COMPACT SUBNETWORK-BASED LARGE VOCABULARY 
CONTINUOUS SPEECH RECOGNITION*

Dong-Hoon Ahn and Minhwa Chung
Department of Computer Science
Sogang University, Seoul, Korea
drahn@nlpeng.sogang.ac.kr, mchung@sogang.ac.kr

ABSTRACT
We present an improved method of compactly organizing the 
decoding network for a semi-dynamic network decoder. In the 
previous work [1], the network management units called sub-
networks were made compact by self-structuring themselves. 
We improve this subnetwork representation in two aspects by 
employing the shared-tail topology [2]. Firstly, we localize the 
decoding algorithm so that it works with a set of subnetworks 
rather than with the whole decoding network. Secondly, we 
align unshared suffixes of pronunciations into a shared tail to 
reduce redundancies. Experimental results on a 20k-word Ko-

rean dictation task show that our algorithm significantly reduces 
the memory requirement and produces additional gains in word 
accuracy by using aligned shared tails.

1. INTRODUCTION
Many speech recognition systems often employ a single-pass 
decoder using the most complex knowledge available such as 
crossword context acoustic models and N(≥3)-gram language 
models [5]. Earlier applications of such detailed knowledge 
give more discrimination to search hypotheses, leading to 
tighter beam pruning and therefore to more efficient decoding. 
In this situation, an efficient representation of search space be-
comes quite important due to limited computational resources. 
Accordingly, recent efforts have focused on identifying and 
removing redundancies in an integrated search network. 
Among others, one notable work is to identify linear tails 
and reorganize an original search network into more compact 
one with shared-tail topology [2]. The shared-tail topology can 
be efficiently obtained by considering indistinguishable states 
in linear tails among successor trees. Since the shared-tail to-

pology is based on the indistinguishability among states, the 
concept is essentially the same as the general automata mini-
mization algorithm [6]. However it can be efficiently implemented 
by further exploiting the tree topology of a search network and 
only concerning the linear tails in LM probability-factorized 
successor tree. Others such as “weight pushing” [7] followed by 
either weighted minimization [7] or “on the fly” network com-
pression using suffix lexicon [4], have employed similar ap-
proaches using such indistinguishability among states.

This paper describes how we have employed the idea of lin-
ear tails in our subnetwork-based semi-dynamic network de-

coder [1] with two distinguished improvements. The first im-

mprovement is on the memory efficiency issue. According to 
the original description in [2], the algorithm requires the whole 
search network as its input. Hence both the whole network and 
the working space for the algorithm should be available in 
memory. As a search network that is not optimized yet is typi-
cally huge and even impossible to be held in the limited mem-

ory, it may not be practically applicable. However, considering 
the facts that linear tails from the same word \(x\) in different suc-
cessor trees (subnetworks) that reach the root of the successor 
tree of \(x\) go into the shared tail, and that each successor tree is 
defined on each LM history, we can “localize” the original al-
gorithm such that the modified algorithm works with a set of 
related subnetworks rather than the whole network. Secondly, 
although the work in [3] has extended the original algorithm [2] 
to allow multiple pronunciation models, there still remain re-
dundancies among similar ones for each word. These redundan-
cies prevail in a tree-structured network. In our approach, while 
linear tails for each word are shared, they are aligned using the 
DP matching procedure to get more compact shared tails.

In the following, we review our subnetwork-based semi-
dynamic network decoder and then present the improved algo-

rithm for constructing shared tails. Finally we describe experi-
mental results on the algorithm efficiency and its effects on 
decoding time and memory usage.

2. SEMI-DYNAMIC NETWORK DECODER
In [1], we have introduced the concept of the semi-dynamic 

network decoder using the language model network and the 
subnetwork-based representation. By “semi-dynamic”, we mean 
a search network is managed with hybrid properties of the static 
and the dynamic network decoder. Thus rather than instantiat-
ing all the network components, a semi-dynamic network de-
coder only manages parts of the whole network to be used at 
hand as the dynamic network decoder does. Again, it differs 
from the latter that concerns its components such as nodes indi-

dually, in that the former concerns each subnetwork that ag-

gregates some of them. This management system consists of the 
following components.

2.1. Self-structuring subnetwork
A subnetwork is a tactfully designed network component so that 
it nodes, arcs, and non-zero weights are separated into individ-


dual sets and can be identified by unique indexes, removing un-
necessary zero-weights. In addition, offsets (byte distances) to
each set are headed so that each set can be identified instantly. We call it self-structuring subnetwork since with a simple translation of offsets into physical addresses on its instantiation, we can identify each component by index. For this reason, subnetworks are used as simple network management units in the semi-dynamic network decoder.

### 2.2. Language model network

A language model (LM) network is a representation that models an arbitrary LM so that a search network is transparently built from the specific language model. Typically, a language model probability is calculated with the current word given its LM history. A language model network is constructed using the fact that regardless of the type of LM, a pair of a LM history and each word following it becomes another history that will be followed by a set of words. Each node in the network corresponds to a LM context with a pair of (history, word) and each arc, to a transition between contexts weighted by its conditional log-probability given a word. A backtrack arc is added to account for unseen word sequences. On its completion, the network minimization algorithm is applied [6].

### 2.3. Search network based on subnetwork representation

After constructing a language model network, the actual search network is derived. During its construction, for each LM context, its following words are gathered into successor trees with factorizations of LM probabilities [2], which are then transformed into self-structuring subnetworks. Finally, a chunk of memory occupied by the subnetworks is written on a disk “as is”. When the complete set of subnetworks is processed we get the complete search network stored in a disk. Figure 1 shows steps to organize subnetwork-based search network.

![Subnetwork-based search network](image)

**Figure 1**: Subnetwork-based search network.

### 2.4. Semi-dynamic network decoder

The search is done with a one-pass Viterbi decoder based on the token-passing algorithm. To accommodate the semi-dynamic network management, the token passing algorithm is modified so that when each token passes over the next subnetwork, the subnetwork is instantiated from disk if not available.

There are two operations to make the semi-dynamic management system more efficient.

#### 2.4.1. Subnetwork caching

To alleviate frequent instantiation requests and limit the memory usages for subnetworks, frequently activated ones are cached in memory. Deactivated ones that are instantiated but have no tokens may reside in memory to get a chance to be activated. However, they will be withdrawn when it is less likely to get tokens in the near future, specifically, when they have no tokens during pre-defined interval of $k$-threshold (in frame) since the last token is dropped.

Interestingly, a subnetwork caching has the same effect as the disk-based LM strategy [8] to cache parts of frequently queried LM probabilities. This is due to the fact that a subnetwork is defined on each LM context equivalent to LM histories.

#### 2.4.2. Subnetwork preloading

To further reduce the decoding time, subnetworks likely to be frequently activated are statically pre-loaded when setting up the search network, to especially lower high rates of instantiations at initial decoding phase. To decide which subnetworks to be preloaded, we estimate the activation frequency of each subnetwork by the likelihood of a LM history associated with it, which can be efficiently calculated from the corresponding path score in a language model network. If the estimate is greater than the pre-specified $p$-threshold, the subnetwork will be pre-loaded.

### 3. COMPACT ALGORITHM

#### 3.1. Motivation

As stated in the introduction, most network minimization algorithms involve the identification of indistinguishable states [2][4][7] and are derived from the general automata minimization algorithm [6]. Two states are indistinguishable if the strings generated by the sets of paths originated from them are equivalent. Thus states are partitioned into one of the equivalence classes defined by the indistinguishability relation. The minimized automaton has a set of states that correspond to each class. [7] has extended it to weighted minimization with weight pushing applicable to the case that the “string” includes weights (probabilities) and output symbols on paths. While the general algorithm requires global knowledge on the whole network to learn equivalence relations among states, requiring much time and spaces to partition them into classes, in [2], they recognize that after LM-factorizing successor trees similar to weight pushing, linear tails reaching the same root of a successor tree are the major source of indistinguishabilities in a tree-structured network, and make such tails shared among successor trees into shared tails. As a successor tree is defined on each predecessor word or word history, a shared tail will be given each LM context except the one with no history. Thus it is noteworthy that it can be individually applied between a successor tree and linear tails reaching it without considering the whole network. However, this seems to be hard in its original implementation [2] because each successor tree is defined depending on another and cannot be separated from the whole network. As a result, the algorithm requires the complete network tagged with phonetic transcriptions of words as an input, which may demand much memory due to the huge network that is not optimized yet.

From the above discussion, we want to design a memory-efficient algorithm that works on the basis of each subnetwork (successor tree) rather than the whole network. Fortunately in our system, a subnetwork is made self-structuring, which enables most operations to be performed on the basis of each sub-
network with the least effects on others. In the next section, subnetwork-based network compaction algorithm is described.

We add one more improvement regarding linear tails in the use of multiple pronunciations of a word. Typically, when multiple pronunciations are used for each word, they are shared in prefixes with the same phonetic labels, but remain unshared in suffixes although they have similar tails that can be shared. This is a sort of redundancy prevailing in a tree-structured network. In our algorithm, while linear tails from each word are shared, they are aligned using the DP matching procedure to get more compact shared tail.

3.2. Algorithm

The compaction algorithm is shown in Figure 2. The algorithm requires a subnetwork-based search network written on a disk. Three modifications are made to the structure of the original subnetwork. Firstly, word ends (nodes identifying the ends of words) reaching the same LM context are shared during network construction. Secondly, a new set called extern set is added in each subnetwork. Since a node may be now linked to a non-root node such as word ends, a subnetwork can have multiple entries where nodes in other subnetworks are linked. Thus for each arc linked to an external node, the subnetwork that owns the node is recorded in the extern set while the node goes in the arc set, where the two sets are on the subnetwork that has the arc. Finally, nodes that begin linear tails are marked. After this setup, for each word end in a subnetwork, we collect information on the membership of each linear tail. This information includes the index of a subnetwork that owns the word end (we.sid), a word end index (we.nid), and the linear tail itself linked to that word end. Since all linear tails reaching the same root are now collected in the shared word end, linear tails linked to each word end can be aligned into a shared tail by using a DP matching procedure. While doing this, we should note that an optimal multiple sequence alignment is never efficiently obtained because it is NP-complete [9]. Due to this reason, we take a simple approach where tails are ordered according to their lengths and then iteratively aligned to the shared tail. This works well, since different pronunciations of each word typically have small differences. The shared tail obtained from alignments is then included into the subnetwork that owns the root linked to its word end and is excluded from subnetworks, where linear tails are included. Finally, nodes, arcs, and weights are re-indexed using a network traversal algorithm. Figure 3 shows how this compaction algorithm works. In Figure 3(a), four linear tails from two subnetworks reach the same LM context. Each word end is shared during network construction and the corresponding linear tails are marked by dashed boxes. These linear tails are collected and aligned into a shared tail. Then, the shared tail is included into the subnetwork with a root linked to the word end w1 and is excluded from subnetworks that own the four linear tails. Finally, subnetworks are traversed to re-index their nodes, arcs, and weights.

4. EXPERIMENTAL RESULTS

4.1. Experimental setup

Our decoder is tested on a 20k-word dictation task for Korean. The acoustic models are trained using 70 hours of Korean read speech and have a set of tied-state CHMMs consisting of 7k tied states with each state of 12 Gaussian mixtures. LMs are trained using 7 million morphemes collected from newspaper articles, broadcast news and novels, with a modified Kneser-Ney estimation method for both bigram and trigram with cutoffs of 2 and 3 respectively, and an entropy pruning at the threshold of 1.0e-9. Also, most frequent 20,659 words from the corpus are used in the lexicon, with 30,908 pronunciations. Thus each word has about 1.5 pronunciations. A test set consists

---

**Figure 2:** Subnetwork-based network compaction algorithm.

**Figure 3:** Illustration of the compaction algorithm.
of 608 sentences without OOVs. Their perplexities are 132.7 (bigram) and 99.5 (trigram).

4.2. Results

Table 1 shows the results of compaction algorithm. In the below, a ‘compact version’ means a network that the compaction algorithm is applied to while ‘baseline’ means a search network in [1]. The significant difference between a bigram and a trigram network results from the property that in case of trigrams, subnetworks with one-word history (bigram subnetworks) are only connected to those with two-word history and can’t be shared with others, i.e. only trigram subnetworks may have shared tails.

Table 1: Subnetwork sizes (compact version/baseline).

<table>
<thead>
<tr>
<th>count</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>node (M) arc (M)</td>
</tr>
<tr>
<td></td>
<td>Mbytes C/B ratio</td>
</tr>
<tr>
<td>bigram</td>
<td>0.51/3.15 1.30/3.95</td>
</tr>
<tr>
<td></td>
<td>22.49/85.96 26.16%</td>
</tr>
<tr>
<td>trigram</td>
<td>2.98/8.69 4.32/7.03</td>
</tr>
<tr>
<td></td>
<td>111.7/176.2 63.39%</td>
</tr>
</tbody>
</table>

Table 2: Memory and time used in the algorithm.

Table 3: The performance of the decoder using trigram.

<table>
<thead>
<tr>
<th>WA (%)</th>
<th>RTF</th>
<th># average act. HMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>baseline</td>
<td>85.50</td>
<td>2.80</td>
</tr>
<tr>
<td>compact</td>
<td>85.67</td>
<td>2.82</td>
</tr>
</tbody>
</table>

5. CONCLUSIONS

We have presented an improved algorithm for network compaction using the linear tail property. The algorithm can be efficiently written with a self-structuring subnetwork-based representation, and the experimental results show that it can reorganize the original structures with a memory-efficient way. Also, the algorithm reduced redundancies present in unshared suffixes of pronunciations by aligning linear tails and these aligned shared tails bring additional gains in word accuracy.

6. REFERENCES