A Breadth-First Representation for Tree Matching in Large Scale Forest-Based Translation

Sumukh Ghodke **Steven Bird** **Rui Zhang**
Department of Computer Science and Software Engineering
University of Melbourne, Victoria 3010, Australia
{sghodke,sb,rui}@csse.unimelb.edu.au

Abstract

Efficient data structures are necessary for searching large translation rule dictionaries in forest-based machine translation. We propose a breadth-first representation of tree structures that allows trees to be stored and accessed efficiently. We describe an algorithm that allows incremental search for trees in a forest and show that its performance is orders of magnitude faster than iterative search. A B-tree index is used to store the rule dictionaries. Prefix-compressed indexes with a large page size are found to provide a balance of fast search and disk space utilisation.

1 Introduction

Statistical machine translation (SMT) uses machine learning and parallel corpora to perform translations automatically. Syntax based SMT systems can be broadly classified into two types based on the input to the system: tree-based and string-based (Mi et al., 2008). In a tree-based system, the input is a parse tree of the source language, whereas in the latter, the input is a sequence of words that is simultaneously parsed and translated. Forest-based translation employs multiple parse trees for each source language sentence.

Forest-based translation can be performed in three main steps. First, the input sentence is parsed into a packed forest, which is then pruned. Next, for each tree in the packed forest, all matching translation rules are found. These translation rules are then combined to form a translation forest. The translation forest is finally decoded to produce a sentence in the target language.

The second step – finding all matching translation rules for each tree in the source sentence forest – is a complex task in itself. It could be performed by enumerating all trees in the forest and then searching for those trees in the translation rule dictionary. Of the enumerated trees, those that are present in the rule dictionary produce the translation forest. However, enumerating all possible trees from a forest incurs an exponential cost and many or most of those generated trees may not exist in the rule dictionary. Hence, a common approach in MT research is to perform the inverse task, and iterate over all the rules to check whether each rule matches anywhere in the source forest. This is a relatively easy task, especially if the rule dictionary is not large. In experimental setups a subset of rules may be used based on the prior knowledge of the rules required for the given test sentences. While that method works for research experiments on translation methods, no prior knowledge is available for online MT systems, and iterating over all rules will not be efficient for large rule dictionaries. Another method used to quicken the search is to limit the depth of the rules. Shallower rule trees are lesser in number. However, the reduction in depth can lower the translation quality.

This paper's contribution is in the second step of forest-based translation. We propose a breadth-first representation of translation rule trees and a sorted index architecture to store and retrieve translation rules efficiently. Together, they allow all translation trees to be discovered at each node in the forest, incrementally. Each node in the forest is expanded to form trees that are present in the rule dictionary. This allows the rule dictionary to contain all rules without any restriction on the depth of the rule or the size of the collection.

Translation rules are stored in the BerkeleyDB (Olson et al., 1999) B-tree index structure. Section 3 describes the breadth-first representation used to store trees in the index. The architecture of the translation system and the method of incremental tree expansion are described in Section 4. Section 4 also discusses the significance of the breadth-first over a depth-first tree storage approach.

Section 5 describes our experimental setup. The experiments measure the total time to construct a translation forest from a pruned source forest. An iterative search over the rule dictionary is the baseline performance measure. In addition to comparing the indexed search with the baseline, this section also analyses the effect of page size and compression on search performance. Multiple rule indexes are created to test the effect of the database parameter settings. The operating system disk cache and the database cache are cleared to simulate a fresh start before operating on each forest. The experiments are conducted on two rule dictionaries and three forest collections. The first rule dictionary has 11.8M rule trees and the second has 33.1M rules. The rules are extracted from NIST (2010a) data while the forest collections are extracted from NIST (2010a; 2010b) data.

2 Background

In this section we present the fundamentals of packed forests and forest-based translation before briefly describing the B-tree index structure.

2.1 Packed Forest

Packed forests or forests are directed hypergraphs and have been used to model and represent several applications in computer science and discrete mathematics (Klein and Manning, 2001).

Directed hypergraphs can be defined as a pair: $H = (V, E)$ where $V = \{v_1, v_2, \cdots, v_n\}$ is the set of nodes and $E = \{E_1, E_2, \cdots, E_m\}$ is the set of hyperedges. Each hyperedge can be defined as a pair: $E_i = (X_i, Y_i) \mid X_i,\ Y_i \subseteq V,\ i = 1, 2, \cdots, m.$

Packed forests have been used in NLP in the area of sentence parsing (Gallo et al., 1993) where the propositions of a parse analysis correspond to the nodes in the hypergraph and rules are represented as hyperedges. A similar model is used in forest based machine translation, where multiple parses of the input sentence are modelled as a forest. In NLP applications, the head of a hyperedge is usually a single node. This allows a single hyperedge to be semantically equivalent to a tree node with its children.

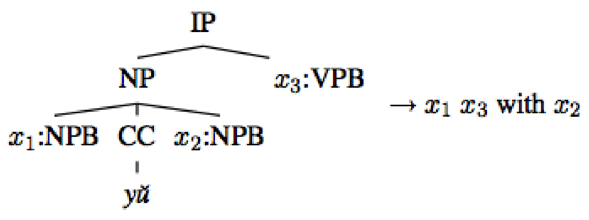


Figure 1: A translation rule reproduced from (Mi and Huang, 2008)

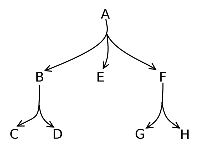
2.2 Forest Based Translation

Forest-based SMT overcomes the limitations of parse errors in tree-based translation and has been shown to be faster than k-best tree-based translation (Mi et al., 2008). For translating a sentence using the forest-based translation technique, we require a parser that can process a source language sentence and produce a packed forest and a translation rule dictionary, or database, which is a collection of *tree-to-string* translation rules (Huang et al., 2006; Liu et al., 2006).

A translation rule is a mapping from a source language tree to a string in the target language. An example translation rule from Chinese to English is shown in Figure 1. The left-hand-side is the source language tree. The Chinese word *yǔ* is translated to *with* in the target side on the right. The x_i variables on the right-hand-side are placeholders for the corresponding elements in the tree. Other numerical parameters associated with each translation rule are not shown here. Note that there may be several rules in the rule dictionary that have identical source language trees but different translations.

Once a source language sentence is parsed into a packed forest and pruned, the next step is to find trees in the forest that have matching rules in the translation rule dictionary. Matching rules are used to produce a translation forest for decoding into a target language string. The forest is a hypergraph made up of hyperedges where each hyperedge has a single source node, while the rules in the rule dictionary are trees.

Recent work by Huang and Mi (2010) has shown that forest expansion can be done in decoders using beam search (Koehn, 2004). However, to the best of our knowledge, no index-based search structures for incrementally finding trees have been proposed to date.



(a)

A(B(C D) E F(G H))

(b)

A3B2C0D0E0F2G0H0

(c)

A3B2E0F2C0D0G0H0

(d)

ABEFCDGH32020000

(e)

Figure 2: Transforming a tree into a breadth-first format

2.3 Indexing Methods

Data structures for indexing can be broadly classified into either memory based or disk based. Memory based indexes are not considered here since an online SMT system with a large translation rule dictionary requires disk based storage.

B-trees are balanced search trees that are designed to work efficiently from the disk. They minimise disk I/O and provide an insert and delete time complexity of *O(log n)*, where *n* is the number of objects in the database. The keys in a B-tree are always in sorted order, therefore providing a guarantee of proximity for comparable keys.

The B-tree implementation in BerkeleyDB is used in our experiments. Since the keys are sorted, consecutive keys often share a common prefix. BerkeleyDB allows the compression of keys in a database via user-defined compress and decompress functions. If no such functions are supplied, it performs prefix compression. Our experiments test the performance of search with page size and compression of keys. Other settings such as the minimum number of keys for each leaf page and cache size allow us to fine tune the performance of the system, but we do not report experiments on those for lack of space.

3 Breadth-first Representation

The storage and efficient retrieval of the left-hand-side of translation rules is closely linked to the format in which they are stored. This section describes the format we use to represent trees both in queries and for rules in the index.

Trees can be represented linearly in several ways. As mentioned in Section 2.2, although a forest is a hypergraph made up of hyperedges, the left-hand-sides of rules in the rule dictionary are actually trees. Each node in a forest can have zero or more outgoing hyperedges. A hyperedge is be treated as a tree node with its children. Source sentence trees are constructed by recursively expanding nodes in the forest.

Figure 2a shows a tree constructed with three hyperedges, having head nodes A, B and F. Figure 2b shows the typical string representation of the tree where a node's descendants are scoped by an open and close bracket. The bracketed format, although visually intuitive, is not efficient for storage and processing. There has been much research on succinct representation of tree structures. Succinct representations encode trees in the most compact fashion by either using a balanced bracket representation or depth first unary degree sequences (Jansson et al., 2007). Such methods often represent the open and close brackets of a tree in bit notations. However, braces can only be matched when the trees are stored in a depth-first format and it cannot be applied to breadth-first representations.

Figure 2c represents the tree as a sequence of label and integer pairs. The labels represent node labels in the tree in a depth first format and the integers that follow each node label give the number of children at that node. Figure 2d shows a similar encoding scheme, but with the tree traversed in a breadth-first format. Finally, Figure 2e shows the format used in this paper. In our format, we separate the node labels and the integers representing the number of child nodes. When using this representation in the index, each node label is represented by a unique integer, which occupies four bytes, whereas the integer representing the number of child nodes is restricted to a maximum value of 255, and can therefore be represented in one byte.

Separating the node labels from integers representing the number of child nodes gives us a greater prefix overlap between consecutive expansions. For example, if we were to use the interleaved expansion method shown in Figure 2d on the hyperedge with head A and then expand the hyperedge with head B, the breadth-first representation of the trees would be A3B0E0F0 and A3B2E0F0C0D0. We can see that the expansion of the node B has resulted in its number of children being updated to 2 from 0. Since, each label is encoded using 4 bytes and the child count using 1 byte, the overlap between the expanded sequences is 9 bytes. With the separated breadth-first format (Figure 2e) the two trees are ABEF3000 and ABEFCD320000, an overlap of 4 node labels or 16 bytes. The separated form helps in increasing the prefix overlap of expanded tree structures.

If, instead, the depth-first notation is used, then the second tree is represented as A3B2C0D0E0F0. Then, even if the integers are separated, the prefix shared by the two queries is only 9 bytes.

To generalise the breadth-first representation and the prefix overlap, consider a tree T containing k nodes, denoted here as an ordered sequence $N = (n_1, n_2, \cdots, n_k)$ in breadth-first order. Each node is either fully expanded or is collapsed, therefore, if k is greater than 1, then the first node, being the root, must be fully expanded. Let the labels of nodes in N be a sequence $L = (l_1, l_2, \cdots, l_k)$ and let $C = (c_1, c_2, \cdots, c_k)$ denote the sequence of the number of children of each node in N. The separated breadth-first representation of T, which is used in this paper, is the sequence $B = (l_1, l_2, \cdots, l_k, c_1, c_2, \cdots, c_k)$. In a tree T, the number of nodes that can be expanded is equal to the number of leaves p, where $p \leq k$. Each expansion of a leaf node in T produces a new tree. Let $S = \{T_i \mid i \in \{1, 2, \cdots, k\}\}$ be the set of all trees produced as a result of the expansion, where $|S| = p$ and each tree, T_i, is an expansion of node n_i in T and its separated breadth-first representation is B_i. Let $N'_i = \{n_j \mid depth(n_j) \leq depth(n_i)\}$ be a set of all nodes of depth less than or equal to node n_i's depth in T. Then, B_i shares a prefix of at least $4\,|N'_i|$ bytes with B.

Now, each node n_i has been considered to have either no child or one ordered set of children. While this is true in a tree, a forest is different. Each node in a forest can have multiple expansions, or hyperedges. The algorithm discussed in the next section therefore iterates over all possible hyperedges at a node.

4 Index Architecture

This section explains the structure of the index and the incremental search algorithm. To present a broader picture of where the index and search components fit within a larger online MT system, a representative block diagram of a complete forest-based MT system is provided in Figure 3. We focus only on the the incremental search algorithm and the rule dictionary in this section. The function of other components in the indexed data store is only briefly explained as they are not critical to the operation of the search algorithm.

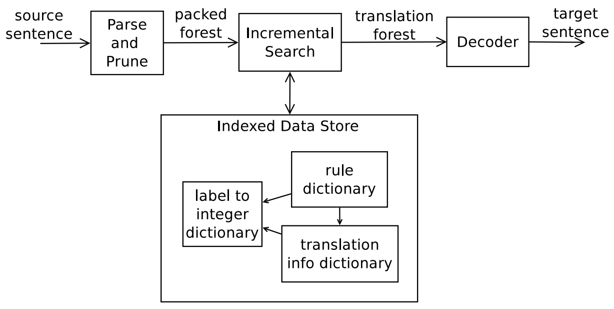


Figure 3: Architecture for an online forest-based translation system

The indexed data store is composed of three dictionaries. The first is a label to integer mapper. It assigns an integer identifier to all unique labels in the system. All tree node labels, including words, that appear on both sides of translation rules are indexed within this dictionary. The integer id for a label can be looked up using the label string as the key while encoding a tree query or while creating the other two dictionaries. A reverse mapping from the integer id to the label is used while constructing the translation forest from encoded rule trees. Any key-value data-structure could be used for the label mapper. It could even be a memory based hash table if it fits in memory.

The other two dictionaries, the rule dictionary and the translation info dictionary, contain the information about translation rules. The rule dictionary or database is a collection of left-hand-sides of translation rule trees accessible by a tree as the key. Each tree in the rule database has a unique integer id as its value. The integer id is the reference to a position in the translation info dictionary. Each rule tree may have one or more target language translations. Hence, each entry in the translation info database is a list of translations. Each element in the list is a tuple of the target language strings and the parameters defining the statistics of that translation. The target language strings are stored as a sequence of integer ids and the parameters are stored in a pre-defined binary format. The translation info database uses a simple record list structure from the BerkeleyDB library.

A large rule dictionary, requires a disk-resident index to support efficient access. The B-tree data-structure is used as the rule database because of its scaling capabilities and its sorted key storage property which is critical to the operation of the incremental search algorithm.

4.1 Incremental Forest Exploration

The incremental search algorithm finds trees within a forest that are also present in the rule dictionary. Nodes are expanded incrementally to ensure that expansions occur only where there is a possibility of a translation rule when expanded. This section also explains the significance of the breadth-first representation to the working of the algorithm and the reason why depth-first encoding will not work.

The algorithm works on a given source language forest where each node has one or more hyperedges. The function Process_Node in Pseudocode 1 iterates over each node in a forest in order to find the trees originating at that node. The incremental search itself is detailed in function Find_Tree_Increment in Pseudocode 1. The increment finding function is now explained here by assuming that a tree is being processed. It can be extended to work with forests as shown in the pseudocode.

Consider the tree T with k nodes, of which p nodes are leaves, as described in Section 3. The separated breadth-first representation of T, B, is essentially made up of two halves. The first half of B contains node label identifiers while the latter half contains child counts. This is true when the elements of B are considered as integers only, but the ratio is different in the binary form of B where the first half is represented using 4 bytes and the second half uses only 1 byte each. Nevertheless, both halves represent the information of nodes in sequence N, where N is in breadth-first order. To find a tree T in the rule dictionary, the tree has to be translated to the binary encoding of B to be searched in the index. Then, the tree would need to be expanded at p nodes to check if there are any possible rules that match when expanded. The process of finding any expansions of a tree is performed recursively in the function Find_Tree_Increment of Pseudocode 1.

When expanding trees, the data-structure needs to be an efficient one to operate on, since this is a recursive task. The algorithm should also facilitate a simple way of tracking the position in the tree where the expansion is taking place. This can be performed very easily because of a prop-

Pseudocode 1 Incremental search for trees

```
proc Get_Translation_Forest sourceForest
  RuleDB points to the translation rule database
  for node in sourceForest do
    call Process_Node with node
  end for

proc Process_Node node
  if node doesn't have hyperedges then
    label ← node's label
    word ← child of node
    # the only tree possible is 'label(word)'
    query ← call Separated_Breadth_First with label(word)
    if query is present in RuleDB then
      add query to translation forest
    end if
    return
  end if
  hyperedges ← all hyperedges at node
  for hyperedge in hyperedges do
    query ← call Separated_Breadth_First with hyperedge
    if query is found in RuleDB then
      add query to translation forest
    end if
    if query is found or (query is not found but prefix of
    query is found) then
      call Find_Tree_Increment with query
    end if
  end for

proc Find_Tree_Increment sbfArray
  labels ← first half of sbfArray
  childCounts ← second half of sbfArray
  lastNd ← node corresponding to labels' last element
  startNd ← node imm. after node corresponding to
  childCounts's last non-zero entry
  # startNd contains the node after the last expanded node
  # in breadth-first order
  for node from startNd to lastNd do
    if node doesn't have hyperedges then
      word ← child of node
      appLabel ← labels + [word's label]
      appChild ← childCounts + [0]
      appChild[node] ← the number of children of node
      query ← appLabel + appChild
      if query is present in RuleDB then
        add query to translation forest
      end if
      continue
    end if
    hyperedges ← all hyperedges at node
    for hyperedge in hyperedges do
      numTails ← number of hyperedge's tail nodes
      appLabel ← labels + [hyperedge's tail labels]
      arrayOfZeroes ← [0] * numTails
      appChild ← childCounts + arrayOfZeroes
      appChild[node] ← numTails
      query ← appLabel + appChild
      if query is found in RuleDB then
        add query to translation forest
      end if
      if query is found or (query is not found but prefix of
      query is found) then
        call Find_Tree_Increment with query
      end if
    end for
  end for

proc Separated_Breadth_First tree
  return the separated breadth first representation of tree
```

erty of the breadth-first representation. The property is that, when we expand node n_i in a tree T, and the $depth(n_i) = depth(T)$ or $depth(n_i) = depth(T) - 1$ and $\nexists\ n_j$ where $j > i$ and n_j is expanded, then, the child nodes of node n_i can simply be appended to N. In the function Find_Tree_Increment, the $startNd$ and $lastNd$ variables point to the start and end of expandable nodes in the tree represented by $sbfArray$. The $startNd$ node is located by finding the node immediately after the last expanded node when traversing the sequence N from k to 1. A node n_i is expanded if c_i is greater than 0. Hence, since this pattern exists, the recursive function treats its input parameter as a list and ignores the fact that it's operating on a tree.

A depth-first representation does not support such a property that allows expansions of nodes to be appended at the end of the list. This is because the tree expansion for searching rules is such that when a node is expanded, all its children have to be included at once, due to which expansions of trees in the depth-first method will almost always have child nodes appear in-between the original sequence of nodes. The only exception is when the very last node in the depth-first sequence is expanded. Having successive expansions differ from one another reduces their proximity in the index.

5 Experiments

5.1 Data

We use two rule collections in our experiments. The rules are extracted from 1.5M word-aligned sentence pairs of Chinese and English from the NIST (2010a) machine translation corpus. The first rule collection contains 11.8M Chinese translation rules, of which about 7.8M are for unique trees. All trees are limited to a depth of five levels. The second rule collection contains 33.1M rules, of which 23.9M are unique. The rules in the second collection are binarized but not limited by depth. The rule collections are 285MB and 2.5GB on disk, respectively, when gzipped.

In total, we use three forest collections. The first forest collection contains 254 forests from NIST (2010a) data. The second forest collection contains 288 forests, also obtained by parsing NIST (2010a) data, but they are binarized to work with the second rule collection. The third forest collection contains 348 binarized forests obtained by parsing NIST (2010b) data.

rule set	page size	regular	compressed
1	4K	1001MB	569MB
1	8K	947MB	537MB
1	16K	960MB	525MB
1	32K	954MB	496MB
2	4K	3.1GB	1.6GB
2	8K	3.0GB	1.5GB
2	16K	3.0GB	1.3GB
2	32K	3.0GB	1.2GB

Table 1: Size of B-tree indexes

5.2 Setup

The experiments are run on an Intel Core2Duo processor (2.4GHz; 4MB cache) desktop machine with 2GB of RAM. The operating system is Ubuntu desktop edition version 10.04.2. The index and program are installed on two different hard disks, both having a disk speed of 5400 rpm. The disk on which the code is installed has a page size of 4KB while the index disk has a page size of 8KB.

The index is created using B-tree index structures in the BerkeleyDB (version 5.1) library. The search algorithm is coded in Python and is open sourced[1]. Database accesses from the python program are through a python wrapper for BerkeleyDB called bsddb3[2]. The B-tree indexes are created with and without prefix compression. We also create indexes with four different page sizes (4K, 8K, 16K, and 32K) to analyse the effect of page size on performance. The sizes of the B-tree rule dictionaries are shown in Table 1.

Caching has a significant impact on performance measurements for disk-based searching. There are a minimum of 2 levels of caching while performing the tests. One cache is handled by the database while the other is a disk level cache managed by the operating system. We clear the database cache by choosing an in-memory cache and re-starting the database process when required. The commands *sync;* followed by *echo 3 > /proc/sys/vm/drop_caches;* are used to clear the operating system cache.

Search time is measured using the *datetime* package in Python version 2.6 and cache performance of the database is measured using BerkeleyDB tools. Every test is run three times and the average time is reported. The exact search process

[1]https://bitbucket.org/leopardspot/forest-search

[2]http://pybsddb.sourceforge.net/bsddb3.html

rule set	forest set	baseline	breadth-first
1	1	$\sim$ 23min	1.55sec
2	2	$\sim$ 80min	15.71sec
2	3	$\sim$ 80min	11.49sec

Table 2: Average time to search a forest

is explained in the specific experiment's sections.

5.3 Incremental Tree Matching

Experiment 1: Comparison with baseline

This experiment compares the performance of the baseline system with the breadth-first index based approach. The baseline checks if each rule in the rule dictionary is found in the forest being translated. If found, the rule is added to the translation forest. Almost one-third of the rules in the full rule collection have a common left-hand side. Also, rules with the same left-hand-side have been found to be consecutive in the collection. Therefore, the outcome of previous rule's check is reused if the current rule is same as the previous one.

The average forest search time with the baseline setup is compared with the best breadth-first index based search method in Table 2. The times reported for the incremental breadth-first search are obtained when using a regular 32K page size index. The average time is the ratio of the total time required to search all forests in a forest set, and the number of forests in the set.

Experiment 2: Page size and compression

This experiment compares the effects of page size and compression on the search time in a breadth-first index. For each forest in a forest collection, the time to search the forest is measured. An average of those times across all forests in the collection is shown in Table 3 for indexes of various page sizes, with and without compression.

Experiment 3: Cache utilisation

In order to measure the cache utilisation of the indexes, the forests are searched sequentially without closing the database or clearing the disk cache. A BerkeleyDB utility is then used to obtain the memory usage statistics which displays the cache hit ratio. The cache usage will depend on the order in which the forests are searched and also on the content of the forests. However, the forests are searched in the same order across all indexes, therefore, their relative cache performances can be meaningfully compared.

rule set	forest set	page size	regular (s)	compressed (s)
1	1	4K	2.35	2.09
1	1	8K	2.01	1.93
1	1	16K	1.69	1.86
1	1	32K	1.55	2.01
2	2	4K	20.86	22.90
2	2	8K	25.42	18.52
2	2	16K	17.73	18.69
2	2	32K	15.71	19.62
2	3	4K	14.13	15.03
2	3	8K	17.86	12.78
2	3	16K	12.63	13.16
2	3	32K	11.49	14.03

Table 3: Average time for breadth-first search

5.4 Analysis

From Table 2, it is clear that the baseline method is more than an order of magnitude slower. It can also be observed that forest sets 2 and 3 take about the same amount of time on average over rule set 2. It is therefore clear that the search time for the baseline method is primarily determined by the size of the rule set and not on the size of the forest. Table 3 shows that the best performance in B-tree indexes is always obtained in the uncompressed, or regular, index and when using the 32K page size. However, the performance of the compressed index seems to stabilise and not deteriorate much beyond the 8K page size. Considering that the compressed indexes are almost half the size of the regular index – or even smaller than half in the case of rule set 2 with 16K and 32K page sizes – they might be the better option to consider for very large collections.

The average run times on a collection of forests depend on the properties of individual forests in the collection. A histogram of the number of forests searched in time intervals shows us that when searching a compressed index the search operates faster on already fast searches and slower on the initially-slow searches. Figure 4 and Figure 5 show the histograms for the third set of forests searched using a compressed and an uncompressed index, respectively, each with a page size of 8K. From the figures, it is evident that the number of forests translated in less than 5s – the first stick in the figures – increase in the compressed index, while the slower forests which take about 200s on a regular index take about 250s or

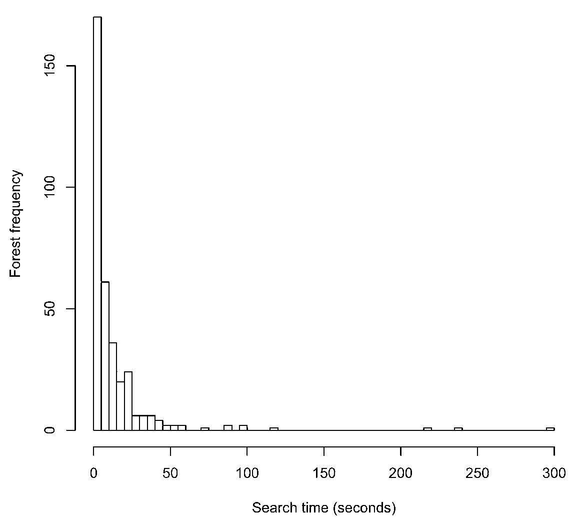


Figure 4: Distribution of forests searched in an 8K compressed index

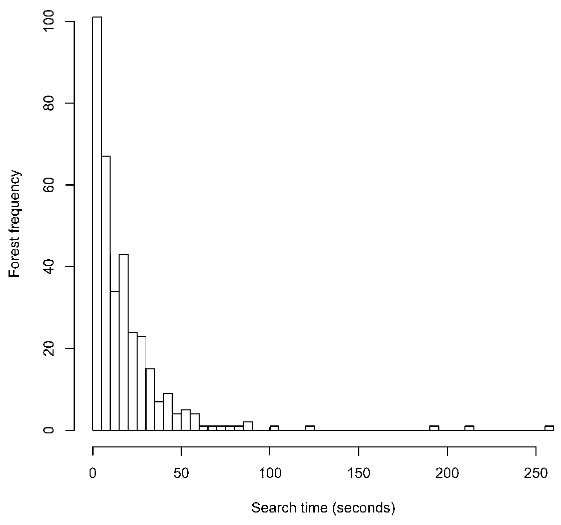


Figure 5: Distribution of forests searched in an 8K uncompressed index

longer on the compressed index. This is true even in other page sizes. Histograms for the 32K page size index – omitted here for lack of space – look similar to Figure 4 and Figure 5, but the compressed index shows a very marginal increase in the number of faster forests and a clear increase in the time for the slower forests. But, even without compression, the 32K index has about the same number of forests translated in less than 5s as the faster 8K compressed index.

Cache utilisation is found to be very high in all cases. The number of pages retrieved from a compressed index is always lower than that for an uncompressed one. Cache misses are clearly not the cause for the slowdown in the compressed indexes. The decompress operation is the most likely reason. However, with compression, more queries, or keys, would fit in one page which may causes a speedup when fewer queries are required while searching a forest.

Another observation is that although rule set 2 is about 3 times as large as rule set 1– both in the number of rules and in the size of the index – the search times do not show a 3-fold increase. This could be the result of various factors. One, the rule set 1 might have more pages cached than rule set 2, because it's a smaller index. Two, the depth of the rules in rule set 1 are limited to 5 whereas the depth is not limited in set 2. We feel that the most likely reason in this case is the first one. However, to verify we would have to test with a larger dataset which is limited in depth.

6 Conclusion and Future Work

We propose a breadth first representation for trees and use that representation in an algorithm to incrementally find all trees in a forest that are also present in a rule dictionary. We compare the performance of the algorithm on two different rule sets and three forest collections. We find our method to outperform an exhaustive search baseline by more than an order of magnitude. We find that the a compressed index provides a balance of fast search and less disk space, but uncompressed indexes are faster with large B-tree page sizes.

In the future, we would like to explore the properties of forests that influence their search time. We would also like to perform more experiments with depth limitation to ascertain if a depth limit improves performance.

Acknowledgements

We thank Dr. Liang Huang from ISI, USA and Dr. Haitao Mi from ICT, China, for their feedback and technical discussions during this research. We also thank Google for their travel scholarship awarded to the first author.

References

Giorgio Gallo, Giustino Longo, Stefano Pallottino, and Sang Nguyen. 1993. Directed hypergraphs and applications. *Discrete Applied Mathematics*, 42:177–201, April.

NIST Multimodal Information Group. 2010a. NIST 2006 Open Machine Translation (OpenMT) Evaluation, Linguistic Data Consortium, Philadelphia.

NIST Multimodal Information Group. 2010b. NIST 2008 Open Machine Translation (OpenMT) Evaluation, Linguistic Data Consortium, Philadelphia.

Liang Huang and Haitao Mi. 2010. Efficient incremental decoding for tree-to-string translation. In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*, EMNLP '10, pages 273–283, Stroudsburg, PA, USA. Association for Computational Linguistics.

Liang Huang, Kevin Knight, and Aravind Joshi. 2006. A syntax-directed translator with extended domain of locality. In *Proceedings of the Workshop on Computationally Hard Problems and Joint Inference in Speech and Language Processing*, CHSLP '06, pages 1–8, Stroudsburg, PA, USA. Association for Computational Linguistics.

Jesper Jansson, Kunihiko Sadakane, and Wing-Kin Sung. 2007. Ultra-succinct representation of ordered trees. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '07, pages 575–584, Philadelphia, PA, USA. Society for Industrial and Applied Mathematics.

Dan Klein and Christopher D. Manning. 2001. Parsing and hypergraphs. In *Seventh International Workshop on Parsing Technologies (IWPT- 2001)*, pages 123–134, Beijing, China, October.

Philipp Koehn. 2004. Pharaoh: a beam search decoder for phrase-based statistical machine translation models. In *Proceedings of the Sixth Conference of the Association for Machine Translation in the Americas*, pages 115–124.

Yang Liu, Qun Liu, and Shouxun Lin. 2006. Tree-to-string alignment template for statistical machine translation. In *Proceedings of the 21st International Conference on Computational Linguistics and the 44th annual meeting of the Association for Computational Linguistics*, ACL-44, pages 609–616, Stroudsburg, PA, USA. Association for Computational Linguistics.

Haitao Mi and Liang Huang. 2008. Forest-based translation rule extraction. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, EMNLP '08, pages 206–214, Stroudsburg, PA, USA. Association for Computational Linguistics.

Haitao Mi, Liang Huang, and Qun Liu. 2008. Forest-based translation. In *Proceedings of ACL-08: HLT*, pages 192–199, Columbus, Ohio, USA, June. Association for Computational Linguistics.

Michael A. Olson, Keith Bostic, and Margo Seltzer. 1999. Berkeley DB. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 43–43, Berkeley, CA, USA. USENIX Association.