Week 7
Depth-first search

1. Generalising BFS
   - Restarts
   - Digraphs
   - Non-unit arc lengths
2. Depth-first search
3. Analysing DFS
   - Background: lemmas, corollaries, theorems
4. Dags and topological sorting
5. Examples for topological sorting
6. Finding cycles

General remarks

- We consider the second main graph-search algorithm, **depth-first search** (DFS).
- And we consider one application of DFS, **topological sorting** of graphs.

Reading from CLRS for week 5

- Chapter 22, Sections 22.3, 22.4.

Restarting BFS

Now what if we need to reach all vertices?

Then we need a spanning **forest**!

This means we need to

restart BFS on a yet unreached vertex,
maintaining the old π and d values,
repeating this until all vertices have been reached.

Running BFS on the previous example $G := (\{1, 2, 3, 4\}, \{\{1, 2\}, \{3, 4\}\})$, first with $s = 1$, and then restarting with $s = 3$, yields

- $\pi[1] = \mathrm{NIL}$, $\pi[2] = 1$, $\pi[3] = \mathrm{NIL}$, $\pi[4] = 3$.
- $d[1] = 0$, $d[2] = 1$, $d[3] = 0$, $d[4] = 1$.

Note that now

$d[v]$ is the distance of v to **its** root.

How many rounds?

For a graph G, how often do we need to restart BFS to cover all vertices?

Let $m \geq 0$ be the number of connected components of G. Then we need to restart BFS exactly m times.

Here it is essential that G is undirected; the directed case, considered in the next subsection, is more complicated.

How many rounds?

For a graph G, how often do we need to restart BFS to cover all vertices?

Let $m \geq 0$ be the number of connected components of G. Then we need to restart BFS exactly m times.

Here it is essential that G is undirected; the directed case, considered in the next subsection, is more complicated.

Running BFS on directed graphs

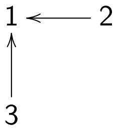
We can run BFS also on a digraph G, with start-vertex s:

1. For digraphs we have **directed spanning trees/forests**.
2. Only the vertices reachable from s *following the directions of the arcs* are in that directed tree (directed from s towards the leaves).
3. Still the paths in the directed tree, from s to any other vertex, are *shortest possible* (given that we obey the given directions of the arcs).
4. So BFS computes *distances* (length of shortest paths) between vertices also for digraphs.

For a digraph there is a much higher chance,
that we might need to restart
if we wish to have a directed *spanning* forest
(*all* vertices included).

Restarts needed to have all vertices

Consider the simple digraph $G := (\{1,2,3\}, \{(2,1),(3,1)\})$



1. Running $\mathrm{BFS}(G,1)$, we obtain the directed tree $(\{1\}, \emptyset)$ (no arc). Not covered are vertices $2, 3$ — *this is fine*, since from vertex 1 you can't reach them.
2. Running $\mathrm{BFS}(G,2)$, we obtain the directed tree $(\{1,2\}, \{(2,1)\})$ (one arc). Not covered is vertex 3 — again *fine*, since from vertex 2 you can't reach vertex 3.

Note that above we run BFS only once, for the given start vertex (no restart).

Restarts needed to have all vertices (cont.)

In order to cover all vertices,

obtaining a **directed spanning forest**,

one needs to run BFS at least two times on the previous example, while keeping d and π (and if the first time you start it with $s = 1$, then you need to run it three times).

We have three possible directed spanning forests here for the previous digraph (containing in the first case three directed trees, and two in the two other cases):

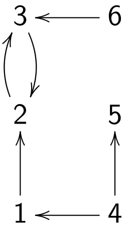
1. $(\{1,2,3\}, \emptyset)$.
2. $(\{1,2,3\}, \{(2,1)\})$.
3. $(\{1,2,3\}, \{(3,1)\})$.

Note that due to keeping the d-array **between the different runs here**, there is no overlap.

On the nature of the restarts

Consider the digraph

$G :=$



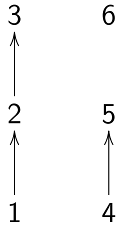
$$= (\{1,2,3,4,5,6\}, \{(1,2),(2,3),(3,2),(6,3),(4,1),(4,5)\}).$$

What happens when we run BFS with restarts, following numerical order of the vertices (when there is a choice)?

The main output of BFS with restarts (and DFS) is *always*
a spanning forest (directed or undirected),
plus additional information
(the d-array for BFS, later the d, f-arrays for DFS).

The directed spanning forest T of G obtained is

$T =$



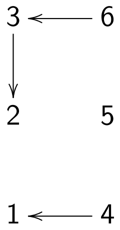
$$= (\{1,2,3,4,5,6\}, \{(1,2),(2,3),(4,5)\}).$$

1. The directed forest consists of three directed rooted trees.
2. For each of the three directed rooted trees, the tree is a shortest-path tree from the root (1 resp. 4 resp. 6) to those vertices *contained* in that tree — but these are not all vertices reachable in G from the root, but vertices previously discovered have been discarded.

Now let's consider *inverse* numerical order.

The directed spanning forest T' of G now is

$$T' =$$



$$= (\{1, 2, 3, 4, 5, 6\}, \{(4, 1), (3, 2), (6, 3)\}).$$

1. The directed forest again consists of three directed rooted trees (this is just coincidence).
2. For each of the three rooted trees, the tree is a shortest-path tree from the root (6 resp. 5 resp. 4) to those vertices *contained* in that tree (this is *not* a coincidence).

That the trees in the directed spanning forest have no vertices in common is ESSENTIAL — BFS must run in **linear time**.

Background: Arcs of different lengths

The edges of (di-)graphs have (implicitly) a length of one unit.

- If arbitrary non-negative lengths are allowed, then we have to generalise BFS to *Dijkstra's algorithm*.
- This generalisation must keep the essential properties, that the distances encountered are the final ones and are *non-decreasing*.
- But now not all edges have unit-length, and thus instead of a simple queue we need to employ a *priority queue*.
- A *priority queue* returns the vertex in it with the *smallest* value (distance).

Depth-first search

Depth-first search (DFS) is another simple but very important technique for searching a graph.

Such a search constructs a spanning forest for the graph, called the **depth-first forest**, composed of several **depth-first trees**, which are rooted spanning trees of the connected components.

DFS recursively visits the next unvisited vertex, thus extending the current path as far as possible; when the search gets stuck in a “corner” it backtracks up along the path until a new avenue presents itself.

DFS computes the **parent** $\pi[u]$ of each vertex u in the depth-first tree (with the parent of initial vertices being NIL), as well as its **discovery time** $d[u]$ (when the vertex is first encountered, initialised to ∞) and its **finishing time** $f[u]$ (when the search has finished visiting its adjacent vertices).

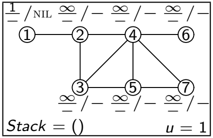
The algorithm

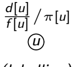
```
DFS($G$)
for each $u \in V(G)$
    $d[u] = \infty$
$time = 0$
for each $u \in V(G)$
    if $d[u] = \infty$
        $\pi[u] = \text{NIL}$
        DFS-VISIT($u$)
```

```
DFS-VISIT($u$)
$time = time + 1$
$d[u] = time$
for each $v \in Adj[u]$
    if $d[v] = \infty$
        $\pi[v] = u$
        DFS-VISIT($v$)
$time = time + 1$
$f[u] = time$
```

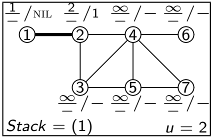
Analysis: DFS-VISIT(u) is invoked exactly once for each vertex, during which we scan its adjacency list once. Hence DFS, like BFS, runs in time $\Theta(V + E)$.

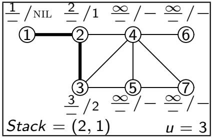
DFS illustrated

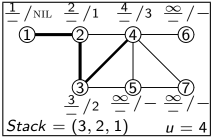


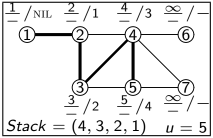


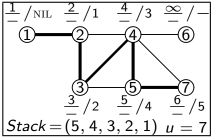
(labelling)

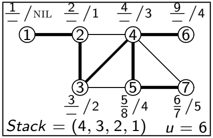


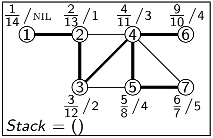












Running DFS on directed graphs

Again (as with BFS), we can run DFS on a digraph G.

- Again, no longer do we obtain spanning trees of the connected components of the start vertex.
- But we obtain a **directed spanning tree** with exactly all vertices reachable from the root (when following the directions of the arcs).

Different from BFS, the root (or start vertex) normally does not play a prominent role for DFS:

1. Thus we did not provide the form of DFS with a start vertex as input.
2. But we provided the forest-version, which tries all vertices (in the given order) as start vertices.
3. This will always cover all vertices, via a **directed spanning forest**.

What are the times good for?

(Directed) DFS trees do not contain shortest paths — to that end their way of exploring a graph is too “adventuresomely” (while BFS is very “cautious”).

Nevertheless, the information gained through the computation of discovery and finish times is very valuable for many tasks. We consider the example of “scheduling” later.

In order to understand this application, we have first to gain a better understanding of the meaning of discovery and finishing time.

Nesting of times

Consider a digraph G with $n := |V(G)|$ the number of vertices. Basic properties of discovery and finishing times are, for all $v, w \in V(G)$:

1. $1 \leq d[v] < f[v] \leq 2n$.
2. $d[v] \neq f[w]$.
3. If $v \neq w$, then $d[v] \neq d[w]$ and $f[v] \neq f[w]$.

So the $d[v]$ and $f[v]$ for $v \in V(G)$ are all different, and together yield exactly the numbers $1, \dots, 2n$.

Nesting of times (cont.)

Discovery and finishing times are produced by recursive calls, and from that follows the basic **Nesting Lemma** for vertices $v, w \in V(G)$, $v \neq w$, which says that we have exactly one of the following four cases ("sequential" versus "nested"):

S1 $d[v] < f[v] < d[w] < f[w]$

S2 $d[w] < f[w] < d[v] < f[v]$

N1 $d[v] < d[w] < f[w] < f[v]$

N2 $d[w] < d[v] < f[v] < f[w]$.

Impossible are:

I1 $d[v] < d[w] < f[v] < f[w]$

I2 $d[w] < d[v] < f[w] < f[v]$.

Analysing reachability

The main topic is **reachability**: for vertices $v, w \in V(G)$ and a digraph G, let

$$v \rightsquigarrow_G w$$

denote that w is reachable from v in G, that is, there is a path from v to w in G. Basic properties of the reachability relation are, for each $u, v, w \in V(G)$:

1. $v \rightsquigarrow_G v$
2. $u \rightsquigarrow_G v$ and $v \rightsquigarrow_G w$ implies $u \rightsquigarrow_G w$.

DFS (as BFS) does not determine general reachability in G (between all pairs of vertices – this can't be done in linear time), but establishes its point of view via the constructed *directed spanning forest* T:

- Recall that $V(T) = V(G)$, $E(T) \subseteq E(G)$, and T is a vertex-disjoint union of rooted directed trees (directed from the root to the leaves).
- So $v \rightsquigarrow_T w$ implies $v \rightsquigarrow_G w$, but not vice versa.

Reachability in the spanning forest

We want to understand what from discovery and finishing times can be deduced about reachability. For a first reading, only the statement of Corollary 4 (the final result) is needed.

The computed spanning forest is naturally closely related to the times:

Lemma 1 (Reachability Lemma)

Consider a digraph G, and the data (T, d, f) computed by running DFS on G (the spanning forest, plus discovery and finishing times). Then for all $v, w \in V(G)$ holds

$$v \rightsquigarrow_T w \text{ iff } d[v] \leq d[w] < f[v].$$

Corollary 2

If $d[v] \leq d[w] < f[v]$, then $v \rightsquigarrow_G w$.

What can be deduced from the discovery time

The following theorem answers, given $d[v]$ and $f[v]$, what can be deduced about $v \rightsquigarrow_G w$ from $d[w]$:

Theorem 3

Consider a digraph G and the data (T, d, f). Consider $v, w \in V(G)$, $v \neq w$.

A *If $f[v] < d[w]$, then $(v, w) \notin E(G)$, but $v \rightsquigarrow_G w$ (with distance ≥ 2) OR $v \not\rightsquigarrow_G w$ is possible.*

B *If $d[v] < d[w] < f[v]$, then $v \rightsquigarrow_G w$.*

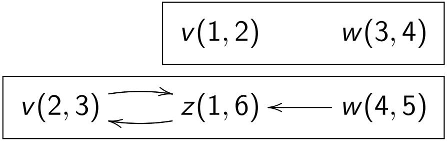
C *If $d[w] < d[v]$, then $v \rightsquigarrow_G w$ OR $v \not\rightsquigarrow_G w$ is possible.*

Proof:

For Case A ("w completely after"), by definition of DFS, if $(v, w) \in E(G)$, then $d[w] < f[v]$.

What can be deduced from the discovery time (cont.)

Examples for the two cases are:



: $v \not\rightsquigarrow_G w$

: $v \rightsquigarrow_G w$

Now consider Case B, that is, $d[v] < d[w] < f[v]$ ("w inbetween"). By Corollary 2 we have $v \rightsquigarrow_T w$, and thus $v \rightsquigarrow_G w$. $\surd$

What can be deduced from the discovery time (cont.)

Finally for Case C, i.e., $d[w] < d[v]$ (“w discovered before”).

First the case $f[w] < d[v]$:

$$\boxed{v(3,4) \qquad\qquad w(1,2)} \quad : \quad v \not\rightsquigarrow_G w$$

$$\boxed{z(1,4) \longrightarrow w(2,3) \longleftarrow v(5,6)} \quad : \quad v \rightsquigarrow_G w$$

Second the case $f[w] > d[v]$. Note that here we have $f[v] < f[w]$ (nesting!) and $w \rightsquigarrow_G v$:

$$\boxed{v(2,3) \longleftarrow w(1,4)} \quad : \quad v \not\rightsquigarrow_G w$$

$$\boxed{v(2,3) \rightleftarrows w(1,4)} \quad : \quad v \rightsquigarrow_G w$$

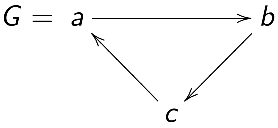
QED

Directed acyclic graphs

An important application of digraphs G is **scheduling**:

- The vertices are the jobs (actions) to be scheduled.
- An arc from vertex u to vertex v means a dependency, that is, action u must be performed **before** action v.

Now consider the situation where we have three jobs a, b, c and the following dependency digraph:



Clearly this can not be scheduled!

In general we require G to by **acyclic**, that is, G must not contain a directed cycle.

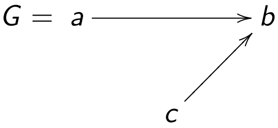
A **directed acyclic graph** is also called a **dag**.

Topological sorting

Given a dag G modelling a scheduling task, a basic task is to find a linear ordering of the vertices ("actions") such that all dependencies are respected.

- This is modelled by the notion of **topological sorting**.
- A topological sort of a dag is a linear ordering of its vertices, such that for every arc (u, v), u appears before v in the ordering.

For example consider



The two possible topological sortings of G are a, c, b and c, a, b.

Finishing times in DAGs

In the case of a dag, we can do better then Theorem 3:

Lemma 4

After calling DFS *on a dag, for every arc* (v, w) *holds* $f[v] > f[w]$ *(finishing times must go down on arcs).*

That we need a dag here, can be seen by considering a digraph which is just a cycle — one of the arcs (v, w) must have $f[v] < f[w]$ (otherwise we would have an infinite descent).

Proof

There are two cases regarding the discovery times of v and w (recall we have an arc $v \to w$, and we want to show $f[v] > f[w]$):

1. $d[v] < d[w]$:
   By Theorem 3, Part A, we have $f[w] < f[v]$. $\surd$
2. $d[v] > d[w]$:
   - $f[v] > f[w]$: done.
   - $f[v] < f[w]$: so $d[w] < d[v] < f[v] < f[w]$.
     By Theorem 3, Part B, we know $w \rightsquigarrow_G v$, and this together with the arc from v to w establishes a cycle in G, contradicting that G is a dag.

QED

Topological sorting via DFS

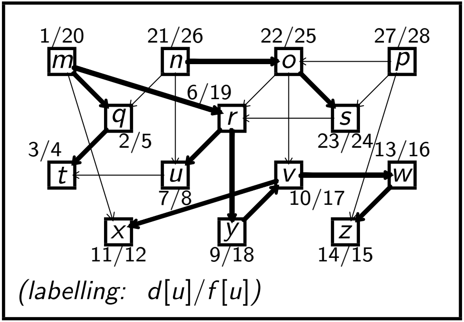
Corollary 5

To topologically sort a dag G, we run DFS on G and print the vertices in reverse order of finishing times.
In other words, sorting the vertices by descending finishing times yields a topological sorting.

- Reverse order can be obtained by putting each vertex on the front of a list as they are finished.
- Note that thus we do not actually need to sort the vertices.
- Thus the running time is **linear** (if sorting would be needed, then it would be only quasi-linear).

Topological sorting illustrated

Consider the result of running the DFS algorithm on the following dag.

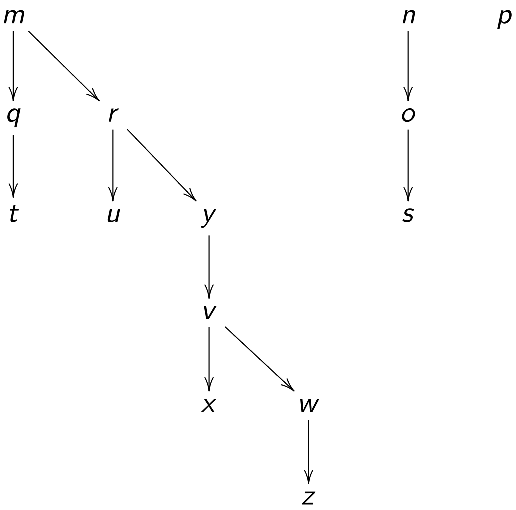


Note that this does not not follow lexicographical order of the vertices (there is one exception).

Listing the vertices in reverse order of finishing time gives the following topological sorting of the vertices:

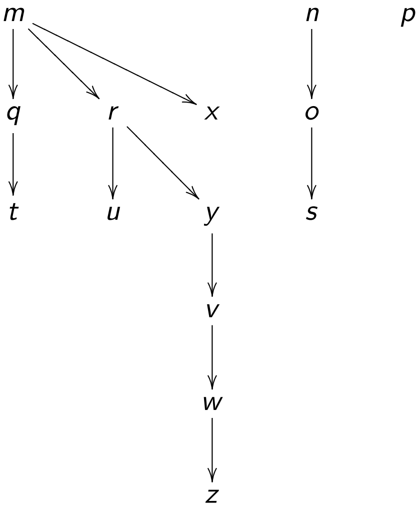
u :	p	n	o	s	m	r	y	v	w	z	x	u	q	t
$f[u]$:	28	26	25	24	20	19	18	17	16	15	12	8	5	4

The extracted directed spanning forest



Replay the discovery and finishing times of DFS on this forest!

A directed spanning BFS-forest (with 2 restarts)



Note how this directed forest is developed in layers!

Remarks on choices

The DFS algorithm, as presented, is **non-deterministic**, i.e., allows choices:

1. in Line 4 of the main body, running through all vertices in the graph — any order of the vertices can be used;
2. in Line 3 for the recursive procedure, running through all neighbours — any ordering of the neighbours can be used.

The point is that these loops run over all elements of a *set* — but a set is unordered, and thus the order in which we run through the elements is not specified.

As the analysis shows, the properties of DFS we considered (run time, DFS forest, meaning of discovery and finishing times) are **independent** of these choices.

Remarks on choices (cont.)

However the concrete outcomes DO dependent on these choices (obviously).

- In order to make the DFS deterministic, and thus the outcome unique, a rule for the choice of vertices is used.
- Often we will use numerical order (as in the lab-implementation), or alphabetical order.

Consider the previous application of DFS to compute a topological sorting. If we would use *reverse alphabetical order*, then we would obtain the topological sorting

vertex	m	n	p	o	q	s	r	u	t	y	v	w	x	z
finishing time	28	26	24	23	20	18	17	14	13	10	9	8	6	2

Deciding acyclicity

For a graph G (i.e., undirected), detecting the existence of a cycle is simple, via BFS or DFS:

G has a cycle (i.e., is not a forest) if and only if BFS resp. DFS will discover a vertex twice.

One has to be a bit more careful here, since the parent vertex will always be discovered twice, and thus has to be excluded, but that's it.

However for digraphs it is not that simple — do you see why?

Revisiting the lemma for topological sorting

In Lemma 5 we said:

> If G has no cycles, then along every arc the finishing times strictly decrease.

So if we discover in digraph G an arc (v, w) with $f[v] < f[w]$, then we know there is a cycle in G.

Is this sufficient criterion also necessary (for having a cycle)?

YES: just consider any cycle, and what must happen with the finishing times on it. We get:

Lemma 6

Consider a digraph G. Then G has a cycle if and only if any/every run of DFS on G yields some arc $(v, w) \in E(G)$ with $f[v] < f[w]$.

Revisiting the lemma for topological sorting

In Lemma 5 we said:

> If G has no cycles, then along every arc the finishing times strictly decrease.

So if we discover in digraph G an arc (v, w) with $f[v] < f[w]$, then we know there is a cycle in G.

Is this sufficient criterion also necessary (for having a cycle)?

YES: just consider any cycle, and what must happen with the finishing times on it. We get:

Lemma 6

Consider a digraph G. Then G has a cycle if and only if any/every run of DFS on G yields some arc $(v, w) \in E(G)$ with $f[v] < f[w]$.

Revisiting the lemma for topological sorting

In Lemma 5 we said:

> If G has no cycles, then along every arc
> the finishing times strictly decrease.

So if we discover in digraph G an arc (v, w) with $f[v] < f[w]$, then we know there is a cycle in G.

Is this sufficient criterion also necessary (for having a cycle)?

YES: just consider any cycle, and what must happen with the finishing times on it. We get:

Lemma 6

Consider a digraph G. Then G has a cycle if and only if any/every run of DFS on G yields some arc $(v, w) \in E(G)$ with $f[v] < f[w]$.

Revisiting the lemma for topological sorting

In Lemma 5 we said:

> If G has no cycles, then along every arc
> the finishing times strictly decrease.

So if we discover in digraph G an arc (v, w) with $f[v] < f[w]$, then we know there is a cycle in G.

Is this sufficient criterion also necessary (for having a cycle)?

YES: just consider any cycle, and what must happen with the finishing times on it. We get:

Lemma 6

Consider a digraph G. Then G has a cycle if and only if any/every run of DFS on G yields some arc $(v, w) \in E(G)$ with $f[v] < f[w]$.

Summary

The **fundamental definition** is that of a DAG:

A DAG is a digraph, which does not have any cycle.

This gives us a possibility to demonstrate that a digraph G is NOT a dag:

Show a cycle in G, that is,
a sequence of vertices connected by arcs,
such that we come back to the original vertex.

But we do not have (yet) an (efficient) possibility to demonstrate that G indeed IS a dag:

How can we show that there is NO cycle?

Summary (cont.)

Now there is a **sufficient criterion** for being a dag:

If a digraph G has a topological sorting,
then G must be a dag.

So we could prove, that G *is* a dag, by demonstrating a topological sorting for G.

- Is this complete? That is, can we always prove that a digraph is a dag, by giving some topological sorting?
- Is this efficient?

The answer to both questions is YES:

- A digraph has a topological sorting iff it is a dag.
- Our algorithm (based on DFS) for computing a topological sorting will succeed on any dag.

Summary (cont.)

Thus we can decide, whether digraph G is a dag, in linear time as follows:

1. Run DFS on it, and obtain the list of vertices sorted by descending finishing times.
2. Check whether this order is a topological sorting (by running through all arcs):
    1. If YES, then G is a dag.
    2. if NO, then G is not a dag.
3. This algorithm does not give a cycle in the NO-case — but indeed by some slight extension of the algorithm we could also extract a cycle in this case.