Compact, Efficient and Unlimited Capacity: Language Modeling with Compressed Suffix Trees

Ehsan Shareghi,♭ Matthias Petri,♮ Gholamreza Haffari♭ and Trevor Cohn♭

Faculty of Information Technology, Monash University
Computing and Information Systems, The University of Melbourne
first.last@{monash.edu,unimelb.edu.au}

Abstract
Efficient methods for storing and querying language models are critical for scaling to large corpora and high Markov orders. In this paper we propose methods for modeling extremely large corpora without imposing a Markov condition. At its core, our approach uses a succinct index—a compressed suffix tree—which provides near optimal compression while supporting efficient search. We present algorithms for on-the-fly computation of probabilities under a Kneser-Ney language model. Our technique is exact and although slower than leading LM toolkits, it shows promising scaling properties, which we demonstrate through $\infty$-order modeling over the full Wikipedia collection.

1 Introduction
Language models (LMs) are critical components in many modern NLP systems, including machine translation (Koehn, 2010) and automatic speech recognition (Rabiner and Juang, 1993). The most widely used LMs are $m$-gram models (Chen and Goodman, 1996), based on explicit storage of $m$-grams and their counts, which have proved highly accurate when trained on large datasets. To be useful, LMs need to be not only accurate but also fast and compact.

Depending on the order and the training corpus size, a typical $m$-gram LM may contain as many as several hundred billions of $m$-grams (Brants et al., 2007), raising challenges of efficient storage and retrieval. As always, there is a trade-off between accuracy, space, and time, with recent papers considering small but approximate lossy LMs (Chazelle et al., 2004; Talbot and Osborne, 2007; Guthrie and Hepple, 2010), or loss-less LMs backed by tries (Stolcke et al., 2011), or related compressed structures (Germann et al., 2009; Heafield, 2011; Pauls and Klein, 2011; Sorensen and Allauzen, 2011; Watanabe et al., 2009). However, none of these approaches scale well to very high-order $m$ or very large corpora, due to their high memory and time requirements. An important exception is Kennington et al. (2012), who also propose a language model based on a suffix tree which scales well with $m$ but poorly with the corpus size (requiring memory of about 20× the training corpus).

In contrast, we make use of recent advances in compressed suffix trees (CSTs) (Sadakane, 2007) to build compact indices with much more modest memory requirements (∼ the size of the corpus). We present methods for extracting frequency and unique context count statistics for $m$-gram queries from CSTs, and two algorithms for computing Kneser-Ney LM probabilities on the fly using these statistics. The first method uses two CSTs (over the corpus and the reversed corpus), which allow for efficient computation of the number of unique contexts to the left and right of an $m$-gram, but is inefficient in several ways, most notably when computing the number of unique contexts to both sides. Our second method addresses this problem using a single CST backed by a wavelet tree based FM-index (Ferragina et al., 2007), which results in better time complexity and considerably faster runtime performance.

Our experiments show that our method is practical for large-scale language modeling, although querying is substantially slower than a SRLM benchmark. However our technique scales much more gracefully with Markov order $m$, allowing unbounded ‘non-Markov’ application, and enables training on large corpora as we demonstrate on the complete Wikipedia dump. Overall this paper illustrates the vast potential succinct indexes have

1For the implementation see: https://github.com/eehsan/lm-sdsl.
for language modelling and other ‘big data’ problems in language processing.

2 Background

Suffix Arrays and Suffix Trees

Let $T$ be a string of size $n$ drawn from an alphabet $\Sigma$ of size $\sigma$. Let $T[i...n] = T[i..n-1]$ be a suffix of $T$. The suffix tree (Weiner, 1973) of $T$ is the compact labeled tree of $n + 1$ leaves where the root to leaf paths correspond to all suffixes of $T \Sigma$, where $\Sigma$ is a terminating symbol not in $\Sigma$. The path-label of each node $v$ corresponds to the concatenation of edge labels from the root node to $v$. The node depth of $v$ corresponds to the number of ancestors in the tree, whereas the string depth of $v$ corresponds to the length of the path-label. Searching for a pattern $\alpha$ of size $m$ in $T$ translates to finding the locus node $v$ closest to the root such that $\alpha$ is a prefix of the path-label of $v$ in $O(m)$ time. We refer to this approach as forward search. Figure 1a shows a suffix tree over a sample text. A suffix tree requires $O(n)$ space and can be constructed in $O(n)$ time (Ukkonen, 1995). The children of each node in the suffix tree are lexicographically ordered by their edge labels. The $i$-th smallest suffix in $T$ corresponds to the path-label of the $i$-th leaf. The starting position of the suffix can be associated its corresponding node in the tree as shown in Figure 1a. All occurrences of $\alpha$ in $T$ can be retrieved by visiting all leaves in the subtree of the locus of $\alpha$. For example, pattern “the night” occurs at positions 12 and 19 in the sample text. We further refer the number of children of a node $v$ as its degree and the number of leaves in the subtree rooted at $v$ as the size of $v$.

The suffix array (Manber and Myers, 1993) of $T$ is an array $SA[0...n-1]$ such that $SA[i]$ corresponds to the starting position of the $i$-th smallest suffix in $T$ or the $i$-th leaf in the suffix tree of $T$. The suffix array requires $n \log n$ bits of space and can also be constructed in $O(n)$ time (Kärkkäinen et al., 2006). Using only the suffix array and the text, pattern search can be performed using binary search in $O(m \log n)$ time. For example, the pattern “the night” is found by performing binary search using $SA$ and $T$ to determine $SA[18, 19]$, the interval in $SA$ corresponding to the suffixes in $T$ prefixed by the pattern. In practice, suffix arrays use $4 \sim 8n$ bytes of space whereas the most efficient suffix tree implementations require at least $20n$ bytes of space (Kurtz, 1999) which are both much larger than $T$ and prohibit the use of these structures for all but small data sets.

Compressed Suffix Structures

Reducing the space usage of suffix based index structure has recently become an active area of research. The space usage of a suffix array can be reduced significantly by utilizing the compressibility of text combined with succinct data structures. A succinct data structure provides the same functionality as an equivalent uncompressed data structure, but requires only space equivalent to the information-theoretic lower bound of the underlying data. For simplicity, we focus on the FM-Index which emulates the functionality of a suffix array over $T$ using $nH_k(T) + o(n \log \sigma)$ bits of space where $H_k$ refers to the $k$-th order entropy of the text (Ferragina et al., 2007). In practice, the FM-Index of $T$ uses roughly space equivalent to the compressed representation of $T$ using a standard compressor such as bzip2. For a more comprehensive overview on succinct text indexes, see the excellent survey of Ferragina et al. (2008).

The FM-Index relies on the duality between the suffix array and the BWT (Burrows and Wheeler, 1994), a permutation of the text such that $T^\text{bwt}[i] = T[SA[i] - 1]$ (see Figure 1). Searching for a pattern using the FM-Index is performed in reverse order by performing $\text{RANK}(T^\text{bwt}[i], c)$ operations $O(m)$ times. Here, $\text{RANK}(T^\text{bwt}[i], c)$ counts the number of times symbol $c$ occurs in $T^\text{bwt}[0...i-1]$. This process is usually referred to as backward search. Let $SA[l_i, r_i]$ be the interval corresponding to the suffixes in $T$ matching $\alpha[i...m-1]$. By definition of the BWT, $T^\text{bwt}[l_i, r_i]$ corresponds to the symbols in $T$ preceding $\alpha[i...m-1]$ in $T$. Due to the lexicographical ordering of all suffixes in $SA$, the interval $SA[l_{i-1}, r_{i-1}]$ corresponding to all occurrences of $\alpha[i-1...m-1]$ can be determined by computing the rank of all occurrences of $c = \alpha[i-1]$ in $T^\text{bwt}[l_i, r_i]$. Thus, we compute $\text{RANK}(T^\text{bwt}[l_i, c])$, the number of times symbol $c$ occurs before $l_i$ and $\text{RANK}(T^\text{bwt}[r_i + 1, c])$, the number of occurrences of $c$ in $T^\text{bwt}[0, r_i]$. To determine $SA[l_{i-1}, r_{i-1}]$, we additionally store the starting positions $C_s$ of all suffixes for each symbol $s$ in $\Sigma$ at a negligible cost of $\sigma \log n$ bits. Thus, the new interval is computed as $l_{i-1} = C_c + \text{RANK}(T^\text{bwt}, l_i, c)$ and $r_{i-1} = C_c + \text{RANK}(T^\text{bwt}, r_i + 1, c)$.

The time and space complexity of the FM-index thus depends on the cost of storing and pre-
processing $T^{\text{bwt}}$ to answer RANK efficiently. A wavelet tree can be used to answer RANK over $T^{\text{bwt}}$ in $O(\log \sigma)$ time. The wavelet tree reduces RANK over an alphabet $\Sigma$ into multiple RANK operations over a binary alphabet which can be answered in $O(1)$ time and $o(n)$ bits extra space by periodically storing absolute and relative RANK counts (Munro, 1996). The alphabet is reduced by recursively splitting symbols based on their code words into subgroups to form a binary tree as shown in Figure 1b for $T^{\text{bwt}}$. To answer $\text{RANK}(T^{\text{bwt}}, i, c)$, the tree is traversed based on the code word of $c$, performing binary RANK at each level. For example, $\text{RANK}(T^{\text{bwt}}, 17, 't')$ translates to performing $\text{RANK}(WT_{\text{root}}, 17, 1) = 12$ on the top level of the wavelet tree, as $t=\text{the}=110$. We recurse to the right subtree of the root node and compute $\text{RANK}(WT_1, 12, 1)$ as there were 12 ones in the root node and the next bit in the code-word of ‘the’ is also one. This process continues until the correct leaf node is reached to answer $\text{RANK}(T^{\text{bwt}}, 17, 't') = 5$ in $O(\log \sigma)$ time. The space usage of a regular wavelet tree is $n \log \sigma + o(n \log \sigma)$ bits which roughly matches the size of the text.\(^2\) If locations of matches are required, ad-

\(^2\)However, if code-words for each symbol are chosen based on their Huffman-codes the size of the wavelet tree

conditional space is needed to access $\text{SA}[i]$ or the inverse suffix array $\text{SA}^{-1}[\text{SA}[i]] = i$. In the simplest scheme, both values are periodically sampled using a given sample rate $\text{SAS}$ (e.g. 32) such that $\text{SA}[i] \pmod{\text{SAS}} = 0$. Then, for any $\text{SA}[i]$ or $\text{SA}^{-1}[i]$, at most $O(\text{SAS})$ RANK operations on $T^{\text{bwt}}$ are required to access the value. Different sample rates, bitvector implementations and wavelet tree types result in a wide variety of time-space tradeoffs which can be explored in practice (Gog et al., 2014).

In the same way the FM-index emulates the functionality of the suffix array in little space, compressed suffix trees (CST) provide the functionality of suffix trees while requiring significantly less space than their uncompressed counterparts (Sadakane, 2007). A CST uses a compressed suffix array (CSA) such as the FM-Index but stores additional information to represent the shape of the suffix tree as well as information about path-labels. Again a variety of different storage schemes exist, however for simplicity we focus on the CST of Ohlebusch et al. (2010) which we use in our experiments. Here, the shape of the tree is stored using a balanced-parenthesis (BP) sequence which for a tree of $p$ nodes requires $\approx 2p$ reduces to $nH_0(T)(1 + o(1))$ bits which can be further be reduced to to $nH_k(T) + o(n \log \sigma)$ bits by using entropy compressed bitvectors.

![Figure 1: Data structures for the sample text $T$=#the old night keeper keeps the keep in the town# with alphabet $\Sigma$={the, old, night, keeper, keeps, keep, in, town, #} and code words $s=0000, \#=0001, i=in=001, p=keep=010, r=keeper=011, s=keeps=1000, o=old=101, t=the=110, n=night=1011$ and $T=town=111$.](image)
bits. Using little extra space and advanced bit-operations, the BP-sequence can be used to perform operations such as string-depth(v), parent(v) or accessing the i-th leaf can be answered in constant time. To support more advanced operations such as accessing path-labels, the underlying CSA or a compressed version of the LCP array are required which can be more expensive.\(^3\) In practice, a CST requires roughly \(4 - 6\)\(n\) bits in addition to the cost of storing the CSA. For a more extensive overview of CSTs see Russo et al. (2011).

**Kneser Ney Language Modelling** Recall our problem of efficient \(n\)gram language modeling backed by a corpus encoded in a succinct index. Although our method is generally applicable to many LM variants, we focus on the Kneser-Ney LM (Kneser and Ney, 1995), specifically the interpolated variant described in Chen and Goodman (1996), which has been shown to outperform other \(n\)gram LMs and has become the de-facto standard.

Interpolated Kneser-Ney describes the conditional probability of a word \(w_i\) conditioned on the context of \(m - 1\) preceding words, \(w_{i-m+1}^{i-1}\), as

\[
P(w_i | w_{i-m+1}^{i-1}) = \max \left[ \frac{c(w_{i-m+1}) - D_m}{c(w_{i-m+1})}, 0 \right] + \frac{D_m N^{1+}(w_{i-m+1})}{c(w_{i-m+1})} \tilde{P}(w_i | w_{i-m+2}^{i-1}),
\]

where lower-order smoothed probabilities are defined recursively (for \(1 < k < m\)) as

\[
\tilde{P}(w_i | w_{i-k+1}^{i-1}) = \max \left[ \frac{N^{1+}(w_{i-k+1}) - D_k}{N^{1+}(w_{i-k+1})}, 0 \right] + \frac{D_k N^{1+}(w_{i-k+1})}{N^{1+}(w_{i-k+1})} \tilde{P}(w_i | w_{i-k+2}^{i-1}).
\]

In the above formula, \(D_k\) is the \(k\)gram-specific discount parameter, and the occurrence count

\[
N^{1+}(\alpha \cdot) = | \{ w : c(\alpha w) > 0 \} |
\]

is the number of observed word types following the pattern \(\alpha\); the occurrence counts \(N^{1+}(\bullet \alpha)\) and \(N^{1+}(\bullet \alpha \cdot)\) are defined accordingly. The recursion stops at unigram level where the unigram probabilities are defined as \(\tilde{P}(w_i) = N^{1+}(\bullet w_i) / N^{1+}(\bullet \cdot)\).

\(\text{3} \text{ Using CSTs for KN Computation}\)

The key requirements for computing probability under a Kneser-Ney language model are two types of counts: raw frequencies of \(m\)grams and occurrence counts, quantifying how many different contexts the \(m\)gram has occurred in. Figure 2 (right) illustrates the requisite counts for calculating the probability of an example 4-gram. In electing to store the corpus directly in a suffix tree, we need to provide mechanisms for computing these counts based on queries into the suffix tree.

The raw frequency counts are the simplest to compute. First we identify the locus node \(v\) in the suffix tree for the query \(m\)gram; the frequency corresponds to the node’s size, an \(O(1)\) operation which returns the number of leaves below \(v\). To illustrate, consider searching for \(c(\text{the night})\) in Figure 1a, which matches a node with two leaves (labelled 19 and 12), and thus \(c = 2\).

More problematic are the occurrence counts, which come in several flavours: right contexts, \(N^{1+}(\alpha \cdot)\), left contexts, \(N^{1+}(\bullet \alpha)\), and contexts to both sides of the pattern, \(N^{1+}(\bullet \alpha \cdot)\). The first of these can be handled easily, as

\[
N^{1+}(\alpha \cdot) = \begin{cases} \text{degree} \(v\), & \text{if } \alpha = \text{label} \(v\) \\ 1, & \text{otherwise} \end{cases}
\]

where \(v\) is the node matching \(\alpha\), and label \((v)\) denotes the path-label of \(v\).\(^5\) For example, keep in has two child nodes in Figure 1a, and thus there are two unique contexts in which it can occur, \(N^{1+}(\text{keep} \bullet) = 2\), while the keep partially matches an edge in the forward suffix tree in Figure 1a as it can only be followed by in, \(N^{1+}(\text{the keep} \bullet) = 1\). A similar line of reasoning applies to computing \(N^{1+}(\bullet \alpha)\). Assuming we also have a second suffix tree representing the reversed corpus, we first identify the reversed pattern (e.g., in \(\text{keep}_R\)) and then use above method to compute the occurrence count (denoted hereafter \(\text{N1P}(t, v, \alpha)\)), where \(t\) is the CST.

\(\text{Supplementary Materials}\) Table 1 for an overview of the complexities of the functionality of the CST that is used in our experiments.

\(\text{Modified Kneser-Ney, proposed by Chen and Goodman (1996), typically outperforms interpolated Kneser-Ney through its use of context-specific discount parameters. The}\)

\(\text{implementation of this with our data structures is straightforward in principle, but brings a few added complexities in terms of dynamic computing other types of occurrence counts, which we leave for future work.}\)

\(\text{See the Supplementary Materials for the explicit algorithm, but note there are some corner cases involving sentinels \# and $, which must be excluded when computing occurrence counts. Such tests have been omitted from the presentation for clarity.}\)

\(\text{In the presented algorithms, we overload the pattern argument in function calls for readability, and use • to denote the query context.}\)
The final component of the Kneser-Ney LM computation is $N^{1+}(\alpha \cdot \star)$, the number of unique contexts considering symbols on both sides of the pattern. Unfortunately this does not map to a simple suffix tree operation, but instead requires enumeration, $N^{1+}(\alpha \cdot \star) = \sum_{s \in F(\alpha)} N^{1+}(\alpha s)$, where $F(\alpha)$ is the set of symbols that can follow $\alpha$. Algorithm 1 shows how this is computed, with lines 7 and 8 enumerating $s \in F(\alpha)$ using the edge labels of the children of $v$. For each symbol, line 9 searches for an extended pattern incorporating the new symbol $s$ in the reverse CST (part of the reverse CST), by refining the existing match $v_R$ using a single backward search operation after which we can compute $N^{1+}(\alpha s)$. Line 5 deals with the special case where the pattern does not match a complete edge, in which case there is only one unique right context and therefore $N^{1+}(\alpha \star s) = N^{1+}(\alpha)$.\footnote{Backward search in the reverse tree corresponds to searching for the reversed pattern appended with one symbol.}

N1P and N1P_FRONTBACK can compute the requisite occurrence counts for n-gram language modelling, however at considerable cost in terms of space and time. The need for twin reverse and forward CSTs incurs a significant storage overhead, as well as the search time to match the pattern in both CSTs. We show in Section 5 how we can avoid the need for the reversed suffix tree, giving rise to lower memory requirements and faster runtime. Beyond the need for twin suffix trees, the highest time complexity calls are string-depth, edge and backward-search. Calling string-depth is constant time for internal nodes, but $O(\text{SAS} \log \sigma)$ for leaf nodes; fortunately we can avoid this call for leaves, which by definition extend to the end of the corpus and consequently extend further than our pattern.\footnote{We assume search patterns do not extend beyond a single sentence, and thus will always be shorter than the edge labels.}

The costly calls to edge and backward-search however cannot be avoided. This leads to an overall time complexity of $O(1)$ for N1P and $O(F(\alpha) \times \text{SAS} \times \log \sigma)$ for N1P_FRONTBACK, where $F(\alpha)$ is the number of following symbols and SAS is the suffix array value sample rate described in Section 2.

**Figure 2:** Counts required for computing $P(\text{keep in the town})$ (right) and the suffix tree nodes required for computing each value (left). The two left-most columns correspond to $v_R^{\text{all}}$ and $v_R$ and are updated using forward-search in the reverse CST, while the righter-most column correspond to $v_F$ and is updated using backward-search in the forward CST. See Algorithm 2 for details.

**Algorithm 1** Two-sided occ., $N^{1+}(\alpha \cdot \star)$

Precondition: $v_R$ in forward CST $t_F$ matches $\alpha$

Precondition: $v_R$ in reverse CST $t_R$ matches $\alpha$

1: function N1P_FRONTBACK($t_F$, $v_F$, $t_R$, $v_R$, $\alpha$)  
2: \hspace{1em} $o \leftarrow 0$
3: \hspace{1em} $d \leftarrow \text{string-depth}(v_F)$
4: \hspace{1em} if $d > |\alpha|$ then
5: \hspace{2em} $o \leftarrow N1P(t_R, v_R, \star)$
6: \hspace{1em} else
7: \hspace{2em} for $u_F \leftarrow \text{children}(v_F)$ do
8: \hspace{3em} $s \leftarrow \text{edge}(u_F, d+1)$
9: \hspace{3em} $u_R \leftarrow \text{back-search}(v_R, s)$
10: \hspace{2em} $o \leftarrow o + N1P(t_R, u_R, \star)$
11: return $o$

4 Dual CST Algorithm

The methods above for computing the frequency and occurrence counts provide the ingredients necessary for computing n-gram language model probabilities. This leaves the algorithmic problem...
Algorithm 2 KN probability $P(w_k^{k-1}(m-1))$

1: function PROBKNESERNEY($t_F, t_R, W, m$)
2:     $v_F \leftarrow \text{root}(t_F)$ \triangleright match for suffix of $w_k^{k-1}$
3:     $v_R \leftarrow \text{root}(t_R)$ \triangleright match for suffix of $w_k^{k-1}$
4:     $v_{\mathit{all}} \leftarrow \text{root}(t_{\mathit{all}})$ \triangleright match for suffix of $w_k^{k-1}$
5:     $p \leftarrow 1$
6:     for $i \leftarrow 1$ to $m$
7:         $v_{\mathit{all}} \leftarrow \text{forw-search}(v_{\mathit{all}}, w_{k-i+1})$
8:         if $i > 1$ then
9:             $v_F \leftarrow \text{back-search}(v_F, w_{k-i+1})$
10:        if $i < m$ then
11:            $v_R \leftarrow \text{forw-search}(v_R, w_{k-i+1})$
12:        $D_i \leftarrow \text{lookup discount for igram}$
13:        if $i = m$ then
14:            $c \leftarrow \text{size}(v_F^{\mathit{all}})$
15:            $d \leftarrow \text{size}(v_R)$
16:        else
17:            $c \leftarrow N1P(t_F, v_F^{\mathit{all}}, w_k^{k-1})$
18:            $d \leftarrow N1P是韩国(\ell_F, v_F, v_R, \mathit{all}, w_k^{k-1})$
19:        if $i > 1$ then
20:            if $v_F$ is valid then
21:                $q \leftarrow N1P(t_F, v_F, w_k^{k-1})$
22:            $p \leftarrow \frac{1}{2}(\max(c - D_i, i) + D_i q)$
23:        else if $i = 1$ then
24:            $p \leftarrow c/N1^+(\alpha \cdot)$
25:        return $p$

Algorithm 3 Precompute KN discounts

1: function PRECOMPUTEDISCOUNTS($t_R, m$)
2:     $c_{k,f} \leftarrow 0$ \forall $k \in [1, m], f \in [1, 2]$
3:     $N_{k,g} \leftarrow 0$ \forall $k \in [1, m], g \in [1, 2]$
4:     $N1^+(\alpha \cdot) \leftarrow 0$
5:     for $v_R \leftarrow \text{descendants(root}(t_R))$ \triangleright depth-first
6:         $d_P \leftarrow \text{string-depth}(parent(v_R))$
7:         $d \leftarrow \text{string-depth}(v_R)$
8:         for $k \leftarrow d_P + 1$ to $\min(d, d_P + m)$ do
9:             $s \leftarrow \text{edge}(v_R, k)$
10:        if $s$ is the end of sentence sentinel then
11:            skip all children of $v_R$
12:        else
13:            if $k = 2$ then
14:                $N1^+(\alpha \cdot) \leftarrow N1^+(\alpha \cdot) + 1$
15:            $f \leftarrow \text{size}(v_R)$
16:            if $1 \leq f \leq 2$ then
17:                $c_{k,f} \leftarrow c_{k,f} + 1$
18:            if $k < d$ then
19:                $g \leftarrow 1$
20:            else
21:                $g \leftarrow \text{degree}(v_R)$
22:        if $1 \leq g \leq 2$ then
23:            $N_{k,g} \leftarrow N_{k,g} + 1$
24:        return $c, N1^+, N1^+(\alpha \cdot)$

of efficiently ordering the search operations in forward and reverse CST structures.

This paper considers an interpolated LM formulation, in which probabilities from higher order contexts are interpolated with lower order estimates. This iterative process is apparent in Figure 2 (right) which shows the quantities required for probability scoring for an example $m$-gram. Equivalently, the iteration can be considered in reverse, starting from unigram estimates and successively growing to large $m$-grams, in each stage adding a single new symbol to left of the pattern. This suits incremental search in a CST in which search bounds are iteratively refined, which has a substantially lower time complexity compared to searching over the full index in each step.

Algorithm 2 presents an outline of the approach. This uses a forward CST, $t_F$, and a reverse CST, $t_R$, with three CST nodes (lines 2–4) tracking the match progress for the full igram ($v_{\mathit{all}}$) and the $(i - 1)$-gram context ($v_F, v_R$), $i = 1, \ldots, m$. The need to maintain three concurrent searches arises from the calls to $\text{size}$, $N1^+(\alpha \cdot)$, $N1^+(\alpha \cdot)$ and $N1^+(\alpha \cdot)$ (lines 14, 15, 17, 21; and 18, respectively). These calls impose conditions on the direction of the suffix tree, e.g., such that the edge labels and node degree can be used to compute the number of left or right contexts in which a pattern appears. The matching process is illustrated in Figure 2 where the three search nodes are shown on the left, considered bottom to top, and their corresponding count operations are shown to the right. The $N1^+(\alpha \cdot)$ calls require a match in the reverse CST (left-most column, $v_R^{\mathit{all}}$), while the $N1^+(\alpha \cdot)$ require a match in the forward CST (right-most column, $v_F$, matching the $(i - 1)$-gram context). The $N1^+(\alpha \cdot)$ computation reuses the forward match while also requiring a match for the $(i - 1)$-gram context in the reversed CST, as tracked by the middle column ($v_R$). Because of the mix of forward and reverse CSTs, coupled with search patterns that are revealed right-to-left, incremental search in each of the CSTs needs to be handled differently (lines 7–11). In the forward CST, we perform $\text{backward search}$ to extend the search pattern to the left, which can be computed very efficiently from the BWT in the Csa.\textsuperscript{9} Conversely in the reverse CST, we must use $\text{forward search}$ as we are effectively extending the reversed pattern to the right; this operation is considerably more costly.

The discounts $D$ on line 12 of Algorithm 2 and $N1^+(\alpha \cdot)$ (a special case of line 18) are precomputed directly from the CSTs thus avoiding several costly computations at runtime. The precomputed

\footnote{See Supplementary Materials Table 1 for the time complexities of this and other CSA and CST methods.}
tion algorithm is provided in Algorithm 3 which operates by traversing the nodes of the reverse CST and at each stage computing the number of m-grams that occur 1–2 times (used for computing $D_m$ in eq. 1), or with $N^{1+}(\cdot \alpha) \in [1 - 2]$ (used for computing $D_k$ in eq. 2), for various lengths of m-grams. These quantities are used to compute the discount parameters, which are then stored for later use in inference.

Note that the PRECOMPUTED DISCOUNTS algorithm can be slow, although it is significantly faster if we remove the edge calls and simply include in our counts all m-grams finishing a sentence or spanning more than one sentence. This has a negligible (often beneficial) effect on perplexity.

5 Improved Single CST Approach

The above dual CST algorithm provides an elegant means of computing LM probabilities of arbitrary order and with a limited space complexity ($O(n)$, or roughly $n$ in practice). However the time complexity is problematic, stemming from the expensive method for computing N1PFRON TBACK and repeated searches over the CST, particularly forward-search. Now we outline a method for speeding up the algorithm by doing away with the reverse CST. Instead the critical counts, $N^{1+}(\cdot \alpha)$ and $N^{1+}(\cdot \alpha \bullet)$ are computed directly from a single forward CST. This confers the benefit of using only backward search and avoiding redundant searches for the same pattern (cf. lines 9 and 11 in Algorithm 2).

The full algorithm for computing LM probabilities is given in Algorithm 4, however for space reasons we will not describe this in detail. Instead we will focus on the method’s most critical component, the algorithm for computing $N^{1+}(\cdot \alpha \bullet)$ from the forward CST, presented in Algorithm 5. The key difference from Algorithm 1 is the loop from lines 6–9, which uses the interval-symbols (Schnattinger et al., 2010) method. This method assumes a wavelet tree representation of the SA component of the CST, an efficient encoding of the BWT as described in section 2. The interval-symbols method uses RANK operations to efficiently identify for a given pattern the set of preceding symbols $P(\alpha)$ and the ranges $SA[l_s, r_s]$ corresponding to the patterns $s\alpha$ for all $s \in P(\alpha)$

---

**Algorithm 4** KN probability $P(w_k | w^{k-1}_{k-1})$ using a single CST

1: function PROBNESERNEY1($\nu$, $w$, $m$)
2: $\nu \leftarrow \text{root}(\nu)$ \triangleright match for context $w^{k-1}_{k-1}$
3: $v^{\text{diff}} \leftarrow \text{root}(\nu)$ \triangleright match for $w^{k-1}_{k-1}$
4: $p \leftarrow 1$
5: for $i \leftarrow 1$ to $m$
6: $v^{\text{diff}} \leftarrow \text{back-search}(\text{[}l_{\nu}(v^{\text{diff}}), r_{\nu}(v^{\text{diff}})], w^{k-1}_{k-1+1})$
7: if $i > 1$ then
8: $v \leftarrow \text{back-search}(\text{[}l_{\nu}(v), r_{\nu}(v)], w^{k-1}_{k-1+1})$
9: $D_i \leftarrow$ discount parameter for $i$-gram
10: if $i = m$ then
11: $c \leftarrow \text{size}(v^{\text{diff}})$
12: $d \leftarrow \text{size}(v)$
13: else
14: $c \leftarrow \text{N1PBACK1}(\nu, v, w^{k-1}_{k-1+1})$
15: $d \leftarrow \text{N1PFRON TBACK1}(\nu, v, w^{k-1}_{k-1+1} \bullet)$
16: if $i > 1$ then
17: if $v$ is valid then
18: $q \leftarrow \text{N1P}(\nu, v, w^{k-1}_{k-1+1} \bullet)$
19: $p \leftarrow \frac{1}{\alpha} (\max(c - D_i, 0) + D_ipq)$
20: else
21: $p \leftarrow c/N^{1+}(\bullet \bullet)$
22: return $p$

**Algorithm 5** $N^{1+}(\cdot \alpha \bullet)$, using forward CST

Precondition: $\nu$ in forward CST $\nu$ matches $\alpha$

1: function N1PFRON TBACK1($\nu$, $\nu$, $\alpha$)
2: $o \leftarrow 0$
3: if string-depth($\nu$) > $|\alpha|$ then
4: $o \leftarrow \text{N1PBACK1}(\nu, \nu, \bullet \alpha)$
5: else
6: for $(l, r, s) \leftarrow \text{int-syms}(\nu, [l_{\nu}(v), r_{\nu}(v)])$ do
7: $l' \leftarrow C_s + l$
8: $r' \leftarrow C_s + r$
9: $o \leftarrow o + \text{N1P}(\nu, \text{node}(l', r'), s\alpha \bullet)$
10: return $o$

by visiting all leaves of the wavelet tree of symbols occurring in $T^{\text{bwt}}[l, r]$ (corresponding to $\alpha$) in $O(|P(\alpha)| \log \sigma)$ time (lines 6–8). These ranges $SA[l', r']$ can be used to find the corresponding suffix tree node for each $s\alpha$ in $O(1)$ time. To illustrate, consider the pattern $\alpha = \text{“night”}$ in Figure 1a. From $T^{\text{bwt}}$ we can see that this is preceded by $s = \text{“old”}$ (1st occurrence in $T^{\text{bwt}}$) and $s = \text{“the”}$ (3rd and 4th); from which we can compute the suffix tree nodes, namely $[15, 15]$ and $[16 + (3 - 1), 16 + (4 - 1)] = [18, 19]$ for “old” and “the” respectively. 11

N1PBACK1 is computed in a similar way, using the interval-symbols method to compute the number of unique preceding symbols (see Supplementary Materials, Algorithm 7). Overall the time complexity of inference for both N1PBACK1

---

10Discounts are computed up to a limit on m-gram size, here set to 10. The highest order values are used for computing the discount of m-grams above the limit at runtime.

11Using the offsets into the SA for each symbol, $C_{old} = 15$ and $C_{new} = 16$, while $-1$ adjusts for counting from 1.
and N1FrontBack1 is $O(P(\alpha) \log \sigma)$ where $P(\alpha)$ is the number of preceeding symbols of $\alpha$, a considerable improvement over N1FrontBack using the forward and reverse CSTs. Overall this leads to considerably faster computation of $m$-gram probabilities compared to the two CST approach, and although still slower than highly optimised LM toolkits like SRILM, it is fast enough to support large scale experiments, and has considerably better scaling performance with the Markov order $m$ (even allowing unlimited order), as we will now demonstrate.

6 Experiments

We used Europarl dataset and the data was numberized after tokenizing, splitting, and excluding XML markup. The first 10$k$ sentences were used as the test data, and the last 80% as the training data, giving rise to training corpora of between 8M and 50M tokens and uncompressed size of up to 200 MiB (see Table 1 for detailed corpus statistics). We also processed the full 52 GiB uncompressed “20150205” English Wikipedia articles dump to create a character level language model consisting of 72M sentences. We excluded 10$k$ random sentences from the collection as test data. We use the SDSL library (Gog et al., 2014) to implement all our structures and compare our indexes to SRILM (Stolcke, 2002). We refer to our dual-CST approach as D-CST, and the single-CST as S-CST.

We evaluated the perplexity across different languages and using $m$-grams of varying order from $m = 2$ to $\infty$ (unbounded), as shown on Figure 3. Our results matched the perplexity results from SRILM (for smaller values of $m$ in which SRILM training was feasible, $m \leq 10$). Note that perplexity drops dramatically from $m = 2 \ldots 5$ however the gains thereafter are modest for most languages. Despite this, several large $m$-gram matches were found ranging in size up to a 34-gram match. We speculate that the perplexity plateau is due to the simplistic Kneser-Ney discounting formula which is not designed for higher order $m$-gram LMs and appear to discount large $m$-grams too aggressively. We leave further exploration of richer discounting techniques such as Modified Kneser-Ney (Chen and Goodman, 1996) or the Sequence Memoizer (Wood et al., 2011) to our future work.

Figure 4 compares space and time of our indexes with SRILM on the German part of Europarl. The construction cost of our indexes in terms of both space and time is comparable to that of a 3/4-gram SRILM index. The space usage of D-CST index is comparable to a compact 3-gram SRILM index. Our S-CST index uses only 177 MiB RAM at query time, which is comparable to the size of the collection (172 MiB). However, query processing is significantly slower for both our structures. For 2-grams, D-CST is 3 times slower than a 2-gram SRILM index as the expensive $N^{1+}(\cdot \cdot \cdot \cdot \cdot \cdot)$ is not computed. However, for large $m$-grams, our indexes are much slower than SRILM. For $m > 2$, the D-CST index is roughly six times slower than S-CST. Our fastest index, is 10 times slower than the slowest SRILM 10-gram index. However, our run-time is independent of $m$. Thus, as $m$ increases, our index will become more competitive to SRILM while using a constant amount of space.

Table 1: Dataset statistics, showing total uncompressed size; and tokens, types and sentence counts for the training partition. For Wikipedia the Word Types, and Tokens are computed based on characters.

| Language | Size(MiB) | Tokens(M) | Word Types | Sentences(K) |
|----------|----------|-----------|------------|--------------|
| BG       | 36.11    | 8.53      | 114930     | 329          |
| CS       | 53.48    | 12.25     | 174592     | 535          |
| DE       | 171.80   | 44.07     | 399354     | 1785         |
| EN       | 179.15   | 49.32     | 124233     | 1815         |
| FI       | 145.32   | 32.85     | 721389     | 1737         |
| FR       | 197.68   | 53.82     | 147058     | 1792         |
| HU       | 52.53    | 12.02     | 318882     | 527          |
| IT       | 186.67   | 48.08     | 178259     | 1703         |
| PT       | 187.20   | 49.03     | 183633     | 1737         |

Wikipedia | 8637    | 9057   | 196   | 87835       |

Figure 3: Perplexity results on several Europarl languages for different $m$-gram sizes, $m = 2 \ldots 10, 15, 20, \infty$.  

$\cdot \cdot \cdot$
Next we analyze the performance of our index on the large Wikipedia dataset. The s-CST, character level index for the data set requires 22 GiB RAM at query time whereas the D-CST requires 43 GiB. Figure 5 shows the run-time performance of both indexes for different mgrams, broken down by the different components of the computation. As discussed above, 2-gram performance is much faster. For both indexes, most time is spent computing N1PFRONTBACK (i.e., \(N^{1+}(\cdot \cdot\cdot)\)) for all \(m > 2\). However, the wavelet tree traversal used in s-CST roughly reduces the running time by a factor of three. The complexity of N1PFRONTBACK depends on the number of contexts, which is likely small for larger mgrams, but can be large for small mgrams, which suggest partial precomputation could significantly increase the query performance of our indexes. Exploring the myriad of different CST and CSA configurations available could also lead to significant improvements in runtime and space usage also remains future work.

7 Conclusions

This paper has demonstrated the massive potential that succinct indexes have for language modeling, by developing efficient algorithms for on-the-fly computing of mgram counts and language model probabilities. Although we only considered a Kneser-Ney LM, our approach is portable to the many other LM smoothing method formulated around similar count statistics. Our complexity analysis and experimental results show favourable scaling properties with corpus size and Markov order, albeit running between 1-2 orders of magnitude slower than a leading count-based LM. Our ongoing work seeks to close this gap: preliminary experiments suggest that with careful tuning of the succinct index parameters and caching expensive computations, query time can be competitive with state-of-the-art toolkits, while using less memory and allowing the use of unlimited context.

Acknowledgments

Ehsan Shareghi and Gholamreza Haffari are grateful to National ICT Australia (NICTA) for generous funding, as part of collaborative machine learning research projects. Matthias Petri is the recipient of an Australian Research Councils Discovery Project scheme (project DP140103256). Trevor Cohn is the recipient of an Australian Research Council Future Fellowship (project number FT130101105).

References

Thorsten Brants, Ashok C Popat, Peng Xu, Franz J Och, and Jeffrey Dean. 2007. Large language models in machine translation. In Proc. EMNLP-CoNLL.

M. Burrows and D. Wheeler. 1994. A block sorting lossless data compression algorithm. Technical Report 124, DEC.

Bernard Chazelle, Joe Kilian, Ronitt Rubinfeld, and Ayelet Tal. 2004. The bloomier filter: An efficient data structure for static support lookup tables. In Proc. SODA, pages 30–39.

Stanley F Chen and Joshua Goodman. 1996. An empirical study of smoothing techniques for language modeling. In Proc. ACL, pages 310–318.
