An Efficient Language Model Using Double-Array Structures

Makoto Yasuhara **Toru Tanaka** †**Jun-ya Norimatsu** **Mikio Yamamoto**
Department of Computer Science
University of Tsukuba, Japan
†norimatsu@mibel.cs.tsukuba.ac.jp

Abstract

Ngram language models tend to increase in size with inflating the corpus size, and consume considerable resources. In this paper, we propose an efficient method for implementing $n$gram models based on double-array structures. First, we propose a method for representing backwards suffix trees using double-array structures and demonstrate its efficiency. Next, we propose two optimization methods for improving the efficiency of data representation in the double-array structures. Embedding probabilities into unused spaces in double-array structures reduces the model size. Moreover, tuning the word IDs in the language model makes the model smaller and faster. We also show that our method can be used for building large language models using the division method. Lastly, we show that our method outperforms methods based on recent related works from the viewpoints of model size and query speed when both optimization methods are used.

1 Introduction

Ngram language models (F. Jelinek, 1990) are widely used as probabilistic models of sentence in natural language processing. The wide use of the Internet has entailed a dramatic increase in size of the available corpora, which can be harnessed to obtain a significant improvement in model quality. In particular, Brants et al. (2007) have shown that the performance of statistical machine translation systems is monotonically improved with the increasing size of training corpora for the language model. However, models using larger corpora also consume more resources. In recent years, many methods for improving the efficiency of language models have been proposed to tackle this problem (Pauls and Klein, 2011; Kenneth Heafield, 2011). Such methods not only reduce the required memory size but also raise query speed.

In this paper, we propose the double-array language model (DALM) which uses double-array structures (Aoe, 1989). Double-array structures are widely used in text processing, especially for Japanese. They are known to provide a compact representation of tries (Fredkin, 1960) and fast transitions between trie nodes. The ability to store and manipulate tries efficiently is expected to increase the performance of language models (i.e., improving query speed and reducing the model size in terms of memory) because tries are one of the most common representations of data structures in language models. We use double-array structures to implement a language model since we can utilize their speed and compactness when querying the model about an $n$gram.

In order to utilize of double-array structures as language models, we modify them to be able to store probabilities and backoff weights. We also propose two optimization methods: `embedding` and `ordering`. These methods reduce model size and increase query speed. `Embedding` is an efficient method for storing $n$gram probabilities and backoff weights, whereby we find vacant spaces in the double-array language model structure and populate them with language model information, such as probabilities and backoff weights. `Ordering` is

Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing, pages 222–232,
Seattle, Washington, USA, 18-21 October 2013. ©2013 Association for Computational Linguistics

a method for compacting the double-array structure. DALM uses word IDs for all words of the $n$gram, and `ordering` assigns a word ID to each word to reduce the model size. These two optimization methods can be used simultaneously and are also expected to work well.

In our experiments, we use a language model based on corpora of the NTCIR patent retrieval task (Atsushi Fujii et al., 2007; Atsushi Fujii et al., 2005; Atsushi Fujii et al., 2004; Makoto Iwayama et al., 2003). The model size is 31 GB in the ARPA file format. We conducted experiments focusing on query speed and model size. The results indicate that when the abovementioned optimization methods are used together, DALM outperforms state-of-the-art methods on those points.

2 Related Work

2.1 Tries and Backwards Suffix Trees

Tries (Fredkin, 1960) are one of the most widely used tree structures in $n$gram language models since they can reduce memory requirements by sharing common prefix. Moreover, since the query speed for tries depends only on the number of input words, the query speed remains constant even if the $n$gram model increases in size.

Backwards suffix trees (Bell et al., 1990; Stolcke, 2002; Germann et al., 2009) are among the most efficient representations of tries for language models. They contain $n$grams in reverse order of history words.

Figure 1 shows an example of a backwards suffix tree representation. In this paper, we denote an $n$gram: by the form $w_1, w_2, \cdots, w_n$ as $\boldsymbol{w}_1^n$. In this example, word lists (represented as rectangular tables) contain target words (here, w_n) of $n$grams, and circled words in the tree denote history words (here, $\boldsymbol{w}_1^{n-1}$) associated with target words. The history words "I eat," "you eat", and "do you eat" are stored in reverse order. Querying this trie about an $n$gram is simple: just trace history words in reverse and then find the target word in a list. For example, consider querying about the trigram "I eat fish". First, simply trace the history in the trie in reverse order ("eat" → "I"); then, find "fish" in list <1>. Similarly, querying a backwards suffix tree about unknown $n$grams is also efficient, because the backwards suffix tree

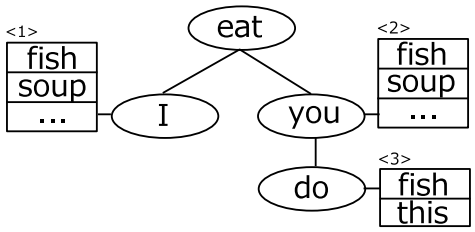

Figure 1: Example of a backwards suffix tree. There are two branch types in a backwards suffix tree: history words and target words. History words are shown in circles and target words are stored in word lists.

representation is highly suitable for the backoff calculation. For example, in querying about the 4gram "do you eat soup", we first trace "eat" → "you" → "do" in a manner similar to above. However, searching for the word "soup" in list <3> fails because list <3> does not contain the word "soup". In this case, we return to the node "you" to search the list <2>, where we find "soup". This means that the trigram "you eat soup" is contained in the tree while the 4gram "do you eat soup" is not. This behavior can be efficiently used for backoff calculation.

SRILM (Stolcke, 2002) is a widely used language model toolkit. It utilizes backwards suffix trees for its data structures. In SRILM, tries are implemented as 64-bit pointer links, which wastes a lot of memory. On the other hand, the access speed for $n$gram probabilities is relatively high.

2.2 Efficient Language Models

In recent years, several methods have been proposed for storing language models efficiently in memory.

Talbot and Osborne (2007) have proposed an efficient method based on bloom filters. This method modifies bloom filters to store count information about training sets. In prior work, bloom filters have been used for checking whether certain data are contained in a set. To store the count information, pairs from <$n$gram,1> to <$n$gram,count> are all added to the set for each $n$gram. To query this language model about the probability of an $n$gram, probabilities are calculated during querying by using these counts. Talbot and Brants (2008) have proposed a method based on perfect hash functions and bloomier filters. This method uses perfect hash functions to store $n$grams and encode values (for exam-

ple, probabilities or counts of $n$grams in the training corpus) to a large array. Guthrie and Hepple (2010) have proposed a language model called ShefLM that uses minimal perfect hash functions (Belazzougui et al., 2009), which can store $n$grams without vacant spaces. Furthermore, values are compressed by *simple dense coding* (Fredriksson and Nikitin, 2007). ShefLM achieves a high compression ratio when it stores counts of $n$grams in the training corpus. However, when this method stores probabilities of $n$grams, the advantage of using compression is limited because floating-point numbers are difficult to compress. Generally, compression is performed by combining the same values but, two floating-point numbers are rarely the same, especially in the case of probability values[1]. These methods implement lossy language models, meaning that, we can reduce the model size at the expense of model quality. These methods also reduce the model performance (perplexity).

Pauls and Klein (2011) have proposed BerkeleyLM which is based on an *implicit encoding* structure, where $n$grams are separated according to their order, and are sorted by word ID. The sorted $n$grams are linked to each other like a trie structure. BerkeleyLM provides rather efficient methods. Variable-length coding and block compression are used if small model size is more important than query speed. In addition, Heafield (2011) has proposed an efficient language model toolkit called KenLM that has been recently used in machine translation systems, for which large language models are often needed. KenLM has two different main structure types: `trie` and `probing`. The `trie` structure is compact but relatively slower to query, whereas the `probing` structure is relatively larger but faster to query.

In this paper, we propose a language model structure based on double-array structures. As we describe in Section 3, double-array structures can be used as fast and compact representations of tries. We propose several techniques for maximizing the performance of double-array structures from the perspective of query speed and model size.

[1]In our experience, it is considerably easier to compress backoff weights than to compress probabilities, although both are represented with floating-point numbers. We use this knowledge in our methods.

3 Double-Array

3.1 Double-Array Structure

In DALM, we use a double-array structure (Aoe, 1989) to represent the trie of a language model. Double-array structures are trie representations consisting of two parallel arrays: $BASE$ and $CHECK$. They are not only fast to query, but also provide a compact way to store tries. In the structure, nodes in the trie are represented by slots with the same index in both arrays. Before proposing several efficient language model representation techniques in Section 4, we introduce double-array themselves. In addition, the construction algorithms for double-arrays are described in Section 3.2 and Section 3.3.

The most naive implementation of a trie will have a two-dimensional array $NEXT$. Let WORDID (w) be a function that returns a word ID as a number corresponding to its argument word w; then $NEXT[n]$[WORDID (w)] (that presents the $WORDID(w)$-th slot of the nth row in the $NEXT$ array) stores the node number which can be transit from the node number n by the word w, and we can traverse the trie efficiently and easily to serialize the array in memory. This idea is simple but wastes the most of the used memory because almost all of the slots are unused and this results in occupying memory space. The double-array structures solve this problem by taking advantage of the sparseness of the $NEXT$ array. The two-dimensional array $NEXT$ is merged into a one-dimensional array $BASE$ by shifting the entries of each row of the $NEXT$ array and combining the set of resulting arrays. We can store this result in much less memory than the serialization of the naive implementation above. Additionally, a $CHECK$ array is introduced to check whether the transition is valid or not because we cannot distinguish which node the information in a particular slot comes from. Using a $CHECK$ array, we can avoid transition errors and move safely to the child node of any chosen node.

As a definition, a node link from a node n_s with a word w to the next node n_{next}in the trie is defined as follows:

$$\begin{aligned} next \quad &\leftarrow \quad BASE[s] + \text{WORDID}\,(w) \\ &if \quad CHECK[next] == s \end{aligned}$$

where s denotes the index of the slot in the double-

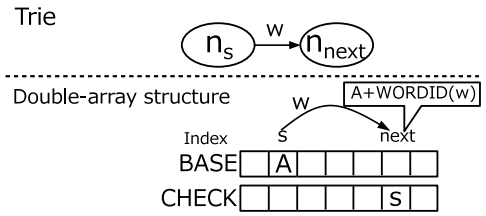

Figure 2: A trie and a corresponding double-array structure. Node n_s is represented by the slots $BASE[s]$ and $CHECK[s]$. A link from a node n_s with a word w is indicated by $CHECK[next] == s$.

array structure which represents n_s. The trie transition from a node n_s with a word w is applied according to the following steps:

Step 1 Calculating the "$next$" destination and

Step 2 Checking whether the transition is correct.

Step 2 specifically means the following:

1. If $CHECK[next] == s$, then we can "move" to the node n_{next};

2. otherwise, we can detect that the transition from the node n_s with the word w is not contained in the trie.

Figure 2 shows an example of a transition from a parent node n_s with a word w.

Next, we describe how the existence of an $n$gram history can be determined (Aoe, 1989). We can iterate over the nodes by the transitions shown above and may find the node representing an $n$gram history. But we should check that it is valid because nodes except for leaf nodes possiblly represent a fragment of some total $n$gram history. We can use *endmarker* symbols to determine whether an $n$gram history is in the trie. We add nodes meaning the *endmarker* symbol after the last node of each $n$gram history. When querying about $\boldsymbol{w}_1^{n-1}$, we transit repeatedly; in other words, we set $s = 0$ and start by applying Step 1 and 2 repeatedly for each word. When we reach the node w_{n-1}, we continue searching for an *endmarker* symbol. If the symbol is found, we know that the $n$gram history $\boldsymbol{w}_1^{n-1}$ is in the trie.

The double-array structure consumes 8 bytes per node because the $BASE$ and $CHECK$ arrays are 4 byte array variables. Therefore, the structure can store nodes compactly in case of a high filling rate. Moreover, node transitions are very fast because they require only one addition and one comparison per transition. We use a double-array structure in DALM, which can maximize its potential.

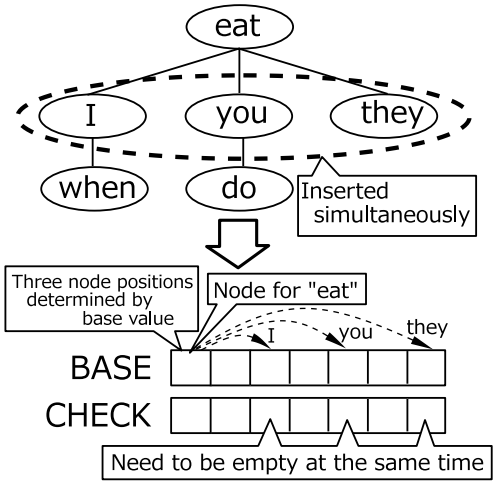

Figure 3: Greedy insertion of trie elements. The children of a node are collectively inserted into the double-array when the $BASE$ value of the node is fixed.

3.2 Greedy Construction

Greedy algorithms are widely used for constructing static double-array structures[2]. The construction steps are as follows:

1. Define the root node of a trie to correspond to index 0 of the double-array structure and

2. Find the $BASE$ value greedily (i.e., in order $1, 2, 3, \cdots$) for all nodes which have fixed their indices in the double-array structure.

In practice, once the $BASE$ value of a node is fixed, the positions of its children are fixed at the same time, and we can find the $BASE$ values for each child recursively.

Figure 3 shows an example of such construction. In this example, three nodes ("I", "you" and "they") are inserted at the same time. This is because the above three node positions are fixed by the $BASE$ value of the node "eat". To insert nodes

[2]We were unable to find an original source for this technique. However, this method is commonly used in double-array implementations.

"I", "you" and "they", the following three slots must be empty (i.e., the slots must not be used by other nodes.):

- $BASE[s]$ + WORDID("I")
- $BASE[s]$ + WORDID("you")
- $BASE[s]$ + WORDID("they")

where s is the index of the node "eat". At the construction step, we need to find $BASE[s]$ which satisfies the above conditions.

3.3 Efficient Construction Algorithm

The construction time for a double-array structure poses the greatest challenge. We use a more efficient method (Nakamura and Mochizuki, 2006) instead of the naive method for constructing a double-array structure because the naive method requires a long time. We call the method "empty doubly-linked list". This algorithm is one of the most efficient construction methods devised to date. Figure 4 shows an example of an empty doubly-linked list. We can efficiently define the $BASE$ value of each node by using the $CHECK$ array to store the next empty slot. In this example, in searching the $BASE$ value of a node, the first child node can be set to position 1, and if that fails, we can successively try positions $3, 4, 6, 8, \cdots$ by tracing the list instead of searching all $BASE$ values $0, 1, 2, 3, 4, 5, \cdots$.

As analyzed by Nakamura and Mochizuki(2006), the computational cost of a node insertion is less than in the naive method. The original naive method requires $O(NM)$ time for a node insertion, where M is a number of unique word types and N is a number of nodes of the trie; the algorithm using an empty double-linked list requires $O(UM)$, where U is the number of unused slots.

As described in Section 5, we divide the trie into several smaller tries and apply the efficient method for constructing our largest models. This is because it is not feasible to wait several weeks for the large language model structure to be built. The dividing method is currently the only method allowing us to build them.

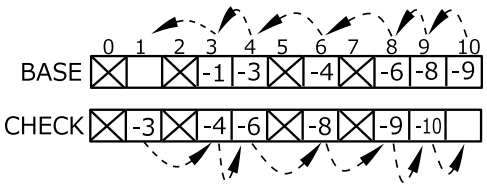

Figure 4: Empty doubly-linked list. Unused $CHECK$ slots are used to indicate the next unused slots, and unused $BASE$ slots are used to indicate previous unused slots. Thus, the $BASE$ and $CHECK$ arrays are used as a doubly-linked list which can reduce the number of ineffective trials.

4 Proposed Methods

4.1 DALM

In this section, we present the application of the double-array structure to backwards suffix trees. As this is the most basic structure based on double-array structures, we refer to it as the `simple` structure and improve its performance as described in the following sections.

To represent a backwards suffix tree as a double-array structure, we should modify the tree because it has two types of branches (target words and history nodes), which must be distinguished in the double-array structure. Instead, we should distinguish the branch type which indicates whether the node is a target word or a history word. We use the *endmarker* symbol (<#>) for branch discrimination. In prior work, the *endmarker* symbol has been used to indicate whether an $n$gram is in the trie. However, there is no need to distinguish whether the node of the tree is included in the language model because all nodes of a backwards suffix tree which represents $n$grams surely exist in the model. We use the *endmarker* symbol to indicate nodes which are end-of-history words. Therefore, target words of $n$grams are children of the *endmarker* symbols that they follow.

By using the *endmarker* symbol, target words can be treated the same as ordinary nodes because all target words are positioned after <#>. Figure 5 shows an example of such construction. We can clearly distinguish target words and history words in the backwards suffix tree.

Querying in the tree is rather simple. For example, consider the case of a query trigram "I eat fish" in the trie of Figure 5. We can trace this trigram in

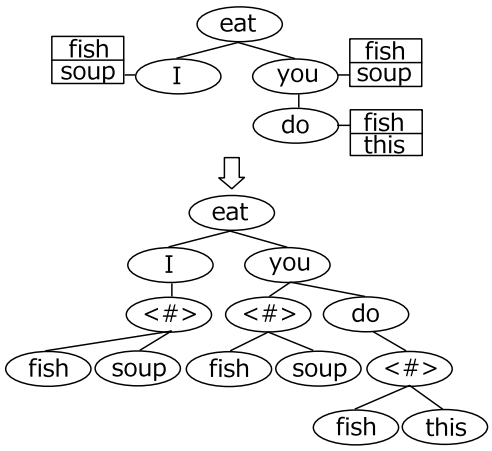

Figure 5: An example of converting a backwards suffix tree. We introduce *endmarker* symbols to distinguish the two branch types. We can treat the tree as an ordinary trie that can be represented by a double-array structure while retaining the advantages of the tree structure.

the same way as the original backwards suffix tree. First, we trace "eat" → "I", then trace that to the *endmarker* symbol <#> and finally find the word "fish".

Next, we describe the procedure for storing probabilities and backoff weights. We prepare a $VALUE$ array to store the probabilities and backoff weights of $n$grams. Figure 6 shows the `simple` DALM structure. The backwards suffix tree stores a backoff weight for each node and a probability for each target word. In `simple` DALM, each value is stored for the respective position of the corresponding node.

4.2 `Embedding`

`Embedding` is a method for reducing model size. In the `simple` DALM structure, there are many vacant spaces in the $BASE$ and $CHECK$ arrays. We use these vacant spaces to store backoff weights and probabilities. Figure 7 shows vacant spaces in the `simple` DALM structure.

First, the $BASE$ array slots of target word nodes are unused because target words are always in leaf positions in the backwards suffix tree and do not have any children nodes. In the example of Figure 7, $BASE[9]$ is not used, and therefore can be used for storing a probability value. This method can reduce the model size because all probabilities are stored into the $BASE$ array. As a result, the $VALUE$ array

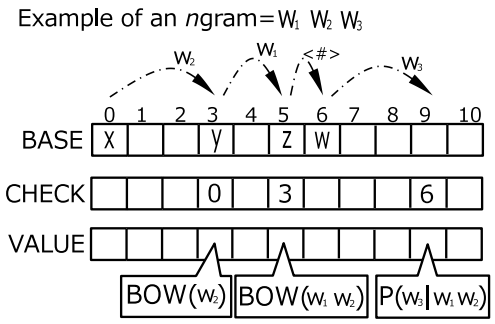

Figure 6: The `simple` DALM data structure. The $BASE$ and $CHECK$ arrays are used in the same way as in a double-array structure. To return probabilities and backoff weights, a $VALUE$ array is introduced.

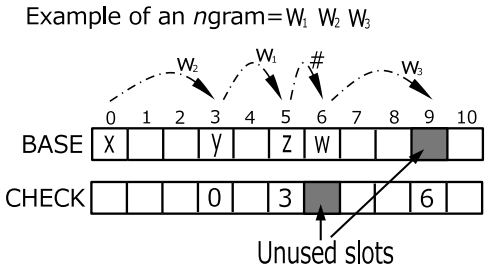

Figure 7: Unused slots in the `simple` DALM structure used for other types of information, such as probabilities.

contains only backoff weights.

Next, the $CHECK$ array slots of *endmarker* symbols are also vacant. We do not need to check for *endmarker* symbol transition because the *endmarker* symbol <#> is seen for all nodes except target word nodes. This means that all *endmarker* symbol transitions are ensured to be correct and the $CHECK$ array slots of *endmarker* symbols do not need to be used. We use this space to store backoff weights.

In order to avoid false positives, we cannot store backoff weights directly. Instead, we store the positions of the backoff weights in the $VALUE$ array as negative numbers. When a query for an unknown $n$gram encounters an *endmarker* symbol node, the value of the $CHECK$ array is never matched because the corresponding value stored there is negative. The same values in the $VALUE$ array can be unified to reduce the memory requirements. Figure 8 illustrates an example of the `embedding` method.

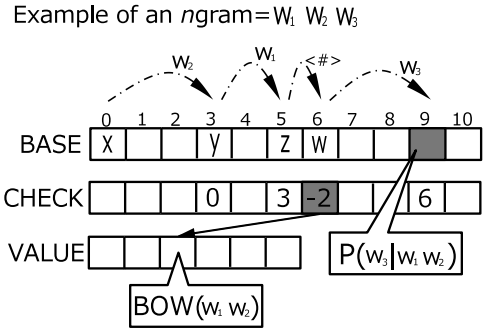

Figure 8: Implementation of the `embedding` method. We use vacant spaces in the $VALUE$ array to store the probabilities and indices of backoff weights. The indices of backoff weights are taken with a negative sign to avoid false positives. Backoff weights are stored in the $VALUE$ array, and the same values in the $VALUE$ array can be unified.

4.3 `Ordering`

`Ordering` is a method for shortening the double-array structure and increasing the query speed. In `ordering`, word IDs are assigned in order of unigram probability. This is done at a preprocessing stage, before the DALM is built.

Before explaining the reasons why this method is effective, we present an *interpretation* of double-array construction in Figure 9 which corresponds to the case presented in Figure 3. In the previous section, we pointed out that the insertion problem is equivalent to the problem of finding the $BASE$ value of the parent node. Here, we expand this further into the idea that finding the $BASE$ value is equivalent to the problem of finding the shift length of an *insertion array*. We can create an *insertion array* which is an array of flag bits set to 1 at the positions of word IDs of children nodes' words. Moreover, we prepare a *used array* which is also a flag bit array denoting whether the original slots in the double-array structure are occupied. In this situation, finding the shift length is equivalent to the problem of finding the $BASE$ value of the slot for the node "eat", and the combined *used array* denotes the size of the double-array structure after insertion.

Figure 10 shows an intuitive example illustrating the efficiency of the `ordering` method. When word IDs are assigned in order of unigram probability, 1s in the *insertion array* are gathered toward

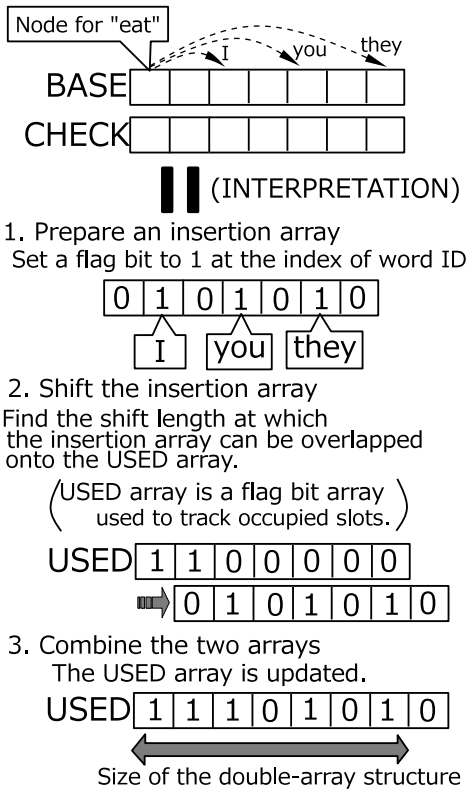

Figure 9: Interpretation of a double-array construction. The insertion problem for the double-array structure is interpreted as a finding problem of a shift length of the insertion array. We can measure the size of the double-array structure in the *used array*.

the beginning of the array. This means that 1s in the *insertion array* form clusters, which makes insertion easier than for unordered *insertion arrays*. This shortens the shift lengths for each insertion array: a shorter double-array structure results.

5 Experiment

5.1 Experimental Setup

To compare the performance of DALM with other methods, we conduct experiments on two $n$gram models built from small and large training corpora. Table 1 shows the specifications of the model.

Training data are extracted from the *Publication of unexamined Japanese patent applications*, which is distributed with the NTCIR 3,4,5,6 patent retrieval task (Atsushi Fujii et al., 2007; Atsushi Fujii et al., 2005; Atsushi Fujii et al., 2004; Makoto Iwayama et al., 2003). We used data for the period from

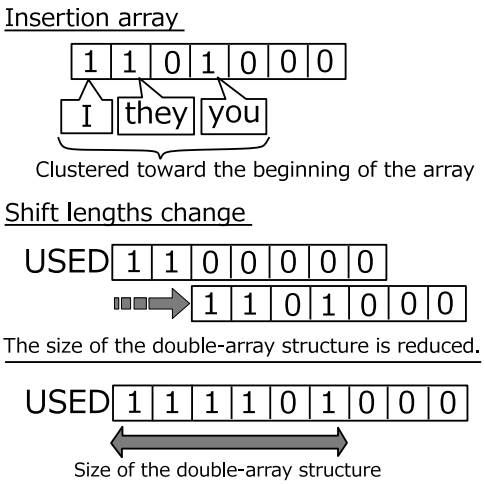

Figure 10: An example of word ID `ordering` efficiency. Word IDs in the insertion array are packed to the front in advance. Therefore, shift lengths for ordered arrays are often shorter than unordered ones. The resulting size of the double-array structure is expected to be smaller than that of an unordered array.

Table 1: Corpus and model specifications.

Model	Corpus Size (words)	Unique Type (words)	Ngram Type (1-5gram)
100 Mwords	100 M	195 K	31 M
5 Gwords	5 G	2,140 K	936 M
Test set	100 M	198 K	-

1,993 to 2,002 and extracted paragraphs containing "background" and "example". This method is similar to the NTCIR 7 Patent Translation Task(Fujii et al., 2008). The small and large training data sets contained 100 Mwords and 5 Gwords, respectively. Furthermore, we sampled another 100 Mwords as a test set to measure the access speed for extracting $n$gram probabilities. We used an Intel® Xeon® X5675 (3.07 GHz) 24-core server with 142 GB of RAM.

Our experiments were performed from the viewpoints of speed and model size. We executed each program twice, and the results of the second run were taken as the final performance.

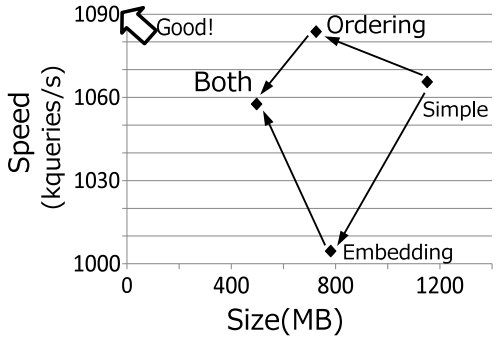

Figure 11: Comparison between tuned and non-tuned double-array structures.

Table 2: Comparison between tuned and non-tuned double-array structures.

Method	Size (MB)	Speed (queries/s)
`Simple`	1,152	1,065,536
`Embedding`	782	1,004,555
`Ordering`	726	1,083,703
`Both`	498	1,057,607

5.2 Optimization Methods

We compared the performance of the DALMs proposed here, namely `simple`, `embedding`, `ordering` and `both`, where `both` indicates that the language model uses both `embedding` and `ordering`. We conducted experiments examining how these methods affect the size of the double-array structures and the query speeds. We used the 100 Mwords model in the comparison because it was difficult to build a DALM using the 5 Gwords model.

The results are shown in Figure 11 and Table 2. While both `ordering` and `embedding` decreased the model size, the query speed was increased by the former and decreased by the latter. `Both` was the smallest and most balanced method.

5.3 Divided Double-Array Structure

Building a double-array structure requires a long time, which can sometimes be impractical. In fact, as mentioned above, waiting on construction of the double-array structure of the 5 Gwords model is infeasible.

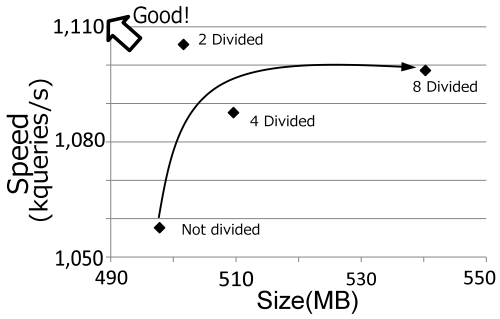

Figure 12: Comparison between divided and original double-array structures.

As described in Section 3.3, the efficient algorithm requires $O(UM)$ time to insert one node and the insertion is iterated N (the total number of insertions) times. If we assume that the number of unused slots at the ith insertion, U_i, is proportional to i, or that $U_i = c \times i$ where c is a proportionality constant, we can calculate the building time as follows: $\sum_{i=1}^{N} U_i M = O\left(MN^2\right)$.

To shorten the build time, we divided the original trie into several parts. Building parts of the original trie is possible because N is reduced. Moreover, these double-array structures can be built in parallel. Note that query results for both original and divided tries are completely equivalent because divided tries hold all the $n$gram statistics of the original trie. This method is similar to that used in *randomized language models* (Talbot and Brants, 2008).

We compared the differences between the methods using the original and divided double-array structures. In the comparison, we also used the 100 Mwords model with the `both` optimization method described in the previous section (Figure 12 and Table 3).

Although dividing the trie increased the size of the DALM slightly, the model size was still smaller than that without optimization. Query speed increased as the number of parts was increased. We attributed this to the divided DALM consisting of several double-array structures, each smaller than the undivided structure which results in an increase. Figure 12 shows that there is a trade-off relation between model size and query speed.

Below, we use the 5 Gwords model in our experiments. In our environment, building a 5 Gwords double-array structure required about 4 days when the double-array structures were divided into 8 parts, even though we used the more efficient algorithm described in Section 3.3. The time required for building the model when the original structure was divided into less than 8 parts was too long. Thus, a more efficient building algorithm is essential for advancing this research further.

Table 3: Comparison between divided and original double-array structures.

Number of parts	Size (MB)	Speed (queries/s)
1	498	1,057,607
2	502	1,105,358
4	510	1,087,619
8	540	1,098,594

5.4 Comparison with Other Methods

Using the 100 Mwords and 5 Gwords models, we compared DALM with other methods (KenLM (Kenneth Heafield, 2011) and SRILM (Stolcke, 2002)). In this experiment, we used the `both` method (which is mentioned above) for DALM and divided the original trie into 8 parts and built double-array structures.

The results are shown in Figure 13 and Table 4; the group on the left shows the results for the 100 Mwords model and the group on the right shows the results for the 5 Gwords model.

The experimental results clearly indicate that DALM is the fastest of all the compared methods and that the model size is nearly the same or slightly smaller than that of KenLM (`Probing`). Whereas KenLM (`Trie`) is the smallest model, it is slower than DALM.

The differences between the 5 Gwords versions of DALM and KenLM (`Probing`) are smaller in comparison with the 100 Mwords models. This is because hash-based language models have an advantage when storing higher-order $n$grams. Large language models have more 5grams, which leads to shorter backoff times. On the other hand, trie-based language models have to trace higher-order $n$grams for every query, which requires more time.

Finally, we discuss practical situations. We con-

Table 4: Comparison between DALM and other methods.

LM	100 Mwords Model		5 Gwords Model	
	Size (MB)	Speed (queries/s)	Size (MB)	Speed (queries/s)
SRILM	1,194	894,138	31,747	729,447
KenLM (`Probing`)	665	1,002,489	18,685	913,208
KenLM (`Trie`)	340	804,513	9,606	635,300
DALM (8 parts)	540	1,098,594	15,345	953,186

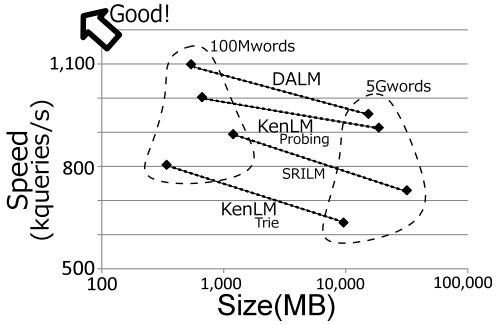

Figure 13: Comparison between DALM and other language model systems.

ducted this study's experiments using test set text written by humans. In some applications such as statistical machine translations, language model systems should compute probabilities of many unnatural $n$grams which will be unknown. This may affect query speed because querying unknown and unnatural $n$grams generate many backoffs. They may results in trie-based LM being slightly faster, because traversing the trie can stop immediately when it detects that a queried $n$gram history is not contained in the trie. On the other hand, hash-based LM such as KenLM probing would repeat queries until finding truncated $n$gram histories in the trie.

6 Conclusion

We proposed a method for implementing language models based on double-array structures. We call this method DALM. Moreover, we proposed two methods for optimizing DALM: `embedding` and `ordering`. `Embedding` is a method whereby empty spaces in arrays are used to store $n$gram probabilities and backoff weights, and `ordering` is a method for numbering word IDs; these methods reduce model size and increase query speed. These two optimization methods work well independently, but even better performance can be achieved if they are combined.

We also used a division method to build the model structure in several parts in order to speed up the construction of double-array structures. Although this procedure results in a slight increase in model size, the divided double-array structures mostly retained the compactness and speed of the original structure. The time required for building double-array structures is the bottleneck of DALM as it is sometimes too long to be practical, even though the model structure itself achieves high performance. In future work, we will develop a faster algorithm for building double-array structures.

While DALM has outperformed state-of-the-art language model implementations methods in our experiments, we should continue to consider ways to optimize the method for higher-order $n$grams.

Acknowledgments

We thank the anonymous reviewers for many valuable comments. This work is supported by JSPS KAKENHI Grant Number 24650063.

References

J.-I. Aoe. 1989. An Efficient Digital Search Algorithm by Using a Double-Array Structure. *IEEE Transactions on Software Engineering*, 15(9):1066–1077.

Atsushi Fujii, Makoto Iwayama, and Noriko Kando. 2004. Overview of the Patent Retrieval Task at NTCIR-4.

Atsushi Fujii, Makoto Iwayama, and Noriko Kando. 2005. Overview of Patent Retrieval Task at NTCIR-5.

Atsushi Fujii, Makoto Iwayama, and Noriko Kando. 2007. Overview of the Patent Retrieval Task at the NTCIR-6 Workshop. pages 359–365.

Djamal Belazzougui, Fabiano C. Botelho, and Martin Dietzfelbinger. 2009. Hash, displace, and compress. In *ESA*, pages 682–693.

Timothy C. Bell, John G. Cleary, and Ian H. Witten. 1990. Text compression.

Thorsten Brants, Ashok C. Popat, Peng Xu, Franz J. Och, and Jeffrey Dean. 2007. Large Language Models in Machine Translation. In *Proceedings of the 2007 Joint Conference on EMNLP-CoNLL*, pages 858–867. ACL.

B. Merialdo F. Jelinek. 1990. Self-organized language modeling for speech recognition.

Edward Fredkin. 1960. Trie memory. *Communications of the ACM*, 3(9):490–499.

Kimmo Fredriksson and Fedor Nikitin. 2007. Simple Compression Code Supporting Random Access and Fast String Matching. In *Proceedings of the 6th international conference on Experimental algorithms*, WEA'07, pages 203–216. Springer-Verlag.

Atsushi Fujii, Masao Utiyama, Mikio Yamamoto, and Takehito Utsuro. 2008. Overview of the Patent Translation Task at the NTCIR-7 Workshop.

Ulrich Germann, Eric Joanis, and Samuel Larkin. 2009. Tightly Packed Tries: How to Fit Large Models into Memory, and Make them Load Fast, Too. In *Proceedings of the Workshop on SETQA-NLP*, pages 31–39. ACL.

David Guthrie and Mark Hepple. 2010. Storing the Web in Memory: Space Efficient Language Models with Constant Time Retrieval. In *Proceedings of the 2010 Conference on EMNLP*, pages 262–272. ACL.

Kenneth Heafield. 2011. KenLM: Faster and Smaller Language Model Queries. In *Proceedings of the Sixth Workshop on Statistical Machine Translation*. ACL.

Makoto Iwayama, Atsushi Fujii, Noriko Kando, and Akihiko Takano. 2003. Overview of Patent Retrieval Task at NTCIR-3.

Yasumasa Nakamura and Hisatoshi Mochizuki. 2006. Fast Computation of Updating Method of a Dictionary for Compression Digital Search Tree. *Transactions of Information Processing Society of Japan. Data*, 47(13):16–27.

Adam Pauls and Dan Klein. 2011. Faster and Smaller N-Gram Language Models. In *Proceedings of the 49th Annual Meeting of the ACL-HLT*, pages 258–267. ACL.

A. Stolcke. 2002. SRILM-an Extensible Language Modeling Toolkit. *Seventh International Conference on Spoken Language Processing*.

David Talbot and Thorsten Brants. 2008. Randomized Language Models via Perfect Hash Functions. In *Proceedings of ACL-08: HLT*.

David Talbot and Miles Osborne. 2007. Smoothed Bloom Filter Language Models: Tera-Scale LMs on the Cheap. In *Proceedings of the 2007 Joint Conference on EMNLP-CoNLL*, pages 468–476.