COSC 2306 Search Technology

Project Assignment, Semester 2, 2009

Overview

The *wavelet tree* [Grossi et al., 2003] is a data structure for representing a sequence $T[1..n]$ over an alphabet Σ of size σ. With a wavelet tree, one can compute $T[i]$ in time $O(\log \sigma)$, and also support the fundamental operations:

- $\mathrm{rank}_c(T, i)$, the number of occurrences of symbol c in $T[1..i]$; and
- $\mathrm{select}_c(T, j)$, the position in T of the j-th occurrence of symbol c.

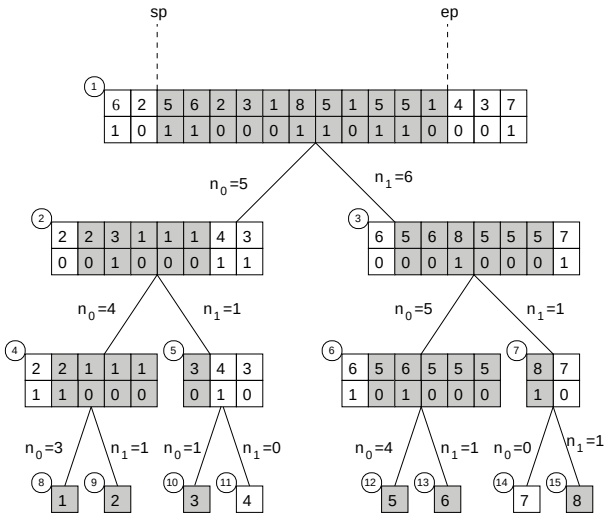

Figure 1: $T = \{6, 2, 5, 6, 2, 3, 1, 8, 5, 1, 5, 5, 1, 4, 3, 7\}$ as a wavelet tree. The top row of each node shows T, the second row the bitvector B_v, and numbers in circles are node numbers for reference in the text. n_0 and n_1 are the number of 0 and 1 bits respectively in the shaded region of the parent node of the labelled branch. Shaded regions show the parts of the nodes that are accessed when listing elements in the region $T[sp = 3..ep = 13]$. Note that only the bitvectors (preprocessed for rank and select) are present in the actual structure, the numbers above each bitvector are included only to aid explanation.

An example of a wavelet tree is shown in Figure 1, and has a structure as follows. At the root, we divide the alphabet Σ into symbols $< c$ and $\geq c$, where c is the median of Σ (assuming Σ can be ordered). Then store bitvector $B_{\mathrm{root}}[1..n]$ in the root node, where $B_{\mathrm{root}}[i] = 0$ if $T[i] < c$ and 1 otherwise. In the example, $\sigma = 8$, and so items 1,2,3 and 4 receive a 0-bit, and 5,6,7, and 8 get a 1-bit in the root node. Now the left child of the root will handle sequence T_{left}, formed by concatenating together all the symbols $< c$ in $T[1..n]$ (respecting the order); and the right child will handle T_{right}, which has the symbols $\geq c$. At the leaves, where all the symbols of the corresponding

T_{leaf} are equal, nothing is stored. (In Figure 1 the corresponding element of T in each leaf is shown for clarity.) It is easy to see that there are $\lceil \log \sigma \rceil$ levels and that n bits are spent per level, for a total of at most $n\lceil \log \sigma \rceil$ bits.

The compressed bitvector representation also allows us to obtain $B[i]$, and to compute rank and select, in constant time over the bitvectors, which enables the $O(\log \sigma)$-time corresponding operations on sequence T. To obtain $T[i]$, first inspect $B_{\text{root}}[i]$. If it is 0, continue recursively on the left child with position $i' = \text{rank}_0(B_{\text{root}}, i)$, otherwise to the right with $i' = \text{rank}_1(B_{\text{root}}, i)$. If a leaf is encountered, then we have found symbol $T[i]$. To compute $\text{rank}_c(T, i)$, if c belongs to the left child of the root, then go left with $i' = \text{rank}_0(B_{\text{root}}, i)$, else right with $i' = \text{rank}_1(B_{\text{root}}, i)$. When arriving at a leaf, the current i value is the answer. Finally, to compute $\text{select}_c(T, j)$ start at the leaf for c. If it is a left child of its parent B_v, continue in B_v with $j' = \text{select}_0(B_v, j)$, else with $j' = \text{select}_1(B_v, j)$. When reaching the root, the current j value is the answer.

Getting Started

Your first task is to implement a wavelet tree as a series of bitvectors which supports a variable length text collection (character-based) as input. Your representation should support the $\text{rank}_c(T, i)$ and $\text{select}_c(T, j)$ operations across any partitioning of the T character array. More specifically, your rank and select functions should support arbitrary ep and sp boundaries as shown in the example. This should be accomplished by supporting binary rank and select over the aggregated bitvector levels. The representation should support binary rank and select operations to count the number of 1-bits and 0-bits in a bitvector. The simplest way to support these operations is to implement a folklore method called *popcounting*. A trivial algorithm is provided here to get started:

```
/* Count the number of 1-bits in an array of integers.
 * INPUT: A word-aligned bitvector of len * sizeof(int) bits.
 * OUTPUT: The total number of 1-bits in the bitvector
 */
int pop (unsigned int *A, int len) {
  int n = 0, i;
  for (i = 0; i < len; i++) {
    unsigned x = A[i];
    while (x != 0) {
      n++;
      x = x & (x - 1);
    }
  }
  return (n);
}
```

This will return the number of 1-bits set in a bitvector represented as a word-aligned array of integers. There are many alternative (and better) methods to implement popcounting. Find one that works well experimentally. Once you have a basic wavelet tree working, quantify the storage and operator efficiency experimentally and provide theoretical bounds if possible. Compare the basic counting operations implemented in your wavelet tree against an on-line stable sort, or other counting method of your choice. You should also implement some sort of scanning baseline to mimic select.

Experimentally compare your wavelet tree against other baselines for different text lengths and alphabet sizes. What space and time trade-offs are possible? An excellent reference paper to gain further insight is:

```
%% The definitive full-text indexing survey.
@article{nm07-csurv,
  author =    {G. Navarro and V. M{\"{a}}kinen},
  title =     {Compressed full-text indexes.},
  journal =   {ACM Computing Surveys},
  year =      {2007},
  volume =    {39},
  number =    {1},
  pages =     {2-1 -- 2-61},
  publisher = {ACM Press},
  address =   {New York, NY, USA},
  annote =    {The self-indexing survey.},
}
```

Path 1 - Time and Space Improvements

Surprisingly, binary rank and select can be implemented in constant time by using a bounded amount of extra space. This is a classical time and space trade-off, and well worth the effort to understand. See section 6 of Navarro and Mäkinen [2007] and the references therein for a description of Clark's original algorithm, along with various improvements. The details of this algorithm will be covered in the lecture on week 9 (September 29).

Your next task is to implement constant time binary rank and select across the bitvector levels in your implementation. Also, you should implement additional variations of popcounting and amortized counting in order to adequately assess the performance impact of the new methods. You should also consider alternatives to offset the additional memory requirements these methods impose on a wavelet tree. An excellent experimental study of minimizing space for these key operations was recently done by Claude and Navarro [2008]. Experimentally (and theoretically) compare the time and space trade-offs of the methods you implement.

Path 2 - Index Construction

The next task is to optimize the construction time of your wavelet tree implementation. To my knowledge, there are no published results on this specific problem. In other words, this is a legitimate research problem. A clever solution to this task is probably a publishable research result. Possible solutions in increasing order of difficulty are:

1. Multi-pass construction in $\mathcal{O}(n \log \sigma)$ time and $\mathcal{O}(n \log \sigma)$ space.
2. Two-pass construction in $\mathcal{O}(n \log \sigma)$ time and $\mathcal{O}(n \log \sigma)$ space.
3. On-line (one-pass) construction in $\mathcal{O}(n \log \sigma)$ time and $\mathcal{O}(n \log \sigma)$ space.

Is it possible to construct a wavelet tree in a single pass in $\mathcal{O}(n)$ time and/or sub-linear space (via external memory)? Provide a solution (or a proof of impossibility)

meeting this requirement, and you will ensure yourself of the maximum mark (and possibly an interesting publishable result too!). A good place to get inspiration for possible lines of attack for this problem can be found in any textbook covering inverted index construction (see for instance, Witten et al. [1999]).

Path 3 - Dynamic Operations

The final path to consider is efficient support of the dynamic operations: insert and delete. These operations are defined as:

- $\text{insert}_c(T, x, i)$: insert character x between $T[i]$ and and $T[i+1]$.
- $\text{delete}_c(T, i)$: delete $T[i]$ from T.

There are a few recent theoretical results on this problem, but practical comparisons are still relatively rare. A good starting point is work of Lee and Park [2007] and González and Navarro [2008]. Implement at least two alternative approaches for insert and delete and compare the results experimentally. Does the addition of dynamic operations have an impact on the performance on your static implementations of rank and select?

Assessment

Hand in a written report (max 8 pages, 11pt font) detailing any successful investigations you carried out into the characteristics of the sample text collection, a description of the approaches that you implemented, plus any "additional enhancements" that you came up with, the experiments that you have carried out, and the results you have achieved. Do not hand in a hard copy of your source code. As part of your submission, (tar up and) mail to me as an attachment your source code (plus a makefile, if necessary) so that if I want to I can compile and benchmark your speed on a common hardware platform (and perhaps using a larger test file).

This is not a group project. Everyone should do their own work. If you use snippets of code not written by you (for instance, an efficient popcounting algorithm), it should be clearly documented in the source code. Tread very carefully if you take this path. A Huffman-shaped wavelet tree is *not* an acceptable solution for this project. Please do not submit complete wavelet tree implementations you did not write. I have seen all of the implementations available on the Internet, so it is in your best interest not to be tempted by the path of least effort.

Submissions should use LaTeX or equivalent, must be in an academic style, and must include a bibliography listing any resource material that you have used (which is why I gave you a sample `BibTeX` entry for one of the key papers). A fraction of the mark will be assigned to presentation. Sloppy or careless formatting – such as is pretty much forced on you by Microsoft Word – will be penalised. In other words, use Word if you must, but be sure that you use it carefully. It is not a good tool for academic writing.

Students aiming for nothing more than a minimal pass mark (30/60) need to complete all of the tasks outlined in the "Getting Started" section, *and* do them well. However most of you want more than a minimal pass, and should aim to complete one or more of the alternative exploration paths. Completing the minimal requirements and one additional exploration path will result in at least a credit (36/60). Completing two

of the three alternative paths will increase the mark to at least a distinction (42/60). Making significant progress on two of the three paths and presenting a reasonable experimental analysis will result in a mark as high as 51/60. If you wish to go beyond a 51, you will need to solve two of the three alternative problems (construction, storage, or dynamic operations) with originality, *and* possess a strong understanding of the theoretical underpinnings of the respective problems.

Getting Help

My office hours are every Tuesday 3:30pm – 4:30pm. In addition, I will be available from 10:30am – 12:30pm on September 29 and October 6. As we will not have a lecture on these days, it is possible that we will schedule individual progress meetings during these time slots. Otherwise, you should use the time to work on your project or come and talk to me if you get stuck.

Deadline

Hand in a hardcopy of your report (no program source code, I want that and a softcopy of your report emailed to me) by 5pm on October 9.

References

F. Claude and G. Navarro. Practical rank/select queries over arbitrary sequences. In A. Amir, A. Turpin, and A. Moffat, editors, *Proceedings of the 15th International Symposium on String Processing and Information Retrieval (SPIRE 2008)*, volume 5280 of *LNCS*, pages 176–187. Springer, November 2008.

R. González and G. Navarro. Improved dynamic rank-select entropy-bound structures. In E. S. Laber, C. F. Bornstein, L. T. Nogueira, and L. Faria, editors, *Proceedings of the 8th Latin American Symposium on Theoretical Informatics (LATIN 2008)*, volume 4957 of *LNCS*, pages 374–386. Springer, April 2008.

R. Grossi, A. Gupta, and J. S. Vitter. Higher-order entropy-compressed text indexes. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2003)*, pages 841–850, January 2003.

S. Lee and K. Park. Dynamic rank-select structures with applications to run-length encoded texts. In B. Ma and K. Zhang, editors, *Proceedings of the 18th Annual Symposium on Combinatorial Pattern Matching (CPM 2007)*, volume 4580 of *LNCS*, pages 95–106. Springer, June 2007.

G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):2–1 – 2–61, 2007.

I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, San Francisco, second edition, 1999.