On the Compression of Lexicon Transducers

Marco Cognetta, Cyril Allauzen, Michael Riley
{cognetta, allauzen, riley}@google.com
Google

Abstract

In finite-state language processing pipelines, a lexicon is often a key component. It needs to be comprehensive to ensure accuracy, reducing out-of-vocabulary misses. However, in memory-constrained environments (e.g., mobile phones), the size of the component automata must be kept small. Indeed, a delicate balance between comprehensiveness, speed, and memory must be struck to conform to device requirements while providing a good user experience.

In this paper, we describe a compression scheme for lexicons when represented as finite-state transducers. We efficiently encode the graph of the transducer while storing transition labels separately. The graph encoding scheme is based on the LOUDS (Level Order Unary Degree Sequence) tree representation, which has constant time tree traversal for queries while being information-theoretically optimal in space. We find that our encoding is near the theoretical lower bound for such graphs and substantially outperforms more traditional representations in space while remaining competitive in latency benchmarks.

1 Introduction

Modern finite-state language processing pipelines often consist of several finite-state transducers in composition. For example, a virtual keyboard pipeline, used for decoding on mobile devices, can consist of a context dependency transducer $C$, a lexicon $L$, and an $n$-gram language model $G$ (Ouyang et al., 2017). A bikey $C$ transducer is used to encode context in gesture decoding, the lexicon transducer $L$ maps from a character string to the corresponding word ID, and the language model $G$ gives the a priori probability of a word sequence. A similar decomposition is often used in speech recognition decoding (Mohri et al., 1996).

These models are then composed as

$$C \circ L \circ G.$$ 

The application of this combined model to an input character string outputs the corresponding word string and probability. Unfortunately, in order to be accurate, these models may need to be large. This problem is aggravated when the composition is performed statically since the state space grows with the product of the input automata sizes. In practice, on-the-fly composition is often used to save space (Mohri et al., 1996; Hori et al., 2004; Caseiro and Trancoso, 2006). Additionally, it is of practical importance to have compact and efficient finite-state language model component representations.

There are a variety of compression schemes available for automata (Daciuk, 2000). These range from general compression algorithms, which do not depend on a specific underlying structure (Daciuk and van Noord, 2001; Daciuk and Weiss, 2011; Mohri et al., 2015) to schemes that try to heavily exploit specific structural properties of the inputs (Watanabe et al., 2009; Sorensen and Allauzen, 2011). Another important consideration is whether the automata can be decompressed just for a queried portion or need to be more fully decompressed. Generic compression algorithms often have relatively good compression ratios over a wide class of machines, but they sacrifice speed and space in use since they often do not admit such selective decompression. In contrast, structurally-specific compression algorithms can have an attractive balance between the compression ratio and query performance, but are limited to precise subclasses of machines. In real-time production systems, the latter method often proves more desirable since a user should not have to wait long or waste space when a query is answered.
Among the transducers mentioned above, the context-dependency transducer $C$ can be represented implicitly (in code) and structurally-specific compression algorithms for the $n$-gram language model $G$ have previously been developed (Sørensen and Allauzen, 2011). This leads us to investigate the compression of the lexicon $L$.

This paper is organized as follows. Section 2 introduces the formal algebraic structures and notation that we will use. Section 3 describes different representations for these algebraic structures. In Section 4, we formally define a lexicon and explore its possible representations. Section 5 develops an information-theoretic bound on the number of bits needed to encode a lexicon, Section 6 presents our encoding, and Section 7 presents experiments on the quality of that encoding. Finally, we offer concluding remarks in Section 8.

2 Preliminaries

2.1 Graphs and Trees

A directed graph (or digraph) $G = (V, A)$ has a finite set of nodes (or vertices) $V$ and a finite set of directed arcs (or edges) $A \subseteq V \times V$. An arc $a = (p[a], n[a])$ spans from a source node $p[a]$ to a destination node $n[a]$. A path $\pi$ is a non-empty list of consecutive arcs $a_1, a_2, \ldots, a_n$ where $p[a_{i+1}] = n[a_i]$. We write $p[\pi] = p[a_1], n[\pi] = n[a_n]$. A cycle is a path $\pi$ with $p[\pi] = n[\pi]$. A digraph is acyclic if it has no cycles. The out-degree of a node $v \in V$ is $|\{w \in V \mid (v, w) \in A\}|$ and the in-degree is $|\{w \in V \mid (w, v) \in A\}|$.

We distinguish several specific digraph cases:

- An out-tree $(V, A, i)$ is an acyclic digraph for which the in-degree of every node is 1 except for the distinguished root node $i \in V$, which has in-degree 0. The nodes with out-degree 0 are called leaves.

- An in-tree $(V, A, f)$ is an acyclic digraph for which the out-degree of every node is 1 except for the distinguished root node $f \in V$, which has out-degree 0. The nodes with in-degree 0 are called leaves.

- A directed bipartite digraph $(V_1 \cup V_2, A)$ partitions the nodes into two disjoint sets $V_1$ and $V_2$ with $A \subseteq (V_1 \times V_2) \cup (V_2 \times V_1)$.

2.2 Finite-State Transducers

A finite-state transducer $T = (\Sigma, \Gamma, Q, E, i, F)$ has a finite input alphabet $\Sigma$, a finite output alphabet $\Gamma$, a finite set of states $Q$, a finite set of transitions $E \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma^* \times Q$, an initial state $i$ and a final set of states $F \subseteq Q$. The symbol $\epsilon$ represents the empty string. A transition $c = (p[e], i[e], o[e], n[e]) \in E$ represents a move from the source state $p[e]$ to the destination state $n[e]$ with the input label $i[e]$ and output label $o[e]$. Associated with any transducer is a directed graph $G(T) = (Q, A)$ where $A = \{(q, q') \in Q \times Q : (q, x, y, q') \in E\}$. Thus, there is a 1:1 correspondence between states and nodes but there may be multiple transitions, with different labelings, that correspond to the same digraph arc. In that case, we say the transition is a digraph multiarc.

A path $\pi = e_1, \ldots, e_n$, a cycle, $p[\pi]$ and $n[\pi]$ are analogously defined to digraphs and define $i[\pi] = i[e_1] \ldots i[e_n]$ and $o[\pi] = o[e_1] \ldots o[e_n]$. $P(q, q')$ denotes the set of all paths in $T$ from state $q$ to $q'$. We extend this to sets in the obvious way: $P(q, R)$ denotes the set of all paths from state $q$ to $q' \in R$ and so forth. A path $\pi$ is successful if it is in $P(i, F)$ and in that case the transducer is said to accept the input string $i[\pi]$ and output $o[\pi]$.

A finite-state transducer is subsequential if it is input deterministic, that is, no two outgoing transitions at the same state share the same input label, and the destination state of any epsilon transition is a final state with no outgoing transitions.

3 Representations

3.1 Graph and Tree Representations

Basic Graph and Tree Representation. A simple digraph representation uses adjacency lists: denote the nodes $V$ by integers from 1 to $N$, let $a$ be an array indexed by the node number, and let $a[q] = (q_1, \ldots, q_n)$ be a list of the nodes $\{q_j \in V : (q, q_j) \in A\}$. An in-tree and out-tree can use this representation where a distinguished integer such as 1 or $|V|$ is used to denote the root. A directed bipartite graph can also use this representation where it may be convenient to number the nodes in $V_1$ from 1 to $|V_1|$ and $V_2$ from $|V_1| + 1$ to $|V|$.

Compact Tree Representation. In the case of trees, there is a particularly compact representation known as LOUDS (Level Order Unary Degree Sequence). We can quantify compactness as follows.

For a finite set with $M$ elements, we require at least $N = \log M$ bits to uniquely encode each el-
ement. We call an encoding scheme succinct if it takes at most \( N + o(N) \) bits to encode any element uniquely.

The LOUDS tree encoding is a succinct representation of ordinal trees (where a node’s children have a total ordering). Given an ordinal tree of \( N \) nodes, it encodes it in \( 2N + 1 \) bits, while the information-theoretic lower bound is \( 2N - O(\log N) \) (Jacobson, 1989). Moreover, \( O(1) \) time parent-child traversals can be implemented using \( o(N) \) extras bits of storage (Geary et al., 2004).

Let \( b \) be a bitstring where \( b[i] \) is the element at index \( i \) when starting from 0. Then, we define \( \text{Rank}_x \) and \( \text{Select}_x \), where \( x \in \{0, 1\} \), as

\[
\text{Rank}_x(n) = |\{i \mid b[i] = x, 0 \leq i < n\}|
\]

\[
\text{Select}_x(n) = \text{the index of the n-th } x \text{ in } b.
\]

These operations can be performed in constant time using \( o(|b|) \) extra bits of space (Vigna, 2008).

The LOUDS encoding is then constructed as follows. We start with the bitstring 10. Then, from the root in breadth-first order, we append 1*0, where \( d \) is the number of children of the current node. Here, we assume the graph is labeled in breadth-first order. Then, a node \( n \) corresponds to the \( n \)-th 1 in the bitstring (or, equivalently, the \((n+1)\)-th 0). We can find the parent or first/last child (if any) using a combination of \( \text{Rank} \) and \( \text{Select} \) queries:

\[
\text{Parent}^b(n) = \text{Rank}^b_0(\text{Select}^b_1(n))
\]

\[
\text{FirstChild}^b(n) = \text{Rank}^b_1(\text{Select}^b_2(n) + 1)
\]

\[
\text{LastChild}^b(n) = \text{Rank}^b_1(\text{Select}^b_2(n + 1) - 1).
\]

From these, we can retrieve the number of children of a node, the \( i \)-th child, whether or not a node is a leaf, and many other operations in constant time (Geary et al., 2004; Delpratt et al., 2006). It is known that \( \text{Select} \) and \( \text{Rank} \) can be performed in constant time in the length of the bitstring by augmenting the bitstring with \( o(N) \) additional bits of information (thus retaining any succinctness properties) (Kim et al., 2005; Vigna, 2008).

### 3.2 Transducer Representations

**Basic Transducer Representation.** A simple transducer representation uses adjacency lists as well, stored in an array \( a \) indexed by states that are denoted by integers from 0 to \( |Q| - 1 \). The value \( a[q] = ((i_1, o_1, q_1), \ldots, (i_n, o_n, q_n)) \) is a list of the elements of \( \{(i_j, o_j, q_j) \in \Sigma \times \Gamma \times Q : (q, i_j, o_j, q_j) \in E\} \). The initial state can be denoted by 0 and the final states can be stored separately. We will call this representation \( \text{AdjList} \) in our experiments where we use 32 bits for each of the input label, output label, and destination state of each transition.

**Compact Transducer Representation.** A more compact transducer representation stores the \( |Q| \) adjacency lists across 2 global arrays as follows. First an array \( \mathcal{I} \), indexed by integers from 0 to \( |Q| \), holds the values \( \mathcal{I}[q] = \sum_{0 \leq i < q} |a[q]| \). Second an array \( \mathcal{A} \), indexed by integers from 0 to \( |E| - 1 \), holds the concatenation of the adjacency lists \( a[0] \cdots a[|Q| - 1] \). The adjacency list for a given state \( q \) can be recovered from \( \mathcal{I} \) and \( \mathcal{A} \) as

\[
a[q] = \bigcup_{i=\mathcal{I}[q]}^{\mathcal{I}[q+1]-1} \{\mathcal{A}[i]\}.
\]

Observe that \( I \) stores a monotonic nondecreasing sequence of integers, hence we encode using a differential coding approach similar to PForDelta (Zukowski et al., 2006). We store \( A \) using a variable-length encoding that ensures that \( \log |Q| + \log |\Sigma| + \log |\Gamma| \) bits are used per entry in \( A \) on average. Final states are stored as superfinal transitions. We will call this representation CmpAdjList in our experiments.

### 4 Lexicons

**Lexicon Definition.** We define a lexicon as a finite binary relation \( L \subset \Sigma^+ \times \Gamma \) that pairs non-empty character strings from the finite alphabet \( \Sigma \) to a word symbol in the finite alphabet \( \Gamma \). This terminology matches our keyboard application described above. For the speech recognition application, the \( \Sigma \) alphabet represents phonemes. We will assume the relation \( L \) is functional and one-to-one. In other words, each character string in the domain of \( L \) maps to only one word (i.e., no homonyms) and each word maps to only one character string (i.e., unique spellings). This is natural for the keyboard application.\(^1\)

**Lexicon Representation** While there are many ways to represent a lexicon, we focus on using a character-to-word finite-state transducer. An advantage of this approach is that we can use trans-
In other words, the prefix graph corresponds to transitions on paths in \( T \) before the output label, the suffix graph to those after the output label, and the bridge graph to those with the output label. It is easy to see a transition in \( T \) corresponds to an arc in exactly one of these sub-graphs. Further, \( Q_p \) and \( Q_s \) partition \( Q \).

The prefix graph is an out-tree rooted at \( i \in Q_p \). Suppose there are two arcs entering some state \( q \in Q_p \). Then there must be two successful paths in \( T \) that pass through \( q \) with the same word label, which is a contradiction.

Similarly, the suffix graph is an in-tree rooted at \( f \in Q_s \). For example, suppose there are two arcs leaving some state \( q \in Q_s \). Then again there must be two successful paths in \( T \) that pass through \( q \) with the same word label, which is a contradiction.

Finally, the bridge graph is a directed bipartite graph with arcs that span from \( Q_s \) to \( Q_p \) because for any successful path in \( T \) the transition with a non-\( \epsilon \) output label is preceded by a subpath with all \( \epsilon \) output labels from the initial state \( i \) and followed by a subpath with all \( \epsilon \) output labels to the final state \( f \). Observe that only bridge arcs in \( A_b \) can be multiarcs of \( G(T) \) since \( L \) is one-to-one.

Figure 1 shows this decomposition for our example with the bridge arcs specially marked.

5 The Optimal Graph Encoding

Now that we have described the canonical form of our lexicon transducer and its graph decomposition, we can begin to devise a compression scheme. We first wish to find the information-theoretic bound on the number of bits required...
to uniquely encode any lexicon graph. That is, among all lexicon transducers with given prefix out-tree and suffix in-tree sizes (and a given number of leaves in each) and \( k \) bridge arcs, how many bits is sufficient to encode them so that they are all pairwise distinguishable?

In this section, we let \( n \) and \( n_\ell \) be the number of nodes and leaves in the prefix out-tree and \( m \) and \( m_\ell \) be the same for the suffix in-tree.

The LOUDS tree encoding is optimal for all \( n \) node ordinal trees up to lower order terms (Jacobson, 1989). This is because there are 
\[
\frac{(2n)}{n}
\]
ordinal trees on \( n \) nodes, and
\[
\log\left(\frac{(2n)}{n}\right) = 2n - O(\log n).
\]
This is compared to the \( 2n + 1 \) bits used by LOUDS. However, when the number of leaves is known, this bound can be reduced. There are 
\[
\frac{(n-2)}{n_\ell(n_\ell-1)}
\]
ordinal trees with \( n \) nodes and \( n_\ell \) leaves (Yamanaka et al., 2012).

We are left with the task of counting the number of valid bridge graphs with \( k \) arcs. Each bridge graph is uniquely defined by choosing a set of \( k \) bridge arcs, i.e., a \( k \) element subset of \( Q_p \times Q_s \). Every state in a minimal lexicon transducer must belong to a successful path, hence every node in its graph must belong to a path from the root of \( Q_p \) to the root of \( Q_s \). A leaf in \( Q_p \) (resp. \( Q_s \)) belongs to such a path if and only if it is the origin (resp. destination) of a bridge arc. Hence, a set of \( k \) bridge arcs \( A_b \in \mathcal{P}_k(Q_p \times Q_s) \) is valid iff for every leaf \( q \) there exists a bridge arc \( a \in A_b \) such that \( q = p[a] \) or \( q = n[a] \). Let \( Q^*_p \) and \( Q^*_s \) be the set of leaves in the prefix and suffix graphs respectively, and \( Q^\ell = Q^\ell_p \cup Q^\ell_s \).

Let \( A_q \) denote the set of sets of \( k \) bridge arcs where the leaf \( q \in Q^\ell \) is not part of an arc:
\[
A_q = \begin{cases} \mathcal{P}_k((Q_p \setminus \{q\}) \times Q_s) & \text{if } q \in Q^\ell_p \\ \mathcal{P}_k(Q_p \times (Q_s \setminus \{q\})) & \text{otherwise.} \end{cases}
\]
A set of bridge arcs is valid if and only if it does not belong to any \( A_q \). Hence, the number of valid sets of bridge arcs is
\[
\big| \mathcal{P}_k(Q_p \times Q_s) \setminus \bigcup_{q \in Q^\ell} A_q \big| = \binom{mn}{k} - \bigg| \bigcup_{q \in Q^\ell} A_q \bigg|.
\]
We can now apply the inclusion-exclusion principle to compute the cardinality of the union in that last term:
\[
\bigg| \bigcup_{q \in Q^\ell} A_q \bigg| = \sum_{\emptyset \neq X \subseteq Q^\ell} (-1)^{|X|+1} \bigg| \bigg( \bigcup_{x \in X} A_x \bigg) \bigg|.
\]
Observe that, for a non-empty subset \( X \) of \( Q^\ell \),
\[
\bigg| \bigcap_{x \in X} A_x \bigg| = \binom{n-i}{k}(m-j)(n-i)(m-j)
\]
and the cardinality of that intersection is:
\[
\bigg| \bigcup_{x \in X} A_x \bigg| = \sum_{i=0}^{n_\ell} \sum_{j=0}^{m_\ell} (-1)^{i+j+1} \sum_{|X| \subseteq Q^\ell, |X| = i} \sum_{|X \cap Q^\ell_p| = j} \bigg| \bigcap_{x \in X} A_x \bigg| = \binom{n_\ell}{i} \binom{m_\ell}{j} (n-i)(m-j).
\]
The last derivation following from
\[
\bigg| \bigg\{ X \subseteq Q^\ell \bigg| i = |X \cap Q^\ell_p|, \ j = |X \cap Q^\ell_s| \bigg\} \bigg| = \binom{n_\ell}{i} \binom{m_\ell}{j}.
\]
We can now complete the computation of the number of valid bridge graphs:
\[
\binom{mn}{k} - \sum_{i=0}^{n_\ell} \sum_{j=0}^{m_\ell} (-1)^{i+j+1} \binom{m_i}{j} \binom{n-i}{k}(m-j)(n-i)(m-j) = \sum_{i=0}^{n_\ell} \sum_{j=0}^{m_\ell} (-1)^{i+j} \binom{n_\ell}{i} \binom{m_\ell}{j} (n-i)(m-j).
\]
We are unaware of any asymptotic analysis of this summation or a way to closely estimate its logarithm. To compare it with our encoding, we use a loose upper bound of \( \binom{nm}{k} \) and Stirling’s approximation to get
\[
\log\binom{nm}{k} \approx nm \log nm - k \log k - (nm - k) \log(nm - k).
\]
Overall, the number of possible lexicon graphs given \( n, m, k, n_\ell, \) and \( m_\ell \) can be found by multiplying the number of \( n \) (\( m \)) node, \( n_\ell \) (\( m_\ell \)) leaf trees
\[
\frac{(n-2)}{(m-1)} \frac{(n-1)}{(m-1)} \frac{(m-2)}{(m_\ell-1)} \frac{(m-1)}{(m_\ell-1)}
\]
by the number of valid bridge graphs.

Finally, we note that by choosing any out-tree as a prefix graph, any in-tree as a suffix graph, and any valid bridge graph, we obtain a graph that is a valid lexicon graph. A minimal lexicon transducer can be derived from that graph by labeling each non-bridge arc with a unique input label (and epsilon output) and each bridge arc with a unique input and output label.

6 Compact Lexicon Encoding
6.1 Encoding the Graph

We encode the prefix, suffix, and bridge graphs separately. Encoding the prefix out-tree and suffix in-tree using LOUDS leads to a natural numbering separately from the arc label data. Encoding fixed node graphs into \( A \) bits, one for each prefix node and bridge arc, to index in the dense array \( A \) holding the concatenation of the bridge-arc adjacency lists of the prefix nodes \( \{q', q \mid (q, q') \in A_s\} \), with \( 0 \) and \( n \) denoting the roots of \( Q_p \) and \( Q_s \), respectively. The LOUDS representation of the prefix and suffix graphs consists of two bitstrings, \( b_p \) of length \( n+1 \) and \( b_s \) of length \( m+1 \), using \( 2(n+m+1) \) bits combined.

We represent the bridge graph using a compact adjacency list approach. We use an array \( A_b \) indexed from \( 0 \) to \( n-1 \) holding the concatenation of the bridge-arc adjacency lists of the prefix nodes \( a_{\ell}[n-1] \). We use a bitmap \( b_\ell \) with \( n+k \) bits, one for each prefix node and bridge arc, to implement an index into \( A_b \) as follows. The bitmap \( b_\ell \) is encoded by concatenating \( 1^{d\ell}0 \) for each prefix node \( q \), where
\[
d = \sum_{q' \in Q_s} |\{q' \in Q_s \mid (q, q') \in A_b\}| = |a[q]|.
\]

We retrieve the number of bridge arcs originating at a node \( q \in Q_p \) by computing
\[
N_b(q) = Select_{0}^{b_p}(q) - Select_{0}^{b_p}(q-1),
\]
and the index in the dense array \( A_b \) to the position where the adjacency list for \( q \) starts by
\[
I_b(q) = \text{Rank}_1^{b_\ell}(\text{Select}_0^{b_\ell}(q) - 1).
\]
The variable-length encoding mentioned in Section 3.2 is used to compress \( A_b \) in \( k \log m \) bits, since the \( k \) entries in \( A_b \) can take at most \( m \) values.

It is possible to reduce the bridge arc adjacency list and multiplicity encoding to
\[
\min(n+k \log m, m+k \log n) + k + 1
\]
bits by noting that the bridge arcs travel unidirectionally from the prefix out-tree to suffix in-tree so we can represent them in either the forward or reverse direction, depending on which uses less space. However, we choose not to do this as it would incur an additional traversal time cost.

In total, our encoding uses
\[
2(n + m + 1) + n + k + k \log m
\]
bits to store the graph. We note that this is asymptotically worse than the best possible from Section 5. Nevertheless, in Section 7, we show empirically that it performs substantially better than the CmpAdjList format and is useful in practice.

6.2 Encoding the Labeling

We now encode the arc labels for each of the three component graphs using four ancillary arrays.

The arrays \( L_p \) and \( L_s \) store the input labels for each of the \( n-1 \) prefix arcs and \( m-1 \) suffix arcs. For \( q \in Q_p \cup \{0\} \), \( L_p[q-1] \) holds the input label for the unique incoming prefix arc to \( q \). Likewise, for \( q \in Q_s \cup \{n\} \), \( L_s[q-n-1] \) holds the input label for the unique outgoing suffix arc to \( q \). Recall that arcs in the prefix out-tree or suffix in-tree always have output label \( \epsilon \).

The arrays \( L_{p'} \) and \( L_{s'} \) store the input and output label for each of the \( k \) bridge arcs, using the same indexing as \( A_b \); the bridge arc corresponding to the \( j \)-th entry in \( A_b \), has input label \( L_{p'}[j] \), output label \( L_{s'}[j] \) and destination \( A_b[j] \).

Each of the arrays \( L_p \), \( L_s \), \( L_{p'} \) and \( L_{s'} \) is compressed using the same variable-length encoding scheme as CmpAdjList. This allows us to directly compare the effect of encoding the graph separately from the arc label data. Encoding finality is simple — only one node, the root of the suffix in-tree, is final.
An overview of the memory layout of our encoding is given in Figure 2. We discuss the practical space savings in Section 7. All together, we call this representation the LOUDS lexicon format in our experiments.

6.3 Traversing the Transducer

We traverse the transducer by constructing the transitions originating at a given state \( q \) on demand.

When \( q \in Q_p \), the set \( E[q] \) of outgoing transitions in \( q \) can be decomposed as

\[
E[q] = E_p[q] \cup E_b[q]
\]

where \( E_p[q] \) represents the transitions corresponding to prefix arcs and \( E_b[q] \) the ones corresponding to bridge arcs. The first component can be computed from the prefix LOUDS tree by

\[
E_p[q] = \bigcup_{q' = \text{FirstChild}\,^p(q)} \{(q, L_p[q'-1], \epsilon, q')\}
\]

and the second component can be recovered from the compact adjacency representation of the bridge graph as

\[
E_b[q] = \bigcup_{j = I_a(q)} \{(q, L_b[j], L_b[j], A_b[j])\}.
\]

When \( q \in Q_s \setminus \{n\} \), there is a single outgoing transition in \( q \) that can be computed from the suffix LOUDS tree as

\[
(q, L_s[q-n-1], \epsilon, n + \text{Parent}\,^s(q-n)).
\]

Finally, when \( q = n \), \( q \) is the root of the suffix out-tree. There are no outgoing transitions in \( q \) but \( q \) is final.

6.4 Closure

In practice, we often use a modified lexicon transducer representing its closure \( T^+ \), which accepts one or more words from the lexicon. For this, an \( \epsilon \)-labeled transition from the final state to the initial state can be added to the canonical transducer.\(^4\)

7 Experiments

We compare our lexicon encoding to the two other transducer representations in Section 3.2. We measure the memory size of the resulting machines as well as their runtimes on a decoding task. We prepare a set of lexicons using the most common 500k words in the Google keyboard (GBoard) Russian language model. We extract the 50k, 100k, \ldots, 500k most frequent words to create a total of 10 lexicons.

We first compare the space used by the AdjList, the CmpAdjList, and the LOUDS lexicon formats. The results are shown in Figure 3. The LOUDS lexicon outperforms the other two formats in every case. On the 500k word lexicon, it is 90.8% smaller than the AdjList format and 58.8% smaller than the CmpAdjList format.

Figure 4 shows the number of bits required to encode the Russian lexicons using our encoding and the upper bound of the optimal encoding. We use the parameters from Table 1 along with the upper bound described in Section 5 and the number of bits for our representation from Section 6. Our graph encoding nearly matches the upper bound approximation in all situations. For the 500k lexicon, the difference between our encoding and the upper bound is less than 2%. In contrast, the standard adjacency list format graph requires ten times more space across all test cases. We now consider the performance of our encoding on a benchmark decoding task consisting of on-the-fly composition with an \( n \)-gram language model followed by shortest path computation, which simulates a typical pipeline in applications. For the language model, we use a 244k state \( n \)-gram model trained

\(^4\)In the keyboard example, this transition might instead be labeled with the space symbol on input.
Table 1: The size of the prefix out-tree and suffix in-tree as well as the number of leaves in each for all of the Russian lexicons. Note that the number of bridge arcs is the same as the number of words.

| # Words | 50k | 100k | 150k | 200k | 250k | 300k | 350k | 400k | 450k | 500k |
|---------|-----|------|------|------|------|------|------|------|------|------|
| Prefix Nodes | 76207 | 148026 | 219494 | 292499 | 360911 | 429080 | 494766 | 558619 | 620429 | 670232 |
| Suffix Nodes | 7867 | 12548 | 15964 | 18187 | 20634 | 22454 | 23850 | 25059 | 25955 | 26977 |
| Prefix Leaves | 13402 | 26371 | 39300 | 52288 | 64894 | 77462 | 89881 | 102067 | 114228 | 124097 |
| Suffix Leaves | 1602 | 2560 | 3266 | 3732 | 4182 | 4512 | 4754 | 4989 | 5221 | 5421 |

Figure 3: The amount of disk space required by each of the FST formats on our Russian language lexicon test set.

Figure 4: The theoretical number of bits required to encode the graphs of the Russian lexicons. We compare our encoding’s exact space requirements with the upper bound as discussed in Section 5.

Figure 5: The time for the decoding task with a 244k state n-gram model. The average over 50 trials is given.

8 Conclusion

In this paper, we described a compact encoding for character-to-word lexicon transducers in canonical minimal form. The transducer graph is decomposed into simpler subgraphs, exploited in the encoding. The arc label data is encoded separately using variable-length compression schemes. We presented an information-theoretic lower bound for the graph encoding and compare the encoding to an asymptotic upper bound approximation.

Our encoding is compared to two alternative formats – adjacency lists with and without variable length compression. Ours is more than 58% smaller while being only ~9% slower in tests on a decoding benchmark. Furthermore, this encoding is very close to the information-theoretic upper bound on all the test cases.

References

Diamantino Caseiro and Isabel Trancoso. 2006. A specialized on-the-fly algorithm for lexicon and language model composition. *IEEE Trans. Audio, Speech & Language Processing*, 14(4):1281–1291.

Jan Daciuk. 2000. Experiments with automata compression. In *Implementation and Application of Automata, 5th International Conference, CIAA 2000*, 124097.
London, Ontario, Canada, July 24-25, 2000, Revised Papers, pages 105–112.

Jan Daciuk and Gertjan van Noord. 2001. Finite automata for compact representation of language models in NLP. In Implementation and Application of Automata, 6th International Conference, CIAA 2001, Pretoria, South Africa, July 23-25, 2001, Revised Papers, pages 65–73.

Jan Daciuk and Dawid Weiss. 2011. Smaller representation of finite state automata. In Implementation and Application of Automata - 16th International Conference, CIAA 2011, Blois, France, July 13-16, 2011. Proceedings, pages 118–129.

O’Neil Delpratt, Naila Rahman, and Rajeev Raman. 2006. Engineering the LOUDS succinct tree representation. In Experimental Algorithms, 5th International Workshop, WEA 2006, Cala Galdana, Menorca, Spain, May 24-27, 2006, Proceedings, pages 134–145.

Richard F. Geary, Rajeev Raman, and Venkatesh Raman. 2004. Succinct ordinal trees with level-ancestor queries. In Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2004, New Orleans, Louisiana, USA, January 11-14, 2004, pages 1–10.

Takaaki Hori, Chiori Hori, and Yasuhiro Minami. 2004. Fast on-the-fly composition for weighted finite-state transducers in 1.8 million-word vocabulary continuous speech recognition. In INTERSPEECH 2004 – ICSLP, 8th International Conference on Spoken Language Processing, Jeju Island, Korea, October 4-8, 2004.

Guy Jacobson. 1989. Space-efficient static trees and graphs. In 30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, USA, 30 October - 1 November 1989, pages 549–554.

Dong Kyue Kim, Joong Chae Na, Ji Eun Kim, and Kunsoo Park. 2005. Efficient implementation of rank and select functions for succinct representation. In Experimental and Efficient Algorithms, 4th International Workshop, WEA 2005, Santorini Island, Greece, May 10-13, 2005. Proceedings, pages 315–327.

Mehryar Mohri. 2000. Minimization algorithms for sequential transducers. Theoretical Computer Science, 234:177–201.

Mehryar Mohri, Fernando Pereira, and Michael Riley. 2002. Weighted finite-state transducers in speech recognition. Computer Speech & Language, 16(1):69–88.

Mehryar Mohri, Fernando C. N. Pereira, and Michael Riley. 1996. Weighted automata in text and speech processing. In ECAI 96, 12th European Conference on Artificial Intelligence, Workshop on Extended Finite State Models of Language. John Wiley & Sons.