CPSC 490 – Problem Solving in Computer Science

Lecture 8: Tree DP, LCA Query, Graph DP

Jason Chiu and Raunak Kumar

2017/01/23

University of British Columbia

A new kind of DP

You have seen tree DP before!

Recall that Dynamic Programming = break down problem recursively into similar sub-problems and then combining their answers

You have seen tree DP before!

Recall that Dynamic Programming = break down problem recursively into similar sub-problems and then combining their answers

When we need to do something on a tree...

- What are the sub-problems?
- What are the base cases?

You have seen tree DP before!

Recall that Dynamic Programming = break down problem recursively into similar sub-problems and then combining their answers

When we need to do something on a tree...

- What are the sub-problems? Subtrees!
- What are the base cases?

You have seen tree DP before!

Recall that Dynamic Programming = break down problem recursively into similar sub-problems and then combining their answers

When we need to do something on a tree...

- What are the sub-problems? Subtrees!
- What are the base cases? Leaf nodes!

You have seen tree DP before!

Recall that Dynamic Programming = break down problem recursively into similar sub-problems and then combining their answers

When we need to do something on a tree...

- What are the sub-problems? Subtrees!
- What are the base cases? Leaf nodes!

Tree DP example: compute the size of the tree

- Leaf (i.e. base case): $f(u) = 1$
- Non-leaf (i.e. recurrence): $f(u) = 1 + \sum_{v=\text{child of } u} f(v)$
- Answer: $f(\text{root})$
- Time complexity: $O(N)$

Order Statistics in a BST

Getting the size of each subtree is easy but surprisingly useful!

Order Statistics in a BST

Getting the size of each subtree is easy but surprisingly useful!

Problem

- Modify your favorite balanced binary search tree (AVL, Splay, Red-Black Tree, etc) to support querying the k-th largest element

Order Statistics in a BST

Getting the size of each subtree is easy but surprisingly useful!

Problem

- Modify your favorite balanced binary search tree (AVL, Splay, Red-Black Tree, etc) to support querying the k-th largest element

Solution

- In every node, keep track of the size of subtree
- Update operation: update child if necessary, then recompute size of current node
- Find k-th largest element: if size(right subtree) $\geq k$, find k-th largest element in right subtree, otherwise find ($k-$size(right subtree) - 1)-th largest element in left subtree
- All operations (insert, delete, find, get k-th largest) are $O(\log n)$

Problem 1 – Thinking Recursively

Given a rooted tree with $1 \leq N \leq 100,000$ nodes, each node containing an integer value, compute the following statistics for every single subtree and for the entire tree:

- Size
- Height
- Diameter (length of longest simple path)
- Average of values
- Median of values
- k-th largest value

Problem 1 – Solution (Part 1)

Size of each subtree

- $f(\text{leaf}) = 1, f(\text{node}) = 1 + \sum f(\text{child})$

Problem 1 – Solution (Part 1)

Size of each subtree

- $f(\text{leaf}) = 1, f(\text{node}) = 1 + \sum f(\text{child})$

Height of each subtree

- $f(\text{leaf}) = 1, f(\text{node}) = 1 + \max f(\text{child})$

Problem 1 – Solution (Part 1)

Size of each subtree

- $f(\text{leaf}) = 1, f(\text{node}) = 1 + \sum f(\text{child})$

Height of each subtree

- $f(\text{leaf}) = 1, f(\text{node}) = 1 + \max f(\text{child})$

Diameter of the tree

- Idea: keep two numbers per node
  - $f(u) =$ length of longest path that starts and ends in subtree of u
  - $g(u) =$ length of longest path that starts in subtree and ends at u
- $f(\text{leaf}) = g(\text{leaf}) = 0$
- $f(\text{node}) = \max\{f(child), 2 + \text{two largest values of } g(\text{child})\}$

Time complexity: $O(N)$

Problem 1 – Solution (Part 2)

Average

- $f(\text{node}) = \text{val}(\text{node}) + \sum f(\text{child})$
- $\text{Avg}(\text{node}) = f(\text{node})/\text{size}(\text{node})$

Time complexity: $O(N)$

Problem 1 – Solution (Part 2)

Average

- $f(\text{node}) = \text{val}(\text{node}) + \sum f(\text{child})$
- $\text{Avg}(\text{node}) = f(\text{node})/\text{size}(\text{node})$

Time complexity: $O(N)$

Median, k-th largest value

- Idea: get sorted list of values from each child, then merge
- How do we merge?
- What data structure do we use?

The Merging Sets Problem

Naive Approach

- Fill up an array, then call sort()
- Time complexity: up to $O(N \log N)$ per sort, so possibly worst case $O(N^2 \log N)$ in a deep tree
- The problem is that we keep moving items we've already merged

The Merging Sets Problem

Naive Approach

- Fill up an array, then call sort()
- Time complexity: up to $O(N \log N)$ per sort, so possibly worst case $O(N^2 \log N)$ in a deep tree
- The problem is that we keep moving items we've already merged

Merging sets with BSTs

- Let's store each set using... a set!
- While there are > 1 sets: pick any arbitrary pair, add all elements from the smaller set to the larger set

The Merging Sets Problem

Naive Approach

- Fill up an array, then call sort()
- Time complexity: up to $O(N \log N)$ per sort, so possibly worst case $O(N^2 \log N)$ in a deep tree
- The problem is that we keep moving items we've already merged

Merging sets with BSTs

- Let's store each set using... a set!
- While there are > 1 sets: pick any arbitrary pair, add all elements from the smaller set to the larger set

Time complexity

- Each time an element moves, it gets into a set twice as big
- $\rightarrow$ Each element is moved at most $O(\log N)$ times
- $\rightarrow O(N \log^2 N)$ in total

Problem 2 – Adding States

You are the head of the Maximum Wolf Trade Syndicate, a large organization of 10000 members. You are planning to hold a retreat involving some of the members.

The organization can be represented (in modern terminology) as a tree of managers. You want to select a subset of people that are connected in this tree.

- How many ways can you form the group if you are part of it?
- What if you are not part of the group?
- What if you want to limit the group size to K? Assume $K \leq 100$.

Problem 2 – Solution (Part 1)

Let $f(u)$ = number of connected subgraphs of subtree rooted at u, that contains u

- For each child v of u, we either choose to include it ($f(v)$ ways), or we exclude it (exactly one way).
- $f(u) = \prod_{\text{child v}}(1 + f(v))$

f(root) gives the number of ways to select a group that contains you!

Time complexity: $O(N)$

Problem 2 – Solution (Part 2)

Let $f(u)$ = number of connected subgraphs of subtree rooted at u, that contains u (we compute this as usual)

Let $g(u)$ = number of connected subgraphs of subtree rooted at u, that does NOT contain u

- For each child v of u, we just add all possible ways of putting the subgraph in that child – since we can't use more than one child without touching the node.
- $g(u) = \sum_{\text{child v}}(f(v) + g(v))$

$g(\text{root})$ gives the number of ways to select a group without you!

Time complexity: $O(N)$

Problem 2 – Solution (Part 3)

We care about size now, so let's add another state to track size.

Problem 2 – Solution (Part 3)

We care about size now, so let's add another state to track size.

Let $f(u, k)$ = number of connected subgraphs of size k containing u in subtree rooted at u

Let $g(u, k)$ = number of connected subgraphs of size k NOT containing u in subtree rooted at u

Problem 2 – Solution (Part 3)

We care about size now, so let's add another state to track size.

Let $f(u, k)$ = number of connected subgraphs of size k containing u in subtree rooted at u

Let $g(u, k)$ = number of connected subgraphs of size k NOT containing u in subtree rooted at u

- At each node, we use the same idea as before, but all we need to do is replace the simple product with a knapsack-like DP

Problem 2 – Solution (Part 3)

We care about size now, so let's add another state to track size.

Let $f(u, k)$ = number of connected subgraphs of size k containing u in subtree rooted at u

Let $g(u, k)$ = number of connected subgraphs of size k NOT containing u in subtree rooted at u

- At each node, we use the same idea as before, but all we need to do is replace the simple product with a knapsack-like DP
- $h(i, j) =$ number of connected subgraphs of size j containing u in subtree rooted at u, given that we only use children $1, \ldots, i$
- $h(0, 1) = 1$, $h(0, j > 1) = 0$, $h(i, j) = \sum_{k=1}^{j} h(i-1, j-k) \cdot f(i, k)$

Problem 2 – Solution (Part 3)

We care about size now, so let's add another state to track size.

Let $f(u, k)$ = number of connected subgraphs of size k containing u in subtree rooted at u

Let $g(u, k)$ = number of connected subgraphs of size k NOT containing u in subtree rooted at u

- At each node, we use the same idea as before, but all we need to do is replace the simple product with a knapsack-like DP
- $h(i, j) =$ number of connected subgraphs of size j containing u in subtree rooted at u, given that we only use children $1, \ldots, i$
- $h(0, 1) = 1$, $h(0, j > 1) = 0$, $h(i, j) = \sum_{k=1}^{j} h(i-1, j-k) \cdot f(i, k)$
- $f(u, k) = h(\#\text{child}, k)$
- $g(u, k) = \sum_{\text{child v}} (f(v, k) + g(v, k))$

Problem 2 – Solution (Part 3)

We care about size now, so let's add another state to track size.

Let $f(u, k)$ = number of connected subgraphs of size k containing u in subtree rooted at u

Let $g(u, k)$ = number of connected subgraphs of size k NOT containing u in subtree rooted at u

- At each node, we use the same idea as before, but all we need to do is replace the simple product with a knapsack-like DP
- $h(i, j) =$ number of connected subgraphs of size j containing u in subtree rooted at u, given that we only use children $1, \ldots, i$
- $h(0, 1) = 1$, $h(0, j > 1) = 0$, $h(i, j) = \sum_{k=1}^{j} h(i-1, j-k) \cdot f(i, k)$
- $f(u, k) = h(\#\text{child}, k)$
- $g(u, k) = \sum_{\text{child } v} (f(v, k) + g(v, k))$

$\sum_{k=1}^{K} f(\text{root}, k)$ and/or $\sum_{k=1}^{K} g(\text{root}, k)$ now gives the answer!
Time complexity: $O(NK^2)$

A different kind of tree DP

Lowest Common Ancestor

Given a tree of $N \leq 100,000$ nodes, answer $Q \leq 100,000$ queries of:

What is the lowest common ancestor of u and v?

LCA DP – Filling the DP Table

There are many solutions, but the following one is particularly useful as it generalizes to many query problems about paths on a tree.

DP State

- `depth[u]` = "depth of u" = number of nodes from the root
- `par[u][k]` = the 2^k-th ancestor of u

How do we fill the `par` array efficiently?

- Fill out the `par` array for all ancestors first (base case = root)
- `par[u][0]` = parent of u
- `par[u][k] = par[par[u][k-1]][k-1]`

Time complexity: $O(N \log N)$

LCA DP – Computing the LCA

DP State

- `depth[u]` = "depth of u" = number of nodes from the root
- `par[u][k]` = the 2^k-th ancestor of u

Algorithm Outline

- Move up the deeper node: say u is deeper, then iteratively replace `u = par[u][k]` where k is the largest integer such that $\texttt{depth[u]} - 2^k \geq \texttt{depth[v]}$
- Move both nodes up to their LCA: iteratively replace `u = par[u][k], v = par[v][k]` where k is the largest integer such that `par[u][k] != par[v][k]`

Time complexity: $O(\log N)$

LCA DP – Computing the LCA

```
1  LCA(u, v):
2    if (depth[u] < depth[v]):
3      swap(u, v)
4    for k = log N to 0:
5      if depth[u] - 2^k >= depth[v]:
6        u = par[u][k]
7    for k = log N to 0:
8      if par[u][k] != par[v][k]:
9        u = par[u][k], v = par[v][k]
10   if u != v:
11     u = par[u][0], v = par[v][0]
12   return u
```

Problem 3

You are given a weighted tree of $N \leq 100,000$ nodes.
Answer $Q \leq 100,000$ queries of the following form:

> What is the length of the shortest path between u and v?

Problem 3 – Solution

DP State

- `par[u][k]` = the 2^k-th ancestor of u
- `dist[u][k]` = length of path from u to the 2^k-th ancestor

Problem 3 – Solution

DP State

- `par[u][k]` = the 2^k-th ancestor of u
- `dist[u][k]` = length of path from u to the 2^k-th ancestor

Recurrence

- `par[u][k] = par[par[u][k-1]][k-1]`
- `dist[u][k] = dist[u][k-1] + dist[par[u][k-1]][k-1]`

Problem 3 – Solution

DP State

- `par[u][k]` = the 2^k-th ancestor of u
- `dist[u][k]` = length of path from u to the 2^k-th ancestor

Recurrence

- `par[u][k] = par[par[u][k-1]][k-1]`
- `dist[u][k] = dist[u][k-1] + dist[par[u][k-1]][k-1]`

Answer

- Find the LCA of u and v
- Use a similar algorithm to find distance from u and v to the LCA
- Output $dist(u \to LCA(u, v)) + dist(LCA(u, v) \to v)$

Time complexity: $O(N \log N + Q \log N)$

Can we DP on a general graph?

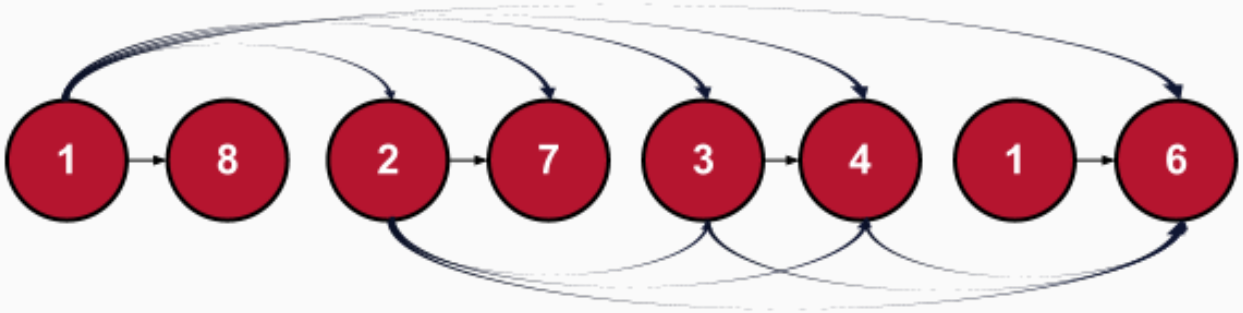
When the graph is a DAG

Remember Longest Increasing Subsequence?

When the graph is a DAG

Remember Longest Increasing Subsequence?

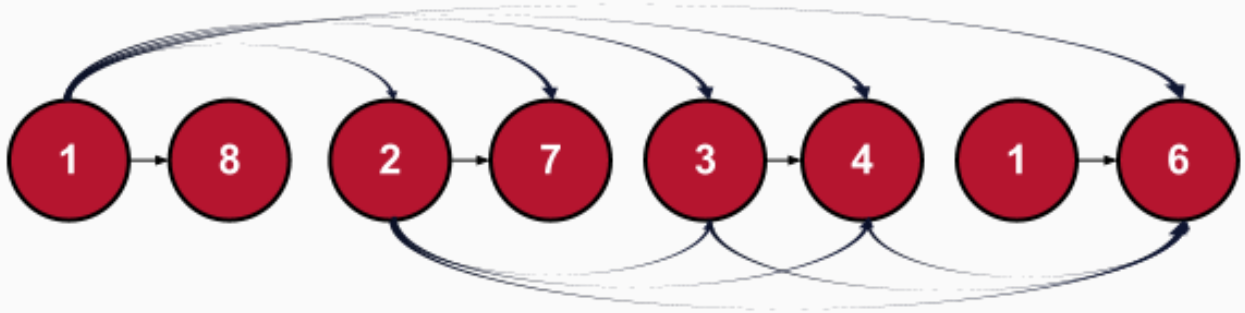
Construct graph where each element of the array is a node, and add an edge $i \to j$ if $i < j$ and $A[i] < A[j]$, then we have a DAG!



When the graph is a DAG

Remember Longest Increasing Subsequence?

Construct graph where each element of the array is a node, and add an edge $i \to j$ if $i < j$ and $A[i] < A[j]$, then we have a DAG!



Finding the LIS is equivalent to finding the longest path in this DAG!

DP for longest path on a DAG: $f(u) = \max_{v \to u} \left(f(v) + d(v, u) \right)$

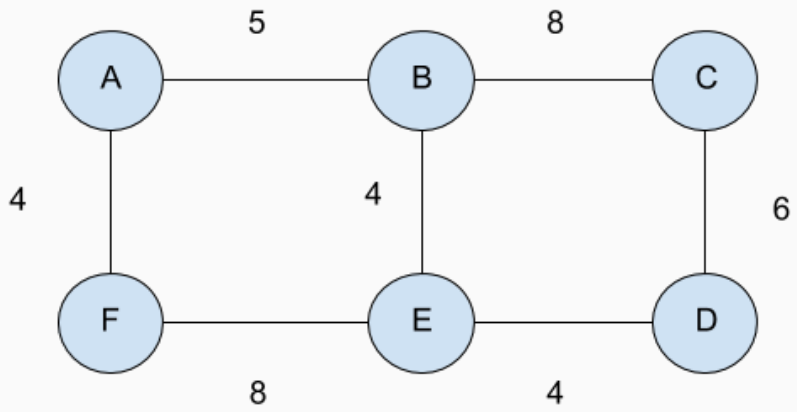
Reducing a graph into a tree or DAG

Remember Maximum Bandwidth? We can now do it FAST!

Reducing a graph into a tree or DAG

Remember Maximum Bandwidth? We can now do it FAST!

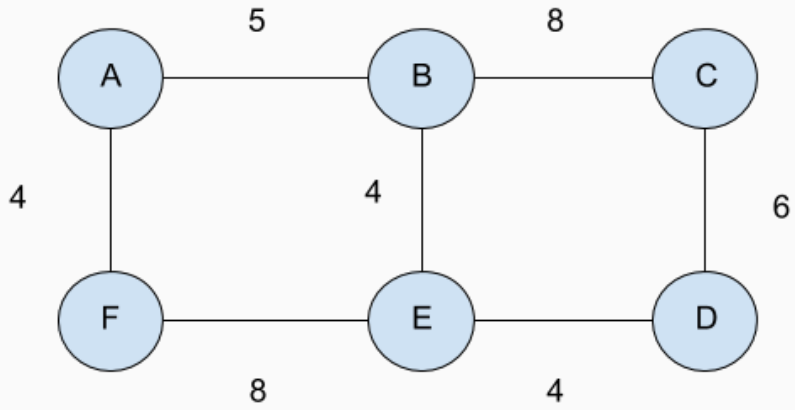
Given a graph ($1 \leq |V|, |E| \leq 100,000$), answer $Q \leq 100,000$ queries about bandwidth of the max-bandwidth path between two nodes.



Reducing a graph into a tree or DAG

Remember Maximum Bandwidth? We can now do it FAST!

Given a graph ($1 \leq |V|, |E| \leq 100,000$), answer $Q \leq 100,000$ queries about bandwidth of the max-bandwidth path between two nodes.



Solution: similar to Problem 3, with `f[u][k]` = the weight of the minimum edge between u and the 2^k-th ancestor of u, on the MST

Other useful trees/DAGs to extract from graph: SCCs, BCCs, etc.

When the recurrence is cyclic

Usually if you have cycles in your recurrence relation, it's a sign that you are doing it wrong!

When the recurrence is cyclic

Usually if you have cycles in your recurrence relation, it's a sign that you are doing it wrong!

However, sometimes you will need to do circular "DP" in which case there are a few ways out

When the recurrence is cyclic

Usually if you have cycles in your recurrence relation, it's a sign that you are doing it wrong!

However, sometimes you will need to do circular "DP" in which case there are a few ways out

- You may be able to rearrange it to be non-circular
  $f(n) = \sum_{k=0}^{n} f(k) \Rightarrow f(n) = \frac{1}{1-n} \sum_{k=0}^{n-1} f(k)$

When the recurrence is cyclic

Usually if you have cycles in your recurrence relation, it's a sign that you are doing it wrong!

However, sometimes you will need to do circular "DP" in which case there are a few ways out

- You may be able to rearrange it to be non-circular
  $f(n) = \sum_{k=0}^{n} f(k) \Rightarrow f(n) = \frac{1}{1-n} \sum_{k=0}^{n-1} f(k)$
- If the total number of DP values you are computing is small, you may be able to solve system of linear equations
  - Variables = each possible value you are finding (e.g. $f(i, j)$)
  - Equations = recurrence relation applied to each variable

When the recurrence is cyclic

Usually if you have cycles in your recurrence relation, it's a sign that you are doing it wrong!

However, sometimes you will need to do circular "DP" in which case there are a few ways out

- You may be able to rearrange it to be non-circular
  $f(n) = \sum_{k=0}^{n} f(k) \Rightarrow f(n) = \frac{1}{1-n} \sum_{k=0}^{n-1} f(k)$
- If the total number of DP values you are computing is small, you may be able to solve system of linear equations
  - Variables = each possible value you are finding (e.g. $f(i, j)$)
  - Equations = recurrence relation applied to each variable
- What if recurrence is not linear? An iterative solution may work!
  - For example, initialize $f(0) = 1$ and $f(n) = 0$ for $n > 1$, then blindly recompute 1000 times using the recurrence.

References

http://codeforces.com/blog/entry/20935

Next Class

String Algorithms

(In other words, more DP!)