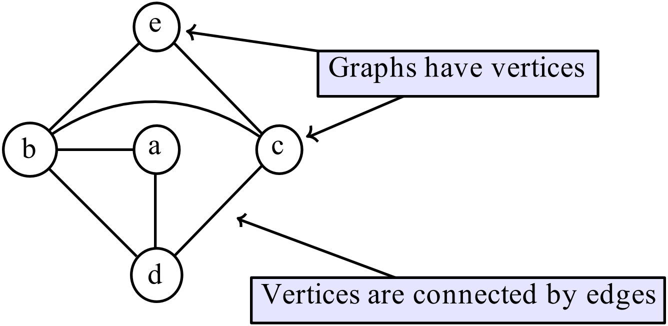
Video 3.1

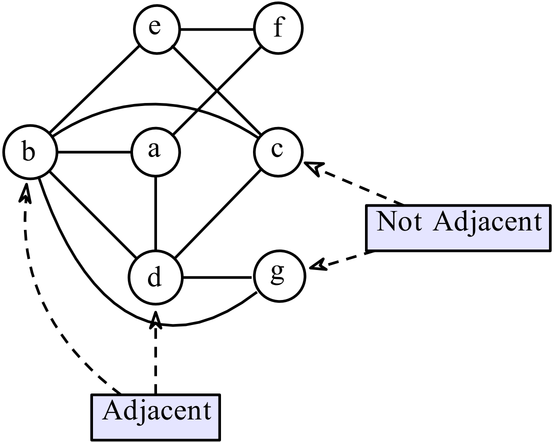
Sampath Kannan

Graphs

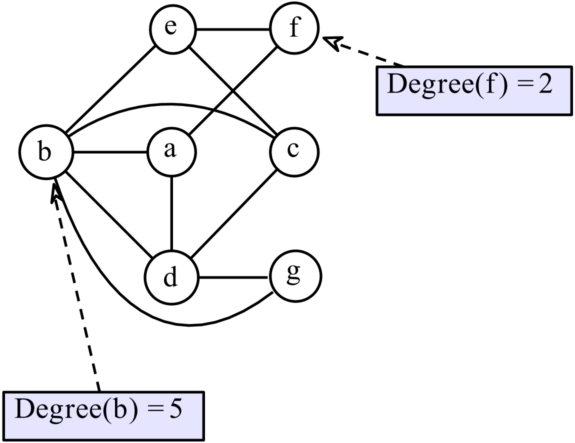
- Graphs are represented as a set of vertices, and a set of edges.
- Sometimes they concretely model a network (roads, communication) but usually represent abstract relationships (people/friendships, documents/similarity)



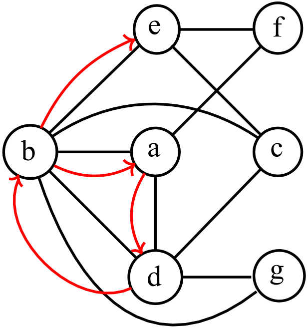
Graph terminology



Graph terminology

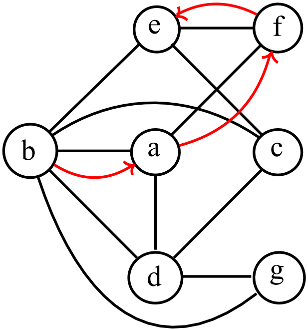


Graph terminology



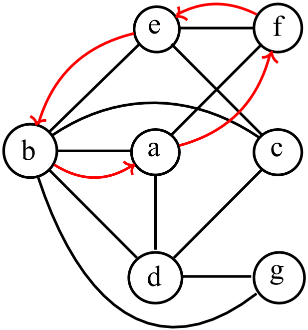
b, a, d, b, e is an example of a path

Graph terminology



b, a, f , e is an example of a simple path of length 3

Graph terminology



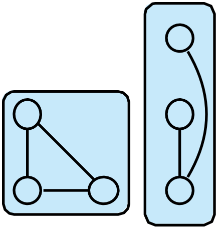
b, a, f , e, b is a cycle but b, a, b and b, a, d , b, e are not cycles

Connected

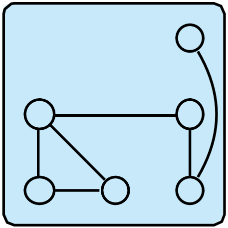
- Two vertices are **connected** if there is a (simple) path between them.
- Being connected is an **equivalence relation**:
  - Reflexive: Every vertex has a path of length 0 to itself
  - Symmetric: If there is a path from u to v then reverse it to get a path from v to u (only works in undirected graphs)
  - Transitive: If there is a path from u to v and a path from v to w then the paths can be concatenated to make a path from u to w.

Types of Graphs

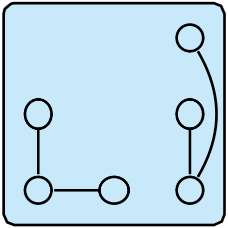
Connected Components



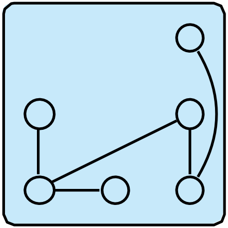
Connected Graph



Acyclic Graph

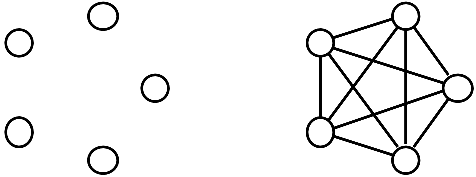


Tree



Number of vertices and edges

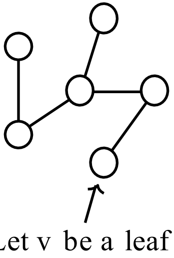
Convention: Use n to denote the number of vertices and m to denote the number of edges in a graph.



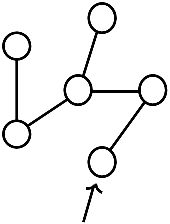
m can be as low as 0 or as high as $\binom{n}{2}$

- In a tree, a vertex of degree 1 is called a **leaf**.
- **Theorem**: Every tree (with $n \geq 2$) has a leaf.
- **Theorem**: A tree T on n vertices has $n - 1$ edges.
  Proof: By induction on n, base case for $n = 1, 2$ is trivial.

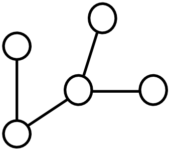
- In a tree, a vertex of degree 1 is called a **leaf**.
- **Theorem**: Every tree (with $n \geq 2$) has a leaf.
- **Theorem**: A tree T on n vertices has $n - 1$ edges.
  Proof: By induction on n, base case for $n = 1, 2$ is trivial.



- In a tree, a vertex of degree 1 is called a **leaf**.
- **Theorem**: Every tree (with $n \geq 2$) has a leaf.
- **Theorem**: A tree T on n vertices has $n-1$ edges.
  Proof: By induction on n, base case for $n = 1, 2$ is trivial.



Let v be a leaf



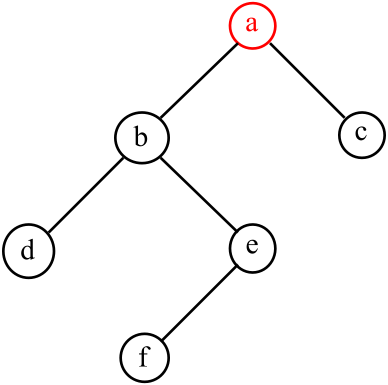
$T - \{v\}$ is a tree on $n-1$ vertices $\Longrightarrow$ it has $n-2$ edges

Video 3.2

Sampath Kannan

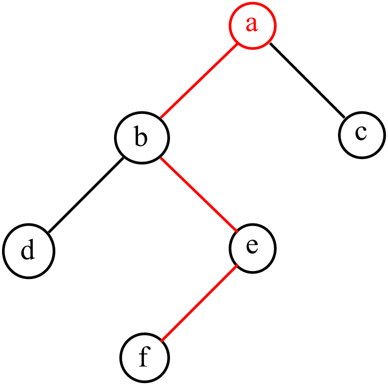
Trees: Revisited

- We've seen examples of rooted trees
  - heaps, binary search trees
- Unrooted trees are just connected, acyclic graphs
  - Can "pick them up" by any vertex to make them rooted



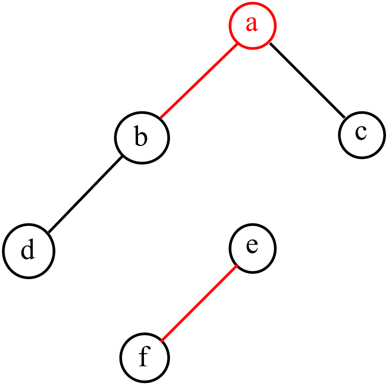
Trees: Revisited

- We've seen examples of rooted trees
  - heaps, binary search trees
- Unrooted trees are just connected, acyclic graphs
  - Can "pick them up" by any vertex to make them rooted
- Unique path from one vertex to another in a tree



Trees: Revisited

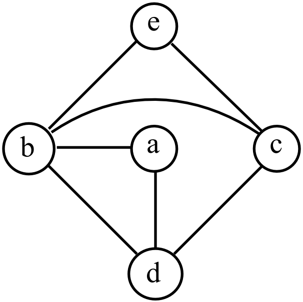
- We've seen examples of rooted trees
  - heaps, binary search trees
- Unrooted trees are just connected, acyclic graphs
  - Can "pick them up" by any vertex to make them rooted
- Unique path from one vertex to another in a tree
- If we remove one edge from a tree, the graph has two connected components



How To Represent Graphs

Adjacency Matrix:

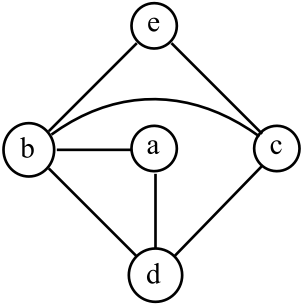
	a	b	c	d	e
a	0	1	0	1	0
b	1	0	1	1	1
c	0	1	0	1	1
d	1	1	1	0	0
e	0	1	1	0	0



How To Represent Graphs

Adjacency Matrix:

	a	b	c	d	e
a	0	1	0	1	0
b	1	0	1	1	1
c	0	1	0	1	1
d	1	1	1	0	0
e	0	1	1	0	0



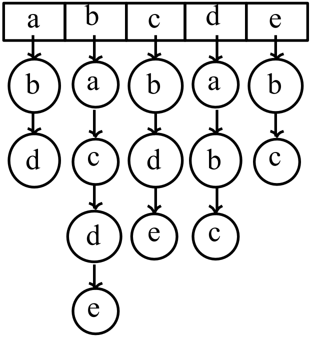
Space required? nxn matrix = $O(n^2)$

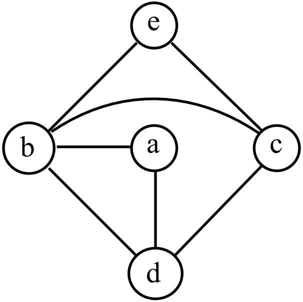
How long does it take to check if an edge (u, v) exists? $O(1)$

- Note: the adjacency matrix is symmetric

How To Represent Graphs

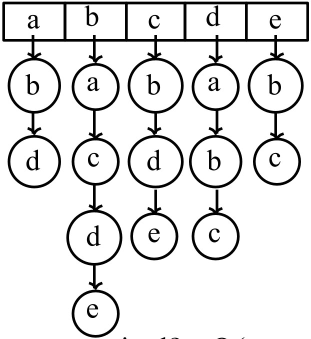
Adjacency List:

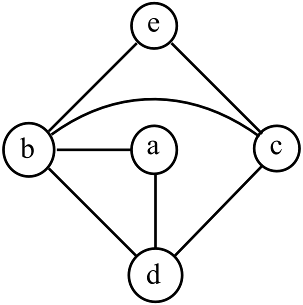




How To Represent Graphs

Adjacency List:





Space required? $= O(n+m)$

How long does it take to check if an edge (u, v) exists?

$= O(deg(u))$

Graph Interface

```
public interface Graph {

  public void addEdge(int u, int v);

  public List<Integer> neighbors(int v);

}
```

Implementations

Adjacency List

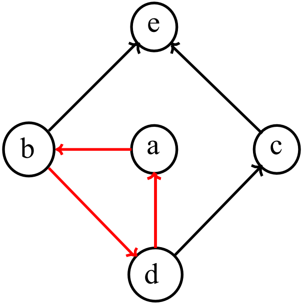
```
public class GraphAdj implements Graph {

  int n;
  private List<Integer>[] adj

  public GraphAdj(int n) {
    this.n = n;
    adj = (LinkedList<Integer>[]) new LinkedList[n];
    for (int i = 0; i < n; i++) {
      adj[i] = new LinkedList<Integer>();
    }
  }

  public void addEdge(int u, int v) {
    adj[u].add(v);
    adj[v].add(u);
  }

  public List<Integer> neighbors(int v) {
    return new LinkedList<Integer>(adj[v]);
  }

}
```

Adjacency Matrix

```
public class GraphMatrix implements Graph {

  int n;
  private boolean[][] adj;

  public GraphMatrix(int n) {
    this.n = n;
    adj = new boolean[n][n]
  }

  public void addEdge(int u, int v) {
    adj[u][v] = true;
    adj[v][u] = true;
  }

  public List<Integer> neighbors(int v) {
    List<Integer> neighbors = new LinkedList<Integer>();
    for (int i = 0; i < n; i++) {
      if (adj[v][i]) {
        neighbors.add(i);
      }
    }
    return neighbors;
  }

}
```

Directed Graphs

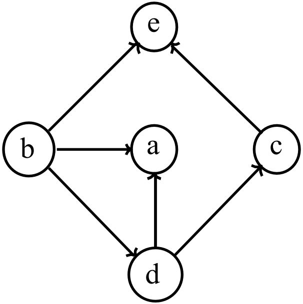
- Edges can be directed
  - Thought of as an ordered pair
  - $(a, b) \neq (b, a)$



Graph contains a cycle

Directed Graphs

- Edges can be directed
  - Thought of as an ordered pair
  - $(a, b) \neq (b, a)$
- **Directed Acyclic Graph (DAG):** A directed graph with no cycles
  - Does not have to be a tree



Graph is a DAG

Directed Graph Interface

```
public interface DirGraph {
  public void addEdge(int u, int v);
  public List<Integer> inNeighbors(int v);
  public List<Integer> outNeighbors(int v);
}
```

Implementations

Adjacency List

```
public class GraphAdj implements DirGraph {

  int n;
  private List<Integer>[] adj

  public GraphAdj(int n) {
    this.n = n;
    adj = (LinkedList<Integer>[]) new LinkedList[n];
    for (int i = 0; i < n; i++) {
      adj[i] = new LinkedList<Integer>();
    }
  }

  public void addEdge(int u, int v) {
    adj[u].add(v);
  }

  public List<Integer> outNeighbors(int v) {
     return new LinkedList<Integer>(adj[v]);
  }

  public List<Integer> inNeighbors(int v) {
    List<Integer> neighbors = new LinkedList<Integer>();
    for (int i = 0; i < n; i++) {
      if (adj[i].contains(v)) {
        neighbors.add(i);
      }
    }
    return neighbors;
  }

}
```

Adjacency Matrix

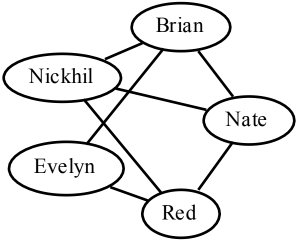
```
public class GraphMatrix implements DirGraph {

  this.n = n;
  private boolean[][] adj;

  public GraphMatrix(int n) {
    this.n = n;
    adj = new boolean[n][n]
  }

  public void addEdge(int u, int v) {
    adj[u][v] = true;
  }

  public List<Integer> outNeighbors(int v) {
    List<Integer> neighbors = new LinkedList<Integer>();
    for (int i = 0; i < n; i++) {
      if (adj[v][i]) {
        neighbors.add(i);
      }
    }
    return neighbors;
  }

  public List<Integer> inNeighbors(int v) {
    List<Integer> neighbors = new LinkedList<Integer>();
    for (int i = 0; i < n; i++) {
      if (adj[i][v]) {
        neighbors.add(i);
      }
    }
    return neighbors;
```

Video 3.3

Sampath Kannan

Graphs

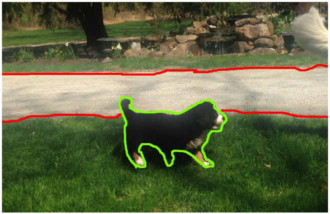
Friend Recommendations



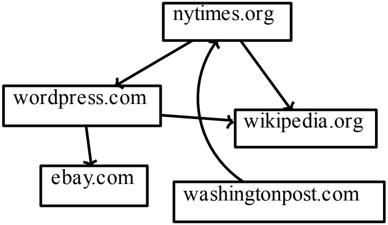
• Graphs model relationships

• Model problem as a graph, then design algorithm

Image Segmentation



Returning Search Results



Exploring Graphs

- If we have a graph we want to be able to explore it in some sensible way

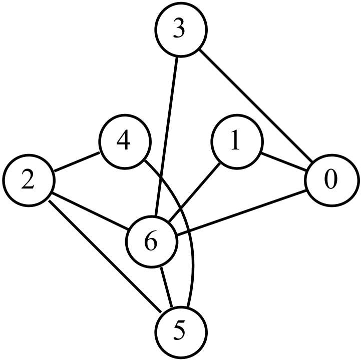
Exploring Graphs

- If we have a graph we want to be able to explore it in some sensible way
- There are important methods for this:
  - Depth-first search (DFS)
  - Breadth-first search (BFS)

Exploring Graphs

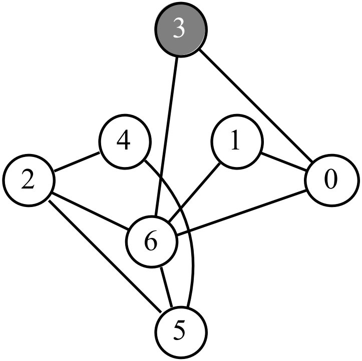
- If we have a graph we want to be able to explore it in some sensible way
- There are important methods for this:
  - Depth-first search (DFS)
  - Breadth-first search (BFS)
- Suppose you are exploring a neighborhood in a new city. . .
  - You might walk as far as you can down each path before turning around (DFS)
  - Or you might only explore the nearby spots before venturing further (BFS)

DFS Details



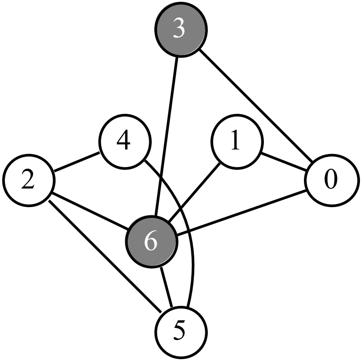
- Vertices are either unseen (white), visited (grey), or finished (black)
- When visiting a vertex u, if there is an edge to an unseen vertex v , follow that edge and visit v. v is now marked as visited.
- Continue recursively at v, if v has no unseen neighbors then v is finished.

DFS Details



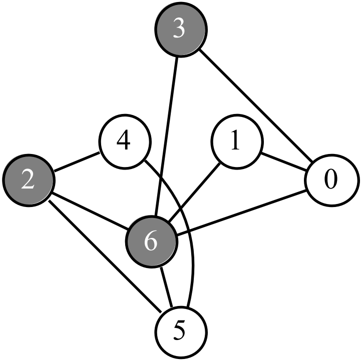
- Vertices are either unseen (white), visited (grey), or finished (black)
- When visiting a vertex u, if there is an edge to an unseen vertex v , follow that edge and visit v . v is now marked as visited.
- Continue recursively at v , if v has no unseen neighbors then v is finished.

DFS Details



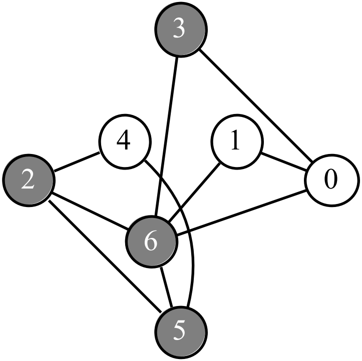
- Vertices are either unseen (white), visited (grey), or finished (black)
- When visiting a vertex u, if there is an edge to an unseen vertex v , follow that edge and visit v . v is now marked as visited.
- Continue recursively at v , if v has no unseen neighbors then v is finished.

DFS Details



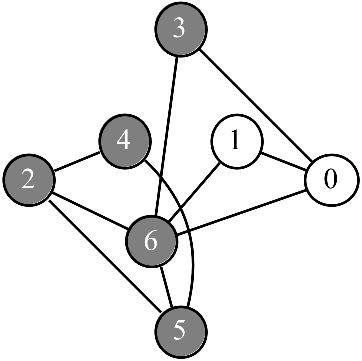
- Vertices are either unseen (white), visited (grey), or finished (black)
- When visiting a vertex u, if there is an edge to an unseen vertex v , follow that edge and visit v . v is now marked as visited.
- Continue recursively at v , if v has no unseen neighbors then v is finished.

DFS Details



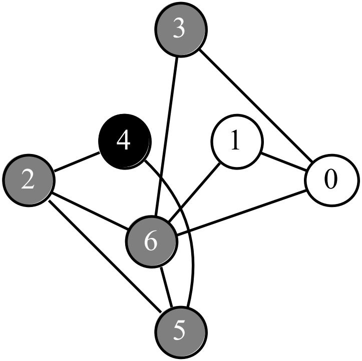
- Vertices are either unseen (white), visited (grey), or finished (black)
- When visiting a vertex u, if there is an edge to an unseen vertex v , follow that edge and visit v . v is now marked as visited.
- Continue recursively at v , if v has no unseen neighbors then v is finished.

DFS Details



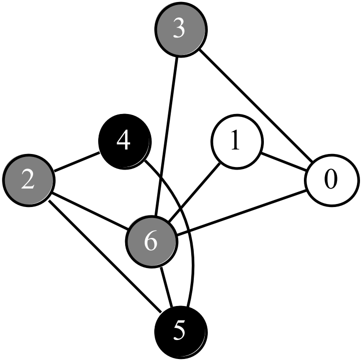
- Vertices are either unseen (white), visited (grey), or finished (black)
- When visiting a vertex u, if there is an edge to an unseen vertex v , follow that edge and visit v . v is now marked as visited.
- Continue recursively at v , if v has no unseen neighbors then v is finished.

DFS Details



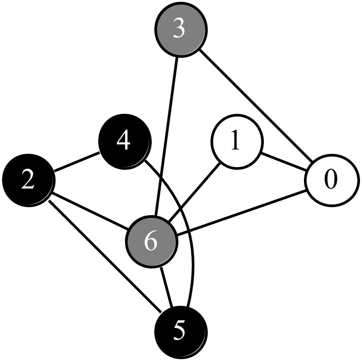
- Vertices are either unseen (white), visited (grey), or finished (black)
- When visiting a vertex u, if there is an edge to an unseen vertex v , follow that edge and visit v. v is now marked as visited.
- Continue recursively at v, if v has no unseen neighbors then v is finished.

DFS Details



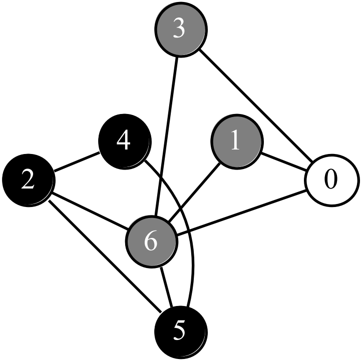
- Vertices are either unseen (white), visited (grey), or finished (black)
- When visiting a vertex u, if there is an edge to an unseen vertex v , follow that edge and visit v . v is now marked as visited.
- Continue recursively at v , if v has no unseen neighbors then v is finished.

DFS Details



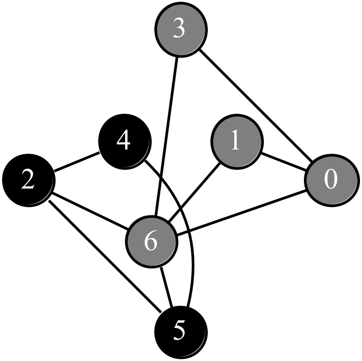
- Vertices are either unseen (white), visited (grey), or finished (black)
- When visiting a vertex u, if there is an edge to an unseen vertex v, follow that edge and visit v. v is now marked as visited.
- Continue recursively at v, if v has no unseen neighbors then v is finished.

DFS Details



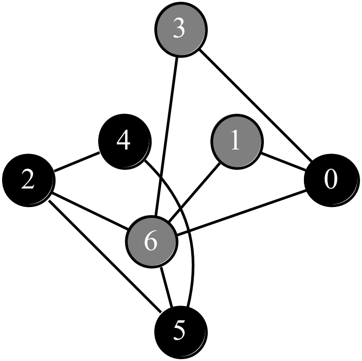
- Vertices are either unseen (white), visited (grey), or finished (black)
- When visiting a vertex u, if there is an edge to an unseen vertex v, follow that edge and visit v. v is now marked as visited.
- Continue recursively at v, if v has no unseen neighbors then v is finished.

DFS Details



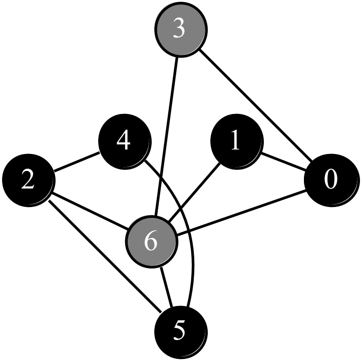
- Vertices are either unseen (white), visited (grey), or finished (black)
- When visiting a vertex u, if there is an edge to an unseen vertex v, follow that edge and visit v. v is now marked as visited.
- Continue recursively at v, if v has no unseen neighbors then v is finished.

DFS Details



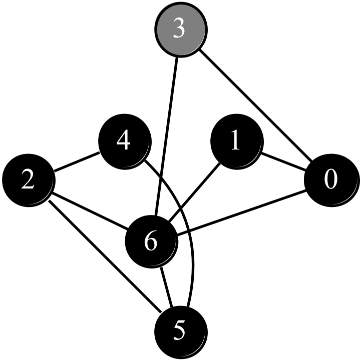
- Vertices are either unseen (white), visited (grey), or finished (black)
- When visiting a vertex u, if there is an edge to an unseen vertex v, follow that edge and visit v. v is now marked as visited.
- Continue recursively at v, if v has no unseen neighbors then v is finished.

DFS Details



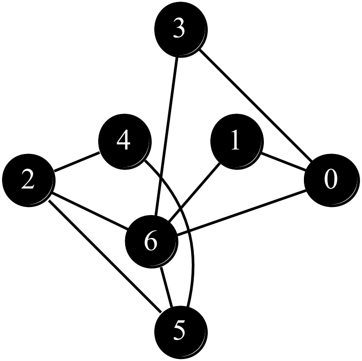
- Vertices are either unseen (white), visited (grey), or finished (black)
- When visiting a vertex u, if there is an edge to an unseen vertex v , follow that edge and visit v. v is now marked as visited.
- Continue recursively at v, if v has no unseen neighbors then v is finished.

DFS Details



- Vertices are either unseen (white), visited (grey), or finished (black)
- When visiting a vertex u, if there is an edge to an unseen vertex v , follow that edge and visit v . v is now marked as visited.
- Continue recursively at v , if v has no unseen neighbors then v is finished.

DFS Details

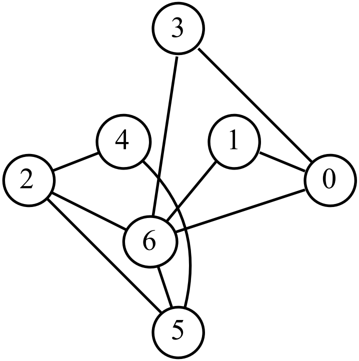


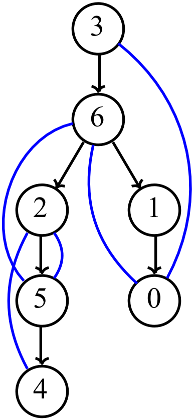
- Vertices are either unseen (white), visited (grey), or finished (black)
- When visiting a vertex u, if there is an edge to an unseen vertex v , follow that edge and visit v . v is now marked as visited.
- Continue recursively at v , if v has no unseen neighbors then v is finished.

DFS pseudocode

```
DFS-VISIT(v, G):
  \\adj(v) is the list of vertices adjacent to v
  v.color = grey
  for each u in adj(v):
    if u.color = white:
      DFS-VISIT(v)
  v.color = black

DFS(G):
  //V(G) is the list of vertices in a graph G
  for each u in V(G):
    if u.color = white
      DFS-VISIT(u, G)
```

Properties of DFS





tree edges: edges traversed by the DFS (gray to white)

back edges: between a vertex and an ancestor (gray to gray)

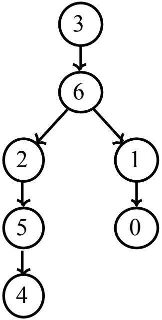
Video 3.4
Sampath Kannan

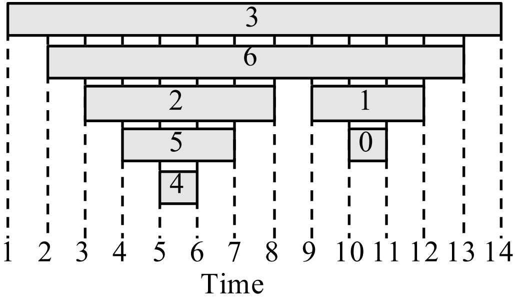
More Properties of DFS

- Imagine a counter (call it time) that advances each time a vertex is visited, or finished.
- For each vertex v there is a time s(v) when it is discovered and a time f(v) when it is finished.

More Properties of DFS

- Imagine a counter (call it time) that advances each time a vertex is visited, or finished.
- For each vertex v there is a time s(v) when it is discovered and a time f(v) when it is finished.
- Theorem: If u is a descendant of v in the DFS tree then $[s(u), f(u)] \subseteq [s(v), f(v)]$.
  In other words, u is seen after v is seen and finished before v finishes.





- **Theorem:** If DFS is at vertex v, and there is a path consisting entirely of unseen vertices from v to u, then u will be discovered and finished before dfs finishes at v

- **Theorem:** If DFS is at vertex v, and there is a path consisting entirely of unseen vertices from v to u, then u will be discovered and finished before dfs finishes at v
- **Proof Sketch:** Suppose for the sake of contradiction that v finishes and u has not been discovered. Let w be the the first unseen vertex on the path from v to u that remains unseen after v finishes. Consider the vertex $w^{/}$ right before w on the path is then discovered which implies that it is also finished before v finishes. Since w is unseen it would have been discovered when visitng $w^{/}$. Contradiction!

Applications of DFS

- Discovering the connected components of a graph. Each time we call DFS-VISIT we discover a new connected component.

  ```
  DFS(G):
    //V(G) is the list of vertices in a graph G
    for each u in V(G):
      if u.color = white
        DFS-VISIT(u, G)
  ```

  How could we modify this to output the connected component of each vertex?

Applications of DFS

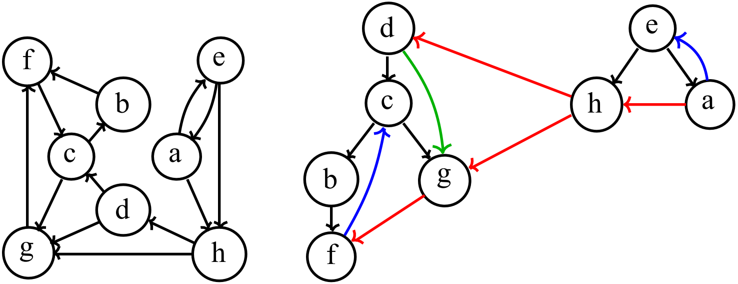
- Discovering the connected components of a graph. Each time we call DFS-VISIT we discover a new connected component.

```
DFS(G):
  //V(G) is the list of vertices in a graph G
  for each u in V(G):
    if u.color = white
      DFS-VISIT(u, G)
```

  How could we modify this to output the connected component of each vertex?

- Telling if a graph is acyclic. The presence of a back edge implies that the graph has cycles.
- Telling if a directed graph is acyclic (next slide).

DFS on Directed Graphs



tree edges: traversed by DFS

back edges: indicate cycles and go from a node to one of its ancestors

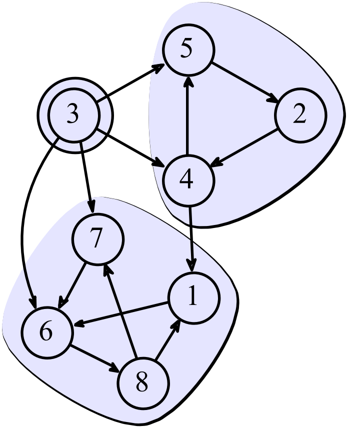
forward edges: go from a node to one of its descendants

cross edges: all other edges

Video 3.5

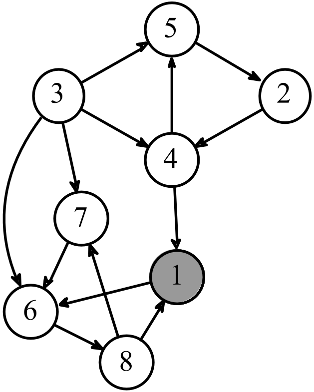
Sampath Kannan

Strongly Connected Components

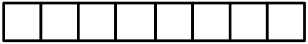


- Two vertices u and v are in the same SCC if u can reach v and v can reach u.
- This is an equivalence relation on vertices (recall reachability from undirected graphs).
  - Reflexive: obvious
  - Symmetric: by definition
  - Transitive: 

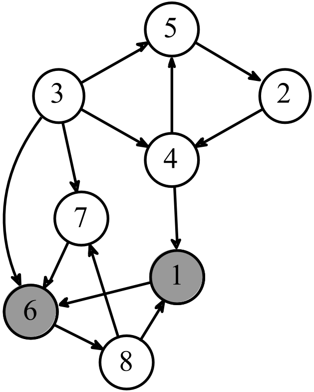
Kosaraju's Algorithm for SCCs



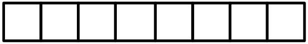
We will use DFS to create a list L which we will then use as an ordering for a second DFS



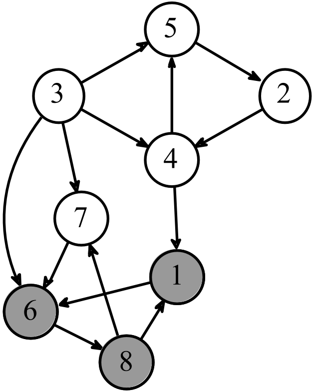
Kosaraju's Algorithm for SCCs



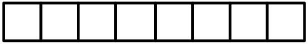
We will use DFS to create a list L which we will then use as an ordering for a second DFS



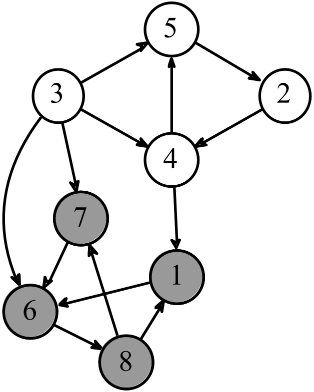
Kosaraju's Algorithm for SCCs



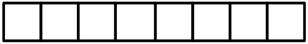
We will use DFS to create a list L which we will then use as an ordering for a second DFS



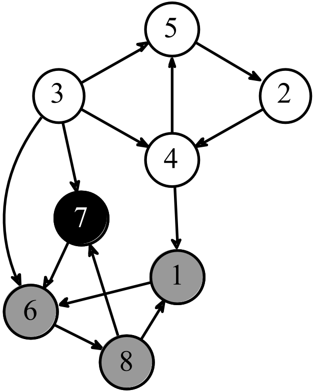
Kosaraju's Algorithm for SCCs



We will use DFS to create a list L which we will then use as an ordering for a second DFS



Kosaraju's Algorithm for SCCs

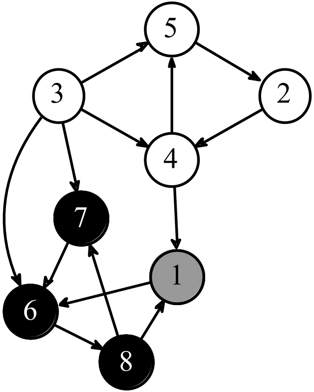


We will use DFS to create a list L which we will then use as an ordering for a second DFS

							7

Each time we finish a vertex we prepend it to L

Kosaraju's Algorithm for SCCs

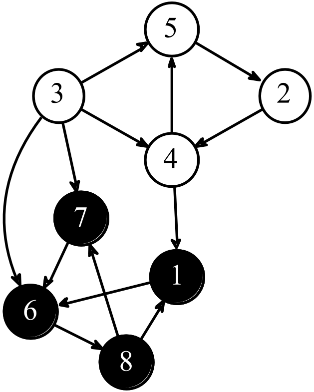


We will use DFS to create a list L which we will then use as an ordering for a second DFS

					6	8	7

Each time we finish a vertex we prepend it to L

Kosaraju's Algorithm for SCCs

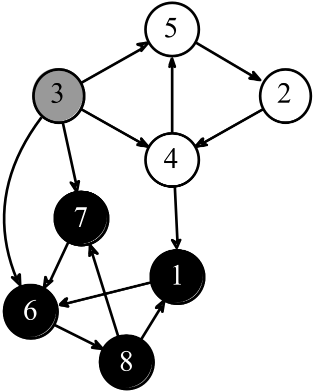


We will use DFS to create a list L which we will then use as an ordering for a second DFS

				1	6	8	7

Each time we finish a vertex we prepend it to L

Kosaraju's Algorithm for SCCs

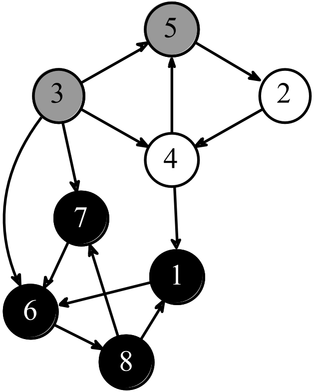


We will use DFS to create a list L which we will then use as an ordering for a second DFS

				1	6	8	7

Each time we finish a vertex we prepend it to L

Kosaraju's Algorithm for SCCs

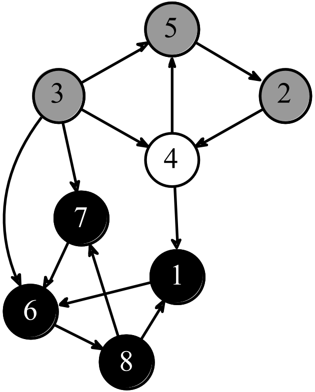


We will use DFS to create a list L which we will then use as an ordering for a second DFS

				1	6	8	7

Each time we finish a vertex we prepend it to L

Kosaraju's Algorithm for SCCs

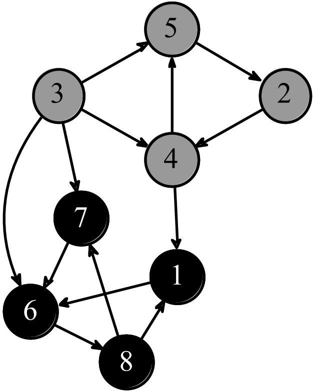


We will use DFS to create a list L which we will then use as an ordering for a second DFS

				1	6	8	7

Each time we finish a vertex we prepend it to L

Kosaraju's Algorithm for SCCs

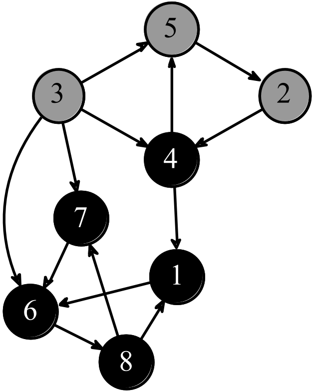


We will use DFS to create a list L which we will then use as an ordering for a second DFS

				1	6	8	7

Each time we finish a vertex we prepend it to L

Kosaraju's Algorithm for SCCs

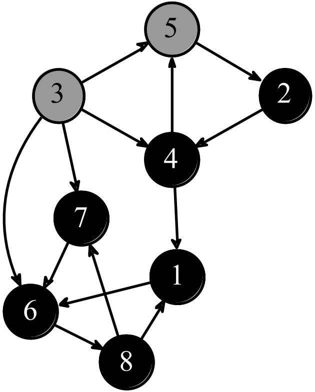


We will use DFS to create a list L which we will then use as an ordering for a second DFS

			4	1	6	8	7

Each time we finish a vertex we prepend it to L

Kosaraju's Algorithm for SCCs

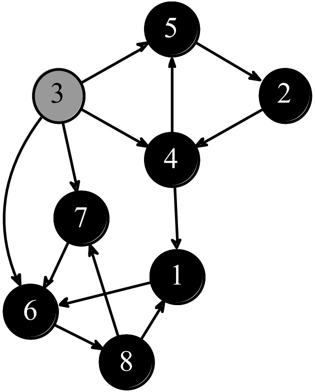


We will use DFS to create a list L which we will then use as an ordering for a second DFS

		2	4	1	6	8	7

Each time we finish a vertex we prepend it to L

Kosaraju's Algorithm for SCCs

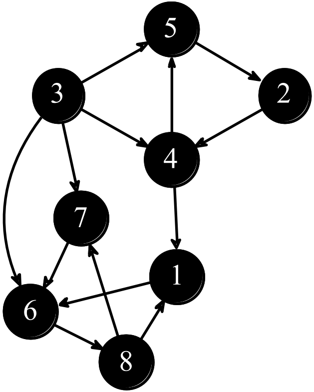


We will use DFS to create a list L which we will then use as an ordering for a second DFS

	5	2	4	1	6	8	7

Each time we finish a vertex we prepend it to L

Kosaraju's Algorithm for SCCs

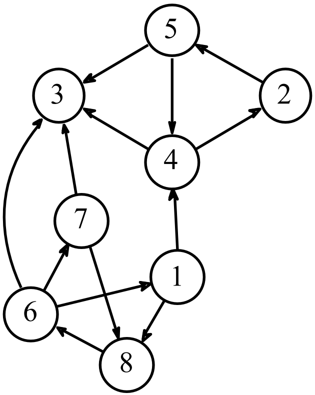


We will use DFS to create a list L which we will then use as an ordering for a second DFS

3	5	2	4	1	6	8	7

Each time we finish a vertex we prepend it to L

Kosaraju's Algorithm for SCCs



We will use DFS to create a list L which we will then use as an ordering for a second DFS

3	5	2	4	1	6	8	7

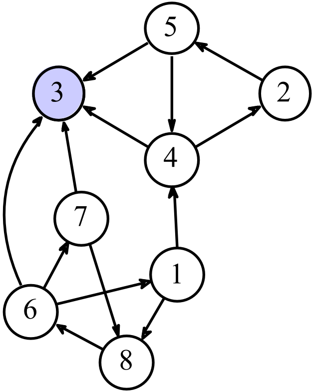
Each time we finish a vertex we prepend it to L

Now we reverse all the edges of the graph (the new graph is called the **transpose**)

We will run DFS on the transpose graph in the order specified by L.

Each time we start a new DFS-Visit we will create a new SCC

Kosaraju's Algorithm for SCCs



We will use DFS to create a list L which we will then use as an ordering for a second DFS

3	5	2	4	1	6	8	7

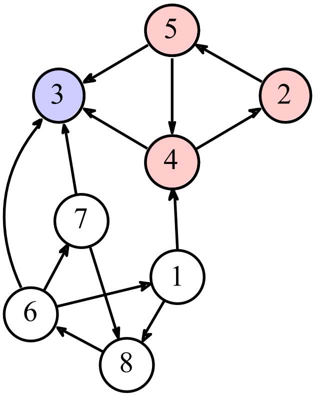
Each time we finish a vertex we prepend it to L

Now we reverse all the edges of the graph (the new graph is called the **transpose**)

We will run DFS on the transpose graph in the order specified by L.

Each time we start a new DFS-Visit we will create a new SCC

Kosaraju's Algorithm for SCCs



We will use DFS to create a list L which we will then use as an ordering for a second DFS

3	5	2	4	1	6	8	7

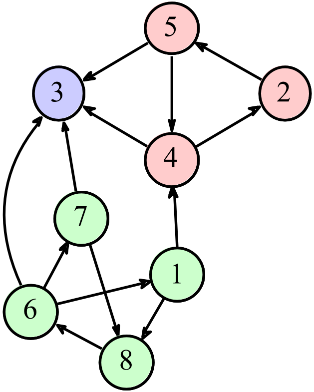
Each time we finish a vertex we prepend it to L

Now we reverse all the edges of the graph (the new graph is called the **transpose**)

We will run DFS on the transpose graph in the order specified by L.

Each time we start a new DFS-Visit we will create a new SCC

Kosaraju's Algorithm for SCCs



We will use DFS to create a list L which we will then use as an ordering for a second DFS

3	5	2	4	1	6	8	7

Each time we finish a vertex we prepend it to L

Now we reverse all the edges of the graph (the new graph is called the **transpose**)

We will run DFS on the transpose graph in the order specified by L.

Each time we start a new DFS-Visit we will create a new SCC

Kosaraju's Pseudocode

```
L=empty list

Visit(u,G):
  u.color=gray
  for each v in G.adj(u):
     if v.color=white:
        Visit(v)
  u.color=black
  L.prepend(u)

Kosaraju(G):
  for each u in V(G):
       if u.color=white then Visit(u,G)
  G'=transpose(G)
  SCC=1
  for each u in V(G')
     if u.assigned=false:
        Assign(u,G',SCC)
         SCC=SCC+1
```

```
Assign(u,G,SCC):
  u.scc=SCC
  u.assigned=true
  for each v in G.adj(u):
     if v.assigned=false:
        Assign(v,G,SCC)
```

Correctness of Kosaraju's

How do we know this algorithm always works?

- **Lemma**: If u comes after v in L then either there is a path from v to u or there is no path from u to v.
  If v comes before u in L then v must have a greater finish time than u. Therefore either u is discovered and finished before v is discovered, or u is discovered after v but still finishes first (recall the theorem from the previous segment). In the first case there can't be a path from u to v since otherwise v would be discovered before finishing u and in the second case we have by construction that there is a path from v to u.

Correctness of Kosaraju's

How do we know this algorithm always works?

- In the second DFS we assign each discovered vertex to the same SCC as the root. We need to make sure each of these assignments is correct.

Correctness of Kosaraju's

How do we know this algorithm always works?

- In the second DFS we assign each discovered vertex to the same SCC as the root. We need to make sure each of these assignments is correct.
- When we assign u to the same SCC as the root v we have by definition found a path from u to v in the original graph (remember we flipped all the edges). So all that's left to check is that there is a path from v to u as well.

Correctness of Kosaraju's

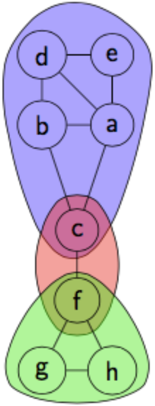
How do we know this algorithm always works?

- In the second DFS we assign each discovered vertex to the same SCC as the root. We need to make sure each of these assignments is correct.
- When we assign u to the same SCC as the root v we have by definition found a path from u to v in the original graph (remember we flipped all the edges). So all that's left to check is that there is a path from v to u as well.
- Since u is undiscovered when we visit v, u must come after v in L. The lemma we proved then implies that either is no path from u to v or there is a path from v to u. We already know that there is a path from u to v so it must be the case that there is a path form v to u

Video 3.6

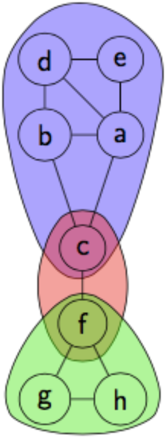
Sampath Kannan

Biconnected Components



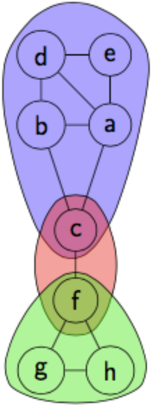
- A connected graph is **biconnected** if removing any one vertex still leaves it connected.
- A **biconnected component** is a maximal subset of the graph that is biconnected.

Biconnected Components



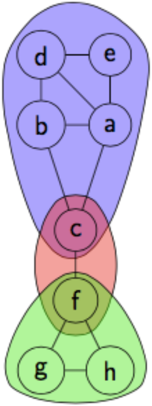
- A connected graph is **biconnected** if removing any one vertex still leaves it connected.
- A **biconnected component** is a maximal subset of the graph that is biconnected.
- A vertex can be in more than one component!

Biconnected Components



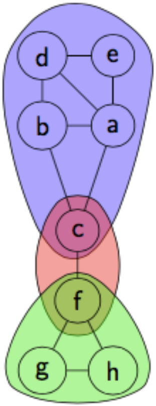
- A connected graph is **biconnected** if removing any one vertex still leaves it connected.
- A **biconnected component** is a maximal subset of the graph that is biconnected.
- A vertex can be in more than one component!
- Equivalence relation on edges: $(u, v) \equiv (r, s)$ if they are in the same component. Two edges are in the same component iff there is a simple cycle containing them. Also $(u, v) \equiv (u, v)$ always.

Biconnected Components



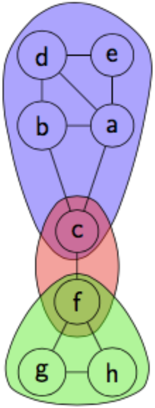
- A connected graph is **biconnected** if removing any one vertex still leaves it connected.
- A **biconnected component** is a maximal subset of the graph that is biconnected.
- A vertex can be in more than one component!
- Equivalence relation on edges: $(u, v) \equiv (r, s)$ if they are in the same component. Two edges are in the same component iff there is a simple cycle containing them. Also $(u, v) \equiv (u, v)$ always.
  - Reflexive and Symmetric: Trivially true

Biconnected Components

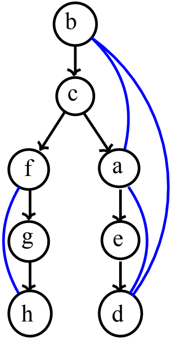


- A connected graph is **biconnected** if removing any one vertex still leaves it connected.
- A **biconnected component** is a maximal subset of the graph that is biconnected.
- A vertex can be in more than one component!
- Equivalence relation on edges: $(u, v) \equiv (r, s)$ if they are in the same component. Two edges are in the same component iff there is a simple cycle containing them. Also $(u, v) \equiv (u, v)$ always.
  - Reflexive and Symmetric: Trivially true
  - Transitive: Let $e_1 = (u, v)$, $e_2 = (r, s)$, $e_3 = (x, y)$. If $e_1 \equiv e_2$ and $e_2 \equiv e_3$ then there is a simple cycle containing e_1 and e_2 as well as one containing e_2 and e_3. We construct another cycle by going from e_1 around the first cycle to an endpoint of e_2, then we use the second cycle to go to the other endpoint of e_2 and then finally we can traverse the rest of the first cycle back to e_1. We can then contract this into a simple cycle.

Articulation Points



A DFS tree



- An **articulation point** is a vertex whose removal disconnects the graph (c and f in this example).
- In the DFS tree we see that an internal vertex v is an articulation point when its removal completely disconnects at least one of its children's subtree from the graph. If v is the root then it is an articulation point when it has more than one child.

Idea of the Algorithm

- Since DFS on undirected graphs produces only tree and back edges, a subtree T_i rooted at a child v_i of v will be disconnected by removing v when there are no vertices in T with back edges that reach "above" v.

Idea of the Algorithm

- Since DFS on undirected graphs produces only tree and back edges, a subtree T_i rooted at a child v_i of v will be disconnected by removing v when there are no vertices in T with back edges that reach "above" v.
- Give DFS-number to each vertex in the order we discover them (i.e. root is 1, first vertex discovered from root is 2, etc). Observe that ancestors always have smaller dfs numbers than descendants.

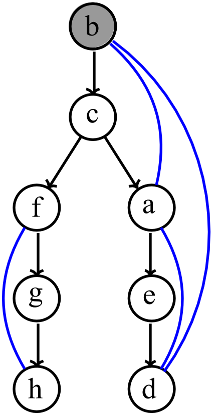
Idea of the Algorithm

- Since DFS on undirected graphs produces only tree and back edges, a subtree T_i rooted at a child v_i of v will be disconnected by removing v when there are no vertices in T with back edges that reach "above" v.
- Give DFS-number to each vertex in the order we discover them (i.e. root is 1, first vertex discovered from root is 2, etc). Observe that ancestors always have smaller dfs numbers than descendants.
- For each vertex v and each subtree T_i rooted at a child v_i of v, compute the smallest DFS-number reachable by a back edge from a vertex in T_i. Call this quantity $low(v_i)$. If $low(v_i) \geq$ DFS-number(v) then v is an articulation point.

Algorithm and Pseudocode

```
discover-count = 1

articulation-vertices(G):
  for u in G.V:
    if u.color = white:
      articulation-vertex-visit(u, G)
      if u.num-children > 1:
         v.ap = true
      else:
         v.ap = false

articulation-vertex-visit(u, G):
  dfs-number[u] = discover-count
  discover-count = discover-count + 1
  low[u] = dfs-number[u]
  u.color = gray
  for each v in G.adj(u):
    if v.color = white:
      articulation-vertex-visit(v, G)
      u.num-children = u.num-children + 1
      if low[v] < low[u]:
        low[u] = low[v]
      if low[v] >= dfs-number[u]
        u.ap = true
     if dfs-number[v] < low[u]:
      low[u] = dfs-number[v]
  u.color = black
```

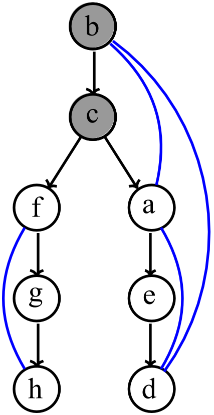
	a	b	c	d	e	f	g	h
low								
dfs-number		1						



Algorithm and Pseudocode

	a	b	c	d	e	f	g	h
low								
dfs-number		1	2					

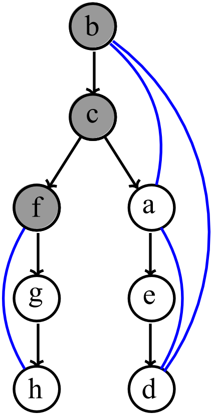
```
discover-count = 1

articulation-vertices(G):
  for u in G.V:
    if u.color = white:
      articulation-vertex-visit(u, G)
      if u.num-children > 1:
         v.ap = true
      else:
         v.ap = false

articulation-vertex-visit(u, G):
  dfs-number[u] = discover-count
  discover-count = discover-count + 1
  low[u] = dfs-number[u]
  u.color = gray
  for each v in G.adj(u):
    if v.color = white:
      articulation-vertex-visit(v, G)
      u.num-children = u.num-children + 1
      if low[v] < low[u]:
        low[u] = low[v]
      if low[v] >= dfs-number[u]
        u.ap = true
    if dfs-number[v] < low[u]:
      low[u] = dfs-number[v]
  u.color = black
```



Algorithm and Pseudocode

```
discover-count = 1

articulation-vertices(G):
 for u in G.V:
   if u.color = white:
     articulation-vertex-visit(u, G)
     if u.num-children > 1:
        v.ap = true
     else:
        v.ap = false

articulation-vertex-visit(u, G):
 dfs-number[u] = discover-count
 discover-count = discover-count + 1
 low[u] = dfs-number[u]
 u.color = gray
 for each v in G.adj(u):
   if v.color = white:
     articulation-vertex-visit(v, G)
     u.num-children = u.num-children + 1
     if low[v] < low[u]:
       low[u] = low[v]
     if low[v] >= dfs-number[u]
       u.ap = true
   if dfs-number[v] < low[u]:
     low[u] = dfs-number[v]
 u.color = black
```

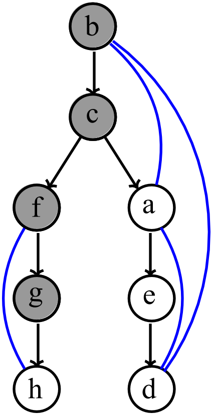
	a	b	c	d	e	f	g	h
low								
dfs-number		1	2			3		



Algorithm and Pseudocode

```
discover-count = 1

articulation-vertices(G):
 for u in G.V:
   if u.color = white:
     articulation-vertex-visit(u, G)
     if u.num-children > 1:
        v.ap = true
     else:
        v.ap = false

articulation-vertex-visit(u, G):
 dfs-number[u] = discover-count
 discover-count = discover-count + 1
 low[u] = dfs-number[u]
 u.color = gray
 for each v in G.adj(u):
   if v.color = white:
     articulation-vertex-visit(v, G)
     u.num-children = u.num-children + 1
     if low[v] < low[u]:
       low[u] = low[v]
     if low[v] >= dfs-number[u]
       u.ap = true
    if dfs-number[v] < low[u]:
     low[u] = dfs-number[v]
 u.color = black
```

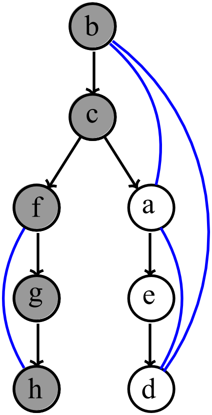
	a	b	c	d	e	f	g	h
low								
dfs-number		1	2			3	4	



Algorithm and Pseudocode

```
discover-count = 1

articulation-vertices(G):
 for u in G.V:
   if u.color = white:
     articulation-vertex-visit(u, G)
     if u.num-children > 1:
        v.ap = true
     else:
        v.ap = false

articulation-vertex-visit(u, G):
 dfs-number[u] = discover-count
 discover-count = discover-count + 1
 low[u] = dfs-number[u]
 u.color = gray
 for each v in G.adj(u):
   if v.color = white:
     articulation-vertex-visit(v, G)
     u.num-children = u.num-children + 1
     if low[v] < low[u]:
       low[u] = low[v]
     if low[v] >= dfs-number[u]
       u.ap = true
    if dfs-number[v] < low[u]:
     low[u] = dfs-number[v]
 u.color = black
```

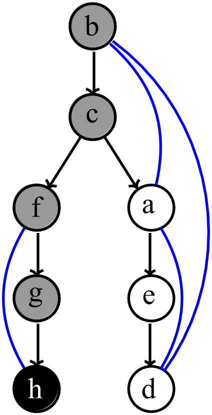
	a	b	c	d	e	f	g	h
low								
dfs-number		1	2			3	4	5



Algorithm and Pseudocode

	a	b	c	d	e	f	g	h
low								3
dfs-number		1	2			3	4	5

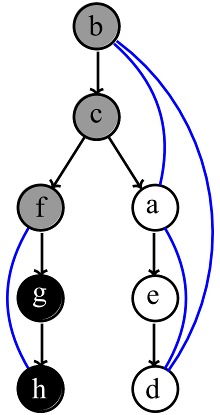
```
discover-count  = 1

articulation-vertices(G):
  for u  in G.V:
    if u.color  = white:
      articulation-vertex-visit(u, G)
      if u.num-children  > 1:
         v.ap  = true
      else:
         v.ap  = false

articulation-vertex-visit(u, G):
  dfs-number[u]  = discover-count
  discover-count  =  discover-count  + 1
  low[u]  = dfs-number[u]
  u.color  = gray
  for each  v  in  G.adj(u):
    if v.color  = white:
      articulation-vertex-visit(v, G)
      u.num-children  = u.num-children  + 1
      if low[v]  < low[u]:
        low[u]  = low[v]
      if low[v]  >= dfs-number[u]
        u.ap  = true
    if dfs-number[v]  < low[u]:
      low[u]  = dfs-number[v]
  u.color  = black
```



Algorithm and Pseudocode

```
discover-count = 1

articulation-vertices(G):
 for u in G.V:
   if u.color = white:
     articulation-vertex-visit(u, G)
     if u.num-children > 1:
        v.ap = true
     else:
        v.ap = false

articulation-vertex-visit(u, G):
 dfs-number[u] = discover-count
 discover-count = discover-count + 1
 low[u] = dfs-number[u]
 u.color = gray
 for each v in G.adj(u):
   if v.color = white:
     articulation-vertex-visit(v, G)
     u.num-children = u.num-children + 1
     if low[v] < low[u]:
       low[u] = low[v]
     if low[v] >= dfs-number[u]
       u.ap = true
   if dfs-number[v] < low[u]:
     low[u] = dfs-number[v]
 u.color = black
```

	a	b	c	d	e	f	g	h
low							3	3
dfs-number		1	2			3	4	5

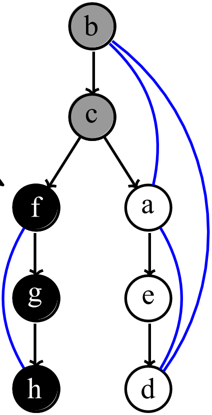


Algorithm and Pseudocode

```
discover-count  = 1

 articulation-vertices(G):
  for u  in G.V:
    if u.color  = white:
      articulation-vertex-visit(u, G)
      if u.num-children  > 1:
         v.ap  = true
      else:
         v.ap  = false

 articulation-vertex-visit(u, G):
  dfs-number[u]  = discover-count
  discover-count  =  discover-count  + 1
  low[u]  = dfs-number[u]
  u.color  = gray
  for each  v  in  G.adj(u):
    if v.color  = white:
      articulation-vertex-visit(v, G)
      u.num-children  =  u.num-children  + 1
      if low[v]  < low[u]:
        low[u]  = low[v]
      if low[v]  >= dfs-number[u]
        u.ap  = true
    if dfs-number[v]  < low[u]:
      low[u]  = dfs-number[v]
  u.color  = black
```

	a	b	c	d	e	f	g	h
low						3	3	3
dfs-number		1	2			3	4	5

Articulation point!

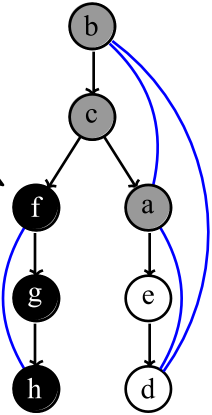


Algorithm and Pseudocode

```
discover-count = 1

articulation-vertices(G):
  for u in G.V:
    if u.color = white:
      articulation-vertex-visit(u, G)
      if u.num-children > 1:
         v.ap = true
      else:
         v.ap = false

articulation-vertex-visit(u, G):
  dfs-number[u] = discover-count
  discover-count = discover-count + 1
  low[u] = dfs-number[u]
  u.color = gray
  for each v in G.adj(u):
    if v.color = white:
      articulation-vertex-visit(v, G)
      u.num-children = u.num-children + 1
      if low[v] < low[u]:
        low[u] = low[v]
      if low[v] >= dfs-number[u]
        u.ap = true
    if dfs-number[v] < low[u]:
      low[u] = dfs-number[v]
  u.color = black
```

	a	b	c	d	e	f	g	h
low						3	3	3
dfs-number	6	1	2			3	4	5

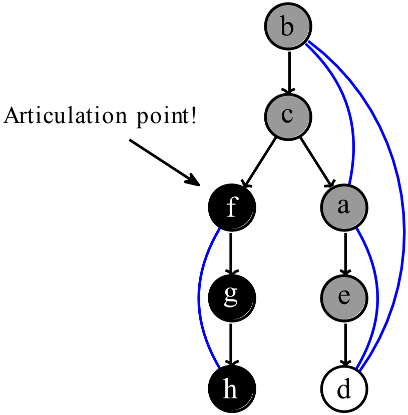
Articulation point!



Algorithm and Pseudocode

```
discover-count  = 1

 articulation-vertices(G):
  for u  in G.V:
    if u.color  = white:
      articulation-vertex-visit(u, G)
      if u.num-children  > 1:
         v.ap  = true
      else:
         v.ap  = false

 articulation-vertex-visit(u, G):
  dfs-number[u]  = discover-count
  discover-count  = discover-count  + 1
  low[u]  = dfs-number[u]
  u.color  = gray
  for each  v  in  G.adj(u):
    if v.color  = white:
      articulation-vertex-visit(v, G)
      u.num-children  = u.num-children  + 1
      if low[v]  < low[u]:
        low[u]  = low[v]
      if low[v]  >= dfs-number[u]
        u.ap  = true
     if dfs-number[v]  < low[u]:
      low[u]  = dfs-number[v]
  u.color  = black
```

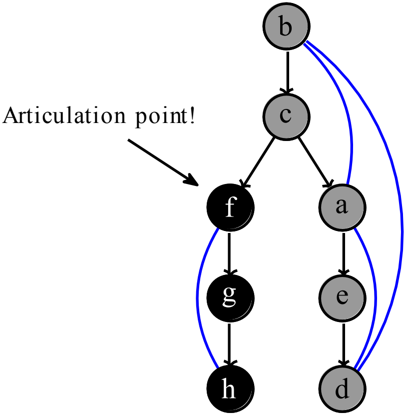
	a	b	c	d	e	f	g	h
low						3	3	3
dfs-number	6	1	2		7	3	4	5



Algorithm and Pseudocode

```
discover-count = 1

articulation-vertices(G):
  for u in G.V:
    if u.color = white:
      articulation-vertex-visit(u, G)
      if u.num-children > 1:
         v.ap = true
      else:
         v.ap = false

articulation-vertex-visit(u, G):
  dfs-number[u] = discover-count
  discover-count = discover-count + 1
  low[u] = dfs-number[u]
  u.color = gray
  for each v in G.adj(u):
    if v.color = white:
      articulation-vertex-visit(v, G)
      u.num-children = u.num-children + 1
      if low[v] < low[u]:
        low[u] = low[v]
      if low[v] >= dfs-number[u]
        u.ap = true
    if dfs-number[v] < low[u]:
      low[u] = dfs-number[v]
  u.color = black
```

	a	b	c	d	e	f	g	h
low						3	3	3
dfs-number	6	1	2	8	7	3	4	5

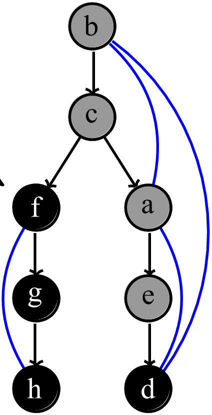


Algorithm and Pseudocode

```
discover-count = 1

articulation-vertices(G):
 for u in G.V:
   if u.color = white:
     articulation-vertex-visit(u, G)
     if u.num-children > 1:
        v.ap = true
     else:
        v.ap = false

articulation-vertex-visit(u, G):
 dfs-number[u] = discover-count
 discover-count = discover-count + 1
 low[u] = dfs-number[u]
 u.color = gray
 for each v in G.adj(u):
   if v.color = white:
     articulation-vertex-visit(v, G)
     u.num-children = u.num-children + 1
     if low[v] < low[u]:
       low[u] = low[v]
     if low[v] >= dfs-number[u]
       u.ap = true
    if dfs-number[v] < low[u]:
     low[u] = dfs-number[v]
 u.color = black
```

	a	b	c	d	e	f	g	h
low				1		3	3	3
dfs-number	6	1	2	8	7	3	4	5

Articulation point!

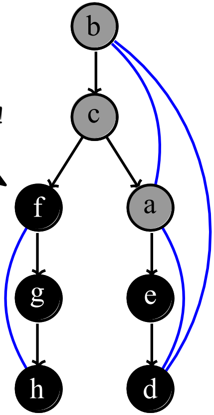


Algorithm and Pseudocode

```
discover-count = 1

articulation-vertices(G):
 for u in G.V:
   if u.color = white:
     articulation-vertex-visit(u, G)
     if u.num-children > 1:
        v.ap = true
     else:
        v.ap = false

articulation-vertex-visit(u, G):
 dfs-number[u] = discover-count
 discover-count = discover-count + 1
 low[u] = dfs-number[u]
 u.color = gray
 for each v in G.adj(u):
   if v.color = white:
     articulation-vertex-visit(v, G)
     u.num-children = u.num-children + 1
     if low[v] < low[u]:
       low[u] = low[v]
     if low[v] >= dfs-number[u]
       u.ap = true
    if dfs-number[v] < low[u]:
     low[u] = dfs-number[v]
 u.color = black
```

	a	b	c	d	e	f	g	h
low				1	1	3	3	3
dfs-number	6	1	2	8	7	3	4	5

Articulation point!

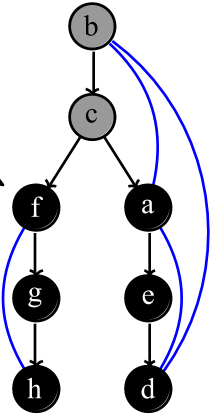


Algorithm and Pseudocode

```
discover-count = 1

articulation-vertices(G):
  for u in G.V:
     if u.color = white:
        articulation-vertex-visit(u, G)
        if u.num-children > 1:
           v.ap = true
        else:
           v.ap = false

articulation-vertex-visit(u, G):
  dfs-number[u] = discover-count
  discover-count = discover-count + 1
  low[u] = dfs-number[u]
  u.color = gray
  for each v in G.adj(u):
     if v.color = white:
        articulation-vertex-visit(v, G)
        u.num-children = u.num-children + 1
        if low[v] < low[u]:
           low[u] = low[v]
        if low[v] >= dfs-number[u]
           u.ap = true
      if dfs-number[v] < low[u]:
        low[u] = dfs-number[v]
  u.color = black
```

	a	b	c	d	e	f	g	h
low	1			1	1	3	3	3
dfs-number	6	1	2	8	7	3	4	5

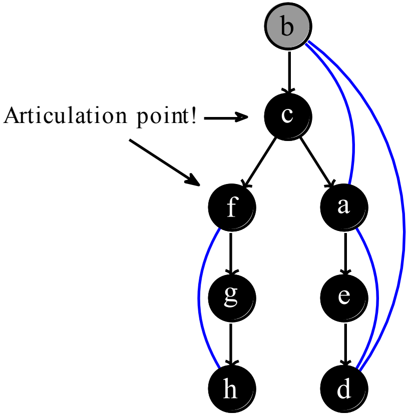
Articulation point!



Algorithm and Pseudocode

	a	b	c	d	e	f	g	h
low	1		1	1	1	3	3	3
dfs-number	6	1	2	8	7	3	4	5

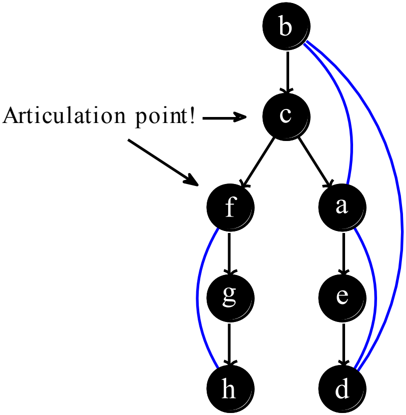
```
discover-count  = 1

 articulation-vertices(G):
  for u  in G.V:
    if u.color  = white:
      articulation-vertex-visit(u, G)
      if u.num-children  > 1:
         v.ap  = true
      else:
         v.ap  = false

 articulation-vertex-visit(u, G):
  dfs-number[u]  = discover-count
  discover-count  = discover-count  + 1
  low[u]  = dfs-number[u]
  u.color  = gray
  for each  v  in  G.adj(u):
    if v.color  = white:
      articulation-vertex-visit(v, G)
      u.num-children  = u.num-children  + 1
      if low[v]  < low[u]:
        low[u]  = low[v]
      if low[v]  >= dfs-number[u]
        u.ap  = true
    if dfs-number[v]  < low[u]:
      low[u]  = dfs-number[v]
  u.color  = black
```



Algorithm and Pseudocode

```
discover-count = 1

articulation-vertices(G):
 for u in G.V:
   if u.color = white:
     articulation-vertex-visit(u, G)
     if u.num-children > 1:
        v.ap = true
     else:
        v.ap = false

articulation-vertex-visit(u, G):
 dfs-number[u] = discover-count
 discover-count = discover-count + 1
 low[u] = dfs-number[u]
 u.color = gray
 for each v in G.adj(u):
   if v.color = white:
     articulation-vertex-visit(v, G)
     u.num-children = u.num-children + 1
     if low[v] < low[u]:
       low[u] = low[v]
     if low[v] >= dfs-number[u]
       u.ap = true
    if dfs-number[v] < low[u]:
     low[u] = dfs-number[v]
 u.color = black
```

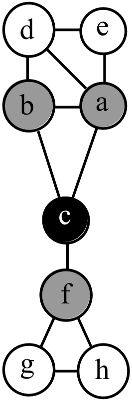
	a	b	c	d	e	f	g	h
low	1	1	1	1	1	3	3	3
dfs-number	6	1	2	8	7	3	4	5



Penn Engineering
ONLINE LEARNING

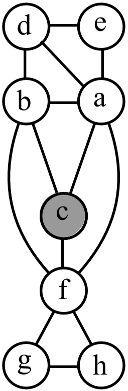
Video 3.7
Sampath Kannan

Breadth first search



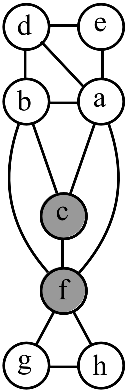
- Vertices will be unseen (white), visited (gray), or finished(black)
- **Key idea**: If a vertex v is visited before vertex u, v will be finished before u. Use a queue to accomplish this
  - visiting = inserted into the queue
  - finishing = adding all unseen neighbors to the queue

Example



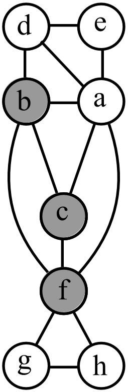

```
BFS(G, s):
  Q = empty queue
  push(Q, s)
  s.color = gray
  while not empty(Q):
    u = pop(Q)
    for each v in G.adj(u):
      if v.color = white
        v.color = gray
        push(Q, v)
    u.color = black
```

Example



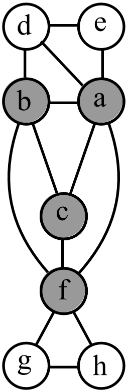

```
BFS(G, s):
  Q = empty queue
  push(Q, s)
  s.color = gray
  while not empty(Q):
    u = pop(Q)
    for each v in G.adj(u):
      if v.color = white
        v.color = gray
        push(Q, v)
    u.color = black
```

Example



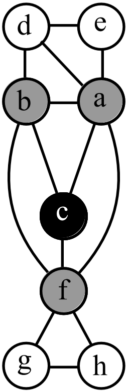

```
BFS(G, s):
  Q = empty queue
  push(Q, s)
  s.color = gray
  while not empty(Q):
    u = pop(Q)
    for each v in G.adj(u):
      if v.color = white
        v.color = gray
        push(Q, v)
    u.color = black
```

Example



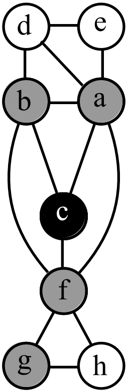

```
BFS(G, s):
  Q = empty queue
  push(Q, s)
  s.color = gray
  while not empty(Q):
    u = pop(Q)
    for each v in G.adj(u):
      if v.color = white
        v.color = gray
        push(Q, v)
    u.color = black
```

Example



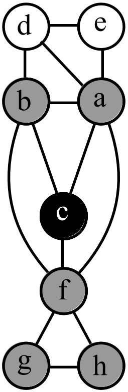

```
BFS(G, s):
  Q = empty queue
  push(Q, s)
  s.color = gray
  while not empty(Q):
    u = pop(Q)
    for each v in G.adj(u):
      if v.color = white
        v.color = gray
        push(Q, v)
    u.color = black
```

Example



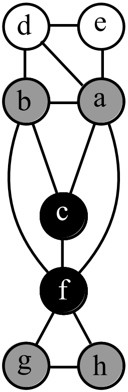

```
BFS(G, s):
  Q = empty queue
  push(Q, s)
  s.color = gray
  while not empty(Q):
    u = pop(Q)
    for each v in G.adj(u):
      if v.color = white
        v.color = gray
        push(Q, v)
    u.color = black
```

Example



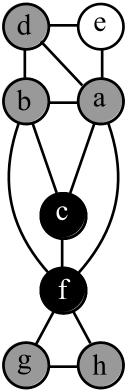

```
BFS(G, s):
  Q = empty queue
  push(Q, s)
  s.color = gray
  while not empty(Q):
    u = pop(Q)
    for each v in G.adj(u):
      if v.color = white
        v.color = gray
        push(Q, v)
    u.color = black
```

Example



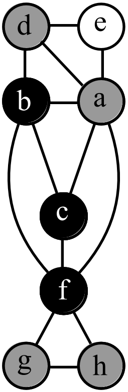

```
BFS(G, s):
  Q = empty queue
  push(Q, s)
  s.color = gray
  while not empty(Q):
    u = pop(Q)
    for each v in G.adj(u):
      if v.color = white
        v.color = gray
        push(Q, v)
    u.color = black
```

Example



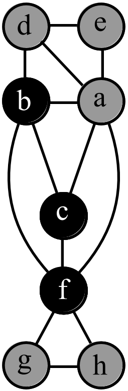

```
BFS(G, s):
  Q = empty queue
  push(Q, s)
  s.color = gray
  while not empty(Q):
    u = pop(Q)
    for each v in G.adj(u):
      if v.color = white
        v.color = gray
        push(Q, v)
    u.color = black
```

Example



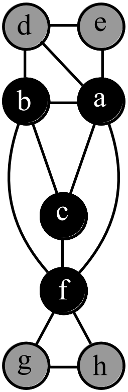

```
BFS(G, s):
  Q = empty queue
  push(Q, s)
  s.color = gray
  while not empty(Q):
    u = pop(Q)
    for each v in G.adj(u):
      if v.color = white
        v.color = gray
        push(Q, v)
    u.color = black
```

Example



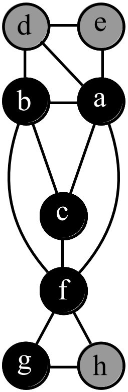

```
BFS(G, s):
  Q = empty queue
  push(Q, s)
  s.color = gray
  while not empty(Q):
    u = pop(Q)
    for each v in G.adj(u):
      if v.color = white
        v.color = gray
        push(Q, v)
    u.color = black
```

Example



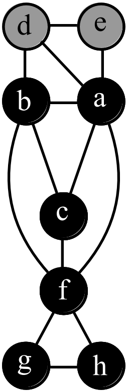

```
BFS(G, s):
  Q = empty queue
  push(Q, s)
  s.color = gray
  while not empty(Q):
    u = pop(Q)
    for each v in G.adj(u):
      if v.color = white
        v.color = gray
        push(Q, v)
    u.color = black
```

Example



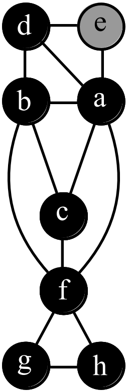

```
BFS(G, s):
  Q = empty queue
  push(Q, s)
  s.color = gray
  while not empty(Q):
    u = pop(Q)
    for each v in G.adj(u):
      if v.color = white
        v.color = gray
        push(Q, v)
    u.color = black
```

Example



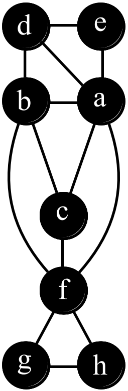

```
BFS(G, s):
  Q = empty queue
  push(Q, s)
  s.color = gray
  while not empty(Q):
    u = pop(Q)
    for each v in G.adj(u):
      if v.color = white
        v.color = gray
        push(Q, v)
    u.color = black
```

Example



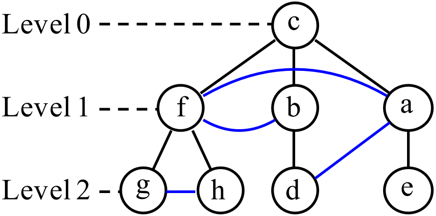

```
BFS(G, s):
  Q = empty queue
  push(Q, s)
  s.color = gray
  while not empty(Q):
    u = pop(Q)
    for each v in G.adj(u):
      if v.color = white
        v.color = gray
        push(Q, v)
    u.color = black
```

Example



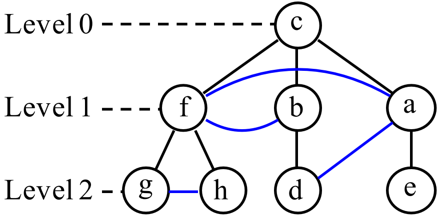

```
BFS(G, s):
  Q = empty queue
  push(Q, s)
  s.color = gray
  while not empty(Q):
    u = pop(Q)
    for each v in G.adj(u):
      if v.color = white
        v.color = gray
        push(Q, v)
    u.color = black
```

Properties of BFS



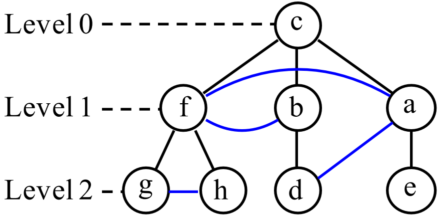
- If u is enqueued while exploring v then level(u) = 1 + level(v).

Properties of BFS



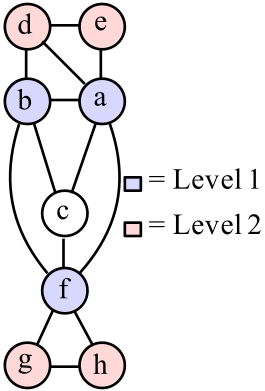
- If u is enqueued while exploring v then level(u) = 1 + level(v).
- All non-tree edges are between vertices whose levels differ by at most 1.

Properties of BFS



- If u is enqueued while exploring v then level(u) = 1 + level(v).
- All non-tree edges are between vertices whose levels differ by at most 1.
- All vertices in a level are explored consecutively. Vertices are explored in order of levels.

More Properties of BFS

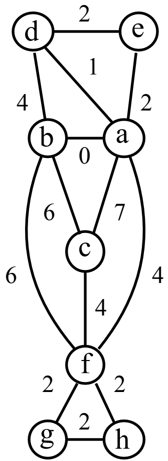


- level(u) is the **distance** from u to v (the starting vertex).
- Distance is the number of edges on the shortest path from v to u.
- One of the key applications of BFS is to find shortest paths to all vertices from a source vertex.

Video 3.8

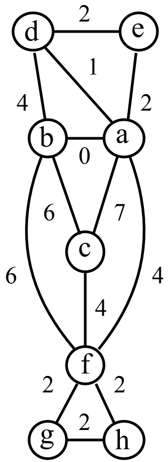
Sampath Kannan

Weighted Graphs



- In a weighted graph each edge has a number on it.
  - Number could be length, cost, time, . . .
  - Building roads? Number could be cost of building
  - Sending data? Number could be time to traverse a link
  - Exploring graph? Number could be the length of an edge

Weighted Graphs



- In a weighted graph each edge has a number on it.
  - Number could be length, cost, time, . . .
  - Building roads? Number could be cost of building
  - Sending data? Number could be time to traverse a link
  - Exploring graph? Number could be the length of an edge
- By default numbers are assumed non-negative, but sometimes they could be negative

Weighted Graph Notation

- A weighted graph is represented as $G = (V, E, w)$ where w is a mapping form edges to numbers
- $w(e)$ is the weight of edge e

Weighted Graph Notation

- A weighted graph is represented as $G = (V, E, w)$ where w is a mapping form edges to numbers
- $w(e)$ is the weight of edge e
- Question: How do we connect all the vertices with the least cost (or length or time)?

  Weight of a set of edges = sum of the weights of the edges in the set.

Weighted Graph Notation

- A weighted graph is represented as G = (V , E , w) where w is a mapping form edges to numbers
- w(e) is the weight of edge e
- Question: How do we connect all the vertices with the least cost (or length or time)?

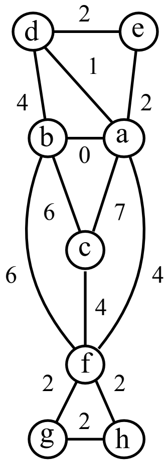
  Weight of a set of edges = sum of the weights of the edges in the set.
- Trees are minimal connected graphs, so we want to find a tree
- A tree that connects all the vertices of a graph is called a **spanning tree**.

Weighted Graph Notation

- A weighted graph is represented as $G = (V, E, w)$ where w is a mapping form edges to numbers
- $w(e)$ is the weight of edge e
- Question: How do we connect all the vertices with the least cost (or length or time)?

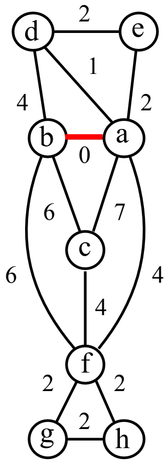
  Weight of a set of edges = sum of the weights of the edges in the set.
- Trees are minimal connected graphs, so we want to find a tree
- A tree that connects all the vertices of a graph is called a **spanning tree**.
- So the problem is to find a **minimum spanning tree** (MST).

Kruskal's Algorithm



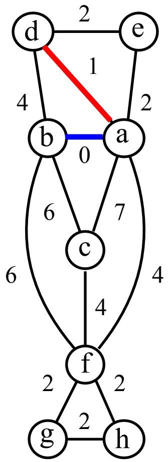
- Kruskal's algorithm is a greedy algorithm for finding a MST
- 1) Sort the edges in increasing order of weight, $e_1, e_2, \ldots, e_m$.
- 2) Initialize solution to empty set
- 3) For each i from 1 to m, if adding e_i to the solution won't create a cycle, add it.

Kruskal's Algorithm



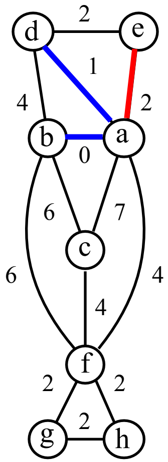
- Kruskal's algorithm is a greedy algorithm for finding a MST
- 1) Sort the edges in increasing order of weight, $e_1, e_2, \ldots, e_m$.
- 2) Initialize solution to empty set
- 3) For each i from 1 to m, if adding e_i to the solution won't create a cycle, add it.

Kruskal's Algorithm



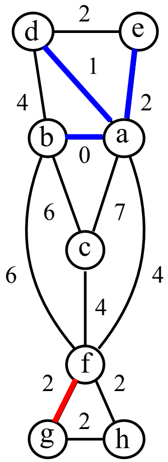
- Kruskal's algorithm is a greedy algorithm for finding a MST
- 1) Sort the edges in increasing order of weight, $e_1, e_2, \ldots, e_m$.
- 2) Initialize solution to empty set
- 3) For each i from 1 to m, if adding e_i to the solution won't create a cycle, add it.

Kruskal's Algorithm



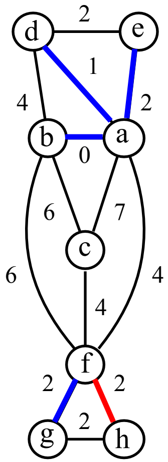
- Kruskal's algorithm is a greedy algorithm for finding a MST
- 1) Sort the edges in increasing order of weight, $e_1, e_2, \ldots, e_m$.
- 2) Initialize solution to empty set
- 3) For each i from 1 to m, if adding e_i to the solution won't create a cycle, add it.

Kruskal's Algorithm



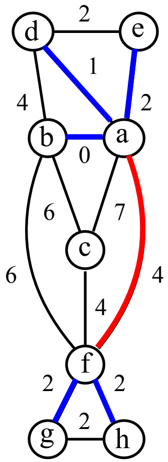
- Kruskal's algorithm is a greedy algorithm for finding a MST
- 1) Sort the edges in increasing order of weight, $e_1, e_2, \ldots, e_m$.
- 2) Initialize solution to empty set
- 3) For each i from 1 to m, if adding e_i to the solution won't create a cycle, add it.

Kruskal's Algorithm



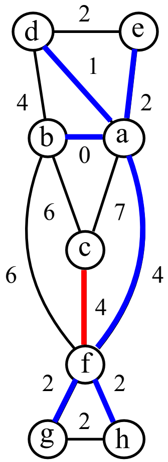
- Kruskal's algorithm is a greedy algorithm for finding a MST
- 1) Sort the edges in increasing order of weight, $e_1, e_2, \ldots, e_m$.
- 2) Initialize solution to empty set
- 3) For each i from 1 to m, if adding e_i to the solution won't create a cycle, add it.

Kruskal's Algorithm



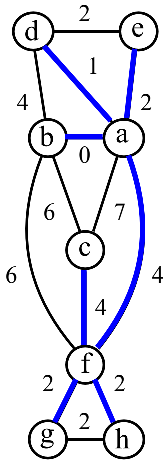
- Kruskal's algorithm is a greedy algorithm for finding a MST
- 1) Sort the edges in increasing order of weight, $e_1, e_2, \ldots, e_m$.
- 2) Initialize solution to empty set
- 3) For each i from 1 to m, if adding e_i to the solution won't create a cycle, add it.

Kruskal's Algorithm



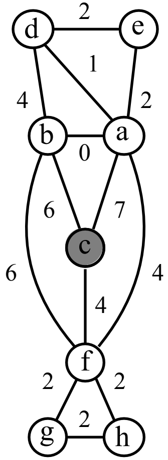
- Kruskal's algorithm is a greedy algorithm for finding a MST
- 1) Sort the edges in increasing order of weight, $e_1, e_2, \ldots, e_m$.
- 2) Initialize solution to empty set
- 3) For each i from 1 to m, if adding e_i to the solution won't create a cycle, add it.

Kruskal's Algorithm



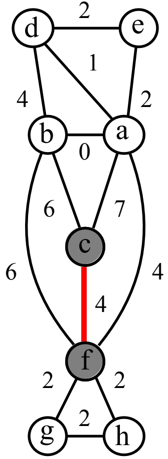
- Kruskal's algorithm is a greedy algorithm for finding a MST
- 1) Sort the edges in increasing order of weight, $e_1, e_2, \ldots, e_m$.
- 2) Initialize solution to empty set
- 3) For each i from 1 to m, if adding e_i to the solution won't create a cycle, add it.

Prim's Algorithm



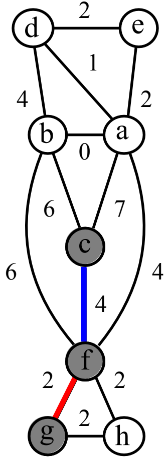
- Prim's is a another greedy algorithm for finding a MST
- 1) Start with one vertex on one side, call it s, and all other vertices on the other side.
- 2) Find the lowest weight edge (u, v) such that u is on s 's side and v isn't and add it to the solution
- 3) Move v to s's side and repeat until all vertices are on s's side.

Prim's Algorithm



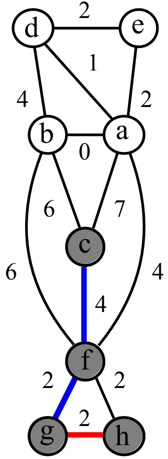
- Prim's is a another greedy algorithm for finding a MST
- 1) Start with one vertex on one side, call it s, and all other vertices on the other side.
- 2) Find the lowest weight edge (u, v) such that u is on s 's side and v isn't and add it to the solution
- 3) Move v to s's side and repeat until all vertices are on s's side.

Prim's Algorithm



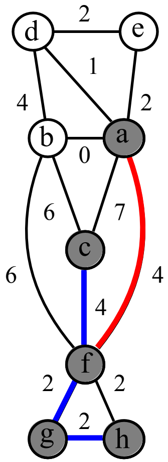
- Prim's is a another greedy algorithm for finding a MST
- 1) Start with one vertex on one side, call it s, and all other vertices on the other side.
- 2) Find the lowest weight edge (u, v) such that u is on s 's side and v isn't and add it to the solution
- 3) Move v to s's side and repeat until all vertices are on s's side.

Prim's Algorithm



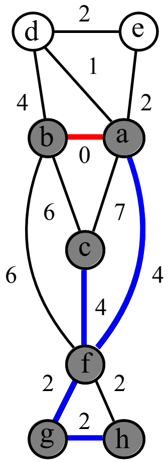
- Prim's is a another greedy algorithm for finding a MST
- 1) Start with one vertex on one side, call it s, and all other vertices on the other side.
- 2) Find the lowest weight edge (u, v) such that u is on s 's side and v isn't and add it to the solution
- 3) Move v to s's side and repeat until all vertices are on s's side.

Prim's Algorithm



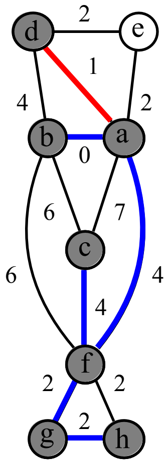
- Prim's is a another greedy algorithm for finding a MST
- 1) Start with one vertex on one side, call it s, and all other vertices on the other side.
- 2) Find the lowest weight edge (u, v) such that u is on s 's side and v isn't and add it to the solution
- 3) Move v to s's side and repeat until all vertices are on s's side.

Prim's Algorithm



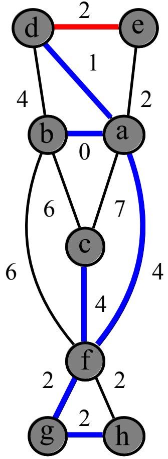
- Prim's is a another greedy algorithm for finding a MST
- 1) Start with one vertex on one side, call it s, and all other vertices on the other side.
- 2) Find the lowest weight edge (u, v) such that u is on s 's side and v isn't and add it to the solution
- 3) Move v to s's side and repeat until all vertices are on s's side.

Prim's Algorithm



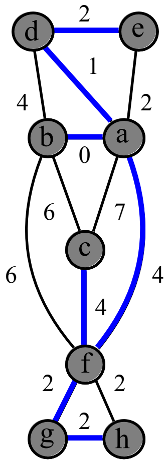
- Prim's is a another greedy algorithm for finding a MST
- 1) Start with one vertex on one side, call it s, and all other vertices on the other side.
- 2) Find the lowest weight edge (u, v) such that u is on s 's side and v isn't and add it to the solution
- 3) Move v to s's side and repeat until all vertices are on s's side.

Prim's Algorithm



- Prim's is a another greedy algorithm for finding a MST
- 1) Start with one vertex on one side, call it s, and all other vertices on the other side.
- 2) Find the lowest weight edge (u, v) such that u is on s 's side and v isn't and add it to the solution
- 3) Move v to s's side and repeat until all vertices are on s's side.

Prim's Algorithm



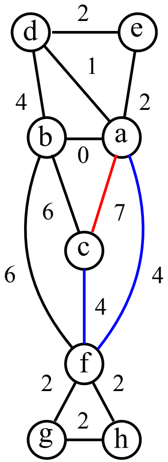
- Prim's is a another greedy algorithm for finding a MST
- 1) Start with one vertex on one side, call it s, and all other vertices on the other side.
- 2) Find the lowest weight edge (u, v) such that u is on s 's side and v isn't and add it to the solution
- 3) Move v to s's side and repeat until all vertices are on s's side.

Video 3.9

Sampath Kannan

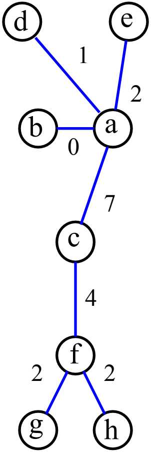
Cycle Property

Theorem: Let e be an edge of maximum weight in a cycle C in G. Then there is an MST that does not contain e.



Cycle Property

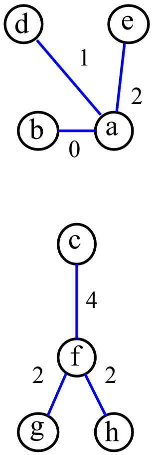
Theorem: Let e be an edge of maximum weight in a cycle C in G. Then there is an MST that does not contain e.



Suppose T* is a tree that contains e

Cycle Property

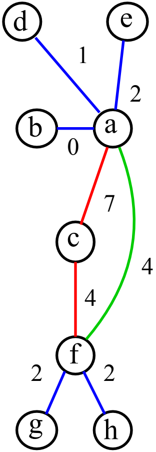
Theorem: Let e be an edge of maximum weight in a cycle C in G. Then there is an MST that does not contain e.



Suppose T * is a tree that contains e
Removing e splits T * into two connected components

Cycle Property

Theorem: Let e be an edge of maximum weight in a cycle C in G. Then there is an MST that does not contain e.



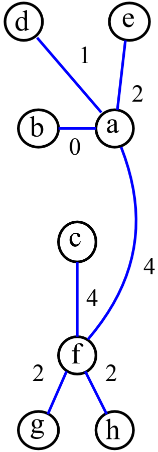
Suppose T * is a tree that contains e

Removing e splits T * into two connected components

The cycle C must have another edge f that goes between the components

Cycle Property

Theorem: Let e be an edge of maximum weight in a cycle C in G. Then there is an MST that does not contain e.



Suppose T^* is a tree that contains e

Removing e splits T^* into two connected components

The cycle C must have another edge f that goes between the components

Adding f to $T^* - e$ yields a tree of weight $w(T^*) - w(e) + w(f) \leq w(T^*)$. So we can always get a tree at least as good without e.

Correctness of Kruskal's

- How does this theorem relate to Kruskal's?

Correctness of Kruskal's

- How does this theorem relate to Kruskal's?
- Look at any edge e_j that Kruskal's doesn't choose
- It must be a part of some cycle where we have already included the other edges.

Correctness of Kruskal's

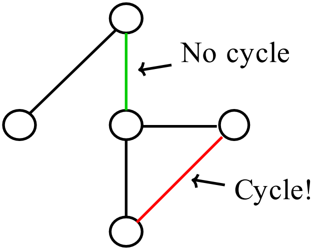
- How does this theorem relate to Kruskal's?
- Look at any edge e_j that Kruskal's doesn't choose
- It must be a part of some cycle where we have already included the other edges.
- Since we sort the edges by weight it must be the heaviest edge in the cycle, so by the theorem its safe to not include it!
- Therefore Kruskal's is correct

Kruskal's Implementation

- When examining e_j we need to recognize if it completes a cycle with already chosen edge set S.
- S defines a graph that has connected components

Kruskal's Implementation

- When examining e_j we need to recognize if it completes a cycle with already chosen edge set S.
- S defines a graph that has connected components
- $e_j = (u, v)$ completes a cycle if and only if u and v are in the same component.



- So we need a way to track the components as edges are added.

Union-Find

- Each vertex is initially in its own connected component.

Union-Find

- Each vertex is initially in its own connected component.
- When an edge (u, v) is added to S, the (seperate) connected components of u and v become one. We need to compute the **union** of the sets of vertices.

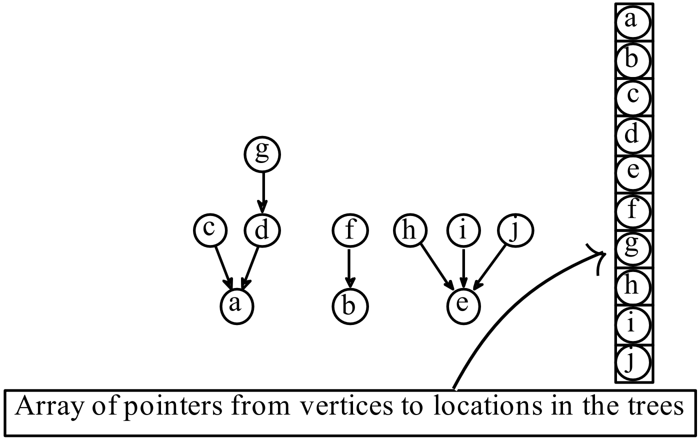
Union-Find

- Each vertex is initially in its own connected component.
- When an edge (u, v) is added to S, the (seperate) connected components of u and v become one. We need to compute the **union** of the sets of vertices.
- To decide whether to add (u, v) we need to **find** components of u and v to see if they are the same.

Union-Find

- Each vertex is initially in its own connected component.
- When an edge (u, v) is added to S, the (seperate) connected components of u and v become one. We need to compute the **union** of the sets of vertices.
- To decide whether to add (u, v) we need to **find** components of u and v to see if they are the same.
- Therefore we need to use the **Union-Find Data Structure**, we will present a simple one next.

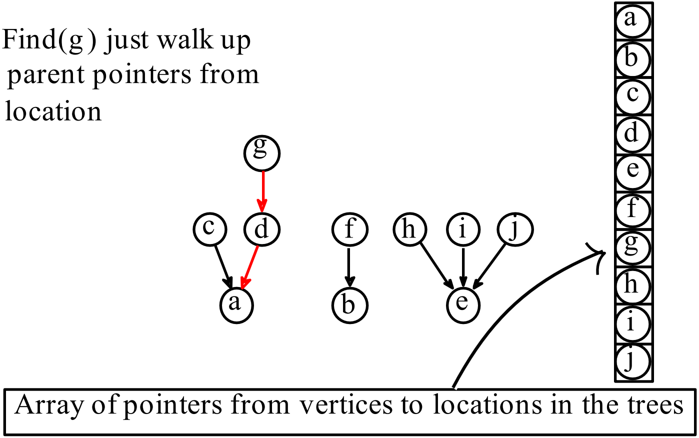
Union-Find Implementation



- Vertices are nodes in a tree with pointers to parent nodes (not children!).
- Name of a component is the vertex at the root.
- Heights of trees never exceed log(n) which is the time for union and find.

Union-Find Implementation

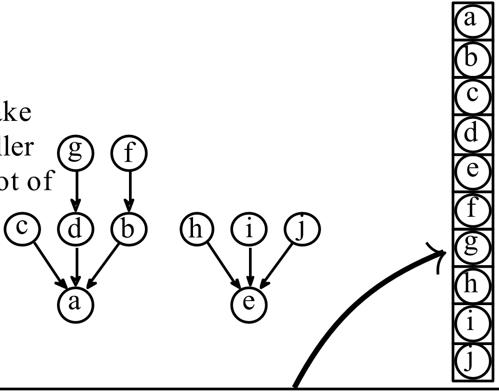
To Find(g) just walk up the parent pointers from g's location



- Vertices are nodes in a tree with pointers to parent nodes (not children!).
- Name of a component is the vertex at the root.
- Heights of trees never exceed log(n) which is the time for union and find.

Union-Find Implementation

For Union(g, b) make the root of the smaller tree point to the root of the larger tree.



Array of pointers from vertices to locations in the trees

- Vertices are nodes in a tree with pointers to parent nodes (not children!).
- Name of a component is the vertex at the root.
- Heights of trees never exceed log(n) which is the time for union and find.

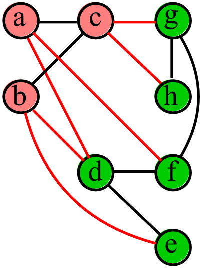
Kruskal's Running Time

- Sorting edge $O(m \log m) = O(m \log n)$ (Why: $n - 1 \leq m \leq n^2$).
- Processing each edge: Two finds and possibly a union is $O(\log n)$ per edge
- Total running time: $O(m \log n)$.

Video 3.10

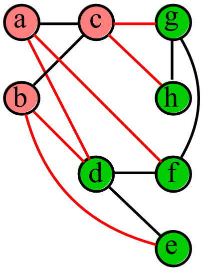
Sampath Kannan

Graph Cuts



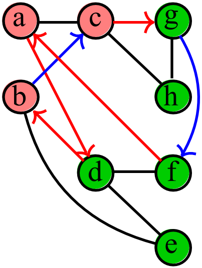
- A **cut** in a graph $G = (V, E)$ is a set of edges defined by a partition of V into S and $V - S$.

Graph Cuts



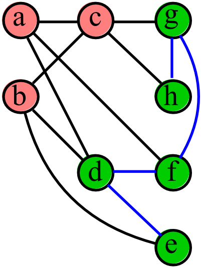
- A **cut** in a graph $G = (V, E)$ is a set of edges defined by a partition of V into S and $V - S$.
- Properties of a cut:
  - In any cut, any spanning tree must contain at least one edge

Graph Cuts



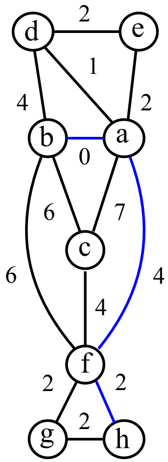
- A **cut** in a graph $G = (V, E)$ is a set of edges defined by a partition of V into S and $V - S$.
- Properties of a cut:
  - In any cut, any spanning tree must contain at least one edge
  - Any cycle contains an even number of cut edges

Graph Cuts



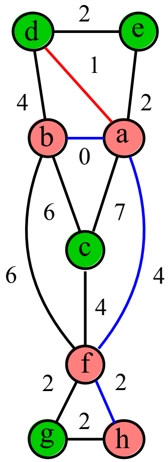
- A **cut** in a graph $G = (V, E)$ is a set of edges defined by a partition of V into S and $V - S$.
- Properties of a cut:
  - In any cut, any spanning tree must contain at least one edge
  - Any cycle contains an even number of cut edges
  - A cut **respects** a set of edges A, if it doesn't cut any of them.

Cut Property



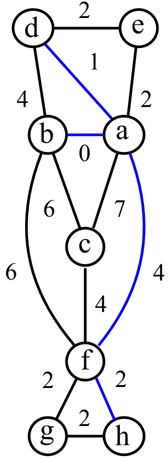
- If A is a subset of edges in an MST...

Cut Property



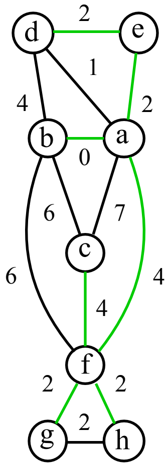
- If A is a subset of edges in an MST...
- and $(S, V - S)$ is a cut respecting A, where e is the lightest edge in the cut

Cut Property



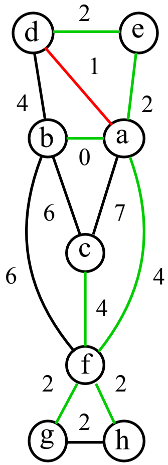
- If A is a subset of edges in an MST...
- and $(S, V - S)$ is a cut respecting A, where e is the lightest edge in the cut
- then there is an MST that contains A and e.

Proof of Cut Property



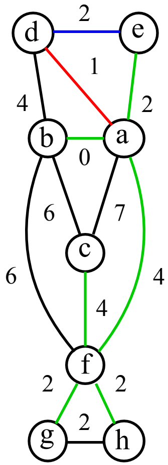
- Suppose the cheapest tree containing A is T without e

Proof of Cut Property



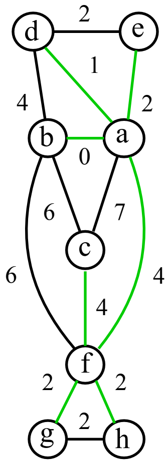
- Suppose the cheapest tree containing A is T without e
- Add e to T

Proof of Cut Property



- Suppose the cheapest tree containing A is T without e
- Add e to T
- Creates cycle, must contain f = e from the cut

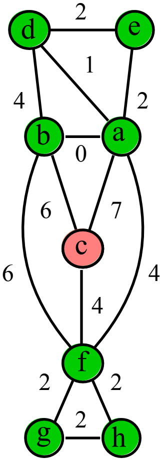
Proof of Cut Property



- Suppose the cheapest tree containing A is T without e
- Add e to T
- Creates cycle, must contain f = e from the cut
- Removing f and adding e creates a tree of weight $w(T) - w(f) + w(e) \leq w(T)$. Which as good as T .

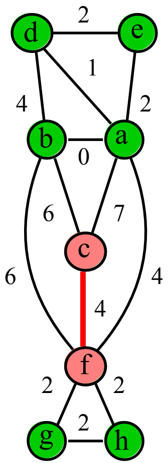
Correctness of Prim's

Prim's algorithm always adds the lightest edge in a cut respecting previously chosen edge.



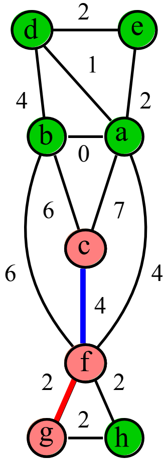
Correctness of Prim's

Prim's algorithm always adds the lightest edge in a cut respecting previously chosen edge.



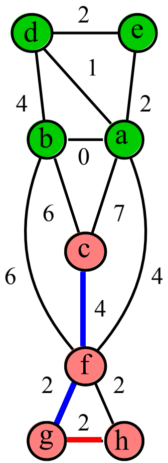
Correctness of Prim's

Prim's algorithm always adds the lightest edge in a cut respecting previously chosen edge.



Correctness of Prim's

Prim's algorithm always adds the lightest edge in a cut respecting previously chosen edge.



Correctness of Prim's

Prim's algorithm always adds the lightest edge in a cut respecting previously chosen edge.