Main topics are graph basics, including topological sort and dynamic programming on DAGs.

Prelude

- Homework 3 is due Thursday March 14.

DFS

- Last time, we started discussing depth-first search or DFS, an \(O(V + E)\) graph traversal algorithm with nice structural properties.
- Here’s a version of it that passed a clock around.

Here’s an example of a depth-first search.

- Similar to rooted trees, we can use the v.pre labels to get a preordering of the vertices “abfgchdlokpeinjm” in that order, and the v.post labels to get a postordering “dkoplhcgfbamjnie” in that order.

Classifying Vertices and Edges

- So let’s say we’re in the middle of running a depth-first search. We can learn a lot about the structure of the graph by using this clock variable.
- Eventually, the algorithm will populate v.pre and v.post for every vertex \(v\).
- But suppose we’re midway through running DFS. Fix a vertex \(v\) and its eventual pre and post values. But consider the clock at the moment we pause the algorithm. \(v\) is
- new if clock < v.pre (DFS(v) has not yet been called)
- active if v.pre ≤ clock < v.post (DFS(v) has been called but not yet returned)
- finished if v.post ≤ clock (DFS(v) has returned)

Being active corresponds to a vertex being on the recursion stack. That means the active vertices form a directed path in G.

In turn, using these definitions, we can partition the edges into four classes depending on how they interact with the depth-first search tree. Unlike vertices, these classes apply to a run of DFS, not a particular moment in time during the run. Consider edge u → v and the moment when DFS(u) begins.

- If v is new, then either we call DFS(v) directly when we iterate over u → v, or another intermediate recursive call will mark v first. Either way, u.pre < v.pre < v.post < u.post.
  - If DFS(u) calls DFS(v) directly, u → v is called a tree edge.
  - Otherwise, u → v is called a forward edge.
- If v is active, then v is on the stack, so v.pre < u.pre < u.post < v.post. G has a directed path from v to u.
  - u → v is called a back edge.
- If v is finished, then v.post < u.pre.
  - u → v is called a cross edge.
- Note that u.post < v.pre cannot happen, because we would add v to the stack before finishing with u.

Again, this classification of edges depends upon the specific depth-first search tree we get, which depends upon the order in which we iterate over vertices and edges.

Detecting Cycles

- So why did we go through defining all these things? Well, we now have the tools to solve some real problems. And the solutions are surprisingly easy.
- First, let’s suppose we’re given a directed graph G. Are there any directed cycles in G?
- Lemma: Directed graph G has a cycle if and only if DFSAll(G) yields a back edge.
  - Suppose there is a back edge u → v. Then G has a directed path from v → u. That path plus u → v is a cycle.
• Suppose there is a cycle. Let $v$ be the first vertex of the cycle visited by DFSAll, and let $u \rightarrow v$ be the predecessor of $v$ in the cycle.
• I claim that DFS($v$) will eventually call DFS($u$).
• The call to DFS($v$) will reach all vertices reachable from $v$ that don’t require going through something already marked. In particular, the cycle itself is such a path to $u$ since $v$ is the first marked vertex.
• But then when DFS($u$) is called, we’ll see $u \rightarrow v$ is a back edge.
• Edge $u \rightarrow v$ is a back edge if and only if $u.post < v.post$. So we can compute a post ordering in $O(V + E)$ time and check if that’s the case for any edge $u \rightarrow v$. If not, there are no directed cycles. It’s only $O(E)$ more things to do after DFSAll, so still $O(V + E)$ time total.

**Topological Sort**

• But why do we care about directed cycles? Directed graphs without directed cycles are called directed acyclic graphs or DAGs.
• Every DAG has a topological ordering of its vertices. Formally, its a total order where $u < v$ if there is an edge $u \rightarrow v$. Less formally, we want to draw the vertices on a line going left to right so there are no edges going from right to left.
• The normal motivation for finding topological orderings is to decide what order to do certain operations. Imagine we have a Makefile with several targets. We could build a graph with targets as vertices and edges going from each target to those that depend on it being built first. You need to compile everything in a topological order.
• Topological orderings don’t exist if there are directed cycles: in any ordering the rightmost vertex of a cycle would have an edge going back to the left.
• However, if there are no directed cycles, there are no back edges after a DFSAll, meaning $u.post > v.post$ for every edge $u \rightarrow v$.
• So, going by decreasing $u.post$, or reverse post ordering, you get a topological ordering!

• In particular, every directed acyclic graph has a topological ordering.
If we want to put the vertices in a separate data structure in order, we can add them in reverse postorder by having a clock tick down from V to 1.

Again, it’s just DFSAll with some extra stuff attached, so $O(V + E)$ time.

Dynamic Programming on DAGs

- But now, let’s back up a bit. Earlier in the semester we were discussing dynamic programming.
- Suppose we have a recurrence to evaluate for some dynamic programming algorithm.
- The dependency graph has one vertex per subproblem and an edge $x \rightarrow y$ for every subproblem $y$ that $x$ depends upon.
- The dependency graph must be acyclic or the naive recursive algorithm would never halt.
- When you solve the recurrence using basic memoization without rewriting it as an iterative algorithm, you’re really doing a depth-first search of the dependency graph, and you’re computing the solutions to the subproblems in postorder.
- The final iterative dynamic programming algorithm you design and analyze is really you evaluating all the subproblems in reverse topological order (since edges point to dependencies, not the other way around).
- That said, we don’t literally do a DFS of the dependency graph. First, the graph is usually represented implicitly. We don’t record all the vertices and edges. Instead we just enumerate over each vertex’s subproblems. It’s essentially the same as if we were going over the adjacency list of a vertex, but technically it is a different.
- Also, we usually have a good idea of the structure of the dependency graph before we start running the algorithm. For example, edit distance uses a regular grid dependency graph with edges between horizontal, vertical, or diagonal neighbors. This structure means we can usually hard-wire the reverse topological order into our final algorithms as a collection of nested loops.
- But this observation that dynamic programming and depth-first search are the same thing can be very useful when dealing with certain problems that are actually defined on directed acyclic graphs.
The longest path problem takes a directed graph $G = (V, E)$ with edge weights $\ell : E \to \mathbb{R}$ and two vertices $s$ and $t$. We want the length of the longest path from $s$ to $t$ that does not repeat any vertices.

The problem is hard to solve in general graphs, but we can solve it quickly if $G$ is a DAG.

We'll do so by answering the more general question of longest path to $t$ from every vertex.

Let $LLP(v)$ be the length of the longest path from $v$ to $t$ or $-\infty$ if no such path exists. If $v = t$ then $LLP(v) = 0$. Otherwise, $G$ is a DAG. Once we choose an edge from which to leave $v$, the total length of the path will be the length of the edge plus the length of the path from $v$’s successor. The path from $v$’s successor can’t accidentally go back to $v$ since $G$ is a DAG, so

$$LLP(v) = \begin{cases} 0 & \text{if } v = t, \\ \max \{ \ell(v \to w) + LLP(w) \mid v \to w \in E \} & \text{otherwise,} \end{cases}$$

Here, a max over nothing gets $-\infty$ because $v$ must not have any outgoing edges.

The dependency graph for this recurrence is the input graph $G$. So evaluating the function using basic memoization is literally performing a depth-first search on $G$. It takes $O(V + E)$ time.

If we want to write it as a more standard dynamic programming algorithm, we can just fill in each $LLP$ value in postorder.

```
LONGESTPATH(s, t):
  for each node $v$ in postorder
    if $v = t$
      $v.LLP \leftarrow 0$
    else
      $v.LLP \leftarrow -\infty$
      for each edge $v \to w$
        $v.LLP \leftarrow \max \{ v.LLP, \ell(v \to w) + w.LLP \}$
  return $s.LLP$
```

Still looking at each edge once, so $O(V + E)$ time.

If you’d prefer traditional shortest paths in $O(V + E)$ time, use a min instead and let $+\infty$ be the fail value.

Oh, and remember maximum independent set on a tree from earlier in the semester? We did essentially the same thing. The dependency graph was the original tree plus edges going down to grandchildren. Postorder of the tree itself was a postorder for the resulting dependency graph.