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 ∞-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 $20\times$ the training corpus).

In contrast, we[1] make use of recent advances in *compressed suffix trees* (CSTs) (Sadakane, 2007) to build compact indices with much more modest memory requirements ($\approx$ 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 modelling, although querying is substantially slower than a SRILM 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

[1] For the implementation see: https://github.com/eehsan/lm-sdsl.

Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing, pages 2409–2418,
Lisbon, Portugal, 17-21 September 2015.

for language modelling and other 'big data' problems in language processing.

2 Background

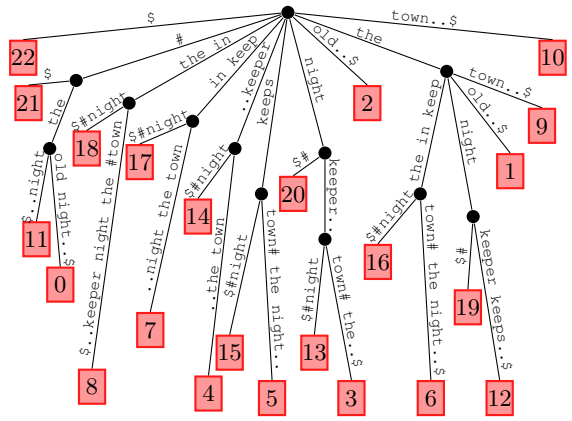
Suffix Arrays and Suffix Trees Let $\mathcal{T}$ be a string of size n drawn from an alphabet Σ of size σ. Let $\mathcal{T}[i..n-1]$ be a *suffix* of $\mathcal{T}$. The *suffix tree* (Weiner, 1973) of $\mathcal{T}$ is the compact labeled tree of $n+1$ leaves where the root to leaf paths correspond to all suffixes of $\mathcal{T}\$$, where \$ is a terminating symbol not in Σ. 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* corresponds to the length of the path-label. Searching for a pattern α of size m in $\mathcal{T}$ translates to finding the *locus* node v closest to the root such that α 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 $\mathcal{T}$ corresponds to the path-label of the i-th leaf. The starting position of the suffix can be associated its corresponding leaf in the tree as shown in Figure 1a. All occurrences of α in $\mathcal{T}$ can be retrieved by visiting all leaves in the subtree of the locus of α. 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 $\mathcal{T}$ is an array $SA[0 \dots n-1]$ such that $SA[i]$ corresponds to the starting position of the i-th smallest suffix in $\mathcal{T}$ or the i-th leaf in the suffix tree of $\mathcal{T}$. 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 $\mathcal{T}$ to determine $SA[18, 19]$, the interval in SA corresponding the the suffixes in $\mathcal{T}$ prefixed by the pattern. In practice, suffix arrays use $4-8n$ 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 $\mathcal{T}$ and prohibit the use of these structures for all but small data sets.

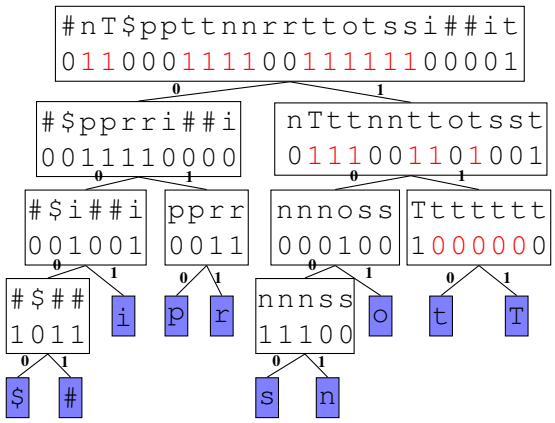
Compressed Suffix Structures Reducing the space usage of suffix based index structure has recently become an active area of research. The space usage of a suffix array can be reduced significantly by utilizing the compressibility of text combined with succinct data structures. A *succinct* data structure 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 $\mathcal{T}$ using $nH_k(\mathcal{T}) + o(n \log \sigma)$ 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 $\mathcal{T}$ uses roughly space equivalent to the compressed representation of $\mathcal{T}$ 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 $\mathcal{T}^{bwt}[i] = \mathcal{T}[SA[i]-1]$ (see Figure 1). Searching for a pattern using the FM-Index is performed in reverse order by performing $\text{RANK}(\mathcal{T}^{bwt}, i, c)$ operations $O(m)$ times. Here, $\text{RANK}(\mathcal{T}^{bwt}, i, c)$ counts the number of times symbol c occurs in $\mathcal{T}^{bwt}[0 \dots i-1]$. This process is usually referred to as *backward search*. Let $SA[l_i, r_i]$ be the interval corresponding to the suffixes in $\mathcal{T}$ matching $\alpha[i \dots m-1]$. By definition of the BWT, $\mathcal{T}^{bwt}[l_i, r_i]$ corresponds to the symbols in $\mathcal{T}$ preceding $\alpha[i \dots m-1]$ in $\mathcal{T}$. Due to the lexicographical ordering of all suffixes in SA, the interval $SA[l_{i-1}, r_{i-1}]$ corresponding to all occurrences of $\alpha[i-1 \dots m-1]$ can be determined by computing the rank of all occurrences of $c = \alpha[i-1]$ in $\mathcal{T}^{bwt}[l_i, r_i]$. Thus, we compute $\text{RANK}(\mathcal{T}^{bwt}, l_i, c)$, the number of times symbol c occurs before l_i and $\text{RANK}(\mathcal{T}^{bwt}, r_i+1, c)$, the number of occurrences of c in $\mathcal{T}^{bwt}[0, r_i]$. 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 Σ at a negligible cost of $\sigma \log n$ bits. Thus, the new interval is computed as $l_{i-1} = C_c + \text{RANK}(\mathcal{T}^{bwt}, l_i, c)$ and $r_{i-1} = C_c + \text{RANK}(\mathcal{T}^{bwt}, r_i+1, c)$.

The time and space complexity of the FM-index thus depends on the cost of storing and pre-



(a) Word-based Suffix Tree.



(b) Wavelet tree and RANK($\mathcal{T}^{bwt}$, 17, 't') = 5.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
$\mathcal{T}[SA[i]]$	$	#	#	#	i	i	p	p	r	r	s	s	n	n	n	o	t	t	t	t	t	t	T
SA	22	21	11	0	18	8	17	7	14	4	15	5	20	13	3	2	16	6	19	12	1	9	10
$\mathcal{T}^{bwt}$	#	n	T	$	p	p	t	t	n	n	r	r	t	t	o	t	s	s	i	#	#	i	t

Figure 1: Data structures for the sample text $\mathcal{T}$="#the old night keeper keeps the keep in the town# the night keeper keeps the keep in the night#$" with alphabet Σ={the, old, night, keeper, keeps, keep, in, town, #} and code words *$=0000*, *#=0001*, *i=in=001*, *p=keep=010*, *r=keeper=011*, *s=keeps=1000*, *o=old=101*, *t=the=110*, *n=night=1001* and *T=town=111*.

processing $\mathcal{T}^{bwt}$ to answer RANK efficiently. A *wavelet tree* can be used to answer RANK over $\mathcal{T}^{bwt}$ in $O(\log \sigma)$ time. The wavelet tree reduces RANK over an alphabet Σ 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 $\mathcal{T}^{bwt}$. To answer RANK($\mathcal{T}^{bwt}, i, c$), the tree is traversed based on the code word of c, performing binary RANK at each level. For example, RANK($\mathcal{T}^{bwt}$, 17, 't') translates to performing RANK($WT_{root}, 17, 1$) = 12 on the top level of the wavelet tree, as `t=the=110`. We recurse to the right subtree of the root node and compute RANK($WT_1, 12, 1$) as there were 12 ones in the root node and the next bit in the codeword of 'the' is also one. This process continues until the correct leaf node is reached to answer RANK($\mathcal{T}^{bwt}$, 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, additional space is needed to access $SA[i]$ or the inverse suffix array $SA^{-1}[SA[i]] = i$. In the simplest scheme, both values are periodically sampled using a given sample rate *SAS* (e.g. 32) such that $SA[i] \mod SAS = 0$. Then, for any $SA[i]$ or $SA^{-1}[i]$, at most $O(SAS)$ RANK operations on $\mathcal{T}^{bwt}$ 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 $\approx 2p$

[2] However, if code-words for each symbol are chosen based on their Huffman-codes the size of the wavelet tree reduces to $nH_0(\mathcal{T})(1 + o(1))$ bits which can be further be reduced to to $nH_k(\mathcal{T}) + o(n \log \sigma)$ bits by using entropy compressed bitvectors.

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 $4 - 6n$ 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 $m$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

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

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

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

In the above formula, D_k is the $k$gram-specific discount parameter, and the *occurrence count* $N^{1+}(\alpha\ \bullet) = |\{w : c(\alpha w) > 0\}|$ is the number of observed word types following the pattern α; the occurrence counts $N^{1+}(\bullet\ \alpha)$ and $N^{1+}(\bullet\ \alpha\ \bullet)$ are defined accordingly. The recursion stops at unigram level where the unigram probabilities are defined as $\bar{P}(w_i) = N^{1+}(\bullet\ w_i)/N^{1+}(\bullet\ \bullet)$.[4]

[3] See *Supplementary Materials* Table 1 for an overview of the complexities of the functionality of the CST that is used in our experiments.

[4] Modified Kneser-Ney, proposed by Chen and Goodman (1996), typically outperforms interpolated Kneser-Ney through its use of context-specific discount parameters. The 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.

3 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(*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, $N^{1+}(\alpha\ \bullet)$, left contexts, $N^{1+}(\bullet\ \alpha)$, and contexts to both sides of the pattern, $N^{1+}(\bullet\ \alpha\ \bullet)$. The first of these can be handled easily, as

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

where v is the node matching α, 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+}(*keep in* $\bullet$) $= 2$, 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+}(*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 keep*$_R$) and then use above method to compute the occurrence count (denoted hereafter N1P(t, v, α)[6], where t is the CST.).

[5] 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.

[6] In the presented algorithms, we overload the pattern argument in function calls for readability, and use $\bullet$ to denote the query context.

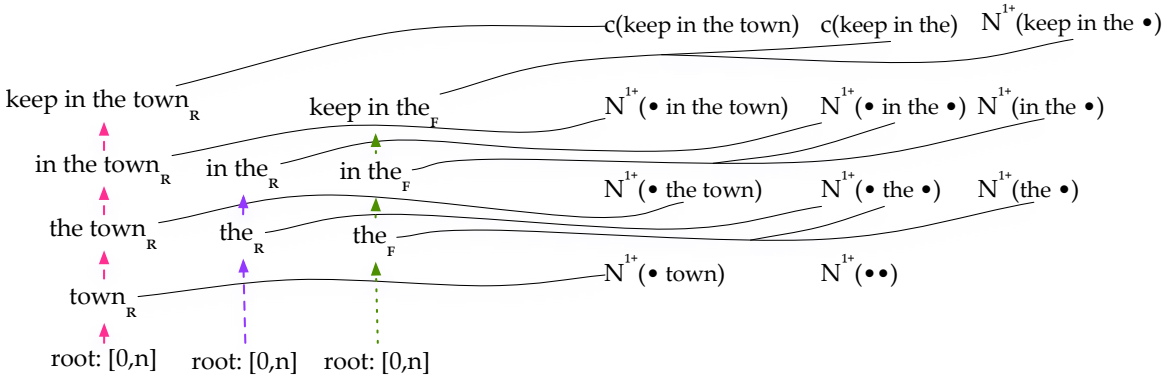


Figure 2: Counts required for computing $P(town|keep\ in\ the)$ (right) and the suffix tree nodes required for computing each value (left). The two left-most columns correspond to $v_{\mathrm{R}}^{\mathrm{all}}$ and v_{R} and are updated using *forward-search* in the reverse CST, while the righter-most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

The final component of the Kneser-Ney LM computation is $N^{1+}(\bullet\ \alpha\ \bullet)$, 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, $N^{1+}(\bullet\ \alpha\ \bullet) = \sum_{s \in F(\alpha)} N^{1+}(\bullet\ \alpha s)$, where $F(\alpha)$ is the set of symbols that can follow α. 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 CSA (part of the reverse CST), by refining the existing match v_{R} using a single backward search operation after which we can compute $N^{1+}(\bullet\ \alpha s)$.[7] Line 5 deals with the special case where the pattern does not match a complete edge, in which case there is only only unique right context and therefore $N^{1+}(\bullet\ \alpha\ \bullet) = N^{1+}(\bullet\ \alpha)$.

N1P and N1PFRONTBACK can compute the requisite occurence counts for $m$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 $O(SAS \log \sigma)$ 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.[8] 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 $O(F(\alpha) \times SAS \times \log \sigma)$ for N1PFRONTBACK, where $F(\alpha)$ is the number of following symbols and *SAS* is the suffix array value sample rate described in Section 2.

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

Precondition: v_{F} in forward CST t_{F} matches α
Precondition: v_{R} in reverse CST t_{R} matches α

```
1:  function N1PFRONTBACK(t_F, v_F, t_R, v_R, α)
2:      o ← 0
3:      d ← string-depth(v_F)
4:      if d > |α| then
5:          o ← N1P(t_R, v_R, • α)
6:      else
7:          for u_F ← children(v_F) do
8:              s ← edge(u_F, d + 1)
9:              u_R ← back-search(v_R, s)
10:             o ← o + N1P(t_R, u_R, • αs)
11:     return o
```

4 Dual CST Algorithm

The methods above for computing the frequency and occurrence counts provide the ingredients necessary for computing $m$gram language model probabilities. This leaves the algorithmic problem

[7] Backward search in the reverse tree corresponds to searching for the reversed pattern appended with one symbol.

[8] We assume search patterns do not extend beyond a single sentence, and thus will always be shorter than the edge labels.

Algorithm 2 KN probability $P\big(w_k|w_{k-(m-1)}^{k-1}\big)$

```
1: function ProbKneserNey(t_F, t_R, w, m)
2:     v_F ← root(t_F)                    ▷ match for suffix of w_{k-(m-1)}^{k-1}
3:     v_R ← root(t_R)                    ▷ match for suffix of w_{k-(m-1)}^{k-1}
4:     v_R^all ← root(t_R)                ▷ match for suffix of w_{k-(m-1)}^{k}
5:     p ← 1
6:     for i ← 1 to m do
7:         v_R^all ← forw-search(v_R^all, w_{k-i+1})
8:         if i > 1 then
9:             v_F ← back-search(v_F, w_{k-i+1})
10:            if i < m then
11:                v_R ← forw-search(v_R, w_{k-i+1})
12:        D_i ← lookup discount for igram
13:        if i = m then
14:            c ← size(v_R^all)
15:            d ← size(v_F)
16:        else
17:            c ← N1P(t_R, v_R^all, • w_{k-i+1}^{k})
18:            d ← N1PFrontBack(t_F, v_F, t_R, v_R, • w_{k-i+1}^{k-1} •)
19:        if i > 1 then
20:            if v_F is valid then
21:                q ← N1P(t_F, v_F, w_{k-i+1}^{k-1} •)
22:                p ← 1/d (max(c − D_i, 0) + D_i q p)
23:        else if i = 1 then
24:            p ← c/N^{1+}(• •)
25:    return p
```

Algorithm 3 Precompute KN discounts

```
1: function PrecomputeDiscounts(t_R, m)
2:     c_{k,f} ← 0   ∀k ∈ [1, m], f ∈ [1, 2]
3:     N^1_{k,g} ← 0   ∀k ∈ [1, m], g ∈ [1, 2]
4:     N^{1+}(• •) ← 0
5:     for v_R ← descendents(root(t_R)) do        ▷ depth-first
6:         d_P ← string-depth(parent(v_R))
7:         d ← string-depth(v_R)
8:         for k ← d_P + 1 to min(d, d_P + m) do
9:             s ← 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:                    N^{1+}(• •) ← N^{1+}(• •) + 1
15:                f ← size(v_R)
16:                if 1 ≤ f ≤ 2 then
17:                    c_{k,f} ← c_{k,f} + 1
18:                if k < d then
19:                    g ← 1
20:                else
21:                    g ← degree(v_R)
22:                if 1 ≤ g ≤ 2 then
23:                    N^1_{k,g} ← N^1_{k,g} + 1
24:    return c, N^1, N^{1+}(• •)
```

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 $i$gram ($v_{\text{R}}^{\text{all}}$) and the $(i-1)$gram context ($v_{\text{F}}, v_{\text{R}}$), $i = 1 \ldots m$. The need to maintain three concurrent searches arises from the calls to *size*, $N^{1+}(\bullet\ \alpha)$, $N^{1+}(\alpha\ \bullet)$ and $N^{1+}(\bullet\ \alpha\ \bullet)$ (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 $N^{1+}(\bullet\ \alpha)$ calls require a match in the reverse CST (left-most column, $v_{\text{R}}^{\text{all}}$), while the $N^{1+}(\alpha\ \bullet)$ require a match in the forward CST (right-most column, v_{F}, matching the $(i-1)$gram context). The $N^{1+}(\bullet\ \alpha\ \bullet)$ 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 *backward search* to extend the search pattern to the left, which can be computed very efficiently from the BWT in the CSA.[9] Conversely in the reverse CST, we must use *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 $N^{1+}(\bullet\ \bullet)$ (a special case of line 18) are precomputed directly from the CSTs thus avoiding several costly computations at runtime. The precomputa-

[9] 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 $N^{1+}(\bullet\ \alpha) \in [1-2]$ (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.[10] Note that the PRECOMPUTEDISCOUNTS 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 N1PFRONTBACK 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, $N^{1+}(\bullet\ \alpha)$ and $N^{1+}(\bullet\ \alpha\ \bullet)$ 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 $N^{1+}(\bullet\ \alpha\ \bullet)$ 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 describes 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 $P(w_k|w_{k-(m-1)}^{k-1})$ using a single CST

```
1: function ProbKneserNey1(t_F, w, m)
2:     v_F ← root(t_F)                          ▷ match for context w_{k-i}^{k-1}
3:     v_F^all ← root(t_F)                      ▷ match for w_{k-i}^{k}
4:     p ← 1
5:     for i ← 1 to m do
6:         v_F^all ← back-search([lb(v_F^all), rb(v_F^all)], w_{k-i+1})
7:         if i > 1 then
8:             v_F ← back-search([lb(v_F), rb(v_F)], w_{k-i+1})
9:         D_i ← discount parameter for igram
10:        if i = m then
11:            c ← size(v_F^all)
12:            d ← size(v_F)
13:        else
14:            c ← N1pBack1(t_F, v_F^all, • w_{k-i+1}^{k-1})
15:            d ← N1pFrontBack1(t_F, v_F, • w_{k-i+1}^{k-1} •)
16:        if i > 1 then
17:            if v_F is valid then
18:                q ← N1p(t_F, v_F, w_{k-i+1}^{k-1} •)
19:                p ← 1/d (max(c − D_i, 0) + D_i q p)
20:        else
21:            p ← c/N^{1+}(• •)
22:    return p
```

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

Precondition: v_F in forward CST t_F matches α

```
1: function N1pFrontBack1(t_F, v_F, α)
2:     o ← 0
3:     if string-depth(v_F) > |α| then
4:         o ← N1pBack1(t_F, v_F, • α)
5:     else
6:         for ⟨l, r, s⟩ ← int-syms(t_F, [lb(v_F), rb(v_F)]) do
7:             l' ← C_s + l
8:             r' ← C_s + r
9:             o ← o + N1p(t_F, node(l', r'), sα •)
10:    return o
```

by visiting all leaves of the wavelet tree of symbols occurring in $\mathcal{T}^{bwt}[l, r]$ (corresponding to α) 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 α = "night" in Figure 1a. From $\mathcal{T}^{bwt}$ we can see that this is preceeded by s ="old" (1^{st} occurrence in $\mathcal{T}^{bwt}$) and s ="the" (3^{rd} and 4^{th}); 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 preceeding symbols (see *Supplementary Materials*, Algorithm 7). Overall the time complexity of inference for both N1PBACK1

[10] Discounts 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.

[11] Using the offsets into the *SA* for each symbol, $C_{\text{old}} = 15$ and $C_{\text{the}} = 16$, while -1 adjusts for counting from 1.

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

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.

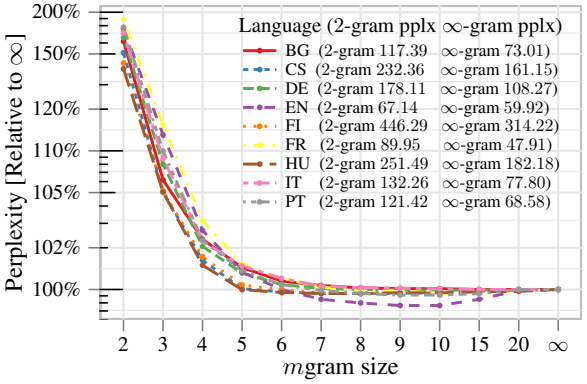


Figure 3: Perplexity results on several Europarl languages for different $m$gram sizes, $m = 2 \ldots 10, 15, 20, \infty$.

and N1PFRONTBACK1 is $O(P(\alpha) \log \sigma)$ where $P(\alpha)$ is the number of preceeding symbols of α, a considerable improvement over N1PFRONTBACK 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 $10k$ 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 $10k$ 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 ∞ (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 $N^{1+}(\bullet\, \alpha\, \bullet)$ 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.

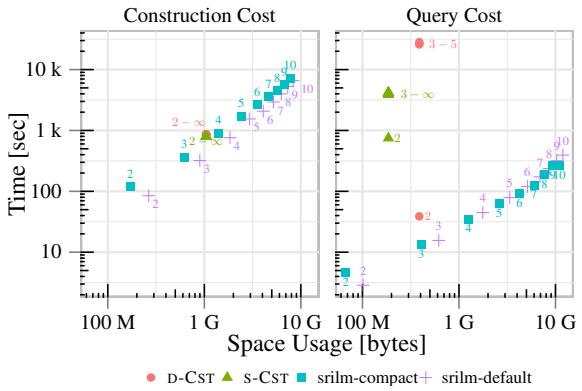


Figure 4: Time versus space tradeoffs measured on Europarl German (de) dataset, showing memory and time requirements.

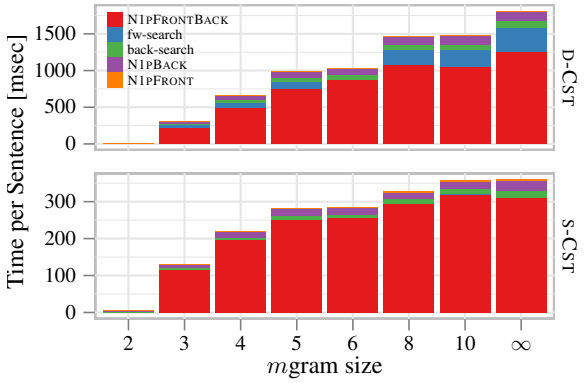


Figure 5: Runtime breakdown of a single pattern averaged over all patterns for both methods over the Wikipedia collection.

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 $m$grams, 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., $N^{1+}(\bullet\ \alpha\ \bullet)$) 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 $m$grams, but can be large for small $m$grams, which suggest partial precomputation could significantly increase the query performance of our indexes. Exploring the myraid 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 modelling, by developing efficient algorithms for on-the-fly computing of $m$gram 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 Ayellet 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.

P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. 2007. Compressed representations of sequences and full-text indexes. *ACM Trans. on Algorithms*, 3(2):article 20.

Paolo Ferragina, Rodrigo González, Gonzalo Navarro, and Rossano Venturini. 2008. Compressed text indexes: From theory to practice. *ACM J. of Exp. Algorithmics*, 13.

Ulrich Germann, Eric Joanis, and Samuel Larkin. 2009. Tightly packed tries: How to fit large models into memory, and make them load fast, too. In *Proc. of the Workshop on Software Engineering, Testing, and Quality Assurance for Natural Language Processing*, pages 31–39.

Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. 2014. From theory to practice: Plug and play with succinct data structures. In *Proc. SEA*, pages 326–337.

David Guthrie and Mark Hepple. 2010. Storing the web in memory: Space efficient language models with constant time retrieval. In *Proc. EMNLP*, pages 262–272.

Kenneth Heafield. 2011. KenLM: Faster and smaller language model queries. In *Proc. WMT*.

Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. 2006. Linear work suffix array construction. *J. ACM*, 53(6):918–936.

Casey Redd Kennington, Martin Kay, and Annemarie Friedrich. 2012. Suffix trees as language models. In *Proc. LREC*, pages 446–453.

Reinhard Kneser and Hermann Ney. 1995. Improved backing-off for m-gram language modeling. In *Proc. ICASSP*, volume 1, pages 181–184.

Philipp Koehn. 2010. *Statistical Machine Translation*. Cambridge University Press, New York, NY, USA.

Stefan Kurtz. 1999. Reducing the space requirement of suffix trees. *Softw., Pract. Exper.*, 29(13):1149–1171.

Udi Manber and Eugene W. Myers. 1993. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948.

Ian Munro. 1996. Tables. In *Proc. FSTTCS*, pages 37–42.

Enno Ohlebusch, Johannes Fischer, and Simon Gog. 2010. CST++. In *Proc. SPIRE*, pages 322–333.

Adam Pauls and Dan Klein. 2011. Faster and smaller n-gram language models. In *Proc. ACL-HLT*.

Lawrence Rabiner and Biing-Hwang Juang. 1993. *Fundamentals of speech recognition*. Prentice-Hall.

L. Russo, G. Navarro, and A. Oliveira. 2011. Fully-compressed suffix trees. *ACM Trans. Algorithms*, 7(4):article 53.

Kunihiko Sadakane. 2007. Compressed suffix trees with full functionality. *Theory Comput. Syst.*, 41(4):589–607.

Thomas Schnattinger, Enno Ohlebusch, and Simon Gog. 2010. Bidirectional search in a string with wavelet trees. In *Proc. CPM*, pages 40–50.

Jeffrey Sorensen and Cyril Allauzen. 2011. Unary data structures for language models. In *Proc. INTERSPEECH*, pages 1425–1428.

Andreas Stolcke, Jing Zheng, Wen Wang, and Victor Abrash. 2011. Srilm at sixteen: Update and outlook. In *Proc. ASRU*, page 5.

Andreas Stolcke. 2002. SRILM–an extensible language modeling toolkit. In *Proc. INTERSPEECH*.

David Talbot and Miles Osborne. 2007. Randomised language modelling for statistical machine translation. In *Proc. ACL*.

Esko Ukkonen. 1995. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260.

Taro Watanabe, Hajime Tsukada, and Hideki Isozaki. 2009. A succinct n-gram language model. In *Proc. ACL Short Papers*, pages 341–344.

Peter Weiner. 1973. Linear pattern matching algorithms. In *Proc. SWAT*, pages 1–11.

Frank Wood, Jan Gasthaus, Cédric Archambeau, Lancelot James, and Yee Whye Teh. 2011. The sequence memoizer. *CACM*, 54(2):91–98.