1 The problems

Last lecture we introduced several graph problems. The problems were Single Source Shortest Paths (SSSP), Single Source Reachability (SSR), Connected Components and Connectivity.

We showed that

SSSP in Linear Time solves *SSR in Linear time* which solves *Connected Components in Linear Time* which solves *Connectivity in Linear Time*

We then defined a simple graph search algorithm called Breadth First Search (BFS) which runs in linear time and solves SSSP and thus the rest of the problems as well in linear time.

Today we will present another simple graph search algorithm called Depth First Search (DFS) that runs in linear time and solves SSR (and hence also Connected Components and Connectivity) in linear time. While DFS does not solve SSSP, it can be used to solve other problems. Today we will see two such problems.

The first is *DAG Topological Order*. A directed acyclic graph (DAG) is a directed graph that does not have any cycles. It is known that every DAG $G = (V, E)$ admits an ordering π of its vertices so that if $(u, v) \in E$, then $\pi(u) < \pi(v)$. Such an ordering is called a topological order. Topological orders exist for every DAG, though there may be multiple such orders. (E.g. if you have a graph with two isolated nodes 1 and 2, both $(1, 2)$ and $(2, 1)$ are valid topological orders.)

The *DAG Topological Order* problem takes as input a directed acyclic graph G and returns some topological order of G.

A potentially simpler problem is *Cycle Detection*. Here one is given a directed graph G as input and one needs to compute whether G contains a cycle (then returning YES), or not (then returning NO). DFS will also solve this problem in linear time.

A more difficult problem than *Cycle Detection* is the *Strongly Connected Components* problem. Here one is given a directed graph and one needs to return a partitioning of the vertices into subsets, $C_1, \ldots, C_t$, where every pair of vertices within the same component C_i is connected by a directed path (this is called strongly connected), and if one can go from some node of C_i to some node of C_j, then there is no path between any node of C_j and any node of C_i (that is you cannot make the components bigger by keeping them strongly connected). DFS can be used to solve this problem in linear time as well. We will not show how to solve Cycle Detection or Strongly Connected Components but you can try to think about how one might be able to do it.

2 Depth First Search (DFS)

Let us discuss BFS briefly. Intuitively, BFS first discovered the neighbors of a vertex and then looked at their neighbors etc, taking a "breadth" approach. DFS instead takes a "depth" approach, where starting from a vertex, DFS attempts to follow a path of vertices never seen before, as deeply as possible, then backtracking and looking for other paths. In other words, instead of visiting all neighbors of s first, it visits the first neighbor, then recursively visits its first neighbor etc, building a path, until the path can no longer be continued without visiting a vertex that was on the path already; then one backtracks.

Let us be more formal. First, one sets **globally**:

For all v: $visited[v] \leftarrow false$. These markers are useful so that we don't visit vertices more than once, similar to BFS.

Then, we have a recursive procedure:

DFS(s, G):
    $visited[s] \leftarrow true$
    For all $x \in Adj[s]$:
        If $visited[x] = false$:
            DFS(x, G).

Above, $Adj[s]$ are the out-neighbors of s, in adjacency list format.

The algorithm above when started on s will find all vertices reachable from s and set $visited[v] = true$ for any such v. The proof of this is by induction on the distance from s:

As a base case, when the distance is 0, there is only one vertex at distance 0 from s, namely s, and $visited[s]$ is set to $true$ for sure. Suppose now that the algorithm sets $visited[v]$ to true for every vertex at distance $\leq i$ for some integer i. Now consider a vertex x at distance $i+1$ from s. Consider the vertex v that is right before x on a shortest path from s to x. We know that $d(s, v) = i$, so by the induction hypothesis, the algorithm does eventually set $visited[v]$ to true. When this happens, x is found in $Adj[v]$ since $(v, x) \in E$. At that point the algorithm will check if $visited[x] = false$. Either $visited[x]$ was already set to true (in which case we are done), or the algorithm will set it to true (and hence we are definitely done).

A similar argument shows that if $visited[x]$ is set to true, then s can indeed reach x: By induction on the steps taken by the algorithm, $visited[x]$ was set to true because some edge (v, x) was scanned, where $visited[v] = true$; by induction v must be reachable from s and due to the edge (v, x), so must be x.

Let's add two things to the algorithm now.

First, instead of only marking visited vertices, the algorithm will also keep track of the tree generated by the depth-first traversal. It does so by marking the "parent" of each visited vertex, aka the vertex that DFS visited immediately prior to visiting the child. We do this by storing a value $\pi(v)$ for each vertex v. Initially these are NIL, and when DFS(x) is called from inside DFS(s)[1] due to a scan of edge (s, x), since this is the first time x is seen, we will set $\pi(x) = s$. In other words, $\pi(x)$ is the in-neighbor of x that was used to find x.

DFS(s, G):
    $visited[s] \leftarrow true$
    For all $x \in Adj[s]$:
        If $visited[x] = false$:
            $\pi(x) \leftarrow s$
            DFS(x, G).

The edges $(\pi(v), v)$ scanned during the entirety of DFS(s, G) form a tree rooted at s called the DFS tree from s.

The second thing we will add is a way to traverse the entire graph G. A single call DFS(s, G) finds the vertices reachable from s, but if we wanted to find connected components (as with BFS), we'd need to define DFS(G).

DFS(G):
    For all $v \in V$: $visited[v] \leftarrow false$, $\pi(v) \leftarrow NIL$
    For all $v \in V$:
        If $visited[v] = false$:
            DFS(v, G).

The above algorithm can be modified to find the connected components of an undirected graph by labeling the vertices visited in DFS(v, G) by v.

[1] We will occasionally omit G from the arguments for simplicity, but keep in mind it should be there.

Runtime of DFS. We will now look at the runtime of DFS(G) (and hence also DFS(s, G)). The first for loop runs in $O(1)$ time per node. Excluding the recursive call, everything inside of the second for loop of DFS(G) and the for loop of all DFS(s, G) calls takes $O(1)$ time every time an edge is scanned.

We can thus express the runtime of DFS as $O(\#$ of node visits + # of edge scans). Assume we have a graph with n nodes and m edges. We know that the # of node visits is $\leq$ n, since once a node v is visited for the first time, $visited[v]$ is set to true and v is never visited again. We also know that an edge (u, v) is scanned only when u or v is visited (if (u, v) is directed it is only scanned when u is visited). Since every node is visited at most once, we know that an edge (u, v) is scanned at most twice (or only once for directed graphs). Thus, # of edges scanned is $O(m)$, and the overall runtime of DFS is $O(m + n)$.

We will now show how to adapt DFS to compute the topological order of a directed acyclic graph (DAG).

3 Topological Order of DAGs

A DAG is a directed graph that does not have any directed cycles, i.e. directed paths that begin and end at the same vertex. Undirected acyclic graphs are trees or forests which are necessarily sparse, but DAGs can be very dense: consider $2n$ vertices, n on the left and n on the right and put a directed edge from every left vertex to every right vertex - there will be $\Theta(n^2)$ edges and no directed cycles. Even though DAGs can be quite dense, they are still very special directed graphs. For instance, they admit a *topological order* as we mentioned earlier.

To adapt DFS to solve the topological order problem, we will augment the implementation of DFS to keep some time stamps. These timestamps will turn out not to be necessary for the correctness of the algorithm but we will use them in our arguments.

DFS with timestamps Let us consider an augmented implementation of DFS. The main idea is that we keep two integer values for every vertex v:

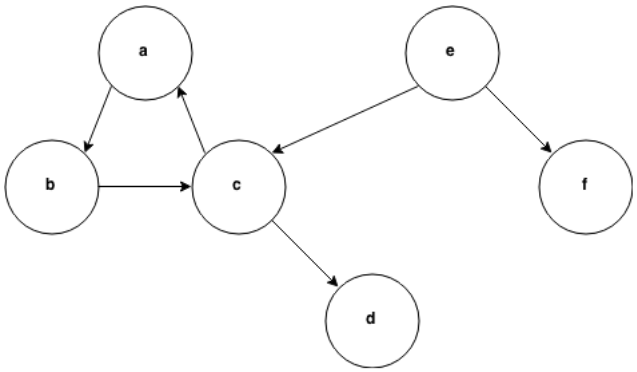
- $d(v)$ is the discovery time of v, i.e. the time when $visited[v]$ was set to true
- $f(v)$ is the finish time of v, i.e. the time after all DFS calls on the children of v (in its DFS tree) are finished.

Time is just an integer. When DFS(G) is called, time starts at 1, and otherwise it is passed on as an argument to DFS($s, G, time$). DFS($s, G, time$) returns an integer which is just the finish time of s plus 1.

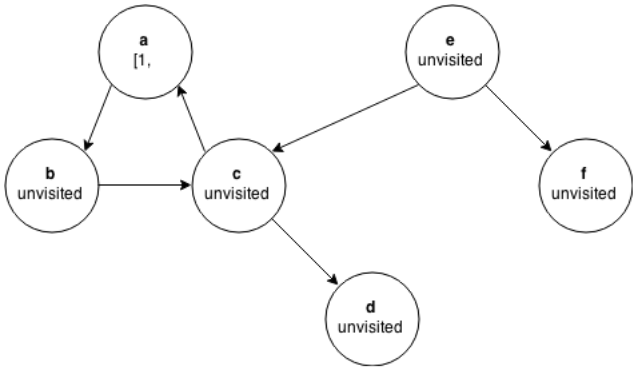
DFS(G):
    $time \leftarrow 1$
    For all $v \in V$:
        $visited[v] \leftarrow false$
        $\pi(v) \leftarrow NIL$
        $d(v), f(v) \leftarrow \infty$
    For all $v \in V$:
        If $visited[v] = false$:
            $time \leftarrow$ DFS$(v, G, time)$.

DFS$(s, G, time)$:
    $visited[s] \leftarrow true$
    $d(s) \leftarrow time$
    $time \leftarrow time + 1$
    For all $x \in Adj[s]$:
        If $visited[x] = false$:
            $\pi(x) \leftarrow s$
            $time \leftarrow$ DFS$(x, G, time)$
    $f(s) \leftarrow time$
    return $(time + 1)$.

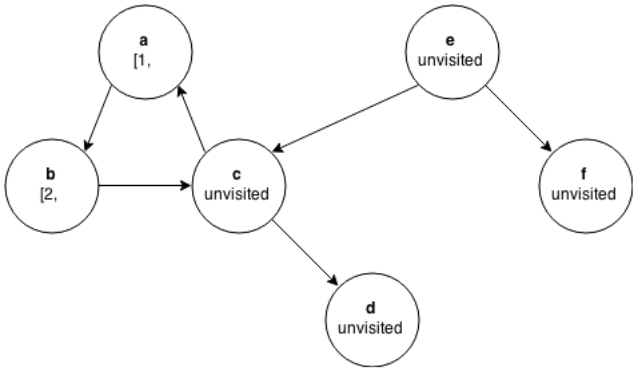
DFS Example. We will now try running DFS (with timestamps) on the example graph below.



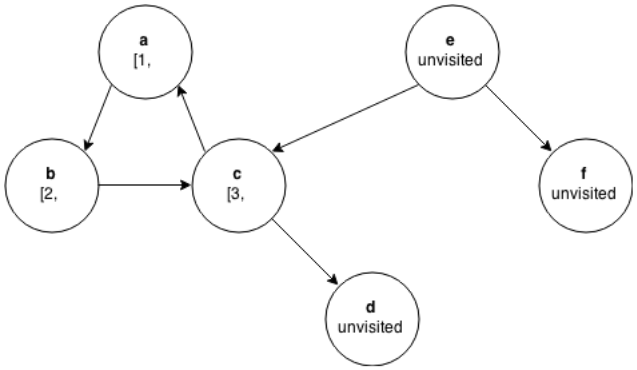
We mark all of the nodes as unvisited and start at an unvisited node, in our case node a.



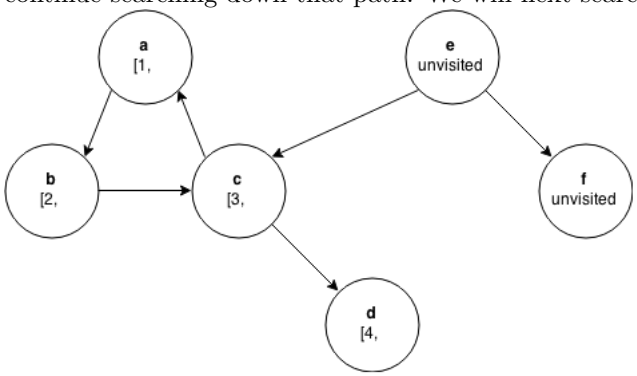
From node a we will visit all of a's children, namely node b.



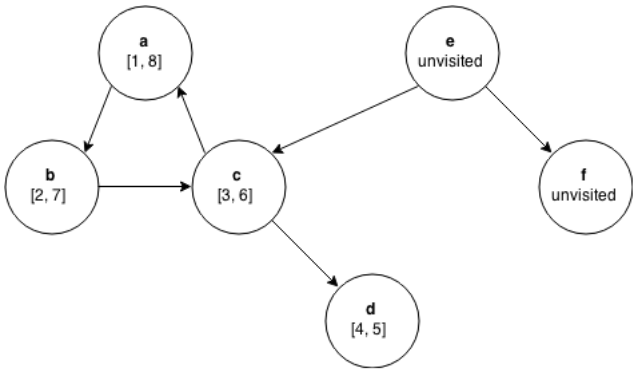
We now visit b's child, node c.



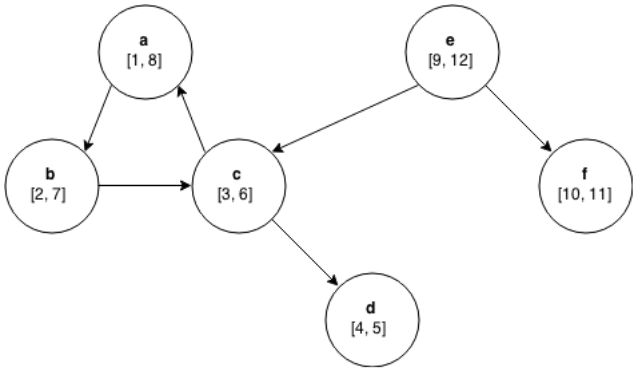
Node c has two children that we must visit. When we try to visit node a we find that node a has already been visited, so we do not continue searching down that path. We will next search c's second child, node d.



Since node d has no children, we return back to its parent node, c, and continue to go back up the path we took, marking nodes with a finish time when we have searched all of their children.



Once we reach our first source node a we find that we have searched all of its children, so we look in the graph to see if there are any unvisited nodes remaining. For our example, we start with a new source node e and run DFS to completion.

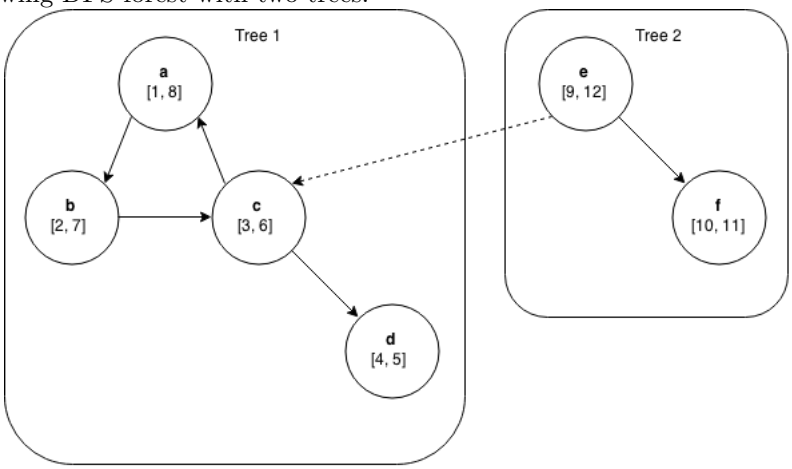


3.1 DFS Trees, Time Interval Properties and Topological Order

DFS trees are defined by the edges created from a parent node to the visited child node.

DFS$(s, G, time)$ computes a tree rooted at s defined by the edges $(p(x), x)$ for x reachable from S (i.e. nodes discovered while the DFS from s is still active).

Running DFS on an entire graph G will create a DFS forest. Our earlier example of DFS on a graph resulted in the following DFS forest with two trees.



Note that even if some node w is reachable from v in a directed graph, v may not be connected to w in the DFS forest. We can see an instance of this above with nodes e and c. Even though node e is connected to node c, the two are not in the same DFS tree because of the source node we started with.

Time Interval Properties. We will now explore some of the properties of time intervals created by running DFS on a graph.

1. If u is a descendant of v in the DFS tree rooted at v, then $[d(u), f(u)] \subset [d(v), f(v)]$.
   This means that for every descendant of a node in a DFS tree we have that the child's time interval is properly contained in the parent's time interval. This makes sense, as we only "finish" a node after we have explored all of its children.

2. If neither u is a descendant of v nor v is a descendant of u, then $[d(u), f(u)] \bigcap [d(v), f(v)] = \emptyset$.
   This means that if two nodes are not connected in a DFS tree, then their intervals don't overlap.

Topological Order. We will show that if we run DFS on a graph G and we sort the vertices in decreasing order by finishing times $f(\cdot)$, we will obtain a topological order:

Claim 1. *If we run DFS(G) on a DAG G, then for all edges (u, v), it must be that $f(v) < f(u)$. Thus a reverse sorted order by f is a topological order.*

Hence, to compute a topological order of G, run DFS(G) and whenever a node is finished, add it to the front of a running list L. We don't even need to keep the time stamps!!

Proof of Claim 1. Consider an edge (u, v). There are two cases:

1. v was visited before u (i.e. $d(v) < d(u)$). Consider DFS$(v, G, time)$. Since we know that DFS$(v, G, time)$ only visits vertices reachable from v, it cannot have visited u. This is because otherwise, G would contain a path from v to u and the edge (u, v), and would thus have a cycle! Hence, u is visited after the entire call to DFS$(v, G, time)$ was finished, and thus $f(u) > d(u) > f(v)$.
2. u was visited before v (i.e. $d(u) < d(v)$). Consider DFS$(u, G, time)$. Since we know that DFS$(u, G, time)$ visits all unvisited vertices reachable from u, and v is unvisited when DFS$(u, G, time)$ is called, DFS$(v, G, time')$ will be called within DFS$(u, G, time)$. Thus, DFS$(v, G, time')$ will finish before DFS$(u, G, time)$, and so $f(v) < f(u)$.

□

The final algorithm without time stamps is as follows:

DFS(G):
  For all $v \in V$: $visited[v] \leftarrow false$, $\pi(v) \leftarrow NIL$
  $L \leftarrow$ new list
  For all $v \in V$:
    If $visited[v] = false$:
      DFS(v, G, L).

DFS(s, G, L):
  $visited[s] \leftarrow true$
  For all $x \in Adj[s]$:
    If $visited[x] = false$:
      $\pi(x) \leftarrow s$
      DFS(x, G, L)
  Insert s into the front of L.

4 (Optional) DFS as a special case of Search*

In the lecture notes of Lecture 9 we presented two very general ways to search a graph. Here we will present DFS as a special case of the second way to search, Search*. Here is its implementation again:

Search*(s, G):
For all v: $visited[v] \leftarrow false$
$D.insert(s)$
While $D.nonempty$:
  $v \leftarrow D.pop$
  If $visited[v] = false$:
    $visited[v] \leftarrow true$
    For all $x \in Adj[v]$:
      $D.insert(x)$.

Return *visited*

Last time we showed that the running time of the algorithm is $O(n + m \cdot t(n))$, where $t(n)$ is the time needed to implement insert and pop on D. In particular, if D is a queue or a stack, the running time is linear, $O(m + n)$.

DFS as a special case of Search*. DFS is Search* implemented with a stack:

DFS(s, G):
For all v: $visited[v] \leftarrow false$
$D.insert(s)$
While $D.nonempty$:
    $v \leftarrow D.pop$
    If $visited[v] = false$:
        $visited[v] \leftarrow true$
        For all $x \in Adj[v]$:
            $D.insert(x)$.
Return *visited*

Because stacks support nonempty, insert and pop in $O(1)$ time, the running time of DFS is linear.

How does our recursive implementation of DFS relate to this Search* implementation? Recursion is typically implemented with a call stack (this is the case in Python). So it is not surprising that the two implementations are equivalent – they perform the same operations (modulo the stack operations), they run in the same asymptotic running time and have the same correctness. Well, this almost exactly true. Actually, to be exactly equivalent, the Search* implementation scans the edges out of a vertex in the reverse order of the recursive implementation. But since the order of the neighbors doesn't really matter for runtime and correctness, we are still fine.