beta$ is specified by an ordered pair of strings (α, β) over the token alphabet Σ. (In addition, we allow these strings to simultaneously start with the token `^` and/or to simultaneously end with the token `$`.) Instances of substring substitution rules are defined as follows:

[1]DOC(u) may include software error messages returned from the web server even though u is valid

Definition 1. (Instance of a substring substitution rule) A pair u_1, u_2 of URLs is an *instance* of a substring substitution rule $\alpha \rightarrow \beta$ if and only if there exist strings p, s s.t. $u_1 = p\alpha s$ and $u_2 = p\beta s$.

For example, the pair of URLs `http://www.site.com/index.html` and `http://www.site-.com` is an instance of the DUST rule "`/index.html$`" $\rightarrow$ "`$`".

The DUST problem. Our goal is to detect DUST and eliminate redundancies in a collection of URLs belonging to a given web site S. This is solved by a combination of two algorithms, one that discovers DUST rules from a URL list, and another that uses them in order to transform URLs to their canonical form.

A *URL list* is a list of records consisting of: (1) a URL; (2) the http return code; (3) the size of the returned document; and (4) the document's sketch. The last two fields are optional. This type of list can be obtained either from the web server logs or from a previous crawl. Note that the URL list is typically only a (non-random) sample of the URLs that belong to the web site.

For a given web site S, we denote by U_S the set of URLs that belong to S. A DUST rule ϕ is said to be *valid* w.r.t. S, if for each $u_1 \in U_S$ and for each u_2 s.t. (u_1, u_2) is an instance of ϕ, $u_2 \in U_S$ and (u_1, u_2) is DUST.

A DUST *rule detection algorithm* is given a list $\mathcal{L}$ of URLs from a web site S and outputs an ordered list of DUST rules. The algorithm may also fetch pages (which may or may not appear in the URL list). The ranking of rules represents the confidence of the algorithm in the validity of the rules.

Note that we did not define until now what a web site is, nor do we limit the scope of the problem to any such definition. The implicit assumption in this case is only that the URL list is a representative sample of some URL collection and that the rules learned will be valid to the entire URL collection. The URL collection may be limited to a single web site, to a certain set of domain names or to a specific directory on the web site. Regardless of the way the URL collection is defined we learn rules specific to that URL collection. For convenience we will define such a URL collection to be a site.

Canonization. Let $\mathcal{R}$ be an ordered list of DUST rules that have been found to be valid w.r.t. to some web site S. We would like to define what is a *canonization* of the URLs in U_S using the rules in $\mathcal{R}$. To this end, we assume that application of any rule $\phi \in \mathcal{R}$ to any URL $u \in U_S$ results in a URL $\phi(u)$ that also belongs to U_S (this assumption holds true in all the data sets we experimented with)[2]. The rules in $\mathcal{R}$ naturally induce a labeled graph $G_{\mathcal{R}}$ on U_S: there is an edge from u_1 to u_2 labeled by ϕ if and only if $u_1 \xrightarrow{\phi} u_2$. Since the dissimilarity between every two adjacent URLs in $G_{\mathcal{R}}$ is at most ϵ, the dissimilarity between URLs that are connected by a path of length k is at most $k\epsilon$ (dissimilarity respects the triangle inequality). We conclude that if $G_{\mathcal{R}}$ has a bounded diameter (as it does in the data

[2]There may exist several ways of applying a rule to a URL. Our discussion assumes that one standard way is always chosen.

sets we encountered), then every two URLs connected by a path are similar. A canonization which maps every URL u to some URL that is reachable from u would thus make sense, because the original URL and its canonical form are guaranteed to be DUST.

A set of *canonical URLs* is a subset $CU_S \subseteq U_S$ that is reachable from every URL in U_S (equivalently, CU_S is a dominating set in the transitive closure of $G_{\mathcal{R}}$). A canonization is any mapping $C : U_S \to CU_S$ that maps every URL $u \in U_S$ to some canonical URL $C(u)$, which is reachable from u by a directed path. Our goal is to find a small set of canonical URLs and a corresponding canonization, which is efficiently computable.

Finding the minimum size set of canonical URLs is intractable, due to the NP-hardness of the minimum dominating set problem. Fortunately, our empirical study indicates that for typical collections of DUST rules found in web sites, efficient canonization is possible. Thus, although we cannot design an algorithm that always obtains an optimal canonization, we will seek one that maps URLs to a *small* set of canonical URLs, and *always* terminates in polynomial time.

Metrics. There are three measures that we use to evaluate DUST detection and canonization. The first measure is *precision*—the fraction of valid rules among the rules reported by the DUST detection algorithm. The second and most important measure is the *discovered redundancy*—the amount of redundancy eliminated in a crawl. Formally, it is defined as the difference between the number of unique URLs in the crawl, before and after canonization, divided by the former.

The third measure is *coverage*: given a large collection of URLs that includes DUST, what percentage of the duplicate URLs is detected. The number of duplicate URLs in a given URL list is defined as the difference between the number of unique URLs and the number of unique document sketches. Since we do not have access to the entire web site, we measure the achieved coverage within the URL list. We count the number of duplicate URLs in the list before and after canonization, and the difference between them divided by the former is the coverage.

One of the standard measures of information retrieval is *recall*, which may seem an appropriate measure for our problem as well. In our case, recall would measure what percent of all correct DUST rules is discovered. However, it is clearly impossible to construct a complete list of all valid rules to compare against, and therefore, recall is not directly measurable in our case, and is replaced by coverage as explained above.

As for performance, we use four complexity measures: (1) running time; (2) storage space (in secondary storage); (3) memory complexity; and (4) number of web pages fetched.

Clearly, there are tradeoffs between the complexity metrics and the success metrics. For example, an algorithm that fetches a large number of web pages is more likely to achieve precision than one that fetches few or none. We therefore contrast precision with the number of web pages fetched, and also study the precision attained when none are fetched.

Chapter 4

Basic Heuristics

Our algorithm for extracting likely string substitution rules from the URL list uses three heuristics: the *large support heuristic*, the *small buckets heuristic*, and the *similarity likeliness heuristic*. Our empirical results provide evidence that these heuristics are effective on web-sites of varying scopes and characteristics.

4.1 Large support heuristic

Our first heuristic is the following:

Large Support Heuristic
The support of a valid DUST *rule is large.*

For example, if a rule "`index.html$`" → "`$`" is valid, we should expect many instances witnessing to this effect, e.g., `www.site.com/d1/index.html` vs. `www.site.com/d1/` and `www.site.com/d3/index.html` vs. `www.site.com/d3/`. We would thus like to discover rules of large support. Note that valid rules of small support are not very interesting anyway, because the savings gained by applying them are negligible.

Finding the support of a rule on the web site requires knowing all the URLs associated with the site. Since the only data at our disposal is the URL list, which is unlikely to be complete, the best we can do is compute the support of rules *in this URL list*. That is, for each rule ϕ, we can find the number of instances (u_1, u_2) of ϕ, for which both u_1 and u_2 appear in the URL list. We call these instances the *support of ϕ in the URL list* and denote them by $\textsc{support}_{\mathcal{L}}(\phi)$. If the URL list is long enough, we expect this support to be representative of the overall support of the rule on the web site.

Note that since $|\textsc{support}_{\mathcal{L}}(\alpha \to \beta)| = |\textsc{support}_{\mathcal{L}}(\beta \to \alpha)|$, for every α and β, our algorithm cannot know whether both rules are valid or just one of them is valid. It therefore outputs the pair α, β instead. Finding which of the two directions is valid is left to the final phase of DustBuster.

Characterization of support size

Given a URL list $\mathcal{L}$, how do we compute the size of the support of every possible rule? To this end, we introduce a new characterization of the support size. Consider a substring α of a URL $u = p\alpha s$. We call the pair (p, s) the *envelope* of α in u. For example, if $u =$ `http://www.site.com/index.html` and $\alpha =$"`index`", then the envelope of α in u is the pair of strings "ˆ`http://www.site.com/`" and "`.html$`". By Definition 1, a pair of URLs (u_1, u_2) is an instance of a substitution rule $\alpha \to \beta$ if and only if there exists a shared envelope (p, s) so that $u_1 = p\alpha s$ and $u_2 = p\beta s$.

For a string α, denote by $E_{\mathcal{L}}(\alpha)$ the set of envelopes of α in URLs that satisfy the following conditions: (1) these URLs appear in the URL list $\mathcal{L}$; and (2) the URLs have α as a substring. If α occurs in a URL u several times, then u contributes as many envelopes to $E_{\mathcal{L}}(\alpha)$ as the number of occurrences of α in u. The following theorem shows that under certain conditions, $|E_{\mathcal{L}}(\alpha) \cap E_{\mathcal{L}}(\beta)|$ equals $\textsc{support}_{\mathcal{L}}(\alpha \to \beta)$. As we shall see later, this gives rise to an efficient procedure for computing support size, since we can compute the envelope sets of each substring α separately, and then by join and sort operations find the pairs of substrings whose envelope sets have large intersections.

Theorem 2. *Let $\alpha \neq \beta$ be two distinct, non-empty, and non-semiperiodic strings. Then,*

$$|\textsc{support}_{\mathcal{L}}(\alpha \to \beta)| = |E_{\mathcal{L}}(\alpha) \cap E_{\mathcal{L}}(\beta)|.$$

A string α is *semiperiodic*, if it can be written as $\alpha = \gamma^k \gamma'$ for some string γ, where $|\alpha| > |\gamma|$, $k \geq 1$, γ^k is the string obtained by concatenating k copies of the string γ, and γ' is a (possibly empty) prefix of γ [15]. If α is not semiperiodic, it is *non-semiperiodic*. For example, the strings "`a.a.a`" and "`/////`" are semiperiodic, while the strings "`a.a.b`" and "`%////`" are not.

Unfortunately, the theorem does not hold for rules where one of the strings is either semiperiodic or empty. For example, let α be the semiperiodic string "`a.a`" and β = "a". Let $u_1 =$ `http://a.a.a/` and let $u_2 =$ `http://a.a/`. There are *two* ways in which we can substitute α with β in u_1 and obtain u_2. Similarly, let γ be "a" and δ be the empty string. There are three ways in which we can substitute γ with δ in u_1 to obtain u_2. This means that the instance (u_1, u_2) will be associated with three envelopes in $E_{\mathcal{L}}(\gamma) \cap E_{\mathcal{L}}(\delta)$ and with two envelopes in $E_{\mathcal{L}}(\alpha) \cap E_{\mathcal{L}}(\beta)$ and not just one. Thus, when α or β are semiperiodic or empty, $|E_{\mathcal{L}}(\alpha) \cap E_{\mathcal{L}}(\beta)|$ can overestimate the support size. On the other hand, such examples are quite rare, and in practice we expect a minimal gap between $|E_{\mathcal{L}}(\alpha) \cap E_{\mathcal{L}}(\beta)|$ and the support size.

Proof of Theorem 2. To prove the identity, we will show a 1-1 mapping from $\textsc{support}_{\mathcal{L}}(\alpha \to \beta)$ onto $E_{\mathcal{L}}(\alpha) \cap E_{\mathcal{L}}(\beta)$. Let (u_1, u_2) be any instance of the rule $\alpha \to \beta$ that occurs in $\mathcal{L}$. By Definition 1, there exists an envelope (p, s) so that $u_1 = p\alpha s$ and $u_2 = p\beta s$. Note that $(p, s) \in E_{\mathcal{L}}(\alpha) \cap E_{\mathcal{L}}(\beta)$, hence we define our mapping as: $f_{\alpha,\beta}(u_1, u_2) = (p, s)$. The main challenge is to prove that $f_{\alpha,\beta}$ is a well defined function; that is, $f_{\alpha,\beta}$ maps every instance (u_1, u_2) to a single pair (p, s). This is captured by the following lemma:

Lemma 3. *Let $\alpha \neq \beta$ be two distinct, non-empty, and non-semiperiodic strings. Then, there cannot be two distinct pairs $(p_1, s_1) \neq (p_2, s_2)$ s.t. $p_1\alpha s_1 = p_2\alpha s_2$ and $p_1\beta s_1 = p_2\beta s_2$.*

We are left to show that f is 1-1 and onto. Take any two instances (u_1, u_2),(v_1, v_2) of (α, β), and suppose that $f_{\alpha,\beta}(u_1, u_2) = f_{\alpha,\beta}(v_1, v_2) = (p, s)$. This means that $u_1 = p\alpha s = v_1$ and $u_2 = p\beta s = v_2$. Hence, necessarily $(u_1, u_2) = (v_1, v_2)$, implying f is 1-1. Take now any envelope $(p, s) \in E_{\mathcal{L}}(\alpha) \cap E_{\mathcal{L}}(\beta)$. By definition, there exist URLs $u_1, u_2 \in \mathcal{L}$, so that $u_1 = p\alpha s$ and $u_2 = p\beta s$. By Definition 1, (u_1, u_2) is an instance of the rule $\alpha \to \beta$, and thus $f_{\alpha,\beta}(u_1, u_2) = (p, s)$. □

In order to prove Lemma 3, we show the following basic property of semiperiodic strings:

Lemma 4. *Let $\alpha \neq \beta$ be two distinct and non-empty strings. If β is both a suffix and a prefix of α, then α must be semiperiodic.*

Proof. Since β is both a prefix and a suffix of α and since $\alpha \neq \beta$, there exist two non-empty strings β_0 and β_2 s.t.

$$\alpha = \beta_0\beta = \beta\beta_2.$$

Let $k = \lfloor \frac{|\alpha|}{|\beta_0|} \rfloor$. Note that $k \geq 1$, as $|\alpha| \geq |\beta_0|$. We will show by induction on k that $\alpha = \beta_0^k\beta_0'$, where β_0' is a possibly empty prefix of β_0.

The induction base is $k = 1$. In this case, $|\beta_0| \geq \frac{|\alpha|}{2}$, and as $\alpha = \beta_0\beta$, $|\beta| \leq |\beta_0|$. Since $\beta_0\beta = \beta\beta_2$, it follows that β is a prefix of β_0. Thus, define $\beta_0' = \beta$ and we have:

$$\alpha = \beta_0\beta = \beta_0\beta_0'.$$

Assume now that the statement holds for $\lfloor \frac{|\alpha|}{|\beta_0|} \rfloor = k - 1 \geq 1$ and let us show correctness for $\lfloor \frac{|\alpha|}{|\beta_0|} \rfloor = k$. As $\alpha = \beta_0\beta$, then $\lfloor \frac{|\beta|}{|\beta_0|} \rfloor = k - 1 \geq 1$. Hence, $|\beta| \geq |\beta_0|$ and thus β_0 must be a prefix of β (recall that $\beta_0\beta = \beta\beta_2$). Let us write β as:

$$\beta = \beta_0\beta_1.$$

We thus have the following two representations of α:

$$\alpha = \beta_0\beta = \beta_0\beta_0\beta_1 \text{ and } \alpha = \beta\beta_2 = \beta_0\beta_1\beta_2.$$

We conclude that:

$$\beta = \beta_1\beta_2.$$

We thus found a string (β_1), which is both a prefix and a suffix of β. We therefore conclude from the induction hypothesis that

$$\beta = \beta_0^{k-1}\beta_0',$$

for some prefix β_0' of β_0. Hence,

$$\alpha = \beta_0\beta = \beta_0^k\beta_0'.$$

□

We can now prove Lemma 3:

Proof of Lemma 3. Suppose, by contradiction, that there exist two different pairs (p_1, s_1) and (p_2, s_2) so that:

$$u_1 = p_1\alpha s_1 = p_2\alpha s_2 \tag{4.1}$$

$$u_2 = p_1\beta s_1 = p_2\beta s_2. \tag{4.2}$$

These equations together with the fact that $(p_1, s_1) \neq (p_2, s_2)$ imply that both $p_1 \neq p_2$ and $s_1 \neq s_2$. Thus, by Equations (4.1) and (4.2), one of p_1 and p_2 is a proper prefix of the other. Suppose, for example, that p_1 is a proper prefix of p_2 (the other case is identical). It follows that s_2 is a proper suffix of s_1.

Let us assume, without loss of generality, that $|\alpha| \geq |\beta|$. There are two cases to consider:

Case (i): Both $p_1\alpha$ and $p_1\beta$ are prefixes of p_2. This implies that also $s_2\alpha, s_2\beta$ are suffixes of s_1.

Case (ii): At least one of $p_1\alpha$ and $p_1\beta$ is not a prefix of p_2.

Case (i): In this case, p_2 can be expressed as:

$$p_2 = p_1\alpha p_2' = p_1\beta p_2'',$$

for some strings p_2' and p_2''. Thus, since $|\alpha| \geq |\beta|$, β is a prefix of α. Similarly:

$$s_1 = s_1'\alpha s_2 = s_1''\beta s_2.$$

Thus, β is also a suffix of α. According to Lemma 4, α is therefore semiperiodic. A contradiction.

Case (ii): If $p_1\alpha$ is not a prefix of p_2, α can be written as $\alpha = \alpha_0\alpha_1 = \alpha_1\alpha_2$, for some non-empty strings $\alpha_0, \alpha_1, \alpha_2$, as shown in Figure 4.1. By Lemma 4, α is semiperiodic. A contradiction. The case that $p_1\beta$ is not a prefix of p_2 is handled in a similar manner. $\square$

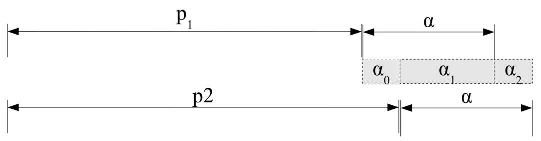


Figure 4.1: Breakdown of α in Lemma 3.

4.2 Small buckets heuristic

While most valid DUST rules have large support, the converse is not necessarily true: there can be rules with large support that are not valid. One class of such rules is substitutions among numbered items, e.g., (`lect1.ps`,`lect2.ps`), (`lect1.ps`,`lect3.ps`), and so on. Another class is substitutions among values of a parameter, e.g., (`day=1`,`day=2`), (day=1,day=3), etc. Some parameter substitutions are DUST and others are not. Moreover, the ranges of many parameter values are too large to efficiently capture as a collection of substring substitutions (e.g., dates). We therefore do not want to capture parameter substitution rules in this phase of the algorithm; we deal with them separately in Chapter 5.3.

We would like to somehow filter out the rules with "misleading" support. The support for a rule $\alpha \rightarrow \beta$ can be thought of as a collection of recommendations, where each envelope $(p, s) \in E_{\mathcal{L}}(\alpha) \cap E_{\mathcal{L}}(\beta)$ represents a single recommendation. Suppose now that one envelope (p, s) is willing to give a recommendation to anybody (any rule) that asks for it, for example "ˆ`http://`" $\rightarrow$ "ˆ". Naturally its recommendations would lose their value. This type of support only leads to many false rules being considered. This is the intuitive motivation for the following heuristic to separate the valid DUST rules from false ones.

If an envelope (p, s) belongs to many envelope sets $E_{\mathcal{L}}(\alpha_1)$, $E_{\mathcal{L}}(\alpha_2)$,..., $E_{\mathcal{L}}(\alpha_k)$, then it contributes to the intersections $E_{\mathcal{L}}(\alpha_i) \cap E_{\mathcal{L}}(\alpha_j)$, for all $1 \leq i \neq j \leq k$. The substrings $\alpha_1, \alpha_2, \ldots, \alpha_k$ constitute what we call a *bucket*. That is, for a given envelope (p, s), *bucket*(p, s) is the set of all substrings α s.t. $p\alpha s \in \mathcal{L}$. An envelope pertaining to a large bucket supports many rules.

Small Buckets Heuristic
Most of the support of valid DUST *substring substitution rules is likely to belong to small buckets.*

4.3 Similarity likeliness heuristic

The above two heuristics use the URL strings alone to detect DUST. In order to raise the precision of the algorithm, we use a third heuristic that better captures the "similarity dimension", by providing hints as to which instances are likely to be similar.

Similarity Likeliness Heuristic
The likely similar support of a valid DUST *rule is large.*

We show below that using cues from the URL list we can determine which URL pairs in the support of a rule are likely to have similar content and which are not. This enables us to filter out instances that are not DUST and thereby calculate the "likely similar support" of a rule. The likely similar support, rather than the complete support, is used to determine whether a rule is valid or not. This heuristic turned out to be very important in removing invalid rules. For example, in a forum web site we examined, the URL list included two sets of URLs `http://politics.domain/story_num` and `http://movies.domain/story_num` with different numbers replacing "`num`". The support of the false rule "`http://politics.domain`" $\rightarrow$ "`http://movies.domain`" was large, yet since the corresponding stories were very different, the likely similar support of the rule was found to be small.

How do we use the URL list to estimate similarity between documents? The simplest case is that the URL list includes a document sketch for each URL. Such sketches are typically available when the URL list is the output of a previous crawl of the web site. Documents sketches, such as the shingles of Broder *et al.* [9], can be used to estimate the similarities among documents.

When the URL list is taken from web server logs, documents sketches are not available. In this case we use document sizes as primitive sketches (document sizes are usually given by web server software). We determine two documents to be similar if their sizes "match". Size matching, however, turns out to be quite intricate, because the same document may have very different sizes when inspected at different points of time or by different users. This is especially true when dealing with forums or blogging web sites. The URL list may provide sizes of documents at different times, and therefore if two URLs have different "size" values in the URL list, we cannot immediately infer that these URLs are not DUST. Instead, we resort to a somewhat more refined strategy.

For each unique URL u that appears in the URL list, we track all its occurrences in the URL list, and keep the minimum and the maximum size values encountered. We denote the interval between these two numbers by I_u. We now say that two URLs, u_1 and u_2, have *mismatching sizes*, if the intervals I_{u_1} and I_{u_2} are disjoint. That is, either the maximum size of u_1 is smaller than the minimum size of u_2 or vice versa. Our experiments show that this size matching heuristic is very effective in improving the precision of our algorithm, often increasing precision by a factor of two.

While size matching is an effective heuristic, it also has its limitations. Valid DUST rules may exist that will never be found when using the size heuristic. An example of such a DUST rule is "`ps`" $\rightarrow$ "`pdf`" which was found in the academic site only when running DustBuster without size matching. The reason is that such documents are likely to have consistently different sizes although their contents are similar.

Chapter 5

DustBuster

In this chapter we describe DustBuster—our algorithm for discovering site-specific DUST rules. DustBuster has four phases. The first phase, presented in Section 5.1, uses the URL list alone to generate a short list of *likely* DUST rules. The second phase, detailed in Section 5.2, removes redundancies from this list. The next phase generates likely parameter substitution rules and is discussed in Section 5.3. The last phase, presented in Section 5.4, validates or refutes each of the rules in the list, by fetching a small sample of pages. The last section discusses the application of the algorithm for DUST canonization.

5.1 Detecting likely DUST rules

Our strategy for discovering likely DUST rules, based on the heuristics described in the previous chapter, is the following: we compute the size of the support of each rule that has at least one instance in the URL list, and output the rules with largest support. Based on Theorem 2, we compute the size of the support of a rule $\alpha \to \beta$ as the size of the set $E_{\mathcal{L}}(\alpha) \cap E_{\mathcal{L}}(\beta)$. That is roughly what our algorithm does, but with three reservations:

(1) Based on the small buckets heuristic, we avoid considering certain rules by ignoring large buckets in the computation of envelope set intersections. Buckets bigger than some threshold T are called *overflowing*, and all envelopes pertaining to them are denoted collectively by O and are not included in the envelope sets.

(2) Based on the similarity likeliness heuristic, we filter support by estimating the likelihood of two documents being similar: We eliminate additional rules by filtering out instances whose associated documents are unlikely to be similar in content. That is, for a given instance $u_1 = p\alpha s$ and $u_2 = p\beta s$, the envelope (p, s) is disqualified if u_1 and u_2 are found unlikely to be similar using the tests introduced in Chapter 4.3. These techniques are provided as a boolean function LikelySimilar which returns false only if the documents of the two input URLs are unlikely to be similar. The set of all disqualified envelopes is then denoted $D_{\alpha,\beta}$.

(3) In practice, substitutions of long substrings are rare. Hence, our algorithm considers substrings of length at most S, for some given parameter S.

To conclude, our algorithm computes for every two substrings α, β that appear in the

URL list and whose length is at most S, the size of the set $(E_{\mathcal{L}}(\alpha) \cap E_{\mathcal{L}}(\beta)) \setminus (O \cup D_{\alpha,\beta})$.

```
1:**Function** DetectLikelyRules(URLList $\mathcal{L}$)
2:  create table ST (substring, prefix, suffix, size_range/doc_sketch)
3:  create table IT (substring1, substring2)
4:  create table RT (substring1, substring2, support_size)
5:  for each record $r \in \mathcal{L}$ do
6:     for $\ell = 0$ to S do
7:        for each substring $\alpha$ of r.url of length $\ell$ do
8:           p := prefix of r.url preceding $\alpha$
9:           s := suffix of r.url succeeding $\alpha$
10:          add ($\alpha$, p, s, r.size_range/r.doc_sketch) to ST
11: group tuples in ST into buckets by (prefix,suffix)
12: for each bucket $B$ do
13:    if (|B| = 1 OR |B| > T) continue
14:    for each pair of distinct tuples $t_1, t_2 \in$ B do
15:       if (LikelySimilar($t_1$, $t_2$))
16:          add ($t_1$.substring, $t_2$.substring) to IT
17: group tuples in IT into rule_supports by (substring1,substring2)
18: for each rule_support R do
19:    t := first tuple in R
20:    add tuple (t.substring1, t.substring2, |R|) to RT
21: sort RT by support_size
22: return all rules in RT whose support size is $\geq MS$
```

Figure 5.1: Algorithm for discovering likely DUST rules.

Our algorithm for discovering likely DUST rules is described in Figure 5.1. The algorithm gets as input the URL list $\mathcal{L}$. It uses three tunable parameters– the maximum substring length, S, the overflowing bucket size, T, and the minimum support size, MS. We assume the URL list has been pre-processed so that: (1) only unique URLs have been kept; (2) all the URLs have been tokenized and include the preceding ˆ and succeeding \$; (3) all records corresponding to errors (http return codes in the 4xx and 5xx series) have been filtered out; (4) for each URL, the corresponding document sketch or size range have been recorded.

The algorithm uses three tables: a substring table ST, an instance table IT, and a rule table RT. Their attributes are listed in Figure 5.1. In principle, the tables can be stored in any database structure; our implementation uses text files.

In lines 5–10, the algorithm scans the URL list, and records all substrings of lengths 0 to S (where S is the maximum substring length) of the URLs in the list. A substring of length 0 represents the empty string, which occurs in some rules. For each such substring α, a tuple is added to the substring table ST. This tuple consists of the substring α, as well as its envelope (p, s), and either the URL's document sketch or its size range. The substrings are then grouped into buckets by their envelopes (line 11). Our implementation does this by sorting the file holding the ST table by the second and third attributes. Note that two substrings α, β appear in the bucket of (p, s) if and only if $(p, s) \in E_{\mathcal{L}}(\alpha) \cap E_{\mathcal{L}}(\beta)$.

In lines 12–16, the algorithm enumerates the envelopes found. An envelope (p, s) contributes 1 to the intersection of the envelope sets $E_{\mathcal{L}}(\alpha) \cap E_{\mathcal{L}}(\beta)$, for every α, β that appear in its bucket. Thus, if the bucket has only a single entry, we know (p, s) does not contribute any instance to any rule, and thus can be tossed away. If the bucket is overflowing (its size exceeds T), then (p, s) is also ignored (line 13).

In lines 14–16, the algorithm enumerates all the pairs (α, β) of substrings that belong to the bucket of (p, s). If the documents associated with the URLs $p\alpha s$ and $p\beta s$ seem likely to be similar either through size matching or document sketch matching (line 15), (α, β) is added to the instance table IT (line 16).

The number of times a pair (α, β) has been added to the instance table is exactly the size of the set $(E_{\mathcal{L}}(\alpha) \cap E_{\mathcal{L}}(\beta)) \setminus (O \cup D_{\alpha,\beta})$, which is our estimated support for the rules $\alpha \to \beta$ and $\beta \to \alpha$. Hence, all that is left to do is compute these counts and sort the pairs by their count (lines 17–22). The algorithm's output is an ordered list of pairs. Each pair representing two likely DUST rules (one in each direction). Only rules whose support is large enough (bigger than MS) are kept in the list.

Complexity analysis. Let n be the number of records in the URL list and let m be the average length (in tokens) of URLs in the URL list. We assume tokens are of constant length. The size of the URL list is then $O(mn)$ bits. We use $\tilde{O}()$ to suppress logarithmic factors, which are typically negligible.

The computation has two major bottlenecks. The first is filling in the instance table IT (lines 12–16). Since the algorithm enumerates all the buckets in ST (there are at most $O(mnS)$ of these), and then enumerates each pair of substrings in each bucket, it could possibly face a quadratic blowup. Yet, since overflowing buckets are ignored, then this step takes only $O(mnST^2)$ time. The second bottleneck is sorting the URL list and the intermediate tables. Since all the intermediate tables are of size at most $O(mnST^2)$, the sorting can be carried in $\tilde{O}(mnST^2)$ time and $O(mnST^2)$ storage space. By using an efficient external storage sort utility, we can keep the memory complexity $\tilde{O}(1)$ rather than linear. The algorithm does not fetch any pages.

5.2 Eliminating redundant rules

By design, the output of the above algorithm includes many overlapping pairs. For example, when running on a forum site, our algorithm finds the pair (“`.co.il/story?id=`”, “`.co.il/story_`”), as well as numerous pairs of substrings of these, such as (“`story?id=`”, “`story_`”). Note that every instance of the former pair is also an instance of the latter. We thus say that the former *refines* the latter. It is desirable to eliminate redundancies prior to attempting to validate the rules, in order to reduce the cost of validation. However, when one likely DUST rule refines another, it is not obvious which should be kept. In some cases, the broader rule is always true, and all the rules that refine it are redundant. In other cases, the broader rule is false as a general rule, and is only valid in specific *contexts* identified by the refining ones.

Nevertheless, in some cases, we can use information from the URL list in order to deduce that a pair is redundant. When two pairs have exactly the same support in the URL list, this gives a strong indication that the latter, seemingly more general rule, is valid only when showing up at the context specified by the former rule. We can thus eliminate the latter rule from the list.

We next discuss in more detail the notion of *refinement* and show how to use it to eliminate redundant rules.

Definition 5. (Refinement) A rule ϕ *refines* a rule ψ if $\textsc{support}(\phi) \subseteq \textsc{support}(\psi)$.

That is, ϕ refines ψ, if every instance (u_1, u_2) of ϕ is also an instance of ψ. Testing refinement for substitution rules turns out to be easy:

Lemma 6. *A substitution rule $\alpha' \to \beta'$ refines a substitution rule $\alpha \to \beta$ if and only if there exists an envelope (γ, δ) s.t. $\alpha' = \gamma\alpha\delta$ and $\beta' = \gamma\beta\delta$.*

Proof. We prove derivation in both directions. Assume, initially, that there exists an envelope (γ, δ) s.t. $\alpha' = \gamma\alpha\delta$ and $\beta' = \gamma\beta\delta$. We need to show that in this case $\alpha' \to \beta'$ refines $\alpha \to \beta$. Take, then, any instance (u_1, u_2) of $\alpha' \to \beta'$. By Definition 1, there exists an envelope (p', s') s.t. $u_1 = p'\alpha's'$ and $u_2 = p'\beta's'$. Hence, if we define $p = p'\gamma$ and $s = \delta s'$, then we have that $u_1 = p\alpha s$ and $u_2 = p\beta s$. Using again Definition 1, we conclude that (u_1, u_2) is also an instance of $\alpha \to \beta$. This proves the first direction.

For the second direction, assume that $\alpha' \to \beta'$ refines $\alpha \to \beta$. Assume that none of $\alpha, \beta, \alpha', \beta'$ starts with ˆ or ends with \$. (The extension to $\alpha, \beta, \alpha', \beta'$ that can start with ˆ or end with \$ is easy, but requires some technicalities, that would harm the clarity of this proof.) Define $u_1 = \hat{}\alpha'\$$ and $u_2 = \hat{}\beta'\$$. By Definition 1, (u_1, u_2) is an instance of $\alpha' \to \beta'$. Due to refinement, it is also an instance of $\alpha \to \beta$. Hence, there exist p, s s.t. $u_1 = p\alpha s$ and $u_2 = p\beta s$. Hence, $p\alpha s = \hat{}\alpha'\$$ and $p\beta s = \hat{}\beta'\$$. Since α, β do not start with ˆ and do not end with \$, then p must start with ˆ and s must end with \$. Define then γ to be the string p excluding the leading ˆ, and define δ to be the string s excluding the trailing \$. We thus have: $\alpha' = \gamma\alpha\delta$ and $\beta' = \gamma\beta\delta$, as needed. □

The characterization given by the above lemma immediately yields an efficient algorithm for deciding whether a substitution rule $\alpha' \to \beta'$ refines a substitution rule $\alpha \to \beta$: we simply check that α is a substring of α', replace α by β, and check whether the outcome is β'. If α has multiple occurrences in α', we check all of them. Note that our algorithm's input is a list of pairs rather than rules, where each pair represents two rules. When considering two pairs (α, β) and (α', β'), we check refinement in both directions.

Now, suppose a rule $\alpha' \to \beta'$ was found to refine a rule $\alpha \to \beta$. Then, $\textsc{support}(\alpha' \to \beta') \subseteq \textsc{support}(\alpha \to \beta)$, implying that also $\textsc{support}_{\mathcal{L}}(\alpha' \to \beta') \subseteq \textsc{support}_{\mathcal{L}}(\alpha \to \beta)$. Hence, if $|\textsc{support}_{\mathcal{L}}(\alpha' \to \beta')| = |\textsc{support}_{\mathcal{L}}(\alpha \to \beta)|$, it must be the case that $\textsc{support}_{\mathcal{L}}(\alpha' \to \beta') = \textsc{support}_{\mathcal{L}}(\alpha \to \beta)$. If the URL list is sufficiently representative of the web site, this gives an indication that every instance of the refined rule $\alpha \to \beta$ that occurs on the web site is also an instance of the refinement $\alpha' \to \beta'$. This makes one of the rules redundant. We choose to keep the refinement $\alpha' \to \beta'$, because it gives the full context of the substitution.

One small obstacle to using the above approach is the following. In the first phase of our algorithm, we do not compute the exact size of the support $|\textsc{support}_{\mathcal{L}}(\alpha \to \beta)|$, but rather calculate the quantity $|(E_{\mathcal{L}}(\alpha) \cap E_{\mathcal{L}}(\beta)) \setminus (O \cup D_{\alpha,\beta})|$. It is possible that $\alpha' \to \beta'$ refines $\alpha \to \beta$ and $\textsc{support}_{\mathcal{L}}(\alpha' \to \beta') = \textsc{support}_{\mathcal{L}}(\alpha \to \beta)$, yet $|(E_{\mathcal{L}}(\alpha') \cap E_{\mathcal{L}}(\beta')) \setminus (O \cup D_{\alpha',\beta'})| < |(E_{\mathcal{L}}(\alpha) \cap E_{\mathcal{L}}(\beta)) \setminus (O \cup D_{\alpha,\beta})|$.

How could this happen? Consider some envelope $(p, s) \in E_{\mathcal{L}}(\alpha) \cap E_{\mathcal{L}}(\beta)$, and let $(u_1, u_2) = (p\alpha s, p\beta s)$ be the corresponding instance of $\alpha \to \beta$. Since $\textsc{support}_{\mathcal{L}}(\alpha' \to \beta') = \textsc{support}_{\mathcal{L}}(\alpha \to \beta)$, then (u_1, u_2) is also an instance of $\alpha' \to \beta'$. Therefore, there exists some envelope $(p', s') \in E_{\mathcal{L}}(\alpha') \cap E_{\mathcal{L}}(\beta')$ s.t. $u_1 = p'\alpha' s'$ and $u_2 = p'\beta' s'$.

α is a substring of α' and β is a substring of β', thus p' must be a prefix of p and s' must be a suffix of s. This implies that any URL that contributes a substring γ to the bucket of (p, s) will also contribute a substring γ' to the bucket of (p', s'), unless γ' exceeds the maximum substring length S. In principle, then, we should expect the bucket of (p', s') to be larger than the bucket of (p, s). If the maximum bucket size T happens to be exactly between the sizes of the two buckets, then (p', s') overflows while (p, s) does not. In this case, the first phase of DustBuster will account for (u_1, u_2) in the computation of the support of $\alpha \to \beta$ but not in the computation of the support of $\alpha' \to \beta'$, incurring a difference between the two.

In practice, if the supports are identical, the difference between the calculated support sizes should be small. We thus eliminate the refined rule, even if its calculated support size is slightly above the calculated support size of the refining rule. However, to increase the effectiveness of this phase we run the first phase of the algorithm twice, once with a lower overflow threshold T_{low} and once with a higher overflow threshold T_{high}. While the lower threshold support is more effective in filtering out false rules the support calculated using the higher threshold is more effective in eliminating redundant rules.

The algorithm for eliminating refined rules from the list is described in Figure 5.2. The algorithm gets as input a list of pairs, representing likely rules, sorted by their calculated support size. It uses three tunable parameters: (1) the *maximum relative deficiency, MRD*, (2) the *maximum absolute deficiency, MAD*; and (3) the *maximum window size, MW*. MRD and MAD determine the maximum difference allowed between the calculated sizes of the supports of the refining rule and the refined rule, when we eliminate the refined rule. MW determines how far down the list we are willing to look for refinements.

The algorithm scans the list from top to bottom. For each rule $\mathcal{R}[i]$, which has not been eliminated yet, the algorithm scans a "window" of rules below $\mathcal{R}[i]$. Suppose s is the calculated size of the support of $\mathcal{R}[i]$. The window size is chosen so that (1) it never exceeds MW; and (2) the difference between s and the calculated support size of the lowest rule in the window is at most the maximum between $MRD \cdot s$ and MAD. Now, if $\mathcal{R}[i]$ refines a rule $\mathcal{R}[j]$ in the window, the refined rule $\mathcal{R}[j]$ is eliminated, while if some rule $\mathcal{R}[j]$ in the window refines $\mathcal{R}[i]$, $\mathcal{R}[i]$ is eliminated.

```
1:Function EliminateRedundancies(pairs_list $\mathcal{R}$)
2:  for i = 1 to $|\mathcal{R}|$ do
3:     if (already eliminated $\mathcal{R}[i]$) continue
4:     to_eliminate_current := false
5:     for j = 1 to $\min(\text{MW}, |\mathcal{R}| - i)$ do
6:        if ($\mathcal{R}[i].\text{size} - \mathcal{R}[i+j].\text{size} >$
                      $\max(\text{MRD} \cdot \mathcal{R}[i].\text{size}, \text{MAD})$)  break
7:        if ($\mathcal{R}[i]$ refines $\mathcal{R}[i+j]$)
8:           eliminate $\mathcal{R}[i+j]$
9:        else if ($\mathcal{R}[i+j]$ refines $\mathcal{R}[i]$) then
10:           to_eliminate_current := true
11:           break
12:     if (to_eliminate_current)
13:        eliminate $\mathcal{R}[i]$
14:  return $\mathcal{R}$
```

Figure 5.2: Eliminating redundant rules.

It is easy to verify that the running time of the algorithm is at most $|\mathcal{R}| \cdot MW$. In our experiments, this algorithm reduces the set of rules by over 90%.

5.3 Parameter substitutions

Inline parameters in URLs typically comply with a standard format. In many sites, an inline parameter name is preceded by ? or & and followed by = and the parameter's value, which is followed by either the end of the URL or &. We can therefore employ a simple regular expression search on URLs in the URL list in order to detect popular parameters, along with multiple examples of values for each parameter. Having detected the parameters, we check for each one whether replacing its value with an arbitrary one is a valid DUST rule. To this end, we exploit the ST table computed by DustBuster (see Figure 5.1), after it has been sorted and divided into buckets. We seek buckets whose prefix attribute ends with the desired parameter name, and then compare the document sketches or size ranges of the relevant URLs pertaining to such buckets.

For each parameter, p, we choose some value of the parameter , v_p, and add two rules to the list of likely rules: the first, replaces the value of the parameter p with v_p, the second rule, omits the parameter altogether. Due to the simplicity of this algorithm, its detailed presentation is omitted.

5.4 Validating DUST rules

So far, the algorithm has generated likely rules from the URL list alone, without fetching even a single page from the web site. Fetching a small number of pages for validating or refuting these rules is necessary for two reasons. First, it can significantly leverage the final precision of the algorithm. Second, the first two phases of DustBuster, which discover likely substring substitution rules, cannot distinguish between the two directions of a rule. The discovery of the pair (α, β) can represent both $\alpha \rightarrow \beta$ and $\beta \rightarrow \alpha$. This does not mean that in reality both rules are valid or false simultaneously. It is often the case that only one of the directions is valid; for example, in many sites removing the substring index.html is always valid, whereas adding one is not. Only by attempting to fetch actual page contents we can tell which direction is valid, if any.

The validation phase of DustBuster therefore fetches a small sample of web pages from the web site in order to check the validity of the rules generated in the previous phases. The validation of a single rule is presented in Figure 5.3. The algorithm is given as input a likely rule R and a list of URLs from the web site and decides whether the rule is valid. It uses two parameters: the *validation count,* N (how many samples to use in order to validate each rule), and the *refutation threshold,* ϵ (the minimum fraction of counterexamples to a rule required to declare the rule false).

```
 1:**Function** ValidateRule(R, $\mathcal{L}$)
 2:  positive := 0
 3:  negative := 0
 4:  **while** (positive < $(1-\epsilon)N$ AND negative < $\epsilon N$) **do**
 5:    u := a random URL from $\mathcal{L}$ on which applying R results in a different URL
 6:    v := outcome of application of R to u
 7:    fetch u and v
 8:    **if** (fetch u failed) continue
 9:    **if** (fetch v failed OR DocSketch(u) $\neq$ DocSketch(v))
10       negative := negative + 1
11:    **else**
12:       positive := positive + 1
13:    **if** (negative $\geq \epsilon N$ )
14:       return FALSE
15:  return TRUE
```

Figure 5.3: Validating a single likely rule.

Remark. The application of R to u (line 6) may result in several different URLs. For example, there are several ways of replacing the string “people” with the string “users” in the URL `http://people.domain.com/people`, resulting in the URLs `http://users.domain.com/people`, `http://people.domain.com/users`, and `http://users.domain.com/users`. Our policy is to select one standard way of applying a rule. For example, in the case of substring substitutions, we simply replace the first occurrence of the substring.

In order to determine whether a rule is valid, the algorithm repeatedly chooses random URLs from the given test URL list until hitting a URL on which applying the rule results in a different URL (line 5). The algorithm then applies the rule to the random URL u, resulting in a new URL v. The algorithm then fetches u and v. Using document sketches, such as the shingling technique of Broder *et al.* [9], the algorithm tests whether u and v are similar. If they are, the algorithm accounts for u as a positive example attesting to the validity of the rule. If v cannot be fetched, or they are not similar, then it is accounted as a negative example (lines 9–12). The testing is stopped when either the number of negative examples surpasses the refutation threshold or when the number of positive examples is large enough to guarantee the number of negative examples will not surpass the threshold.

One could ask why we declare a rule valid even if we find (a small number of) counterexamples to it. There are several reasons: (1) the document sketch comparison test sometimes makes mistakes, since it has an inherent false negative probability; (2) dynamic pages sometimes change significantly between successive probes (even if the probes are made at short intervals); and (3) the fetching of a URL may sometimes fail at some point in the middle, after part of the page has been fetched. By choosing a refutation threshold smaller than one, we can account for such situations.

Each parameter substitution rule is validated using the code in Figure 5.3. The validation of substring substitutions is more complex, as it needs to address directions and refinements. The algorithm for doing so is given in Figure 5.4. Its input consists of a list of pairs representing likely substring transformations, $(\mathcal{R}[i].\alpha, \mathcal{R}[i].\beta)$, and a test URL list $\mathcal{L}$.

For a pair of substrings (α, β), we use the notation $\alpha < \beta$ to denote that either $|\alpha| < |\beta|$ or $|\alpha| = |\beta|$ and α precedes β in the lexicographical order. In this case, we say that the rule *shrinks* the URL. We give precedence to shrinking substitutions. Therefore, given a pair (α, β), if $\alpha > \beta$, we first try to validate the rule $\alpha \to \beta$. If this rule is valid, we ignore the rule in the other direction since, even if this rule turns out to be valid as well, using this rule during canonization is only likely to create cycles, i.e., rules that can be applied an infinite number of times because they cancel out each others' changes. If the shrinking rule is invalid, though, we do attempt to validate the opposite direction, so as not to lose a valid rule. Whenever one of the directions of (α, β) is found to be valid, we remove from the list all pairs refining (α, β)– once a broader rule is deemed valid, there is no longer a need for refinements thereof. By eliminating these rules prior to validating them, we reduce the number of pages we fetch. We assume that each pair in $\mathcal{R}$ is ordered so that $\mathcal{R}[i].\alpha > \mathcal{R}[i].\beta$.

The algorithm eliminates a rule from the list either if it is found to be false or if it or its counterpart in the other direction is a refinement of a valid rule. Note that when a rule $\alpha' \to \beta'$ refines a rule $\alpha \to \beta$, then the validity of the latter immediately implies the validity of the former. The refining rule is thus made redundant and can be eliminated from the list (lines 3–10).

The running time of the algorithm is at most $O(|\mathcal{R}|^2 + N|\mathcal{R}|)$. Since the list is assumed to be rather short, this running time is manageable. The number of pages fetched is $O(N|\mathcal{R}|)$ in the worst-case, but much smaller in practice, since we eliminate many redundant rules after validating rules they refine. In typical scenarios, we had to attempt to validate roughly a third of the original list of rules.

```
1:Function Validate(rules_list $\mathcal{R}$, test_URLList $\mathcal{L}$)
2   create an empty list of rules LR
3:  for i = 1 to $|\mathcal{R}|$ do
4:    for j = 1 to i - 1 do
5:      if ($\mathcal{R}[j]$ was not eliminated AND $\mathcal{R}[i]$ refines $\mathcal{R}[j]$)
6:        eliminate $\mathcal{R}[i]$ from the list
7:        break
8:    if ($\mathcal{R}[i]$ was eliminated)
9:      continue
10:   if (ValidateRule($\mathcal{R}[i].\alpha \rightarrow \mathcal{R}[i].\beta$, $\mathcal{L}$))
11:     add $\mathcal{R}[i].\alpha \rightarrow \mathcal{R}[i].\beta$ to LR
12:   else if (ValidateRule($\mathcal{R}[i].\beta \rightarrow \mathcal{R}[i].\alpha$, $\mathcal{L}$))
13:     add $\mathcal{R}[i].\alpha \rightarrow \mathcal{R}[i].\beta$ to LR
14:   else
15:     eliminate $\mathcal{R}[i]$ from the list
16: return LR
```

Figure 5.4: Validating likely rules.

5.5 Application for URL canonization

Finally, we explain how the discovered DUST rules may be used for canonization of a URL list. Our canonization algorithm is described in Figure 5.5. The algorithm receives a URL u and a list of valid DUST rules, $\mathcal{R}$. The idea behind this algorithm is very simple: it repeatedly applies to u all the rules in $\mathcal{R}$, until there is an iteration in which u is unchanged (lines 6–7). It is easy to see that u is always mapped to a canonical URL to which it has a directed path in the graph $G_{\mathcal{R}}$.

```
1:Function Canonize(URL $u$, rules_list $\mathcal{R}$)
2:  for k = 1 to $MA$ do
3:    prev := u
3:    for i = 1 to $|\mathcal{R}|$ do
4:      u := A URL obtained by applying $\mathcal{R}[i]$ to u[1]
5:    if (prev = u)
6:      break
7:  output u
```

Figure 5.5: Canonization algorithm.

We limit the number of iterations to the parameter MA, because otherwise the algorithm could have entered an infinite loop (if the graph $G_{\mathcal{R}}$ contains cycles). Since MA is a constant,

[1]There may exist several ways of applying a rule to a URL. Our discussion assumes that one standard way is always chosen.

chosen independently of the number of rules, the algorithm's running time is *linear* in the number of rules. Recall that the general canonization problem is hard, so we cannot expect this algorithm to always produce a minimum size canonization. Nevertheless, our empirical study shows that the savings obtained using this algorithm are high.

We believe that the algorithm's common case success stems from two features. First, our policy of choosing shrinking rules whenever possible typically eliminates cycles. Second, our elimination of refinements of valid rules leaves a small set of rules, most of which do not affect each other.

Chapter 6

Experimental Results

6.1 Experiment setup

We experiment with DustBuster on four web sites: a dynamic forum site, an academic site (www.ee.technion.ac.il), a large news site (cnn.com) and a smaller news site (nydailynews.com). In the forum site, page contents are highly dynamic, as users continuously add comments. The site supports multiple domain names. Most of the site's pages are generated by the same software. The news sites are similar in their structure to many other news sites on the web. The large news site has a more complex structure, and it makes use of several sub-domains as well as URL redirections. The academic site is the most diverse: It includes both static pages and dynamic software-generated content. Moreover, individual pages and directories on the site are constructed and maintained by a large number of users (faculty members, lab managers, etc.)

In the academic and forum sites, we detect likely DUST rules from web access logs, whereas in the news sites, we detect likely DUST rules from a crawl log. The small news site crawl includes $9,456$ URLs (see Table 6.1) and the large news site crawl includes $11,883$ unique URLs. In these two cases, the log size is equal to the number of unique URLs since the URL list is produced from a crawl and we do not crawl the same page twice. In the forum and academic sites, we repeat our experiment four times with four different access logs pertaining to different time periods. In the forum site, each of the four logs covers a single day. Each daily log includes roughly 40,000 entries, accounting for around 15,000–20,000 unique URLs (as some URLs are accessed multiple times). In the academic site, there is less traffic, hence we use logs representing roughly a week of activity. Each weekly log includes between 300,000 and 500,000 entries, accounting for around 15,000 unique URLs. In the validation phase, we use random entries from additional logs, different from those used to detect the rules. The canonization algorithm is tested on yet another log, different from the ones used to detect and validate the rules.

Web Site	Log Size	Unique URLs
Forum Site	38816	15608
Academic Site	344266	17742
Large News Site	11883	11883
Small News Site	9456	9456

Table 6.1: Log sizes.

6.2 DustBuster parameter settings

Although the parameters are only heuristically chosen to obtain good results, we use the same parameter values for all four sites and our empirical results demonstrate that these values give good results for sites that vary in structure and size. In our experiments, the maximum substring length, S, is set to 35 tokens, greater than the width of the widest rule. S should be big enough to include the widest rule, which includes the most context and refines all other redundant rules. This ensures the effectiveness of the second phase in eliminating redundant rules. For example, the rule `http://www.cnn.com` → `http://cnn.com` refines both "www.cnn" → "cnn" and `http://www` → `http://`. However, if S is set to 10, this rule will not be discovered, and since neither of the other two rules refines the other, they will both appear in the likely DUST rules. On the other hand, the larger S is, the longer the algorithm runs.

The max bucket size used for detecting DUST rules, T_{low}, was set to 6, and the max bucket size used for eliminating redundant rules, T_{high}, was set to 11. The former is set to a lower value because we are only interested in rules whose support comes from small buckets. On the other hand, the latter is used to compare the support sizes of two rules in order to test whether one refines the other. In this context, using a small threshold gives noisy results, so we use a larger threshold in order to increase the consistency of these counts.

In the elimination of redundant rules, we allowed a relative deficiency, MRD, of up to 5%, and an absolute deficiency, MAD, of 1. The implication of setting these parameters to higher values is increasing the likelihood of eliminating more general valid rules. On the other hand, using lower values may lead to missing some of the redundant rules. The maximum window size, MW, was set to 1100 rules in our experiments. The lower this value is, the more likely it is that some redundant rules will not be eliminated. On the other hand, a higher value increases the memory usage of the algorithm. Empirically, our choice appears to provide a good tradeoff.

The value of MS, the minimum support size, was statically set to 3 in all our experiments so the reduction in the crawl will be high. The algorithm uses a validation count, N, of 100. In all our experiments, extra validations beyond 50 and up to 100 do not affect the results.

Finally, the canonization uses a maximum of 10 iterations. Again, empirically, the reduction achieved remains the same for any number of iterations beyond 2 and up to 10.

6.3 Shingles

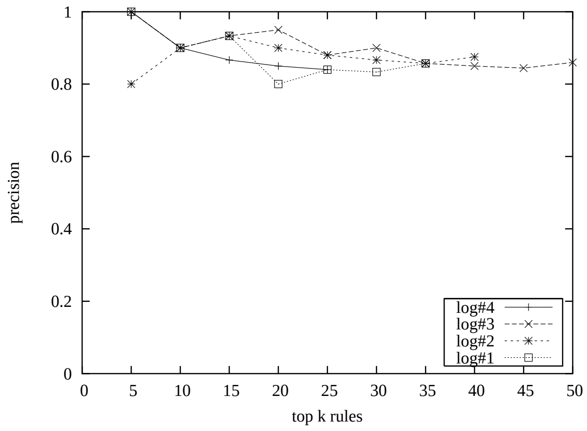
The validation uses shingling to detect similarity of html, text, pdf and ps files converted to text. Given document D, $S_n(D)$ is defined as the set of all word sequences of length n, or *n-grams*, in D (n=10 in our experiments). A hash is then applied to each of these n-grams, and the hashed n-grams are called *shingles* [9]. The resemblance of two documents A and B is then defined as $r(A, B) = \frac{S_n(A) \cap S_n(B)}{S_n(A) \cup S_n(B)}$. Shingles can be used to efficiently test the resemblance between two documents. For a given document D, the shingle with the lowest value can be used as a document sketch. The probability that two such shingles are identical is equal to the resemblance of the associated documents. Two text documents d_1, d_2 are said to be *similar*, if the resemblance between them exceeds $1 - \delta$, where $\delta > 0$ is some small tunable threshold value. Binary documents are hashed using the MD5 message digest algorithm [24] and compared for an exact match.

In our experiments, n-grams of length 10 are extracted from each document, a hash is applied to all the n-grams to produce shingles, and the shingle with the lowest value is chosen. The process is repeated 4 times with four different hash functions resulting in a document sketch of 4 shingles. In most of our experiments, unless stated otherwise, when comparing two documents, we require that all 4 shingles be identical. In some of our experiments we require that only two shingles of the four are similar.

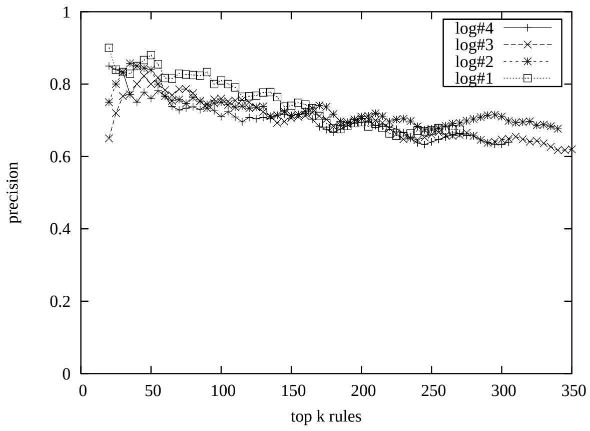
It should be noted that our algorithm is agnostic to the method used to estimate the similarity of two documents. Although the shingles method is effective, efficient, and widely used, it has its limitations. For example, in the small news site, the print version of the article includes many additional links, anchor text and images. When requiring that four of the four shingles for each document be equal, and setting ϵ to 5%, the rule "`http://www.nydailynews.com/news/v-pfriendly/story/`" $\rightarrow$ "`http://www.nydailyne-ws.com/news/story/`" does not pass the validation phase. This rule passes the validation phase when we set ϵ to 10% and declare two documents similar if at least two of the four shingles are equal. There is of course a tradeoff between the dust coverage achieved and precision.

6.4 Detecting likely DUST rules and eliminating redundant ones

DustBuster's first phase (cf. Figure 5.1) scans the log and detects a very long list of likely DUST rules (substring substitutions). Subsequently, the redundancy elimination phase (cf. Figure 5.2) dramatically shortens this list. In all of our experiments, the latter phase has eliminated over 90% of the rules in the original list (see Table 6.2). For example, in the largest log in the academic site, 26,899 likely rules were detected in the first phase, and only 2041 (8%) remained after the second; in a smaller log 10,848 rules were detected, of which only 354 (3%) were not eliminated. In the large news site 12,144 were detected, 1243 remained after the second phase. In the forum site, much fewer likely rules were detected, e.g., in one log 402 rules were found, of which 37 (9%) remained. We believe that the smaller number



(a) Forum site, with size matching, 4 logs



(b) Academic site, with size matching, 4 logs

Figure 6.1: Precision@k achieved by likely rule detection:
(DustBuster's first phase) *without* fetching actual content.

of rules is a result of the forum site being more uniformly structured than the academic one, as most of its pages are generated by the same web server software.

Web Site	Rules Detected	Rules after 2^{nd} Phase	Rules Remaining (percent)
Forum Site	402	37	9.2%
Academic Site	26899	2041	7.6%
Large News Site	12144	1243	9.76%
Small News Site	4220	96	2.3

Table 6.2: Rule elimination in second phase.

In Figures 6.1(a) and 6.1(b), we examine the precision level in the short list of likely rules produced at the end of these two phases. These graphs deal with the forum and academic sites when size matching is used. Recall that no page contents are fetched in these phases. As this list is ordered by likeliness, we examine the *precision@k*; that is, for each top k rules in this list, the curves show which percentage of them are later deemed valid (by DustBuster's validation phase) in at least one direction. Figure 6.1(a) shows results obtained with 4 different logs in the forum site, and Figure 6.1(b) shows results from 4 logs of the academic site. We observe that, quite surprisingly, DustBuster's detection phase achieves a very high precision rate even though it does not fetch even a single page. In the forum, out of the 40–50 detected rules, over 80% are indeed valid. In the academic site, over 60% of the 300–350 detected rules are valid, and of the top 100 detected rules, over 80% are valid.

This high precision is achieved, to a large extent, thanks to size matching. The log includes false rules. For example, the forum site includes multiple domains, and the stories in each domain are different. Thus, although we find many pairs `http://domain1/story_num` and `http://domain2/story_num` with the same num, these represent different stories. Similarly, in the academic site, we see pairs like `http://site/course1/lecture-num.ppt` and `http://site/course2/lecture-num.ppt`, although the lectures are different. Such false rules are not detected, since stories/lectures typically vary in size. Figure 6.2 illustrates the impact of size matching in the academic site. We see that when size matching is not employed, the precision drops by around 50%. Thus size matching reduces the number of accesses needed for validation. Nevertheless, size matching has its limitations– valid rules may be missed at the price of increasing precision. For example, when running on the academic site without size matching, DustBuster discovers the rule "`ps`" → "`pdf`", linking two versions of the same document, whereas size matching discards this rule. When running DustBuster from crawl logs, where document sketches from the previous crawl are readily available for validation, we use the document sketches to filter support rather than the size matching heuristic.

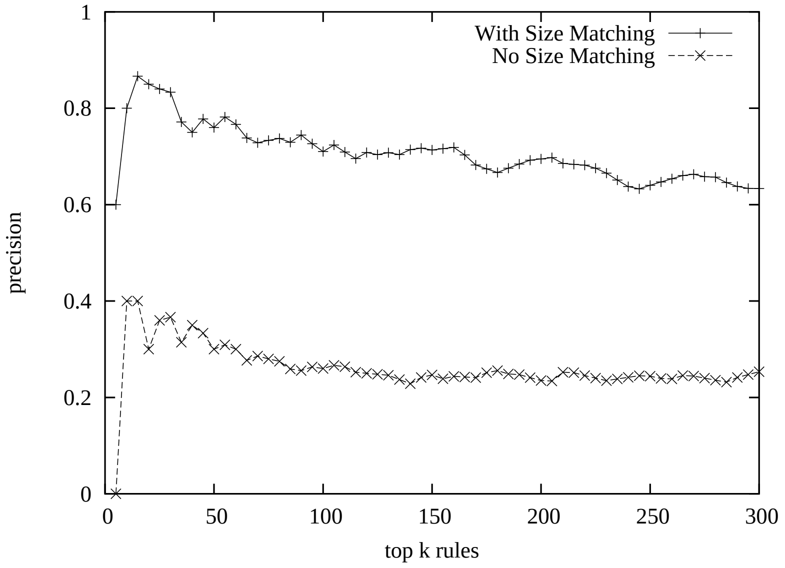
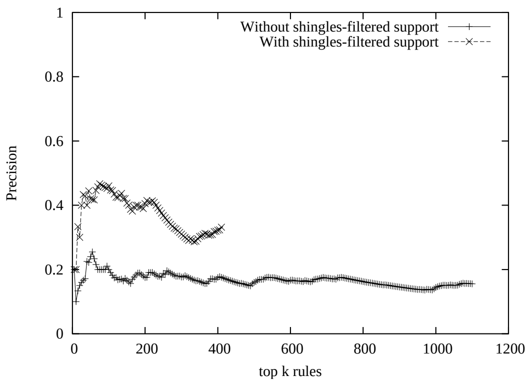


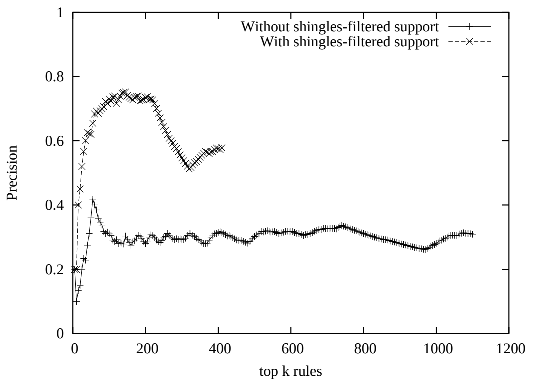
Figure 6.2: Impact of size matching on precision:
Precision@k achieved by likely rule detection (DustBuster's first phase) *without* fetching actual content. One academic log.

Figure 6.3(a) shows results obtained in the large news site (cnn.com) with and without shingles-filtered support. When we use neither shingles-filtered support nor size matching, over 24% of the first 50 detected rules are valid. The precision rises to 42% of the first 50 detected when we use shingles-filtered support. In addition, filtering reduces the list by roughly 70%. At the same time this reduction in detected rules has little impact on the final outcome after validations, most rules filtered out by shingles are indeed invalid.

The precision@k in the large news site is low compared to the precision@k in the academic site shown in Figure 6.1(b). When examining the likely rule list we observe that some



(a) Impact of shingle matching



(b) Impact of shingle matching with lower similarity threshold

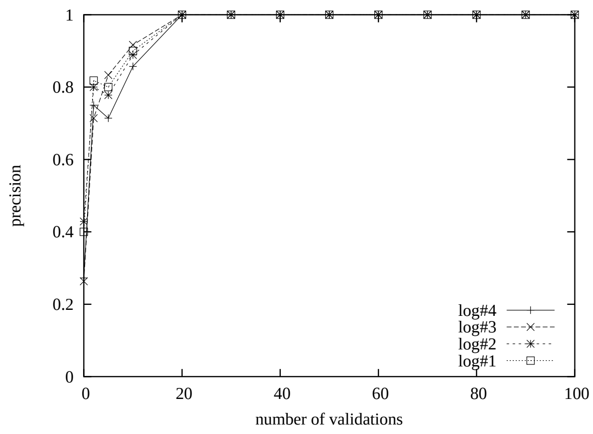
Figure 6.3: Impact of shingle matching:
Precision@k achieved by likely rule detection (DustBuster's first phase) *without* fetching actual content.

rules that we expect to be valid, such as "`http://edition.cnn.com/2006/SHOWBIZ/`" $\rightarrow$ "`http://www.cnn.com/2006/SHOWBIZ/`", do not pass the validation phase. After reducing the similarity threshold by requiring at least two of the four shingles to be identical and increasing ϵ to 10% these rules are found valid. We can observe the effect of changing these two values in Figure 6.3(a) and Figure 6.3(b). The precision increased both with and without filtering support for similarity. The precision@k went up from 0.4% at 200 to 0.74%. This demonstrates the tradeoff among the similarity requested, the validation threshold, and the number of rules found to be valid.

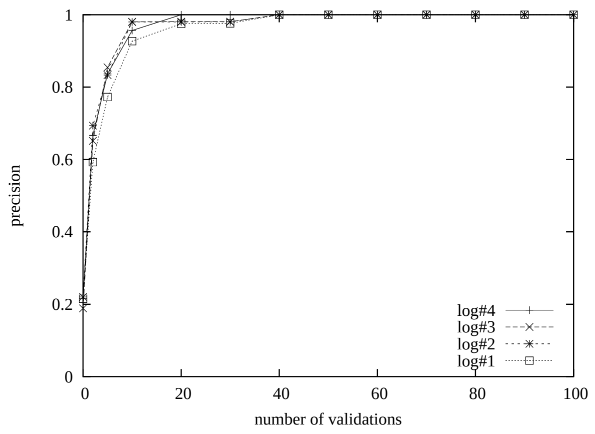
An interesting phenomenon is apparent in Figures 6.2 and 6.3(a), where the precision begins low and rises. This is due to the large support for rules that transfer one valid URL to another albeit not with similar content. E.g., the academic site's top five rules include four variations on the interesting rule "`/thumbnails/`" $\rightarrow$ "`/images/`" which we find invalid because images are compared using their MD5 message digests. Another example is "`http://www.movies-forum.domain`" $\rightarrow$ "`http://www.politics-forum.domain`" found using the forum web logs. The large support for this rule stems from the similar URL structure of the two sub-sites, which are running the same software. For example, "`http://www.politics-forum.domain/story_100`" and "`http://www.movies-forum.domain/story_100`" are valid but different in content. Size matching alleviates this problem, and raises the initial precision to 0.6 at 5 and 0.8 at 10.

6.5 Validation

We now study how many validations are needed in order to declare that a rule is valid; that is, we study what the parameter N in Figure 5.4 should be set to. To this end, we



(a) Forum, 4 different logs



(b) Academic, 4 different logs

Figure 6.4: Precision among rules that DustBuster attempted to validate vs. number of validations used (N).

run DustBuster with values of N ranging from 0 to 100, and check which percentage of the rules found to be valid with each value of N are also found valid when N=100. We run this experiment only for substring substitution rules. The results from conducting this experiment on the likely DUST rules found in 8 logs (4 from the academic site and 4 from the forum site) are shown in Figures 6.4(a) and 6.4(b). In these graphs, we only consider rules that DustBuster attempts to validate. Since many valid rules are removed (in line 6 of Figure 5.4) after rules that they refine are deemed valid, the percentage of valid rules among those that DustBuster attempts to validate is much smaller than the percentage of valid rules in the original list. In all these experiments, 100% precision is reached after 40 validations. Moreover, results obtained in different logs are consistent with each other. Our aggressive elimination of redundant rules reduces the number of rules we need to validate. For example, on one of the logs in the forum, the validation phase was initiated with 28 pairs representing 56 likely rules (in both directions). Of these, only 19 were checked, and the rest were removed because they or their counterparts in the opposite direction were deemed valid either directly or since they refined valid rules. We conclude that the number of actual pages that need to be fetched in order to validate the rules is very small.

At the end of the validation phase, DustBuster outputs a list of valid substring substitution rules without redundancies. In the small news site we detected 5 rules (see Table 6.3), whereas in the large news site we found 62 rules. In the forum site, we detect a handful of rules, whereas in the academic site, 40–52 rules are found. The list of 7 rules found in one of the logs in the forum site is depicted in Figure 6.5 below. These 7 rules or refinements thereof appear in the outputs produced using each of the studied logs. Some studied logs include 1–3 additional rules, which are insignificant (have very small support). Similar consistency is observed in the academic site outputs. We conclude that the most significant DUST rules can be adequately detected using a fairly small log with roughly 15,000 unique URLs.

Web Site	Valid Rules Detected
Forum Site	7
Academic Site	52
Large News Site	62
Small News Site	5

Table 6.3: The number of rules found to be valid.

1	"`.co.il/story_`"	→	"`.co.il/story?id=`"
2	"`&LastView=&Close=`"	→	""
3	"`.php3?`"	→	"`?`"
4	"`.il/story_`"	→	"`.il/story.php3?id=`"
5	"`&NewOnly=1&tvqz=2`"	→	"`&NewOnly=1`"
6	"`.co.il/thread_`"	→	"`.co.il/thread?rep=`"
7	"`http://www.scifi.forum/story_`"	→	"`http://www.scifi.forum/story?id=`"

Figure 6.5: Valid substring substitution rules detected in a forum site
excluding refinements of valid rules.

6.6 Coverage

We now turn our attention to coverage, or the percentage of duplicate URLs discovered by DustBuster, in the academic site. When multiple URLs have the same document sketch, all but one of them are considered duplicate URLs, or *duplicates*. We use a new test log in order to study how much of the duplicates in it are detected by rules found by running DustBuster on another log. We detect duplicates in the test log by fetching the contents of all of its URLs, and computing their document sketches. Figure 6.6 classifies these duplicates. As the figure shows, 47.1% of the duplicates in the test log are eliminated by DustBuster's canonization algorithm using rules discovered on another log. An additional 25.7% of the duplicates consist of images, such as arrows, bullets and horizontal lines, which are reused and copied to various locations. This is partly due to the use of automatic tools, which make use of the same images.

17.9% of duplicates stem from exact duplicates of files. This is mainly due to replicating a course's site from a previous semester and modifying it. In case files are added or modified extensively, the DUST rule linking the two semesters does not pass the validation phase. Another cause for exact copies is a research paper appearing in all the authors individual home-pages.

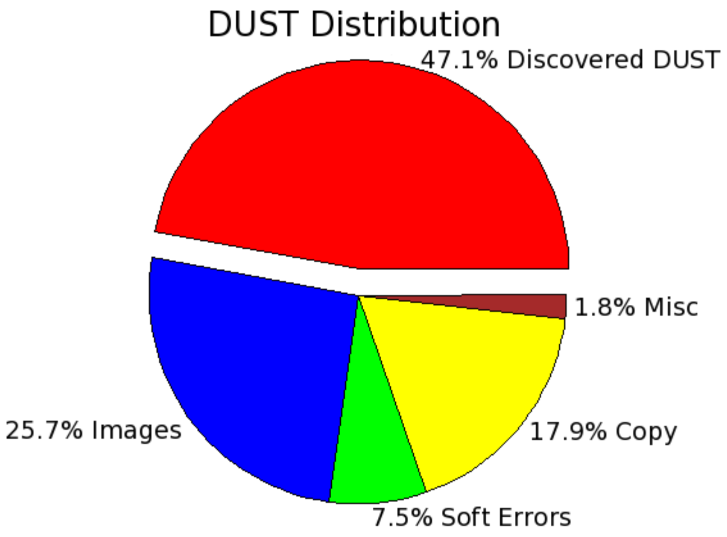


Figure 6.6: DUST classification, academic.

Another 7.5% of the duplicates are due to soft errors, such as empty search results, non-existing user error messages, and various software errors, or in general, pages that do not provide any useful information. Note that classifying these as duplicates is only an artifact of our methodology, which computes similar shingles for similar error messages. Nevertheless, this type of duplicates cannot be expected to be discovered by an algorithm such as DustBuster. Overall, DustBuster appears to discover most of the duplicates that stems from systematic rules, and discovers the majority of duplicates when discounting soft errors.

6.7 Savings in crawl size

The next measure we use to evaluate the effectiveness of the method is the discovered redundancy, i.e., the percent of the URLs we can avoid fetching in a crawl by using the DUST rules to canonize the URLs. To this end, we performed a full crawl of the academic site, and recorded in a list all the URLs fetched. We performed canonization on this list and counted the number of unique URLs before (U_b) and after (U_a) canonization. The discovered redundancy is then given by $\frac{U_b - U_a}{U_b}$. We found this redundancy to be 18% (see Table 6.4), meaning that the crawl could have been reduced by that amount. In the two news sites, the DUST rules were learned from the crawl logs and we measured the reduction that can be achieved in the next crawl. The small news site gave us less than 0.9% reduction when requiring that four shingles be equal and $\epsilon = 5\%$, but when requiring two shingles to be equal and setting ϵ to 10% we get a 26% reduction. The increase in the reduction is due to the lower similarity required to pass validation. This again shows the significance of the similarity measure used. The big news site gave us a 6% reduction with $\epsilon = 5\%$. We did not crawl the forum site and therefore do not have results for this site.

Web Site	Reduction Achieved
Academic Site	18%
Large News Site	6%
Small News Site	26%

Table 6.4: The reduction achieved in each site.

Chapter 7

Conclusions

We have introduced the problem of mining site-specific DUST rules. Knowing about such rules can be very useful for search engines: It can reduce crawling overhead by up to 26% and thus increase crawl efficiency, and consequently, improve search accuracy. It can also reduce indexing overhead. Moreover, knowledge of DUST rules is essential for canonizing URL names, and canonical names are very important for statistical analysis of URL popularity based on PageRank or traffic. We presented DustBuster, an algorithm for mining DUST very effectively from a URL list. The URL list can either be obtained from a web server log or a crawl of the site.

DustBuster cannot of course detect DUST that does not stem from rules. However, there are DUST rules that DustBuster does not find. An example is a rule that substitutes two strings in the URL simultaneously. For example in a site we examined `http://somenewsite.-com/print.pl?sid=06/06/27/2` is a printer friendly version of `http://somenewsite.com/-newsvac/06/06/27/2.shtml`. As can be seen, the URL is changed in two places simultaneously. Future work may devise an algorithm to detect such rules, as well as more complex ones.

Finally, some pages on the web have duplicate versions where the information is spread over multiple pages. Future work may use ideas from DustBuster in order to devise an algorithm that detects such duplicates. For example, by using containment rather than similarity, one may be able to map all the fragment pages to the one page that contains them all.

Bibliography

[1] Apache http server version 2.2 configuration files. `http://httpd.apache.org/docs/2.2/configuring.html`.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB)*, pages 487–499, 1994.

[3] Analog. `http://www.analog.cx/`.

[4] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform resource identifiers (URI): Generic syntax. `http://www.ietf.org/rfc/rfc2396.txt`.

[5] K. Bharat and A. Z. Broder. Mirror, Mirror on the Web: A Study of Host Pairs with Replicated Content. *Computer Networks*, 31(11–16):1579–1590, 1999.

[6] K. Bharat, A. Z. Broder, J. Dean, and M. R. Henzinger. A comparison of techniques to find mirrored hosts on the WWW. *IEEE Data Engineering Bulletin*, 23(4):21–26, 2000.

[7] J. Borges and M. Levene. Data mining of user navigation patterns. In *Proceedings of the International Workshop on Web Usage Analysis and User Profiling (WEBKDD).*, pages 92–111, 1999.

[8] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Proceedings of the 7th International World Wide Web Conference (WWW)*, pages 107–117, 1998.

[9] A. Z. Broder, S. C. Glassman, and M. S. Manasse. Syntactic clustering of the web. In *Proceedings of the 6th International World Wide Web Conference (WWW)*, pages 1157–1166, 1997.

[10] J. Cho, N. Shivakumar, and H. Garcia-Molina. Finding replicated web collections. In *Proceedings of the 19th ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 355–366, 2000.

[11] C. de la Higuera. Current trends in grammatical inference. In *Proceedings of the 8th International Workshop on Structural and Syntactic Pattern Recognition (SSPR), 3rd*

International Workshop on Statistical Techniques in Pattern Recognition (SPR), pages 28–31, 2000.

[12] F. Douglis, A. Feldman, B. Krishnamurthy, and J. Mogul. Rate of change and other metrics: a live study of the world wide web. In *Proceedings of the 1st USENIX Symposium on Internet Technologies and Systems (USITS)*, 1997.

[13] R. A. Finkel, A. B. Zaslavsky, K. Monostori, and H. W. Schmidt. Signature extraction for overlap detection in documents. In *Proccedings of the 25th Australasian Computer Science Conference (ACSC)*, pages 59–64, 2002.

[14] Google Inc. Google sitemaps. `http://sitemaps.google.com`.

[15] D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational*. Cambridge University Press, 1997.

[16] N. Jain, M. Dahlin, and R. Tewari. Using bloom filters to refine web search results. In *Proceedings of the 7th International Workshop on the Web & Databases (WebDB)*, pages 25–30, 2005.

[17] T. Joachims. Optimizing search engines using clickthrough data. In *Proceedings of the 8th International Conference on Knowledge Discovery and Data (SIGKDD)*, pages 133–142, 2002.

[18] T. Kelly and J. C. Mogul. Aliasing on the world wide web: prevalence and performance implications. In *Proceedings of the 11th International World Wide Web Conference (WWW)*, pages 281–292, 2002.

[19] T. K. Landauer and S. T. Dumais. A solution to plato's problem: The latent semantic analysis theory of acquisition, induction, and representation of knowledge. *Psychological Review*, (104):211–240, 1997.

[20] H. Liang. A URL-String-Based Algorithm for Finding WWW Mirror Host. Master's thesis, Auburn University, 2001.

[21] D. Lin. Automatic retrieval and clustering of similar words. In *Proceedings of the 36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics (COLING-ACL)*, pages 768–774, 1998.

[22] K. Monostori, R. A. Finkel, A. B. Zaslavsky, G. Hodász, and M. Pataki. Comparison of overlap detection techniques. In *Proceedings of the 10th International Conference on Conceptual Structures (ICCS)*, pages 51–60, 2002.

[23] R. Reischuk and T. Zeugmann. Learning one-variable pattern languages in linear average time. In *Computational Learing Theory*, pages 198–208, 1998.

[24] R. Rivest. The MD5 message-digest algorithm. http://www.ietf.org/rfc/rfc1321.txt.

[25] StatCounter. http://www.statcounter.com/.

[26] P. D. Turney. Mining the Web for Synonyms: PMI-IR versus LSA on TOEFL. In *Proceedings of the 12th European Conference on Machine Learning (ECML)*, pages 491–502, 2001.

[27] WebLog Expert. http://www.weblogexpert.com/.

[28] J. Zobel and A. Moffat. Exploring the similarity space. *SIGIR Forum*, 32(1):18–34, 1998.