MOBILE DEVICE DETECTION BASED ON USER AGENT STRINGS

G.C. Hocking*

Study Group participants

Colin Please, Ludovic Tangpi, Asha Tailor, Dario Fanucchi,
Byron Jacobs, Shaun Kimmelman and Graeme Hocking

Industry representatives

Rumbidzai Mukungunugwa and Ismail Dhorat

Abstract

The study group was asked to consider methods by which the identification signal from a mobile device, called a *user-agent string*, is recognized and processed and to determine if this could be done faster. Two approaches were taken. In the first a search of string matching techniques in the existing literature was conducted to identify any improvements and secondly some code was written and some simple preprocessing of the database was conducted. Both strands promised to yield some significant improvements in search time.

1 Introduction

When any computer or mobile device contacts a server to request a web page, the request includes what is known as a "user agent" string, which identifies the device and the type of browser being used, for example, manufacturer, device type and screen size. Typical examples of such strings are:

*School of Chemical and Mathematical Sciences, Murdoch University, Perth, Western Australia.
email: G Hocking@murdoch.edu.au

Mozilla/5.0 (iPod; U; CPU iPhone OS 3_1_1 like Mac OS X; en-us)
AppleWebKit/528.18 (KHTML, like Gecko) Mobile/7C145,

or

BlackBerry7100i/4.1.0 Profile/MIDP-2.0 Configuration/CLDC-1.1 VendorID/103.

In particular, as the number of mobile devices and versions of browsers increases it is becoming increasingly difficult to rapidly detect the type and particular features of the device in a time that is satisfactory to the user. One of the options to rapidly identify the device is to compare the "User Agent" strings (UAS) to a list on the `WURFL` (Wireless Universal Resource File). This list is continuously updated as new devices and browsers hit the market. It currently has around 13,000 entries and is growing rapidly.

However, to complicate things further, "User Agent" strings (UAS) are **not** standardised, and can in fact be changed (either deliberately or inadvertantly) by the user. Strings are in no particular format, length or order and often there is no perfect match - user agent strings may be in a different order, use different abbreviations or even be wrong. Currently, a *Levenshtein* search algorithm [7], string reduction and heuristics are used to match strings and the whole database is searched.

The challenge put to the MISG was to detect mobile devices and their relevant properties in an optimal manner based on the UAS, either by optimizing the existing algorithms or introducing a hybrid of newer faster algorithms.

The group decided to take two parallel approaches to the problem. Firstly, rather than try to develop new techniques, it was felt a thorough literature search of string matching algorithms would be an effective way to find an improvement on the existing method if one existed. Secondly, code to mimic the current process was written so that the group could "play" with any new ideas such as subdivision of the database, buffering or ordering, and finally some attempt to implement any new algorithms. Some possible work for the future was identified.

2 Literature

A search of the literature revealed two types of string matching algorithms. One involves exact matching, while the other requires only some degree of similarity or "inexact" matching. It is clear that "exact" matching of the full incoming UAS is inappropriate in this situation, but some of the proposed re-ordering methods involve searching for exact substrings within either the database or the incoming string, and consequently we must consider both.

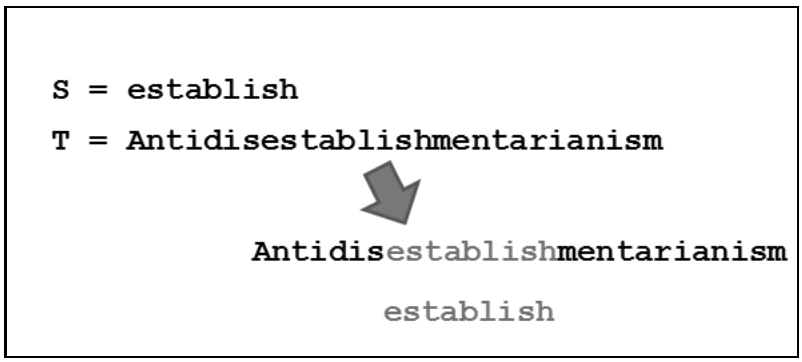

Figure 1: *Example of exact string matching. S is the incoming string and T is the list of characters.*

2.1 Exact String Matching

So-called "exact" string matching involves *alignment* and *matching* of the exact string, so that the two strings are moved backwards and forwards until there is an alignment of characters (if possible) as shown in Figure **??**.

Current algorithms in this category include;

- Brute Force - Easy to implement as the string is simply moved along the list until a match is made. This is effective, but slow with a worst case of $O(m \times n)$ operations, where m is the number of characters in the string, S, and n is the number of characters in the string being searched, T.

- Knuth-Morris-Pratt (KMP) - This algorithm [2, 6] searches along the string looking for a match with the first character. If a match is found, then it proceeds to the next character to check it. This continues until a complete match is found or a mismatch occurs. While the matching sequence is underway the algorithm also checks for the occurrence of the first character, so that if a mismatch occurs it returns to the next occurence of that character rather than to the beginning. The improvement on the above method is significant if the first character doesn't occur very often!

- Boyer-Moore Algorithm (BMA) - This algorithm [1] is the current industry standard exact-text searching algorithm. It involves jumping ahead m characters (the length of the incoming string) and searching backwards. If the last character doesn't match, then we can move on. If the last character doesn't appear in the string at all, then can jump forward another m places. Thus the method is potentially very quick for longer strings - in fact faster the longer is the string.

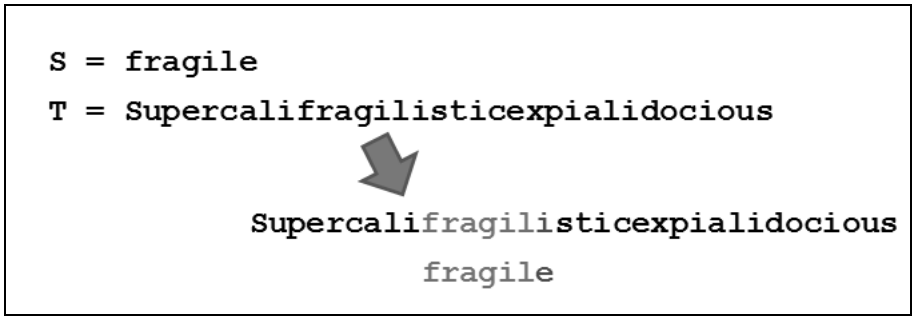

Figure 2: *An example of inexact matching. S is the incoming string and T is the list characters.*

In the current problem, however, the uncertainty in the User Agent String being sent by the mobile device means that exact matching is unlikely to be successful if an attempt is made to match the full string. Therefore, the more difficult case of "inexact" matching must be used instead.

2.2 Inexact String Matching

In this type of algorithm, we try to find the *best fit* between two strings, see Figure 2. In order to do this we need to effectively define the "distance" between the two strings, that is, come up with a definition of the difference between the two strings, so that the "closest" can be chosen. Ideally, the distance between two identical strings will be zero only if they are identical, and become larger as they differ more.

In mathematical terms these would be described as a *metric*. Some existing algorithms in the literature include;

- Hamming Distance, [5] - Number of positions at which aligned symbols are different in strings of the same length or the number of substitutions required to make the two strings identical. This only applies to strings of the same length, however, which limits its application here.
- Edit Distance (Levenshtein) which first appeared in English in [7] - This metric is defined by the smallest number of *edits* to change string S into string T. Allowable edits are substitution, deletion, or insertion of a single character. The insertion of a character allows the strings to be of different lengths, unlike the Hamming distance.
- Damerau - Levenshtein distance - This method [3] is the same as the Levenshtein edit distance with the allowance of transposition of one character with that adjacent. This allows for a simple mis-spelling for example.

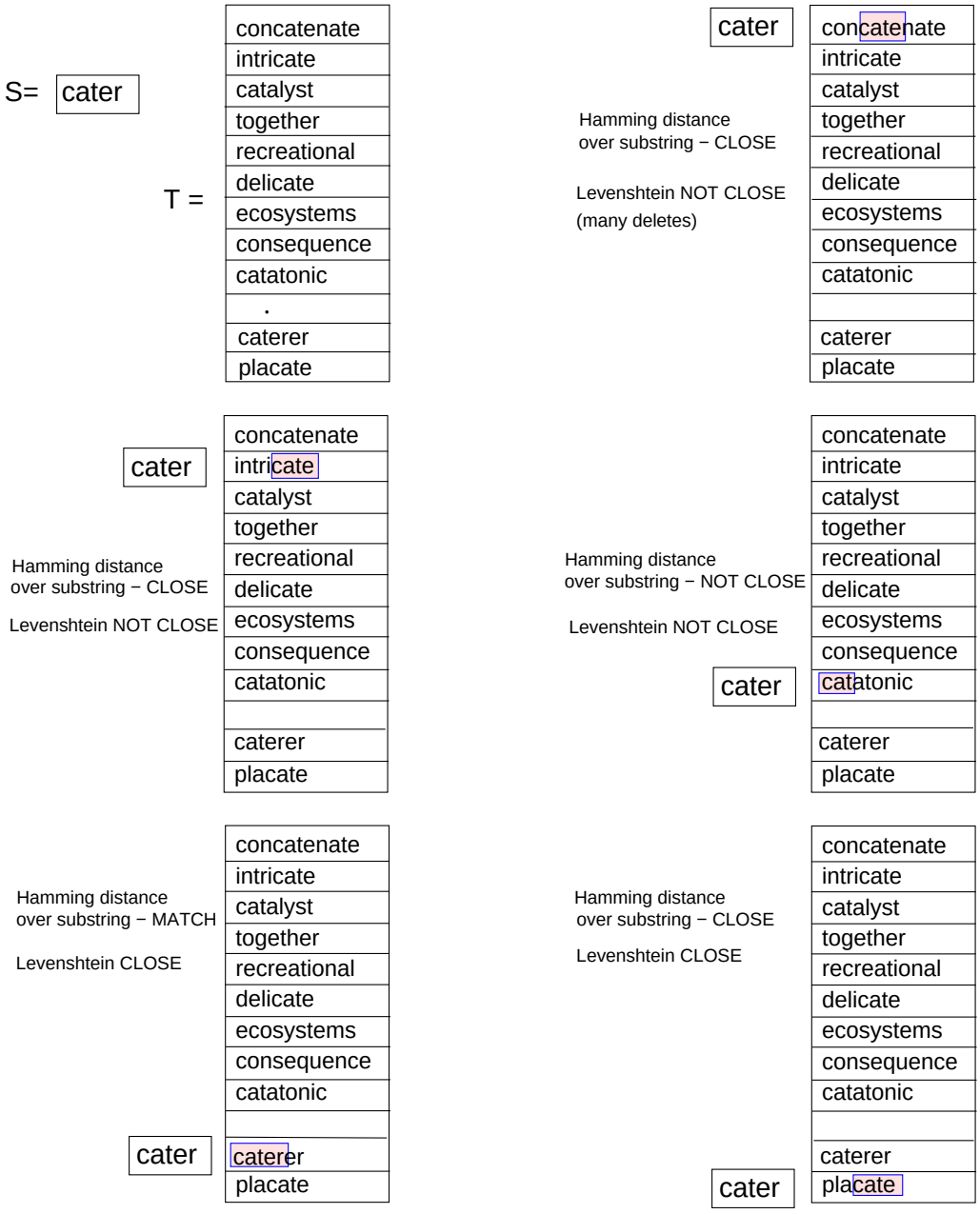

Figure 3: Comparison of Hamming distance on a substring vs. the Levenshtein algorithm over the full string - Hamming distance "appears" closer more often, so perhaps does not give such a clear differentiation.

- Longest Common Subsequence - Longest subsequence in both strings. For example, *diff* in Unix, see [4].
- Longest Common Substring - Longest common substring between the two strings. This should not be confused with the previous algorithm. A substring must be consecutive characters but a subsequence need not be - see [4]. This algorithm can be designed to be very fast by employing suffix trees (see later).

Fast dynamic programming search methods exist for each of these algorithms. Figure 3 gives a comparison of Hamming Distance verses Levenshtein edits where the Hamming distance has been restricted to look for a matching string first and then compute, that is, Hamming distance on a substring. One can see that the Hamming distance gives a closer match more often, which may not be a good thing because it provides less differentiation between cases.

2.3 Suffix Tree Example

Suffix trees [8, 9, 10] are a method for increasing the speed of the search algorithm for substrings. The idea is to pre-process the string to make the comparison search much more efficient. The tree is created by considering all of the characters in the full string. If the character occurs on multiple occasions then it is used as a branch from the root. If it only occurs on one occasion then the unique substring beginning with this character forms a branch. Thus the tree looks like Figure 4. The search then consists of following a branch in the tree, which will always be unique. If the end of the branch is reached without a match then there is no match.

In the given example, the letters in the word are i,s and p and all have multiple occurences and so each has a branch. The numbers in the squares at the terminus of each branch indicate the number of the letter at which that particular branch began. Note that it is unique for each. An incoming string can be compared against this tree and searches only need to be conducted along individual branches, greatly reducing the search time.

This method can be used either for exact or inexact (longest common substring) searches and after preprocessing of the string being searched it can be performed in $O(m)$ time if programmed efficiently.

2.4 Multiple Strings

The methods of searching a single string in this application must be extended to multiple strings, since the database consists of a list. In that case, the time of the search can in general be written as $O\left(k \times C(n,m)\right)$ where $C(n,m)$ is the cost of a particular string-matching algorithm and k is the number of items on the list.

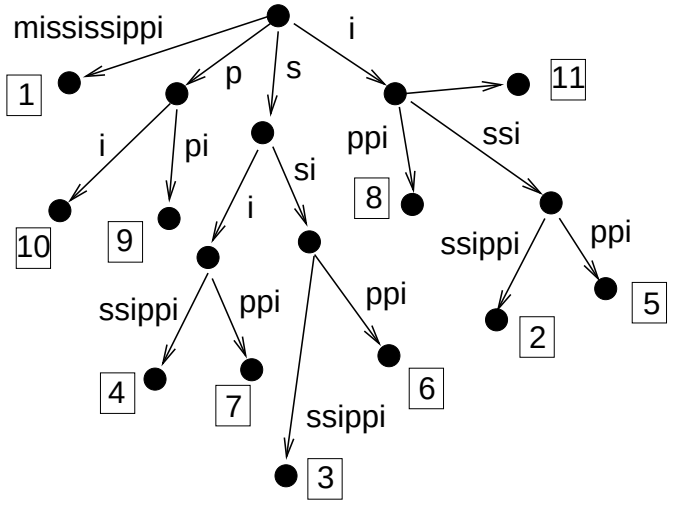

Figure 4: Example of suffix tree for "Mississippi".

The literature contains methods of both exact and inexact matching that can be implemented either with or without preprocessing of either the list or the incoming string. If these algorithms are combined with suffix string pre-processing of the `WURFL` database, then $C(m, n)$ can be reduced to $O(m)$, so that total search time is of order $O(k \times m)$.

3 Implementation

The second strand of the work by the study group was to implement the existing algorithms and see if some simple improvements could be made by judicious searching. Using data provided by the industry representatives, the algorithms were implemented in `Python`, and then some testing was done to see if making a few basic changes led to a significant improvement. Tests were performed on a sample of the database of only 40 listings, so times are representative only. However, there is no reason to think that the speed up should not scale up to the full set or even improve search times further.

3.1 Existing method

In this algorithm, a full search is made of the entire WURFL database looking for the best match using the Levenshtein algorithm [7], that is, compare everything, for

example as shown below, programmed in `Python`;

```
for i in xrange(len(database)):
   a = lev2(database[i], user)
   if a <= b:
               b = a
               j = i
return database[j]
```

The routine `lev2` performs the Levenshtein algorithm on the strings in the list. Several runs with trial data through the WURFL database gave average search times of 0.316 seconds and a worst case of 0.42 seconds. This method searches the unprocessed database to exhaustion.

3.2 "Simple" improvements

Several very simple suggested improvements were made to the above code. Again using the Levenshtein algorithm, a threshold minimum value (or proximity) was defined and rather than search through the whole database, the search stopped as soon as this threshold value was reached. Tuning this threshold parameter to get an adequate match would be required in practice. This simply means adding a line of code to that above which dictates an end if b is less than the threshold value. In that case,

- Running time in the "worst" case: **0.419**.
- Running time in the "best" case: **0.128**,

giving a $2\frac{1}{2}$ times speed up. This of course depends on how far down the list one must go before a suitable match is found, and also on how accurate the user wishes to be, or how large the threshold value is allowed to be. These results indicate a significant improvement with just this simple modification. The database was not pre-processed in any way.

Another approach tried was to subdivide the database into various categories. In other words, do some pre-processing of the `WURFL` database to create lists of particular common substrings, for example manufacturer, and then only search the relevant category. In other words, check the incoming string for a keyword and then only search the relevant section of the re-ordered database. For example, the database may be divided into S = ['Nokia', 'Samsung', 'Ludovic', 'Acer', ...]. If the incoming string contains one of these key strings, then the search is greatly reduced. Results of a trial of this idea gave

Algorithm	**Time Range**	**Comment**
Basic Brute Force	0.316-0.419	No Pre-processing
Threshold	0.128-0.419	No Pre-processing
Subdivision with threshold	0.074-0.286	Pre-Process into e.g. manufacturers
Caching	-	Extra coding - inadequate data
Order by Popularity	-	Promising - inadequate data
Suffix Trees	-	Huge potential - longer pre-processing

Table 1: Table summarizing methods considered by the group.

- Running time in the "worst" case: **0.286**,
- Running time in the "best" case: **0.074**,

a further reduction of 50% in search time. The idea of pre-processing the WURFL database appears to provide hope for great improvements in search time by restricting the number of strings that need to be compared, especially as the list grows.

A similar proposal is to order the WURFL database in frequency of request for each record over some recent time period. Those strings that appear most often would be placed near the top of the list so that they are reached first in the search. Using the threshold value idea suggested above this method should provide significant reduction in search time for the most common devices. However, the MISG group was not able to implement this scheme due to a lack of appropriate data. In a similar vein, the idea of buffering the most recent "hits" was discussed. This buffer could then be searched before the database was tested. As the database grows, older and obsolete machines will drop to the bottom of the pre-processed list. They will therefore have the longest search times, but it is also to be expected that they would have a smaller number of users.

3.3 Summary

Table 1 provides a summary of the simple implementations of modified search algorithms tried during the MISG. Some of the suggested improvements were not tested because we did not have suitable data.

The act of preprocessing the list into manufacturers and then ordering by recent frequency, combined with introducing the threshold value for Levenshtein distance has lead to a speed up of almost 5 times in the simple trials here. Only real testing can determine if these speed-ups are achievable.

4 Final Remarks

The study group performed a full search of the existing literature and discovered some new approaches that may prove beneficial (such as suffix strings) in the implementation of the string comparison itself. The current scheme was implemented and several suggested improvements were programmed and tested with positive results. Further suggestions were not tested because there was inadequate data available, but hold great promise for significant speed up in the service delivery. It seems the most likely approach for rapid speed up is to pre-process the `WURFL` database each time that it is uploaded to better organize the search. While this may take some time, it needs to be done only once to give very good reductions in search times.

References

[1] Boyer, R.S and Moore, J.S. "A fast string searching algorithm". *Comm. ACM* **20** (1977), 762772. *doi:*10.1145/359842.359859.

[2] Cormen, T.H., Leiserson, C.E., Rivest, R.L. and Stein, C. *Introduction to Algorithms* 2nd ed., MIT Press and McGraw-Hill, (2001) *ISBN 978-0-262-03293-3*, 923931.

[3] Damerau, F.J. "A technique for computer detection and correction of spelling errors", *Comm. ACM* **7** (1964), 171-176.

[4] Gusfield, D. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology* USA, Cambridge University Press (1999). ISBN 0-521-58519-8.

[5] Hamming. R.W. "Error detecting and error correcting codes", *Bell System Technical J.* **29** (2) (1950), 147160, MR0035935.

[6] Knuth D, Morris, D.J. Jr and Pratt, V. "Fast pattern matching in strings". *SIAM J. Computing* **6** (2) (1977), 323350. *doi:*10.1137/0206024.

[7] Levenshtein, V.I. "Binary codes capable of correcting deletions, insertions, and reversals" *Soviet Physics Doklady* **10** (1966), 70710.

[8] McCreight, E.M. "A Space-Economical Suffix Tree Construction Algorithm". *J. ACM* **23** (2) (1976), 262272. *doi:*10.1145/321941.321946.

[9] Ukkonen, E. "On-line construction of suffix trees". *Algorithmica* **14** (3) (1995), 249260. *doi:*10.1007/BF01206331

[10] Weiner, P. “Linear pattern matching algorithm”. 14th Annual IEEE Symposium on Switching and Automata Theory. (1973), pp. 111. *doi:*10.1109/SWAT.1973.13