Space-Efficient Construction of the Burrows-Wheeler Transform

Timo Beller, Maike Zwerger, Simon Gog, and Enno Ohlebusch

1 Institute of Theoretical Computer Science, University of Ulm, 89069 Ulm, Germany
2 Department of Computing and Information Systems, The University of Melbourne, VIC, 3010, Melbourne, Australia
Simon.Gog@unimelb.edu.au

Abstract. The Burrows-Wheeler transform (BWT), originally invented for data compression, is nowadays also the core of many self-indexes, which can be used to solve many problems in bioinformatics. However, the memory requirement during the construction of the BWT is often the bottleneck in applications in the bioinformatics domain.

In this paper, we present a linear-time semi-external algorithm whose memory requirement is only about one byte per input symbol. Our experiments show that this algorithm provides a new time-memory trade-off between external and in-memory construction algorithms.

1 Introduction

In 1994 Burrows and Wheeler presented the Burrows-Wheeler transform (BWT). This reversible transformation produces a permutation of the input string, in which symbols tend to occur in clusters. Because of this clustering, in virtually all cases the BWT compresses much easier than the original string, and Burrows and Wheeler suggested their transformation as a preprocessing step in data compression. Data compression has become a major application for the Burrows-Wheeler transform, e.g. it is the basis of the bzip2 algorithm.

Interestingly, the BWT has become the core of self-indexes which have applications in bioinformatics and information retrieval. In the data compression scenario it is possible to split a large input and construct the BWT for small blocks, since decoding and encoding are done sequentially. However, this is not possible for self-indexes because the optimal search routine requires the BWT of the whole text. In this case, both the runtime and the memory requirement of the construction of the BWT are critical. In the past, there were impressive improvements in algorithms constructing the suffix array. Theoretical worst-case time complexity, practical runtime and memory footprint have been improved. As the BWT can easily (fast and space efficiently) be obtained from the suffix array, the construction of the BWT profited indirectly from these improvements. However, \(n \log n\) bits seems to be a lower memory bound for fast suffix array construction. On the other hand, this memory bound seems not to be valid for BWT construction, as there are algorithms that directly construct the BWT.
using less than $n \log n$ bits, but still depend on the input size. External algorithms take only a given amount of memory, which is independent of the input size and normally user defined. In the past, external algorithms were presented that compute the suffix array or the BWT, e.g. \cite{4, 6, 8, 11}. While this approach finally solves the memory problem (the algorithm needs only as much memory as available), it is commonly known that external algorithms have a significant slow down.

Thus, external algorithms are only used when the input does not fit in RAM. Currently, this happens already for quite small files: In our experiments on a machine equipped with 8 GB RAM, the suffix array construction algorithm \texttt{divsufsort} has already suffered from swapping effects for inputs larger than 1.5 GB. A direct computation of the BWT may allow bigger inputs: An implementation of Sadakane \cite{2} can construct the BWT for inputs up to 3 GB on that machine. But this implementation is limited to inputs of 4 GB (even if much more RAM would be available). We show in this paper that the space requirements can further be improved: We present a new semi-external algorithm to compute the BWT. Semi-external algorithms are in between internal algorithms and external algorithms. To be more precise, semi-external algorithms are—at least in this paper—algorithms that are allowed to use an input dependent amount of memory (like internal algorithms), but also use disk memory (like external algorithms). In practice, semi-external algorithms store all data on disk that is accessed sequentially, while data with random access pattern is kept in main memory. Our implementation has no limitation on the input size and can construct the BWT of a 6 GB file with only 8 GB of RAM. In contrast, internal suffix array construction algorithms would need over 54 GB of RAM (or 31 GB if bit compression would be used) to compute the suffix array of a 6 GB file, because they must keep at least the input and the output in memory.

2 Preliminaries

Let $\Sigma$ be an ordered alphabet of size $\sigma$ whose smallest element is the so-called sentinel character \$. In the following, $S$ is a string of length $n$ on $\Sigma$ having the sentinel character at the end (and nowhere else). For $1 \leq i \leq n$, $S[i]$ denotes the character at position $i$ in $S$. For $i \leq j$, $S[i..j]$ denotes the substring of $S$ starting with the character at position $i$ and ending with the character at position $j$. Furthermore, $S_i$ denotes the $i$-th suffix $S[i..n]$ of $S$. The suffix array $SA$ of the string $S$ is an array of integers in the range 1 to $n$ specifying the lexicographic ordering of the $n$ suffixes of $S$, that is, it satisfies $S_{SA[1]} < S_{SA[2]} < \cdots < S_{SA[n]}$.

The suffix array $SA$ is often enhanced with the so-called LCP-array containing the lengths of longest common prefixes between consecutive suffixes in $SA$. Formally, the LCP-array is an array so that $LCP[1] = -1 = LCP[n + 1]$ and $LCP[i] = |lcp(S_{SA[i-1]}, S_{SA[i]})|$ for $2 \leq i \leq n$, where $lcp(u, v)$ denotes the longest common prefix between two strings $u$ and $v$. The Burrows-Wheeler

\footnote{http://code.google.com/p/libdivsufsort/}
\footnote{http://researchmap.jp/muuw41s7s-1587/#_1587}