Lecture 9 — DFS, Weighted Graphs, and Shortest Paths

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2011)

Lectured by Guy Blelloch — September 27, 2011

Today:

- Depth First Search
- Priority First Search
- Dijkstra's single source shortest path algorithm

1 Depth-First Search (DFS)

Last time, we looked at breadth-first search (BFS), a graph search technique which, as the name suggests, explores a graph an increasing order of hop count from a source node. In this lecture, we'll begin discussing another equally common graph search technique, known as depth-first search (DFS). Instead of exploring 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.

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), or finding the strongly connected components (SCC) of a graph. We will touch on these problems briefly.

For starters, we'll 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 ∈ X) then X
    else iterate DFS' (X ∪ {v}) (N_G(v))
in DFS'({}, v) end
```

The line `iterate DFS' (X ∪ {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 ∈ A:  S = f(S, a)
```

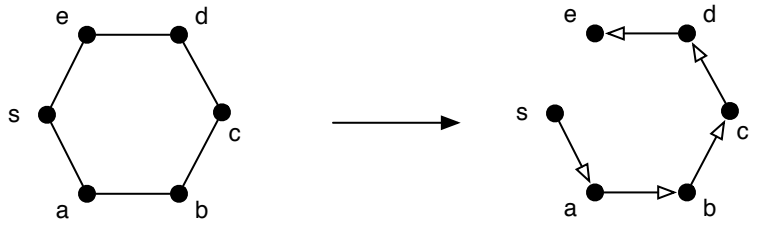
This doesn't specify the order in which the elements of A are considered. All we know is that all of them are going to be considered in some order sequentially. What this means for the DFS algorithm is that

when the algorithm visits a vertex v (i.e., $\texttt{DFS'}\,(X, v)$ is called), it picks the first outgoing edge vw_1, through iterate, calls $\texttt{DFS'}\,(X \cup \{v\}, w_1)$ to fully explore the graph reachable through vw_1. We know we have fully explored the graph reachable through vw_1 when the call $\texttt{DFS'}\,(X \cup \{v\}, w_1)$ 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 $\texttt{DFS'}\,(X, v)$ returns.

As an example, if $\texttt{DFS'}\,(X, v)$ 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, $\texttt{DFS'}\,(X, v)$ will invoke $\texttt{DFS'}\,(X \cup \{v\}, w_1)$ and after this finishes, it will invoke $\texttt{DFS'}\,(X', w_2)$, and $\texttt{DFS'}\,(X'', w_3)$ in turn. Notice the difference between X, X' and X''—this is because $\texttt{DFS'}$ 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 get a better sense of the situation, we'll consider 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 little example, since the search wraps all the way around, we couldn't know until the end that e would be visited (in fact it is 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 $\mathbf{P}$-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.

1.1 Beefing Up DFS

Thus far, our DFS function is only capable of reporting all vertices reachable from a given starting point. How can we change it so that DFS is a more useful building block? Interestingly, most algorithms based on DFS can be expressed with just two operations associated with each vertex visited: an *enter* operation and an *exit* operation. The *enter* operation is applied when first entering the vertex, and the *exit* operation is applied upon leaving the vertex when all neighbors have been explored.

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, DFS can be expressed as follows.

```
fun DFS (G, S_0, v) = let
  fun DFS' ((X, S), v) =
    if (v ∈ X) then (X, S)
    else let
      val S' = enter(S, v)
      val (X', S'') = iterate DFS' (X ∪ {v}, S') (N_G(v))
      val S''' = exit(S'', v)
    in (X', S''') end
in DFS'((({}, S_0), v) end
```

The updated algorithm bears much similarity to our basic DFS algorithm, except now we maintain a state that is effectively treaded through and where state transitions occur by applying the *enter* and *exit* functions in the depth-first order. At the end, `DFS` returns an ordered pair (X, S), which represents the set of vertices visitd and the final state S.

1.2 Topological Sorting

How can we apply the DFS algorithm to solve a specific problem? 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]

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

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

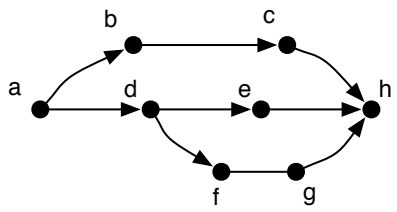
1. reflexivity — $a \leq_p a$,
2. antisymmetry — if $a \leq_p b$ and $b \leq_p a$, then $b = a$, and
3. transitivity — if $a \leq_p b$ and $b \leq_p c$ then $a \leq_p c$.

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 1.1 (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 S0 = []
fun enter(X,_) = X
fun exit(X,v) = v::X
```

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.

We included a dummy source vertex o to make sure that we actually visit all vertices. This is a useful and common trick in graph algorithms.

2 Priority First Search

Generalizing BFS and DFS, *priority first search* visits the vertices in some priority order. This priority order can either be static and decided ahead of time, or can be generated on the fly by the algorithm. To apply priority first search, we only need to make sure that at every step, we have a priority value for all the unvisited vertices adjacent to the visited vertices. This allows us to pick the best (highest priority) among them. When we visit a vertex, we might update the priorities of the remaining vertices. One could imagine using such a scheme for exploring the web so that the more interesting part can be explored without visiting the whole web. The idea might be to rank the outgoing links based on how interesting the tag on the link appears to be. Then, when choosing what link to visit next, choose the best one. This link might not be from the page you are currently on.

Many famous graph algorithms are instances of priority first search. For example, Dijkstra's algorithm for finding single-source shortest paths (SSSP) from a single source on a weighted graph and Prim's algorithm for finding Minimum Spanning Trees (MST).

3 Shortest Weighted Paths and Dijkstra's Algorithm

The single-source shortest path (SSSP) problem is to find the shortest (weighted) path from a source vertex s to every other vertex in the graph. We'll need a few definitions to describe the problem more formally. Consider a graph (either directed or undirected) graph $G = (V, E)$. A *weighted graph* is a graph $G = (V, E)$ along with a weight function $w \colon E \to \mathbb{R}$ that associates with every edge a real-valued weight. Thus, the *weight of a path* is the sum of the weights of the edges along that path.

Problem 3.1 (The Single-Source Shortest Path (SSSP) Problem)**.** Given a weighted graph $G = (V, E)$ and a source vertex s, the *single-source shortest path (SSSP) problem* is to find the shortest weighted path from s to every other vertex in V.

We will use $\delta_G(u, v)$ to indicate the weight of the shortest path from u to v in the weighted graph G. Dijkstra's algorithm solves the SSSP problem when all the weights on the edges are non-negative. Dijkstra's is a very important algorithm both because shortest paths have many applications but also because it is a very elegant example of an efficient greedy algorithm that generates optimal solutions on a nontrivial task.

Weighted Graph Representation. There are a number of ways to represent weights in a graph. More generally, we might want to associate any sort of value with the edges (e.g. in Assignment 4). That is, we have a "label" function of type $w \colon E \to$ `label` where `label` is the type of the label.

The first representation we consider translates directly from viewing edge labels as a function. We keep a table that maps each edge (a pair of vertex identifiers) to its label (or weight). This would have type

(`label` *vertexVertexTable*)

i.e. the keys would be pairs of vertices (hence *vertexVertexTable*), and the values are labels.

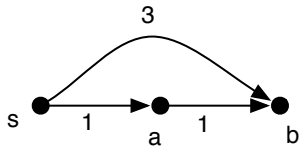
Another way to associate values with edges is to use a structure similar to the adjacency set representation for unweighted graphs and try to piggyback labels on top of it. In particular, instead of associating a

set of neighbors with each vertex, we can have a table of neighbors that maps each neighbor to its label (or weight). It would have type:

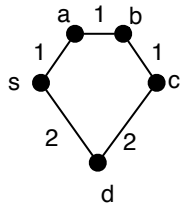
$$(\texttt{label } \textit{vertexTable})\textit{vertexTable}.$$

This is the representation we will be using for Dijkstra's algorithm.

Before describing Dijkstra's algorithm, we would like to understand why we need a new algorithm for the weighted case. Why, in particular, doesn't BFS work on weighted graphs? Consider the following directed graphs on 3 nodes:



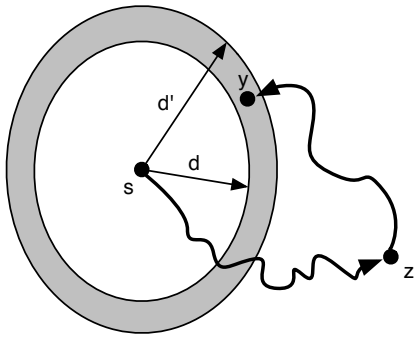
In this example, BFS would visit b then a. This means when we visit b, we assign it an incorrect weight of 3. Since BFS never visit it again, we'll never update it to the correct value. Note that we cannot afford to visit it again as it will be much more expensive. This problem still persists even if the graph is geometric (and satisfies the triangle inequality), as the following example shows:



The crux of Dijkstra's algorithm is the following lemma, which suggests prioritiy values to use and guarantees that such a priority setting will lead to the weighted shortest paths.

Lemma 3.2. *Consider a (directed) weighted graph $G = (V, E)$, $w\colon E \to \mathbb{R}_+ \cup \{0\}$ with no negative edge weights, a source vertex s and an arbitrary distance value $d \in \mathbb{R}_+ \cup \{0\}$. Let $X = \{v \in V : \delta(s, v) \leq d\}$ be the set of vertices that are at most d from s and $d' = \min\{\delta(s, u) : u \in V \setminus X\}$ be the nearest distance strictly greater than d. Then, if $V \setminus X \neq \emptyset$, there must exist a vertex u such that $\delta(s, u) = d'$ and a shortest path to u that only goes through vertices in X.*

Proof. Let $Y = \{v \in V : \delta(s, v) = d'\}$ be all vertices at distance exactly d'. Note that the set Y is nonempty by definition of d' and since $V \setminus X \neq \emptyset$.



Pick any $y \in Y$. We'll assume for a contradiction that that all shortest paths to y go through some vertex in $Z = V \setminus (X \cup Y)$ (i.e., outside of both X and Y). But for all $z \in Z$, $d(s,z) > d'$. Thus, it must be the case that $d(s,y) \geq d(s,z) > d'$ because all edge weights are non-negative. This is a contradiction. Therefore, there exists a shortest path from s to y that uses only the vertices in $X \cup Y$. Since $s \in X$ and the path ends at $y \in Y$, it must contain an edge $v \in X$ and $u \in Y$. The first such edge has the property that a shortest path to u only uses X's vertices, which proves the lemma. □

This suggests an algorithm that by knowing X, derives d' and one such vertex u. Indeed, X is the set of explored vertices, and we can derive d' and a vertex u attaining it by computing $\min\{d(s,x) + w(xu) : x \in X, u \in N_G(x)\}$. Notice that the vertices we're taking the minimum over is simply $N_G(X)$. The following algorithm uses a priority queue so that finding the minimum can be done fast. Furthermore, the algorithm makes use of a dictionary to store the shortest path distance, allowing for efficient updates and look-ups. We show the algorithm's pseudocode below.

```
fun dijkstra(G, u) =
  let fun dijkstra'(D, Q) =
      case (PQ.deleteMin(Q))
        of (PQ.empty, _) ⇒ D
        | ((d, v), Q) ⇒
          if ((v, _) ∈ D) Dijkstra'(D, Q)
          else let
            fun relax (Q, (u, w)) = PQ.insert (d + w, u) Q
            val Q' = iterate relax Q (N_G(v))
          in dijkstra'(D ∪ {(v, d)}, Q') end

  in dijkstra'({}, PQ.insert(empty,(0.0, u)))
  end
```

This version of Dijkstra's algorithm differes somewhat from another version that is sometimes used. First, the `relax` function is often implemented as a `decreaseKey` operation. In our algorithm, we simply add in a new value in the priority queue. Although this causes the priority queue to contain more entries, it doesn't affect the asympotic complexity and obviates the need to have the `decreaseKey` operation, which can be tricky to support in many priority queue implementations.

Second, since we keep multiple distances for a vertex, we have to make sure that only the shortest-path distance is registered in our answer. We can show inductively through the lemma we proved already that the first time we see a vertex v (i.e., when `deleteMin` returns that vertex) gives the shortest path to v. Therefore, all subsequence occurences of this particular vertex can be ignored. This is easy to support because we keep the shortest-path distances in a dictionary which has fast lookup.

4 SML code

Here we present the SML code for DFS and Dijkstra.

4.1 DFS with Enter and Exit Functions

```
functor TableDFS(Table : TABLE) =
struct
  open Table
  type vertex = key
  type graph = set table

  fun N(G : graph, v : vertex) =
    case (Table.find G v) of
        NONE => Set.empty
      | SOME(ngh) => ngh

  fun DFS (S0, enter, exit) (G : graph, s : vertex) =
    let
      fun DFS'((X : set, S), v : vertex) =
        if (Set.find X v) then (X, S)
        else
          let
            val S' = enter(S, v)
            val (X',S'') = Set.iter DFS' (Set.insert v X, S') (N(G,v))
            val S''' = exit(S'',v)
          in (X',S''')
          end
    in
      DFS'((Set.empty, S0), s)
    end
end
```

4.2 Topological Sort with DFS

```
functor TableTopSort(Table : TABLE) =
struct
  structure dfs = TableDFS(Table);

  fun topSort(G, v) =
  let
      val S0 = nil
      fun enter(S, v) = S
      fun exit(S, v) = v::S
      val (X, S) = dfs.DFS (S0, enter, exit) (G, v)
  in
      S
  end
end
```

4.3 Dijkstra

```
functor TableDijkstra(Table : TABLE) =
struct
```

```
  structure PQ = Default.RealPQ
  type vertex = Table.key
  type 'a table = 'a Table.table
  type weight = real
  type 'a pq = 'a PQ.pq
  type graph = (weight table) table

  (* Out neighbors of vertex v in graph G *)
  fun N(G : graph, v : vertex) =
    case (Table.find G v) of
      NONE => Table.empty()
    | SOME(ngh) => ngh

  fun Dijkstra(u : vertex, G : graph) =
    let
      val insert = Table.insert (fn (x,_) => x)

      fun Dijkstra'(Distances : weight table,
                    Q         : vertex pq) =
        case (PQ.deleteMin(Q)) of
            (NONE, _)       => Distances
          | (SOME(d, v), Q) =>
            case (Table.find Distances v) of

              (* if distance already set, then skip vertex *)
              SOME(_) => Dijkstra'(Distances, Q)

            | NONE =>
              let
                val Distances' = insert (v, d) Distances
                fun relax (Q,(u,l)) = PQ.insert (d+l, u) Q

                (* relax all the out edges *)
                val Q' = Table.iter relax Q (N(G,v))
              in
                Dijkstra'(Distances', Q')
              end
    in
      Dijkstra'(Table.empty(), PQ.singleton (0.0, u))
    end
end
```