CS 6363.003.21S Lecture 13—March 23, 2021

Main topics are #graph_basics, including #breadth-first_search and #depth-first_search.

Depth-first search

- Today, we're going to study properties and applications of depth-first search. Probably the most common way to implement depth-first search or DFS it is to use recursion.

```
DFS(v):
  mark v
  PreVisit(v)
  for each edge vw
      if w is unmarked
          parent(w) ← v
          DFS(w)
  PostVisit(v)
```

- The PreVisit and PostVisit are placeholder functions for doing whatever extra work you want to do before and after fully processing a node.
- We can extend this algorithm to mark all vertices in the graph by using a wrapper function.

```
DFSALL(G):
  PREPROCESS(G)
  for all vertices v
      unmark v
  for all vertices v
      if v is unmarked
          DFS(v)
```

- Like before, PreProcess lets us do a bit of work before working with G.
- We still mark each vertex once and therefore handle each directed edge once, so the running time is O(V + E).

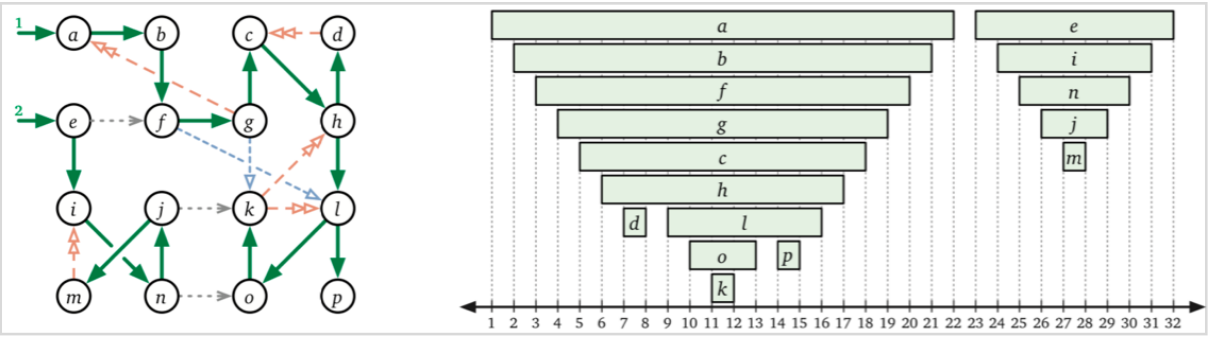
Preorder and Postorder

- The applications for DFS all come from the useful order in which it marks vertices.
- To see that, let's pass around a clock variable that increments every time we start or stop visiting a vertex.

```
DFSALL(G):
  clock ← 0
  for all vertices v
      unmark v
  for all vertices v
      if v is unmarked
          clock ← DFS(v, clock)
```

```
DFS(v, clock):
  mark v
  clock ← clock + 1;  v.pre ← clock
  for each edge v→w
      if w is unmarked
          w.parent ← v
          clock ← DFS(w, clock)
  clock ← clock + 1;  v.post ← clock
  return clock
```

- We assign v.pre just after pushing v onto the recursion stack and assign v.post just before popping it from the stack.
  - v.pre is often called the *starting time* of v.
  - v.post is often called the *finishing time* of v.
  - and [v.pre, v.post] is called the *active interval* of v.
- So, because stack timelines are always disjoint or nested, [u.pre, u.post] and [v.pre, v.post] are either disjoint or nested. In fact, [u.pre, u.post] contains [v.pre, v.post] if and only if DFS(v) is called during the execution of DFS(u).
- And because we only make recursive calls when there are edges, there must be a directed path from u to v in this case. In particular, the set of vertices on the recursion stack form a directed path in G.
- Here's an example of a depth-first search.

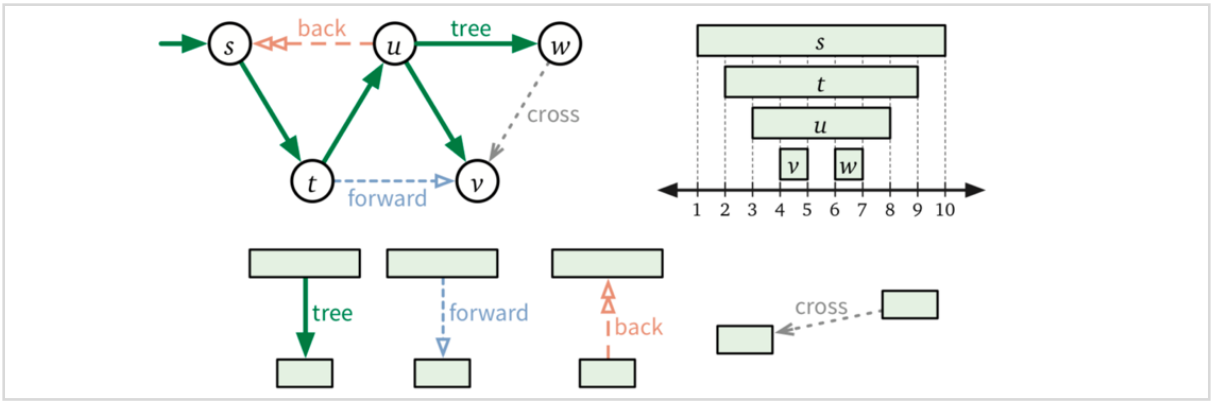


- Similar to rooted trees, we can use the v.pre labels to get a *preordering* of the vertices "abfgchdlokpeinjm" in that order, and the v.post labels to get a *postordering* "dkoplhcgfbamjnie" in that order.

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 final depth-first search forest. Unlike vertices, these classes apply to an entire *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.



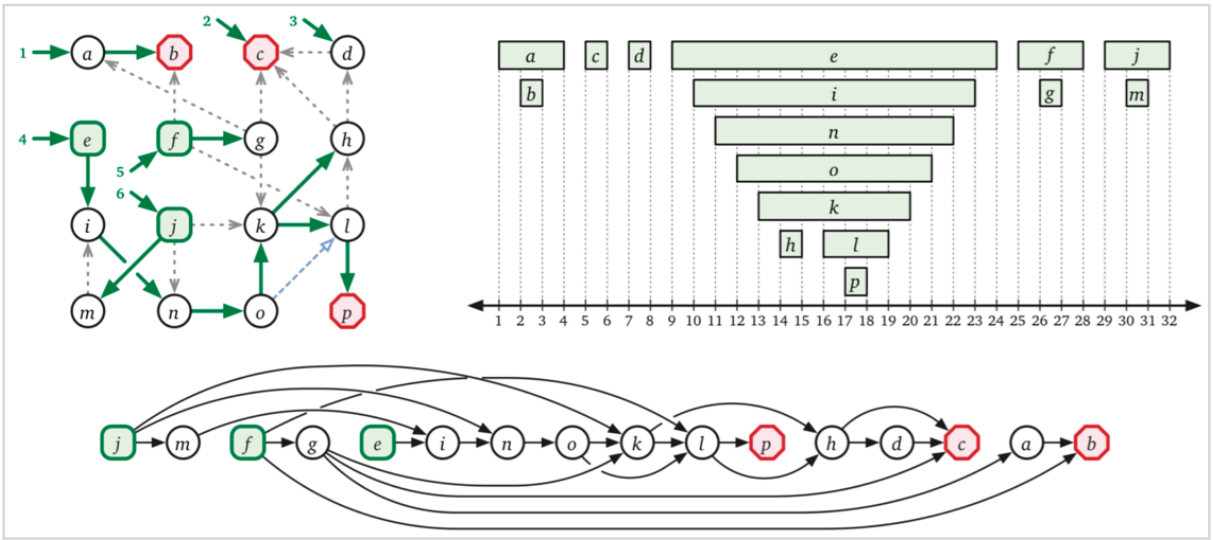
Detecting Cycles

- 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 ➔ v be the predecessor of v in the cycle.
  - I claim that DFS(v) will eventually call DFS(u).

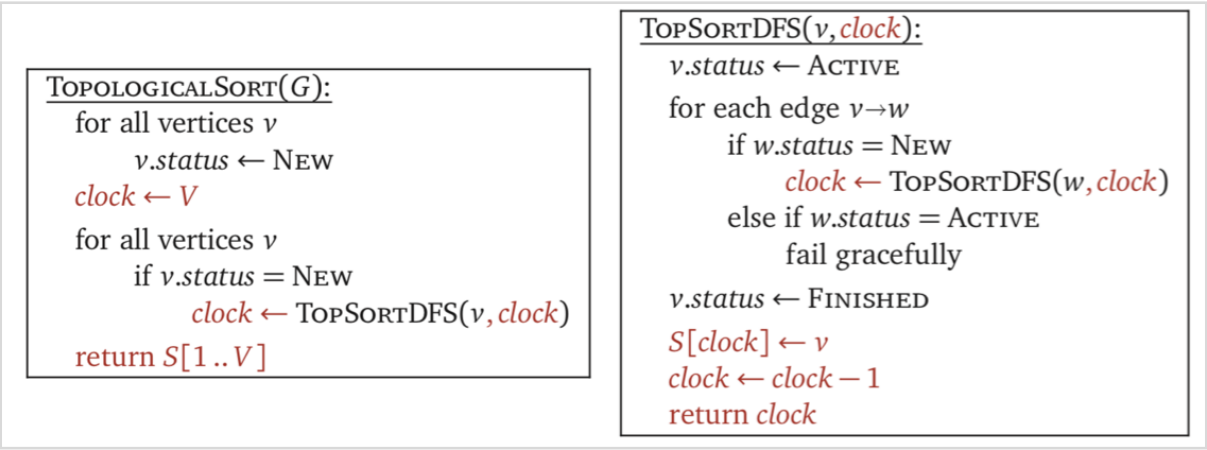
    - The call to DFS(v) will reach all vertices reachable from v that don't require going through something already marked. In particular, the cycle itself is such a path to u since v is the first marked vertex.
    - But then when DFS(u) is called, we'll see u ➜ v is a back edge.
- Edge u ➜ v is a back 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 ➜ 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 ➜ 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 ➜ 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.



```
TopologicalSort(G):
  for all vertices v
      v.status ← New
  clock ← V
  for all vertices v
      if v.status = New
          clock ← TopSortDFS(v, clock)
  return S[1..V]
```

```
TopSortDFS(v, clock):
  v.status ← Active
  for each edge v→w
      if w.status = New
          clock ← TopSortDFS(w, clock)
      else if w.status = Active
          fail gracefully
  v.status ← Finished
  S[clock] ← v
  clock ← clock − 1
  return clock
```

- 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 ➔ 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 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 ➔ 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 answer 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 -infinity if no such path exists. If v = t then LLP(v) = 0. Otherwise, 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

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

- Here, a max over nothing gets -infinity 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 ← 0
      else
          v.LLP ← −∞
          for each edge v→w
              v.LLP ← max{v.LLP, ℓ(v→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 +infinity be the fail value.