CS 4349.003.19F Lecture 14–October 14, 2019

Main topics for #lecture include #depth-first_search and #topological_sort.

Depth-first Search

- Last week, we discusses graph data structures and graph searching using a procedure called WhateverFirstSearch. Here it is for directed graphs.

```
WHATEVERFIRSTSEARCH(s):
  put s into the bag
  while the bag is not empty
      take v from the bag
      if v is unmarked
          mark v
          for each edge v→w
              put w into the bag
```

- Depending on what data structure you use for the "bag", you'll discover vertices in different orders. For example, a stack leads to a depth-first search or DFS. Intuitively, you keep drilling down down down as you discover new vertices reachable from s, only backing up when you have to.
- However, it's more common to describe a depth-first search using a recursive procedure.

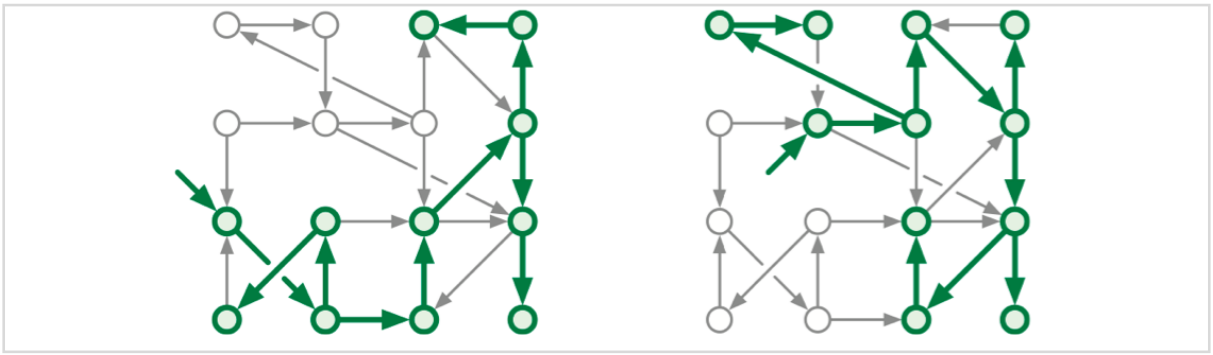
```
DFS(v):
  if v is unmarked
      mark v
      for each edge v→w
          DFS(w)
```

- Let's consider a couple modifications to make the algorithm more efficient in practice and to also let us use DFS for other things. First, we'll check if a vertex is unmarked *before* we recursively explore it. Second, we'll add some unspecified PreVisit and PostVisit routines that can be used for applications. **[Write this on the board]**.

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

- As we saw last week, calling DFS(s) on a completely unmarked graph will result in us marking exactly the vertices reachable from s. In an undirected graph, those vertices form exactly s's connected component. Things are more subtle in a directed graph, though, as

the set of reachable vertices is not symmetric. We may reach different vertices depending on which s we choose.



- If we want to guarantee we eventually reach every vertex, we need a wrapper function like we saw last Wednesday. We can also do some preprocessing for the whole graph before running the for loop if we like. **[Write this on the board]**.

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

- We still mark each vertex once and therefore handle each directed edge once, so the running time is O(V + E).

Preorder and Postorder

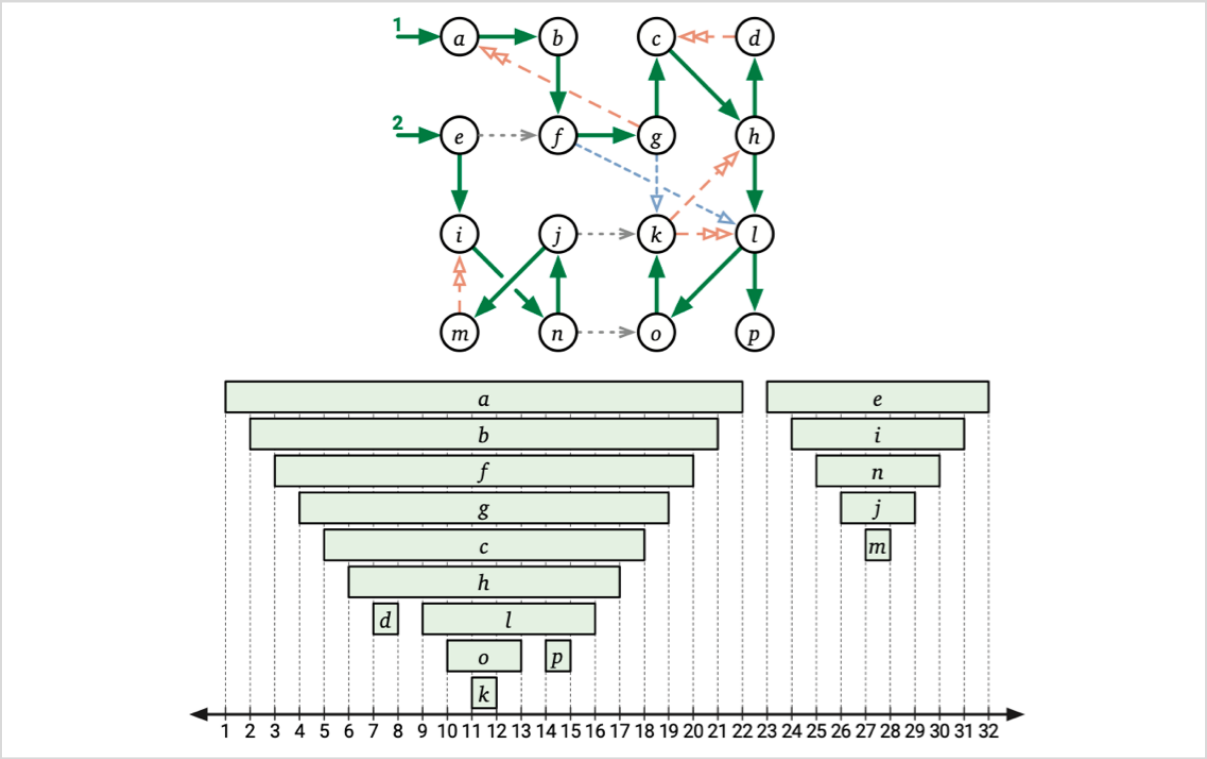
- So what kind of processing should we do? Well, the applications for DFS all come from the useful order in which it marks vertices.
- To see that, let's use those procedures to maintain a clock variable that increments every time we start or stop visiting a vertex.

```
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 with the active intervals drawn below. The forest edges described by the parent variables are the solid ones in the figure.



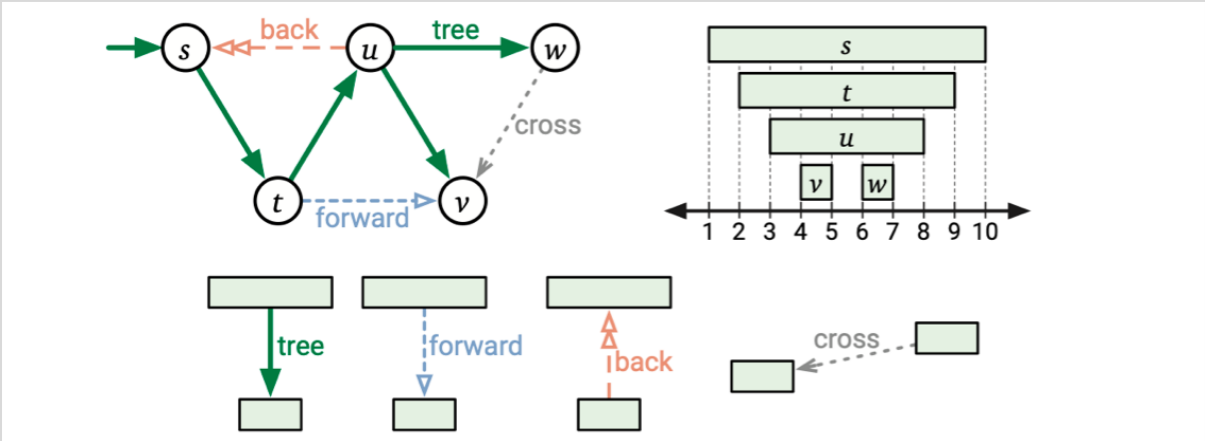
- Similar to rooted trees, we can use the v.pre labels to get a *preordering* of the vertices "abfgchdlokpeinjm" in that order, and the v.post labels to get a *postordering* "dkoplhcgfbamjnie" in that order.

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 in one of three states *at that time*.
  - *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 some u ➔ v.
  - If v is new when DFS(u) begins, 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, v.parent = u and u ➔ v is called a *tree edge*.
    - Otherwise, u ➔ v is called a *forward edge*.
  - If v is active when DFS(u) begins, then v is already 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 when DFS(u) begins, then v.post < u.pre.
    - u ➔ v is called a *cross edge*.
  - Note that u.post < v.pre cannot happen, because we must add v to the stack before finishing with u.



- The exact classification of edges we get depends upon the specific depth-first search trees we get, which depends upon the order in which we iterate over vertices and edges.

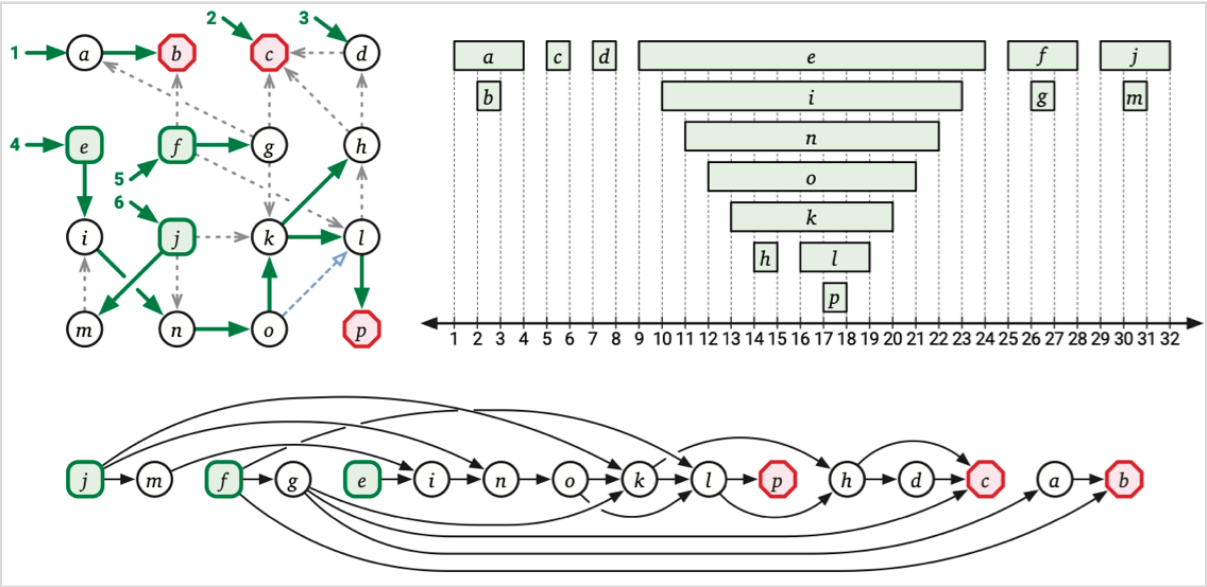
Detecting Cycles

- 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.
  - The call to DFS(v) will reach all vertices reachable from v that don't require going

through something already marked.
    - The cycle itself is such a path to u since v is the first marked vertex, so DFS(v) eventually calls DFS(u).
    - 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 here's an algorithm for detecting cycles:
    - Call DFSAll(G) to compute a post ordering in O(V + E) time.
    - For each edge u ➜ v
        - If u.post < v.post
            - Return "Cycle!"
    - Return "No cycle."
- 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 directed 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 call to 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(v, 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.
- There are many more applications of depth-first search including using it as a different way to think about or even implement dynamic programming algorithms. Unfortunately, we don't have time this semester to get into it. See Erickson 6 or CLRS Chapter 22 if you're interested.
- On Wednesday, we'll move away from graph searching and turn to another fundamental problem: computing minimum weight spanning trees.