Spanning trees

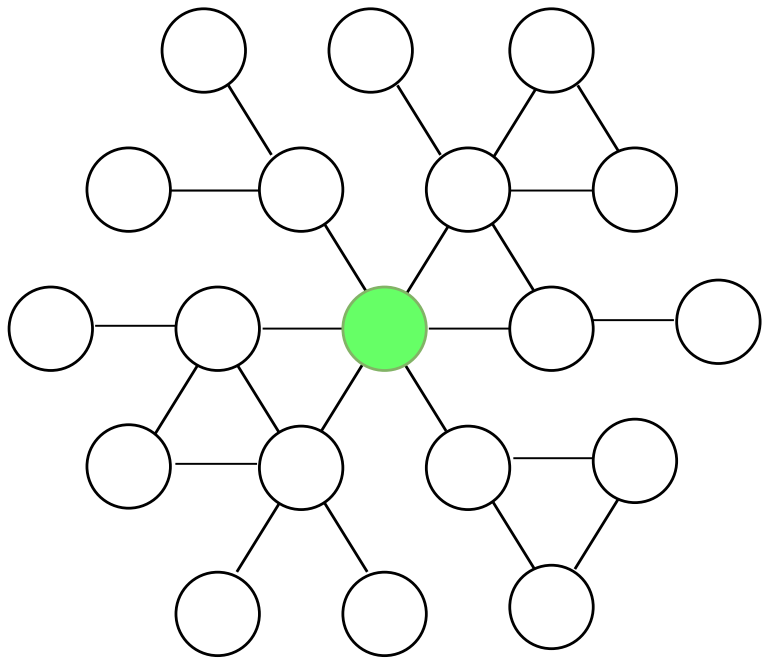
Lecturer: Arran Stewart

Outline

- What is a spanning tree? Why do we use them?
- What algorithms can we use to construct spanning trees?

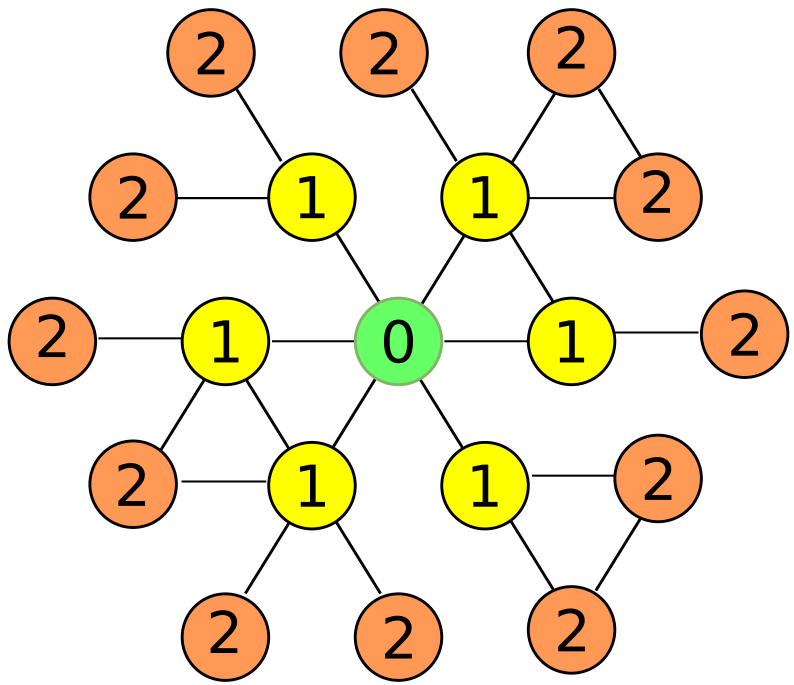
BFS in graphs

- Recall that when we do breadth-first search in graphs, we visit nodes in the graph in "layers" –
- we start with an initial node n, then we visit nodes which are one link distant from n, then 2, and so on.
- If our graph happened to consist of nodes like those below . . .



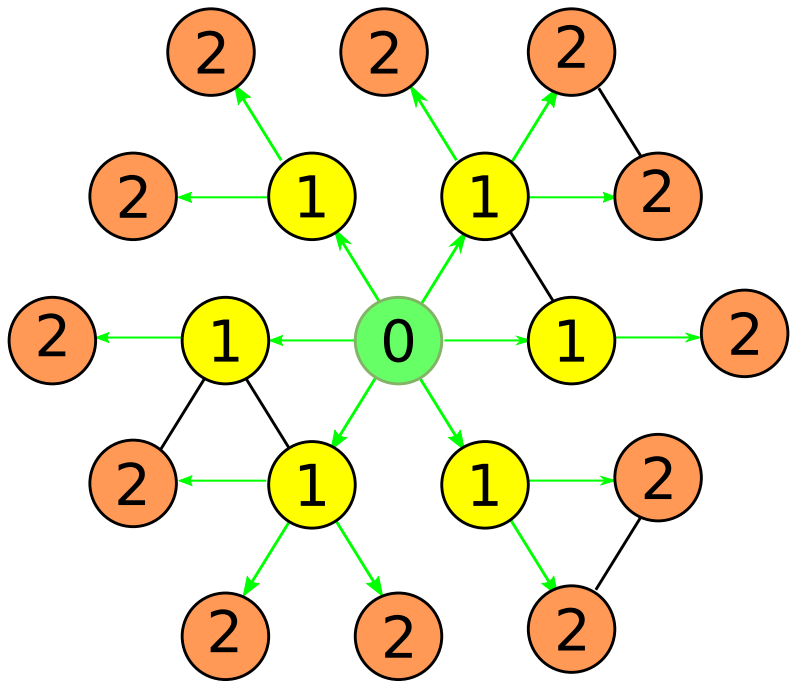
BFS in graphs

- . . . then the nodes would get visited in this order:



BFS in graphs

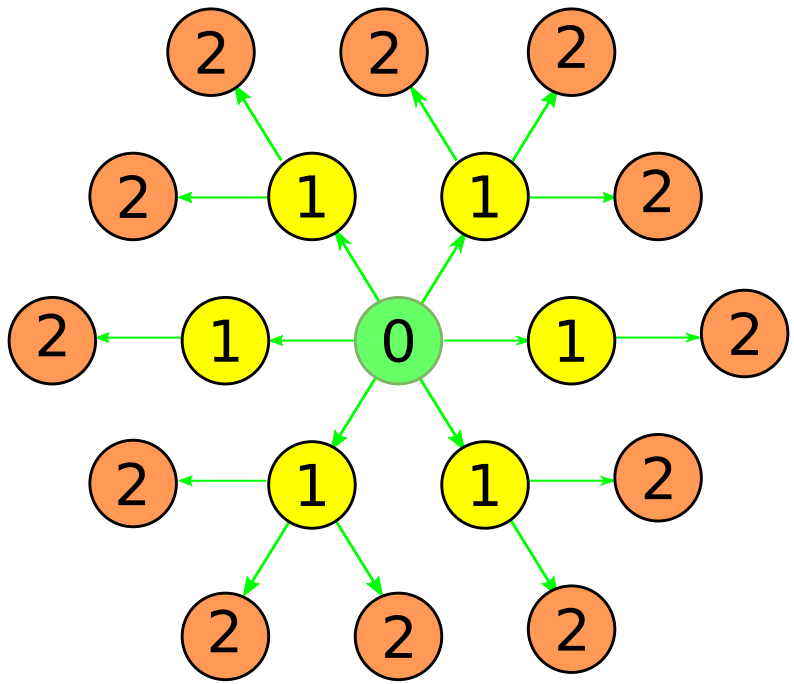
Furthermore, even though the *graph* has cycles, if we keep track of which node we "came from" while doing our visit, we will *never* from a cycle – since we never visit a node we've already visited.



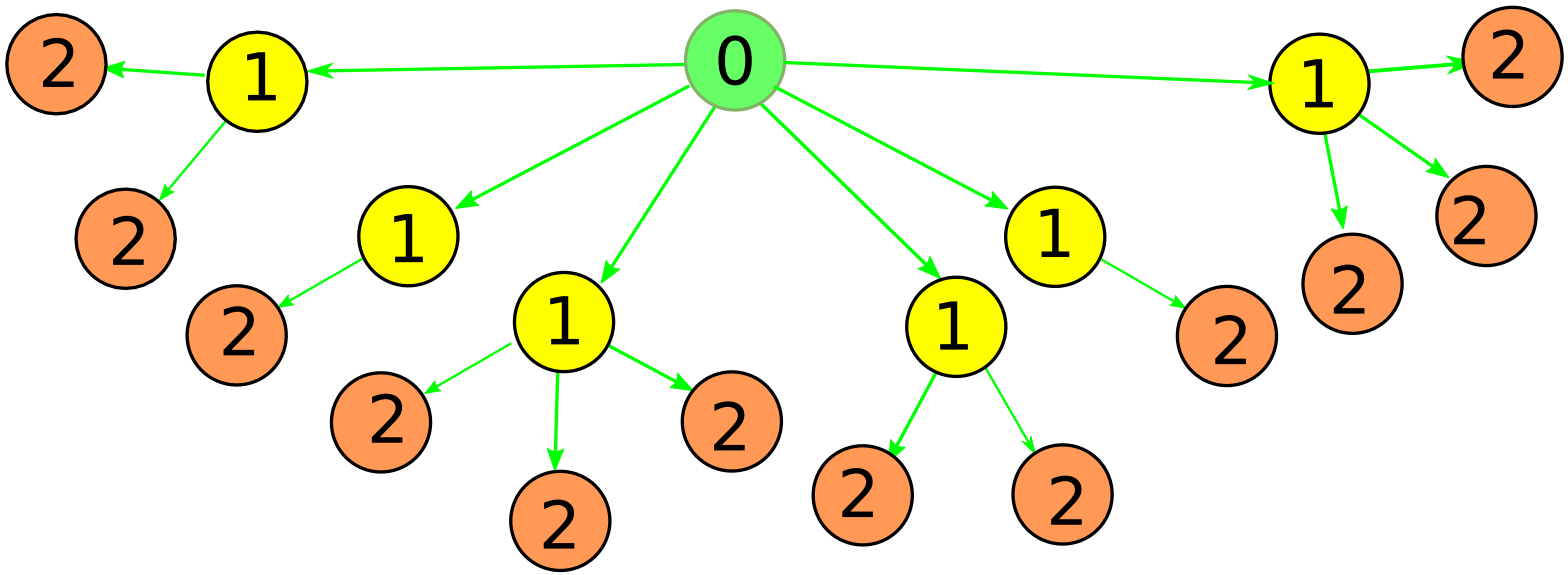
BFS in graphs

And if we keep track of the vertex we "came from" – then there'll be exactly one path between any two vertices. (Ignore the "direction" arrows.)

Which is exactly what a *tree* is: a tree is just a graph in which there is exactly one path between any two vertices.



BFS in graphs



BFS in graphs

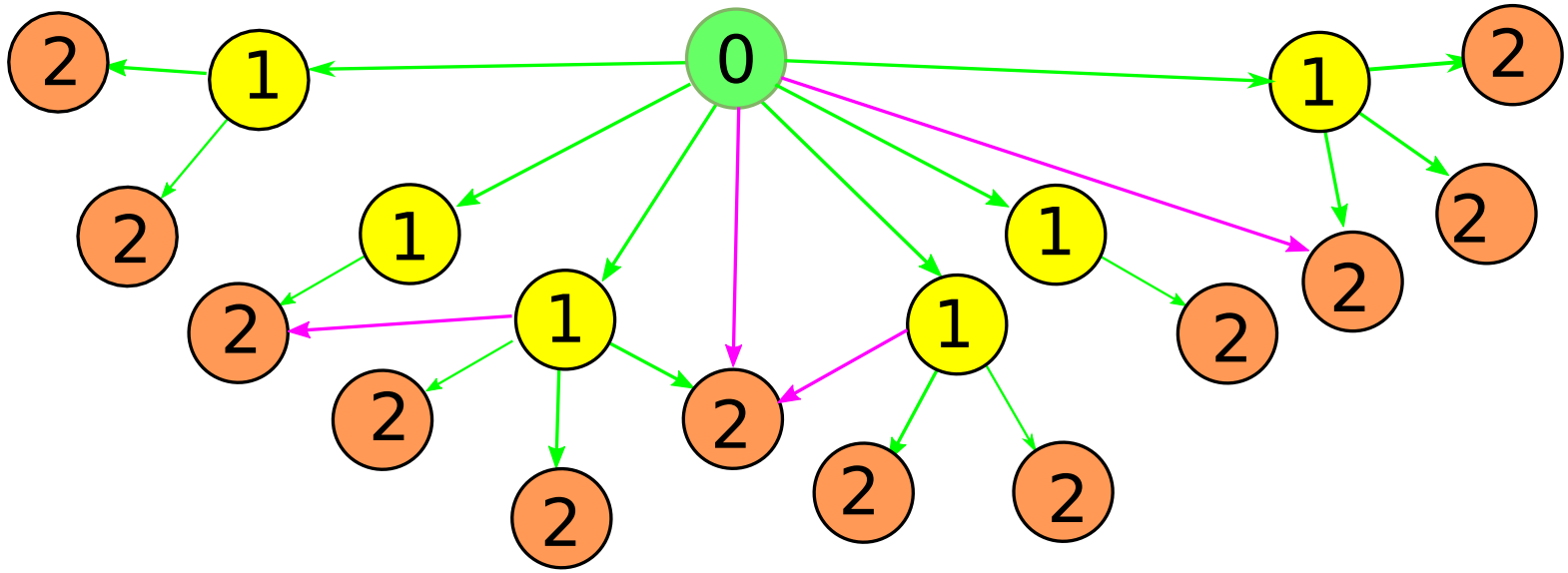
So, we can use the BFS algorithm to turn a *graph* into a *tree*.

Furthermore, there are no "extra" edges ...

BFS in graphs

So, we can use the BFS algorithm to turn a *graph* into a *tree*.

Furthermore, there are no "extra" edges . . .



. . . unlike in this graph (in which there is *more than one path* between some vertices).

Terminology

Let us define some useful terminology:

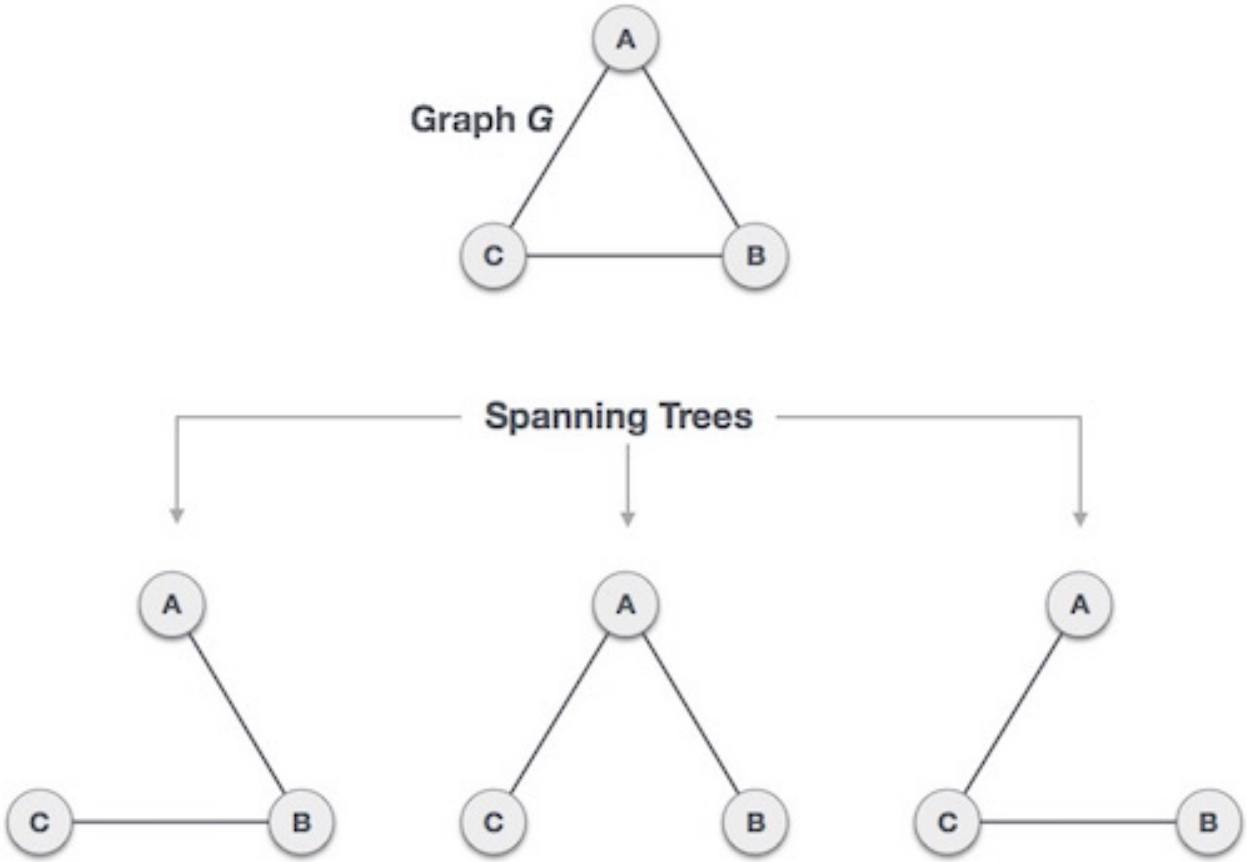
- A **subgraph** of a graph G is any subset of the edges and vertices of G.
- A **connected subgraph** is one in which there is *some* path between any two vertices in the graph.

Terminology

- We have constructed what is called a **spanning tree** of our graph G.
  - Or to be precise, once we leave off the arrow-heads, we will have.
- A spanning tree is a **connected subgraph** of G that spans all vertices.
  - In other words, it includes all the *vertices* of G, but may not include all the edges.

Spanning trees

For any one graph, we can have *multiple* spanning trees:



Minimum spanning trees

- Furthermore, because we used no "extra edges", we have what is called a **minimimum spanning tree** (MST) of our graph.
- A minimum spanning tree is a tree which has the minimum total "cost";
- and for an unweighted graph, the cost of a tree is the number of edges.
- We used the minimum number of edges to construct our tree, hence it is a minimum spanning tree.

Minimum spanning trees

So, to summarize: a minimum spanning tree . . .

- . . . counts as a **tree** because it has no cycles (we say it is acyclic).
- . . . is "**spanning**" because it spans the graph – it covers every vertex.
- . . . is **minimum** because it has the smallest cost amongst all possible spanning trees.

Why do we want MSTs?

MSTs are useful because many problems turn out to make use of them.

For instance – suppose we want to build roads between a number of towns, and we are interested in keeping our costs low.

We therefore might want to build the *smallest* number of roads possible, that will nevertheless allow people to travel from any town to any other town.

The result will be minimum spanning tree: the towns are nodes, the roads are edges, and the cost of each road could be the cost of building it.

(In this case, we would want to use a weighted graph, rather than unweighted – for each road, we'd want to associate with it some number, namely the cost to build it.)

Why do we want MSTs?

Other places we might want to use MSTs could be ...

- in working out connections needed on a circuit board to connect components
- in working out what cables are needed in order to connect telephone exchanges in a telecommunications network
- performing cluster analysis of data (working out how a data set can be organized into clusters of data points with a small "distance" between points in each cluster).

MSTs for weighted graphs

- If we want to find an MST for a *weighted* graph – that is, a graph where each edge is associated with some number, a *weight* – then we have to amend our algorithm somewhat.
- We will study **Prim's algorithm** for generating an MST for a weighted graph.

Greedy algorithms

- Prim's algorithm is what is called a *greedy* algorithm.
- A greedy algorithm is one in which we don't have an overall "strategy" for achieving some goal, but just do whatever looks best at the moment.
- For instance, suppose we want to get to the top of a mountain as quickly as possible. (And are very good at ascending even the steepest surfaces.)
- In that case, if we had the choice between two roads – a gentle road and a steep road – we would take the steep road, because it ascends most quickly.
- A *greedy strategy* for ascending the mountain would be one where, at every possible choice, we took the steepest path.
- (In fact, from this analogy we get a computational search technique known as "hill-climbing".)

Greedy algorithms

- For some sorts of problems, greedy algorithms won't give us a good solution; but luckily, for the MST problem, they do.

Prim's algorithm

For our basic BFS algorithm, we used a **queue** to store a list of nodes we had to visit.

In Prim's algorithm, we use a **priority queue**.

Recall that in a normal queue, the first item *enqueued* into the queue is also the first item we get when we *dequeue*.

But in a priority queue, our items can be put in order of *priority*, and we instead get the item in the queue with highest priority.

Prim's algorithm

As with the BFS algorithm, we will also need an array **colour**, which can hold the values
white (0), grey (1) or black (2) for each vertex.

White means "not visited at all", grey means "added to the queue, but not yet visited", and black means "visited".

We'll need (as with BFS) an array **p** ("p" standing for "parent"), where **p**(v) contains the immediate parent of each node in the MST.

And finally, we'll also need an array **key**, with the weights of the edges in the tree.

Algorithm pseudocode

Prim's algorithm is in fact extremely similar to our BFS algorithm...

BFS algorithm

```
- put v into the queue Q
- colour[v]=white for all vertices v,
- p(v) is undefined.
- while Q is not empty:
   - dequeue w from the head of Q
   - for each vertex x adjacent to w:
     - if colour[x] is white:
       - p(x) = w
       - colour[x] = grey
       - enqueue x onto the tail of Q
     - colour[w] = black
```

Algorithm pseudocode

Except that for additional steps shown here in blue:

Prim's algorithm

```
- put v into the queue Q
- colour[v]=white for all vertices v,
- p(v) is undefined.
- while Q is not empty:
   - dequeue w from the head of Q
   - for each vertex x adjacent to w:
     - if colour[x] is white:
       - p(x) = w
       - colour[x] = grey
       - enqueue x onto the tail of Q
     - else if colour[v] is grey:
       - if key[v] > edgeWeight(w,v):
         - key[v] = edgeWeight(w,v)
         - p(v) = w
     - colour[w] = black
```

Algorithm result

At the end of the MST search, every vertex in the graph will have the colour black and the parent array **p** will contain the edges of the MST.

We will not step through Prim's algorithm in detail; but Java code for the algorithm is provided in `PrimMST.java` in your Java code bundle, and it is recommended you try out the code in Eclipse.