6.851: Advanced Data Structures Spring 2010

Lecture 8 — March 2, 2010

Prof. Erik Demaine

1 Overview

In the last lecture we studied suffix trees and suffix arrays. These two data structures are useful for answering a variety of queries on strings such as string matching, string frequency and longest repeating substring. They offer advantages over more generic data structures like binary search trees and hashes. We learned that a linear-time algorithm exists for converting between suffix trees and suffix arrays, and that there exists an algorithm for efficiently constructing suffix arrays, the DC3 algorithm. Finally, we studied a modification to suffix arrays that allows one to perform document retrieval queries in $O(|P| + d)$ time, where P is the pattern to search a set of documents for and d is the number of documents containing P.

In today's lecture two problems are discussed: Level Ancestor Query (LAQ) and Least Common Ancestor (LCA). LAQ will be covered fully today, while LCA will be completed next lecture. Both problems have the same setting: given a rooted tree T and a node v, find an ancestor of v with some property. For LAQ, we will study various approaches with different preprocessing and query times, culminating in a data structure with $O(n)$ preprocessing and $O(1)$ query time. For LCA, we will study a different but related problem (Range Minimum Query or RMQ) which will help us to solve LCA.

2 Least Ancestor Queries (LAQ)

First we introduce notation. Let $h(v)$ be the height of a node v in a tree. Given a node v and level l, $LAQ(v, l)$ is the ancestor a of v such that $h(a) - h(v) = l$. Today we will study a variety of data structures with various preprocessing and query times which answer $LAQ(v, l)$ queries. For a data structure which requires $f(n)$ query time and $g(n)$ preprocessing time, we will denote its running time as $\langle g(n), f(n)\rangle$. The following algorithms are taken from the set found in a paper from Bender and Farach-Colton[1].

2.1 Algorithm A: $\langle O(n^2), O(1)\rangle$

Basic idea is to use a look-up table with one axis corresponding to nodes and the other to levels. Fill in the table using dynamic programming by increasing level. This is the *brute force* approach.

2.2 Algorithm B: $\langle O(n \log n), O(\log n)\rangle$

The basic idea is to use **jump pointers**. These are pointers at a node which reference one of the node's ancestors. For each node create jump pointers to ancestors at levels $1, 2, 4, \ldots, 2^k$. Queries

are answered by repeatedly jumping from node to node, each time jumping more than half of the remaining levels between the current ancestor and goal ancestor. So worst-case number of jumps is $O(\log n)$. Preprocessing is done by filling in jump pointers using dynamic programming.

2.3 Algorithm C: $\langle O(n), O(\sqrt{n})\rangle$

The basic idea is to use a **longest path decomposition** where a tree is split recursively by removing the longest path it contains and iterating on the remaining connected subtrees. Each path removed is stored as an array in top-to-bottom path order, and each array has a pointer from its first element (the root of the path) to it parent in the tree (an element of the path-array from the previous recursive level). A query is answered by moving upwards in this *tree of arrays*, traversing each array in $O(1)$ time. In the worst case the longest path decomposition may result in longest paths of sizes $k, k-1, \ldots, 2, 1$ each of which has only one child, resulting in a tree of arrays with height $O(\sqrt{n})$. Building the decomposition can be done in linear by by precomputing node heights once and reusing them to find the longest paths quickly.

2.4 Algorithm D: $\langle O(n), O(\log n)\rangle$

The basic idea is to use **ladder decomposition**. The idea is similar to longest path decomposition, but each path is extended by a factor of two backwards (up the tree past the root of the longest path). If the extended path reaches the root, it stops. From the ladder property, we know that node v lies on a longest path of size at least $h(v)$. As a result, one does at most $O(\log n)$ ladder jumps before reaching the root, so queries are done in $O(\log n)$ time. Preprocessing is done similarly to Algorithm C.

2.5 Algorithm E: $\langle O(n \log n), O(1)\rangle$

The idea is to combine jump pointers (Algorithm B) and ladders (Algorithm D). Each query will use one jump pointer and one ladder to reach the desired node. First a jump is performed to get at least halfway to the ancestor. The node jumped to is contained in a ladder which also contains the goal ancestor.

2.6 Algorithm F: $\langle O(n), O(1)\rangle$

An algorithm developed by Dietz[2] also solves LCA queries in $\langle O(n), O(1)\rangle$ but is more complicated. Here we combine Algorithm E with a reduction in the number of nodes for which jump pointers are calculated. The motivation is that if one knows the level ancestor of v at level l, one knows the level ancestor of a descendant of v at level l'. So we compute jump pointers only for leaves, guaranteeing every node has a descendant in this set. So far, preprocessing time is $O(n + L \log n)$ where L is the number of leaves. Unforunately, for an arbitrary tree, $L = O(n)$.

2.6.1 Building a tree with $O(\frac{n}{\log n})$ leaves

Split the tree structure into two components: a **macro-tree** at the root, and a set of **micro-trees** (of maximal size $\frac{1}{4}\log n$) rooted at the leaves of the macro-tree. Consider a depth-first search, keeping track of the orientation of of the ith edge, using 0 for downwards and 1 for upwards. A micro-tree can be described by a binary sequence, e.g. $W = 001001011$ where for a tree of size n, $|W| = 2n - 1$. So an upper bound the number of micro-trees possible is $2^{2n-1} = 2^{2(\frac{1}{4}\log n)-1} = O(\sqrt{n})$. But this is a loose upper bound, as not all binary sequences are possible, e.g. $00000\ldots$. A valid micro-tree sequences has an equal number of zeros and ones and any prefix of a valid sequence as at least as many zeros as ones.

2.6.2 Use macro/micro-tree for a $\langle O(n), O(1)\rangle$ solution to LAQ

We will use macro/micro-trees to build a tree with $O(\frac{n}{\log n})$ leaves and compute jump pointers only for its leaves ($O(n)$ time). We also compute all microtrees and their look-up tables (see Algorithm A) in $O(\sqrt{n}\log n)$ time. So total preprocessing time is $O(n)$. A query $LAQ(v, l)$ is performed in the following way: If v is in the macro-tree, jump to the leaf descendant of v, then jump from the leaf and climb a ladder. If v is in a micro-tree and $LAQ(v, l)$ is in the micro-tree, use the look-up table for the leaf. If v is in a micro-tree and $LAQ(v, l)$ is not in the micro-tree, then jump to the leaf descendant of v, then jump from the leaf and climb a ladder.

3 Least Common Ancestor (LCA)

Now we move onto the second problem presented today: least common ancestor. For two nodes u, v in a rooted tree T, we define $LCA(u, v)$ to be the node in T with the minimum height that is an ancestor of both u and v. A related problem is the **range minimum query (RMQ)** problem. For two indices i, j in an array A, we define $RMQ(i, j)$ to be the index of the smallest value in $A[i, j]$. Though these two problems may not appear to be very similar, we will see how they are related (in this and next lecture).

3.1 LCA and RMQ are related

Lemma 1. *A $\langle f(n), g(n)\rangle$ algorithm for RMQ gives a $\langle f(2n-1)+O(n), g(2n-1)+O(1)\rangle$ algorithm for LCA.*

Use depth-first search as an Euler tour of rooted tree T. Use three sequences (see Figure 1):

- $E = \{1, 2, 4, 2, 5, 8, 5, 9, 5, 2, \ldots\}$, the nodes in sequence visited during Euler tour.
- $L = \{1, 2, 3, 2, 3, 4, \ldots\}$, the depth of each node in E.
- $R = \{1, 2, 8, \ldots\}$, the first appearance of the node in E (in order of node id).

$|E| = |L| = 2n-1$, $|R| = n$. Claim: $LCA(u, v)$ is the shallowest node visited in the Euler tour from u to v (or from v to u). So $LCA(u, v) = E[RMQ_L(R[u], R[v])]$. We will see more next time....

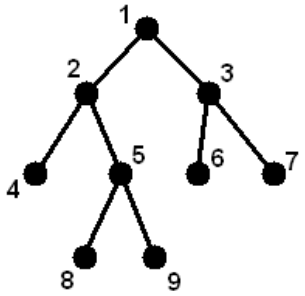

Figure 1: An example tree.

References

[1] M. Bender, M. Farach-Colton. The Level Ancestor Problem simplified. Lecture Notes in Computer Science. 321: 5-12. 2004.

[2] P. Dietz. Finding level-ancestors in dynamic trees. Algorithms and Data Structures, 2nd Workshop WADS '91 Proceedings. Lecture Notes in Computer Science 519: 32-40. 1991.

6.851 Advanced Data Structures
Spring 2010