Lecture 8

The level ancestor problem LAA

Problem: Input: Footed tree \( T \) with \( n \) vertices
Query: For node \( v \), return the depth \( d \) of the ancestor \( LAA(v, d) \)

Equivalent: \( LAA(v, d) \) < depth 
\[ d + l = \text{depth}(v) \]

Notation: An algorithm that needs \( f(n) \) preprocessing & \( g(n) \) query time is denoted as \( (f(n), g(n)) \)

Goal: Find a \( (O(n), 0(1)) \) algorithm

5 we present a series of improvements to achieve the \( (O(n), O(1)) \) bound
Algorithm A: $O(n^2), O(1)$
Table look-up

Array that can be filled column by column using a simple dynamic program.

Algorithm B: $O(n \log n), O(\log n)$
Jump-pointers

- We store pointers at every node that jump up to the $2^1, 2^2, 2^3, 2^4, \ldots$ depth($v$) ancestor of $v$

\[ \text{using the pointers we get at least halfway to } LA(v, d) \]

\[ \leq O(\log n) \text{ query time} \]

\[ \Rightarrow \text{we need } O(n \log n) \text{ storage, computing is easy, using dynamic programming} \]

\[ \Rightarrow (\text{we fill a } \log n \text{ array}) \]
Alg $C <O(n), O(n^2)>$

Long Path decomposition

- We break $T$ into disjoint paths as follows
  - remove the largest path & recurse on the connected components
  - store every path as an array and "connect" it to its "parent"
  - connect every node in $T$ to its entry in the path decomposition
  - max depth of the path decomposition is $\Theta(n^2)$

$O(n^2)$ Example
Storage + prep. = $O(n)$
Alg D \{O(n), O(log n)\}
Ladder Decomposition

- Use the longest path decomposition, but extend every path by its length a factor of 2 (or less if the root "comes")

\[
\begin{tikzpicture}
  \node at (0,0) (v) {v};
  \node at (2,2) (p) {P};
  \node at (3,4) (l) {ladder};
  \draw (v) -- (p);
  \draw (p) -- (l);
  \end{tikzpicture}
\]

Fact: \( v \) lies on a path in the decomposition of length at least \( \text{height}(v) \)!

\[\rightarrow\text{using the ladder we double the height or reach the root}\]

L \( \Theta(\log n) \) query time

Alg E \{O(n \log n), O(1)\}
Ladders + jump pointers

- Ladder: exponentially increasing hops up the tree
- Jump pointers: exponentially decreasing hops up the tree
* We do the preprocessing for Alg B & Alg D
  \[ \rightarrow O (h \log n) \]

* A query does the following
  + use a jump pointer to reach \( v' \)
  + climb a ladder from \( v' \)

  \[ \begin{align*}
  & \quad \text{we jumped up at least } \frac{h}{2} \text{ nodes} \\
  & \quad \text{\( v' \) lies on a path } \geq \frac{h}{2} \\
  & \quad \Rightarrow \text{ladder goes up } \geq \frac{h}{2} \text{ out of } v'
  \end{align*} \]

  \[ \Rightarrow \text{we reach } LA(v,e) \text{ at 1 jump + 1 ladder climbing} \]

\[ \Rightarrow \text{small improvement: store jump pointers only at leaves} \]

\[ \begin{align*}
\text{Jump pointers can be computed using the ladder decomposition, not} \\
\text{dynamic program, see Alg (x)}
\end{align*} \]

\[ \Rightarrow \text{Query: answer } LA(v,e) \text{ only at leaves} \]

\[ \text{Preprocessing: } O (n + L \log n) \]

\[ \text{# of leaves in } T \]
Alg F \{ O(n), O(1) \}

Micro-Macro Tree Algorithm

Idea: precompute small subtrees of $T$

- This gives only $O(n/\log n)$ leaves of $T$

\[ \begin{array}{c}
\text{Macrofree} \\
\text{remaining part}
\end{array} \]

3 Microtrees
- Maximal Trees of $\frac{3}{4} \log n$ nodes

- There are at most $C \frac{\log n}{4 \log n}$ many distinct microtrees

\[ C \frac{\log n}{4 \log n} \leq \frac{1}{4} \log n \leq \frac{1}{2} n \]

( More direct technique next slide)

- The macrotree has at most

- We solve for any possible microtree the $L4$ problem and store it in a look-up table. This takes $O(\sqrt{n} \cdot \log^3 n)$ time
* Leaf pointers can be computed in $O(\log n)$ time each (using ladders implies total $O(n)$)

* Query $L(A(v, x))$: if we Macrotree
  - jump to leaf
  - jump-up pointer ($\alpha x$)
  - ladder climbing ($\alpha x$)
  - if we Microtree
    - if $e$ belongs to Microtree use look-up table
    - else go to the leaf in Macrotree and search with $\alpha$

---

How to enumerate all bounds for microtrees

- Every tree is specified by its up/down edge sequence of a DFS (DFS1 = $2 \cdot 1E1$)

- Query 0/1 pattern = $2^{\frac{4}{4} \log \frac{1}{4} = \log 2 \cdot \log \frac{1}{4} = \sqrt{2}}$

- We define the string to be associated tree as the longest valid prefix $P$

  - #0 = #1
  - All prefixes of $P$: #0 > #1
The LCA - Problem

**LCA**

Input: Tree (rooted) with \( n \) nodes
Q: \( LCA(x, y) \) returns the **lowest** Common Ancestor of \( x, y \)

**RMQ**

Input: Array \( A \)
Q: \( \text{RMQ}_A[i, j] \), index of the minimum element in \( A[i, j] \)

Lemma: \( A < f(n), g(n) > \) Algorithm for RMQ
induces a \( f(2n-1) + O(n), g(2n-1) + O(n) \) algorithm for LCA

---

E: Euler-tour: Sequence of nodes traversed in DFS

\[ E = 1, 2, 1, 3, 4, 6, 4, 7, 8, 7, 9, 7, 4, 3, 5, 3, 1 \]

L = Level array (for \( E \))

\[ L = 2, 1, 2, 1, 3, 4, 3, 4, 5, 4, 3, 2, 3, 2, 1 \]

\( |E| = |L| = 2n - 1 \)
First occurrence of a node is stored in \( R \) (representatives)

\[ R = 2, 1, 2, 4, 5, 15, 6, 8, 11, 3 \]

The LCA \((u,v)\) is the shallowest node visited at the Euler Tour between \( u \rightarrow v \), that is \( E [RMQ_L(CR[u],RE[v])] \)

**Algorithm A**

\(<O(\log u), O(1)>\)

Sparse table algo. for RMQA

We use a sparse look up table \( M \)

\( M \) has size \( O(\log u) \) and can be computed in \( O(\log u) \) time using dyn. prog.

\[ \text{RMQA}(i,j) : \]

largest \( 2^k \) block between \( i,j \)
We find 2 Blocks that cover $A[i;j]$, comparing their minima answers the RMQ query

<table>
<thead>
<tr>
<th>Alg B</th>
<th>${O(u), O(1)}$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Algorithm for RMQ+1</td>
</tr>
</tbody>
</table>

- RMQ+1 is RMQ but the entries $A[i;j], A[i+1;j]$ differ by $\pm 1$
- For solving LCA we have to solve RMQ+1

Idea: splitting A in blocks of size $\frac{\log u}{2}$

\[ A' \]

$A'$ stores the RMQ+1 queries minimum for each block

\[ A'|1 = \lceil \frac{u}{\log u} \rceil \]

\[ B' \]

$B'$ stores the index where the minimum is rebuilder for each block

**Queries:**

RNA here can be computed with Alg B for $A'$
We can compute $\text{RMQ}_A(i, j)$ by combining
1. $\text{RMQ}_A(i, e)$ $e =$ end of block of $i$
2. $\text{RMQ}_A(e', j)$ $e' =$ start of block of $j$
3. $\text{RMQ}_A(e, e')$ Alg $A^*$ $(\log \frac{n}{u}, O(1))$

To answer 1. and 2. we precompute

Fact: If two blocks differ by the same number at each position they have the same RMQ answer

→ we can normalize all blocks
→ there are at most $2^{\log_2 \frac{1}{u}} = \sqrt{u}$ normalized blocks
→ store $\sqrt{u}$ tables of size $\log u \times \log u$
→ $O(\sqrt{u} \log^2 u)$ storage / time
Back to RMQ (general case)

- $RMQ_A(i, j)$ equals the LCA of the Cartesian tree of $A$!

- The Cartesian tree can be build in linear time