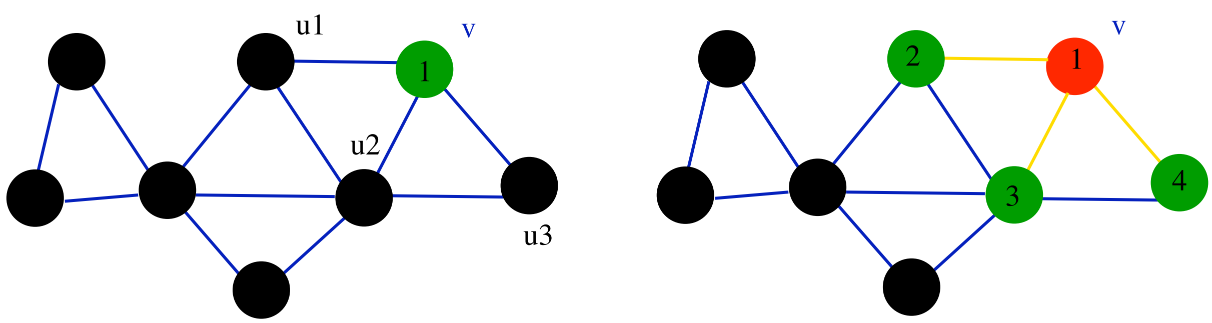
Breadth-First Search (BFS)

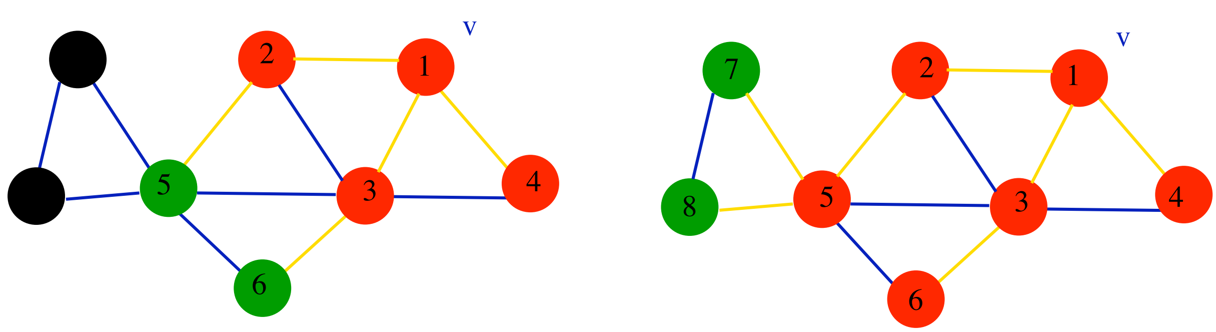
Intuition: **BFS**(vertex v)

To start, all vertices are *unmarked*.

- *Start* at v. *Visit* v and mark as *visited*.
- *Visit* every *unmarked* neighbour u_i of v and mark each u_i as visited.
- *Mark* v finished.
- *Recurse* on each vertex marked as *visited* in the order they were visited.



BFS of an undirected Graph



Q: *What information about the graph can a BFS be used to find?*

-

-

Q: *What does the BFS construct?*

Q: *What is an appropriate ADT to implement a BFS given an adjacency list representation of a graph?*

which has the operations:

-

-

-

Q: *What information will we need to store along the way?*

-

-

-

The BFS Algorithm

We will use `p[v]` to represent the *predecessor* of v and `d[v]` to represent the *number of edges* from v (i.e., the *distance* from v).

```
BFS(G=(V,E),v)
  for all vertices u in V
    color[u] := black
    d[u] := infinity;     \\ we use infinity to denote
    p[u] := NIL;          \\   "not connected"
  end for
  initialize an empty queue Q;
  color[v] := green;
  d[v] := 0;
  p[v] := NIL;
  ENQUEUE(Q,v);
  while not ISEMPTY(Q) do
    u := DEQUEUE(Q);
    for each edge (u,w) in E do
      if (color[w] == black) then
        color[w] := green;
        d[w] := d[u] + 1;
        p[w] := u;
        ENQUEUE(Q,w);
      end if
    end for
    color[u] := red;
  end while
END BFS
```

Complexity of BFS(G,v)

Q: *How many times is each node* `ENQUEUE`*ed?*

Therefore, the *adjacency list of each node* is examined at most once, so that the total running time of `BFS` is

or linear in the size of the adjacency list.

NOTES:

- BFS will visit only those vertices that are *reachable* from v.
- If the graph is *connected (in the undirected case)* or *strongly-connected (in the directed case)*, then this will be all the vertices.
- If not, then we may have to call BFS on more than one start vertex in order to see the whole graph.

Q: *Prove that $d[u]$ really does represent the length of the shortest path (in terms of number of edges) from v to u.*

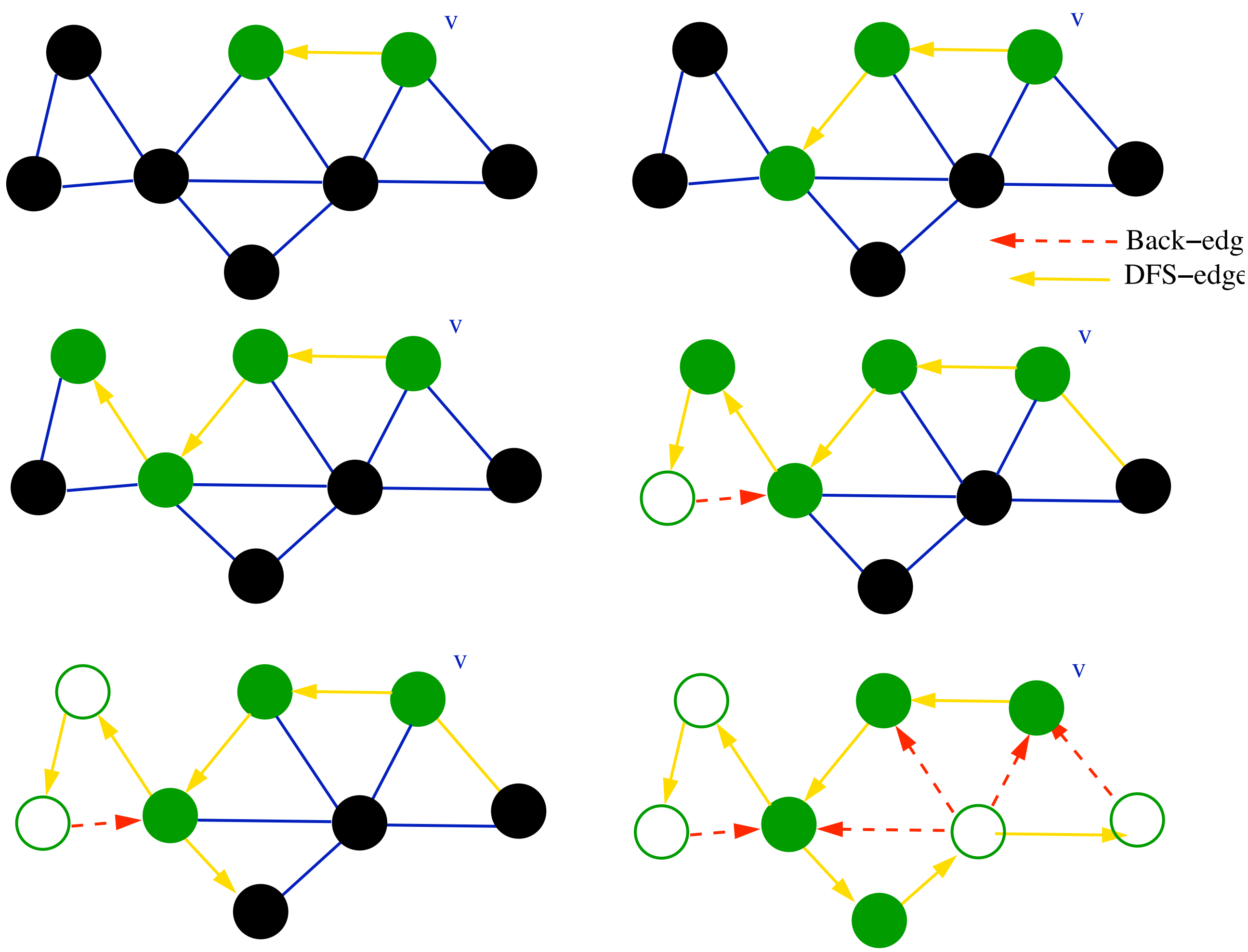
Depth-First Search

Intuition: DFS(G,v)

All *vertices* and *edges* start out *unmarked*.

- Walk as *far* as possible *away* from v visiting vertices
- If the *current vertex* has not been *visited*,
  - Mark as visited and the edge that is traversed as a DFS edge.
- Otherwise, if the *vertex* has been visited,
  - mark the traversed edge as a back-edge, back up to the previous vertex
- When the current vertex has only *visited neighbours left* mark as *finished (white)*.
- *Backtrack* to the *first vertex* that is not *finished*.
- Continue.

Example.



Just like BFS, *DFS* constructs a spanning-tree and gives *connected component* information.

Q: *Does it find the shortest path between v and all other vertices?*

Implementing a DFS

Q: *Which ADT would be the most helpful for implementing DFS given an adjacency list representation of G?*

with the operations

-
-
-

Q: *What additional data (for each vertex) should we keep in order to easily determine whether an edge is a cross-edge or a DFS-edge?*

- $d[v]$ will indicate the
- $f[v]$ will indicate the

Algorithm DFS(G,s)

```
DFS(G=(V,E),s)
  for all vertices v in V
    color[v] := black;
    d[v] := infinity;
    f[v] := infinity;
    p[v] := NIL;
  end for
  initialize an empty stack S;
  color[s] := green; d[s] := 0; p[s] := NIL;
  time := 0;
  PUSH(S,(s,NIL));
  for each edge (s,v) in E do
    PUSH(S,(s,v));
  end for
  while not ISEMPTY(S) do
    (u,v) := POP(S);
    if (v == NIL) then        // Done with u
      time := time + 1;
      f[u] := time;
      color[u] := white;
    end if
    else if (color[v] == black) then
      color[v] := green;
      time := time + 1;
      d[v] := time;
      p[v] := u;
      PUSH(S,(v,NIL));        // Marks the end of v's neighbors
      for each edge (v,w) in E do
        PUSH(S,(v,w));
      end for
(*) end if
  end while
END DFS
```

Complexity of DFS(G,s)

Q: *How many times does DFS visit the neighbours of a node?*

-

- Therefore, the adjacency list of each vertex is visited at most once.

- So the *total running time* is just like for BFS, $\Theta(n+m)$ i.e., linear in the size of the adjacency list.

Note that the gold edges, or the DFS edges form a *tree* called the DFS-tree.

Q: Is the *DFS tree* unique for a given graph G starting at s?

For certain applications, we need to distinguish between different types of edges in E:

We can specify edges on *DFS-tree* according to how they are *traversed* during the search.

- **Tree-Edges** are the edges in the DFS tree.

- **Back-Edges** are edges from a vertex u to an ancestor of u in the DFS tree.

- **Forward-Edges*** are edges from a vertex u to a descendent of u in the DFS tree.

- **Cross-Edges*** are all the *other edges* that are not part of the *DFS tree* (from a vertex u to another vertex v that is neither an ancestor nor a descendent of u in the DFS tree).

Q: *Which variable facilitates distinguishing between these edges?*

Q: *How can a DFS be used to determine whether a graph G has any cycles?*

(Note: A *cycle* is a path from a vertex u to itself.)

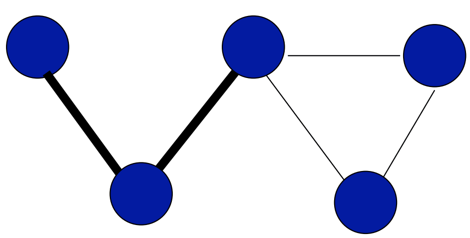
Q: *How can we detect cross-edges during a DFS?*

Minimum Cost Spanning Trees (MCSTs)

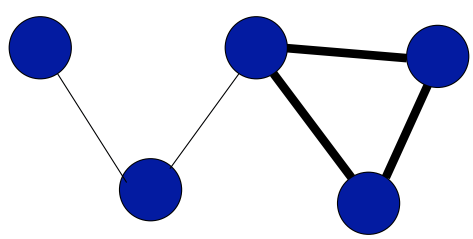
- Let $G = (V, E)$ be a *connected, undirected* graph with *edge weights* $w(e)$ for each edge $e \in E$.

- A *tree* is a subset of edges $A \subset E$ such that A is *connected* and *does not contain a cycle*.

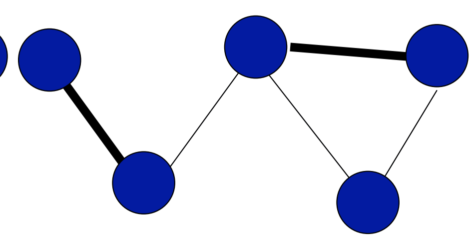
Thick edges are in A, the thin ones are not. The first one is a tree and the other two are not.



Tree

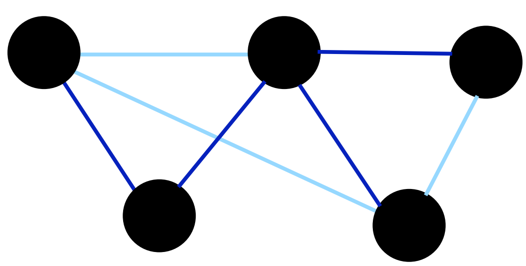


Not a tree (has a cycle)



Not a tree (not connected)

A *spanning tree* is a tree A such that *every vertex* $v \in V$ is an *endpoint* of at least *one edge* in A.



Q: How *many edges* must any *spanning tree* contain?

How would you prove this is true?

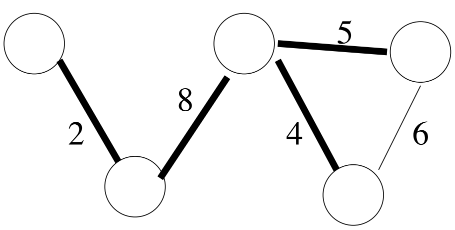
Q: *What algorithms have we seen to construct a spanning tree?*

A *minimum cost spanning tree* (**MCST**) is a spanning tree A such that the *sum of the weights is minimized* for all possible spanning trees B.

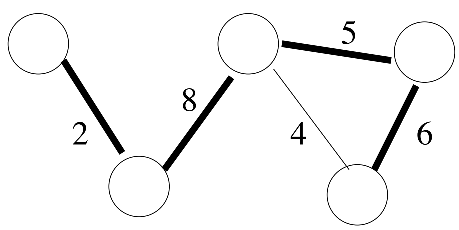
Formally:

$$w(A) = \sum_{e \in A} w(e) \leq w(B)$$

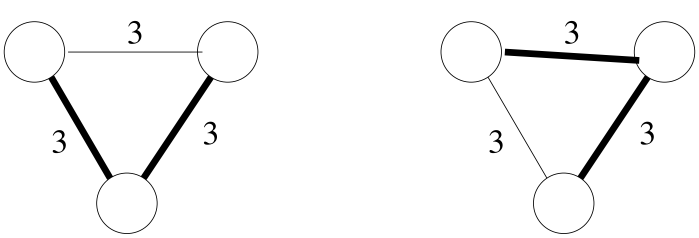
for all other spanning trees B.



Minimum Cost Spanning Tree



Spanning tree, but not minimum cost



Two different MCSTs on the same graph

Q: *What is an example of an application in which you would we want to be able to find a MCST?*

A:

We will look at two algorithms for constructing MCSTs. The first is *Prim's Algorithm*.

Prim's Algorithm

Prim's algorithm uses a *Priority Queue ADT*.

A *priority queue* is just like a queue except that *every item in the queue* has a *priority* (usually just a number).

More formally, a *priority queue* consists of

- a set of elements
- each element has a priority
- The operations:
  - `ENQUEUE(x,p)`:
  - `ISEMPTY()`:
  - `EXTRACT_MIN()`: