6.897: Advanced Data Structures Spring 2005

Lecture 21 — April 21, 2005

Prof. Erik Demaine *Scribe: Yoyo Zhou*

1 Overview

In this lecture, we consider data structures using close to the information theoretic space. We survey some existing low-space results, including for suffix trees and arrays, which will be the focus of next lecture. We also describe RANK and SELECT, two primitives using $o(n)$ space that are often used in succinct data structures. We also discuss the isomorphism between binary tries, rooted ordered trees, and balanced parentheses.

2 Preliminaries

Suppose that Z is the information-theoretic optimum number of bits to store some data.

Definition 1. *An **implicit data structure** uses only Z bits. An alternative definition is that it stores only its input data, arranged in some order.*

Definition 2. *A **succinct data structure** uses only $Z + o(Z)$ bits.*

Definition 3. *A **compact data structure** uses only $O(Z)$ bits.*

Some results on low-space data structures are summarized below. All except the first one are for static data structures.

- implicit dynamic search trees [FG03]: support $O(\log n)$ time worst-case insert, delete, and predecessor/successor queries, in the comparison model. They store just some permutation of the input keys.
- succinct dictionaries [BM99], [Pag01]: use $\lg \binom{u}{n} + O(\frac{n(\lg\lg n)^2}{\lg n})$ bits, and support $O(1)$ membership queries; u is the size of the universe from which the n elements are drawn.
- succinct binary tries [MR01]. There are $C_n = \binom{2n}{n}/(n+1) \sim 4^n = 2^{2n}$ distinct binary tries. This data structure uses $2n + o(n)$ bits and supports $O(1)$ left child, right child, parent, and subtree size queries.
- compact k-ary trie [BDMRRR05]. There are $C_n^k = \binom{kn+1}{n}/(kn+1)$ distinct k-ary tries; $\lg C_n^k \sim (\lg k + \lg e)n$. This structure uses $(\lceil \lg k \rceil + \lceil \lg e \rceil)n + o(n)$ bits (very close to succinct, for large k) and supports $O(1)$ child-of-label-i, parent, and subtree size queries.
- succinct ordered rooted tree [BDMRRR05]. There are C_n distinct ordered rooted trees. The data structure uses $2n+o(n)$ bits and supports $O(1)$ i^{th} child, parent, and subtree size queries.

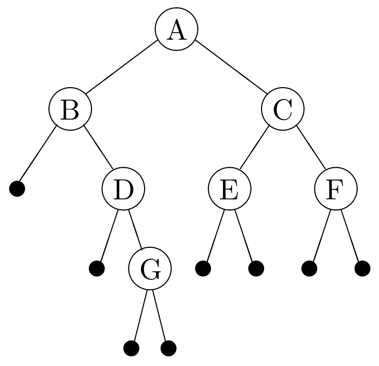

Figure 1: A binary trie, with external nodes attached.

3 Representation of Binary Tries

To motivate our future work, we introduce a space-efficient representation of binary tries. Given a binary trie, its ***level order representation*** is defined by first attaching external nodes, and then traversing it across each level and appending a 1 for an internal node and a 0 for an external node. This is equivalent to an initial 1, followed by traversing the tree across levels, appending 1 or 0 for whether the left child exists, and 1 or 0 for whether the right child exists. For the example tree from Figure 1, the level order representation is 111011101000000. This representation needs just $2n + 1$ bits, which is very close to optimal.

For the representation to be useful, we must be able to navigate it efficiently. We will show how to find the left and right children, as well as the parent of any given node. Say our node is the i^{th} internal of the trie. Then, we show its children are at positions $2i$ and $2i + 1$. Assume the original node was at position $i + j$, i.e. there are j external nodes before the i^{th} internal node. The $i - 1$ previous internal nodes have $2(i - 1)$ children. Of these, $i - 1$ have appeared before as internal nodes, and j appeared as external nodes. Then, the left child of our current node is at position $(i + j) + 2(i - 1) - (i - 1) - j + 1 = 2i$.

4 RANK and SELECT

We first define two useful functions on bit sequences, and describe implementations with $O(1)$ query time and $o(n)$ space.

Definition 4. RANK(i) *is the number of 1's at or before index i.*

Definition 5. SELECT(i) *is the index of the* i^{th} *1-bit.*

Using RANK and SELECT, we can easily define the navigation functions for binary tries. Starting with the node on position k, we have:

- LEFT-CHILD$(k) = 2 \cdot$ RANK(k)
- RIGHT-CHILD$(k) = 2 \cdot$ RANK$(k) + 1$
- PARENT$(k) =$ SELECT$(\lfloor k/2 \rfloor)$

4.1 Implementation of RANK [Jac89]

The main idea is to use indirection, and use lookup tables on strings of $\frac{1}{2}\lg n$ bits. Since there are $\sqrt{n}$ distinct strings of this length, $\frac{1}{2}\lg n$ possible queries for each, and $\lg\lg n$ bits to store the answer, the space used is $O(\sqrt{n}\lg n \lg\lg n)$ bits.

We would like to break the n-bit string into segments and store the rank at the beginning of each segment; however, with $O(n/\lg n)$ segments, it would require $\lg n$ space to store each rank, which makes space usage linear. Instead, first we break the n-bit string into $\lg^2 n$-bit chunks; it takes $O(n/\lg n)$ space to store the rank at the beginning of each chunk. Then break each $\lg^2 n$-bit chunk into $\frac{1}{2}\lg n$-bit segments. Now storing the rank within the chunk at the beginning of each segment only costs $O(\lg\lg n)$ bits of space, because the value is at most $\lg^2 n$. This requires a total space of $O(\frac{n}{\lg n}\lg\lg n) = o(n)$. Rank can be computed as the sum of the chunk value, the segment value, and then the value from the lookup table.

4.2 Implementation of SELECT [CM96]

We split the string into buckets such that each contains $\lg n \lg\lg n$ 1's (except possibly the last one, which can have fewer). An array stores the index of the beginning of each bucket; it uses $O(\frac{n}{\lg n \lg\lg n}\lg n) = O(\frac{n}{\lg\lg n})$ space. (Reducing this size is an open problem.)

Say that the size of a bucket is r. If $r \geq (\lg n \lg\lg n)^2$, then store the bucket as an array whose i^{th} element is the index of the i^{th} 1. Then the space used is $O(\lg n \lg\lg n \cdot \lg n)$ for each bucket of at least $(\lg n \lg\lg n)^2$ bits. Thus, the amortized (per bit) space used is $O(1/\lg\lg n)$, so the total space is $O(n/\lg\lg n)$.

Otherwise, we know $\lg n \lg\lg n \leq r \leq (\lg n \lg\lg n)^2$. We recurse this structure inside the bucket, using r in place of n. After a constant number of recursions, we reach a range of at most $\frac{1}{2}\lg n$, so we can store a lookup table using sublinear space (similar to RANK).

5 Binary Tries, Rooted Ordered Trees, and Balanced Parentheses

It is no coincidence that for a given n, there are the same number (the n^{th} Catalan number, C_n) of binary tries, rooted ordered trees, and balanced parentheses expressions; isomorphisms exist between them. To convert a binary trie into a rooted ordered tree (with an extra node), convert right paths into children, in order, of the parent of the top of the path; the root's right path becomes the child of an extra node. To convert a rooted order trie into a balanced parentheses expression, traverse the tree depth-first, writing a "(" the first time a node is visited and ")" for the last time it is visited. It is not difficult to show that inverses of these transformations exist, so that they are isomorphisms.

We can then associate each node in the binary trie with a node in the rooted ordered tree and the left parenthesis from when it was first visited in the balanced parentheses. This allows us to mkae associations in Table 1.

The result in the last row is important because it allows LEAF-RANK and LEAF-SELECT to be defined using RANK$_P$ and SELECT$_P$, which are RANK and SELECT with respect to a fixed pattern P, where

binary trie	rooted ordered tree	balanced parentheses
node x	node x	left paren. X
left child of x	first child of x	$X+1$
right child of x	next sibling of x	X's matching right paren. $+ 1$
parent of x	previous sibling of x or parent of x	matching left paren. of $X-1$ or $X-1$
size of x's subtree	size of subtree of x and siblings to its right	$\frac{1}{2}$(parent's matching right paren. $-X$)
x is a leaf	x is a last sibling and a leaf	X begins "())"

Table 1: Associations between structural elements.

$|P| = O(1)$; this can also be done in sublinear space [MRR03]. Then the leftmost leaf in a node x's subtree can be found as LEAF-SELECT(LEAF-RANK(x) + 1), and the rightmost leaf similarly.

6 Survey of Compact Suffix Trees and Arrays

There are several interesting results on small-space suffix trees/arrays, some of which will be covered in the next lecture. Let T be the text, P be the pattern, and Σ the alphabet. An important open problem is whether there exists a structure which uses $O(|T|)$ space and can search in $O(|P)$ time.

- Grossi and Vitter [GV00] give a structure using $(\frac{1}{\epsilon}+O(1))|T|\lg|\Sigma|$ bits of space and $O(\frac{|P|}{\lg_{|\Sigma|}|T|}+\lg^{\epsilon}_{|\Sigma|}|T|\cdot|output|)$ query time.
- Ferragina and Manzini [FM00] give an "opportunistic data structure" using $O(H_k(T)|T| + \frac{|T|}{\lg|T|}|\Sigma|\lg|\Sigma|) + o(|T|)$ bits of space and $O(|P| + \lg^{\epsilon}|T|\cdot|output|)$ query time. Here H_k is k^{th}-order entropy, the optimal compression rate using a sliding windows of size k. This result holds for any constant k.
- Sadakane [Sad03] gives the best known result for large alphabets: $O(H_0(T)|T| + |\Sigma|\lg|\Sigma|) + o(|T|)$ bits and $O(|P|\lg|T| + \lg^{\epsilon}|T|\cdot|output|)$ query time.
- For low-space construction, Hon, Sadakane, and Sung [HSS03] show a suffix tree construction using $O(|T|\lg|\Sigma|)$ working space and $O(|T|\lg^{\epsilon}|T|)$ time. An earlier result by Hon and Sadakane [HS02] gives array-like behavior.

References

[BDMRRR05] D. Benoit, E. Demaine, J. Munro, R. Raman, V. Raman, S. Rao: *Representing Trees of Higher Degree*, Algorithmica (to appear) (2005).

[BM99] A. Brodnik, J. Munro: *Membership in Constant Time and Almost-Minimum Space*, SIAM Journal on Computing 28(5): 1627-1640 (1999).

[CM96] D. Clark, J. Munro: *Efficient Suffix Trees on Secondary Storage*, Symposium on Discrete Algorithms 1996: 383-391.

[FM00] P. Ferragina, G. Manzini: *Opportunistic Data Structures with Applications*, Foundations of Computer Science 2000: 390-398.

[FG03] Gianni Franceschini, Roberto Grossi: *Optimal Worst-Case Operations for Implicit Cache-Oblivious Search Trees*, WADS 2003: 114-126.

[GV00] R. Grossi, J. Vitter: *Compressed suffix arrays and suffix trees with applications to text indexing and string matching*, Symposium on the Theory of Computing 2000: 397-406.

[HS02] W. Hon, K. Sadakane: *Space-Economical Algorithms for Finding Maximal Unique Matches*, Combinatorial Pattern Matching 2002: 144-152.

[HSS03] W. Hon, K. Sadakane, W. Sung: *Breaking a Time-and-Space Barrier in Constructing Full-Text Indices*, Foundations of Computer Science 2003: 251-260.

[Jac89] G. Jacobson: *Space-efficient Static Trees and Graphs*, Foundations of Computer Science 1989: 549-554.

[MR01] J. Munro, V. Raman: *Succinct Representation of Balanced Parentheses and Static Trees*, SIAM Journal on Computing 31(3): 762-776 (2001).

[MRR03] J. Munro, V. Raman, S. Rao: *Space Efficient Suffix Trees*, Journal of Algorithms 39(2): 205-222 (2001).

[Pag01] R. Pagh: *Low Redundancy in Static Dictionaries with Constant Query Time*, SIAM Journal of Computing 31(2): 353-363 (2001).

[Sad03] K. Sadakane: *New text indexing functionalities of the compressed suffix arrays*, Journal of Algorithms 48(2): 294-313 (2003).