4 Connectivity, BFS

- BFS (connectivity, bipartiteness)
- Up next: DFS, topological sorting

4.1 Graph Definitions

Definition 4.1
A **cycle** is a path with $v_0 = v_k$ and all other vertices distinct, with $k \geq 3$ (or $k \geq 2$ for digraphs).

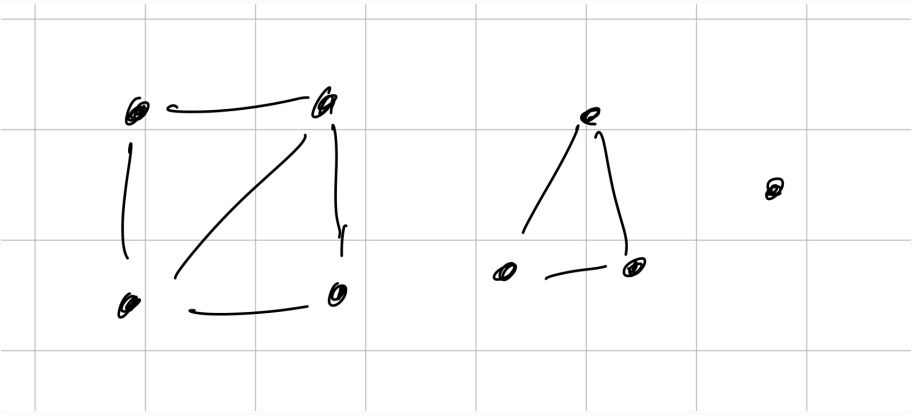
Definition 4.2
The **distance** between vertices u and v is the length of a shortest path between them.

4.2 Connectivity

Definition 4.3
An undirected graph is **connected** if for all $u, v \in V$, there exists a path from u to v and a path from v to u. Note that the second condition is redundant since our edges are all undirected.

Definition 4.4
A **component** is a set of vertices such that for all pairs of vertices in the set, there is a path between them, and no superset of this set has this property.

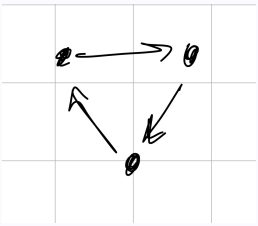
Example 4.5



In this graph there are 3 components.

Definition 4.6
A digraph is **strongly connected** if for all $u, v \in V$, there exists a path from u to v and from v to u.

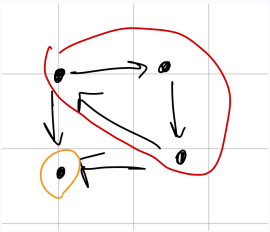
Example 4.7



This directed graph is strongly connected.

Definition 4.8
A **strongly connected component** of a digraph is a maximal set of vertices for which the induced subgraph is strongly connected.

Example 4.9



This graph has two strongly connected components, the three nodes circled in red, and the singular node circled in yellow.

4.3 Trees

Definition 4.10
A **forest** is a graph with no cycles.

A **tree** is a connected forest.

Note 4.11
An n-vertex tree has $m = n - 1$ edges.

Definition 4.12
If we designate one vertex of a tree as the **root**, then the (unique) neighbor of any vertex that is closer to the root is called its **parent**, and all other neighbors are its **children**.

A **leaf** of a tree is a degree-1 vertex.

A digraph with no directed cycles is called a **directed acyclic graph**, or **DAG**.

4.4 Connectivity

Definition 4.13
To understand connectivity, we can construct a **spanning tree**, a subgraph that includes all the vertices (i.e., is "spanning"), and is a tree.

We'll specify a tree by giving the parent $pr[v]$ of every vertex v, with $pr[root] = \varnothing$

4.4.1 Algorithm for constructing a spanning tree

```
Generic algorithm for constructing a spanning tree for the
component containing s (rooted at s):

Let T be the graph with one vertex, s, with pr[s] = null.
While there exists an edge {u, v} joining a vertex u of T
  with a vertex v not in T:
    Add vertex v and edge {u, v} to T
    Set pr[v] = u
```

We claim that the resulting graph is always a tree.

Lemma 4.14
This tree spans the component containing s.

Proof. Any vertex in this tree has a path to s: we can repeatedly take parents until they lead back to s. There is a path between any tow vertices u and v in the tree: consider paths from u to s and v to s, and we can join them together to form a path from u to v.

On the other hand, if u is not in T, then there is no path from u to S. If there were, we could follow the path and find an edge from a vertex not in T to a vertex in T. But no such edge can remain when the algorithm terminates. □

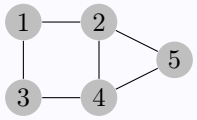
4.4.2 Breadth-first search (BFS)

The main idea is to explore the graph in order of increasing distance from s.

We say that a vertex is **exhausted** if it is not adjacent to any vertex outside of the tree.

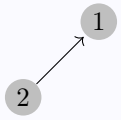
```
BFS(s):
Let T be the tree with on vertex, s, with pr[s] = null.
Repeat:
    Let u be the unexhausted vertex that
      joined the tree earliest (the *active vertex*)
    Choose an edge {u, v} where v is not in T
    Add vertex v and edge {u, v} to T
    Set pr[v] = u
Until all vertices are exhausted.
```

Example 4.15
Suppose we perform $BFS(1)$ on

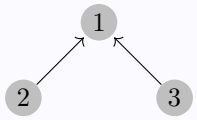


Node 1 is our first active vertex.

We add the node 2 to our graph. Note that the arrows here do not signify that our graph is directed, but rather the direction one must go to travel to the root.



Now we add 3.



Now 2 is our active vertex.