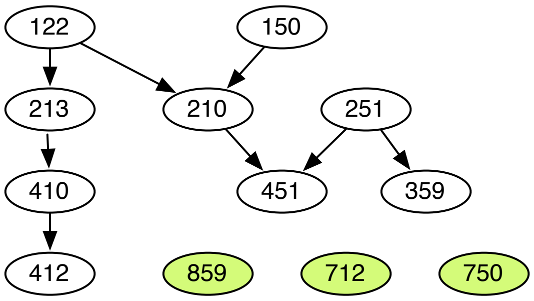
Lecture 18: Depth-First Search (DFS)

built on 2019/06/20 at 12:09:32

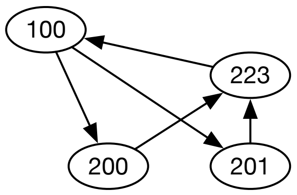
We will begin with a motivating toy problem.

1 A Toy Example

At an unknown college, Alice is an ambitious freshman who wants to do everything. She plans to take the following classes during her undergraduate career: 122, 150, 210, 213, 251, 359, 451, 410, 412, 750, 859, and 712. But she knows that she should only take one CS class per semester—and most classes have prerequisites that prevent her from getting in right away. According to her research, the course catalog indicates the following prerequisite structure, depicted as a directed graph, shown on left:



Example I: a typical prerequisite chain



Example II: a chain without a possible schedule

For example, she cannot take 451 until she is done with 210 and 251—although she could take 859 or 712 in her first semester (after all, most graduate courses don't have any formal prerequisite list).

We would like to help her *construct a schedule to take exactly one CS class per semester*. This is known known more formally as the *topological sort* problem or simply TopSort. But before we try to generate a schedule for her, we might be interested in finding out whether the graph has a cycle. In a more general context, what we have is a dependency graph and a cycle in a dependency graph indicates a deadlock. For Alice, this would mean such a schedule doesn't exist and she will never graduate if she insists on taking all these courses. In this lecture, we will also look at the cycle detection problem for both directed and undirected graphs.

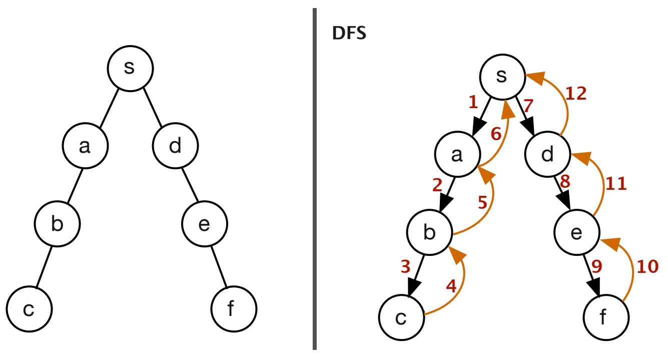
For both problems, we will develop an algorithm using a graph traversal idea called *depth-first search* that looks at an edge at most twice!

2 DFS: Depth-First Search

We'll spend the bulk of this lecture discussing a common graph search technique, known as depth-first search (DFS). The depth-first search approach proceeds by going as deep as it can until it runs out of unvisited vertices, at which point it backs out until it finds a node with an unvisited neighbor and goes there in the same manner.

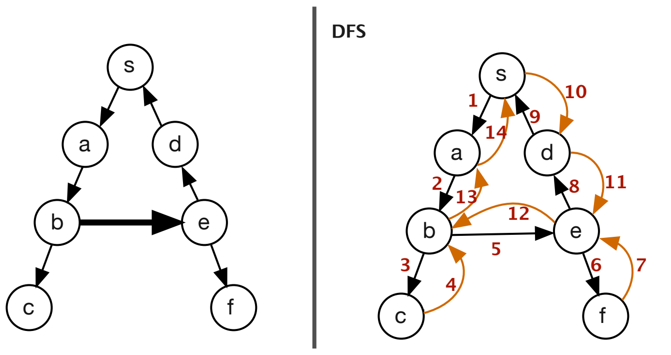
DFS can be used to find all vertices reachable from a start vertex v, to determine if a graph is connected, or to generate a spanning tree. But it will not find shortest paths. It is, however, useful in some other applications such as topologically sorting a directed graph (TopSort), cycle detection, etc. We will touch on some of these problems briefly.

Example 2.1 *Let's consider the following simple V-shaped graph:*



In this example, a DFS starting form s will go as deep as it can: at s, we could go to 1 *or* 4 *as the first vertex. Suppose we choose* 1*, then we will proceed to visit* 2 *and* 3*, then after hitting a dead end, we back out to* 2 *then back to* 1*, and proceed down* 4 *to* 5 *to* 6 *and back out.*

Example 2.2 *As another example, we'll look at a slightly more complicated version of the graph above, where we join the nodes b and e together with an edge* $b \to e$*.*



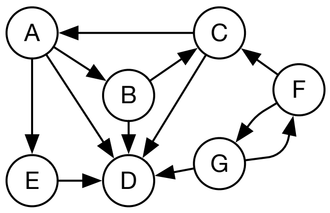
With the addition of just one edge, the order in which DFS visits vertices changes drastically. Suppose we start at s and choose to go down node 1 *first. We will visit* 2*,* 3*, then back out to* 2*, then since* 2 *has* 5 *as its neighbor, we'll visit* 5*, which in turn, will take us to* 4 *and to s, but s has been visited before, so we won't visit that—we back out. We'll then go to* 5*,*6*, back to* 5*, to* 2*, to* 1*, to s. We will try to visit* 4 *but quickly realize that* 4 *has been visited, so again, we back out.*

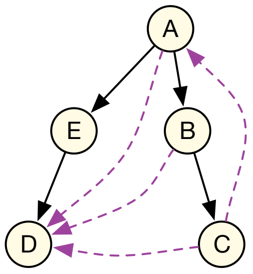
How do we turn this idea into code? Let's first consider a simple version of depth-first search that simply returns a set of reachable vertices. In the algorithm below, G is a graph we're traversing and `visited` is the set of vertices that we have visited already. Furthermore, we assume we have the function `neighbors(G, vtx)` that gives the neighbors of `vtx` as an iterable. To be more precise, this represents the out neighbors in the case of a directed graph and all the neighbors in the case of an undirected graph.

```
// G: graph, visited: set of visited nodes, v: a node
void dfsHelper(Graph G, Set<Integer> visited, int v) {
  if (!visited.contains(v)) {
    visited.add(v); // vtx v is visited
    for (Integer w : G.adj(v))
      dfsHelper(G, visited, w);
  }
}
void dfs(UndirectedGraph G, int v) {
  Set<Integer> visited = new HashSet<>();
  dfsHelper(G, visited, v);
}
```

What this means for the DFS algorithm is that when the algorithm visits a vertex v (i.e., `dfsHelper` on v is called), it picks the first outgoing edge vw_1 and calls `dfsHelper` on w_1 to fully explore the graph reachable through vw_1. We know we have fully explored the graph reachable through vw_1 when the call `dfsHelper` on w_1 that we made returns. The algorithm then picks the next edge vw_2 and fully explores the graph reachable from that edge. The algorithm continues in this manner until it has fully explored all out-edges of v. At this point, the looping is complete—and the call `dfsHelper` on v returns.

To gain a better understanding, can we predict what will happen if we call `dfs` on the following graph? (Answer: the tree on the right, where the dashed lines indicate "attempting to visit an already visited vertex and backing out.")



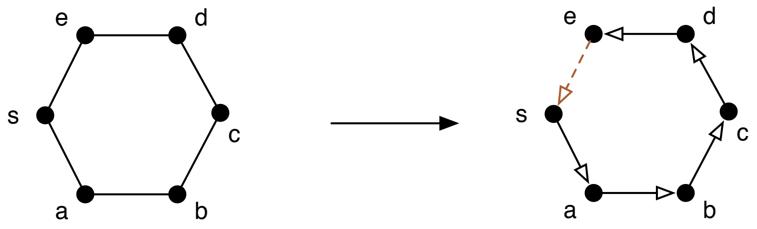


2.1 Cycle Detection — Take I

Let $G = (V, E)$ be a simple, undirected graph. The *cycle detection* problem on an undirected graph asks, is there a cycle in G? How would we modify the generic DFS algorithm above to solve this problem? The crucial observation is the following:

> **Observation:** If there is path from u to v (possibly, $v = u$ itself), then DFS from u will eventually reach v.

To see this in context, consider running DFS on C_6 (pictured below).



If we start at s, we'll eventually get back to s from a different direction. If, however, we drop an edge, what we have is a line graph on 6 nodes and as expected, DFS will reach all the vertices but cannot find a way to back to the starting point. This intuition leads to the following code (which assumes that the graph is connected):

```java
import java.util.*;

class CDState {
  public boolean foundCycle;
}

public class CycleDetection {
  static void dfsHelper(UndirectedGraph G, Set<Integer> visited,
                        int v, int parent, CDState state) {
    if (!visited.contains(v)) {
      visited.add(v); // vtx v is visited
      for (Integer w : G.adj(v)) {
        // visit everything except the immediate node we came from
        if (w!=parent) dfsHelper(G, visited, w, state);
      }
    }
    else {
      state.foundCycle = true;
    }
  }
  public static boolean cycleDetection(UndirectedGraph G, int v) {
    Set<Integer> visited = new HashSet<>();
    CDState state = new CDState();
    state.foundCycle = false;
    dfsHelper(G, visited, v, state);
    return state.foundCycle;
  }
}
```

The code relies on the fact that there is a cycle iff. we see a vertex we have already seen. We leave the proof of correctness as an exercise to the reader. For now, several things are clear from this code:

First, a DFS traversal looks like sequence of events where we enter and exit vertices or simply "touch" a vertex if we have seen it before. Moreover, each enter event has a corresponding exit event. That is to say, if DFS is called at a node u, that call will return at some point.

Second, every edge reachable from the starting point is looked at at least once and at most twice. As an immediate consequence of this observation, we have the following lemma:

Lemma 2.3 *Depth-first search takes $O(m + n)$ time assuming each set operation takes $O(1)$ time.*

Proof: Each edge in the graph is traversed at most twice and every time we traverse an edge, we do an `in` operation on `visited`, which we assume to cost $O(\log n)$. Moreover, for each vertex v visited by DFS, we compute `visited` $\cup \{v\}$, which costs $O(1)$. Adding up these two costs gives us the claimed bound. ∎

2.2 Beefing Up and Abstracting DFS

Building on an observation we made earlier, a DFS traversel can be described in terms of 3 types of events: entering a vertex, exiting a vertex, and touching (i.e., revisiting a node we've already discovered). Indeed, most algorithms based on DFS can be expressed with operations associated with these events. The *enter* operation is applied when first entering the vertex, the *exit* operation is applied upon leaving the vertex when all neighbors have been explored, and the *touch* operation is applied when the algorithm attempts to visit a vertex that has already been entered (but not necessarily exited).

In other words, depth-first search defines an ordering—the *depth-first ordering*—on the vertices where each vertex is visited twice, and the following code "treads through" the vertices in this order, applying *enter* the first time a particular vertex is visited and *exit* the other time that vertex is seen.

As such, we can write a template for DFS algorithms based on these operations:

```
def dfsHelper(G, visited, state, v):
    if v not in visited:
        # enter v, update state accordingly
        visited.add(v)
        for w in neighbors(G, v):
            dfsHelper(G, visited, state, w)
        # exit v, update state accordingly
    else:
        # touch v
```

```
def dfs(G, s):
    initial_state = # fill in initial state
    dfsHelper(G, new HashSet(), initial_state, s)
```

2.3 Cycle Detection — Take II

Using the template code, we can easily write cycle detection as follows:

```
initial_state = new CDState() where initial_state.found_cycle = False
on touch:  state.found_cycle = True
on enter: do nothing
on exit: do nothing
```

There is a cycle if and only if the DFS's state has `found_cycle` to be True. Notice that this code may produce wrong answers when the graph is directed (do you see why?).

Exercise: Modify the algorithm to solve cycle detection in a directed graph.

3 Topological Sorting

We now look at an example that applies this approach to topological sorting (TopSort). For this problem, we'll only need the *exit*; thus, the *enter* function simply returns the state as-is.

Directed Acyclic Graphs. A directed graph that has no cycles is called a *directed acyclic graph* or DAG. DAGs have many important applications. They are often used to represent dependence constraints of some type. Indeed, one way to view a workflow is as a DAG where the vertices are the jobs that need to be done, and the edges the dependences between them (e.g. a has to finish before b starts). Mapping such a computation DAG into an execution plan so that all dependences are obeyed is a common task. The graph of dependencies cannot have a cycle, because if it did then the system would deadlock and not be able to make progress.

The idea of topological sorting is to take a directed graph that has no cycles and order the vertices so the ordering respects reachability. That is to say:

if a vertex u is reachable from v, then v must be lower in the ordering.

In other words, if we think of the input graph as modeling dependencies (i.e., there is an edge from v to u if u depends on v), then topological sorting finds a partial ordering that puts a vertex *after* all vertices that it depends on.

To make this view more precise, we observe that a DAG defines a so-called *partial order* on the vertices in a natural way:

For vertices $a, b \in V(G)$, $a \leqslant_p b$ if and only if there is a directed path from a to b[1]

Remember that a partial order is a relation $\leqslant_p$ that obeys

1. reflexivity — $a \leqslant_p a$,
2. antisymmetry — if $a \leqslant_p b$ and $b \leqslant_p a$, then $b = a$, and
3. transitivity — if $a \leqslant_p b$ and $b \leqslant_p c$ then $a \leqslant_p c$.

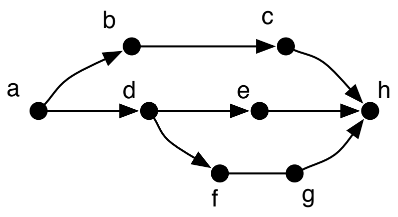
In this particular case, the relation is on the vertices. It's not hard to check that the relation based on reachability we defined earlier satisfies these 3 properties.

Armed with this, we can define the topological sorting problem formally:

Problem: A *topological sort* of a DAG is a total ordering $\leqslant_t$ on the vertices of the DAG that respects the partial ordering (i.e. if $a \leqslant_p b$ then $a \leqslant_t b$, though the other direction needs not be true).

For instance, consider the following DAG:

[1]We adopt the convention that there is a path from a to a itself, so $a \leqslant_p a$.



We can see, for example, that $a \leqslant_p c$, $d \leqslant h$, and $c \leqslant h$. But it is a partial order: we have no idea how c and g compare. From this partial order, we can create a total order that respects it. One example of this is the ordering $a \leqslant_t b \leqslant_t \leqslant_t c \leqslant_t d \leqslant_t e \leqslant_t f \leqslant_t g \leqslant_t h$. Notice that, as this example graph shows, there are many valid topological orderings.

Solving TopSort using DFS. Let's now go back to DFS. To apply the generic DFS discussed earlier, we need to specify what to do upon entering, exiting, and touching a vertex. For this, our state maintains a Stack of visited vertices (initially empty)—and we use the following *enter* and *exit*:

```
initial_state = new LinkedList()
on enter: do nothing
on exit: state.addFirst(v) // prepend
```

We claim that at the end, the ordering in the stack returned specifies the total order. (We'll need to pop out all the elements and put them in a list as we discover them.)

Why is this correct? The correctness crucially follows from the property that DFS fully searches any unvisited vertices that are reachable from it before returning. Consider any vertex $v \in V$. Suppose we first enter v with an initial list L_v. Now all unvisited vertices reachable from v, denoted by R_v, will be visited before exiting. But then, when exiting, v is placed at the start of the list (i.e., prepended to L_v). Therefore, all vertices reachable from v appear after v (either in L_e or in R_v), as required.

Finally, there is a slight problem that we need to address. In the current algorithm, we run DFS from some starting vertex, so if a vertex isn't reachable from the starting vertex, it won't be included in the output. How can we fix this? We can include a dummy source vertex o to make sure that we actually visit all vertices. This is a useful and common trick in graph algorithms.