Dynamic Programming Matching for Large Scale Information Retrieval

Eiko Yamamoto
Communications Research Laboratory, Kyoto Japan
eiko@crl.go.jp
Masahiro Kishida **Yoshinori Takenami**
Sumitomo Electric Information Systems Co., Ltd., Osaka Japan
{kishida-masahiro, takenami-yoshinori}@sei.co.jp
Yoshiyuki Takeda **Kyoji Umemura**
Toyohashi University of Technology, Aichi Japan
{take@ss.ics, umemura@tutics}.tut.ac.jp

Abstract

Though dynamic programming matching can carry out approximate string matching when there may be deletions or insertions in a document, its effectiveness and efficiency are usually too poor to use it for large-scale information retrieval. In this paper, we propose a method of dynamic programming matching for information retrieval. This method is as effective as a conventional information retrieval system, even though it is capable of approximate matching. It is also as efficient as a conventional system.
Keywords: Dynamic programming, Corpus-based, Japanese.

1 Introduction

The dynamic programming method is well-known for its ability to calculate the edit distance between strings. The method can also be applied to information retrieval. Dynamic programming matching can measure the similarity between documents, even if there are partial deletions or insertions. However, there are two problems in applying this method to information retrieval. One problem is search effectiveness. It is poor because dynamic programming matching lacks an adequate weighting schema. The second problem is computational efficiency. Also, lack of an adequate indexing schema means that dynamic programming matching usually has to process the entire document.

Yamamoto et al. proposed a method of dynamic programming matching with acceptable search effectiveness (Yamamoto et al., 2000; Yamamoto, Takeda, and Umemura, 2003). They report that the effectiveness of dynamic programming matching improves by introducing an IDF (Inverse Document Frequency) weighting schema for all strings that contribute similarity. They calculate matching weights not only for words but also for all strings. Although they report that effectiveness is improved, the speed of their method is slower than that of conventional dynamic programming matching, and much slower than that of a typical information retrieval system.

In this paper, we aim to improve the retrieval efficiency of the dynamic programming method while keeping its search effectiveness. From a mathematical point of view, we have only changed the definition of the weighting. The mathematical structure of similarity remains the same as that of the dynamic programming method proposed by (Yamamoto et al., 2000; Yamamoto, Takeda, and Umemura, 2003). Although it has the same definition, the new weighting method makes it possible to build a more efficient information retrieval system by creating the index in advance. To our surprise, we have observed that our proposed method is not only more efficient but also more effective.

2 Similarities Based on Dynamic Programming Matching

In this section, we introduce several similarities proposed by (Yamamoto et al., 2000; Yamamoto, Takeda, and Umemura, 2003). All of them are a

form of dynamic programming matching. These similarities include translation of the edit distance. This distance has been described by several authors. We have adopted *Korfhage's* definition: 'the edit distance is the minimum number of edit operations, such as insertion and deletion, which are required to map one string into the other' (Korfhage, 1997).

There are three related similarities. The first is dynamic programming matching, which is simply conversion of the edit distance. The second similarity is an extension of the first similarity, introducing a character weighting for each contributing character. The third and proposed similarity is an extension of the second one, using string weight instead of character weight.

2.1 Dynamic Programming Matching

As stated above, dynamic programming (DP) matching is a conversion of edit distance. We call this similarity SIM1. While the edit distance (ED) is a measure of difference, counting different characters between two strings , SIM1 is a measure of similarity, counting matching characters between two strings. ED and SIM1 are defined as follows:

Definition 2.1 *Edit Distance (Korfhage, 1997)*
Let α and β be strings, x and y be a character, and "" be empty string.

- If both strings are empty then
  $ED(\text{“”}, \text{“”}) = 0.0$
- If $x \neq y$ then
  $ED(x, y) = 1.0$
- If their first characters are the same then
  $ED(x\alpha, x\beta) =$
  $MIN(ED(\alpha, x\beta), ED(x\alpha, \beta),$
  $ED(\alpha, \beta) + 1.0)$
- Otherwise
  $ED(x\alpha, y\beta) =$
  $MIN(ED(\alpha, y\beta), ED(x\alpha, \beta),$
  $ED(\alpha, \beta))$

Definition 2.2 *SIM1*
Let α and β be strings, x and y be a character, and "" be empty string.

- If both strings are empty then
  $SIM1(\text{“”}, \text{“”}) = 0.0$
- If $x \neq y$ then
  $SIM1(x, y) = 0.0$
- If their first characters are the same then
  $SIM1(x\alpha, x\beta) =$
  $MAX(SIM1(\alpha, x\beta), SIM1(x\alpha, \beta),$
  $SIM1(\alpha, \beta) + 1.0)$
- Otherwise
  $SIM1(x\alpha, y\beta) =$
  $MAX(SIM1(\alpha, y\beta), SIM1(x\alpha, \beta),$
  $SIM1(\alpha, \beta))$

2.2 Character Weight DP Similarity

SIM1 adds 1.0 to the similarity between two strings for every matching character, and this value is constant for all the time. Our assumption for the new function is that different characters make different contributions. For example, in Japanese information retrieval, Hiragana characters are usually used for functional words and make a different contribution than Kanji characters, which are usually used for content words. Thus, it is natural to assign a different similarity weight according to the nature of the character. The below method of defining Character Weight DP Similarity adds not 1.0 but a specific weight depending on the matching character. We call this similarity SIM2. It resembles *Ukkonen's* Enhanced Dynamic Programming ASM (Approximate String Matching) (Berghel and Roach, 1996). The weight is expressed by a function called $Score$. SIM2 is defined as follows:

Definition 2.3 *SIM2*
Let α and β be strings, x and y be a character, and "" be empty string.

- If both strings are empty then
  $SIM2(\text{“”}, \text{“”}) = 0.0$
- If $x \neq y$ then
  $SIM2(x, y) = 0.0$
- If their first characters are the same then
  $SIM2(x\alpha, x\beta) =$
  $MAX(SIM2(\alpha, x\beta), SIM2(x\alpha, \beta),$
  $SIM2(\alpha, \beta) + Score(x))$
- Otherwise
  $SIM2(x\alpha, y\beta) =$
  $MAX(SIM2(\alpha, y\beta), SIM2(x\alpha, \beta),$
  $SIM2(\alpha, \beta))$

2.3 String Weight DP Similarity

DP procedure usually considers just a single character at a time, but since some long substrings can receive good scores, it is natural to consider all prefixes of the longest common prefix, not just the next character.

While SIM2 uses a character weight whenever a character matches between strings, a single character may not be enough. In some cases, even when each character has a low weight, the string as a whole may be a good clue for information retrieval. For example, "chirimenjyako" is a Japanese word that could be a retrieval key word. This word, which means "boiled and dried baby sardines," consists only of Hiragana characters "chi-ri-me-n-jya-ko" but each character would make a small contribution in SIM2.

The proposed similarity is called String Weight DP Similarity, which is a generalization of SIM2. We call this similarity SIM3. It considers the weight of all matching strings and is defined as follows:

Definition 2.4 *SIM3*

Let α and β be strings, x and y be a character, and "" be empty string.

- If both strings are empty then
  $SIM3(\text{“”}, \text{“”}) = Score(\text{“”}) = 0.0$
- Otherwise
  $SIM3(\alpha, \beta) =$
  $MAX(SIM3s(\alpha, \beta), SIM3g(\alpha, \beta))$
  - $SIM3s(\xi\alpha, \xi\beta) =$
    $MAX(Score(\gamma) + SIM3(\delta\alpha, \delta\beta))$
    where $\xi(= \gamma\delta)$ is the maximum length string matching from the first character.
  - $SIM3g(x\alpha, y\beta) =$
    $MAX(SIM3(\alpha, y\beta), SIM3(x\alpha, \beta),$
    $SIM3(\alpha, \beta))$

2.4 Weighting Function

Yamamoto et al. have used IDF (Inverse Document Frequency) as a weight for each string. The weight is computed using a *Score* function as follows:

Definition 2.5 *Yamamoto et al.'s $Score$ function*

Let ξ be string, $df(\xi)$ the frequency of documents including ξ in the document set for retrieval, and N be the number of documents in the set.

$Score(\xi) = IDF(\xi) = -log(df(\xi)/N)$

The standard one-character-at-a-time DP method assumes that long matches cannot receive exceptionally good scores. In other words, it regards $Score(\xi)$ as 0 if the length of ξ is greater than one. If the $Score$ function obeys the inequality, $Score(\delta\gamma) < Score(\delta) + Score(\gamma)$ for all substrings δ and γ, the best path would consist of a sequence of single characters, and we would not need to consider long phrases. However, we are proposing a different $Score$ function. It sometimes assigns good scores to long phrases, and therefore SIM2 has to be extended into SIM3 to establish a DP procedure that considers more than just one character at a time.

3 Proposed Weighting Function

Although SIM3, as shown in Section 2.3, has reasonable effectiveness, its computation is harder than that of the edit distance, and much harder than that of the similarity used in a conventional information retrieval system. In this paper, we have modified the weighting function so that it keeps its effectiveness while improving efficiency. To achieve this improvement, we use the SIM3 with the same definition but with a different score function.

3.1 Proposed String Weighting

We reduce the computational cost by limiting strings that have positive scores. First, we select *bigrams* as such strings. In other words, we assign a score of zero if the length of the string does not equal to 2. Several language systems use Kanji characters (e.g. Chinese and Japanese), and *bigram* is an effective indexing unit for information retrieval for these language systems (Ogawa and Matsuda, 1997). In addition, we may assume that the contribution of a longer string is approximated by the total *bigram* weighting. We have also restricted our attention to infrequent *bigrams*. Thus, we have restricted the weighting function *Score* as follows, where K is the number decided by the given query.

- If string length is 2 and $cf(\xi) < K$ then
  $Score(\xi) = -log(df(\xi)/N)$
- Otherwise $Score(") = 0.0$

3.2 Using a Suffix Array for Indexing

Since we have restricted the number of matching strings, and all the matching strings appear in

a query, we can collect all the positions of such strings. To make it possible, we need some indexing in advance. We have used a suffix array for this index. Below we summarize our proposed algorithm using a suffix array:

I. Make a suffix array of the document set.

II. For each query,
   A. Make a set of substrings consisting of two characters (*bigram*).
   B. For a given number n, extract the total n of less frequent *bigrams*, calculating corpus frequency.
   C. For each *bigram* from step B,
      i. Record all positions in which the *bigram* appears in the query and document set,
      ii. Record all documents that contain the *bigram*.
   D. For each document recorded,
      i. Compute the similarity between the query and the document with SIM3, using the recorded position of the corresponding *bigram*.
      ii. Assign the similarity to the document.
   E. Extract the most similar 1000 documents from the recorded documents as a retrieval result for the query.

We call the retrieval method described above Fast Dynamic Programming (FDP). In general, retrieval systems use indexes to find documents. FDP also uses an index as a usual method. However, unlike conventional methods, FDP requires information not only on the document identification but also on the position of *bigrams*.

Manber and Myers proposed a data structure called "suffix array." (Manber and Myers, 1993) Figure 1 shows an example of suffix array. Each suffix is expressed by one integer corresponding to its position. We use this suffix array to find out the position of selected *bigrams*. A suffix array can be created in $O(N \log(N))$ time because we need to sort all suffixes in alphabetical order. We can get the position of any string in $O(\log(N))$ time by a binary search of suffixes and by then obtaining its corresponding position.

4 Experiment

In the experiment, we compared the proposed FDP method with SIM1, SIM2, and SIM3, which were described in Section 2. We measured three values:

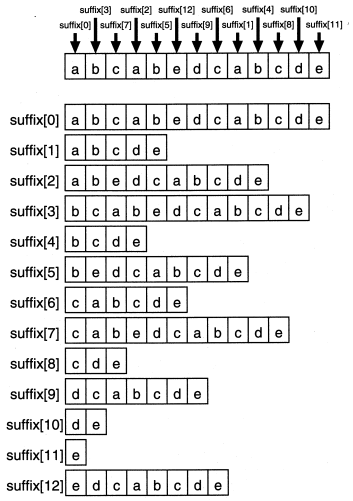

Figure 1: Suffix Array

search effectiveness, memory usage, and execution time.

We used the NTCIR1 collection (NTCIR Project, 1999). This collection consists of 83 retrieval topics and roughly 330,000 documents of Japanese technical abstracts. The 83 topics include 30 training topics (topic01-30); the rest are for testing (topic31-83). The testing topics were more difficult than the training topics. Each topic contains five parts, "TITLE", "DESCRIPTION", "NARRATIVE", "CONCEPT", and "FIELD." We retrieved using "DESCRIPTION," which is retrieval query and a short sentence.

All the experiments reported in this section were conducted using a dual AMD Athlon MP 1900+ with 3GB of physical memory, running TurboLinux 7.0.

4.1 Search Effectiveness

The proposed FDP method restricts the number of *bigrams* that can contribute to string matching. That is, only a small number of strings are considered. It was not clear whether FDP maintains its effectiveness like SIM3. To verify it, we compared the effectiveness of FDP with that of SIM1, SIM2, and SIM3. We also needed to know how the effectiveness might vary by the number of *bigrams*. We set number n at 5, 10, 15, 20, 30, 50, and 500. They were named FDP5, FDP10, FDP15, FDP20, FDP30, FDP50, and FDP500, respectively.

Table 1: Statistical Significant Test for difference of MAP ($\alpha = 0.005, \nu = 83 - 1$)

	SIM2	SIM3	FDP5	FDP10	FDP15	FDP20	FDP30	FDP50	FDP500
SIM1	$<<$	$<<$	$<<$	$<<$	$<<$	$<<$	$<<$	$<<$	$<<$
SIM2		$<<$	$=$	$<$	$<<$	$<<$	$<<$	$<<$	$<<$
SIM3			$=$	$=$	$<$	$<<$	$<<$	$<<$	$<<$
FDP5				$=$	$<<$	$<<$	$<<$	$<<$	$<<$
FDP10					$=$	$<<$	$<$	$<$	$<$
FDP15						$<$	$=$	$=$	$=$
FDP20							$=$	$=$	$=$
FDP30								$=$	$=$
FDP50									$=$

Table 2: Search Effectiveness for Topic01-30

Method	11 pt. average	R-precision
SIM1	0.1349	0.1790
SIM2	0.1948	0.2296
SIM3	0.2691	0.3024
FDP5	0.2547	0.2649
FDP10	0.2948	0.3089
FDP15	0.3109	0.3446
FDP20	0.3207	0.3574
FDP30	0.3176	0.3421
FDP50	0.3131	0.3377
FDP500	0.3172	0.3419

Table 3: Search Effectiveness for Topic31-83

Method	11 pt. average	R-precision
SIM1	0.0545	0.0845
SIM2	0.1245	0.1596
SIM3	0.1807	0.2083
FDP5	0.1277	0.1505
FDP10	0.1766	0.2013
FDP15	0.2144	0.2280
FDP20	0.2398	0.2621
FDP30	0.2353	0.2485
FDP50	0.2354	0.2488
FDP500	0.2350	0.2477

The NTCIR1 collection also contains a relevance judgment. We obtained the 11-point average precision and R-precision using standard tools called TRECEVAL. And we tested about statistical significance for difference of MAP (Mean Average Precision) (Kishida et al., 2002).

Tables 2 and 3 show the search effectiveness for all methods. We found that FDP20 is the most effective. Table 1 shows the results of one-sided t-test for difference of MAP $\bar{x_i} - \bar{y_i}$, where $\bar{x_i}$ and $\bar{y_i}$ are MAP of i-th method in the first row and MAP of i-th method in the first column, respectively. The level of significance α is 0.005 and the degree of freedom ν is $83 - 1$. The Symbols $<<, <, =$ represent "much less than α", "less than α, and "not less than α", respectively. We found that except for FDP5 and FDP10, the other FDPs are significantly more effective than SIM3 at a level of significance 0.005. In additional, this shows that FDP30, FDP50, and FDP500 are not significantly more effective than FDP20. These have demonstrated our proposed FDP method maintains its effectiveness, even though the strings that contribute similarity are restricted to a small number of *bigrams*. Also, it is interesting that the FDP with 20 *bigrams* is significantly more effective than the one with many more *bigrams*.

4.2 Memory Usage

The proposed method needs to record all the positions considered *bigrams*. A memory area is therefore required to hold position information; in the worst case, the memory size required is the product of the number of documents and the number of substrings in a query. This means the memory requirement could be very large. However, using FDP, we have found that the amount of memory requested is of a reasonable size.

In other words, the size of the memory area is the total sum of collection frequency for all strings that contribute similarity. We examined the amount of memory used by comparison for the total sum of collection frequency.

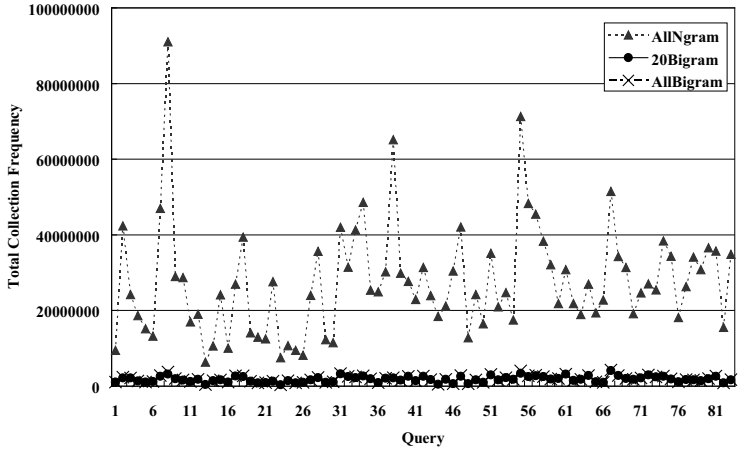

Figure 2: Memory Usage (Total Number of Collection Frequency for Each String)

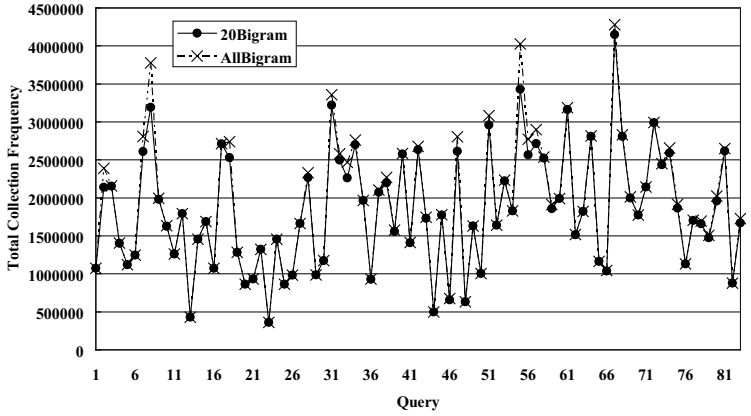

Figure 3: Memory Usage for Different Number of Restricted Bigrams

Figure 2 shows the total sum of collection frequency for three kinds of string sets. In the figure, AllNgram is for sets of all substrings considered by SIM3, AllBigram is for sets of all *bigrams*, and 20Bigram is for sets of 20 *bigrams* considered by FDP20. The field surrounded by the plot line and the horizontal axis represents the total sum of collection frequency. As the figure shows, AllBigram and 20Bigram occupy a much smaller field than AllNgram. This means the memory requirement of FDP is much smaller than that of SIM3. This result shows that FDP is possible to efficiently perform large-scale information retrieval on a computer with a reasonable amount of memory.

Figure 3 shows enlarged graphs of AllBigram and 20Bigram from Figure 2. The figure shows that 20Bigram equals AllBigram for most queries, but not always. However, as shown in Table 2 and Table 3, FDP20 actually has the highest precision in all FDPs. This means that considering more *bigrams* is not necessarily an advantage. Probably, by choosing substrings with a high contribution, we manage to get rid of noisy strings.

4.3 Execution Time

We measured execution time under the same conditions as described in Section 4.1. Notice we implemented SIM1, SIM2, and SIM3 in C language. On the other hand, FDP is implemented in Java (JDK1.3.1.04). When we noted the time required

to make a suffix array, we found that FDP took 1.5 times as long as SIM in Figure 4. Thus, for the same algorithm, the execution speed of Java is generally slower than that of C language.

Figures 5 and 6 show the time taken to retrieve for each topic01-30 and topic31-83. In the figures, the vertical axis is the number of documents, and the horizontal axis is the execution time. We found that all SIMs took much longer than FDPs. This demonstrates that our algorithm in Section 3 sharply improves execution speed. Moreover, we found that execution time did not increase exponentially even if the candidate documents for retrieval increased; instead, the retrieval collection becomes larger and larger. This suggests that FDP is an effective DP technique for large-scale information retrieval.

5 Related Work

Our proposed technique is a type of DP matching. The most typical application of DP matching is gene information research, because DP is effective for gene information matching. However, this system has a very slow processing speed.

In recent years, advances in this field of research have meant that high-speed systems have been required for gene information retrieval. A high-speed gene information retrieval system called BLAST was developed (Setubal and Meidanis, 2001). BLAST has achieved higher processing speed by using heuristics that specify characteristic gene arrangements, rather than using DP matching. In contrast, we have managed to achieve fast matching using the DP technique.

Moreover, in music information retrieval, an error in copying a tune corresponds to a deficit (deletion) and insertion of data. For this reason, a music search engine has been built based on the DP technique (Hu and Dannenberg, 2002). Since there is a great deal of music information available these days, scalability is also an important problem for music information retrieval systems. Our proposed DP method is scalable and can cope with deficits. It therefore has potential applications in music information retrieval.

6 Conclusion

In this study, we proposed a DP matching method for large-scale information retrieval. To improve its efficiency, this method selects the strings that contribute more to retrieval. This selection process reduces the memory requirement and frequency of memory access. We conclude that our method is suitable for large-scale information retrieval where approximate matching is required.

Acknowledgement

This work was supported in The 21st Century COE Program "Intelligent Human Sensing," from the Ministry of Education, Culture, Sports, Science, and Technology.

References

Hal Berghel and David Roach. 1996. An extension of Ukkonen's enhanced dynamic programming ASM algorithm. *Journal of ACM TOIS*, 4(1):94–106.

Kazuaki Kishida, Makoto Iwayama, and Koji Eguchi. 2002. Methodology and Pragmatics of Retrieval Experiments at NTCIR Workshop. Pre-meeting Lecture at the NTCIR-3 Workshop.

Ning Hu and Roger B. Dannenberg. 2002. Comparison of Melodic Database Retrieval Techniques Using Sung Queries. *Proceedings of JCDL 2002*, 301–307.

Robert R. Korfhage. 1997. Information Storage and Retrieval. *WILEY COMPUTER PUBLISHING*, 291—303.

Udi Manber and Gene Myers. 1993. Suffix arrays: a new method for on-line string searches. *SIAM Journal of Computing*, 22(5):935–948.

NTCIR Project. http://research.nii.ac.jp/ntcir/.

Yasushi Ogawa and Toru Matsuda. 1997. Overlapping statistical word indexing: A new indexing method for Japanese text. *Proceedings of SIGIR97*, 226–234.

Joao Carlos Setubal and Joao Meidanis. 2001. *Introduction to Computational Molecular Biology*. BrooksCole Publishing Company.

Eiko Yamamoto, Mikio Yamamoto, Kyoji Umemura, and Kenneth W. Church. 2000. Dynamic Prgramming: A Method for Taking Advantage of Technical Terminology in Japanese Documents. *Proceedings of IRAL2000*, 125–131.

Eiko Yamamoto, Yoshiyuki Takeda, and Kyoji Umemura. 2003. An IR Similarity Measure which is Tolerant for Morphological Variation. *Journal of Natural Language Processing*, 10(1):63–80. (in Japanese).

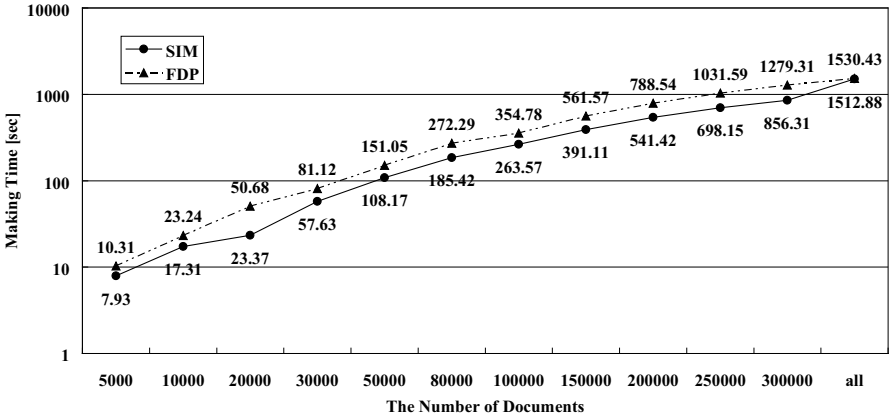

Figure 4: Suffix Array Generation Time

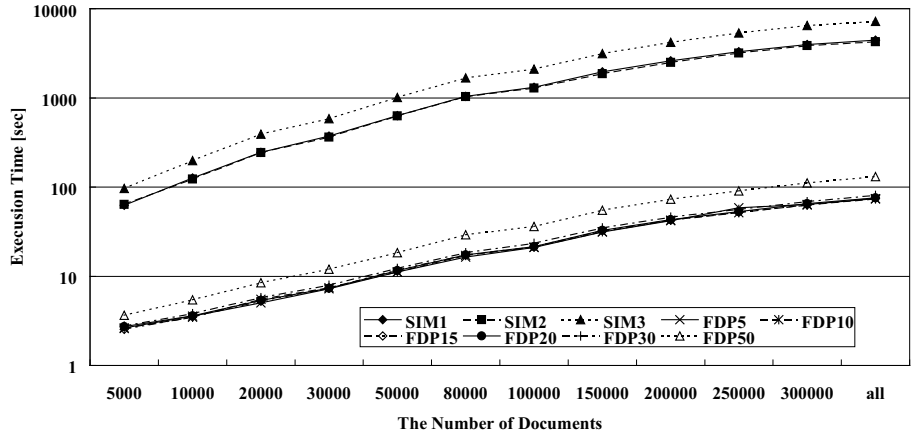

Figure 5: Execution Time for Topic01-30

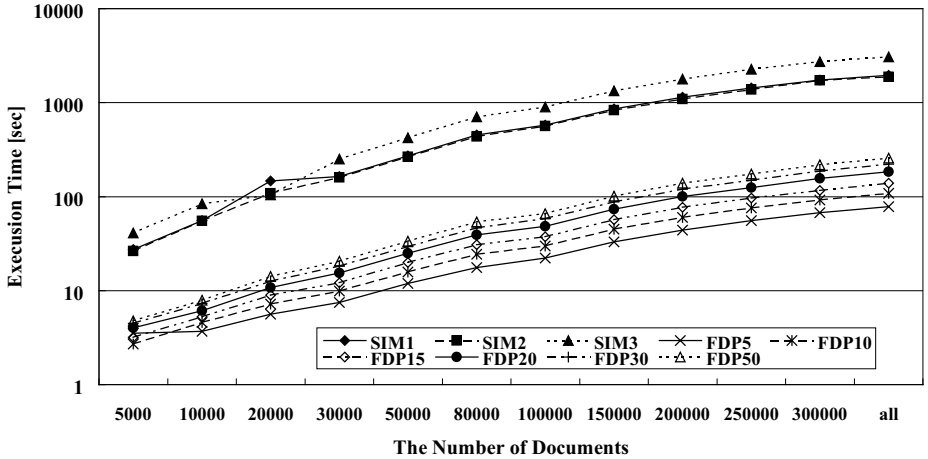

Figure 6: Execution Time for Topic31-83