Binary Lifting, Euler Tours, and their applications in Lowest Common Ancestor Problems

Max Godfrey 09/04/2022

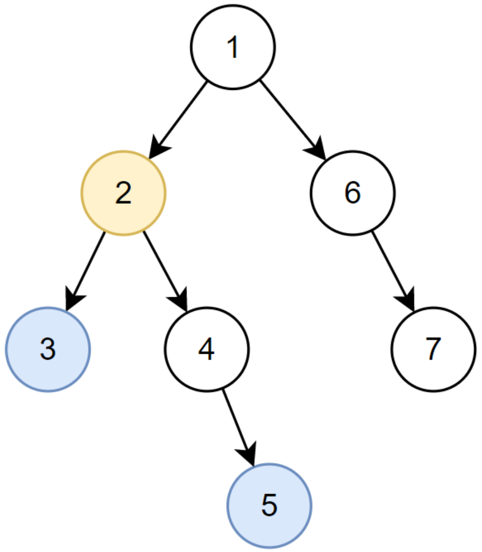
Table of Contents

- LCA with Binary Lifting
- LCA with Euler Tour
- Algorithmic applications of LCA
- Functional/Successor/Vortex Graphs

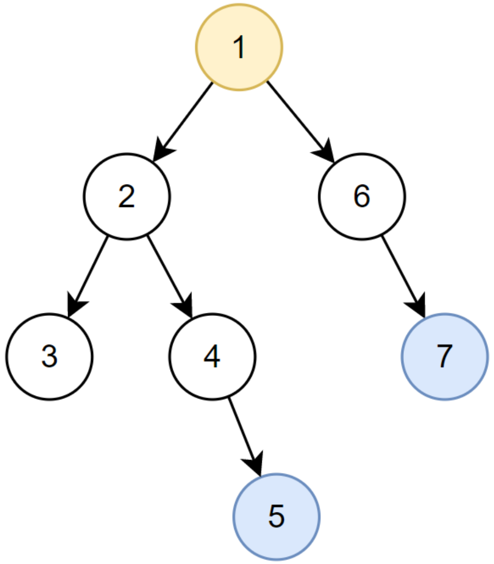
Lowest Common Ancestor

- Definitions:
  - In a rooted tree, we say that node A is an *ancestor* of node B if A is encountered on the path from the root down to B
  - The *lowest common ancestor* of nodes A and B is the node with greatest distance from the root which is an *ancestor* of both A and B
  - Commonly denoted $LCA(u, v)$
  - Note: It is possible that $LCA(a, b) = a$
  - Trivially, $LCA(a, b) = LCA(b, a)$ and $LCA(a, a) = a$

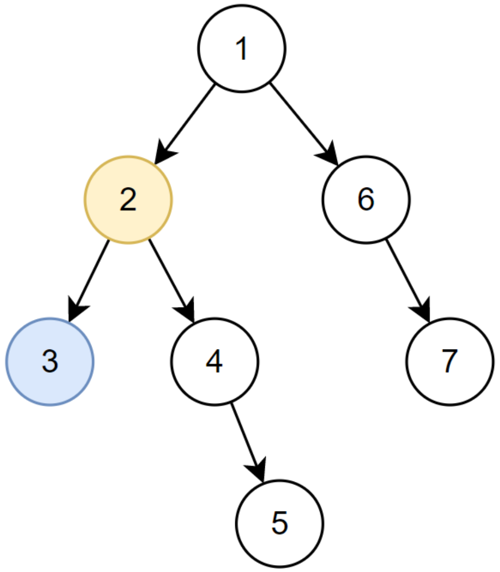
LCA Examples



$LCA(3, 5) = 2$



$LCA(5, 7) = 1$



$LCA(3, 2) = 2$

Naïve LCA Algorithm

Let $H_i = Distance(root, i)$

To calculate $LCA(u, v)$:

- Walk either u or v up the tree until $H_u = H_v$
  - ie. $u = par[u]$ or $v = par[v]$
- Walk u and v up the tree simultaneously until they are the same node

$LCA(u, v)$ has a complexity of $O(N)$

Binary Lifting

- The problem with the naïve algorithm is the costly linearity of walking nodes up the tree

- Jump Tables:
  - Consider a 2D array $jump$, where $jump[v][p] = 2^p$-th parent of node v

  - $jump[i][0] = par[i],$ for all $1 \leq i \leq N$
  - $jump[i][p] = jump[\ jump[i][p-1]\ ][p-1],$ for all $2 \leq p \leq \lfloor \log_2 N \rfloor$

    ie. The 2^p-th parent of node i is the 2^{p-1}-th parent of the node 2^{p-1} steps above i

- Table is constructed in $O(N \log_2 N)$

Getting nodes to the same height

- Suppose that node u is K nodes closer to the root than node v
- According to its binary representation, $K = 2^x + 2^y + \ ...$
- If we take all the jumps from v corresponding to K's set bits, we get to u

- $O(\log_2 N)$ - much better!

Finding the LCA

- As we walk up the tree from two distinct nodes of the same height:
  - For some time, the walks point to different nodes
  - Eventually, they will point to the same node, and continue to do so until they reach the root

- We must find the lowest height up the tree where the walks point to different nodes
  - In decreasing powers of two, if $jump[u][pow] \neq jump[v][pow]$, then:
    - $u = jump[u][pow]$ and $v = jump[v][pow]$
  - If the jump table points to the same node for each walk, we are not guaranteed that the node pointed to is the LCA: it could be any node from the LCA up to the root, so we do not take the jump in this case
  - Once we find the node we are looking for, the LCA is its immediate parent

Binary Lifting Code:

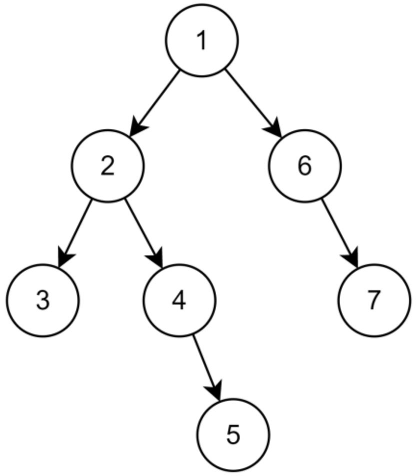
```
void dfs(int at, int h = 1) {
  H[at] = h;
  for (auto child : down[at]) {
    dfs(child, h + 1);
  }
}
```

```
for (int i = 1; i <= N; ++i) {
  jump[i][0] = par[i];  // Set par[root] = root.
}
for (int p = 1; p < 18; ++p) {
  for (int i = 1; i <= N; ++i) {
    jump[i][p] = jump[jump[i][p - 1]][p - 1];
  }
}
```

```
int lca(int a, int b) {
  if (H[b] > H[a]) {
    swap(a, b);
  }
  for (int i = 17; i >= 0; --i) {
    if (H[jump[a][i]] >= H[b]) {
      a = jump[a][i];
    }
  }
  if (a == b) {
    return a;
  }
  for (int i = 17; i >= 0; --i) {
    if (jump[a][i] != jump[b][i]) {
      a = jump[a][i];
      b = jump[b][i];
    }
  }
  return jump[a][0];
}
```

Euler Tour Technique

- In GT, an Eulerian Tour is a trail which visits every edge exactly one (no nodes may be repeated)
- Regarding trees:
  - Traversal where we add nodes every time we “look” at them
  - Imagine tracing the shape of a rooted tree with a pencil



The vertices visited in the Euler tour of this tree are:
1, 2, 3, 2, 4, 5, 4, 2, 1, 6, 7, 6, 1

Euler Tour Code

- $start[i]$ stores the first occurrence of node i in the Euler Tour.

```
void dfs(int at, int par) {
  start[at] = euler_tour.size();
  euler_tour.push_back(at);
  for (auto child : adj[at]) {
    if (child != par) {
      dfs(child, at);
      euler_tour.push_back(at);
    }
  }
}
```

Some Observations

- The entire subtree of node u is contained between the first and last occurrences of u in the Euler Tour array
  - If node a is an ancestor of node u, then u will occur at least once between the first and last occurrences of a in the Euler Tour array; there will be no occurrences of u ouside these bounds
- $LCA(u, v)$ will be contained somewhere in the Euler Tour array between $start[u]$ and $start[v]$ inclusive
  - Additionally, there will be no higher nodes than the LCA contained in this subarray. This is because the Euler Tour never goes higher than necessary when moving from subtree to subtree

- Given these properties, finding the LCA now reduces to RMQ

RMQ

- We can build a Range Tree or Sparse Table over the Euler Tour array
- Define a merge function like this (only one value needs storing per ST node):

```
inline int merge(int i, int j) {
  return (H[i] < H[j] ? i : j);
}
```

- DS Recap:

	Build	Query	Update
Sparse Table	$O(N \log_2 N)$	$O(1)$	$O(N \log_2 N)$
Range Tree	$O(N)$	$O(\log_2 N)$	$O(\log_2 N)$

- Thankfully, there are no updates, so LCA queries can be done in constant time!

Applications in Informatics

- LCA is Often used in Tree DP:
  - The path between nodes u and v goes up from u to $LCA(u, v)$ and then back down to v
  - We can answer some types of path queries by computing DP values down to every node in the tree, and then modifying our answer according to $DP[LCA(u, v)]$

- Binary Lifting:
  - A great way to speed up "Finding the k-th element above something"
  - Requires some kind of hierarchy (either intrinsic, or contrived by you)

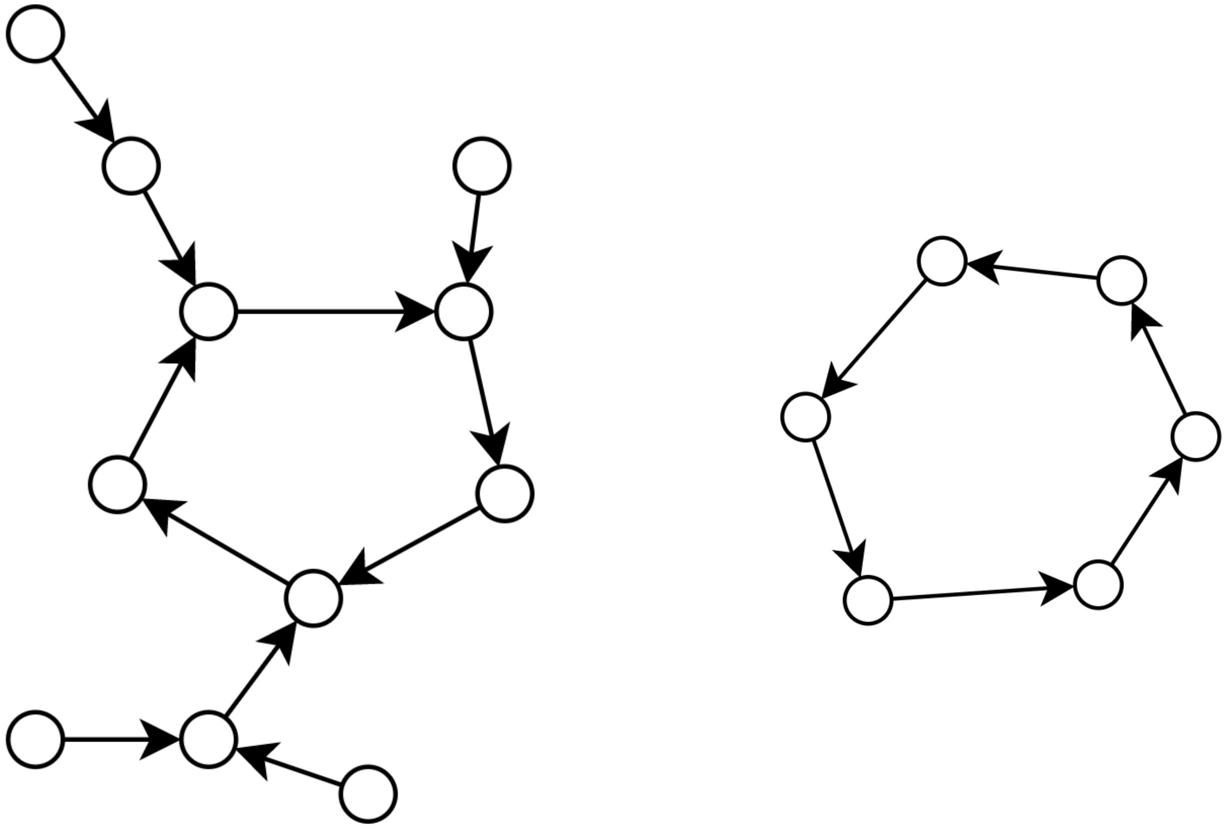
- Euler Tours:
  - Fantastic for flattening trees, and they are very versatile: https://codeforces.com/blog/entry/18369

Functional Graphs

- A Functional Graph is a directed graph where each node has one outgoing edge
  - Trivially, sinks cannot exist, however there may be sources
  - Imagine taking a tree, and then adding an edge between any two nodes
  - A Functional Graph has exactly one cycle

- Defined by N elements where $succ[i]$ represents the successor of node i

Functional Graph Examples



Some Observations

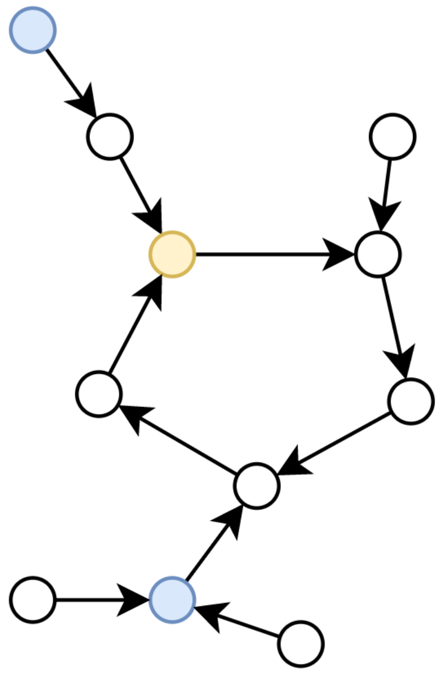
- It is possible for a successor array to describe many subgraphs (ie. A ‘Functional Forest’): each of these subgraphs *must* contain a cycle
- Once we are on a cycle, we cannot leave it:
  - There is only one outgoing edge from each node, so we are stuck going round and round forever
- “A cycle possibly with some ‘trees’ hanging off it”

- We know from elementary GT that detecting the presence of a cycle is trivial and can be done in $O(N)$
- Hence, if we find a node which is part of a Functional Graph’s cycle, we can combine the above observations to find every node on its cycle in linear time

Problem: Joining Couples

- You are given a Functional Graph of N nodes and must answer Q queries:
  - Given two nodes a and b, find the shortest distance required for a and b to travel such that they can reach a common node, or report that it is impossible for them to meet

- $1 \leq N, Q \leq 10^5$
- Time Limit: 1 second



Sample:
For the two blue nodes pictured, the shortest distance required for them to reach a common node is 5.
The common node has been highlighted yellow.

Joining Couples

- “A Cycle possibly with some ‘trees’ hanging off it”
- There are three cases which we need to deal with here:
  1. Nodes a and b are on the same ‘tree’
  2. It is possible for a and b to meet, but they aren’t on the same ‘tree’
  3. It isn’t possible for a and b to meet

Joining Couples

- “A Cycle possibly with some ‘trees’ hanging off it”
- Case 1: a and b are on the same ‘tree’
  - This can be solved using LCA, but first we must know all of the cycles
    - There are several ways to do this, an easy one is to run a DFS on each ‘Functional Subgraph’: when we reach a node we have already seen, we can construct the cycle from it
    - To work out the ‘roots’, reverse all the edges and traverse the cycle’s nodes. If a node on the cycle has outgoing (reversed) edges which don’t lead to another node on the cycle, then it is a ‘root’
  - DFS from all the ‘roots’ and proceed as normal with LCA

Joining Couples

- "A Cycle possibly with some 'trees' hanging off it"
- Case 2: a and b can meet, but they aren't on the same 'tree'
  - We are going to have to walk through the cycle
  - Firstly, let's find out the distance to the cycle from these nodes:
    - If either of them is already on the cycle, we should know
    - Otherwise, the distance to the cycle is just H_i
  - If we know the indices of their respective trees' roots, determining the additional distance we need to cover once we are on the cycle is easy
    - Observe that it is suboptimal for both nodes to move around the cycle
    - We should either walk from a's 'root' to b's 'root' or vice versa

Joining Couples

- “A Cycle possibly with some ‘trees’ hanging off it”
- Case 3: It is impossible for a and b to meet
  - This is only true if the nodes aren’t on the same ‘Functional Subgraph’
  - If nodes are on the same ‘Functional Subgraph’, they can always meet by making their way to the cycle in 0 or more steps and walking round to find each other
  - There are several ways to determine whether or not a and b can meet – I believe that an easy one is to ID all the ‘Functional Subgraphs’. If the IDs don’t match, they can never meet

Your Problemset

Go ahead and solve the following:

1. [Optional] CSES Company Queries II: https://cses.fi/problemset/task/1688/
   - If you are comfortable implementing LCA, start with #2. If not, start here.
2. Jim Thomas
3. Max Flow
4. Joining Couples
5. Yourcraft
6. [Hard] Crayfish Scrivener
   - When you solve this one, come speak to me about one of *C++'s biggest secrets*… or just look at my Orac 2 submission to the problem
7. [Very Hard] Designated Cities

Thank you for your attention!

Further Reading

- Reducing LCA to RMQ: http://www-di.inf.puc-rio.br/~laber/lca-rmq.pdf
- Farach-Colton-Bender: https://cp-algorithms.com/graph/lca_farachcoltonbender.html
  - This algorithm uses the Euler Tour, but makes some observations pertaining to the height array (RMQ±1 as opposed to RMQ)
  - Overall, it yields $O(N)$ preprocessing and $O(1)$ queries, but the implementation is messy
- Various applications of Euler Tours in trees explained: https://codeforces.com/blog/entry/18369