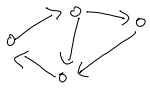
CS 341 - Algorithms, Winter 2016. University of Waterloo

Lecture 7: Directed graphs

We will study directed graphs and see what BFS/DFS can do. We will study a very clever algorithm in identifying all strongly connected components of a directed graph.

Directed Graphs



In a directed graph, each edge has a direction.

When we say a directed edge uv, we mean the edge is pointing from u to v, and u is called the tail and v is called the head of the edge.

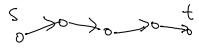
Sometimes, the name arc is used to emphasize that it is a directed edge.

Given a vertex v, $\text{indeg}(v)$ denotes the number of directed edges with v as the head and we call them the incoming edges to v. Similarly, $\text{outdeg}(v)$ denotes the number of directed edges with v as the tail and we call them the outgoing edges.

Directed graphs are useful in modeling asymmetric relations (e.g. web page links, one-way streets, etc).

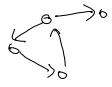
We are interested in studying the connectivity properties of a directed graph.

We say t is <u>reachable</u> from s if there is a directed path from s to t.



A directed graph is called <u>strongly connected</u> if for every pair of vertices $u, v \in V$, u is reachable from v and v is reachable from u.

strongly connected



not strongly connected.

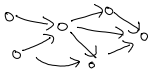
A subset $S \subseteq V$ is called strongly connected if for every pair of vertices $u, v \in S$, u is reachable from v and v is reachable from u.

A subset S is called a <u>strongly connected component</u> if S is a maximally strong connected subset, i.e. S is strongly connected but $S+v$ is not strongly connected for any $v \notin S$.

two strong components

A directed graph is a <u>directed acyclic graph (DAG)</u> if there are no directed cycles.

directed cycle



directed acyclic.

We are interested in the following questions:

- Is a directed graph strongly connected?

- Is a directed graph acyclic ?
- Find all strongly connected components of a directed graph.

It will turn out that all these problems can be solved in $O(n+m)$ time.

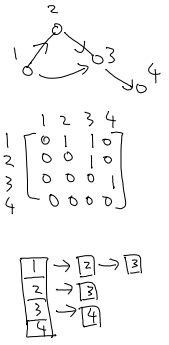
Graph representations

Both adjacency matrix and adjacency list can be defined for directed graphs.

In the adjacency matrix A, if ij is a directed edge, then $A_{ij} = 1$.

In the adjacency list, if ij is an edge, then j is on i's linked list.

As in undirected graphs - we will use the adjacency list representation, since we would like to design $O(n+m)$ algorithms.



Reachability

Before studying these questions, let's first consider a simpler question of checking reachability.

Given a directed graph and a vertex s, both BFS and DFS can be used to find all vertices reachable from s in $O(n+m)$ time.

Both BFS and DFS are defined as in undirected graph, except that we only explore out-neighbors.

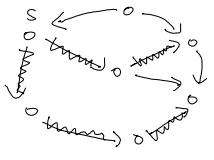
DFS algorithm

Input: a directed graph $G=(V,E)$, a vertex $s \in V$.

Output: all vertices reachable from s

```
(main program)   visited[v] = false  ∀ v∈V .   // global variable array
                 visited[s] = true .  time = 1 .  explore(s)

explore(u)       // recursive function explore.
for each out-neighbor v of u
        start[u] = time .     time ← time + 1
        if visited[v] = false
                visited[v] = true .  explore(v).
```

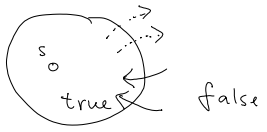


finish[u] = time. time ← time + 1

The time complexity is $O(n+m)$, and t is reachable from s if and only if visited[t] = true.

We leave checking these claims as exercises (same proofs as in undirected graphs).

When we look at all vertices reachable from s, the subset form a "directed cut" with no outgoing edges.



We can define BFS analogously.

An important property that it preserves is that it computes all the shortest path distances from a given vertex s.

We leave it as an exercise to verify this property.

BFS / DFS trees

As in undirected graphs, when a vertex v is first visited, we remember its parent as the vertex u when v is first visited from.

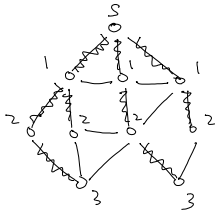
The edges (v, parent[v]) form a tree, and BFS trees and DFS trees are defined.

BFS trees

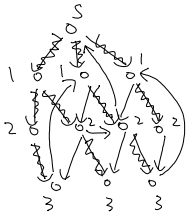
By setting dist[v] = dist[parent[v]] + 1, we compute all shortest path distances from s.

In undirected graphs, for all non-tree edges uv, dist[u] ∈ { dist[v]-1, dist[v], dist[v] +1 }.

In directed graphs, there could be non-tree edges uv, with large difference between dist[u] and dist[v], but in this case we must have dist[u] > dist[v] as they are "back edges".



undirected BFS tree



directed BFS tree

← edges going backward

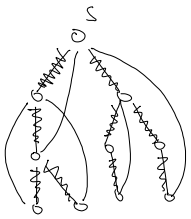
DFS trees

In undirected graphs, all non-tree edges are back edges (going back to ancestors).

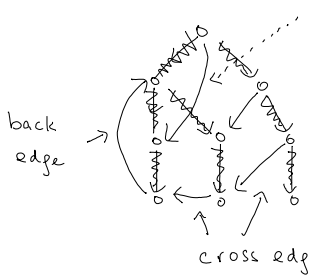
In directed graphs, some non-tree edges could be "cross edges" and "forward edges".

, forward edge

In directed graphs, some non-tree edges could be "cross edges" and "forward edges".



undirected DFS tree



directed DFS tree

We see that directed graphs are a bit more complicated.

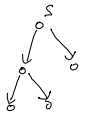
Strongly connected graphs

First, we consider the problem of checking whether a directed graph is strongly connected.

From the definition, we need to check $\Omega(n^2)$ pairs and see if there is a directed path between them.

In undirected graphs, it is enough to pick an arbitrary vertex s, and check whether all vertices are reachable from s.

What would be a corresponding "succint" condition to check in directed graphs?



Just checking that whether all vertices are reachable from s is not enough, e.g.

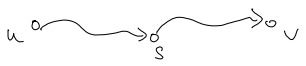
Checking reachability from every vertex would work, but it takes $\Omega(n(n+m))$ time, too slow.

The following observation allows us to reduce the number of pairs to check to $O(n)$.

Observation G is strongly connected if and only if every vertex v is reachable from s and s is reachable from every vertex v.

proof $\Rightarrow$) is trivial, because there is a path between every ordered pair of vertices by definition.

$\Leftarrow$) The condition says that for every vertex v, there is a path from s to v, as well as a path from v to s.



Then, for any u, v, there is a path from u to v, by combining the path from u to s, and the path from s to v. So, there is a path from u to v for any $u, v \in V$. □

We know how to check all vertices are reachable from s in $O(n+m)$ time by BFS or DFS.

How do we check whether s is reachable from all vertices efficiently?

There is a simple trick to do it.

Given G, we reverse the direction of all the edges to obtain G^R.

Then s is reachable from all vertices in G if and only if all vertices in G^R are reachable from s.

This is because a path from s to v in G becomes a path from v to s in G^R.

To check whether all vertices in G^R are reachable from s, we can do it in one BFS/DFS.

To summarize, we have the following algorithm.

Algorithm (strong connectivity)

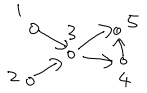
- Check whether all vertices in G are reachable from s.
- reverse the direction of all edges in G to obtain G^R.
- check whether all vertices in G^R are reachable from s.
- If both yes, then say "yes"; otherwise say "no".

The correctness of the algorithm follows from the observation proved above.

The time complexity is $O(n+m)$; we leave it as an exercise that the reversing step can be done in $O(n+m)$ time.

Directed acyclic graphs

Directed acyclic graphs are directed graphs without directed cycles.



They are useful in modeling dependency relations (e.g. course prerequistie).

In such situations, it would be useful to find an ordering of the vertices, so that all the edges go forward. This is called a topological ordering of the vertices (e.g. an ordering to take courses).

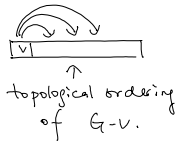
Proposition A directed graph is acyclic if and only if it has a topological ordering.

proof ⇐) Since all the directed edges go forward, there are no directed cycles.

⇒) We will show that any directed acyclic graph has a vertex v of indegree zero.

Then, we can put v as the first vertex in the ordering, and we consider $G-v$.

Since $G-v$ is also acyclic, there is a topological ordering of $G-v$ by induction on the number of vertices, and we are done.

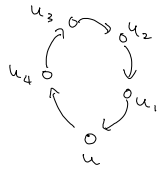


It remains to argue that any directed acyclic graph has a vertex of zero indegree.

Suppose for contradiction that every vertex has indegree at least one.

Then, we start from an arbitrary vertex u, and go to an in-neighbor u_1, and go to an

in-neighbor u_2, and so on.



It is always possible since every vertex has an in-neighbor.

If some in-neighbor repeats, then we find a directed cycle.

But it must repeat at some point, since the graph is finite. □

There are two approaches to find a topological ordering of a directed acyclic graph efficiently.

Approach 1 (sketch) Just follow the above proof.

Keep finding vertices of degree zero and put them in front of the remaining vertices.

We leave it as a problem for you to implement this algorithm in $O(n+m)$ time.

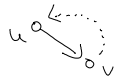
Approach 2 This is perhaps not as intuitive, but it will be useful for us in the next section.

The idea is to do a DFS on the whole graph (meaning that if we got stuck, we start the DFS again on some unvisited vertex, until every vertex is visited, just like what we did in finding all connected components in undirected graphs).

The DFS can be done in any ordering of vertices (in particular, no information about topological ordering or vertices of indegree zero, etc).

Claim If G is directed acyclic, for any directed edge uv, $\text{finish}[v] < \text{finish}[u]$ in any DFS.

Proof There are two cases to consider.



case 1: $\text{start}[v] < \text{start}[u]$. Since the graph is acyclic, u is not reachable from v.

So, u cannot be a descendant of v, and by the parenthesis property, the two intervals $[\text{start}[v], \text{finish}[v]]$ and $[\text{start}[u], \text{finish}[u]]$ must be disjoint.

The only possibility left is that $\text{start}[v] < \text{finish}[v] < \text{start}[u] < \text{finish}[u]$.

case 2: $\text{start}[u] < \text{start}[v]$.

Then, since v is unvisited when u started, v will be a descendent of u.

By the parenthesis property, we have $\text{start}[u] < \text{start}[v] < \text{finish}[v] < \text{finish}[u]$. □

With the claim in place, we have the following algorithm for finding a topological ordering.

Algorithm (topological ordering)

- run DFS on the whole graph.

- output the ordering with decreasing finishing time.
- check whether it is a topological ordering.

The correctness of the algorithm follows from the claim.

The algorithm can be implemented in $O(n+m)$ time. (Don't do sorting! Just put a vertex in a queue when it is finished, by adding one line in the code.)

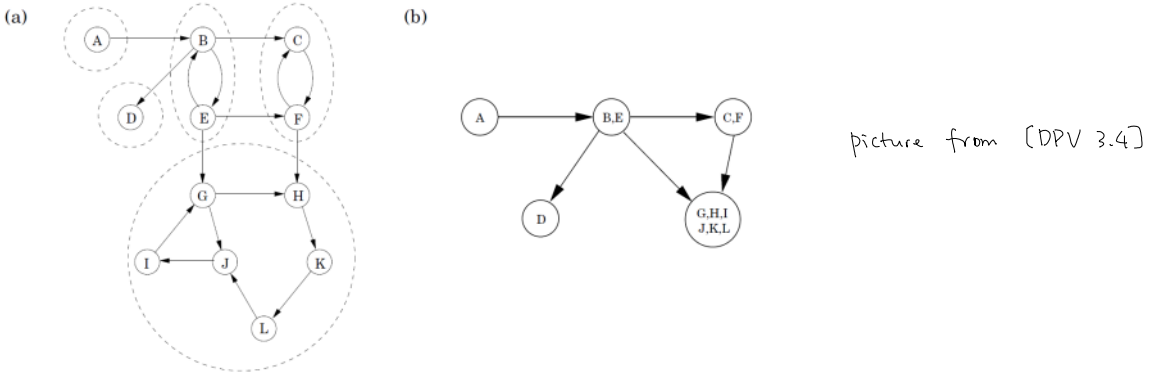
Strongly connected components

Finally, we consider the (more difficult) problem of finding all strongly connected components.

We will combine and extend the previous ideas to obtain an $O(n+m)$ time algorithm.

First, we get a good idea about how a general directed graph looks like.

A preliminary observation: two strongly connected components are vertex disjoint. If two components C_1 and C_2 share a vertex, then $C_1 \cup C_2$ is also strongly connected, contradicting maximality of C_1, C_2.



picture from [DPV 3.4]

In the picture, when every strongly connected component is thought of as a single vertex, then the resulting directed graph is acyclic. We leave it as an exercise to prove it.

Idea 1: Suppose we start a DFS in a "sink component" C (a component with no outgoing edges), then we can identify the component C.

It is because every vertex in C is reachable from the starting vertex, but no vertices outside.

So, just read off the vertices with visited[v] = true will recover C.

This suggests the following strategy.

- Find a vertex v in a sink component C.

- Do a DFS or BFS to identify C.
- Remove C from the graph and repeat.

Now, the question is how to find a vertex in a sink component efficiently?

It doesn't look easy. Can you think of how?

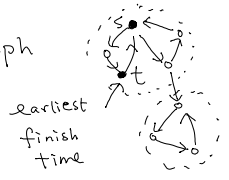
Idea 2 (X): In the picture above, when we think of each component as a (big) vertex, the graph is a directed acyclic graph.

From the previous section about directed acyclic graphs, we know that if we do a DFS on the whole graph, the node with the earliest finishing time is a sink.

This suggests the following strategy.

- Run DFS on the whole graph and obtain an ordering in increasing finishing time.
- Use this ordering in the previous strategy (i.e. start from the first vertex in this ordering, remove all vertices reachable from it and say it is a strong component, then go to the next un-removed vertex in the ordering, and repeat).

This is a very nice strategy, but unfortunately it doesn't work.

For example, consider the graph 

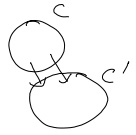
, if we start the DFS at s, then the node t has the earliest finish time, but it is not in a sink component.

Idea 3: Not all is lost. The DFS ordering will still give us some useful information about the topological ordering of the components.

In particular, although we couldn't say that the vertex with smallest finish time is in a sink component, we could prove that the vertex with largest finish time is in a source component.

The proof is similar to the claim in topological ordering.

Claim: If C and C′ are strong components and there are edges from C to C′, then the largest finish time in C is bigger than the largest finish time in C′.



proof: There are two cases to consider.

Case 1: If the first vertex v visited in $C \cup C'$ is in C′, since vertices in C are not reachable from v but all vertices in C′ are reachable from v, at the time when

v is finished, all vertices in C' are finished, while all vertices in C have not started.

Case 2: If the first vertex v visited in $C \cup C'$ is in C, since vertices in $C \cup C'$ are reachable from v, all vertices in $C \cup C'$ will be finished before v. and thus $v \in C$ will have the largest finish time in $C \cup C'$. □

With the claim, we know that if we first do a DFS, and then we do a DFS again using a decreasing order of finish time, then we will visit ancestor components before we visit descendent components.

But this is not good, since we really want to start in a sink component and cut it out first.

Idea 4: Reverse the graph! (Like what we did in checking strong connectivity.)

The first observation is that the strong components don't change when we reverse the direction of all the edges.

Very important for us, source components become sink and vice versa!

Therefore, the ordering we have in G following the topological ordering of the components (from sources to sinks) becomes an ordering in G' following the reverse topological ordering of the components (from sinks to sources).

Now, we can just follow this ordering to do the DFS to cut out sink components one at a time as in idea 1 and idea 2.

Algorithm (strong components)

① Run DFS on the whole graph using an arbitrary ordering of vertices.

② Obtain a new ordering by decreasing order of finish time in previous execution of DFS.

③ Reverse the graph G to obtain the graph G^R.

④ Follow the new ordering to explore the graph G^R to cut out the components one at a time.

To be more precise, we expand step ④ in more details.

- Let i be the vertex of i-th largest finishing time after step ②.
- Let $c = 1$. It is a variable counting the number of strong components.

- for $1 \le i \le n$ do

    if visited[i] = false

        DFS(G^R, i)

        mark all the vertices reachable from i in this iteration to be in component c.

        $c \leftarrow c+1$.

The proof of correctness follows from our long discussion.

We leave it as an exercise to check that all the steps can be implemented in $O(n+m)$ time.

This algorithm is very clever and it may take more time to understand fully.

Reference: [DPV 3.3-3.4]