Enhancing crawling coverage of dynamic web applications by automatically detecting content-affecting parameters in HTTP requests

Authors et al.: Disclosed Anonymously
IP.com number: IPCOM000215791D
IP.com Electronic Publication: March 12, 2012
Enhancing crawling coverage of dynamic web applications by automatically detecting content-affecting parameters in HTTP requests

1. Background

There are generally two ways to modify the content of a page in a web site:

1. Using a different URL (e.g., page1.html and page2.html)
2. Using a single URL with different parameter values (e.g., index.php?content=TV&day=25 and index.php?content=car&day=30)

Parameters that affect the contents presented to the user are referred to as "content-affecting parameters", whereas parameters that do not - are called "superfluous parameters" (as the day parameter in the example above). The identification of a parameter as a content-affecting one is of great importance for a successful completion of many tasks. For example, an automated web crawler has to follow links containing a content-affecting parameter with a new value (in order to gain full coverage), while ignoring links in which only superfluous parameters change their value (in order to avoid exploring the same page twice, possibly entering an endless loop).

A simple and naïve solution for the problem of identifying content-affecting parameters, which simply reports on every multiple-value parameter, is deemed to fail: an application may contain a superfluous parameter whose value frequently changes.

A different solution that currently exists is to maintain a hard-coded list of known content-affecting parameters. This solution suffers from the inherent problems of using hard-coded information: There is no way to detect parameters with non-common names as content-affecting ones. Moreover, this method automatically considers parameters appearing in the list as content-affecting ones, even if they do not play this role in the tested application. Needless to say, that this approach almost surely fails when dealing with applications that have content written in foreign languages.

A better notion than the previous one (although without a specific algorithm) was suggested in [*]. This implies that a parameter that changes its value between two requests with substantially the same responses - is a likely candidate for being a superfluous parameter.

This article presents a more generalized, statistical-learning based method that is less prone to cases where parameters might not behave as expected (for example: a superfluous parameter that does not change its value although the response stays the same). In addition, the method uses cases where the content does change, to infer about the role of each parameter. Thus the establishment of the decision is based on more information.

The rest of the paper is organized as follows: Section 2 provides an overview of the system. Sections 3 and 4 describe methods to determine how well a parameter functions as a content-affecting parameter. Section 5 suggests a method for setting a threshold score above which a parameter is considered to be a content-affecting one. Section 6 deals with the problem of how to decide if two responses are similar or not, and section 7 concludes the paper.

2. Overview of the system

The system performs its analysis after an initial scan of the application, where it is exposed only to a limited number of requests (and their responses). These requests
are named *sent requests*. In addition, there exists a path-limit constraint, which allows the initial scan to send a bounded number of requests sharing the same path (requests that are not sent due to this limitation are referred to as *filtered requests*). The system then organizes all sent requests into **groups according to their path** (every group represents one path). Analysis is applied only to groups with a path that could have been visited more, if the path-limit constraint was not enforced. It is worth mentioning that the system discards URLs that are non-worthy for the analysis, such as static pages (JavaScript files or text files). These pages can not be influenced by the value of parameters.

As the method requires a certain amount of requests (with corresponding responses) in order to make reasonable decisions, it possibly needs to **sample more requests** from the filtered requests list. As mentioned above, some requests may be encountered during the initial scan, but not sent due to the path-limit constraint. If the number of sent requests for a certain path is lower than a certain threshold, there is a need to sample more requests of that path (and fetch their responses). Sampling is done uniformly over the filtered requests, so that more frequent parameters have a better chance of being chosen.

Having enough data from every group of requests, the system now **analyzes the parameters** appearing in every group. Every parameter is assigned with a score that describes the extent to which that parameter can explain content changes between different responses. The score reflects how correlated the parameter is with the content being served by the application. Parameters that are assigned with a high enough score – are considered as content-affecting ones.

The system can apply two methods in order to calculate the correlation between a certain parameter and the behavior of the application. More precisely, the system calculates a correlation between the changes of the parameter's values and the changes of the content of the pages: content should change when a content-affecting parameter changes its value, and vice versa. Section 3 and 4 describe possible methods for the calculation of such a score. Both assumes the existence of an algorithm that can decide whether two pages have the same content or not (possible implementation of such an algorithm is described in section 6).

3. **Vector based approach for correlation calculation**

   This approach represents data using vectors, both for storing the parameter's different values, and for storing data regarding the responses. Both vectors are \( d \)-dimensional vectors, where \( d \) is the number of requests that are analyzed in every group. Every entry corresponds to a different request. Setting \( d \) with the value of 13 is usually enough. The data corresponding to the responses is represented by a vector of response indices. Using an algorithm to determine similarity between responses, we can assign responses with indices, such that every pair of similar responses shares the same index, and every pair of different responses has two different indices.

   Representation of a parameter's data is done in the same manner: every possible value of a parameter is assigned with an index, and the final result is a \( d \)-dimensional vector whose \( i \)-th entry represents the index of the value of the parameter in the \( i \)-th request. Note, that the case of a parameter not appearing in a certain request needs to be treated specifically (for example by setting its value to zero).

   Having the responses vector and the various parameters vectors, the system can
now maintain a **match vector** for every parameter: the $i$-th entry of this vector would represent a match (+1) or a mismatch (-1) between the corresponding response index and the corresponding parameter value index (or 0 if the parameter does not appear in that request).

The final score of the parameter is obtained by summing the entries of its match vector (possibly after normalizing the entries using the number of parameters in each URL). This way, the algorithm favors parameters whose value changes when the response changes, and vice versa.

In order to obtain reliable results, this algorithm needs to assign indices in a manner that will result in a maximum possible match between response indices and parameter values indices. This is not a trivial problem, but a solution can be found using an algorithm for finding a maximum weighted matching in a bipartite graph, as described in the following paragraphs.

**An algorithm for assigning indices using maximum weighted matching**

The algorithm starts by assigning indices for every response and for every parameter value in an arbitrary way (for example: by the order of requests).

Denote by $V : v_1, ..., v_n$ the indices of the parameter's values (some may be zero). Denote by $U : u_1, ..., u_n$ the response indices (positive integers).

The algorithm searches for a partial mapping $F : V \to U$ that given a parameter index $v$, will possibly output a new index $F(v)$, such that the number of indices $i$ for which $u_i = F(v_i)$ is maximal. Having this mapping, the new indices for the parameter values should be obtained.

The algorithm constructs a bipartite graph with all possible unique parameter indices (excluding zero) on one side (marked as $k_1, ..., k_p$), and all possible unique response indices on the other side (marked as $l_1, ..., l_q$).

For every pair $(k_i, l_j)$ we add an edge with weight $c$, where $c$ is the number of requests with response index $k_i$ and parameter value index $l_j$.

Solving the maximum weighted matching problem for the bipartite graph yields the optimal translation between indices.

As an illustration, the following figure shows the graph produced for a parameter vector (1, 1, 1, 2, 2, 2) and a response vector (1, 2, 3, 4, 4, 4):
4. Matrix based approach for correlation calculation

The previous algorithm can be modified to use matrices instead of vectors. The change increases the memory consumption of the algorithm, but avoids the problem of indices assignment.

Every matrix captures the similarity between various responses (or parameters). Data is kept in an upper triangular matrix (due to symmetry), and the entry \((i, j)\) corresponds to the similarity of objects \(i\) and \(j\), thus taking the values of -1 or 1 (or 0 if the parameter does not appear in the URL).

The match matrix is obtained by multiplying matrices element-by-element, and the final score is the sum of the elements of the match matrix.

6. Setting a threshold for content-affecting parameters

The value of the threshold above which a parameter is considered a content-affecting parameter can greatly influence the performance of the system.

In general, a content-affecting parameter is expected to appear in most of the requests, and to truly affect the content presented to the user. However, there could be cases where a content-affecting parameter is not part of the request data (for example: a request for the home page of a web application), and there could also be cases where the application does not behave in a typical manner (for example: same content is served although the parameter's value changes). The latter can also happen due to mistakes in the page similarity algorithm.

The system, therefore, uses the following formula to calculate the threshold:

\[
\text{Threshold} = N \times AR \times (1 - 2FR)
\]

where \(N\) is the total number of analyzed changes (number of requests in the vector-based approach or number of comparisons in the matrix-based approach); \(AR\) is the minimum fraction of requests where a content-affecting is expected to appear at; and \(FR\) is the maximum expected fraction of
cases where a content-affecting parameter can fail in explaining the behavior of the application (the coefficient 2 is due to the fact that in case of a failure the score strictly decreases).

5. Detection of similarity between web pages

When coming to determine whether two pages are similar or not, the system adopts the notion that two pages are similar if it is worthwhile for a web scanner that seeks for security issues to scan both pages in order to achieve a full coverage of the application (other notions can be used as well). Following this notion, two pages serving the same content, but in different languages – are considered the same. The same applies for two pages serving, for example, content regarding two versions of the same product. However, two pages concerning products from different sections in the application are not considered the same. For this purpose, the system utilizes a similarity algorithm that

- Analyzes the page structure (rather than its textual content). This is done by extracting the response Document Object Model (DOM).
- Allows some degree of freedom between similar pages. This can be done by using Levenshtein edit distance as a measure of distance between the structures of pages: if the edit distance is lower than a certain threshold, the pages are considered the same. Using locally sensitive hash functions on the page structure can also achieve the desired result.

7. Conclusion

The paper presented a system that can detect parameters which have a direct influence on the content that is served to the user, amongst all application parameters. This ability can greatly enhance the crawling coverage of the application. Some technical details that can improve the performance of the system were omitted from the description of the system for the sake of simplicity.

8. References