Lecture 11: Range-Minimum Queries

Lecturer: Travis

February 24th, 2015

Today we'll see how to use succinct representations of trees to build another useful succinct data structure, one that supports range-minimum queries (RMQs). After that, we'll briefly go over what you should know for the exam. On Thursday we'll talk about the Projects course.

The query RMQ(i, j) on an array $A[1..n]$ can be defined to return either the value of the minimum element in $A[i..j]$, the position of one of its occurrences, or both.

There's a folklore data structure for RMQs that takes $\mathcal{O}(n \log n)$ words of memory and answers queries in $\mathcal{O}(1)$ time:

For each position $p \leq n$ and power $q \leq \lfloor \lg n \rfloor$, we store the value of the minimum element in $A[p..p + 2^q - 1]$ and the position of one of its occurrences there; given a range $[i..j]$, we choose the largest power q of 2 at most $j - i + 1$ and return the smaller of the the minimum elements in the ranges $A[i..i + 2^q - 1]$ and $A[j - 2^q + 1..j]$, and the position of one of its occurrences in $A[i..j]$.

This data structure is a $(\lg n)$-factor larger than A itself, which is all we need if we don't care about query time. Several researchers proposed earlier linear-space data structures, but the best-known one is by Farach and Bender (2000), who reduced RMQs to lowest-common-ancestor (LCA) queries in Cartesian trees.

To build a Cartesian tree for $A[1..n]$, we choose an occurrence of the minimum element, say in position p, and make it the root. We then make its two subtrees the Cartesian trees for $A[1..p-1]$ and $A[p+1..n]$.

PAUSE WHILE TRAVIS DRAWS A CARTESIAN TREE ON THE BOARD.

If the ith node is an ancestor of the jth node in the Cartesian tree for A, then $A[i]$ is the minimum element in $A[i..j]$. (WHY?) If the jth node is an ancestor of the ith nodes, then $A[j]$ is the minimum element.

Otherwise, the ith and jth nodes are in the left and right subtrees of some node v — their *lowest common ancestor*. Notice that v's parent is either to the left of the ith node or to the right of the jth, so v is the highest node between the ith and the jth. Therefore, if v is the mth node from the left, then $A[m]$ is the minimum element in $A[i..j]$.

PAUSE WHILE TRAVIS WAVES HIS ARMS IN FRONT OF THE TREE ON THE BOARD.

So, how can we support fast LCA queries? Well, suppose we write the balanced-parentheses representation of the shape of the Cartesian tree for A and, for each parenthesis, write the depth of the node it corresponds to.

PAUSE WHILE TRAVIS DOES THIS FOR THE TREE ON THE BOARD.

Notice that all the nodes with the smallest depth between the ith and jth nodes are children of their LCA. Therefore, if we can find RMQ for the depths between the parentheses for the ith and jth nodes, then we can find their LCA, so we can find RMQ(i, j).

This seems pretty circular, but notice the depths for consecutive parentheses differ only by 1. Let's consider the special case of supporting RMQs on a sequence in which consecutive values differ by 1.

We can store such a sequence as a binary string. Suppose we break this string into blocks of length proportional to $\lg n$.

For each block, we store the number of 1s and 0s until the beginning of the block (or we can store $o(n)$ bits to be able to answer rank queries), the minimum value in that block, and the position of an occurrence of that minimum.

Consider the $\mathcal{O}(n/\log n)$-element sequence of the blocks' minima. Using the folklore solution, we can build a $\mathcal{O}(n)$-word RMQ data structure for this sequence. With this, given i and j, we can find an occurrence of the element strictly between the two blocks containing the ith elements and jth element.

Suppose we can somehow find in $\mathcal{O}(1)$ time the minimum element in the prefix or suffix of a block. Then in $\mathcal{O}(1)$ time we can find the minimum element between the ith and the jth elements. That is, we can find in $\mathcal{O}(1)$ time a parenthesis with minimum depth between the closing parenthesis for the ith node and the opening parenthesis for the jth node.

How do we find the minimum element in a suffix or prefix of a block? Well, the blocks have length $\mathcal{O}(\log n)$, so we can split them each into $\mathcal{O}(1)$ mini-blocks of length $\lg(n)/2$, and use a *universal table* of size $2^{\lg(n)/2} \log^{\mathcal{O}(1)} n \subseteq \mathcal{O}\left(n^{1/2+\epsilon}\right)$.

So, given i and j, we find a parenthesis with minimum depth between the parentheses for the ith and jth nodes in the Cartesian tree, find the parent v of the node corresponding to that parenthesis, look up v's rank r using succinct-tree tricks (that Simon may have showed you), and return $A[r]$ and r.

Theorem

We can store an array $A[1..n]$ *in* $\mathcal{O}(n)$ *words such that given* i *and* j*, in* $\mathcal{O}(1)$ *time we can return the value of the minimum element in* $A[i..j]$ *and the position of one of its occurrences there.*

That's pretty cool but notice that, if we only want the position of an occurrence of the minimum element and not its value, then we need only the shape of the Cartesian tree, *which we can encode in* $2n$ *bits*.

Suppose we don't store A and that, instead of using the folklore solution, we use the solution we just developed to support RMQs on the blocks' minima. That takes $\mathcal{O}(n/\log n)$ words, or $\mathcal{O}(n)$ bits.

For any positive constant ϵ there exists another constant c such that, if we make the blocks each $c \lg n$ bits long, then everything apart from the balanced parentheses representation takes ϵn bits.

(This slows the queries down by a factor proportional to $1/\epsilon$, which disappears in the asymptotic notation.)

Theorem

We can store $(2+\epsilon)n + \mathcal{O}(1)$ *bits such that, given* i *and* j*, in* $\mathcal{O}(1)$ *time we can return the position of a minimum element in* $A[i..j]$*.*

(Actually, it's possible to improve the space bound to $2n + o(n)$ bits, which is optimal to within lower order terms, but we're not going to do that in this course.)