CS 341 - Algorithms, Spring 2021. University of Waterloo

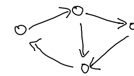
Lecture 7: Directed Graphs

We study directed graphs and what BFS/DFS can do in directed graphs.

A highlight is a very clever algorithm to identify all strongly connected components in linear time.

Directed Graphs

In a directed graph, each edge has a direction.



When we say uv is a directed edge, we mean the edge is pointing from u to v, $\overset{u}{\circ} \to \overset{v}{\circ}$

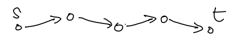
and u is called the <u>tail</u> and v is called the <u>head</u> of the edge.

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

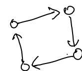
Directed graphs are useful in modeling asymmetric relations (e.g. web page links, one-way streets, etc).

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.

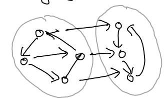


strongly connected

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.

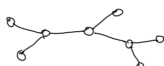
A subset $S \subseteq V$ is called a <u>strongly connected component</u> if S is a maximally strongly connected subset, i.e. S is strongly connected but $S+v$ is not strongly connected for any $v \notin S$.



2 strongly connected components

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

Note that a directed acyclic graph, unlike its undirected counterpart, could have many edges.



undirected acyclic graph

directed acyclic graph

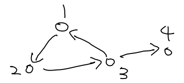
We are interested in designing algorithms to answer the following basic questions:

1. Is a given graph strongly connected?
2. Is a given graph directed acyclic?
3. Find all strongly connected components of a given directed graph.

As in undirected graphs, it will turn out that there are $O(n+m)$-time algorithms to solve

these problems, but they are not as easy as the algorithms for undirected graphs.

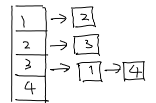
Graph Representations



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

In the adjacency matrix A, if ij is a directed edge, then $A_{ij} = 1$; otherwise $A_{ij} = 0$.

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



As in undirected graphs, we will only use adjacency list in this part of the course, as only this allows us to design $O(n+m)$-time algorithms.

Reachability

Before studying the above questions, we first study a simpler question of checking reachability.

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

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

DFS Algorithm

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

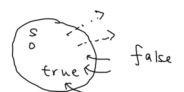
Output: All vertices reachable from s.

```
[Main program]    visited[v] = false ∀ v∈V.   time = 1.   visited[s] = true.   explore(s).

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

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

When we look at all vertices reachable from s, the subset form a "directed cut" with no outgoing edges (but could have incoming edges into the subset).



We leave as exercises in checking these claims. The proofs are the same as in undirected graphs.

We can define BFS for directed graphs analogously, by only exploring out-neighbors.

An important property of BFS is that it computes the shortest path distances from s to all other vertices. We leave it as an important exercise to check this claim.

BFS / DFS Trees

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

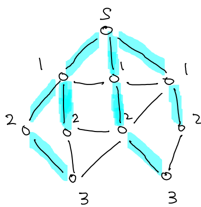
The edges (v, parent[v]) form a tree, and both BFS trees and DFS trees are defined in this way.

BFS trees

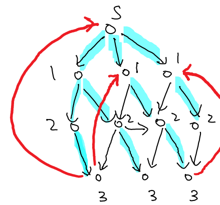
By setting dist[v] = dist[parent[v]] + 1, we compute all shortest path distances from s.

In undirected graphs, for all non-tree edges uv, $dist[v] - 1 \le dist[u] \le dist[v] + 1$.

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



undirected BFS tree

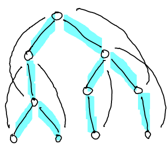


directed BFS tree

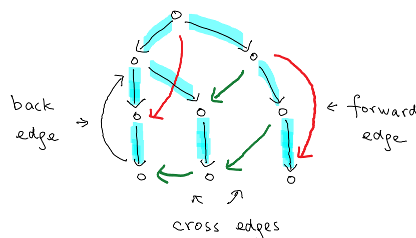
DFS trees

In undirected graphs, all non-tree edges are back edges (see L06).

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



undirected DFS tree



directed DFS tree

Structured in directed graphs are more complicated.

Strongly Connected Graphs

We are ready to consider the first 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. So we just check reachability for O(n) pairs.

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

It is easy to find examples for which just checking reachability from s is not enough.

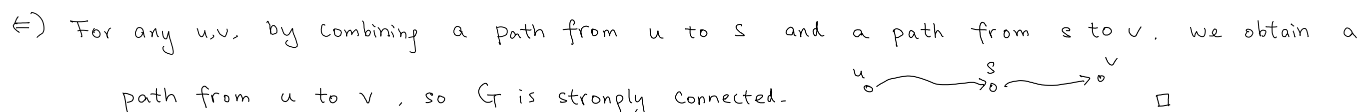
Checking reachability from every vertex would work, but it would take $\Omega(n(n+m))$ time, too slow.

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

<u>Observation</u> G is strongly connected if and only if every vertex v is reachable from s and s is reachable from every vertex v, where s is an arbitrary vertex.

<u>Proof</u> $\Rightarrow$) is trivial by the definition of a strongly connected graph.

$\Leftarrow$) For any u, v, by combining a path from u to s and a path from s to v, we obtain a path from u to v, so G is strongly connected. □



We know how to check whether 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, by reversing the direction of the edges.

<u>Claim</u> Given G, we reverse the direction of all the edges to obtain G^R.

There is a directed path from v to s in G iff there is a directed path from s to v in G^R.

So, s is reachable from all vertices in G iff every vertex is reachable from s in G^R.

With this claim, we can check whether s is reachable from every vertex in G by doing one BFS/DFS in G^R from s.

To summarize, we have the following algorithm.

Algorithm (strong connectivity)

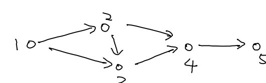
1. Check whether all vertices in G are reachable from s by one BFS/DFS.
2. Reverse the direction of all the edges in G to obtain G^R.
3. Check whether all vertices in G^R are reachable from s by one BFS/DFS.
4. If both yes, return "strongly connected"; otherwise return "not strongly connected".

The correctness of the algorithm follows from the observation and the claim above.

The time complexity is $O(n+m)$ time.

We leave it as a simple exercise to construct G^R in linear time.

<u>Directed Acyclic Graphs</u>



Directed acyclic graphs are directed graphs without directed cycles.

They are useful in modeling dependency relations (e.g. course prerequisties. software installation).

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 <u>topdopical ordering</u> of the vertices (e.g. an ordering to take the courses).

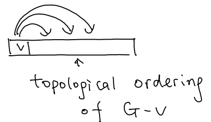
<u>Proposition</u> A directed graph is acyclic if and only if there is a topological ordering of the vertices.

<u>Proof</u> $\Leftarrow$) Since all the directed edges go forward, there are no directed cycles.

$\Rightarrow$) We will prove 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 $G-v$.

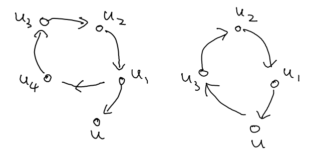
Since $G-v$ is also acyclic, there is a topological ordering of $G-v$ by induction on the number of vertices, and we are done.



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

Suppose by 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 of u, and then go to an in-neighbor u_2 of u_1. and so on.



It is always possible since every vertex has in-degree at least one.

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

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

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

<u>Approach 1</u> (Sketch) Just follow the procedure in the above proof.

That is, keep finding a vertex of indegree zero in the remaining graph and put it in the beginning of the ordering.

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

<u>Approach 2</u> This is perhaps less intuitive. but the ideas will be useful in the next section as well.

The idea is to do a DFS on the whole graph (i.e. start a DFS on an arbitrary vertex, but if not all vertices are visited, start a DFS on an unvisited vertex, and so on, until all vertices are visited, just like what we would do in finding all connected components of an undirected graph).

Note that this DFS can be done in any ordering of vertices. In particular, we don't need to start at a vertex of indegree zero nor do we need any information about a topological ordering.

In the following proof, we use that the parenthesis property of starting and finishing time holds

for directed graphs as well. Please check.

Lemma If G is directed acyclic, then for any directed edge uv, $finish[v] < finish[u]$ for any DFS.

Proof We consider two cases.

Case 1: $start[v] < start[u]$.

Since the graph is acyclic, u is not reachable from v.

So, u cannot be a descendant of v.

By the parenthesis property, the intervals $[start[v], finish[v]]$ and $[start[u], finish[u]]$ must be disjoint.

The only possibility left is $start[v] < finish[v] < start[u] < finish[u]$, proving the lemma in this case.

Case 2: $start[u] < start[v]$.

Then, since v is unvisited when is started and uv is an edge, v will be a descendant of u in the DFS tree. This is the same as the argument used in the back edge property in L06.

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

Using the lemma, we have the following simple algorithm for computing a topological ordering.

Algorithm (Topological Ordering / Directed Acyclic Graphs)

1. Run DFS on the whole graph.
2. Output the ordering with decreasing finishing time.
3. Check if it is a topological ordering. If not, return "not acyclic".

Correctness: The lemma proves that if the graph is acyclic, then all edges go forward in this ordering. On the other hand, if the graph is not acyclic, then there is no topological ordering by the proposition.

Time Complexity: The algorithm can be implemented in $O(n+m)$ time.

Note that we don't need to do sorting for the second step.

Just put a vertex in a queue when it is finished, by adding one line in the code.

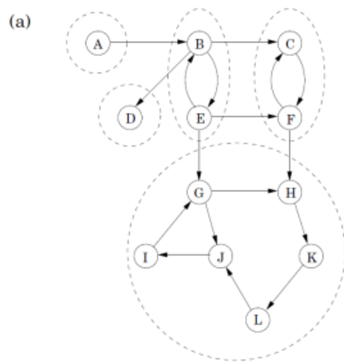
Strongly Connected Components (SCC)

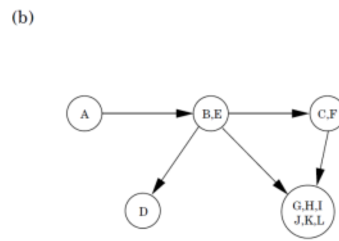
Finally, we consider the more difficult problem of finding all strongly connected components. We will combine and extend the previous ideas to obtain an $O(n+m)$ time algorithm.

First, let's get a good idea about how a general directed graph looks like.

Observation: Two strongly connected components are vertex disjoint. If two strongly connected components

Observation: Two strongly connected components are vertex disjoint. If two strongly connected 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.





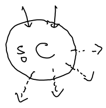
picture from [DPV 3.4]

In the picture, when every strongly connected component is "contracted" into a single vertex, then the resulting directed graph is acyclic.

This is true in general. We leave the proof as an exercise.

So, a general directed graph is a directed acyclic graph on its strongly connected components.

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



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

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

This suggests the following strategy.

1. Find a vertex v in a sink component C.
2. Do a DFS/BFS to identify C.
3. Remove C from the graph and repeat.

So, 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: Do "topological sort".

As discussed above, if each strong component is "contracted" to a single vertex, the resulting graph is acyclic.

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.

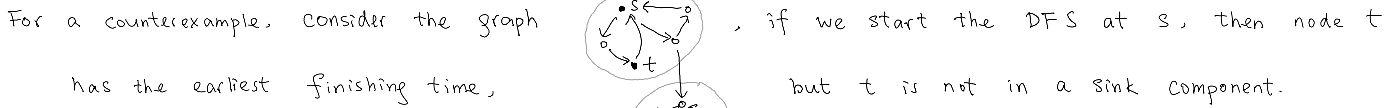
This suggests the following strategy.

1. Run DFS on the whole graph and obtain an ordering in increasing finishing time.

2. Use this ordering in the previous strategy in idea 1 to take out one sink component at a time.

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

For a counterexample, consider the graph



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

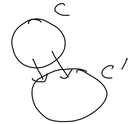
Idea 3: The natural strategy doesn't work, but a modification of it may still work.

The observation is that the DFS ordering still gives us useful information about a topological ordering of components.

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

The proof of the following lemma is similar to the proof of the lemma in topological ordering.

Lemma If C and C' are strong components and there are edges from C to C', then the largest finishing time in C is bigger than the largest finishing time in C'.



Proof Again, we consider two cases.

Case 1: The first vertex v visited in $C \cup C'$ is in C'.

Note that vertices in C are not reachable from v but all vertices in C' are reachable from v.

By the time when v is finished, all vertices in C' are finished, while all vertices in C haven't started.

Case 2: The first vertex v visited in $C \cup C'$ is in C.

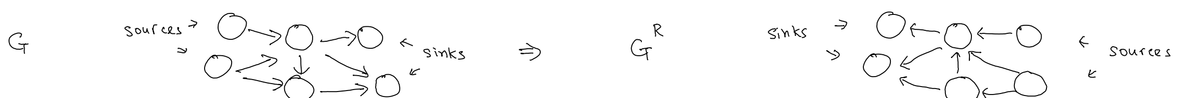
Since vertices in $C \cup C'$ are reachable from v, all vertices in $C \cup C'$ will be finished before v is finished, and so $v \in C$ will have the largest finishing time in $C \cup C'$. □

With this lemma, we know that if we first do a DFS and order the vertices in decreasing order finishing time, and then do a DFS again using this ordering, then we will visit "ancestor components" before we visit "descendant components".

But this is not what we want, as we want to start in a sink component and cut it out first.

Idea 4: Reverse the graph so that sources become sinks!

First, observe that the strong components in G are the same as the strong components in G^R.



Very important for us, source components in G become sink components in G^R and vice versa.

Therefore, the ordering we have in G following a topological ordering of the components from

sources to sinks becomes an ordering in G^R following a topological ordering of the components from sinks to sources.

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

Finally, we can summarize the algorithm.

Algorithm (Strong Components)

1. Run DFS on the whole graph G using an arbitrary ordering of vertices.
2. Order the vertices in decreasing order of finishing times obtained in step 1.
3. Reverse the graph G to obtain the graph G^R
4. Follow the ordering in step 2 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.

(i) Let i be the vertex of i-th largest finishing time in step 2.

(ii) Let $c = 1$. // It is a variable counting the number of strong connected components.

(iii) For $1 \le i \le n$ do

    If visited[i] = false
        DFS(G^R, i)
        Mark all the vertices reachable from i in G^R in this iteration to be in component c.
        $c \leftarrow c+1$

The proof of correctness follows from our long discussion.

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

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

Reference : [DPV 3.3-3.4]