Linear time semi-external Burrows-Wheeler transform

Mikko Rautiainen

Introduction

Burrows-Wheeler transform is a transform that rearranges a string into a form that is usually more compressible. BWT is a reversible transform that doesn't add any extra data for strings which have a sentinel character, a unique end-of-string character smaller than any other character. BWT is used in bioinformatics and data compression, for example the bzip2 compression algorithm uses BWT.

The implementation is based on an algorithm described in the paper "Space-Efficient Construction of the Burrows-Wheeler Transform" by Beller, Zwerger, Gog and Ohlebusch. The important parts of the algorithm are its linear runtime and its low asymptotic memory requirements, one byte of RAM for one byte of input. The algorithm achieves the low memory requirements by storing certain data in the hard drive. Parts of the algorithm can be made to run faster when the algorithm is allowed to use extra memory. The algorithm's extra memory usage is constant and selected by the user when running the algorithm.

Algorithm

The implementation is based on the algorithm described in the paper. For more details, consult the paper.

The algorithm divides the input into L-type suffixes and S-type suffixes. An L-type suffix is a suffix that is lexicographically greater than the one after it, and an S-type suffix is lexicographically lesser than the one after it. Suffixes are never equal because the algorithm adds a sentinel character, a unique character smaller than any other character, to the end of the string. The sentinel character is always considered to be an S-type suffix. S-type suffixes which have an L-type suffix to their left are LMS-type suffixes (LeftMost S). Indices where an LMS-type suffix starts are LMS-indices. A substring that starts and ends at an LMS-type index is an LMS-substring.

The algorithm works in eight steps:

Step 1 passes through the entire input twice. On the first pass, the algorithm counts the number of times each symbol appears in the text. On the second pass, the algorithm finds all LMS-type indices, and outputs them in two lists. $A_{LMS,\ Left}$ has the indices such that the indices are ordered by the symbol where they point, and the order in which they appear in the text for indices which point to same symbols. $A_{LMS,\ Default}$ has the indices in the same order they appear in the text. The algorithm also returns $n_{L,}$, count of how many L-type suffixes start with each symbol, calculated in the second pass.

Step 2 takes the LMS-indices from $A_{LMS,\ Left}$ and walks through the L-parts of the LMS-substrings. The step uses a list A_L. First all LMS-indices from $A_{LMS,\ Left}$ are put into A_L. Next each element in the list A_L is processed one by one. For an element i, if the suffix at i-1 is also an L-type suffix, the element i-1 is put into the list. If it is not, i is outputted to the end of output $A_{LMS,\ Right}$.

Step 3 takes the indices from $A_{LMS,\ Right}$ and walks through the S-parts of the LMS-substrings. Again the step uses a list A_S. All indices from $A_{LMS,\ Right}$ are put into A_S and all elements in the list A_S are processed one by one. For an element i, if the suffix at i-1 is also an S-type suffix, i-1 is put into the list. Otherwise i is outputted to the end of A_{LMS}.

At this point, A_{LMS} contains all the LMS-indices in the order of their substrings.

Step 4 creates a renamed string S' from the LMS-indices. Each LMS-substring is given a new symbol and for each LMS-index in A_{LMS}, the symbol of its substring is put into S'.

Step 5 either recursively calculates the Burrows-Wheeler transform of S' if there are duplicate symbols in S', or directly using a counting sort if all symbols in S' are unique. The result is returned as BWT'

Step 6 has two parts. Part A calculates the inverse suffix array of S' using BWT'. Part B uses the inverse suffix array and $A_{LMS,\ Default}$ calculated in step 1 and outputs a list $A_{LMS,\ Sorted}$ of the LMS-indices.

At this point $A_{LMS,\ Sorted}$ has the LMS-indices in the order of their suffixes.

Step 7 does the same as step 2, except that it takes a list BWT where the output of the algorithm will be written. When an element is taken from the list A_L and another inserted, the algorithm inserts the correct character into the finished BWT string.

Step 8 does the same as step 3, except that it takes a list BWT where the output of the algorithm will be written. When an element is taken from the list A_S and another inserted, the algorithm inserts the correct character into the finished BWT string.

The paper describes a memory optimization for reducing memory usage. Some of the lists have a special usage pattern. Elements are inserted into them in random order, but they are removed sequentially and elements are never inserted before the last removed element. This can be used to create an external priority queue structure: the list is split into blocks of size n, and each block except the current one is kept on the hard drive. When inserting an element into a location, both the element and the location are written into the file representing the block. When all elements kept in memory are removed, the file for the next block is read and the elements are written into memory to their correct positions. Lists $A_{LMS,\ Left}$, A_L, A_S, S' and BWT use the external priority queue structure. This enables them to be kept mostly in hard drive with just a constant memory overhead.

The implementation contains two modes: in-files mode uses the memory optimization described above and in-memory keeps all data in memory. The implementation uses the null character \0 as the sentinel character, and works only on inputs which don't contain that character. The full code of the implementation is available at a GitHub repository[1].

Experiments

Experiments were run on the CS department's ukko cluster. Each datapoint is an average of four runs. The only modification to the algorithm was putting temporary files at /tmp instead of the working directory to reduce slowdowns from communicating with the file server. The program was compiled with optimization options "-O3 -DNDEBUG" for each experiment. Scripts for generating the data and running each experiment are in the GitHub repository.

The data was divided into four types. Genome is samples from the human genome. Quote is samples from wikiquotes. Random is random data from /dev/urandom. Repeat is the character 'a' repeating. Genome had files with sizes 10 Mb, 50 Mb, 100 Mb, 250 Mb and 500 Mb. Quote had 10 Mb, 50 Mb, 100 Mb and 250 Mb. Random had 10 Mb, 50 Mb and 100 Mb. Repeat had 10 Mb, 50 Mb and 100 Mb. The genome and quotes data was chosen because they are examples of real world data. Random data was chosen because it will probably lead to a high running time, note however

1 https://github.com/maickrau/pspa2014BWT

that it is probably not worst case data for the algorithms running time. Repeat data was chosen as a best case for the algorithms running time.

First experiment was testing the performance of the in-files mode using varying amounts of extra memory. Each data file was run with extra memory of 25%, 50%, 75%, 100%, 125%, 150%, 200%, 250%, 300% and 400% of the size of the input data. For example, at 25% extra memory and 10 Mb input file, the program was allowed to use 2,5 Mb of extra memory.

The second experiment was testing the slowdown of using in-files mode compared to in-memory mode. 50 Mb files of each type were run on both in-files and in-memory mode. The in-files mode was given extra memory of 300% of the input data's size.

The third experiment measured how much time each part of the algorithm takes. Running time was measured using GNU profiler by compiling the program with options "-g -pg" and using the gprof tool. Running time was measured on in-files mode on the 50 Mb genome file, using 25% extra memory and 200% extra memory.

Results

Experiment one: The graphs below show how much time the program took compared to the fastest running time. Every data point is normalized so that 1 is the fastest time for that file. At low amounts of extra memory, the program ran slower with each data type. The effects of extra memory varied depending on data type.

For genome data (figure 1), performance increased with increasing memory until about 200% extra memory, where it plateaued. Using only 25% extra memory took about 30% more time than running at the fastest amount of extra memory.

For quote data (figure 2), performance increased with more memory all the way to 400% extra memory. Using only 25% extra memory took about 25% more time than running at the fastest amount of extra memory, except for the 10 Mb file where the difference was more noticable at 40%.

For random data (figure 3), increasing memory lead to steadily increasing performance. At 25% extra memory, the program took about 25-30% more time than at maximum memory.

For repeat data (figure 4), increasing memory lead to rapidly increasing performance until around 200% extra memory. At 400% extra memory, performance was actually worse than at 200% extra memory. Using only 25% extra memory increased the running time by about 50%, much more than for the other data types.

Experiment two: Figure 5 shows running time for each 50 Mb file using in-memory mode and in-files mode. The table below shows the difference both as a percentage increase from in-memory mode and an absolute difference in running time. Using in-files mode had a drop in performance for all data types. The data shows that file I/O has a noticable overhead, however the overhead seems to be closer to a constant than a percentage increase. The overhead did not vary noticably between the two real world data types, genome and quote. The different data types also had very different running times.

File	Percentage difference	Absolute difference
Genome 50Mb	35%	13 s
Quotes 50Mb	33%	14 s
Random 50Mb	15%	9,7 s
Repeat 50Mb	170%	4 s

Experiment three: Figures 6 and 7 and the table below show percentage of running time for each measured step. Unfortunately, gcc's optimization inlined steps 4-6 and so the profiler couldn't gather data from those steps. The profiler could still measure time spent in steps 1, 2, 3, 7 and 8 but the other steps are all lumped into one section in the chart. It can be seen that steps 7 and 8 were much slower than steps 2 and 3 despite being almost identical. Surprisingly, even though step 3 is slower than step 2, step 7 is slower than step 8. The time taken for each step did not vary between using 25% extra memory and using 200% extra memory.

	25% extra memory	200% extra memory
Step 1	4,78%	5,43%
Step 2	8,49%	8,30%
Step 3	12,69%	12,93%
Step 7	21,37%	19,79%
Step 8	17,95%	17,34%
Other	34,72%	36,22%

Sources

T. Beller, M. Zwerger, S. Gog, E. Ohlebusch: *Space-Efficient Construction of the Burrows-Wheeler Transform*, Springer Lecture Notes in Computer Science Volume 8214, 2013, pages 5-16

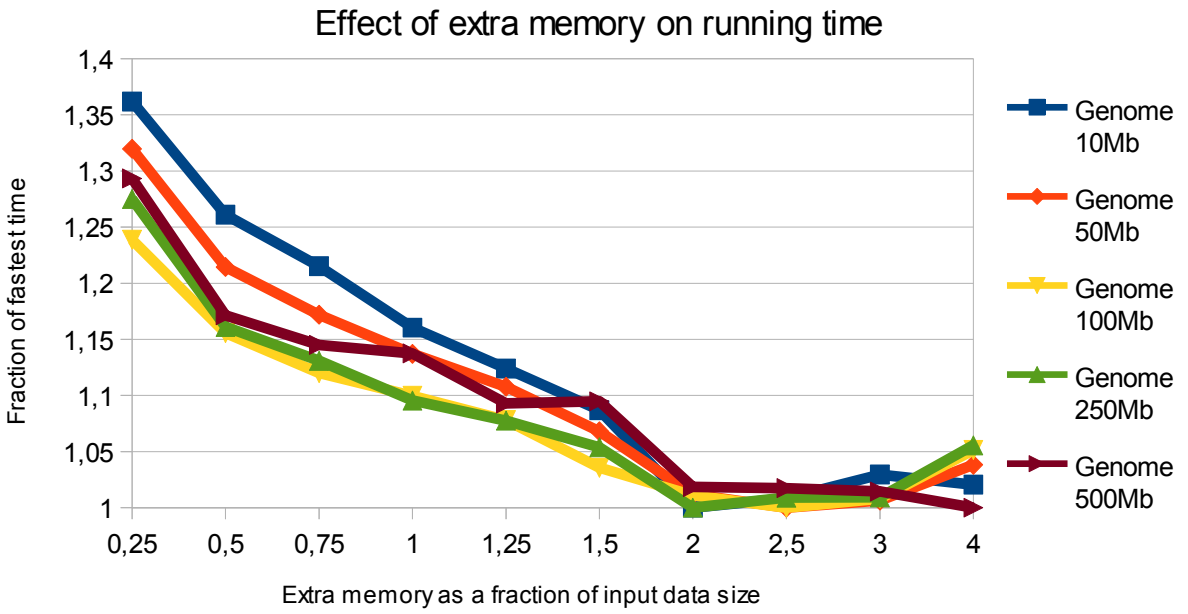

Figure 1

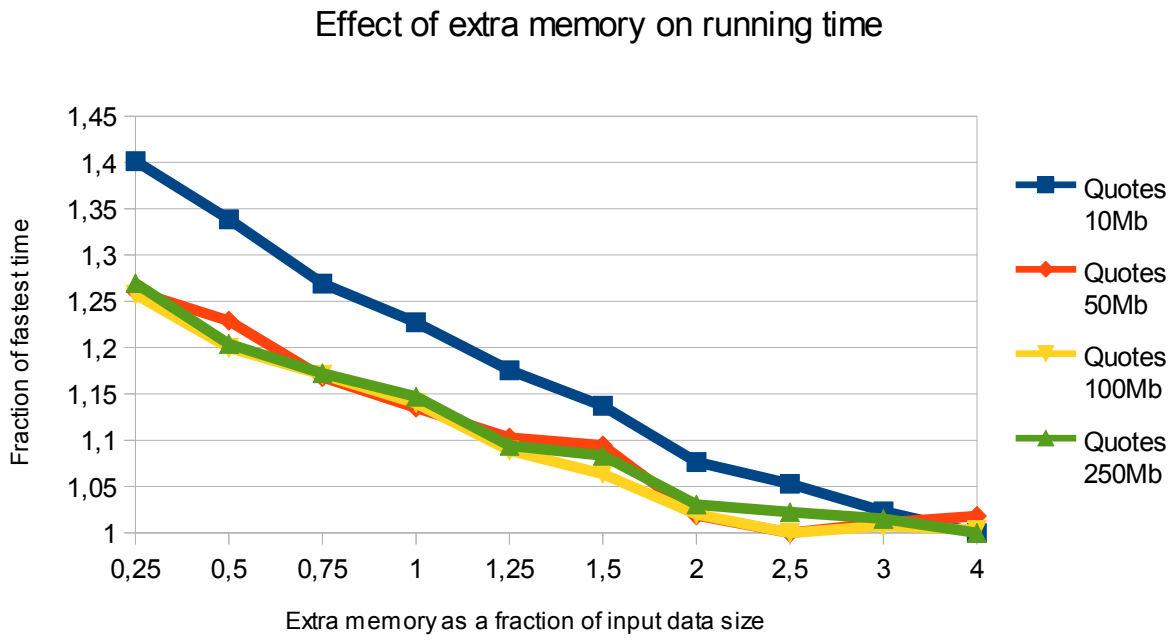

Figure 2

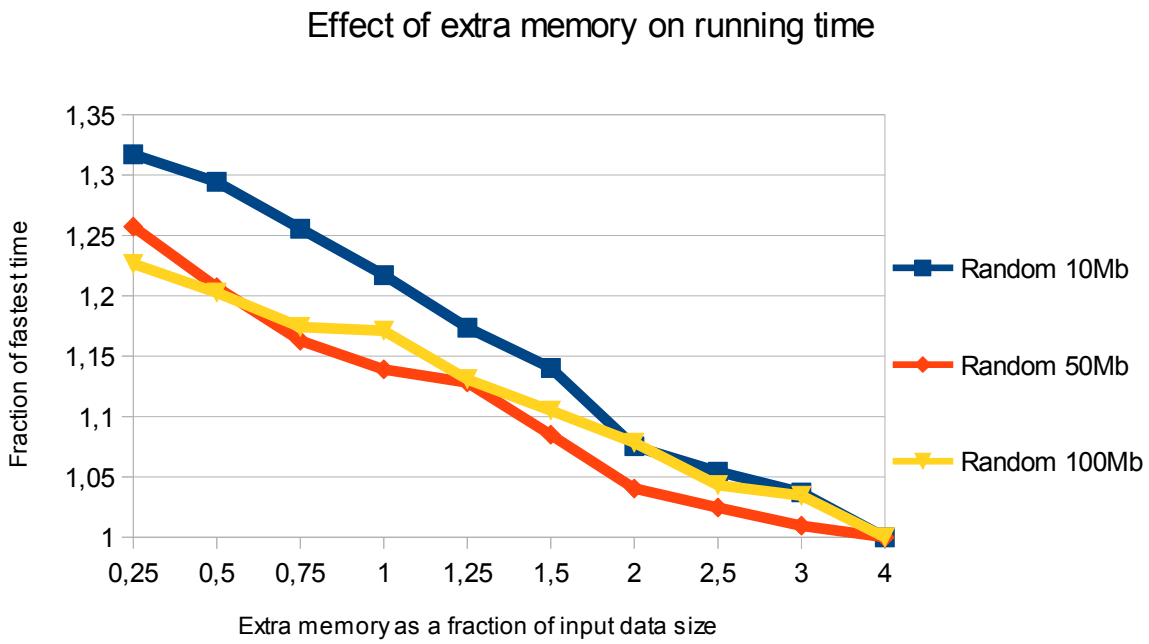

Figure 3

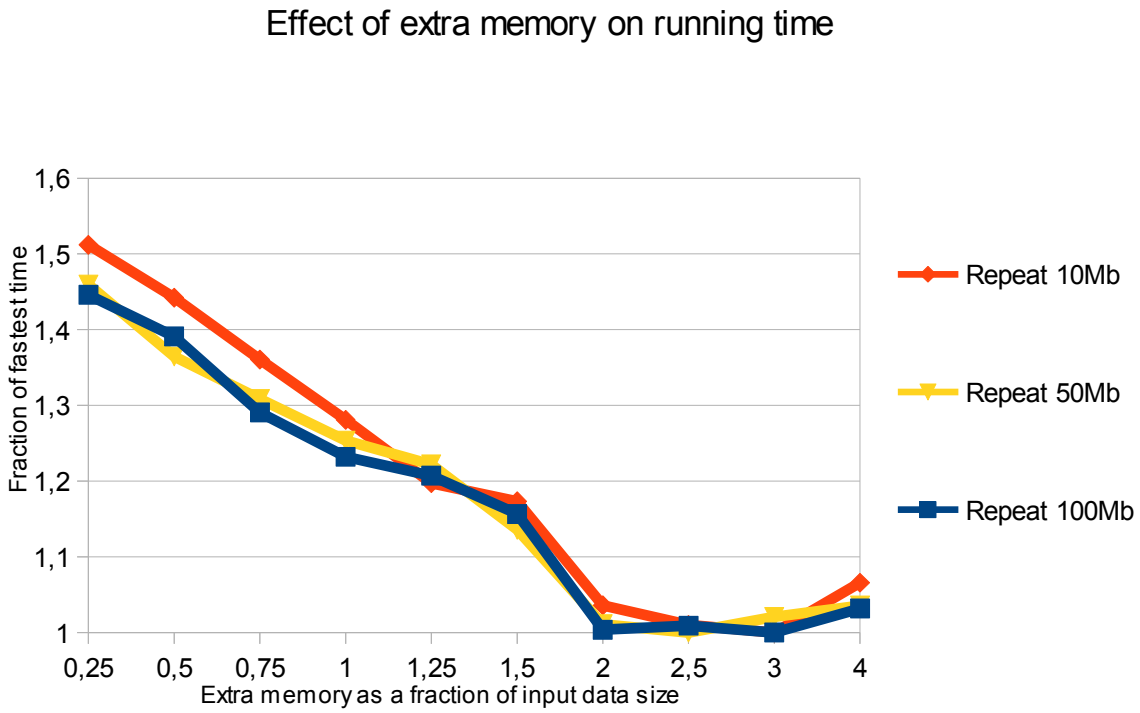

Figure 4

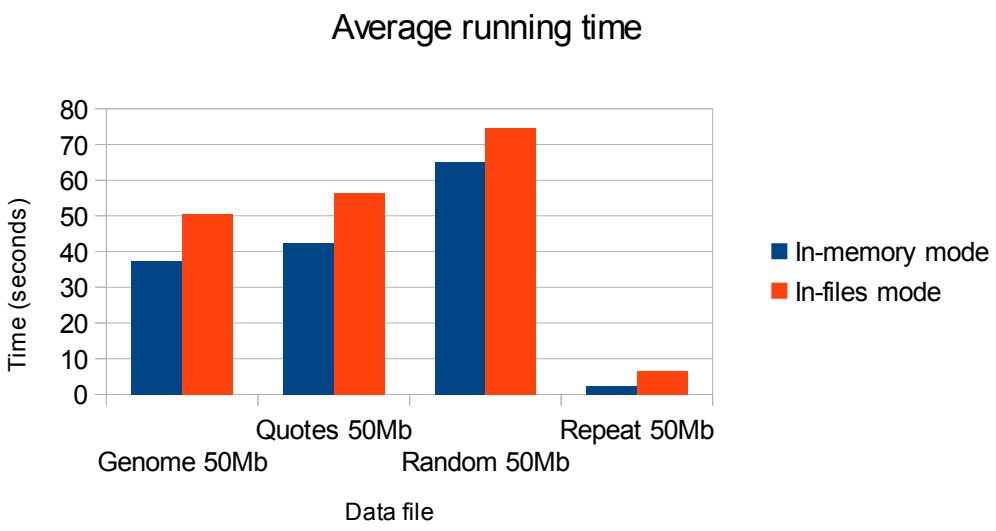

Figure 5

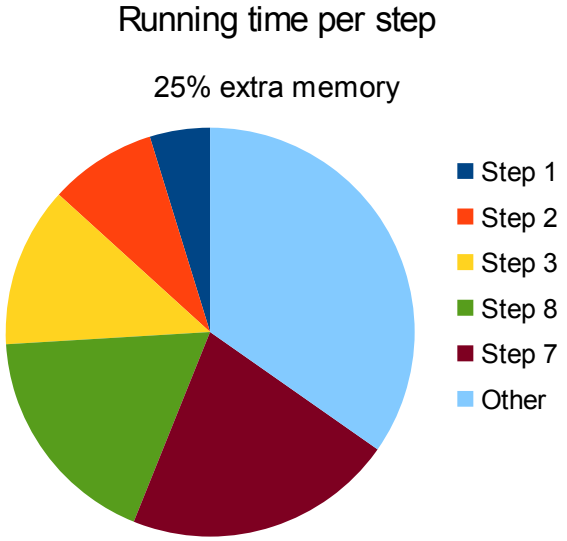

Figure 6

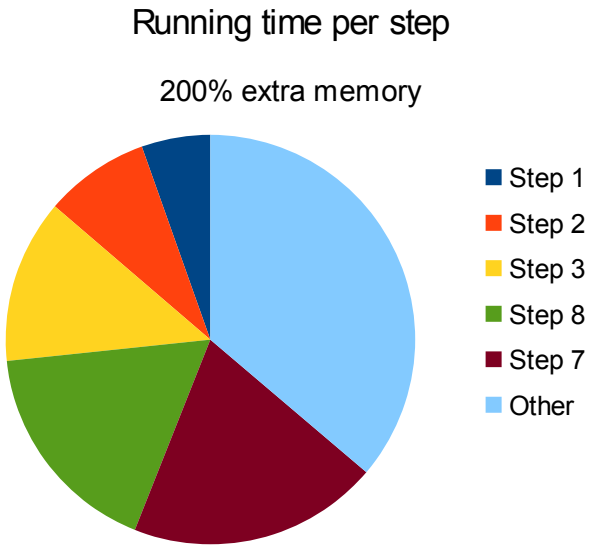

Figure 7