Lecture 9 — Depth-First Search, TopSort, and Single-Threaded Arrays

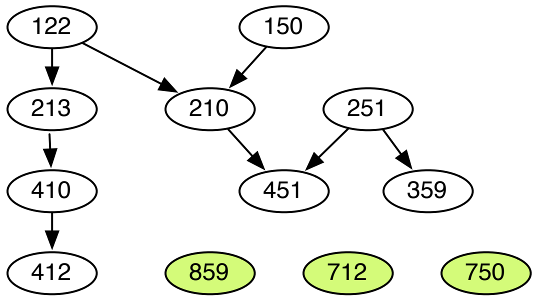
Parallel and Sequential Data Structures and Algorithms, 15-210 (Spring 2012)

Lectured by Kanat Tangwongsan — February 14, 2012

Material in this lecture: The main theme of this lecture is *depth-first search*.
- Depth-first Search
- Using DFS for Cycle Detection and Topological Sort
- Single-Threaded Arrays.

1 A Toy Example

Alice is an ambitious freshman who wants to be well-versed in both theory and systems. She plans to take the following classes during her undergraduate career: 15-122, 15-150, 15-210, 15-213, 15-251, 15-359, 15-451, 15-410, 15-412, 15-750, 15-859, and 15-712. 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:



For example, she cannot take 15-451 Algorithms Design and Analysis until she is done with 15-210 and 15-251—although she could take 15-859 Advanced Algorithms or 15-712 Advanced and Distributed Operating Systems in her first semester (after all, most graduate classes don't have any a formal prerequisite list).

We would like to help her *construct a schedule to take exactly one CS class per semester*. This is 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 unless she drops some of the classes. In this lecture, we will also look at the cycle detection problem for both directed and undirected graphs.

†Lecture notes by Guy E Blelloch, Margaret Reid-Miller, and Kanat Tangwongsan.

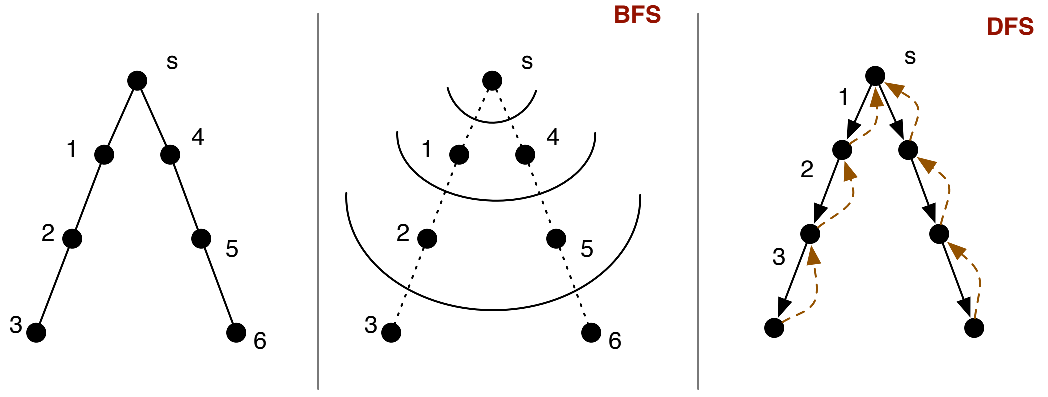
For both problems, we will develop an algorithm using a graph traversal idea called *depth-first search* that looks at an edge at most twice!

2 DFS: Depth-First Search

Last time, we looked at breadth-first search (BFS), a graph search technique which, as the name suggests, explores a graph in the increasing order of hop count from a source node. We'll spend the bulk of this lecture discussing another equally-common graph search technique, known as depth-first search (DFS). Unlike BFS which explores vertices one level at a time in a breadth first manner, 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.

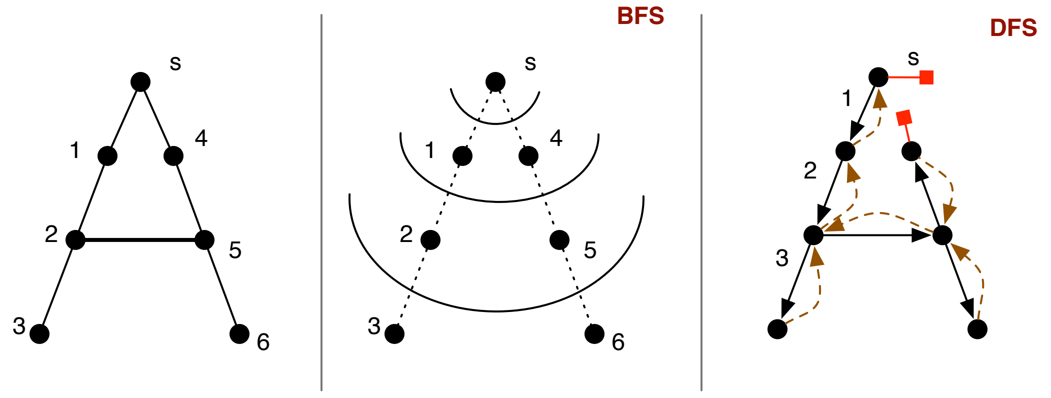
As with BFS, DFS can be used to find all vertices reachable from a start vertex v, to determine if a graph is connected, or to generate a spanning tree. Unlike BFS, it cannot be used to find shortest unweighted paths. But, instead, it is useful in some other applications such as topologically sorting a directed graph (TopSort), cycle detection, or finding the strongly connected components (Scc) of a graph. We will touch on some of these problems briefly.

Example 2.1. *To contrast BFS with DFS, let's consider the following simple V-shaped graph:*



In this example, a BFS starting from s will first visit s, then visit vertices at distance 1 *from s* (1 *and* 4), *then vertices at distance* 2 *from s* (2 *and* 5), *then vertices at distance* 3 *away* (3 *and* 6)*–and we're done. Whereas, 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* 2 *and* 5 *together with an edge.*



Nothing changes when we run BFS despite that extra edge between 2 *and* 5*; however, 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. Notice that in this case, the algorithm returns exactly the same set as BFS—but the crucial difference is that DFS visits the vertices in a different order (depth vs. breadth). In the algorithm below, X denotes the set of visited vertices—we call them X because they have been "crossed out." Like before, $N_G(v)$ represents the out-neighbors of v in the graph G.

```
fun DFS($G$,$v$) = let
  fun DFS'($X$,$v$) =
    if ($v \in X$) then $X$
    else iterate DFS' ($X \cup \{v\}$) ($N_G(v)$)
in DFS'({},$v$) end
```

The line `iterate DFS'` $(X \cup \{v\})$ $(N_G(v))$ deserves more discussion. First, the iterator concept is more or less universal and in broad strokes, `iterate` f `init` A captures the following:

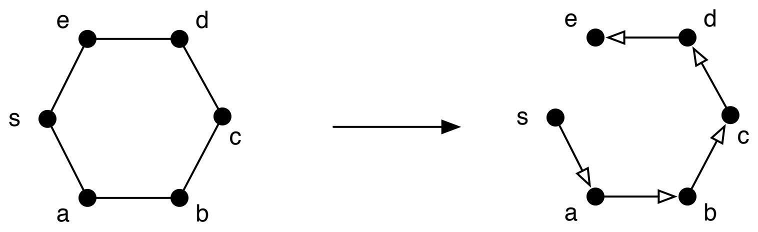
```
$S$ = init
foreach $a \in A$:  $S$ = $f$($S$, $a$)
```

This doesn't specify the order in which the elements of A are considered. All we know is that all of them are going to be considered in some order sequentially. What this means for the DFS algorithm is that when the algorithm visits a vertex v (i.e., DFS'(X, v) is called), it picks the first outgoing edge vw_1, through iterate, calls DFS'$(X \cup \{v\}, w_1)$ to fully explore the graph reachable through vw_1. We know we have fully explored the graph reachable through vw_1 when the call DFS'$(X \cup \{v\}, w_1)$ that we made returns. The algorithm then picks the next edge vw_2, again through iterate, 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, iterate is complete—and the call DFS'(X, v) returns.

As an example, if DFS'(X, v) is called on a vertex with $N_G(v) = \{w_1, w_2, w_3\}$ and iterate picks w_1, w_2, and w_3 in this order, then upon called, DFS'(X, v) will invoke DFS'$(X \cup \{v\}, w_1)$ and after this finishes, it will invoke DFS'(X', w_2), and DFS'(X'', w_3) in turn. Notice the difference between X, X' and X''—this is because DFS' accumulates visited vertices.

Is this parallel? At first look, we might think this approach can be parallelized by searching the outgoing edges in parallel. This would indeed work if the searches initiated never "meet up" (e.g., the graph is a tree, so then what's reachable through each edge would be independent). However, when the graphs reachable through the outgoing edges are shared, visiting them in parallel creates complications because we don't want to visit a vertex twice and we don't know how to guarantee that the vertices are visited in a depth-first manner.

To understand the situation better, we'll look at a simple example: a cycle graph on 6 nodes (C_6). The left figure shows a 6-cycle C_6 with nodes s, a, b, c, d, e and the right figure shows the order (as indicated by the arrows) in which the vertices are visited as a result of starting at s and first visiting a.



In this example, since the search wraps all the way around, we couldn't know until the end that e would be visited (in fact, it got visited last), so we couldn't start searching s's other neighbor e until we are done searching the graph reachable from a. More generally, in an undirected graph, if two unvisited neighbors u and v have any reachable vertices in common, then whichever is explored first will always wrap all the way around and visit the other one.

Indeed, depth-first search is known to be **P**-complete, a class of computations that can be done in polynomial work but are widely believed that they do not admit a polylogarithmic span algorithm. A detailed discussion of this topic is beyond the scope of our class. But this provides further evidence that DFS might not be highly parallel.

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 that if there is path from u to v (well, v could be u itself), then DFS from u will eventually reach v. To see this in context, consider running DFS on C_6, again. 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:

```
fun DFS($G$,$v$) = let
  fun DFS'(($X$,found_cycle),$v$) =
    if ($v \in X$) then ($X$,true)
    else iterate DFS' ($X \cup \{v\}$,found_cycle) ($N_G(v)$)
in DFS'(($\emptyset$,false),$v$) end
```

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. *The depth-first search algorithm has cost $O((m+n)\log n)$ work and span.*

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

2.2 Beefing Up 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:

```
fun DFS ($G,\Sigma_0,s$) = let
  fun DFS' $p$ (($X,\Sigma$),$v$) =
    if ($v \in X$) then (X, $\underline{\text{touch}}(\Sigma,v,p)$)
    else let
      val $\Sigma'$ = $\underline{\text{enter}}(\Sigma,v,p)$
      val ($X',\Sigma''$) = iterate (DFS' $v$) ($X \cup \{v\},\Sigma'$) $N_G(v)$
      val $\Sigma'''$ = $\underline{\text{exit}}(\Sigma'',v,p)$
    in ($X',\Sigma'''$) end
in DFS' nil (($\emptyset,\Sigma_0$),$s$) end
```

The updated algorithm bears much similarity to our basic DFS algorithm, except now we pass along a state (initially Σ_0) as we "tread through" the graph, applying the operations corresponding to the events we encounter. Each operation is supplied the current state, the current vertex v, and the parent vertex p (i.e., p is the vertex that calls DFS on v). At the end, DFS returns an ordered pair (X,Σ), which represents the set of vertices visitd and the final state Σ. Note also that in a directed graph, the neighborhood of v would be the set of outgoing edges $N^+(v)$.

2.3 Cycle Detection — Take II

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

```
$\Sigma_0$ = false
fun touch($\Sigma, v, p$) = true
fun enter($\Sigma, v, p$) = $\Sigma$
fun exit($\Sigma, v, p$) = $\Sigma$
```

There is a cycle if and only if DFS returns `(_, true)`. Notice that this code may produce wrong answers when the graph is directed (do you see why?).

Exercise 1. *Modify the code 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 parallel computation 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 onto processors so that all dependences are obeyed is the job of a scheduler. You have seen this briefly in 15-150. 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 $a, b \in V(G)$, $a \leq_p b$ if and only if there is a directed path from a to b[1]

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

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

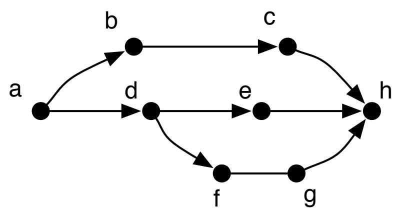
[1]We adopt the convention that there is a path from a to a itself, so $a \leq_p a$.

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 3.1 (Topological Sorting(TopSort)). A *topological sort* of a DAG is a total ordering $\leq_t$ on the vertices of the DAG that respects the partial ordering (i.e. if $a \leq_p b$ then $a \leq_t b$, though the other direction needs not be true).

For instance, consider the following DAG:



We can see, for example, that $a \leq_p c$, $d \leq h$, and $c \leq h$. 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 $a \leq_t b \leq_t \leq_t c \leq_t d \leq_t e \leq_t f \leq_t g \leq_t h$. 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 topologically sort a graph, we create a dummy source vertex o and add an edge from it to all other vertices. We then do run DFS from o. To apply the generic DFS discussed earlier, we need to specify two functions *enter* and *exit* and an initial state. For this, the state maintains a list of visited vertices (initially empty)—and we use the following *enter* and *exit* functions:

```
val $\Sigma_0$ = []
fun enter($\Sigma, v, p$) = $\Sigma$
fun exit($\Sigma, v, p$) = $v$::$\Sigma$
```

We claim that at the end, the ordering in the list returned specifies the total order.

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.

4 Single-Threaded Arrays

The challenge in using arrays in a functional setting is data persistency, so we have to be careful since by default, updating an array requires copying the whole array. This means that if we make an update, we also have to keep the old version. This also means we cannot simply overwrite a value in an existing array. The `ArraySequence` implementation meets the persistency requirement by generating a new copy of the sequence every time an operation is performed; while the implementation we have is persistent, it is very expensive to perform an update. In the following, we will describe an interface for a single-threaded sequence that supports efficient updates on the most recent version but works correctly on any version (the cost for performing an update on any other version is undefined). This is similar to the concept of benign effects covered in 15-150. As you know from 15-150, such an interface only appears functional to a sequential observer, which means that we won't be able to safely update single-threaded arrays in parallel.

We refer to the type of this sequence as a `stseq` and it supports the following interface:

	Work	Span
`fromSeq(S) :` α `seq` $\to$ α `stseq` Converts from a regular sequence to a stseq.	$O(\lvert S\rvert)$	$O(1)$
`toSeq(ST) :` α `stseq` $\to$ α `seq` Converts from a stseq to a regular sequence.	$O(\lvert S\rvert)$	$O(1)$
`nth ST` i `:` α `stseq` $\to$ `int` $\to$ α Returns the i^{th} element of ST. Same as for seq.	$O(1)$	$O(1)$
`update` (i, v) S `: (int` $\times$ α`)` $\to$ α `stseq` $\to$ α `stseq` Replaces the i^{th} element of S with v.	$O(1)$	$O(1)$
`inject` I S `: (int` $\times$ α`) seq` $\to$ α `stseq` $\to$ α `stseq` For each $(i, v) \in I$ replaces the i^{th} element of S with v.	$O(\lvert I\rvert)$	$O(1)$

The costs for `nth`, `update` and `inject` assume the user is using the most recent version. Once again, we will not define the costs unless operating on the most recent version.