1 Depth first search

1.1 The Algorithm

Besides breadth first search, which we saw in class in relation to Dijkstra's algorithm, there is one other fundamental algorithm for searching a graph: depth first search. To better understand the need for these procedures, let us imagine the computer's view of a graph that has been input into it, in the adjacency list representation. The computer's view is fundamentally *local* to a specific vertex: it can examine each of the edges adjacent to a vertex in turn, by traversing its adjacency list; it can also mark vertices as visited. One way to think of these operations is to imagine exploring a dark maze with a flashlight and a piece of chalk. You are allowed to illuminate any corridor of the maze emanating from your current position, and you are also allowed to use the chalk to mark your current location in the maze as having been visited. The question is how to find your way around the maze.

We now show how the depth first search allows the computer to find its way around the input graph using just these primitives.

Depth first search uses a **stack** as the basic data structure. We start by defining a recursive procedure search (the stack is implicit in the recursive calls of search): search is invoked on a vertex v, and explores all previously unexplored vertices reachable from v.

```
Procedure search(v)
    vertex v
    explored(v) := 1
    previsit(v)
    for (v, w) ∈ E
        if explored(w) = 0 then search(w)
    rof
    postvisit(v)
end search


Procedure DFS (G(V, E))
    graph G(V, E)
    for each v ∈ V do
        explored(v) := 0
    rof
    for each v ∈ V do
        if explored(v) = 0 then search(v)
    rof
end DFS
```

By modifying the procedures previsit and postvisit, we can use DFS to solve a number of important problems, as we shall see. It is easy to see that depth first search takes $O(|V| + |E|)$ steps (assuming

previsit and postvisit take $O(1)$ time), since it explores from each vertex once, and the exploration involves a constant number of steps per outgoing edge.

The procedure search defines a tree (** well, actually a *forest*, but let's not worry about that distinction right now **) in a natural way: each time that search discovers a new vertex, say w, we can incorporate w into the tree by connecting w to the vertex v it was discovered from via the edge (v, w). The remaining edges of the graph can be classified into three types:

- Forward edges - these go from a vertex to a descendant (other than child) in the DFS tree.
- Back edges - these go from a vertex to an ancestor in the DFS tree.
- Cross edges - these go from "right to left"– there is no ancestral relation.

Remember there are four types of edges; the fourth is the "tree edges", which were edges that led to a new vertex in the search.

Question: Explain why if the graph is undirected, there can be no cross edges.

One natural use of the previsit and postvisit procedures is that they could each keep a counter that is increased each time one of these routines is accessed; this corresponds naturally to a notion of time. Each routine could assign to each vertex a preorder number (time) and a postorder number (time) based on the counter. If we think of depth first search as using an explicit stack, then the previsit number is assigned when the vertex is first placed on the stack, and the postvisit number is assigned when the vertex is removed from the stack. Note that this implies that the intervals $[preorder(u), postorder(u)]$ and $[preorder(v), postorder(v)]$ are either disjoint, or one contains the other.

An important property of depth-first search is that the contents of the stack at any time yield a path from the root to some vertex in the depth first search tree. (Why?) This allows us to prove the following property of the postorder numbering:

Claim 1.
If $(u, v) \in E$ then $postorder(u) < postorder(v) \iff (u, v)$ is a back edge.

Proof. If $postorder(u) < postorder(v)$ then v must be pushed on the stack before u. Otherwise, the existence of edge (u, v) ensures that v must be pushed onto the stack before u can be popped, resulting in $postorder(v) < postorder(u)$ — contradiction. Furthermore, since v cannot be popped before u, it must still be on the stack when u is pushed on to it. It follows that v is on the path from the root to u in the depth first search tree, and therefore (u, v) is a back edge.

The other direction is trivial. □

Exercise. *What conditions do the preorder and postorder numbers have to satisfy for (u, v) to be a forward edge? A cross edge?*

Claim 2.
$G(V, E)$ has a cycle iff the DFS of $G(V, E)$ yields a back edge.

Proof. If (u, v) is a back edge, then (u, v) together with the path from v to u in the depth first tree form a cycle.

Conversely, for any cycle in $G(V, E)$, consider the vertex assigned the smallest postorder number. Then the edge leaving this vertex in the cycle must be a back edge by Claim 1, since it goes from a lower postorder number to a higher postorder number. □

1.2 Application of depth first search: Topological sorting

Consider the following problem: the vertices of a graph represent tasks, and the edges represent precedence constraints: a directed edge from u to v says that task u must be completed before v can be started. An important problem in this context is scheduling: in what order should the tasks be scheduled so that all the precedence constraints are satisfied?

We now suggest an algorithm for this scheduling problem. Given a directed graph $G(V, E)$, whose vertices $V = \{v_1, \ldots v_n\}$ represent tasks, and whose edges represent precedence constraints: a directed edge from u to v says that task u must be completed before v can be started. The problem of topological sorting asks: in what order should the tasks be scheduled so that all the precedence constraints are satisfied.

Note: The graph must be acyclic for this to be possible. (Why?) Directed acyclic graphs appear so frequently they are commonly referred to as DAGs.

Exercise. *If the tasks are scheduled by decreasing postorder number, then all precedence constraints are satisfied.*

There's another way to think about topologically sorting a DAG. Each DAG has a *source*, which is a vertex with no incoming edges. Similarly, each DAG has a *sink*, which is a vertex with no outgoing edges. (Proving this is an exercise.) Another way to topologically order the vertices of a DAG is to repeatedly output a source, remove it from the graph, and repeat until the graph is empty. Why does this work? Similarly, once could repeatedly output sinks, and this gives the reverse of a valid topological order. Again, why?

Exercise. *Give a linear-time algorithm that takes as input DAG G and two vertices s, t and returns the number of simple paths from s to t in G.*

2 Breadth-first search review

- While DFS uses a stack, BFS uses a **queue**, and that's what makes the two algorithms fundamentally different.
- BFS is also a $O(|E| + |V|)$ search, often done from just one node in a graph.
- BFS gives you a list of nodes in increasing path-length from source.

Exercise. *Suppose we defined forward, back, and cross edges analogously to DFS. Show that:*

1. *in a BFS of an unweighted directed graph, there are no forward edges.*
2. *in a BFS of an undirected graph, there are no forward or back edges. Moreover, for any cross edge (v, u), either $dist[v] = dist[u]$ or $dist[v] = dist[u] \pm 1$.*

3 Shortest Paths

3.1 Review

In class, we saw several shortest paths algorithms:

- **Djikstra's Algorithm** (single-source): modified BFS
- **Bellman-Ford** (single-source): for $|V|$ times, go through the edges and, upon looking at (u, v) update the shortest path to v if going to u and then via (u, v) would give a better answer than what we have so far
- **Floyd-Warshall** (all pairs): dynamic programming; the subproblems are $D(i, j, k)$, the shortest path from i to j using only intermediate vertices in $\{1, \ldots, k\}$

Question: What are the constraints on the graph for each of the above algorithms to work?

Question: What are the runtimes for each of these algorithms?

Exercise. *Consider the shortest paths problem in the special case where all edge costs are non-negative integers. Describe a modification of Dijkstra's algorithm that works in time $O(|E| + |V|m)$, where m is the maximum cost of any edge in the graph.*

3.2 Johnson's Algorithm

One of the difficulties that we encounter with Djikstra's algorithm is that of negative edge weights. The first way we might try to fix this problem is by adding some large number to the weight of every edge, in order to get rid of all the negative weights; however, this doesn't preserve the shortest paths. (Why?)

Instead, we will give a slightly different way to do this type of edge weight modification, known as Johnson's Algorithm.

Exercise. *Suppose that we assign some weight $w(v)$ to each vertex v, and that for each edge (u, v), we add $w(u) - w(v)$ to its weight. Prove that the shortest path between any pair of vertices is the same in the updated graph as in the original (albeit with a different weight).*

Exercise. *Here is a method for determining the edge weights $w(v)$: add a new vertex v_0, and add an edge (v_0, v) with weight 0 for each vertex v in the original graph. Then, we set $w(v)$ to be the length of the shortest path from v_0 to v.*

Exercise. *We can put together the facts above to obtain an algorithm: run Bellman-Ford to figure out the weights to assign to the vertices, augment the edge weights of hte graph, and then run Djikstra's Algorithm from each vertex.*

What is the runtime? How does it compare to Floyd-Warshall?

4 Linear programs: an example reduction

Exercise. *Express the following problem as a linear programming problem: A gold processor has two sources of gold ore, source A and source B. In order to kep his plant running, at least three tons of ore must be processed each day. Ore from source A costs \$20 per ton to process, and ore from source B costs \$10 per ton to process. Costs must be kept to less than \$80 per day. Moreover, Federal Regulations require that the amount of ore from source B cannot exceed twice the amount of ore from source A. If ore from source A yields 2 oz. of gold per ton, and ore from source B yields 3 oz. of gold per ton, how many tons of ore from both sources must be processed each day to maximize the amount of gold extracted subject to the above constraints?*

Exercise. *How would you encode TSP (the Traveling Salesman Problem) as an integer linear program? Recall that given a weighted graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}_{\geq 0}$, the TSP asks for a cycle in G that covers each vertex exactly once and has minimal weight.*