Programming Club Notes

Least Common Ancestor Queries

Zachary Friggstad

October 9, 2018
Basic problem: given a tree $T = (V; E)$ rooted at a vertex $r$ and two vertices $u, v$, find the least common ancestor of $u$ and $v$.

This is the deepest vertex that lies somewhere above (i.e. is an ancestor) of both $u$ and $v$.

Denoted $\text{lca}(u, v)$
Observe \( \text{lca}(v, v) = v \).

Note for \( u, v \) that \( \text{lca}(u, v) = u \) if and only if \( u \) lies on the \( r - v \) path in the tree \( T \).

Also, \( \text{lca}(r, v) = r \) for any \( v \) (recall \( r == \) root).
The only thing we assume is that we know the parent $p(v)$ of any vertex $v$ in the tree. Say $p(r) = \text{nil}$.
A simple algorithm: construct the $v - r$ path by crawling up the tree (i.e. following $p()$). Follow the $u - r$ path by crawling up the tree. The start of the common suffix of these paths is $\text{lca}(u, v)$.

A single query takes $O(n)$ time.
A Faster Approach

If we anticipate many queries, we can do some preprocessing to speed them up.

**Preprocessing**: compute the following

- For each $v \in V$
  - $\text{depth}(v)$: the distance to the root, so $\text{depth}(r) = 0$
- For each $v \in V$ and each $0 \leq i \leq \log_2 n$
  - $p(v, i)$: the vertex that is $2^i$ steps above $v$.
  - If $\text{depth}(v) < 2^i$, then $p(v, i) = \text{nil}$

We can compute all of these values in $O(n \log n)$ time (see this later).
Given these values, here is how to compute $\text{lca}(u, v)$.

Rough idea: walk the deepest node upward by taking $2^i$-length steps for decreasing $i$ but only if this does not overshoot $\text{depth}(v)$.

$\text{depth}(v) = 3$
$\text{depth}(u) = 29$
$\text{depth}(u) - \text{depth}(v) = 26 = 2^4 + 2^3 + 2^1$

After, $u, v$ have the same depth. If $u = v$ now, we are done!
Otherwise, walk $u, v$ upward simultaneously taking $2^i$-length steps for decreasing $i$ but only if this does not merge the nodes (don’t want to merge too far up the tree).

Once done, $\text{lca}(u, v)$ is the common parent of $u$ and $v$. 
Step 1: Walk the deeper node up the depth of the other.

- Assume $\text{depth}(u) \geq \text{depth}(v)$ (swap $u$, $v$ otherwise).
- $\textbf{for } i = \log_2 n \textbf{ down to } 0$
  - $\textbf{if } \text{depth}(u) - 2^i \geq \text{depth}(v) \textbf{ then } u \leftarrow p(u, i)$.

Now $u, v$ are at the same depth and they have the same lca as the old $u, v$. 
Step 2: Walk \( u, v \) up simultaneously until they have a common parent.

- If \( u = v \) already, then return \( v \)
- For \( i = \log_2 n \) down to 0
  - If \( p(u, i) \neq p(v, i) \) then \( u, v \leftarrow p(u, i), p(v, i) \)
- Return \( p(u, 0) \) (i.e. the parent of \( u \))

Note the if in the loop also catches the case when \( p(u, i) = p(v, i) = \text{nil} \) (i.e. \( \text{depth}(u) < 2^i \)).

Entire query is processed in \( O(\log n) \) time!
Using just $p(u)$ information, we can compute $\text{depth}(u)$ and $p(u, i)$ entries with dynamic programming!

$$\text{depth}(u) = \begin{cases} 
\text{depth}(p(u)) + 1 & \text{if } u \neq r \\
0 & \text{if } u = r 
\end{cases}$$

For $p(u, i)$, apart from obvious base cases just take two $2^{i-1}$ hops up.

$$p(u, i) = \begin{cases} 
p(u) & \text{if } i = 0 \\
nil & \text{if } p(u, i - 1) = \text{nil} \\
p(p(u, i - 1), i - 1) & \text{otherwise}
\end{cases}$$

If you are truly laze, you don’t even need to fill these values by preprocessing, just fill the table on the fly! Total time spent filling all DP table entries is $O(n \log n)$. 
Practical Tips

Usually vertices in a graph are numbered 0 through \( n - 1 \). Use \(-2\) to represent \textit{nil} and \(-1\) to represent \textit{not yet computed} in the DP table.

```c
#define MAXN 100000
#define LOGN 17 // the smallest k such that \( 2^k > \text{MAXN} \)

int depth[MAXN];
int p[MAXN][LOGN];

// initialize DP tables to -1: entries not yet computed
memset(depth, -1, sizeof(depth));
memset(p, -1, sizeof(p));

// read in the parent pointers into p[v][0] entries,
// remembering to initialize p[root][0] = -2, i.e. nill

// start all "i from log n down to ..." loops at LOGN-1
```