6.854 Advanced Algorithms

Lecture 4: September 14, 2005 Lecturer: David Karger

Suffix Trees and Fibonacci Heaps

4.1 Suffix Trees

Recall that our goal is to find a pattern of length m in a text of length n. Also recall that the trie will contain a size $|\Sigma|$ array at each node to give $O(m)$ lookups.

4.1.1 Size of the Trie

Previously, we have seen a construction algorithm that was linear in the size of the trie. We would like to show that the size of the trie is linear in the size of the text, so that the construction algorithm takes $O(n)$ time. We can achieve this size goal by using a compressed suffix tree. In a compressed tree, each node has strictly more than one child. Below, we see the conversion from a uncompressed suffix tree to a compressed suffix tree.

```
o
 \ b
  o                        o
   \ b            =>        \  bbb
    o                        o
     \ b
      o
```

How will this change the number of nodes in the trie? Since there are no nodes with only one child, this is a full binary tree (i.e. every internal node has degree ≥ 2).

Lemma 1 *In any full tree, the number of nodes is not more than twice the number of leaves.*

When we use the trailing \$, the number of leaves in the trie is the number of suffixes. So does this mean that there are n leaves and the tree is of size $O(n)$? Yes; however, the number of nodes isn't necessarily the full size of the tree – we must store the substrings as well. For a string with distinct characters, storing strings on the edges could lead to a $O(n^2)$ size algorithm. Instead, we just store the starting and ending index in the original text on each edge, meaning that the storage at each node is $O(1)$, so the total size for the tree is in fact $O(n)$.

With the compressed tree, we can still perform lookups in $O(m)$ time using the *slowfind* algorithm which compares one character at a time from the pattern to the text in the trie. When *slowfind* encounters a compressed node, it checks all of the characters in the node, just as if it were traversing the uncompressed series of nodes.

4.1.2 Building the Trie

A simple approach to building the compressed tree would be to build an uncompressed tree and then compress it. However, this approach would require quadratic time and space.

The construction algorithm for compressed tries will still insert $S_1 ... S_n$ in order. As we go down the trie to insert a new suffix, we may need to leave in the middle of an edge. For example, consider the trie that contains just `bbb`. To insert `ba`, we must split the edge :

```
   o                        o
    \                        \ b
     \ bbb                    o
      \                    a / \ bb
       o                    o   o
```

Splitting an edge is easy; we will create one new node from the split and then one new node (a leaf) from the insertion.

One problem with compressed trie is where to put suffix links from the compressed edges. Another problem is that we previously described the time to operate on the tree in terms of n (the number of characters in the text); however, n may now be greater than the number of nodes.

fastfind is an algorithm for descending the trie if you know that the pattern is in the trie. *fastfind* only checks the first character of a compressed edge; all the other characters must match if the first does because the pattern is in the trie and because there is no branch in the edge (ie, if the pattern is there and there is no branch, it must match the entire edge or stop in the middle of the edge). If the pattern is shorter than the edge, then *fastfind* will stop in the middle of the edge. Consequently, the number of operations in a *fastfind* is linear in the number of checked nodes in the trie rather than the length of the pattern.

Suppose we have just inserted $S_i = aw$ and are at the newly created leaf which has parent node p_i. We maintain the following invariant:

Invariant: Every internal node except for the current parent has a suffix link (ignore for now the issue of where the suffix links point).

```
         /
        /     SL
  g_i  o ==============o
      / \              |  w1
 w1  /   ...           |
    o p_i              o alpha
    |
    |
 w2 |
    o s_i
```

Now we describe the consruction in detail. Let g_i be a parent of p_i. To insert S_{i+1}: ascend to g_i, traverse the suffix link there, and do a *fastfind* of w_1, which takes you to node α (thus maintaining the invariant for next time). Make a suffix link from p_i to α. From there, do a *slowfind* on w_2 and do the insertions that you need. Since p_i was previously a leaf node, it has no suffix link yet. g_i was previously an internal node, so it has a suffix link. w_1 is the part of S_i that was already in the trie below g_i (i.e., it was p_i), which is why we can use *fastfind* on it. w_2 is the part of S_i that was not previously in the trie.

The running time analysis will be in two parts. The first part is the cost from the suffix of g_i to the suffix of p_i. The second is the cost from the suffix of p_i to the bottom of the search. The cost of going up is constant, since there are only two steps thanks to the compressed edges.

Looking at the second cost (the *slowfind* part), we see that it is the number of charactersr in the length difference between the suffix of p_i and p_{i+1}, which is $|p_{i+1}| - |p_i| + 1$. The sum of this term over all i is $|p_n| - |p_0| + n = O(n)$.

For the first cost, recall that *fastfind*'s runtime will be upperbounded by the runtime of *slowfind*. It takes at most $|g_{i+1}| - |g_i|$ time to reach g_{i+1}. If g_{i+1} is below the suffix of p_i, then there is no cost. If the suffix of p_i is below g_{i+1}, then the suffix of p_i is p_{i+1} and the *fastfind* only takes one step from g_{i+1} to p_{i+1}, so the cost is $O(1)$.

The progression of the insert is

- suffix of g_i
- g_{i+1}
- suffix of p_i
- p_{i+1}

The total time is linear in the size of the compressed tree, which is linear in the size of the input.

4.2 Heaps

Prim and Dijkstra's algorithms for shortest paths and minimum spanning trees were covered in 6.046. Both are greedy algorithms that start by setting node distances to infinity and then relaxing the distances while choosing the shortest. To perform these operations, we use a priority queue (generally implemented as a heap). A heap is a data structure that will support insert, decrease-key, and delete-min operations (and perhaps others).

With a standard binary heap, all operations run in $O(\log n)$ time, so both algorithms take $O(m \log n)$ time. We'd like to improve the performance of the heap to get a better running time for these algorithms. We could show that $O(\log n)$ is a lower bound on the time for delete-min, so the improvement will have to come from somewhere else. The Fibonacci Heap performs a decrease-key operation in $O(1)$ time such that Prim and Dijkstra's algorithms require only $O(m + n \log n)$ time,

Idea: During insertions, perform the minimal work possible. Rather than performing the whole insert, we'll just stick the node onto the end of some list, taking $O(1)$ time. This would require us to do $O(n)$ work to perform delete-min. However, we can put that linear amount of work to good use to make the next delete-min more efficient.

The Fibonacci heap uses "Heap Ordered Trees," meaning that the children of every node have a key greater than their parent and that the minimum element is at the root. For Fibonacci heaps, we will have only 1 child pointer, a doubly linked list of children, and parent pointers at every node.

The time to merge two HOTs is constant: compare the two root keys and attach the HOT with the larger root as a child of the smaller root.

To insert into a HOT, compare the new element x and the root. If x is smaller, it becomes the new root and the old root is its child. If x is larger, it is added to the list of children.

To decrease a key, you prune the node from the list of children and then perform a merge.

The expensive operation is delete-min. Finding the minimum node is easy; it is the root. However, when we remove the root, we might have a large number of children that need to be processed. Therefore, we wish to keep the number of children of any node in the tree relatively small. We will see how to do this next lecture