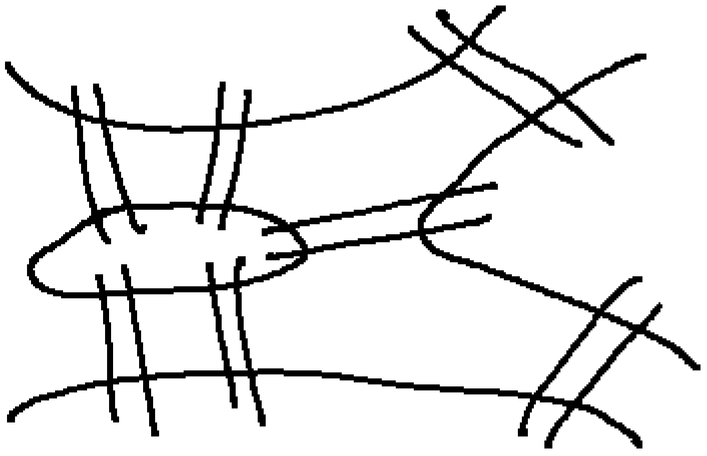
Further Data Structures

- The story so far
  - Saw some fundamental operations as well as advanced operations on arrays, stacks, and queues
  - Saw a dynamic data structure, the linked list, and its applications.
  - Saw the hash table so that insert/delete/find can be supported efficiently.
  - Saw trees and and applications to searching.
- This week we will
  - Introduce graphs as a data structure.
  - Study operations on graphs including searching.

Introduction to Graphs

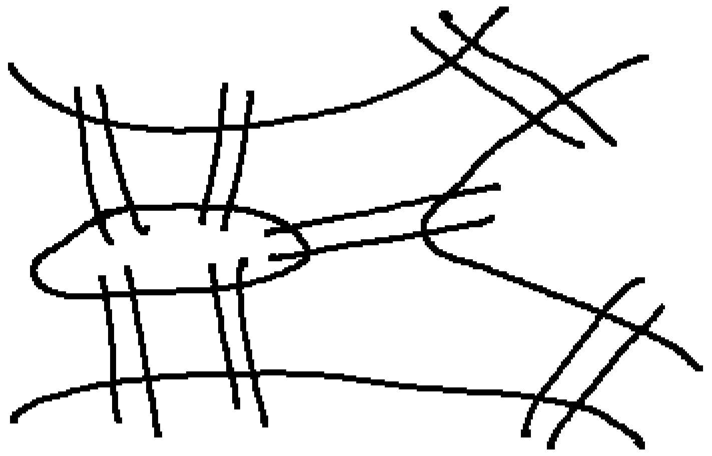
- Consider the following problem.
- A river with an island and bridges.
- The problem is to see if there is a way to start from some landmass and using each bridge exactly once, return to the starting point.

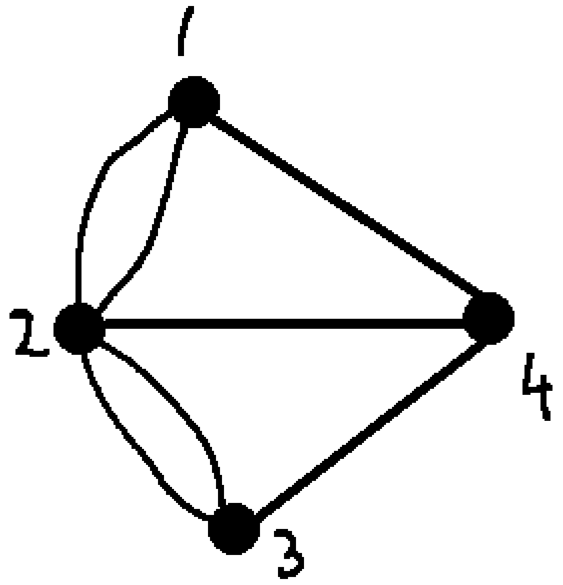


Introduction to Graphs

- The above problem dates back to the 17th century.
- Several people used to try to solve it.
- Euler showed that no solution exists for this problem.
- Further, he exactly characterized when a solution exists.
- By solving this problem, it is said that Euler started the study of graphs.

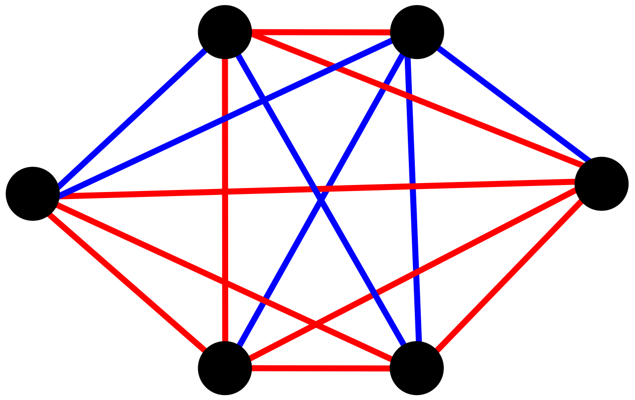
Introduction to Graphs





- The figure on the right shows the same situation modeled as a graph.
- There exist several such classical problems where graph theory has been used to arrive at elegant solutions.

Introduction to Graphs



- Another such problem: In any set of at least six persons, there are either three mutual acquaintances or three mutual strangers.

Introduction to Graphs

- Formally, let V be a set of points, also called as vertices.
- Let $E \subseteq V \times V$ be a subset of the cross product of V with itself. Elements of E are also called as edges.
- A graph can be seen as the tuple (V, E). Usually denoted by upper case letters G, H, etc.

Our Interest

- Understand a few terms associated with graphs.
- Study how to represent graphs in a computer program.
- Study how traverse graphs.
- Study mechanisms to find paths between vertices.
- Spanning trees
- And so on...

Few Terms

- Recall that a graph G = (V, E) is a tuple with E being a subset of VxV.
- Scope for several variations: for u, v in V
  - Should we treat (u,v) as same as (v,u)?

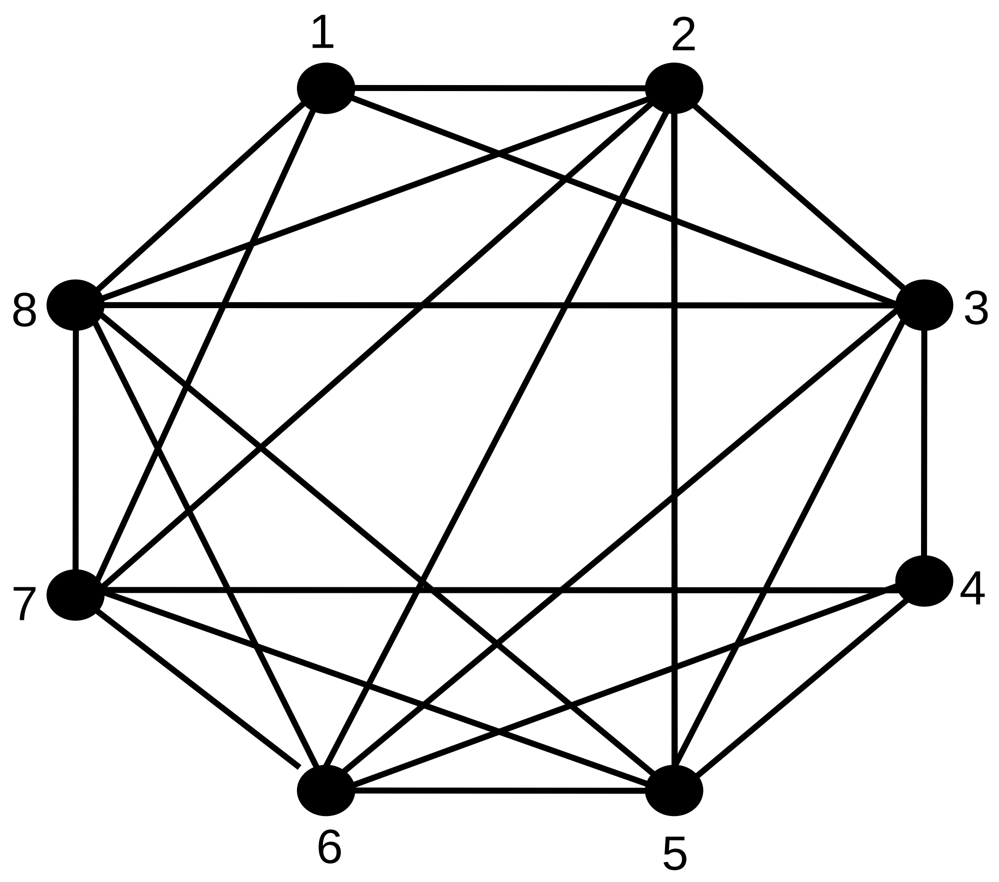
Few Terms

- Recall that a graph G = (V, E) is a tuple with E being a subset of VxV.
- Scope for several variations: for u, v in V
  - Should we treat (u,v) as same as (v,u)? In this case, the graph is called as a undirected graph.
  - Treat (u,v) as different from (v,u).

Few Terms

- Recall that a graph G = (V, E) is a tuple with E being a subset of VxV.
- Scope for several variations: for u, v in V
  - Should we treat (u,v) as same as (v,u)? In this case, the graph is called as a undirected graph.
  - Treat (u,v) as different from (v,u). In this case, the graph is called as a directed graph.
  - Should we allow (u,u) in E? Edges of this kind are called as self-loops.

Undirected Graphs



- In this case, the edge (u,v) is same as the edge (v,u).
  - Normally written as edge uv.

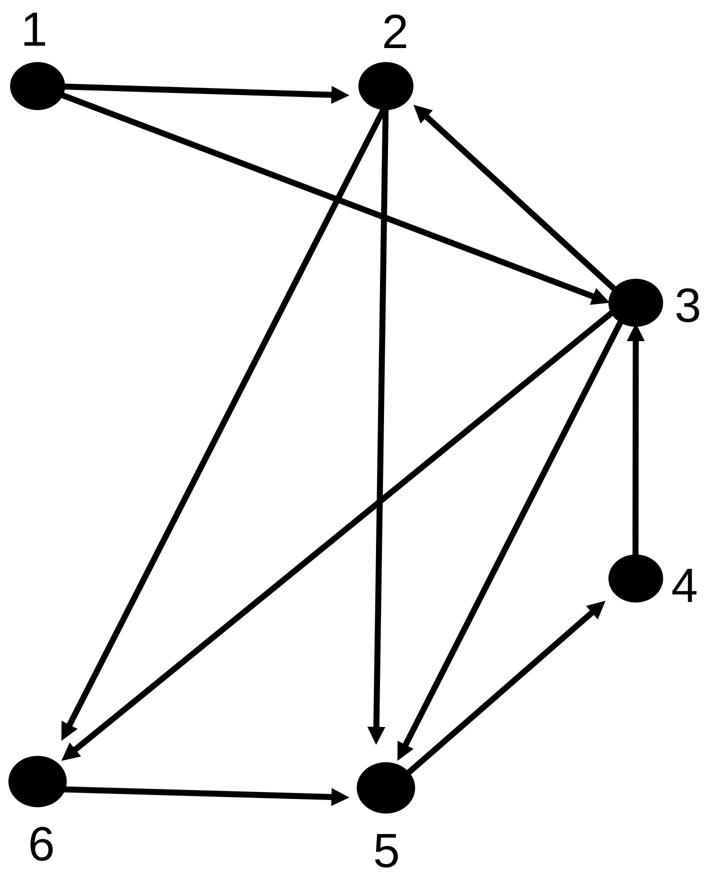
Undirected Graphs

- The degree of a node v in a graph G = (V,E) is the number of its neighbours.
  - It is denoted by d(v).
- In the above example, the degree of vertex 4 is 4. The neighbors of vertex 4 are {3, 5, 6, 7}.
- The degree of a graph G = (V,E) is the maximum degree of any node in the graph and is denoted $\Delta(G)$. Sometimes, written as just Δ when G is clear from the context.
  - Thus, $\Delta = \max_{v \in V} d(v)$.
  - Thus $\Delta = 6$ for the above graph.

Some Terms

- In a graph G = (V,E), a path is a sequence of vertices $v_1, v_2, \cdots, v_i$, all distinct, such that $v_k v_{k+1} \in E$ for $1 \le k \le i - 1$.
- If, under the above conditions, $v_1 = v_i$ then it is called a cycle.
- The length of such a path(cycle) is $i - 1$(resp. i).
- An example: 3 – 8 – 5 – 2 in the above graph is a path from vertex 3 to vertex 2.
- Similarly, 2 – 7 – 6 – 5 – 2 is a cycle.

Directed Graphs



- In this case, the edge (u,v) is distinct from the edge (v,u).
  - Normally written as edge ⟨u, v⟩.

Directed Graphs

- Have to alter the definition of degree as
- in-degree(v) : the number of neighbors w of v such that (w,v) in E.
- out-degree(v) : the number of neighbors w of v such that (v,w) in E.
- in-degree(4) = 1
- out-degree(2) = 2.

Directed Graphs

- Have to alter the definition of path and cycle to directed path and directed cycle.

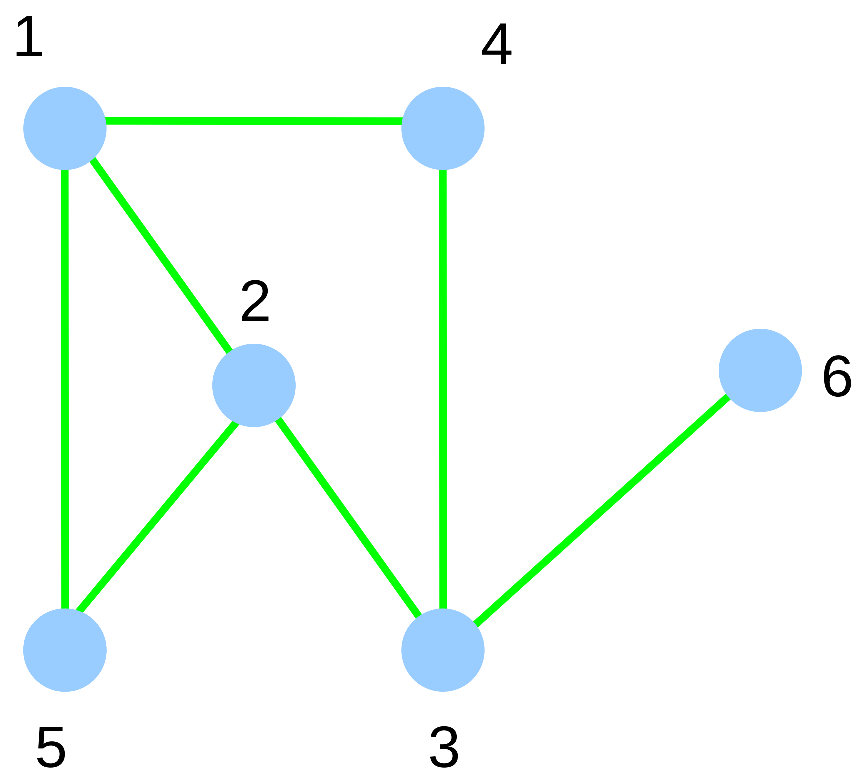
Representing Graphs

- How to represent graphs in a computer program.
- Several ways possible.

Adjacency Matrix

- The graph is represented by an $n \times n$–matrix where n is the number of vertices.
- Let the matrix be called A. Then the element A[i, j] is set to 1 if $(i, j) \in E(G)$ and 0 otherwise, where $1 \leq i, j \leq n$.
- The space required is $O(n^2)$ for a graph on n vertices.
- By far the simplest representation.
- Many algorithms work very efficiently under this representation.

Adjacency Matrix Example



A	1	2	3	4	5	6
1	0	1	0	1	1	0
2	1	0	1	0	1	0
3	0	1	0	1	0	1
4	1	0	1	0	0	0
5	1	1	0	0	0	0
6	0	0	1	0	0	0

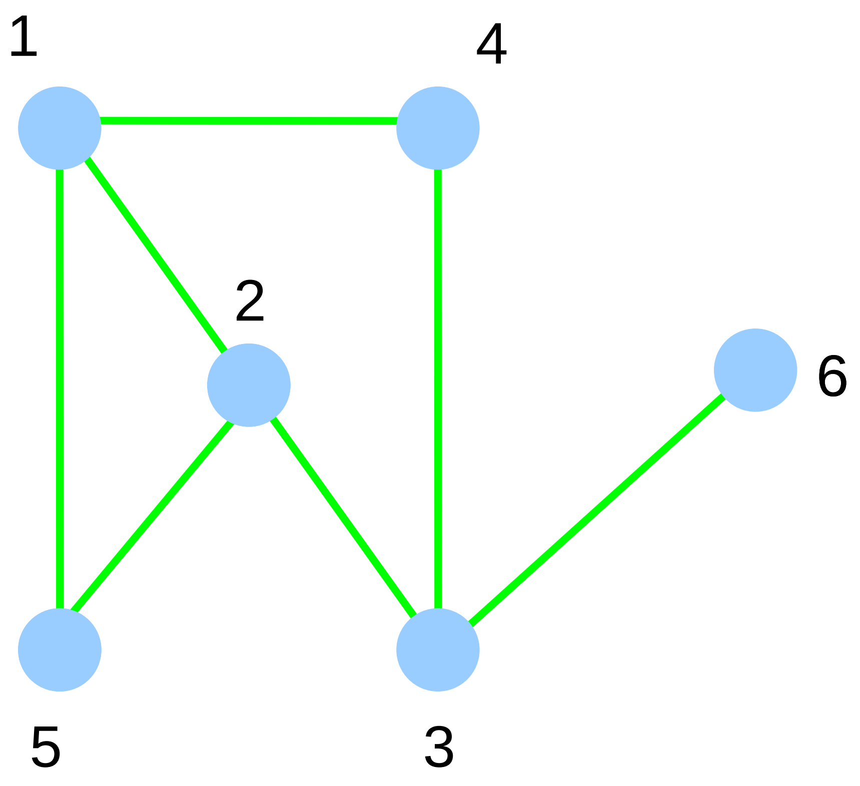
Adjacency Matrix Observations

- Space required is n^2
- The matrix is symmetric and 0,1—valued.
  - For directed graphs, the matrix need not be symmetric.
- Easy to check for any u,v whether uv is an edge.
- Most algorithms also take $O(n^2)$ time in this representation.
- The following is an exception: The Celebrity Problem.

Adjacency List

- Imagine a list for each vertex that will contain the list of neighbours of that vertex.
- The space required will only be O(m).
- However, one drawback is that it is difficult to check whether a particular pair (i, j) is an edge in the graph or not.

Adjacency List Example



$$\begin{tikzcd}
1 \arrow[r] & 2 \arrow[r] & 5 \arrow[r] & 4 \\
2 \arrow[r] & 5 \arrow[r] & 1 \arrow[r] & 3 \\
3 \arrow[r] & 2 \arrow[r] & 6 \arrow[r] & 4 \\
4 \arrow[r] & 1 \arrow[r] & 3 & \\
5 \arrow[r] & 1 \arrow[r] & 2 & \\
6 \arrow[r] & 3 & &
\end{tikzcd}$$

Adjacency List

- Useful representation for sparse graphs.
- Extends to also directed graphs.

Other Representations

- Neighbor maps

Searching Graphs

- A fundamental problem in graphs. Also called as traversing a graph.
- Need to visit every vertex.
- Can understand several properties of a graph using a traversal.
- Two main techniques : breadth first search, and depth first search.

Breadth First Search

- Recall level order traversal of a tree.
  - Starting from the root, visits every vertex in a level by level manner.
- Let us develop breadth first search as an extension of level order traversal.
- A few questions to be answered before we develop breadth first search.

Breadth First Search

- Question 1: For a graph, no notion of a root vertex.
- So, where should BFS start from?

Breadth First Search

- Question 1: For a graph, no notion of a root vertex.
- So, where should BFS start from?
- So, have to specify a starting vertex. Typically denoted s.
- Still other problems exist.

Breadth First Search

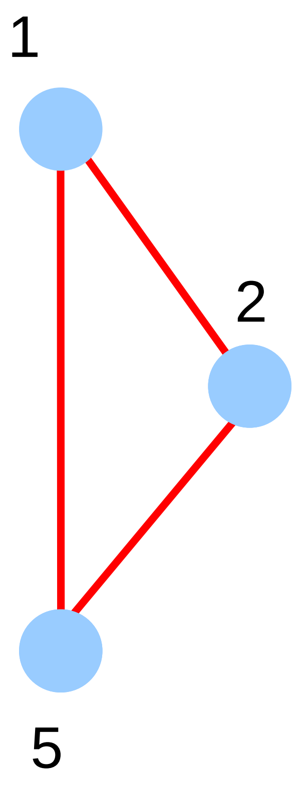
- In a tree, using level order traversal, each vertex is visited also exactly once.
  - Why?

Breadth First Search

- In a tree, using level order traversal, each vertex is visited also exactly once.
  - Recall that a tree is connected and has no cycles.

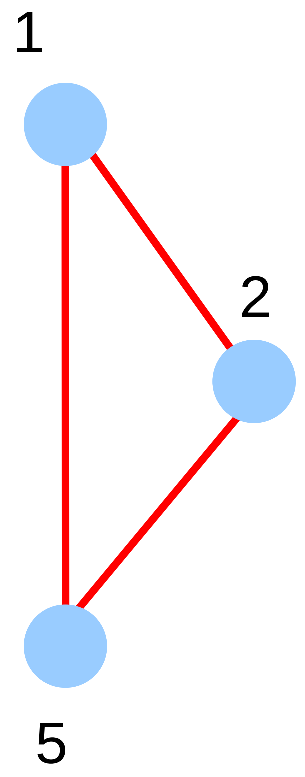
Breadth First Search

- In a tree, using level order traversal, each vertex is visited also exactly once.
  - Recall that a tree is connected and has no cycles.
- In a graph, that is no longer guaranteed.
  - Start from s = 2 and do a level order traversal.



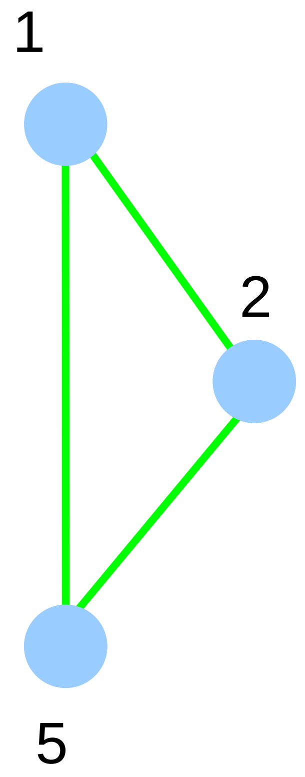
Breadth First Search

- In a tree, using level order traversal, each vertex is visited also exactly once.
  - Recall that a tree is connected and has no cycles.
- In a graph, that is no longer guaranteed.
  - Start from s = 2 and do a level order traversal
  - One of 1 or 5 visited more than once.



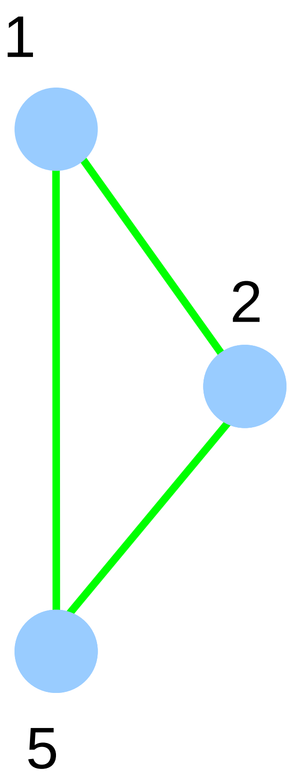
Breadth First Search

- Question 2: How to resolve that problem?



Breadth First Search

- Question 2: How to resolve that problem?
- Can remember if a vertex is already visited.
- Each vertex has a state among VISITED, NOT_VISITED, IN_PROGRESS.
- Why three states instead of just two?
  - Need them for a later use.

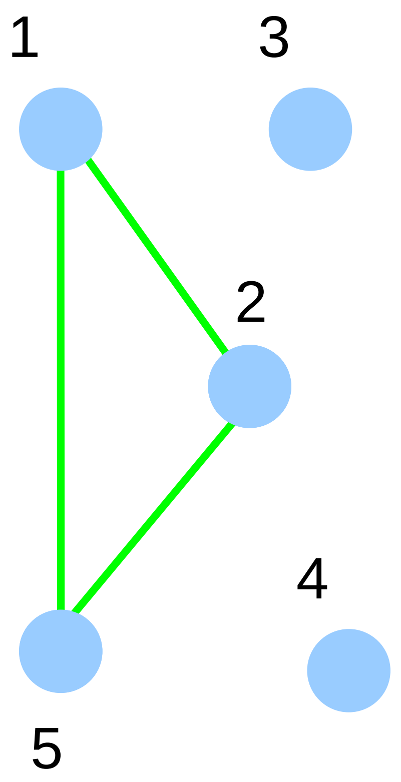


Breadth First Search

- Question 3: Can all vertices be reached from s?

Breadth First Search

- Question 3: Can all vertices be reached from s?
- For example, when s = 2, vertex 3 can never be visited.
- What to do with those vertices?
- Answer depends on the idea behind graph searching via BFS.



Breadth First Search

- The basic idea of breadth first search is to find the least number of edges between s and any other vertex in G.
  - The same property holds for level order traversal of a tree also with s as the root.
- Starting from s, we can thus visit vertices of distance k before visiting any vertex of distance k+1.
- For that purpose, define $d_s(v)$ to be the least number of edges between s and v in G.

Breadth First Search

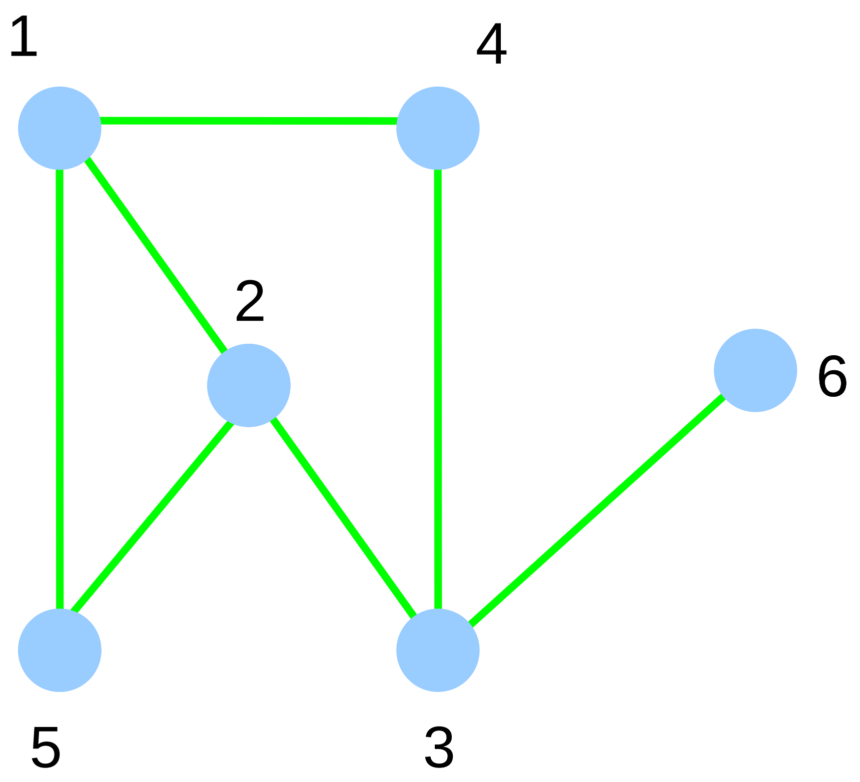
- So, for vertices v that are not reachable from s, can say that $d_s(v)$ is ∞
- Alike a level order traversal of a tree, can use a queue to store vertices in progress.

BFS Procedure

```
Procedure BFS(G)
for each v ∈ V do
π(v)= NIL;state[v] = NOT_VISITED; d(v) = ∞;
End-for
d[s] = 0; state[s] = IN_PROGRESS; π[s]= NIL,
Q = EMPTY; Q.Enqueue(s);
While Q is not empty do
v = Q.Dequeue();
for each neighbour w of v do
    if state[w] = NOT_VISITED then
        state[w] = IN_PROGRESS; π[w] = v;
        d[w] = d[v] + 1; Q.Enqueue(w);
    end-if
end-for
state[v] = FINISHED
end-while
```

BFS Example

- Start from s = 2.



	1	2	3	4	5	6
d :	∞	0	∞	∞	∞	∞
π :	–	–	–	–	–	–

BFS – Additional Details

- What is the runtime of BFS?

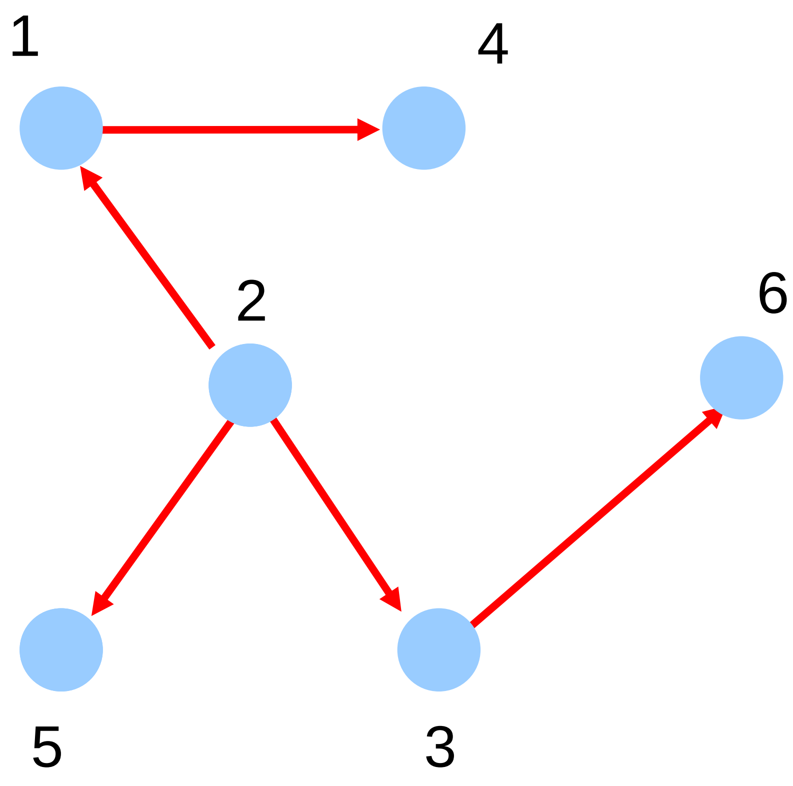
BFS – Additional Details

- What is the runtime of BFS?
  - How many times does each vertex enters the queue?
  - Each edge is considered only once.
- Therefore, the runtime of BFS should be O(m + n).

BFS – Additional Details

- The π value of a vertex v denotes the vertex u that discovered v.
- The π values maintained during BFS can be used to define a subgraph of G as follows.
- Define the predecessor subgraph of G = (V,E) as
  - $G_\pi = (V_\pi, E_\pi)$ where
  - $V_\pi = \{v \in V : \pi(v) \mathrel{!=} NULL\} \cup \{s\}$, i.e., all vertices reached during a BFS from s, and
  - $E_\pi = \{(\pi(v), v) \in E : v \in V_\pi - \{s\}\}$, directed edges from the parent of a vertex to the vertex.

BFS Example Contd...



Properties of BFS

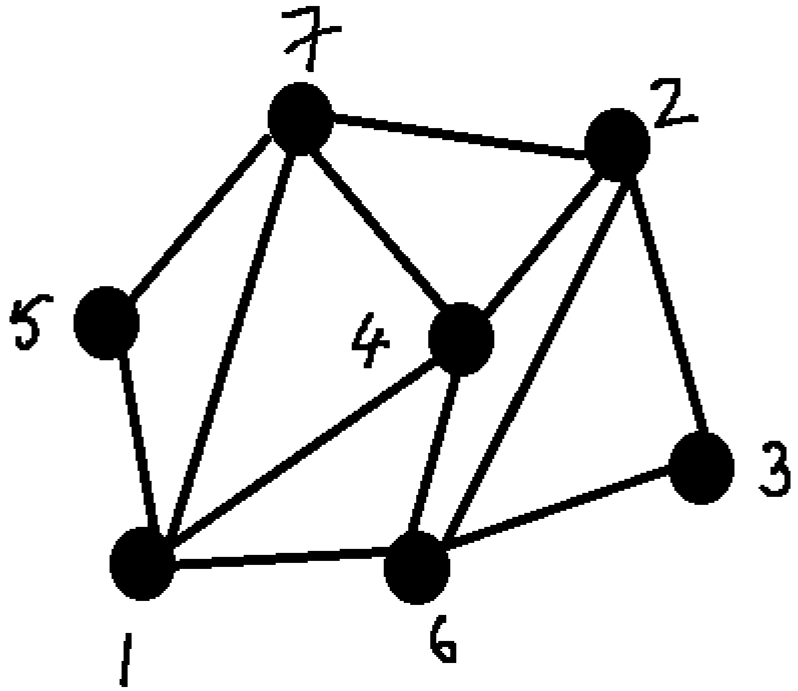
- Consider the time at which a vertex v has entered the queue.
- The state of v at that instant changes from NOT_VISITED to IN_PROGRESS.
- $d_s(v)$ changes to a finite value, and
- $d_s(v)$ can never change after that instant.

Classifying Edges

- Can classify edges of G according to BFS from a given s as follows.
- The edges of E_π are also called as tree edges.
- It holds that for a tree edge (u, v), d(v) = d(u) + 1.
- The edges of $E_N := E \setminus E_\pi$ are called as non-tree edges.
- These edges can be further classified as follows.

Classifying Edges

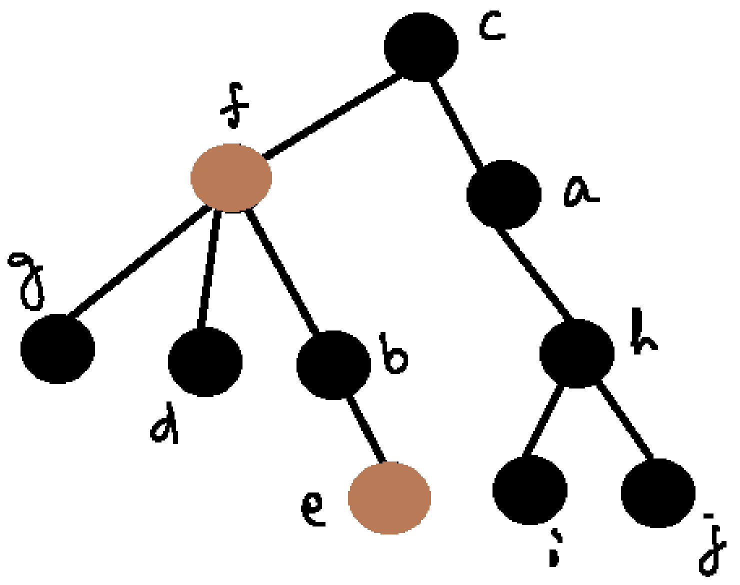
- Identify the tree- and the non-tree edges according to a BFS on the following graph. Choose vertex number 3 as the start vertex.

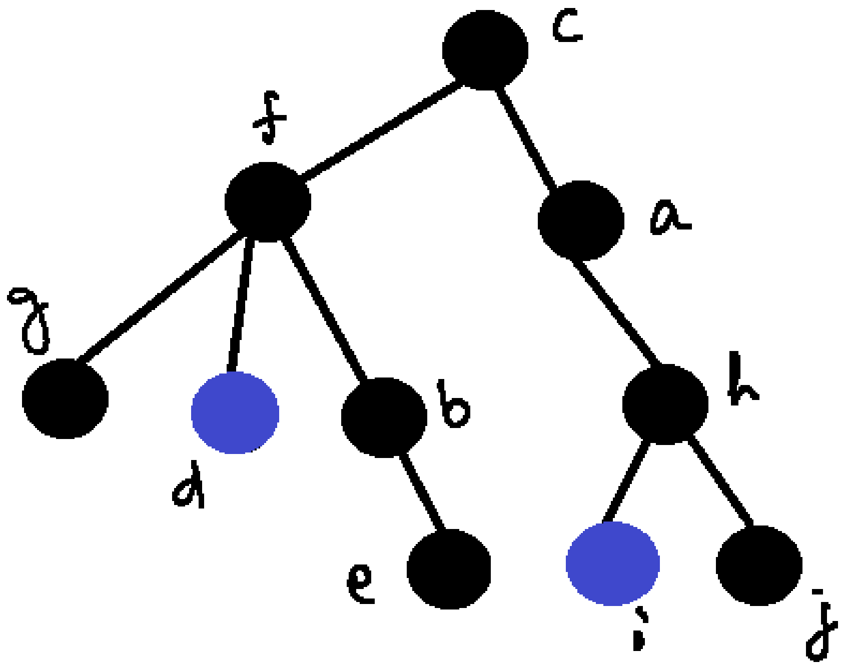


- Pick vertices in their order.

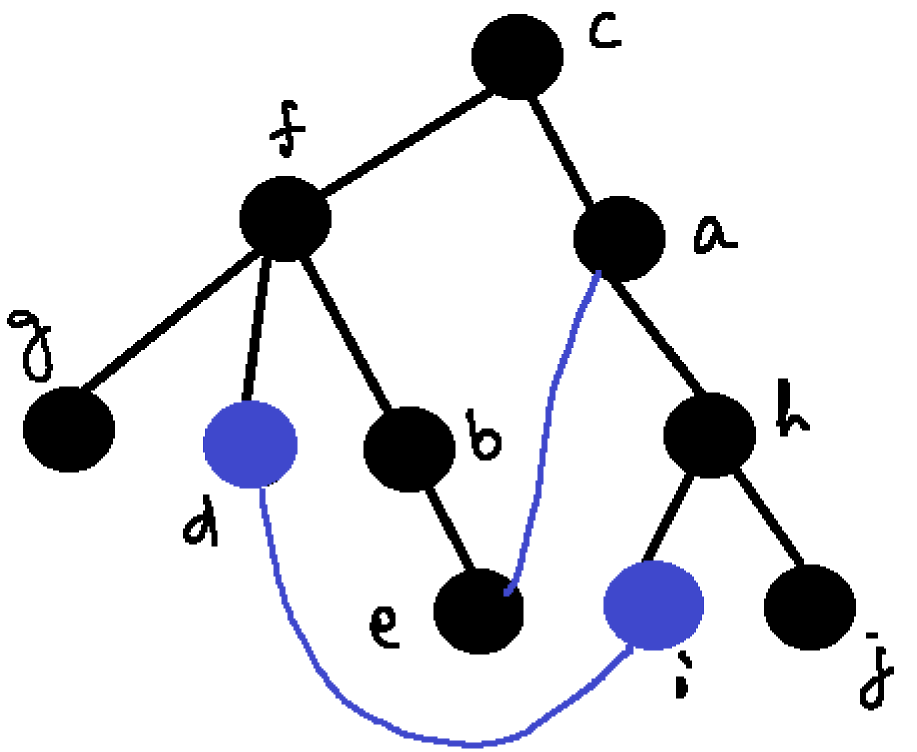
Classifying Edges – The Non-Tree Edges

- First, consider the predecessor subgraph. It is a tree. Call this tree as T_{BFS}.
- Tree edges according to BFS share a parent-child relationship.
- For any pair of vertices u, v:
  - Either they share an ancestor-descendant relation in T_{BFS}.
  - Or they do not.



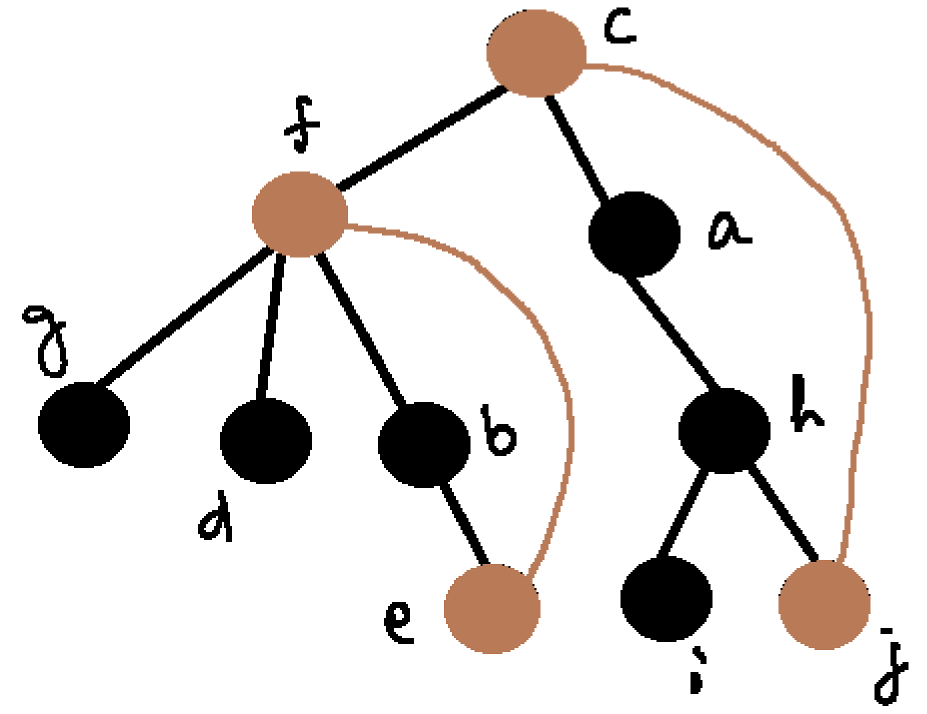


Classifying Edges – The Non-Tree Edges



- For any pair of vertices u, v:
  - Either they share an ancestor-descendant relation in T_{BFS}.
  - Or they do not.
    - (u, v) called as a cross edge. Examples (d,i) and (b,a).

Classifying Edges – The Non-Tree Edges



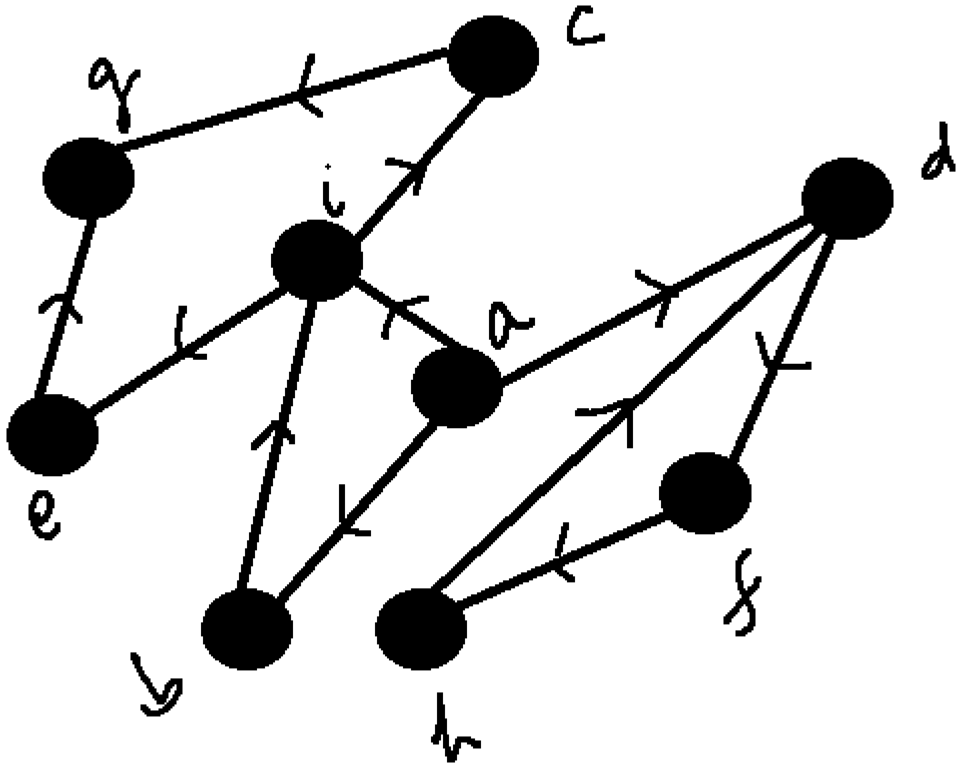
- For any pair of vertices u, v with (u,v) an edge in G:
  - Either they share an ancestor-descendant relation in T_{BFS}.
  - If u is an ancestor of v, then (u,v) is a forward edge.
  - If u is a descendant of v, then (u,v) is a back edge.

Directed or Undirected

- Most of the above observations hold even if G is directed.
  - The classification in fact makes more sense for directed graphs.
  - There can be back edges, but no forward edges.
- Can thus extend the notion of BFS to directed graphs.

Complete Example

- Perform BFS on the directed graph below with vertex a as the start vertex.
- Classify the edges of the graph according to the BFS.



BFS – Colors instead of States

- It is common to associate colors to the three states.
  - GREEN : Done vertices, VISITED
  - ORANGE : In progress/ In Queue
  - RED: Not visited yet.

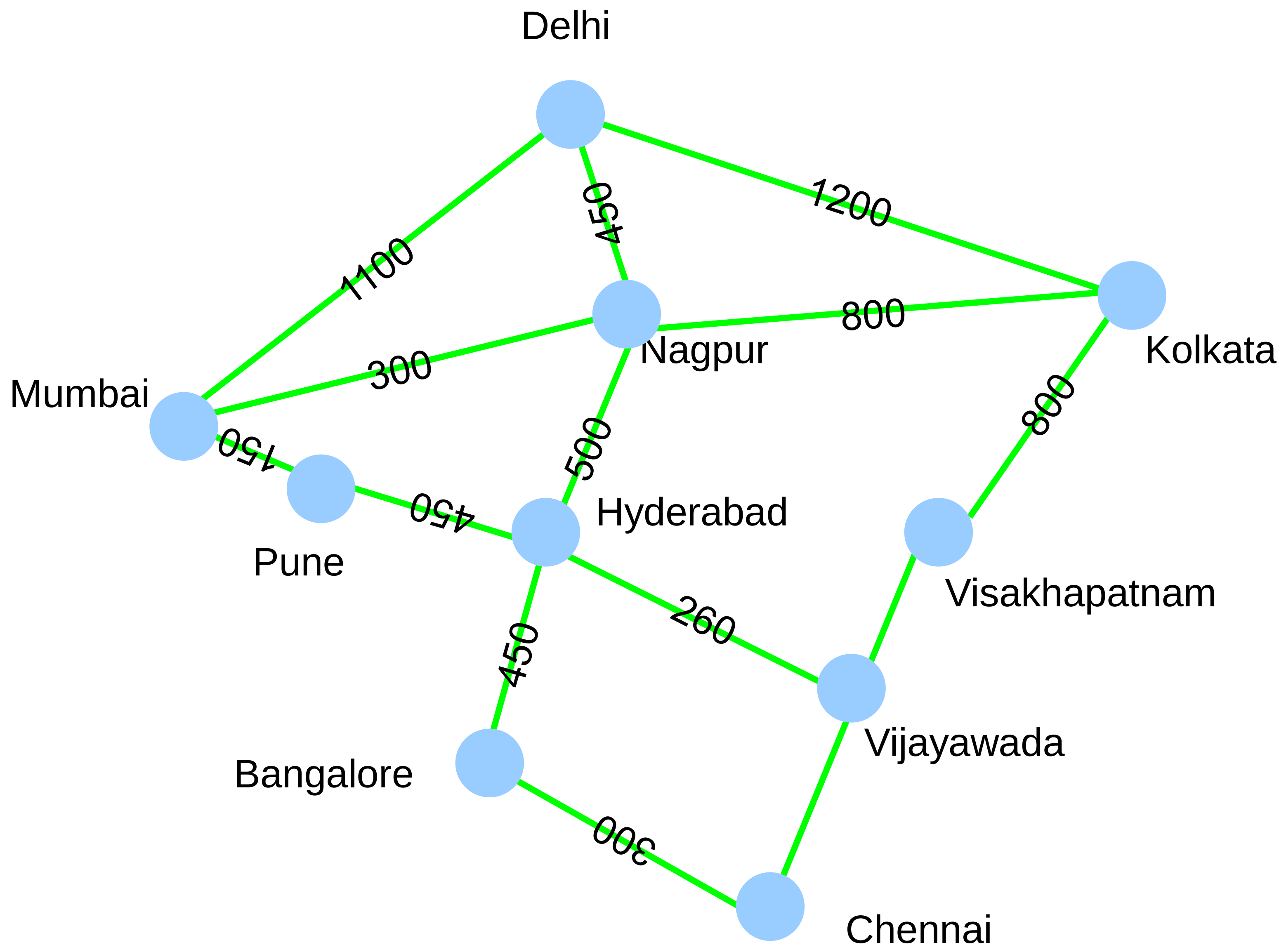
Towards Weighted BFS

- So, far we have measured $d_s(v)$ in terms of number of edges in the path from s to v.
- Equivalent to assuming that each edge in the graph has equal (unit) weight.
- But, several settings exist where edges may have unequal weights.

Towards Weighted BFS

- Consider a road network.
- Junctions can be vertices and roads can be edges.
- Can use such a graph to find the best way to reach from point A to point B.
- Best here can mean shortest distance/shortest delay/....
- Clearly, all edges need not have the same distance/delay/.

Towards Weighted BFS



A Few Problems

- Problem I : Given two points u and v, find the shortest distance between them.
- Problem II : Given a starting point s, find the shortest distance from s to all other points.
- Problem III : Find the shortest distance between all pairs of points.

A Few Problems

- Turns out that Problem I is not any easier than Problem II.
- Problem III is definitely harder than Problem II.
- We shall study problem II, and possibly Problem III.

Weighted Graphs

- The setting is more general.
- A weighted graph G = (V, E, W) is a graph with a weight function W : E –> R.
- Weighted graphs occur in several settings
  - Road networks
  - Internet

Problem II : Single Source Shortest Paths

- Problem II is also called the single source shortest paths problem.
- Let us extend BFS to solve this problem.
- Notice that BFS solves the problem when all the edge weights are 1.
  - Hence the reason to extend BFS

SSSP

- Extensions needed
  - 1. Weights on edges

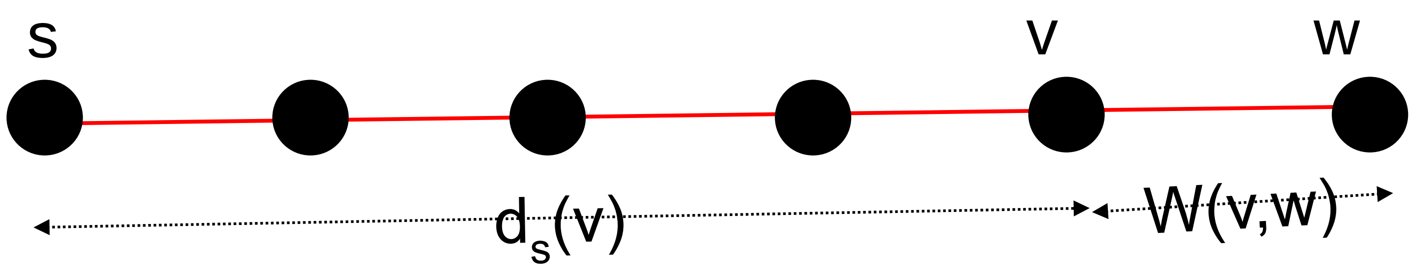
SSSP

- Extensions needed
  1. Weights on edges
  2. How to know when a node is finished.

SSSP

- Extensions needed
  - 1. Weights on edges
  - 2. How to know when a node is finished.
- For a vertex v, $d_s(v)$ will now refer to the shortest distance from s to v.
- Initially, like in BFS, $d_s(v) = \infty$ for all vertices v except s, and $d_s(s) = 0$.

Weighted BFS



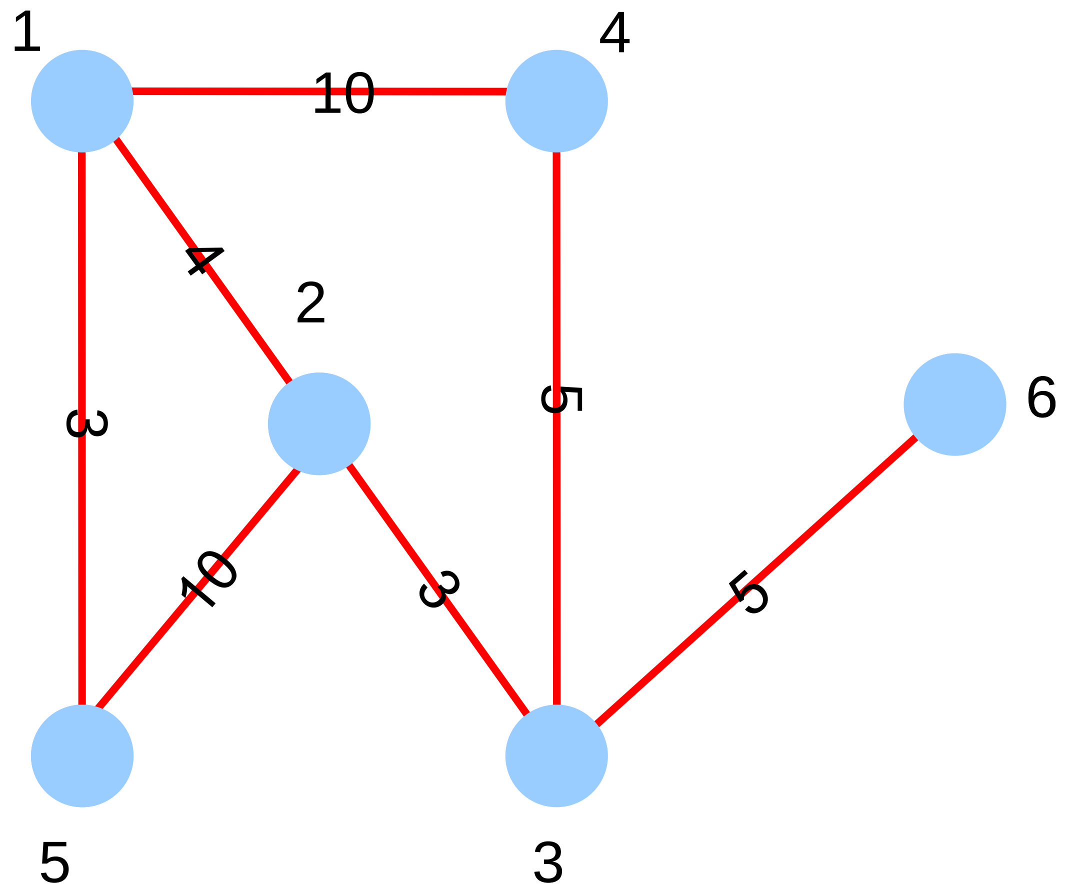
- Update $d_s(v)$ with weights.
- Also, weights on edges mean that if v is a neighbor of w in the shortest path from s to w, then $d_s(w) = d_s(v) + W(v,w)$.
  - Instead of $d_s(w) = d_s(v) + 1$ as in BFS.
- We will call this as the first change to BFS.

SSSP

- Notice that in BFS a node has three states : NOT_VISITED, VISITED, IN_QUEUE
- A vertex in VISITED state should have no more changes to $d_s()$ value.
- What about a vertex in IN_QUEUE state?
  - such a vertex has some finite value for $d_s(v)$.
  - Can $d_s(v)$ change for such vertices?
  - Consider an example.

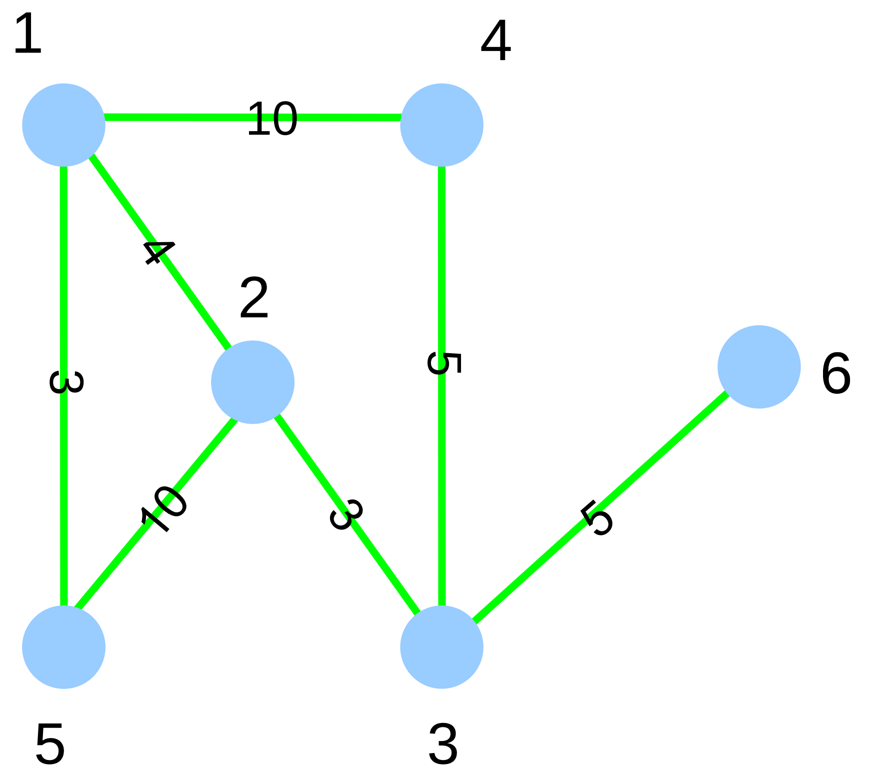
Weighted BFS

- Consider s = 2 and perform weighted BFS.



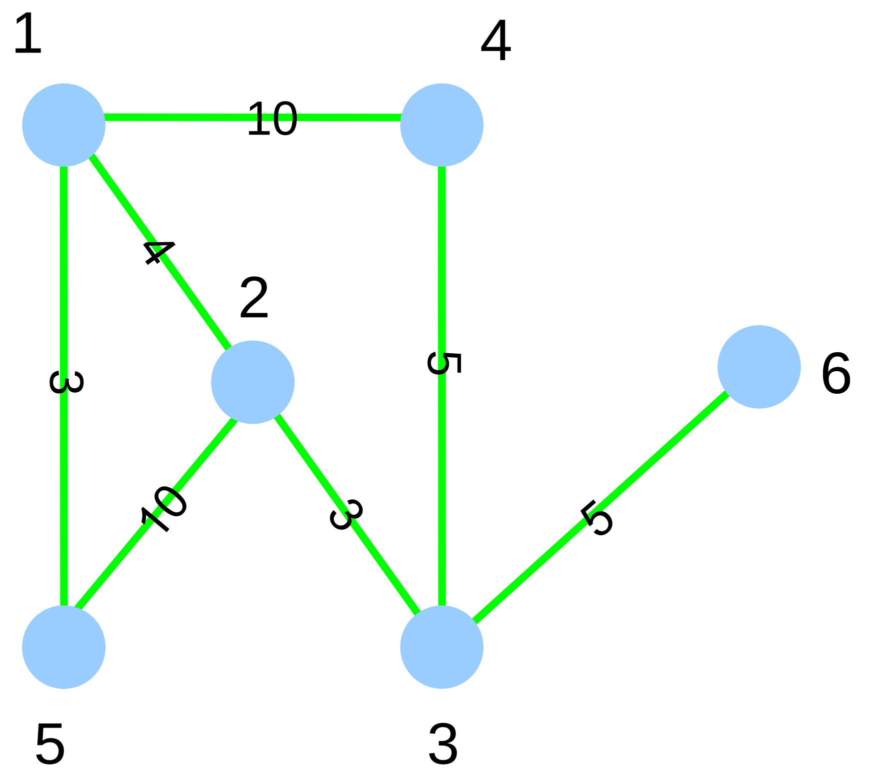
Weighted BFS

- Consider s = 2.
- From s, we will enqueue1, 5, and 3 with d(1) = 4, d(5) = 10, d(3) = 3, in that order.
- While vertex 5 is still in queue, can visit 5 from vertex 1 also.



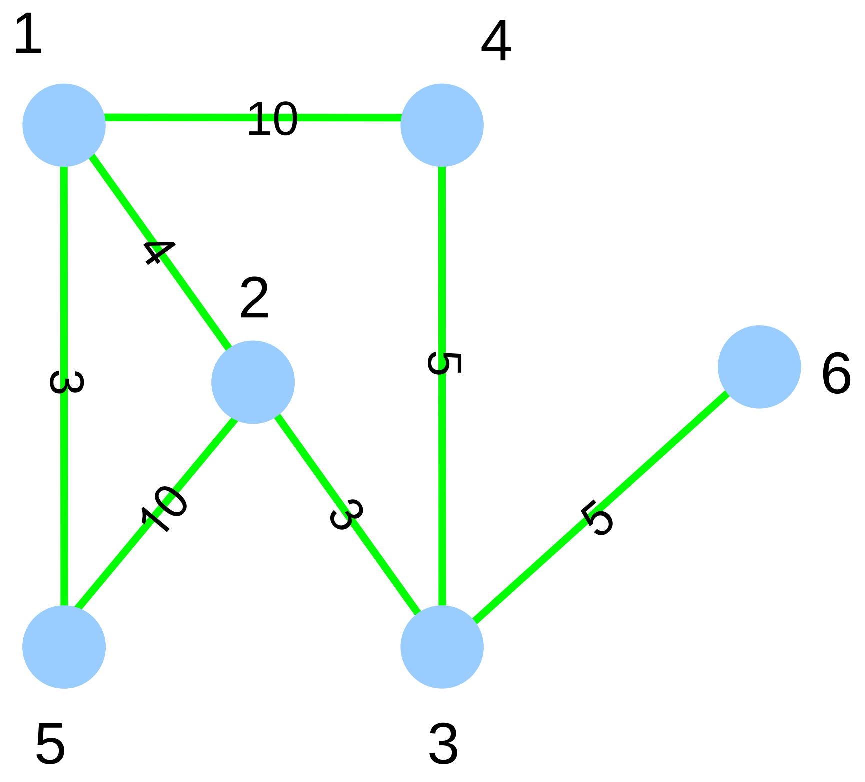
Weighted BFS

- Moreover, the weight of the edge 2- 5 is 10 whereas there is a shorter path from 2 to 5 via the path 2 – 1 – 5.
- So, it suggests that d(v) should be changed while v is still in the queue.



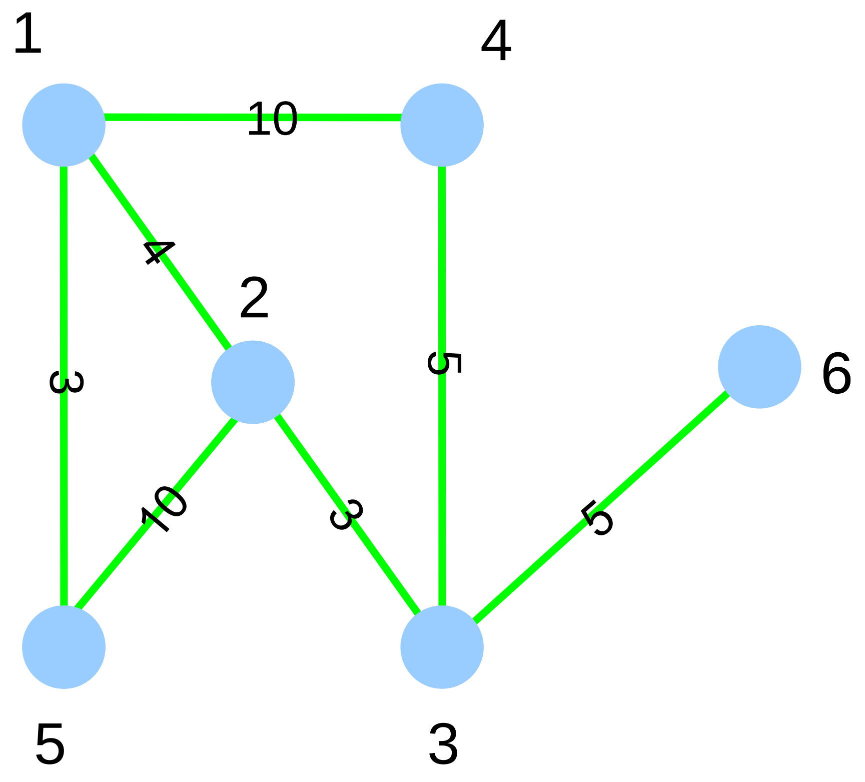
Weighted BFS

- Update d(v) for v in queue also.
- While v is in queue, we can check if d(v) is more than the distance along the new path.
- If so, then update d(v) to the new smaller value.
- Change 2 to BFS.



Weighted BFS

- Does that suffice?
- In the same example, if we change the order of vertices from 1, 5, 3 to 5, 1, 3, then vertex 5 will not be in queue when 1 is removed from the queue.



Weighted BFS

- So, the simple fix to change d(v) while v is still in queue does not work.
- May need to update d(v) even when v is not in queue?
  - But how long should we do so?

Weighted BFS

- Can do so as long as there are changes to some d(v)?
  - No need of a queue then, in this case really.
- Will this ever stop?
- Indeed it does. Why?
  - Intuitively, there are only a finite number of edges in any shortest path.

Weighted BFS

- Why does this ever stop?
- Consider a vertex v and the path from s to v of the least cost.

An Algorithm for SSSP

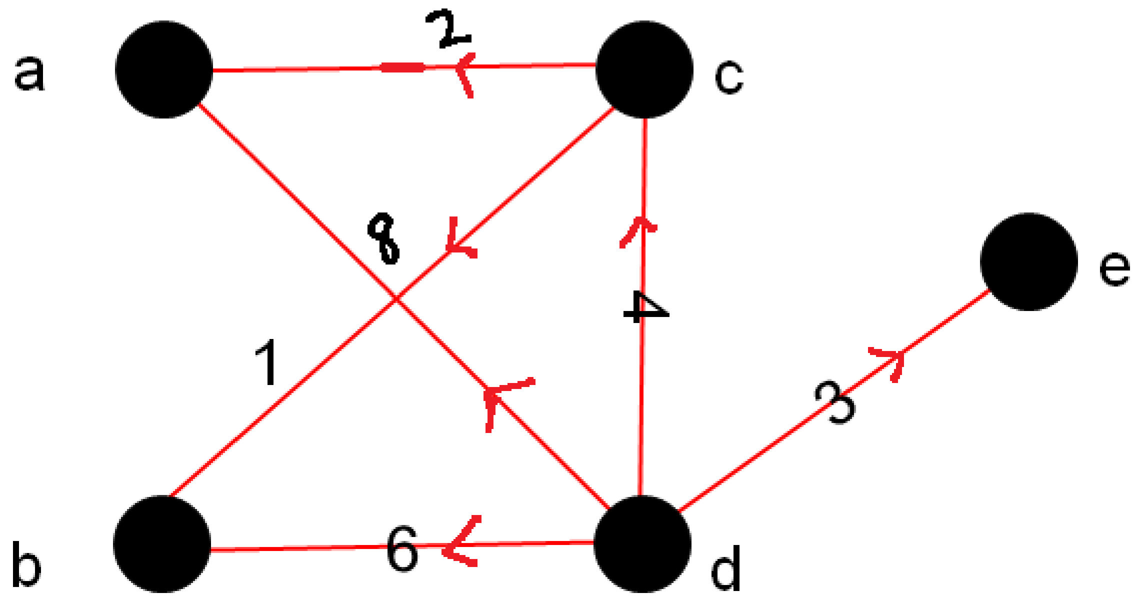
```
Algorithm SSSP(G,s)
begin
    for all vertices v do
        $d(v) = \infty$; $\pi(v) = NIL$;
    end-for
    $d(s) = 0$;
    for n-1 iterations do
        for each edge (v,w) do
            if $d(w) > d(v) + W(v,w)$ then
                $d(w) = d(v) + W(v,w)$; $\pi(w) = v$;
            end-if
        end-for
    end-for
end
```

Algorithm SSSP

- The above algorithm is called the Bellman-Ford algorithm.
- The algorithm requires O(mn) time.
  - For each of the n-1 iterations, we consider each edge once.
  - Has O(1) compute per edge.
- Just as in BFS, works also on directed graphs.
- Forms the basis of several algorithms for the Internet.

Example Algorithm SSSP

- Start vertex = d. Employ the Bellman-Ford algorithm to find shortest path from d to all other vertices.



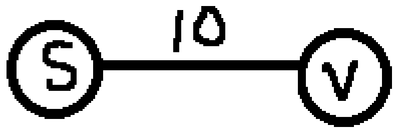
Thinking about the Bellman-Ford Algorithm

```
Algorithm SSSP(G,s)
begin
    for all vertices v do  $d(v) = \infty$; $\pi(v)$ = NIL;
    $d(s) = 0$;
    for n-1 iterations do
        for each edge (v,w) do
            if $d(w) > d(v) + W(v,w)$ then
                $d(w) = d(v) + W(v,w)$; $\pi(w) = v$;
end
```

- Why n-1 iterations are required?
- Let us prove the following via induction.

Thinking about the Bellman-Ford Algorithm

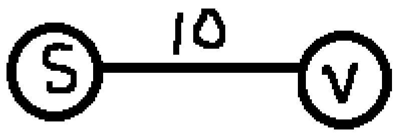
- Consider the source vertex s.
- For s, d(s) = 0 is the best possible result.
- So, s is FINISHED.



- Now consider a vertex v such that the shortest path from s to v contains only one edge, say (s,v).
- The edge (s,v) appears at some iteration of the second for loop in the first iteration of the main loop.
- At that point, d(v) is set correctly.

Thinking about the Bellman-Ford Algorithm

- Consider the source vertex s.
- For s, d(s) = 0 is the best possible result.
- So, s is FINISHED.



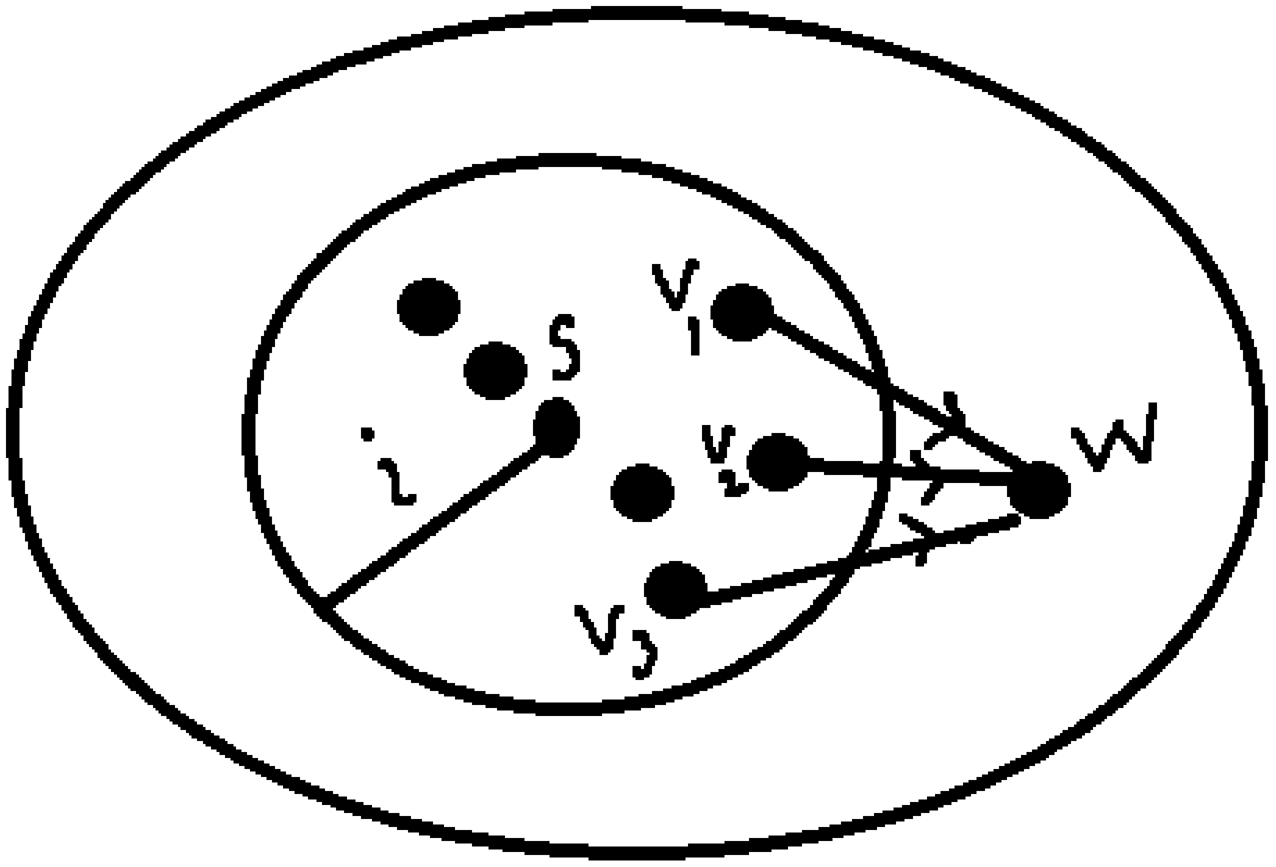
- Now consider a vertex v such that the shortest path from s to v contains only one edge, say (s,v).
- The edge (s,v) appears at some iteration of the second for loop in the first iteration of the main loop.
- At that point, d(v) is set correctly.
- Does that mean that all neighbors of s FINISH in one iteration?

Thinking about the Bellman-Ford Algorithm

- In that fashion, let every vertex v with a shortest path having at most i edges enter the FINISHED state at the end of i iterations.
- This certainly holds for i = 0. (and i =1 too!)
- Can we use induction to continue the proof?

The Proof

- In pictures…



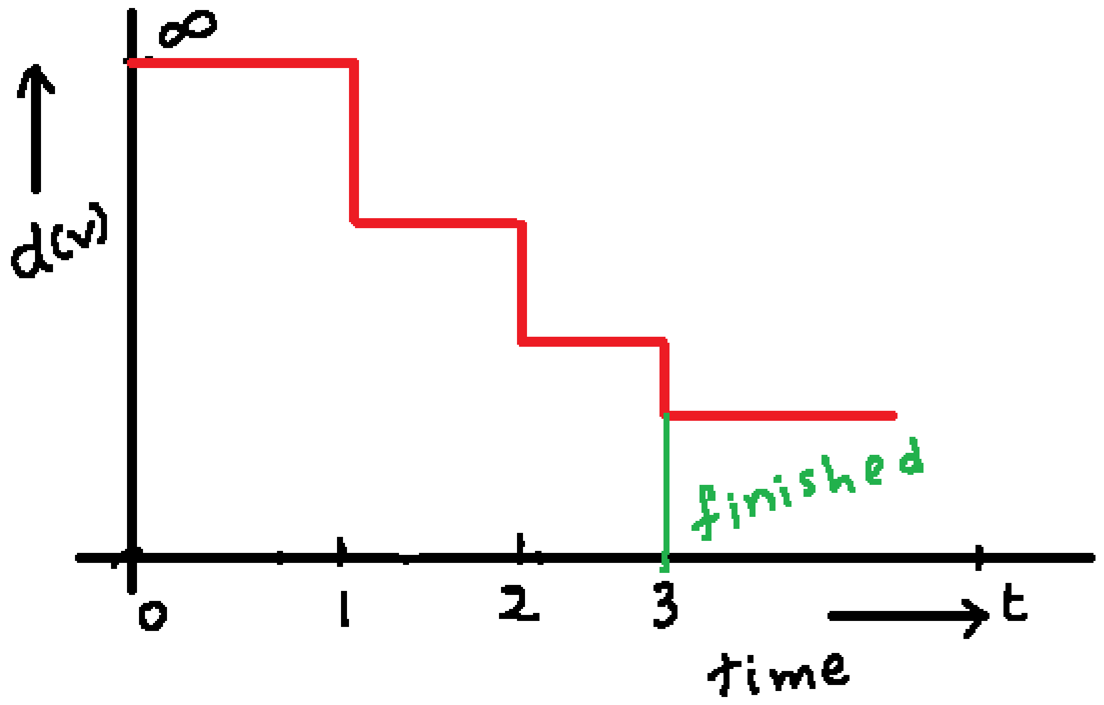
Algorithm SSSP

- The time taken by the Bellman-Ford algorithm is too high compared to that of BFS.
- Can we improve on the time requirement?
- Most of the time is due to

Algorithm SSSP

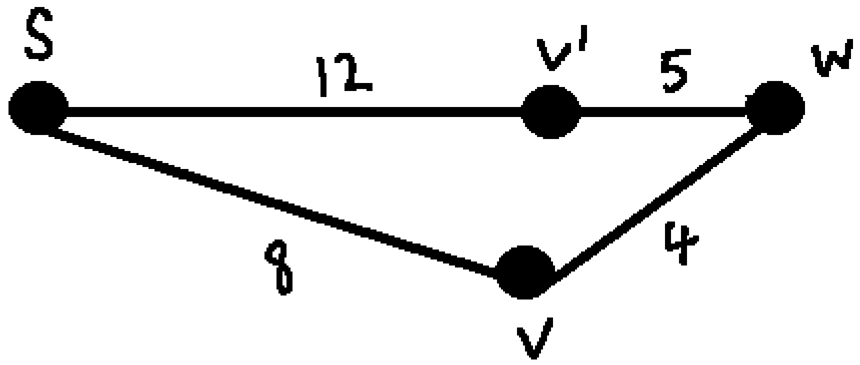
- The time taken by the Bellman-Ford algorithm is too high compared to that of BFS.
- Can we improve on the time requirement?
- Most of the time is due to
  - Repeatedly considering edges, and as a result
  - Updating d(v) possibly many times
- Need to know how to stop updating d(v) for any vertex v.
- This is what we will develop next.

To Improve the Runtime



- When is a vertex FINISHED?
- When no further shorter path can be found to v from s.
  - Equivalently, when d(v) can no longer decrease.

A Considered Edge



```
void process(e) /*e = (v,w)*/
begin
    if d(w) > d(v) + W(v,w) then
        d(w) = d(v) + W(v,w)
    end
end
```

- We say an edge e = (v, w) is **considered** if the above routine is executed for e.
- The impact is to possibly lower d(w), indicating that a **better** path to w from s is available via v.

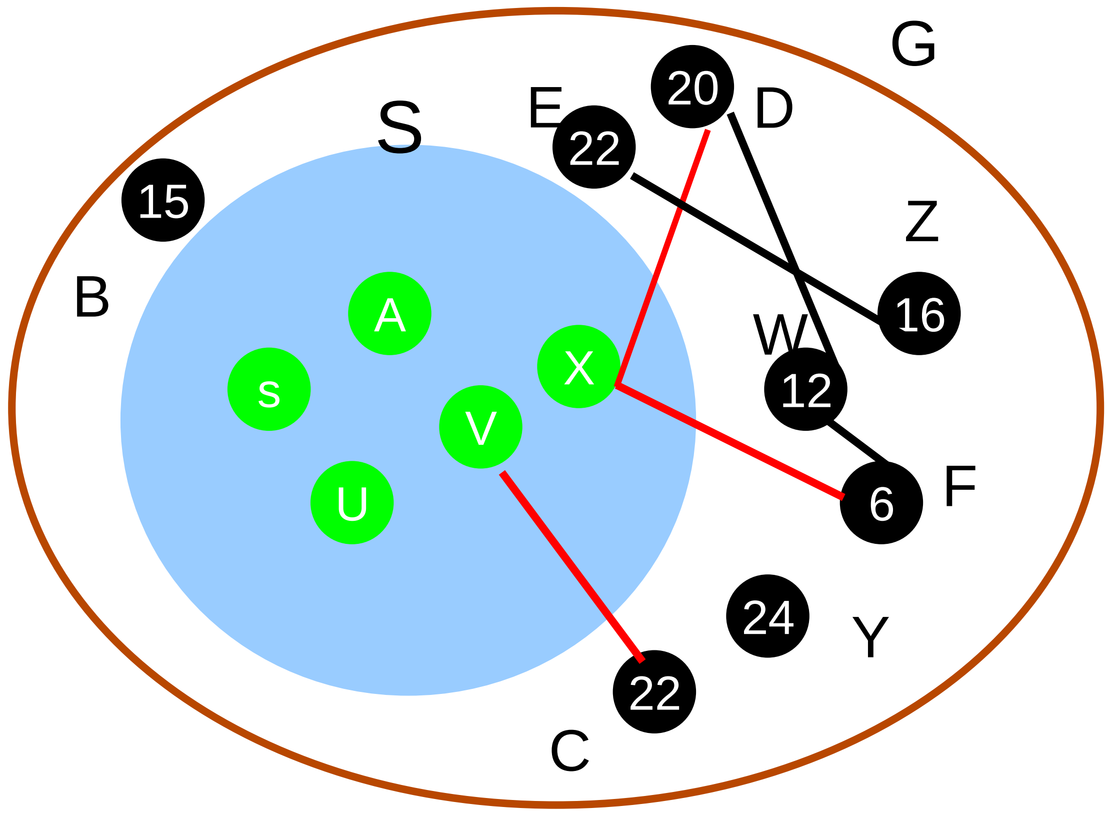
To Improve the Runtime

- For this to happen, consider the following.
  - A few vertices, say S, are FINISHED. By that we also mean that for any v in S, $d_s(v)$ CANNOT decrease any further.
  - Plus, all the edges with at least one endpoint in S are the only edges considered.
  - Other vertices in V \ S, have some d() value.

To Improve the Runtime

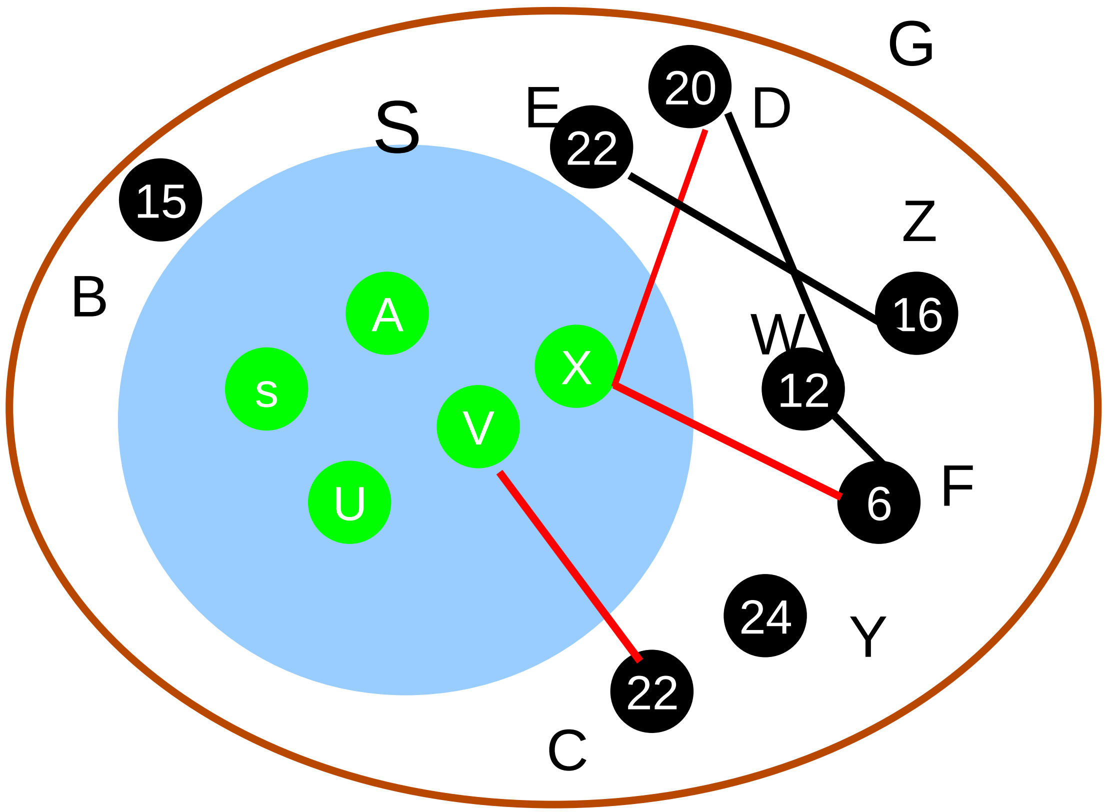
- For this to happen, consider the following.
  - Let each edge have a positive weight.
  - A few vertices, say S, are FINISHED.
  - Plus, all the edges with at least one endpoint in S are the only edges considered.
  - Other vertices in V \ S, have some d() value.
  - Which of these cannot improve d() any more?

The Setting



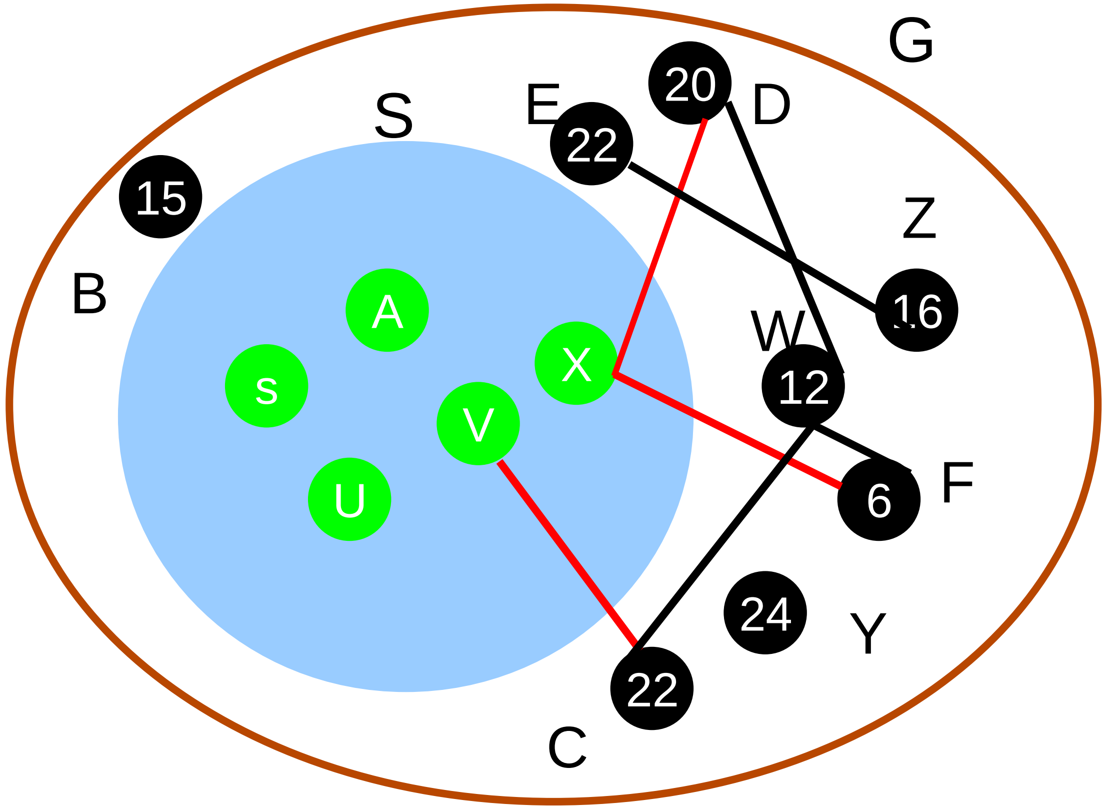
- Green vertices are FINISHED. No further changes to $d_s()$
- Red edges, edges with at least one end point as a green vertex are the ONLY edges PROCESSED.
- Numbers on black vertices indicate their d() value using only green vertices as intermediate vertices.

The Setting



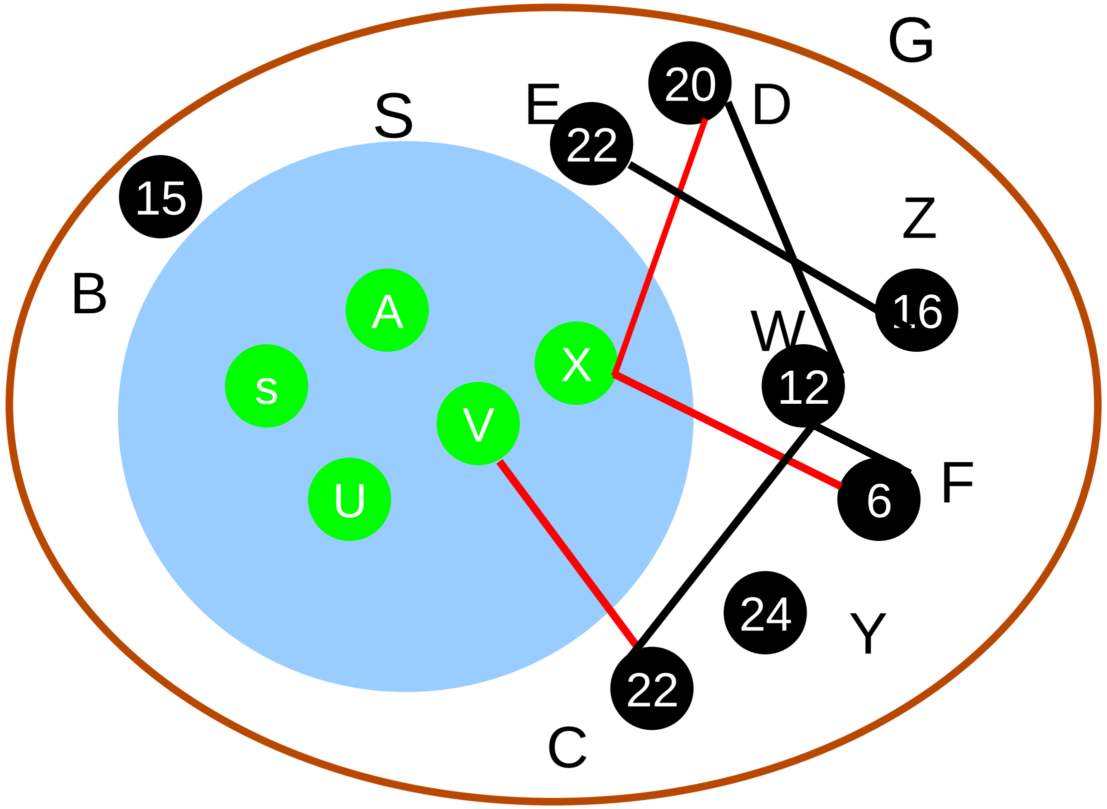
- Suppose we want to add one more vertex to the set S.
- Which of the black vertices is FINISHED?

The Setting



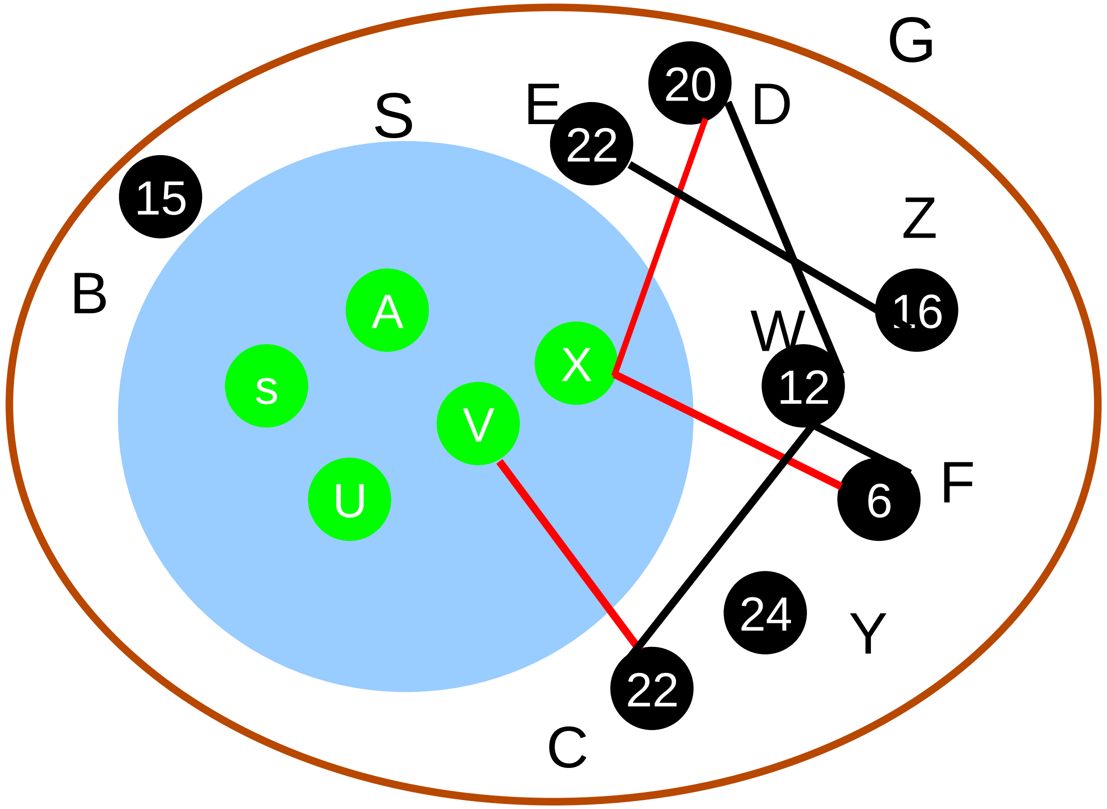
- Notice that there could be edges between the black vertices also.
  - None of them are processed so far.

The Setting



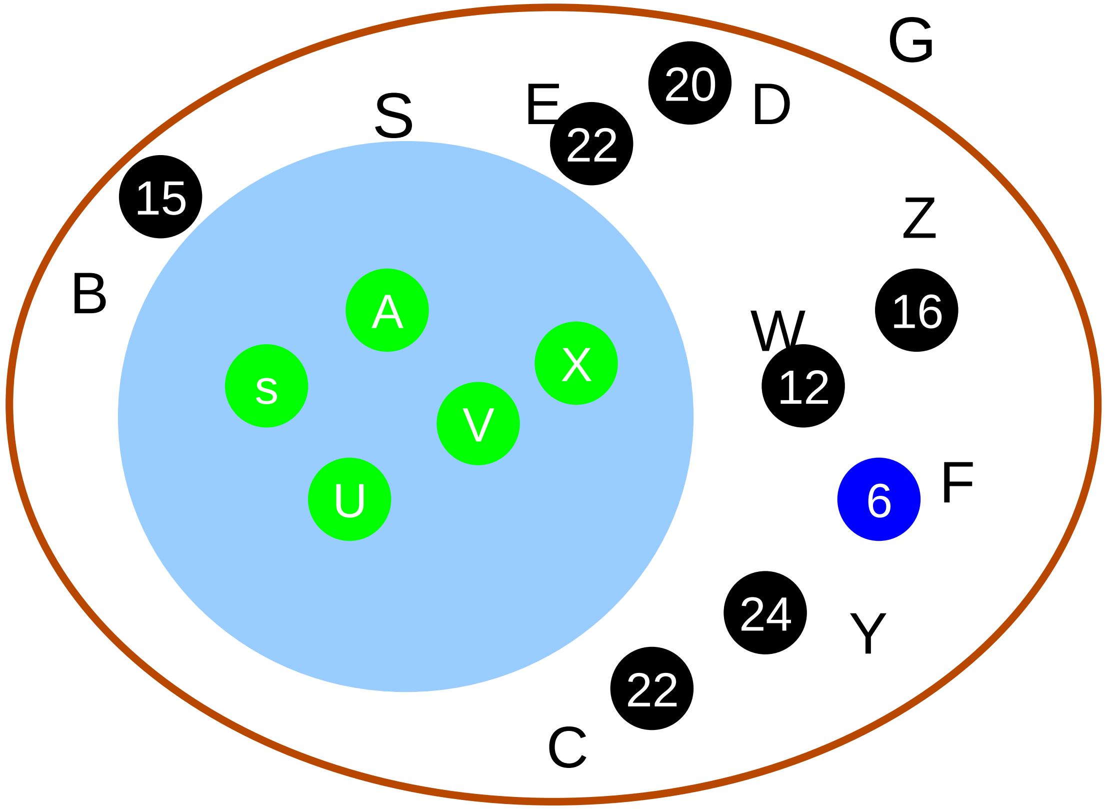
- Consider the vertex v with the smallest d() value among the black vertices.
- Any more decrease to d(v) would involve using at least one more edge between two black vertices.

The Setting



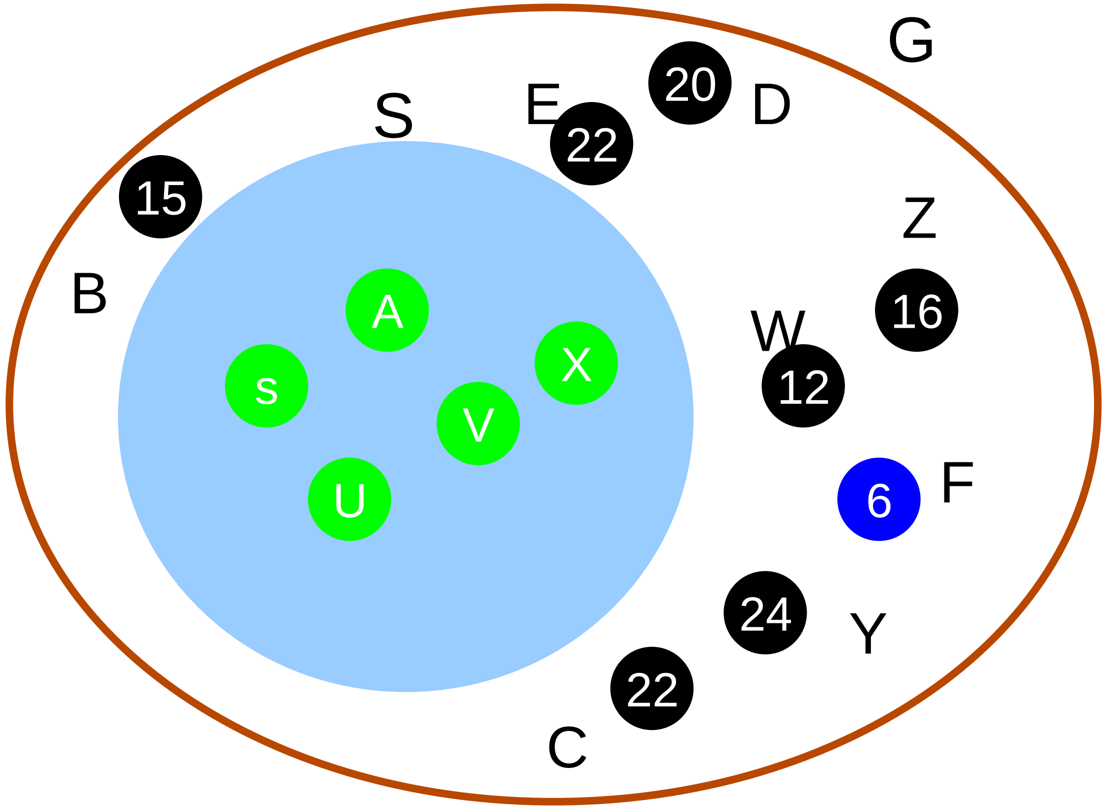
- Consider the vertex v with the smallest d() value among the black vertices.
- Any more decrease to d(v) would involve using at least one more edge between two black vertices.
- But all edge weights are positive.

The Setting



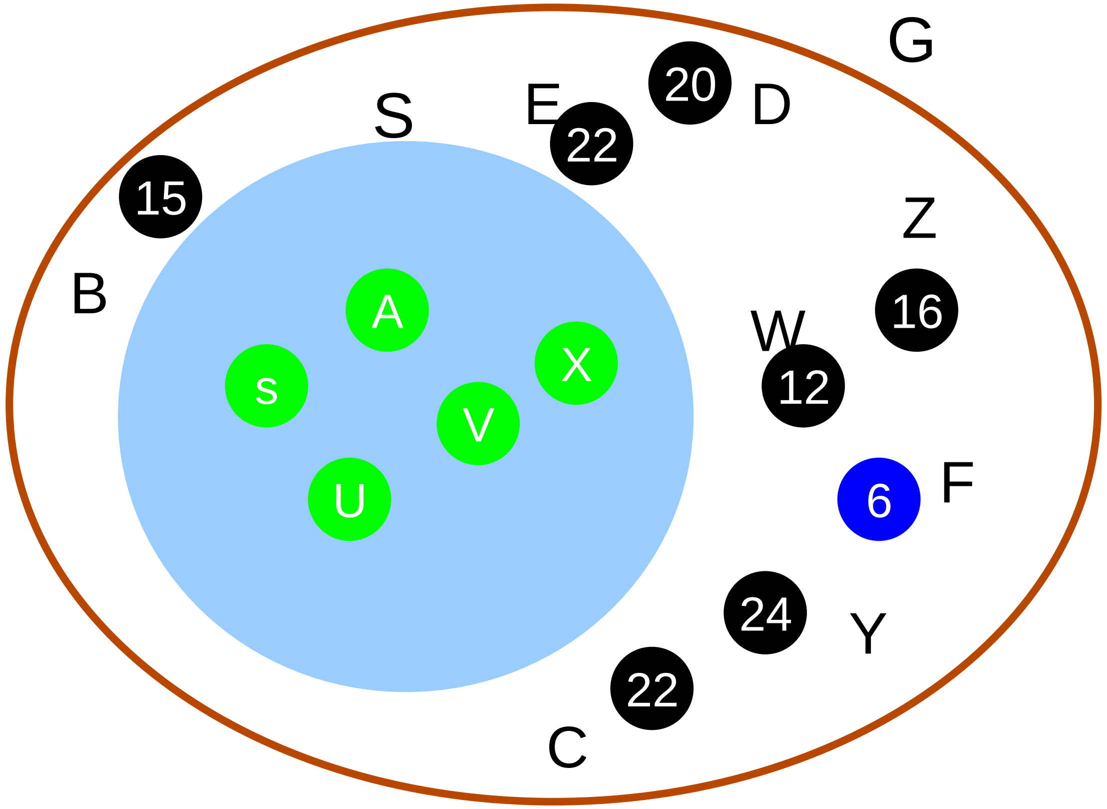
- Therefore, such a vertex with the smallest d() value among the black vertices can no longer decrease its d() value.

The Setting



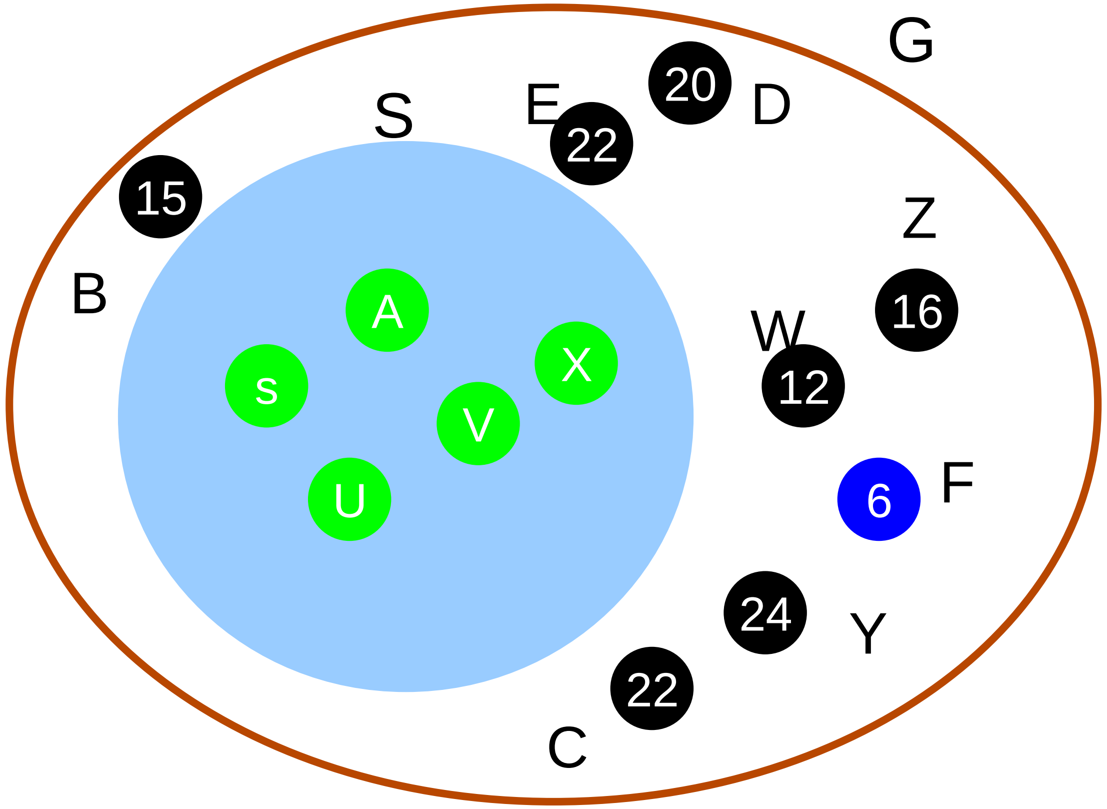
- Further, moving any other vertex w to S may mean that d_s(w) is not FINAL when w is moved to S.

The Setting



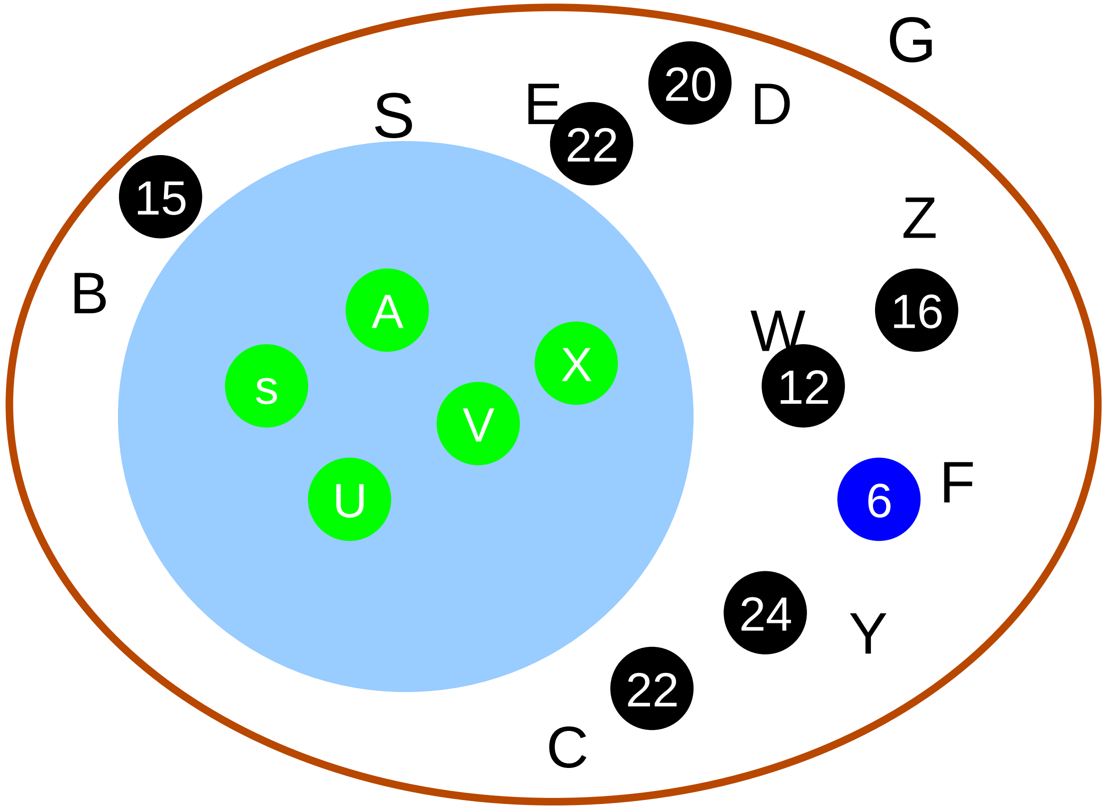
- Suggests that the vertex with the smallest d() value is FINISHED.
- Hence, can be moved to the set S.

The Setting



- But, how did we pick the set S so far?

The Setting

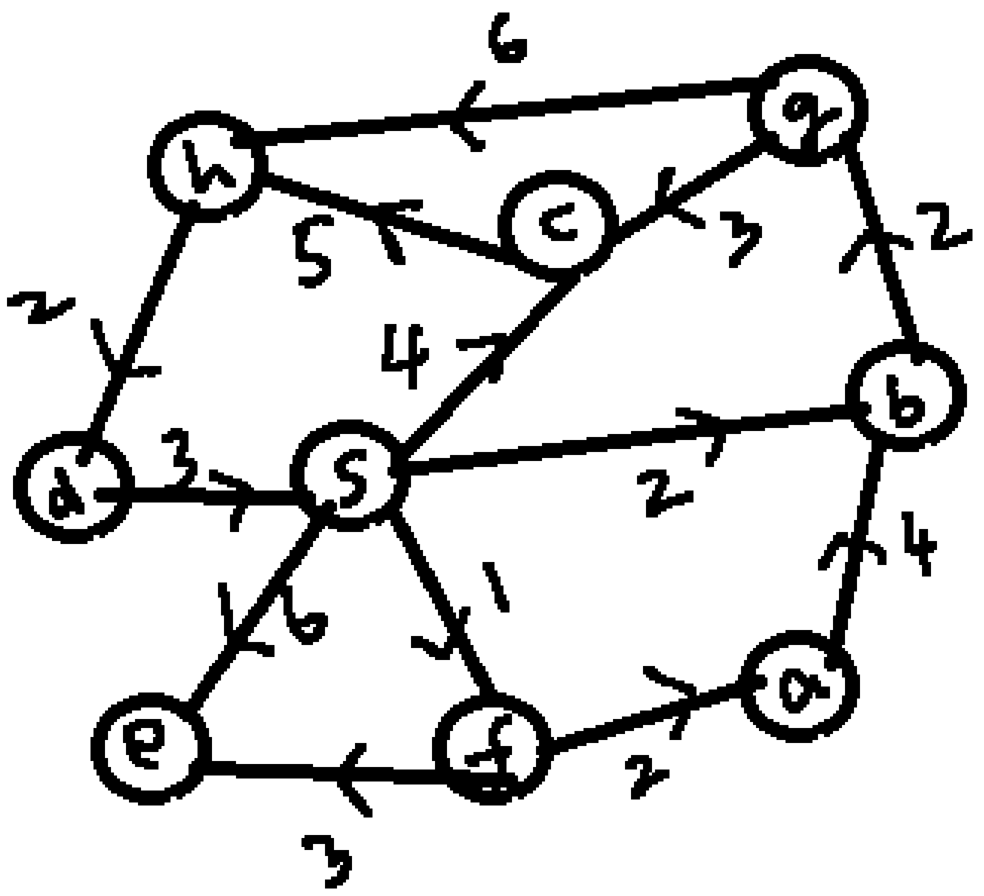


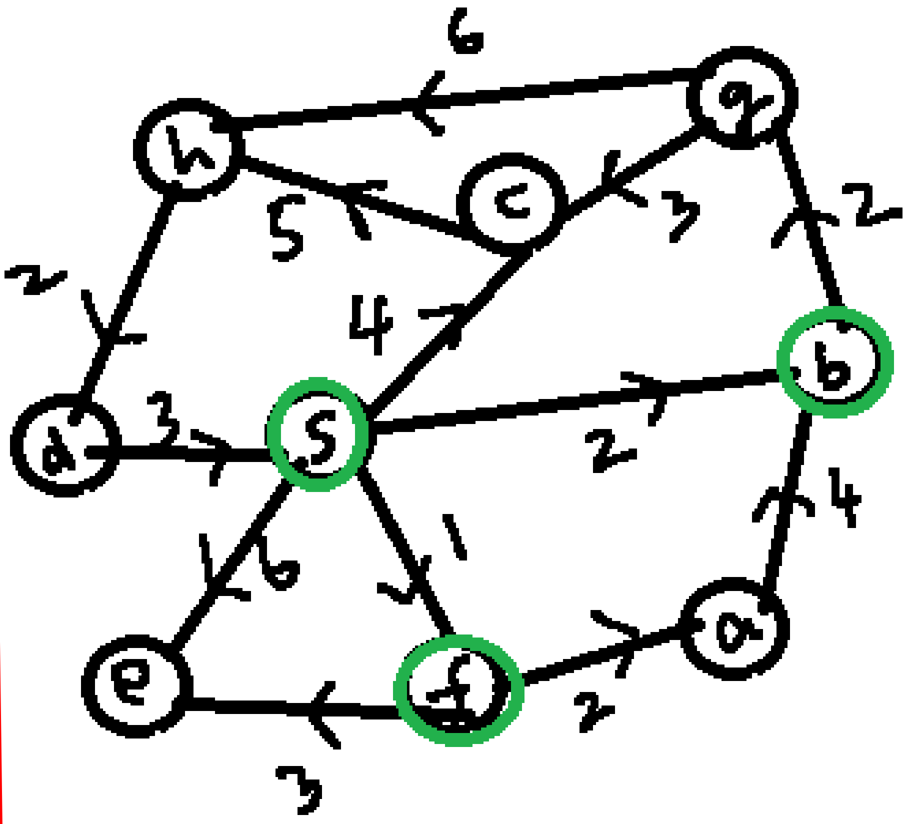
- Initially, only vertex s is in the set S.
  - As d(s) = 0.
- Incrementally, populate S with more vertices.

The Algorithm

- Can develop the above idea into an algorithm.
- The basic step in the algorithm is to proceed iteratively.
- Each iteration, one vertex is moved to set S according to the least d() rule.

Quick Exercise





The Algorithm

- What is the effect of moving a vertex v to S?
  - The neighbors of v may find a better path from s.
  - So, we will update d(w) for neighbors w of v, if necessary.
- Every neighbor of v?

The Algorithm

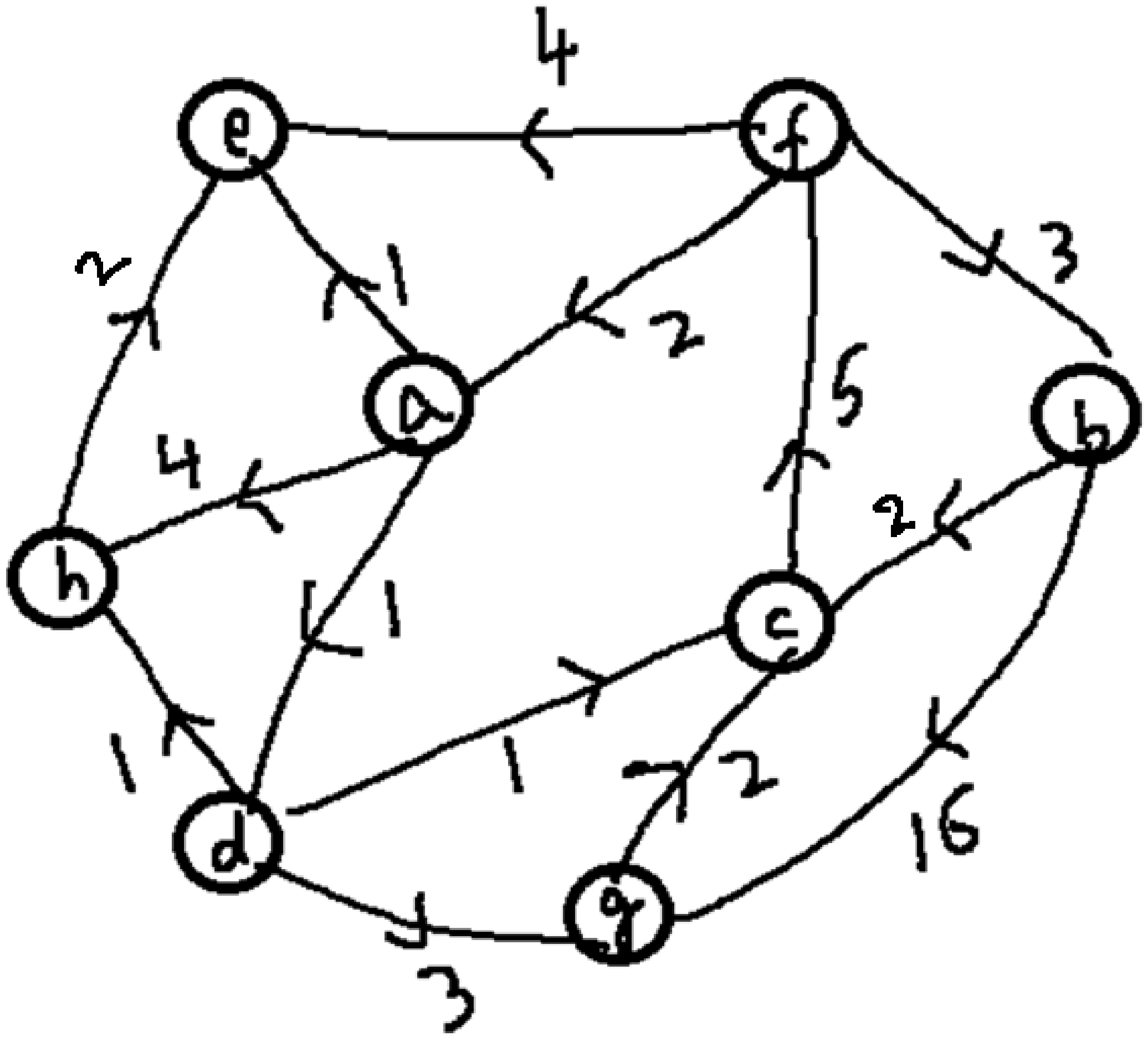
- What is the effect of moving a vertex v to S?
  - The neighbors of v may find a better path from s.
  - So, we will update d(w) for neighbors w of v, if necessary.
- Every neighbor of v?
- No, those in S can never decrease their d() value.
  - So, check only neighbors w that are not in S.

The Algorithm

```
Algorithm SSSP(G, s)
begin
    for all vertices v do
        d(v) = ∞; p(v) = NIL;
    end-for
    d(s) = 0;
    for n iterations do
        v = the vertex with the least d() value among V \ S;
        Add v to S
        for each neighbor w of v in V \ S do
            d(w) = min { d(w), d(v) + W(v,w) }
        end-for
    end-for
```

Example

- Start vertex is c.



The Algorithm

- The program resembles the BFS approach.
  - Instead of a queue, maintain order on the vertices according to their d() values.
  - Need three states now.
- The above algorithm is essentially the Dijkstra's algorithm.
- How to analyze the above algorithm?
- Requires answers to a few questions.
- How to know which vertex in V \ S has the least d() value?
- How to know if a vertex is in V \ S or not?

The Algorithm

- How to know which vertex in V \ S has the least d() value?
  - Think of the binary heap.
  - A heap supports an efficient deleteMin().
  - Use d() values as the priority.
- How to know if a vertex is in V \ S or not?
  - Remember the state of vertices.
- How to update the d() value of a vertex?
  - Can simply use operation DecreaseKey(v, δ) to update d(v).
  - The above choice of a heap solves also this problem.

Analyzing the Algorithm

- To analyze the algorithm when using a heap
  - How many DeleteMin() operations are performed on the heap?
  - How many DecreaseKey() operations are called?

Analyzing the Algorithm

- To analyze the algorithm when using a heap
  - How many DeleteMin() operations are performed on the heap?
  - Answer: At most n.
  - How many DecreaseKey() operations are called?
  - Answer: At most m

- Each DeleteMin() takes O(log n) time
- Each DecreaseKey() operation takes O(log n) time.
- So, the total time is O((n+m)log n).

Advanced Solutions

- Advanced data structures exist to decrease the runtime to O(m + n log n).
  - Fibonacci heaps, ...
- A good case study of how to separate algorithm from the data structure.
  - Any data structure that supports deleteMin and decreaseKey can be used
  - The algorithm will still be correct.

Exercise

- Pick any of the applicable data structures of your choice and find the runtime of Dijkstra’s algorithm with your choice.

Yet Another Traversal

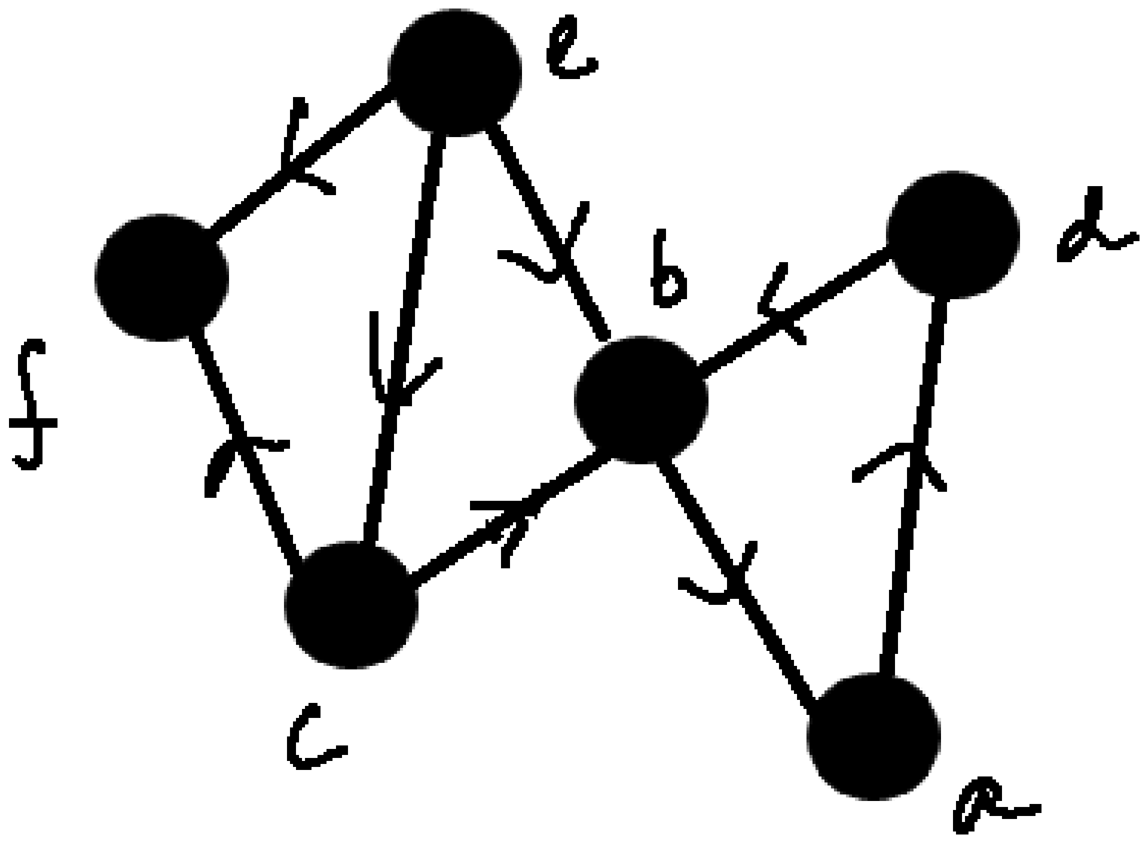
- In BFS, vertices not reachable from s are never listed.
- So the entire graph may not be visited at all.
- We will study yet another traversal mechanism for graphs.
  - Visits the entire graph!!
- This is called Depth First Search (DFS) and has important applications.
- Several graph algorithms use DFS as a subroutine.

Yet Another Traversal

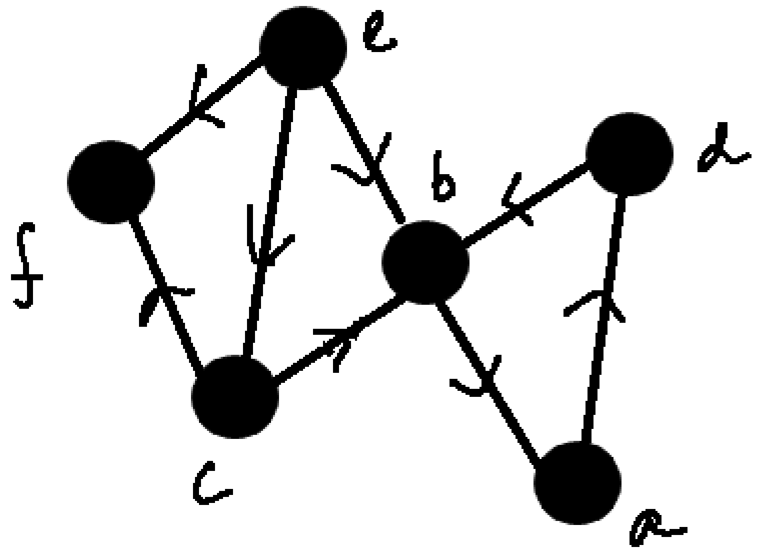
- One way to think of the new traversal is to consider using a stack instead of a queue in BFS.
- We will also add one more operation to the stack apart from push and pop.
  - Peek() on a stack S returns the top of the stack without deleting the top element.
  - If the stack is empty, returns NULL.
- Let us understand by an example.

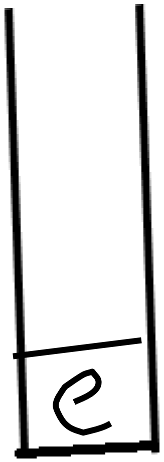
DFS

- Let us start from vertex e.

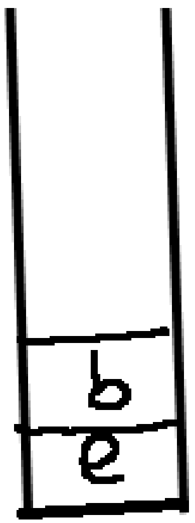


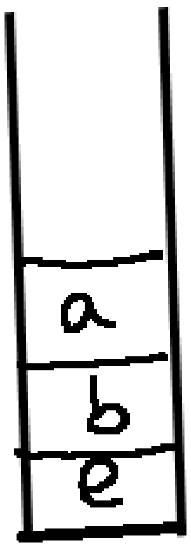
DFS

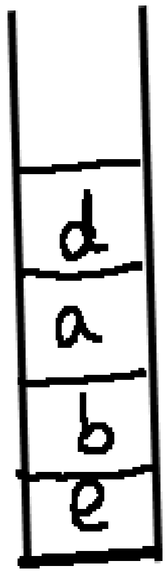




$\Pi(e) = \text{nil}$





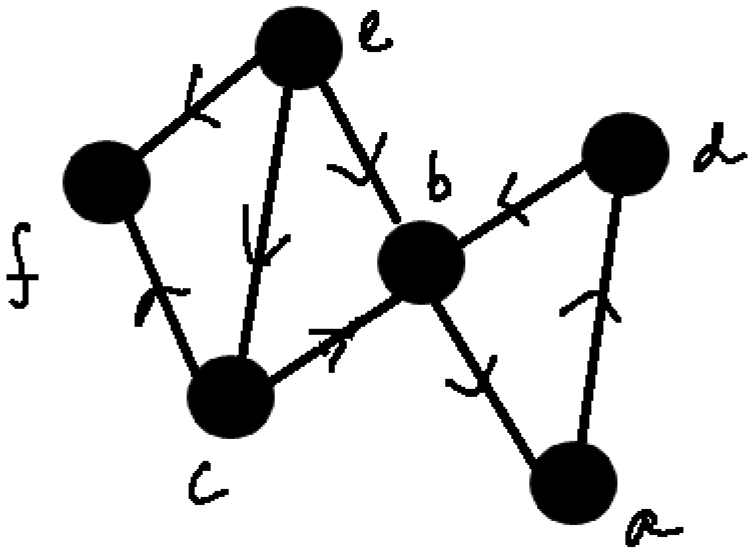


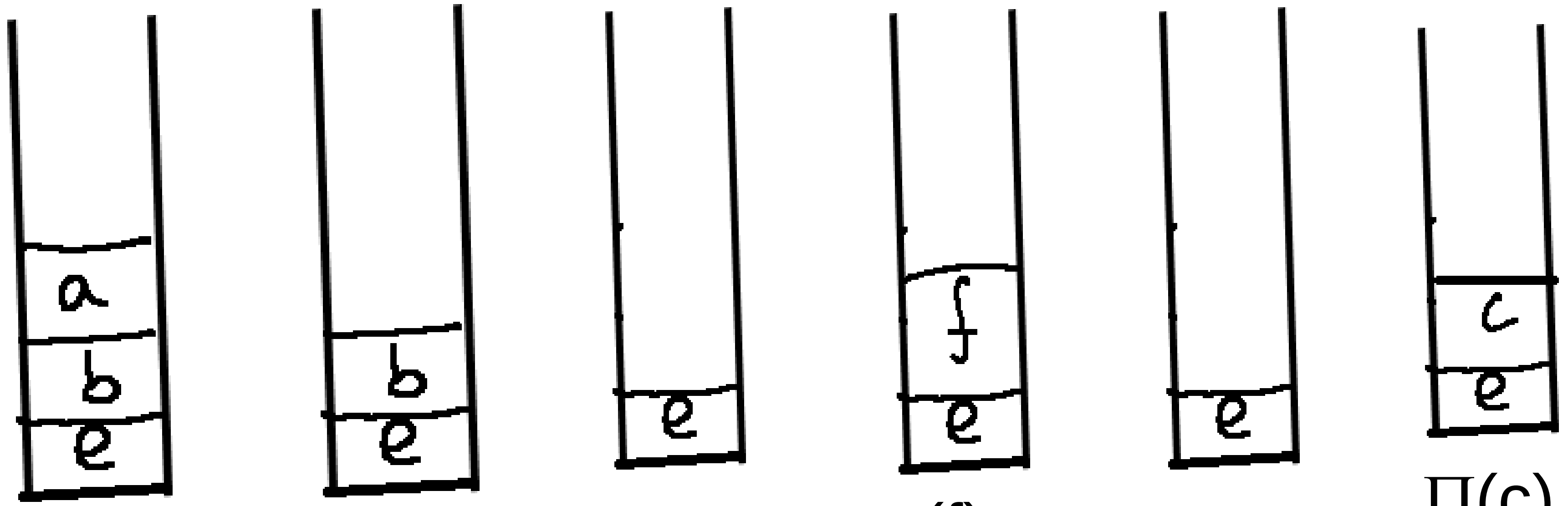
$\Pi(b) = e$

$\Pi(a) = b$

$\Pi(d) = a$

DFS





$\Pi(f) = e$

$\Pi(c) = e$

Depth First Search

- The idea of DFS is as follows.
- Start from a specified start vertex, say s.
- Explore from s as deep as possible. This suggests that we go from s, to one of its out neighbors x, to a neighbor of x, ...
- When to stop? When there are no new out-neighbors to explore from a given point.

DFS

- Further, when the stack is empty, and there are still vertices that are not visited, pick another start vertex.
- Repeat until all vertices are visited.
- A big departure from BFS, but the goals are different.
    - In DFS, we aim to understand the structure of the graphs. To be done via several auxiliary information recorded during DFS.
    - In BFS, the idea is to find shortest paths.

Depth First Search

- Alike BFS, have to keep track of the state of a vertex.
- A vertex can be in three states: VISITED, NOT_VISITED, IN_STACK
- Normal to associate colors to these states such as
  - VISITED – GREEN
  - NOT_VISITED – RED
  - IN_STACK – ORANGE
- Why the third state? Same reason as BFS.

DFS

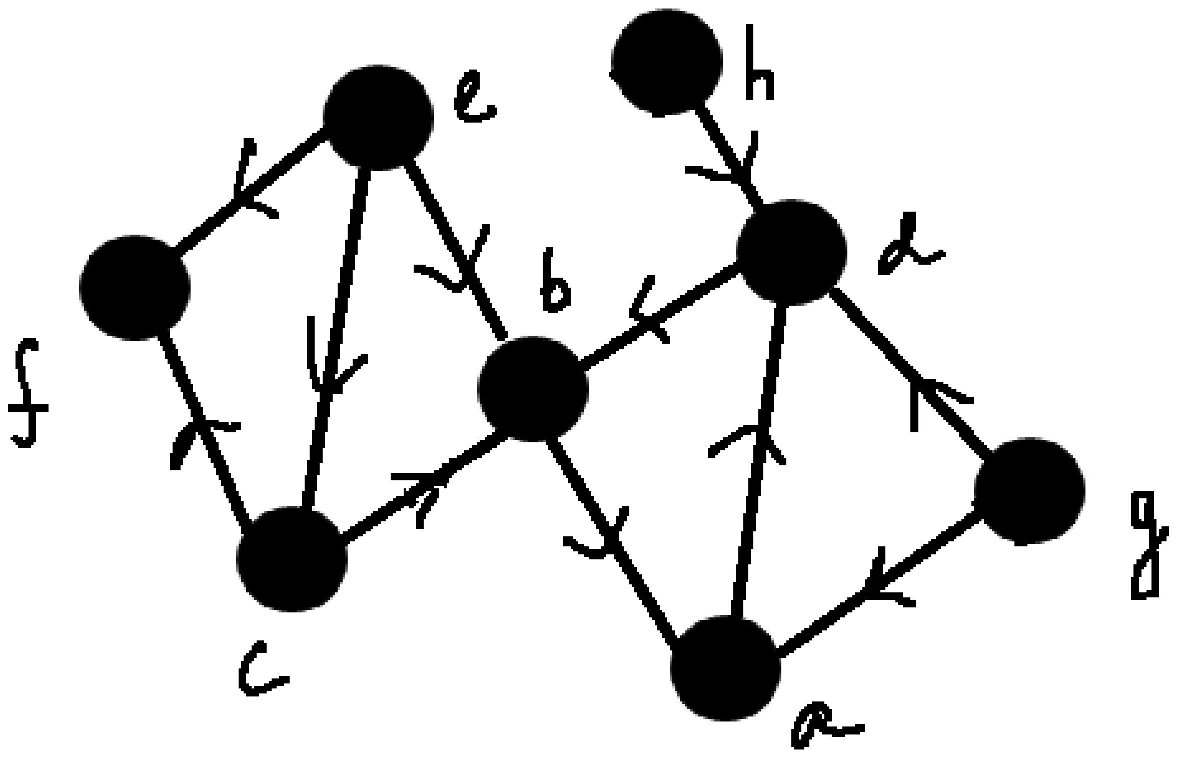
- In a programmatic sense, can use recursion to manage the stack.
- So, the modified program looks as below.

DFS

```
Procedure DFS(G)
Begin
    for each vertex v do
        $\pi$(v) = NIL;
        state(v) = NOT_VISITED;
    end-for
    for each vertex v do
        if state(v) = NOT_VISITED then
            state(v) =  IN_STACK
            VisitDFS(v)
        end-if
    end-for
End.
```

```
Procedure VisitDFS(v)
Begin
    for each out-neighbor w of v do
        if state(w) = NOT_VISITED then
            $\pi$(w) = v;
            state(w) = IN_STACK
            VisitDFS(w)
        end-if
    end-for
    state(v) = VISITED;
End.
```

DFS – An Example



Discovery and Finish Times

- With every vertex, can also associate start and finish times.
- Start with time = 0 at the beginning.
- Associate (record) time for the following events
  - A vertex changes state from NOT_VISITED to IN_STACK. Meaning a new vertex is discovered
  - A vertex in IN_STACK state changes state to VISITED. Meaning, a discovered vertex finishes processing.

Discovery and Finish Times

- The first time a vertex v changes state from NOT_VISITED to IN_STACK, the current time is recorded as the discovery time of v,
- denoted d(v)
- The time at which vertex v changes state from IN_STACK to VISITED is recorded as the finish time of v
- denoted f(v)

DFS

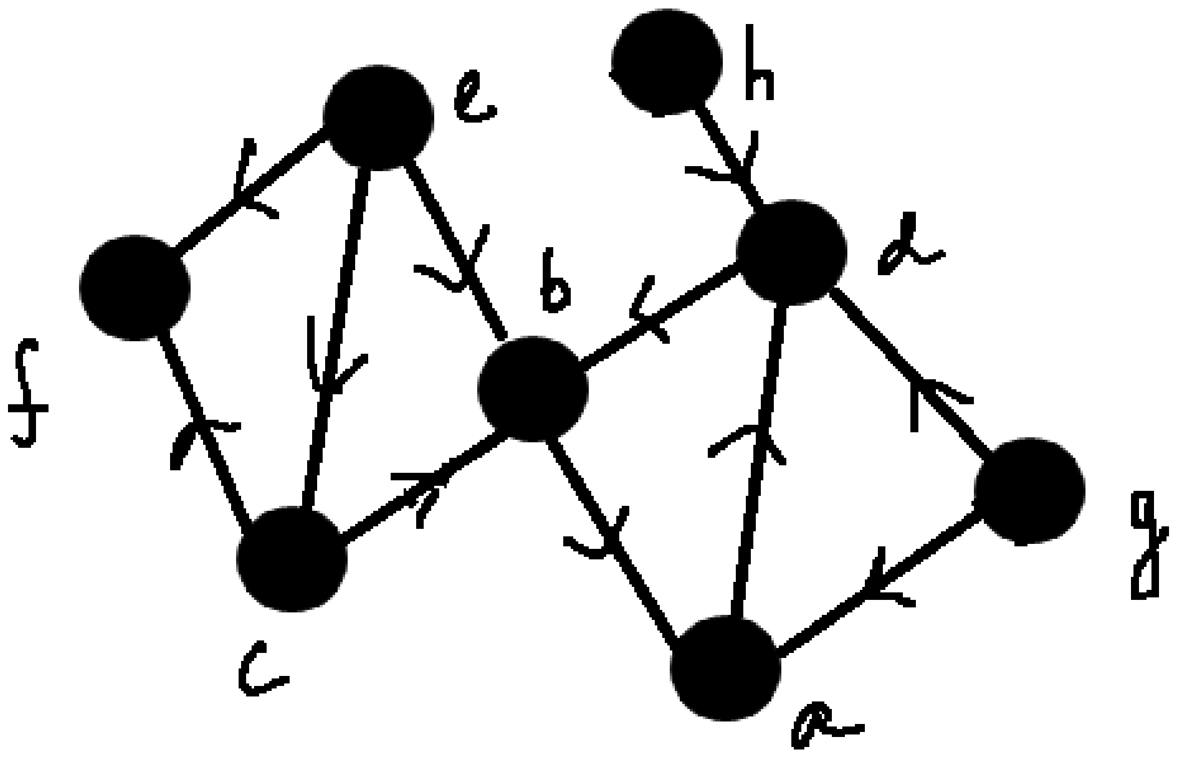
```
Procedure DFS(G)
Begin
    for each vertex v do
        state(v)  = NOT_VISITED
        $\pi$(v) = NIL;
    end-for
    time = 1
    for each vertex v do
        if state(v) = NOT_VISITED then
            state(v) = IN_STACK
            d(v) = time++;
            VisitDFS(v)
        end-if
    end-for
End.
```

```
Procedure VisitDFS(v)
Begin
    for each neighbor w of v do
        if state(v) = NOT_VISITED then
            $\pi$(w) = v;
            state(w) = IN_STACK
            d(w) = time++
            VisitDFS(w)
        end-if
    end-for
    state(v) = VISITED;
    f(v) = time++;
End.
```

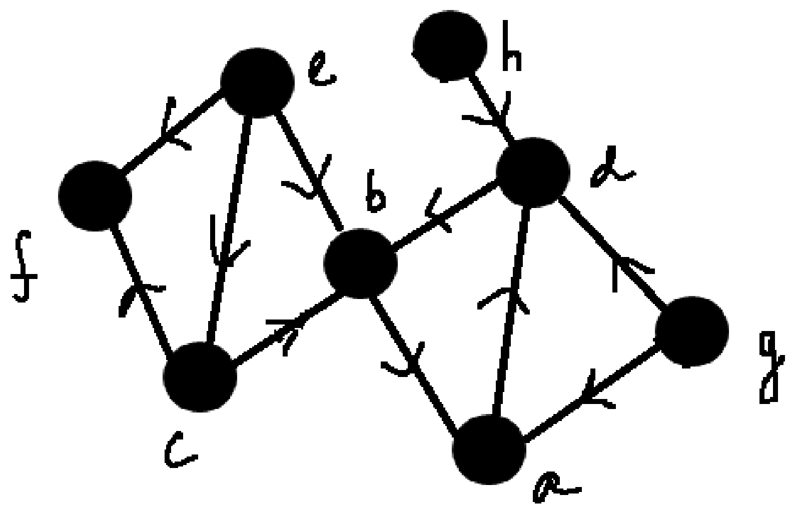
Discovery and Finish Times

- Several graph properties can be observed using the d() and the f() times.
- Interesting algorithms can be designed relying mostly on d() and f() times.
- We will see at least one such example later.

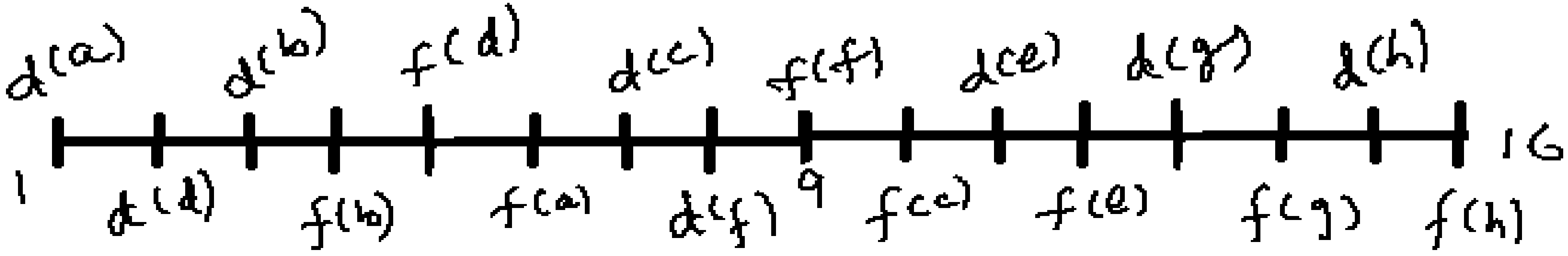
Discovery and Finish Times – Example



DFS – Complete Example



Vertex	Discovery Time	Finish Time
a	1	6
b	3	4
c	7	10
d	2	5
e	11	12
f	8	9
g	13	14
h	15	16



Classifying Edges

- Recall the edge classification done for BFS. Can do so also for DFS.
- The edges of E_π are also called as tree edges.
- The edges of $E_N := E \setminus E_\pi$ are called as non-tree edges.
- These edges can be further classified as follows.

Classifying Edges

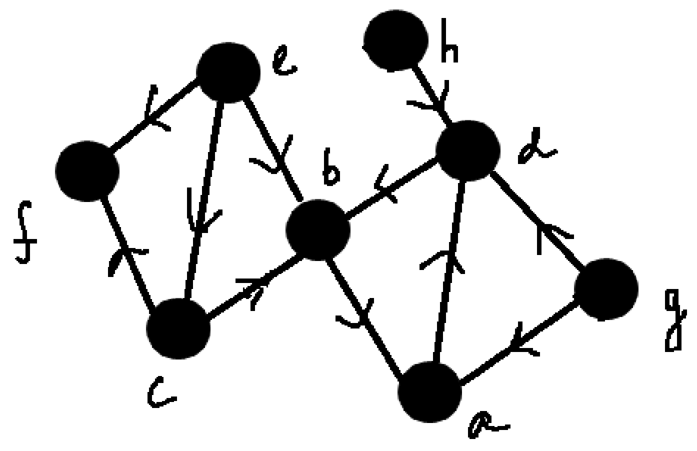
- Back Edges
  - An edge (u, v) ∈ E_N is called as a back edge if v is an ancestor of u in the DFS forest.
  - In other words, u can be reached from v using tree edges, but there is an edge from u to v also.
- Forward Edge
  - Edges (u, v) which connect a vertex u to a descendant of u in the DFS forest. (v is a descendant of u)
  - In other words, v can be reached from u using tree edges, but there is an edge from u to v also.

Classifying Edges

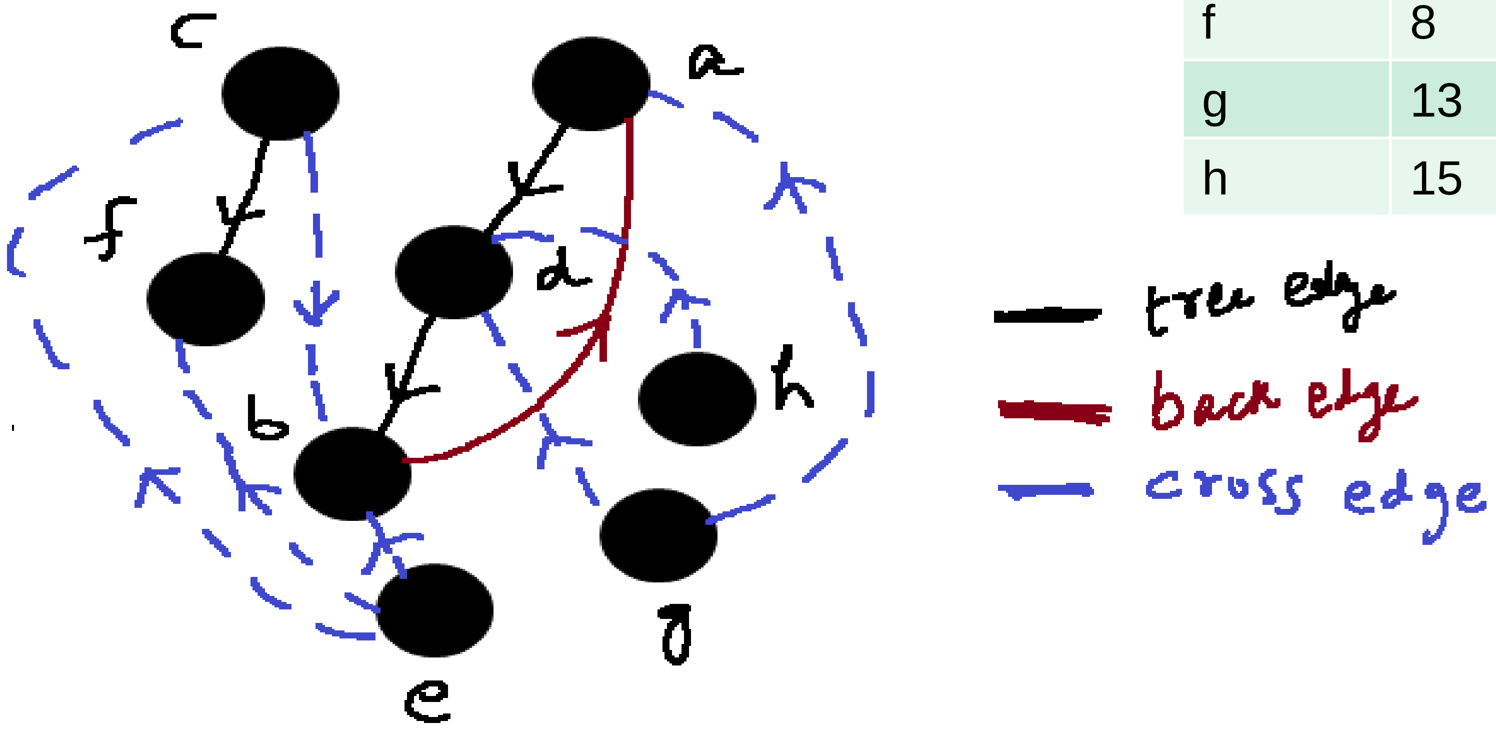
- Cross Edges : Edges (u, v) where u and v do not share any ancestor/descendant relationship.

- Depending on the type of the edge, certain relations between d() and f() values of the endpoints hold.
- If (u,v) is a cross edge then the intervals [d(u), f(u)] and [d(v], f(v)] do not overlap.
- Find other such relations.

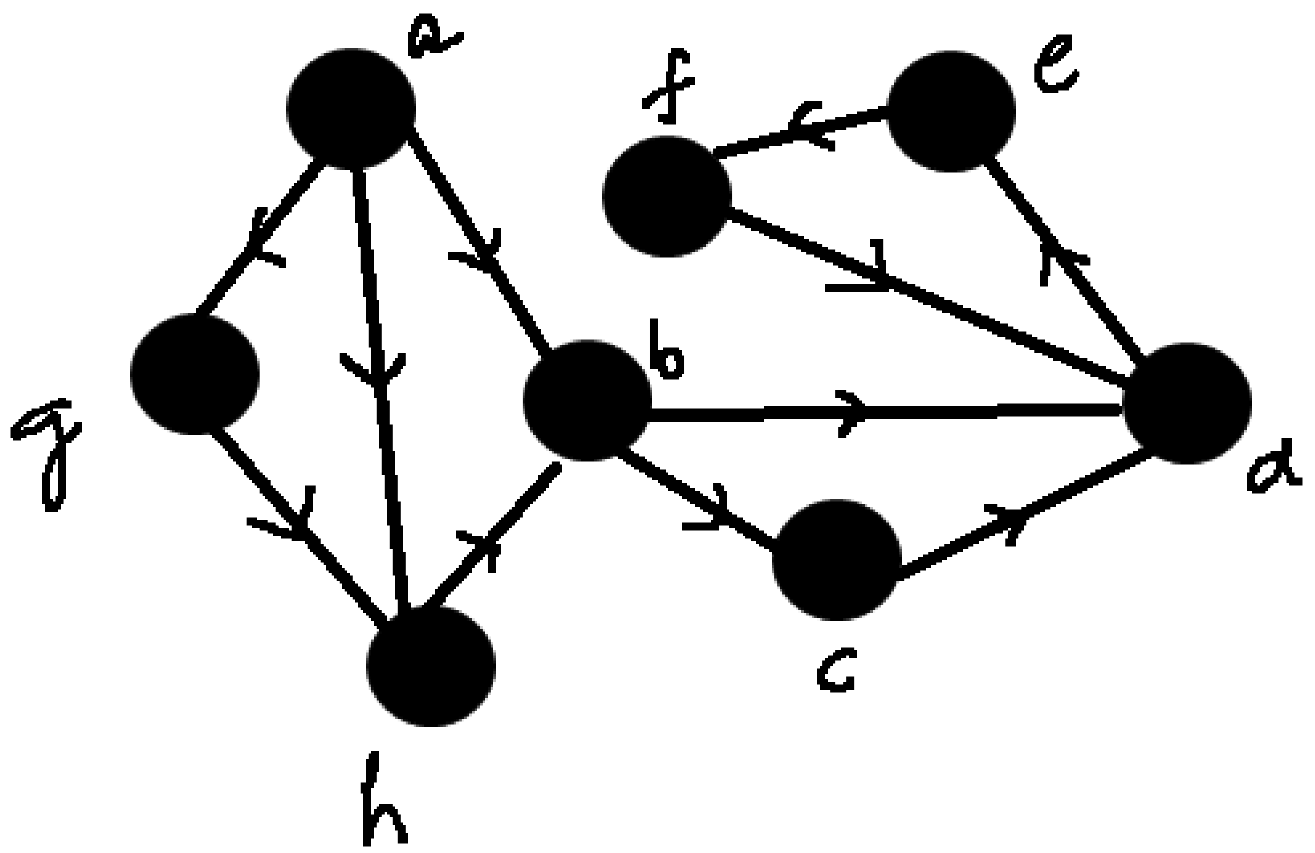
Example



Vertex	Discovery Time	Finish Time
a	1	6
b	3	4
c	7	10
d	2	5
e	11	12
f	8	9
g	13	14
h	15	16

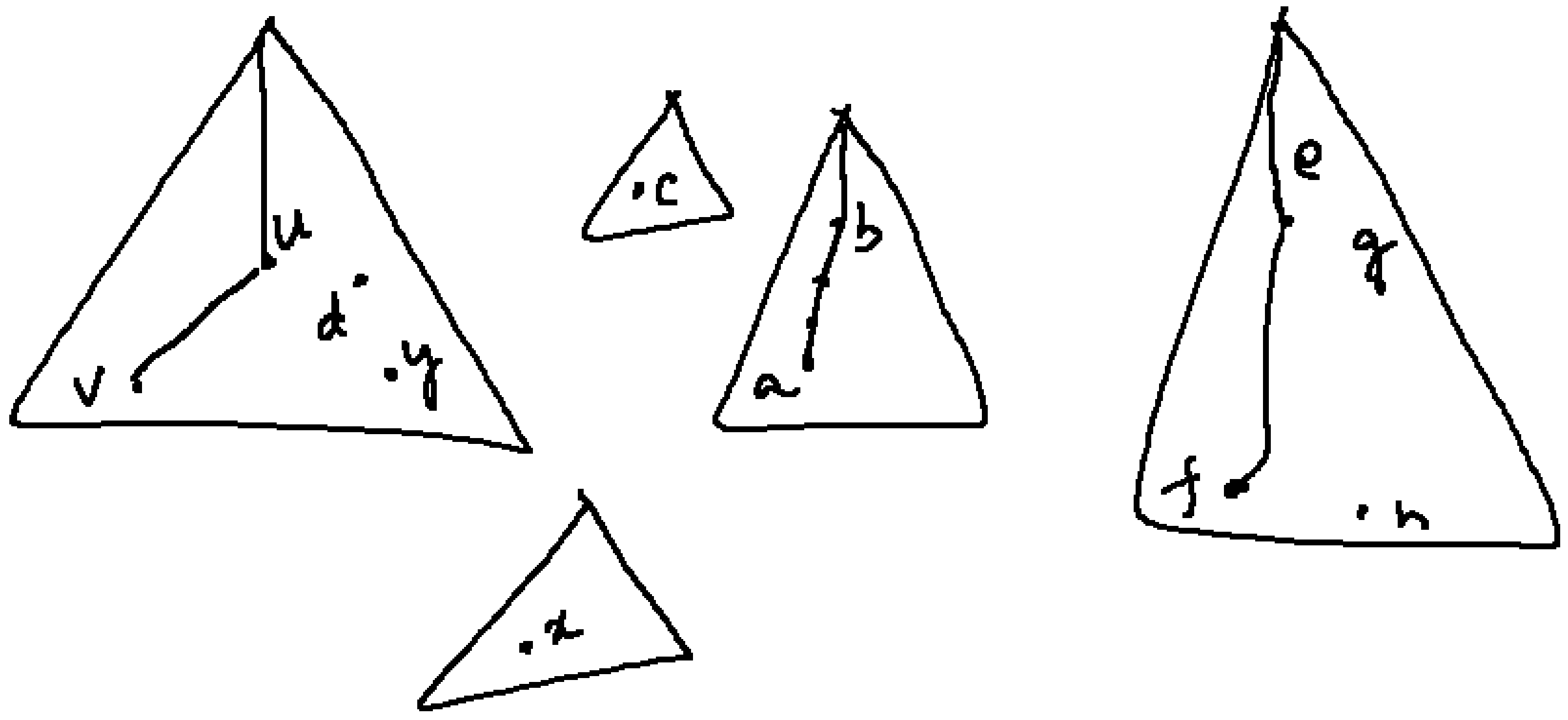


DFS Example

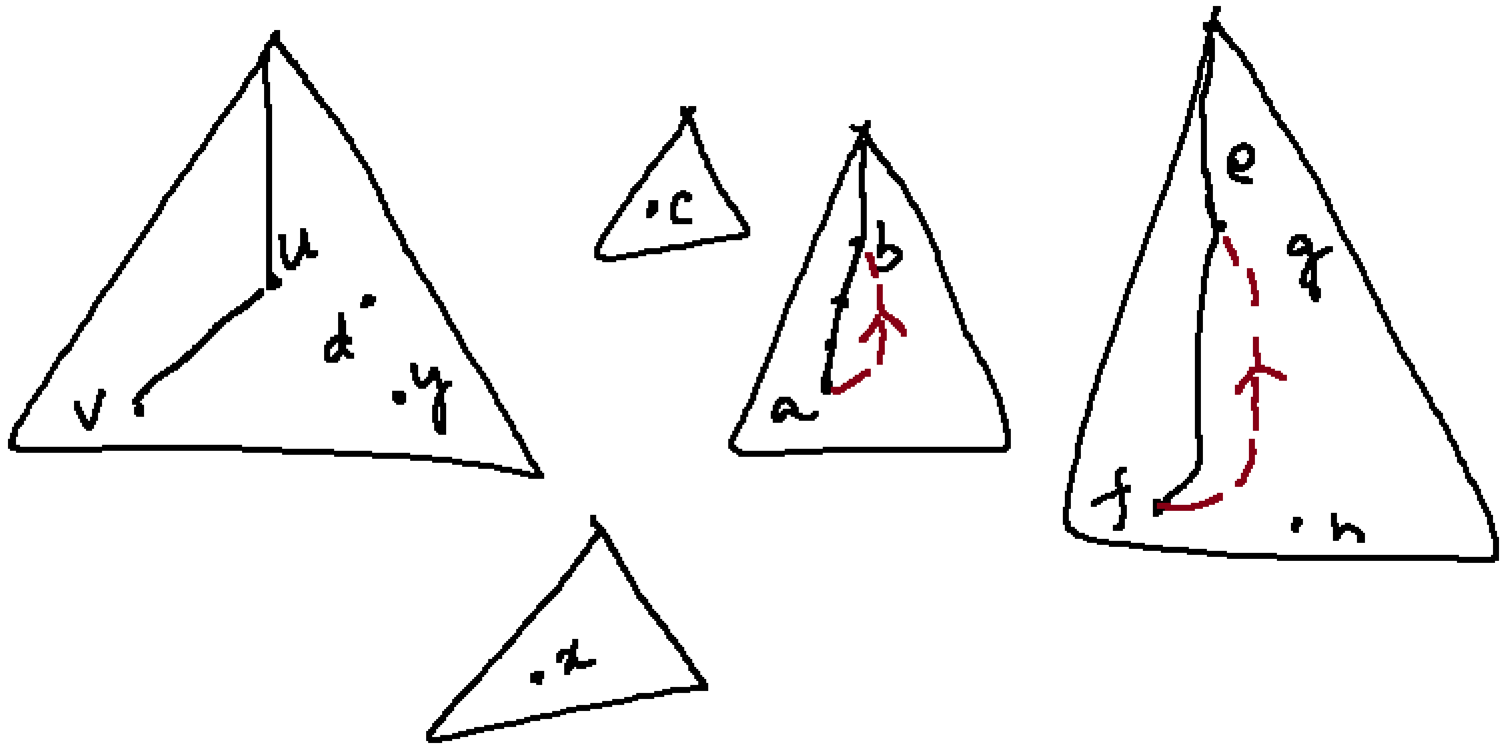


Edge Classification

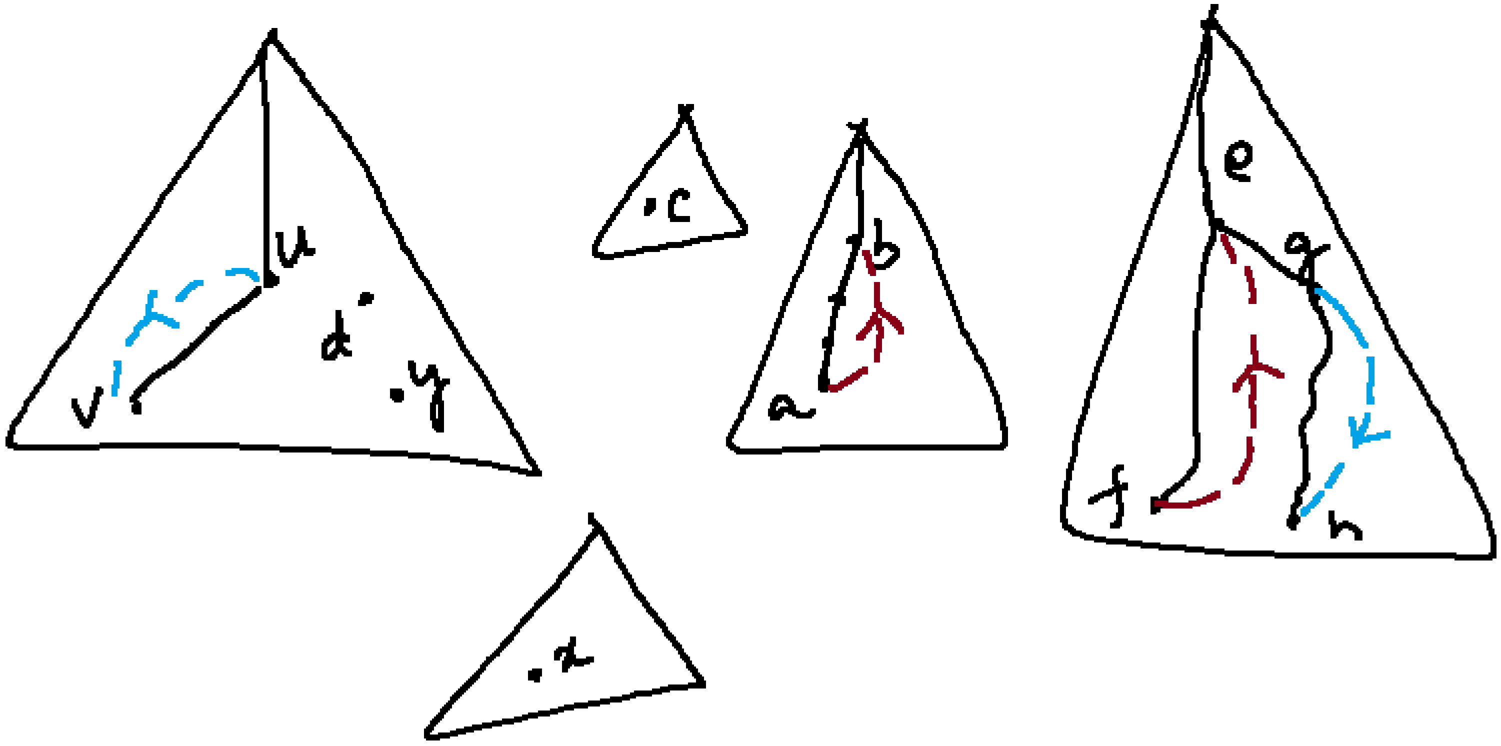
- Another attempt to explain edge classification according to DFS.
- Consider the following DFS forest.



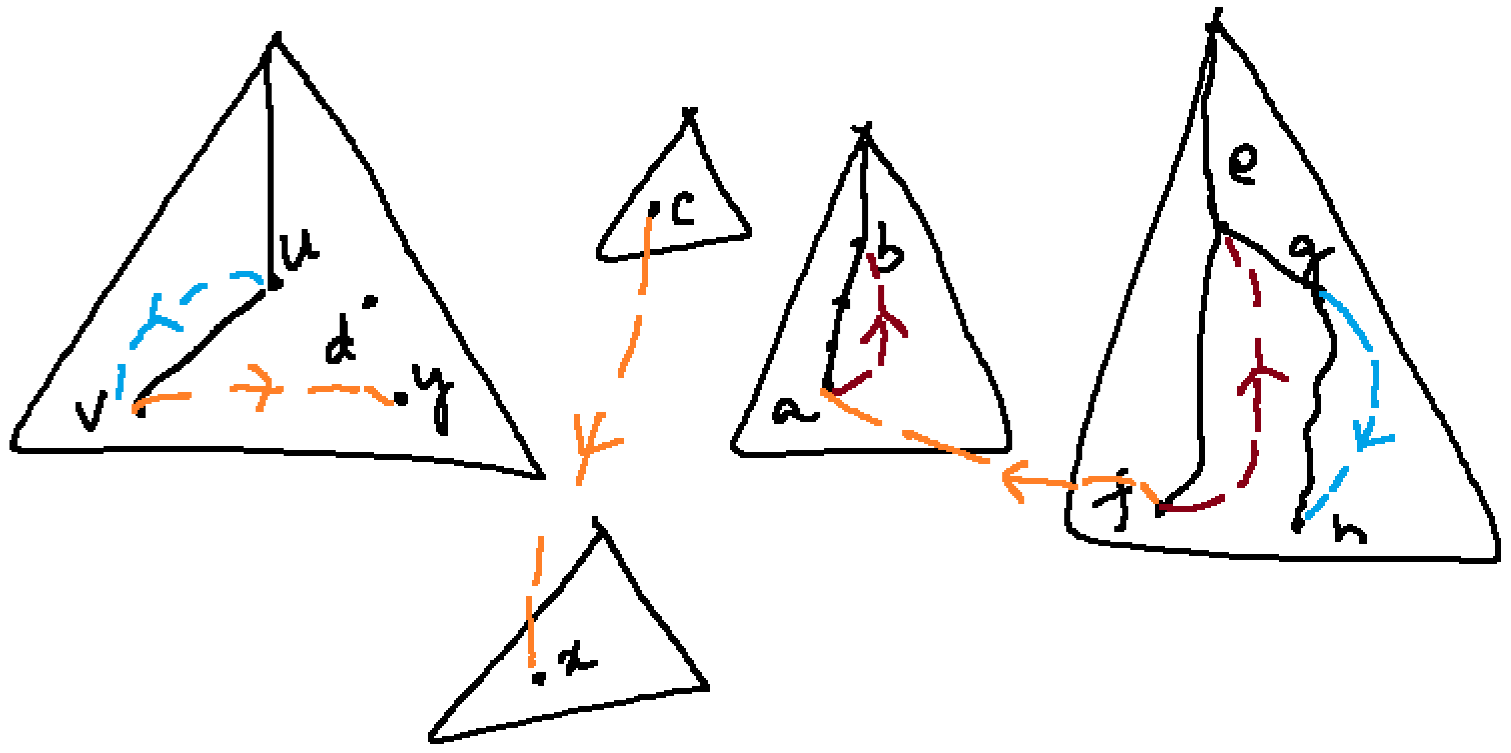
Back Edges



Forward Edges



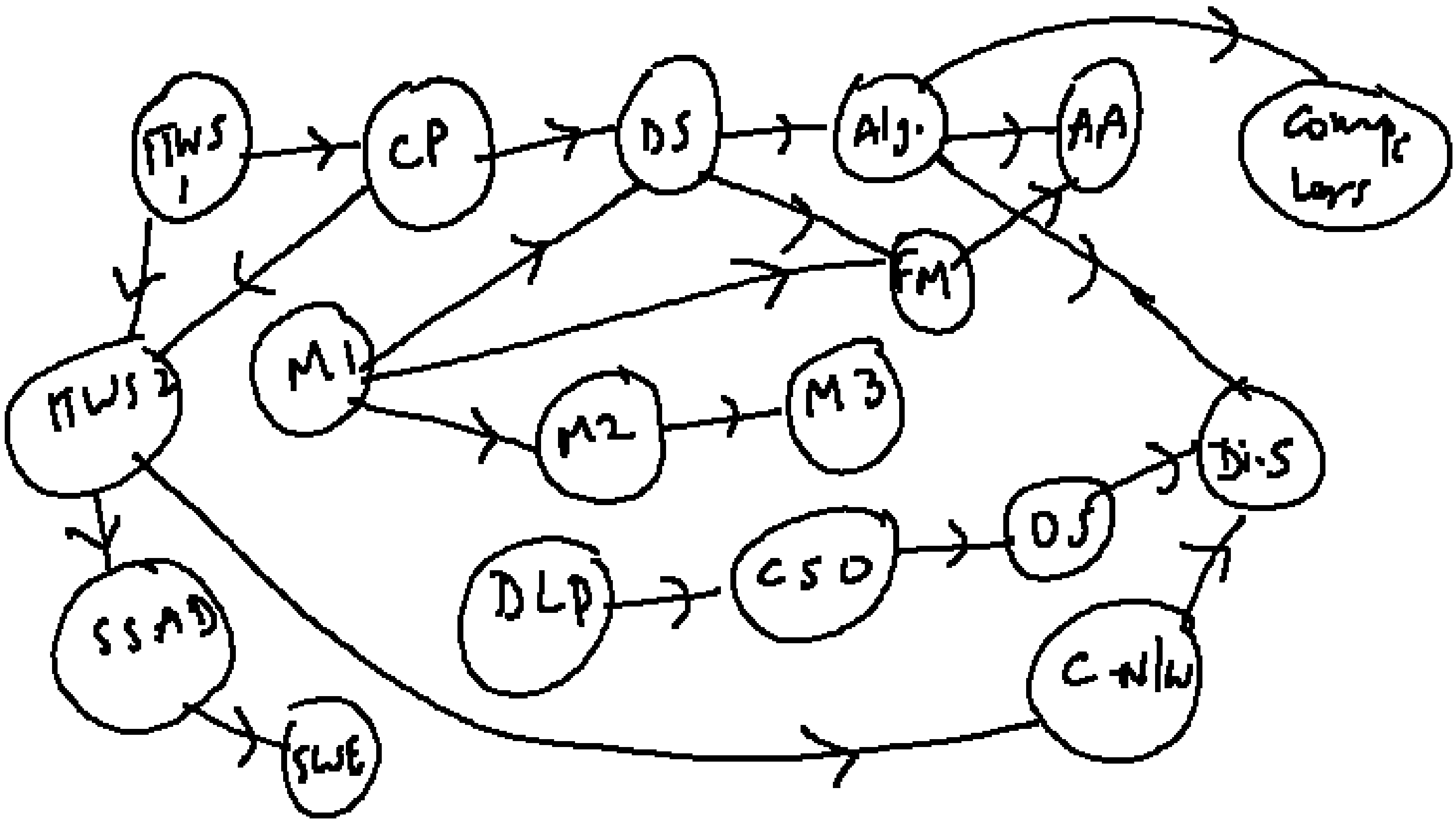
Cross Edges



Application of DFS - I

- Consider the UG curriculum here.
- Some courses have pre-requisites.
- A small picture illustrates this idea.

Example


ITWS 1
CP
DS
Alg.
AA
Compilers
ITWS 2
M1
FM
M2
M3
Di.S
OS
CSO
DLP
C-N/W
SSAD
SWE

Some Questions

- How long do you really need to complete the program if you are allowed to do as many courses as possible each semester?

Some Questions

- How long do you really need to complete the program if you are allowed to do as many courses as possible each semester?
- How soon can you take some course, by finishing all its prerequisites.

Some Questions

- How long do you really need to complete the program if you are allowed to do as many courses as possible each semester?
- How soon can you take some course, by finishing all its prerequisites.
- What is an order of the courses?

The Graph Based Solution

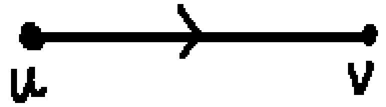
- The last question indicates some ordering on the vertices of the graph.
- The graph we have in this case is a directed graph.
- Additionally, there cannot be cycles in the graph.
- Such a graph is called as a directed acyclic graph, DAG for short.

The Graph Based Solution

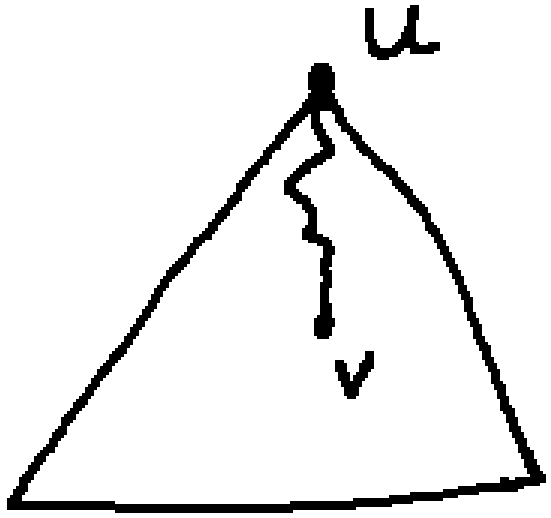
- We will use just DFS to arrive at our solution as follows.
- Consider performing DFS on the graph G.
- Let (u,v) be an edge in G.
- If G is a DAG, then it holds that $f(u) > f(v)$.

f(u) > f(v)

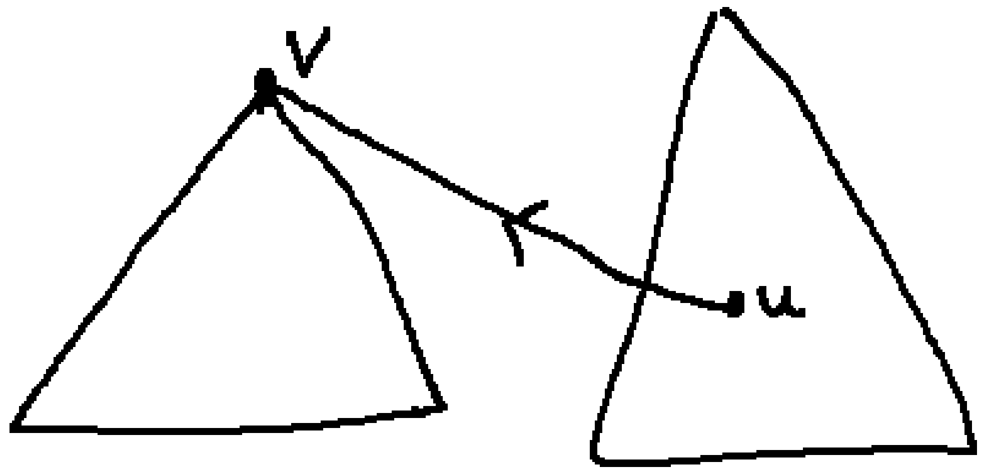
- If G is a DAG, then it holds that f(u) > f(v).

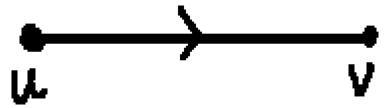


- Can be proved as follows. Let d(u) < d(v). Then the case is clear. The DFS procedure from u would definitely reach v, finishes at v, then returns to u.
- In this case, (u,v) is either a tree edge or a forward edge.



f(u) > f(v)



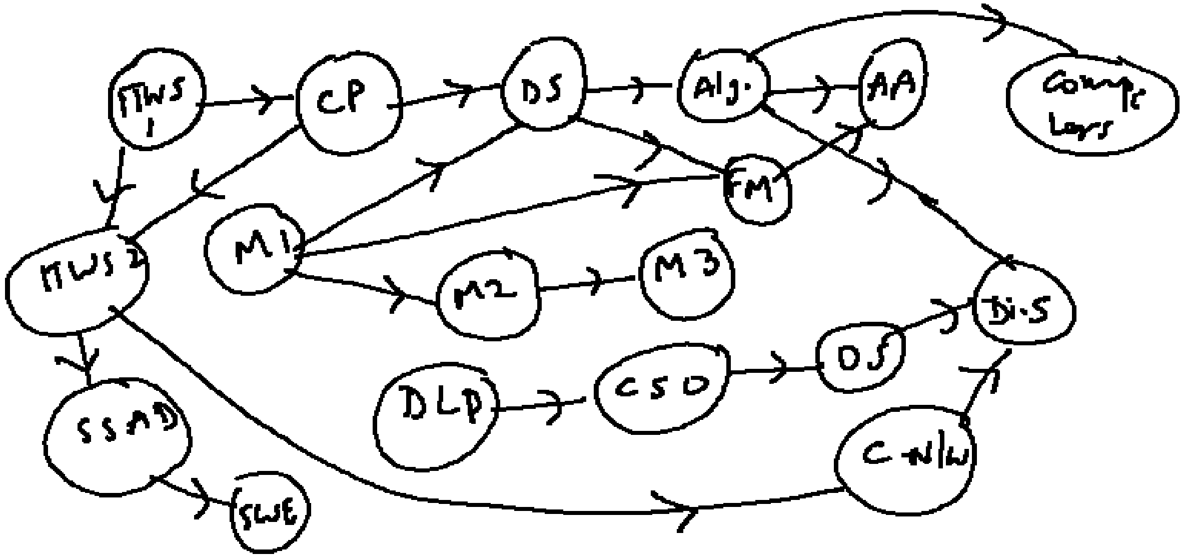


- On the other hand, if d(v) < d(u), then the DFS procedure at v cannot visit u. Why? So, in this case, also f(u) > f(v).
- The edge (u,v) appears as a cross edge.

The Solution

- We need an ordering of vertices such that:
  - If (u,v) is an edge, then u appears before v in the order.
- The simple solution to produce such an ordering of the vertices is to therefore perform a DFS and produces vertices in the decreasing order of their finish times.

Example



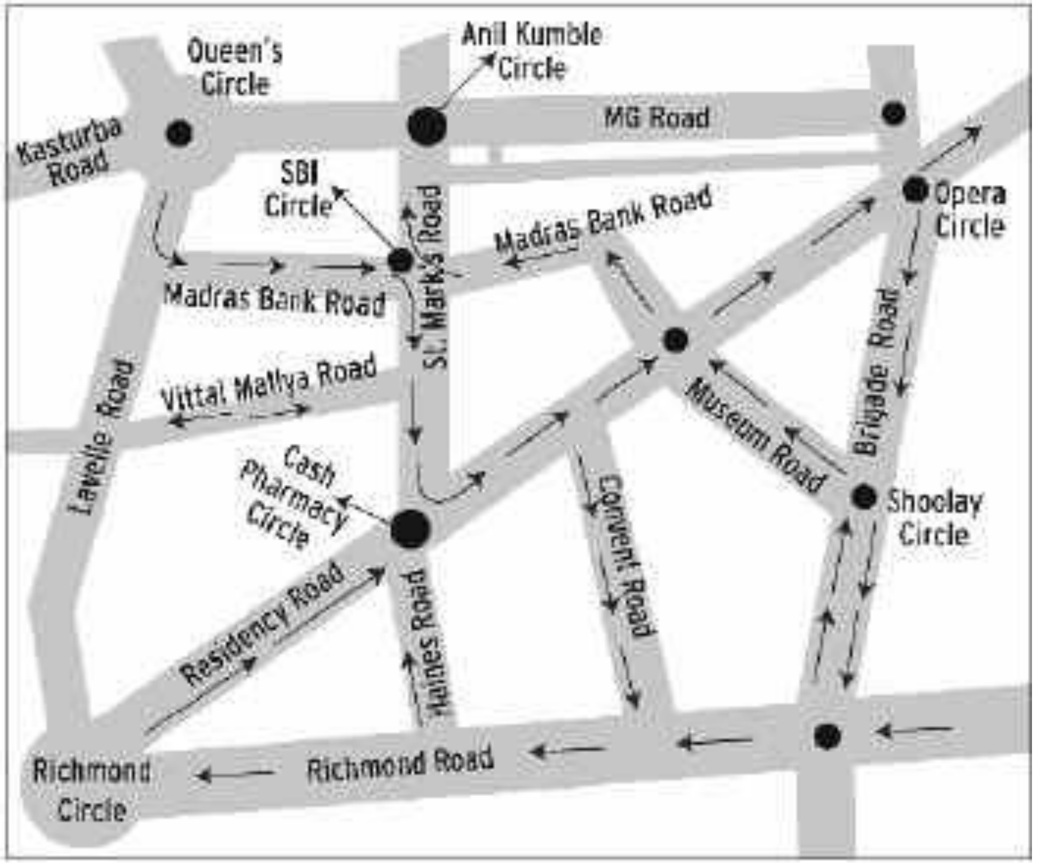
Sorted order

DLP, CSO, ITW1, CP, OS, ITW2, CN. SSAD, SWE, M1, M2, M3, DS, Alg, Comp., DiSy, FM, AAlg

Vertex	F(v)
DS	12
ITW1	32
ITW2	27
SSAD	24
CP	31
SWE	2
Alg	11
FM	5
Aalg	4
CSO	35
DLP	36
OS	30
CN	26
DiS.	8
Compilers	10
M1	18
M2	17
M3	16

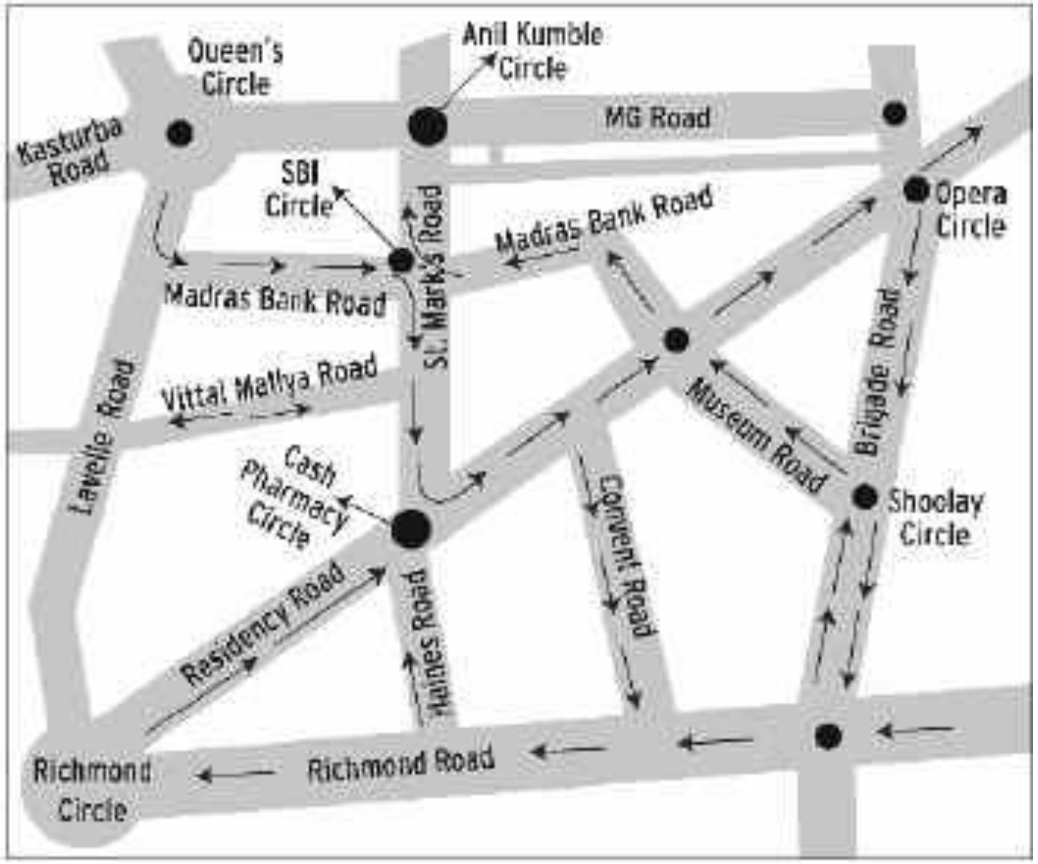
Application of DFS – II

- Suppose one day all the roads in the city are made directional.
  - like one-way roads.
- Can we go from one point to another point, and also come back respecting the directions?



Application of DFS – II

- Suppose one day all the roads in the city are made directional.
  - like one-way roads.



Map of Bangalore Circa 2005, From The Hindu

Applications of DFS – II

- The general problem is as follows.
- Given a directed graph G = (V, E) and two vertices U and v, is there a path between u to v and vice-versa.
- Does the above hold for every pair of vertices u, v?

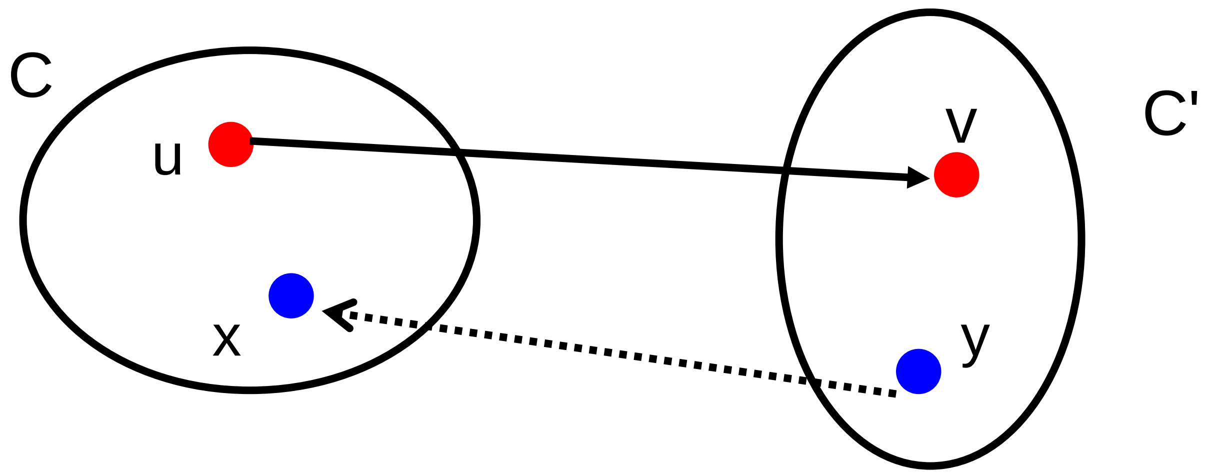
Applications of DFS – II

- The second question is more general than the first. Can be solved also.
- Problem: Given a directed graph G, partition V into maximal subsets so that each pair of vertices in each subset have a directed path between them.
  - u, v in V_i, u is reachable form v and v is reachable from u.
- Each such maximal set is called a strongly connected component.
- The partition is called as the Strongly Connected Components (SCC) of G.

Applications of DFS – II

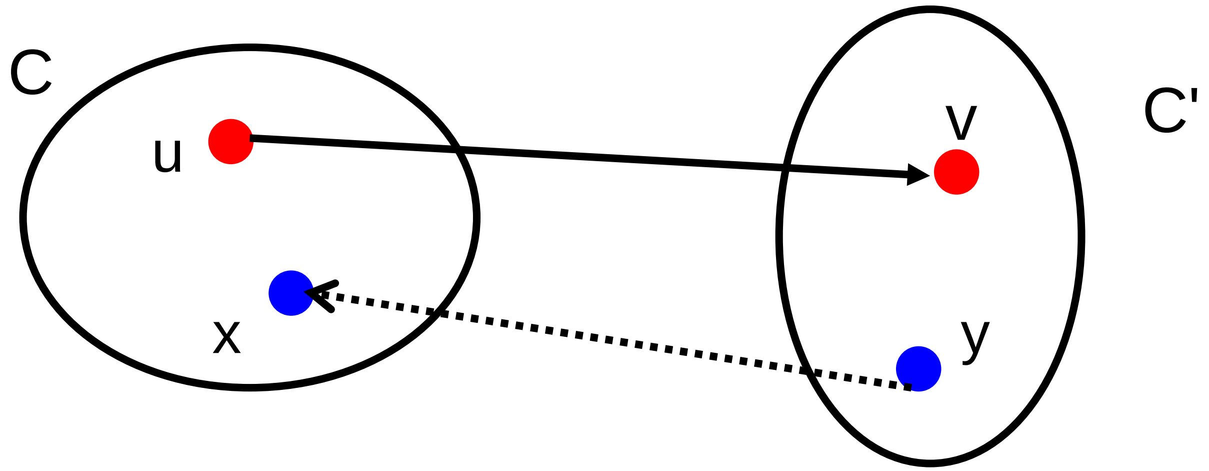
- Can use DFS to find the strongly connected components of a given directed graph.
- Requires a few thought questions.

Applications of DFS – II



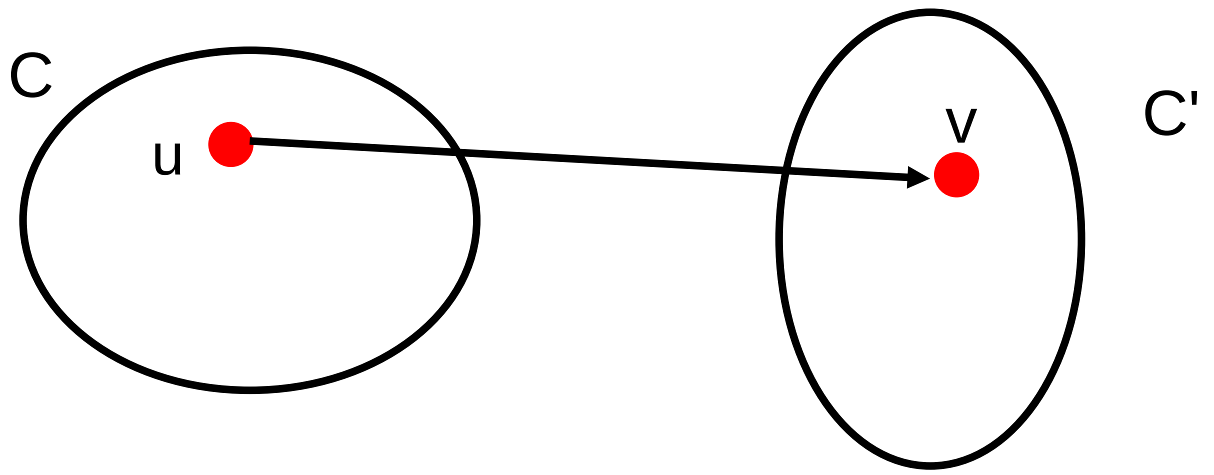
- Let C, C' be two (distinct) strongly connected components of G.
- Let u in C, and v in C' be such that (u,v) in an edge.
- There cannot be an edge from some y in C' to an x in C.

Applications of DFS – II



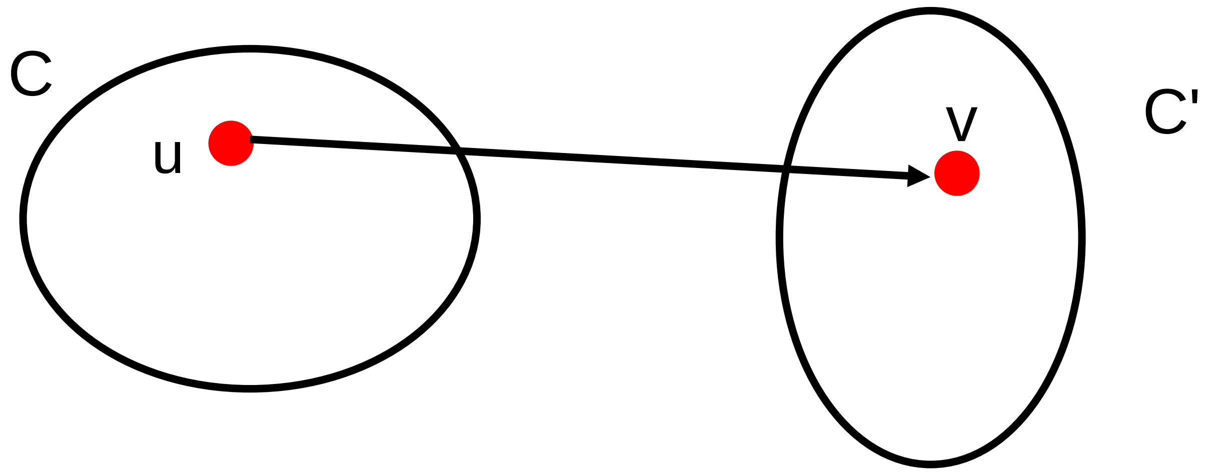
- If such an edge (y,x) exists, then C and C' are not maximal.

Applications of DFS – II



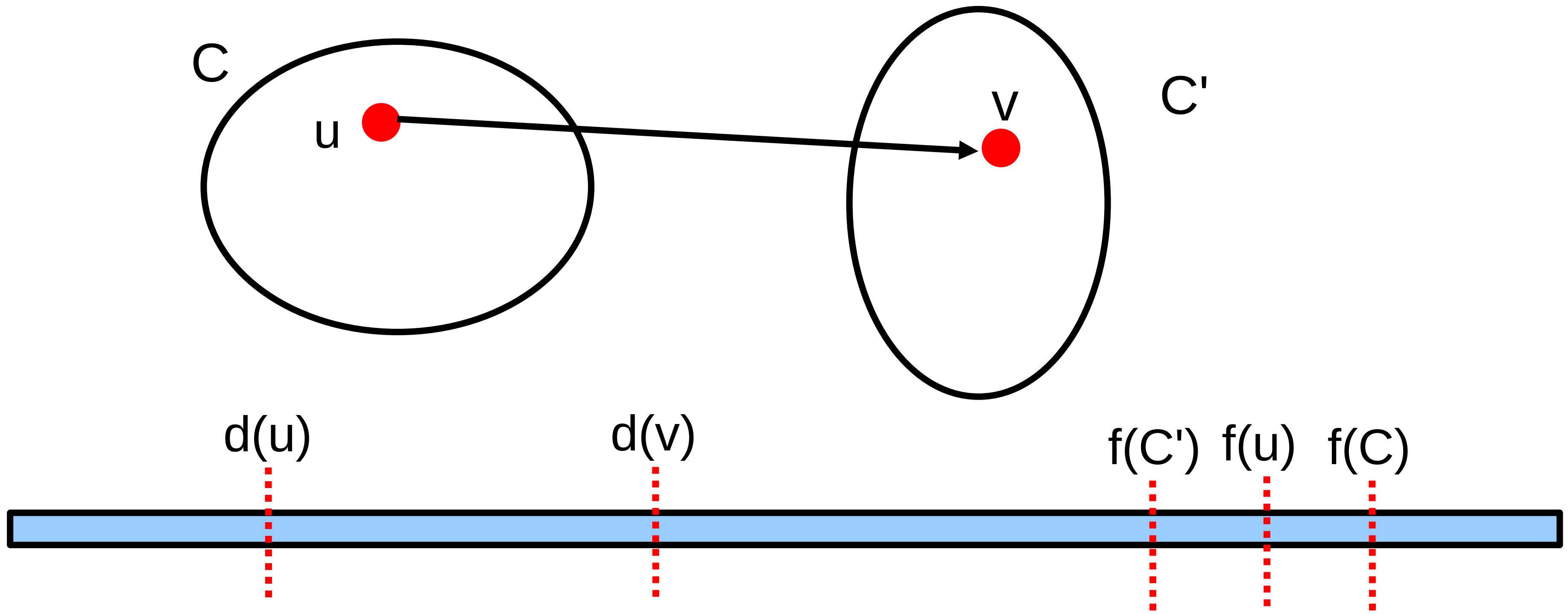
- For an SCC C, set $f(C) = \max_{v \text{ in } C} f(v)$.
- Given such C and C' and u, v with u in C, and v in C’ and (u,v) is an edge, in any DFS of G, C finishes after C'.

Applications of DFS – II



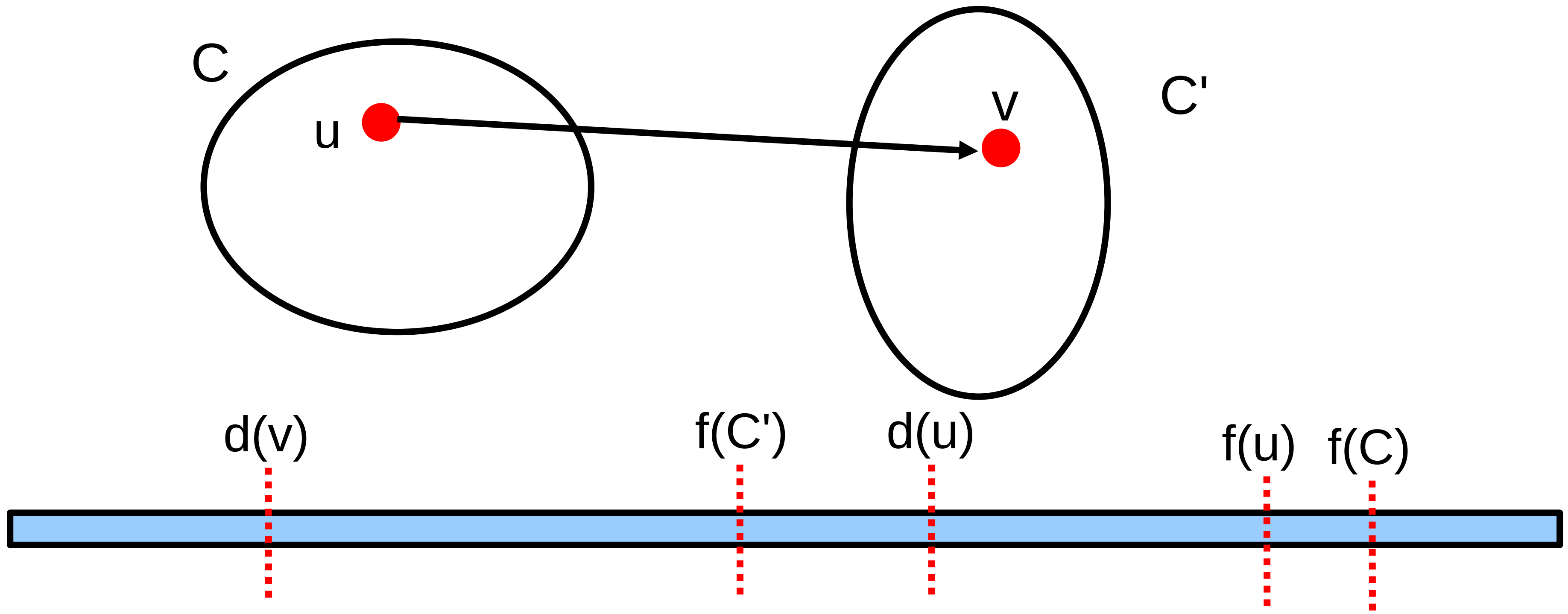
- For an SCC C, set f(C) = $\max_{v \text{ in } C}$ f(v).
- Given such C and C' and u, v with u in C, and v in C’ and (u,v) is an edge, in any DFS of G, C finishes after C'.
- Proof: Consider the case that d(u) < d(v).
- Then, DFS from u clearly enters C' via (u,v) or some other edge.

Applications of DFS – II



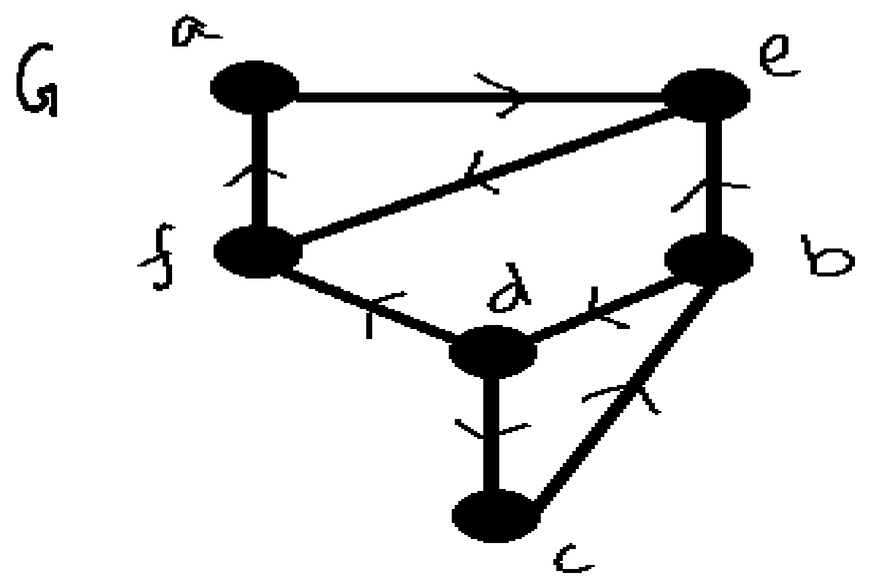
- Thus, v would be a descendant of u.
- Once DFS enters C', all vertices of C' would be visited before backtracking to a vertex in C.
- Thus, C' finishes before C.

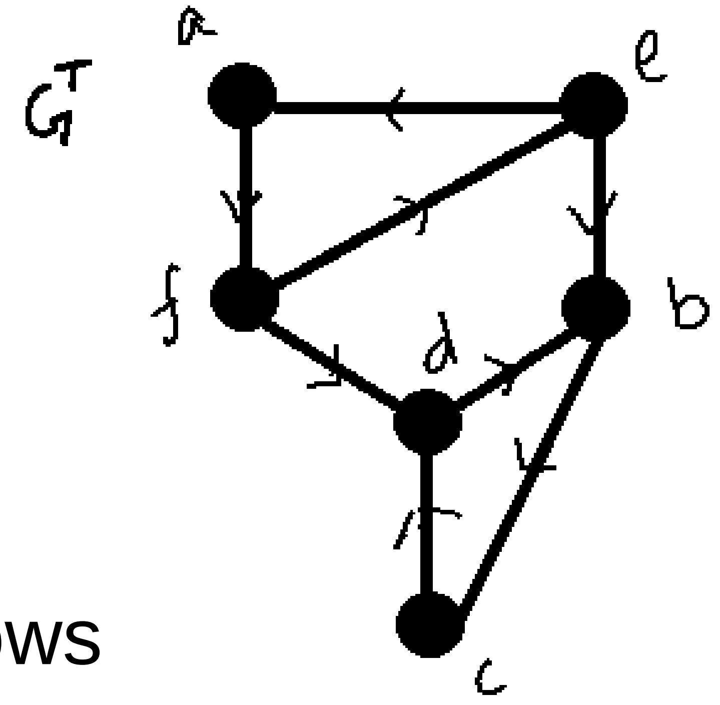
Applications of DFS – II



- Similar observations hold also if d(u) > d(v).
- In this case, DFS from v has to finish C' and cannot enter C.
- Observation 1: f(C) is also larger than f(C')

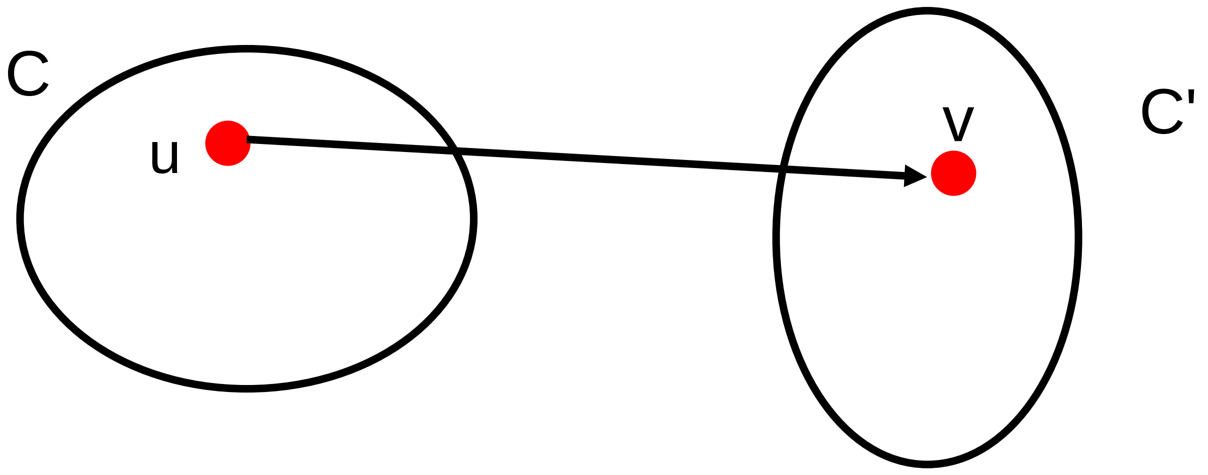
Applications of DFS – II





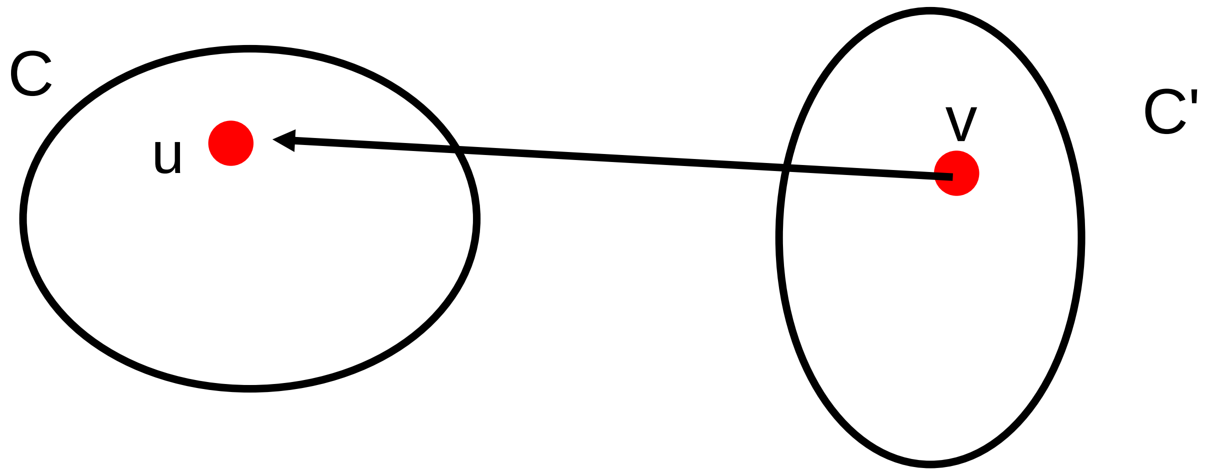
- Define the graph G^T as follows
  - $V(G^T) = V(G)$
  - $E(G^T) = \{ (v,w) \mid (w,v) \text{ in } E(G)\}$
- In essence, invert the directions of edges in G to get G^T.
- Observation 2: The SCCs of G and G^T are identical.

Applications of DFS – II



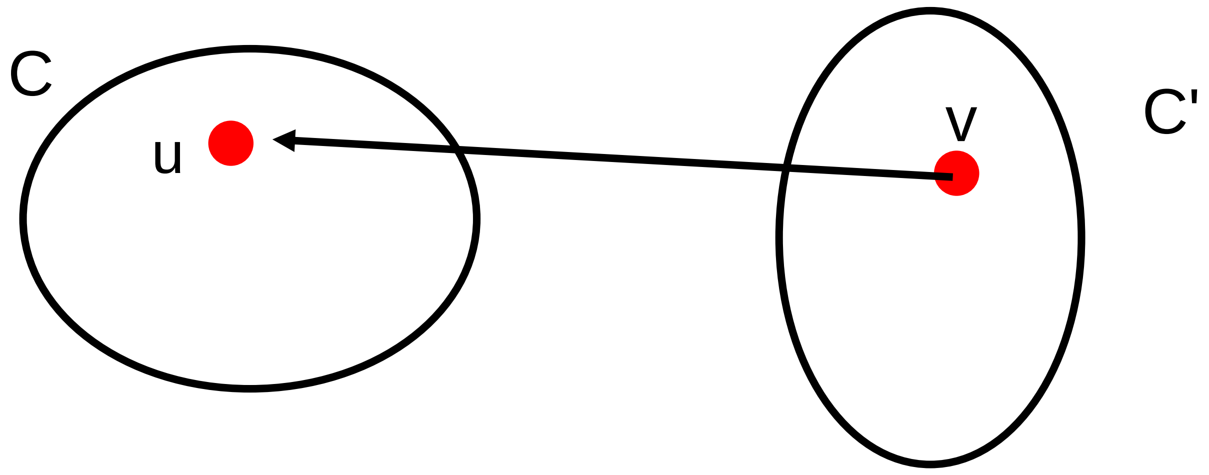
- Why are the observations important?
- Consider setting up a DFS in G^T so that
  - the DFS will visit only vertices in each component.
- How to set this up?

Applications of DFS – II



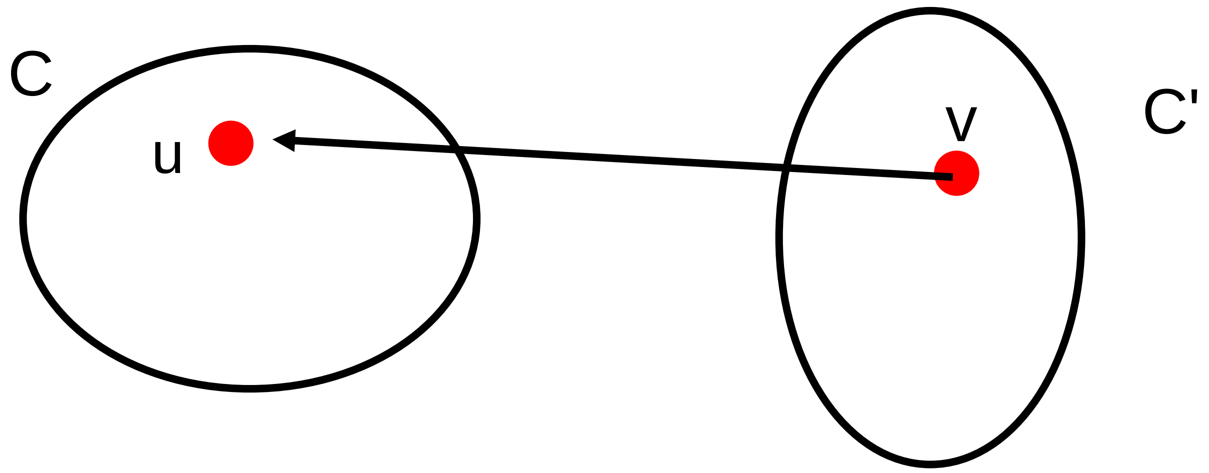
- Notice that in G^T, the edge (u,v) in G appears as (v,u).
- Also, a DFS in G^T that starts in C cannot enter C'.
- Why?

Applications of DFS – II



- How to do DFS in G^T such that it visits only vertices in one component before having to start again.
- If in a DFS in G, C finishes later than C', then there is some vertex in C whose finish time is more than the finish time of all vertices in C'.
- So, can start with such a vertex.

Applications of DFS – II

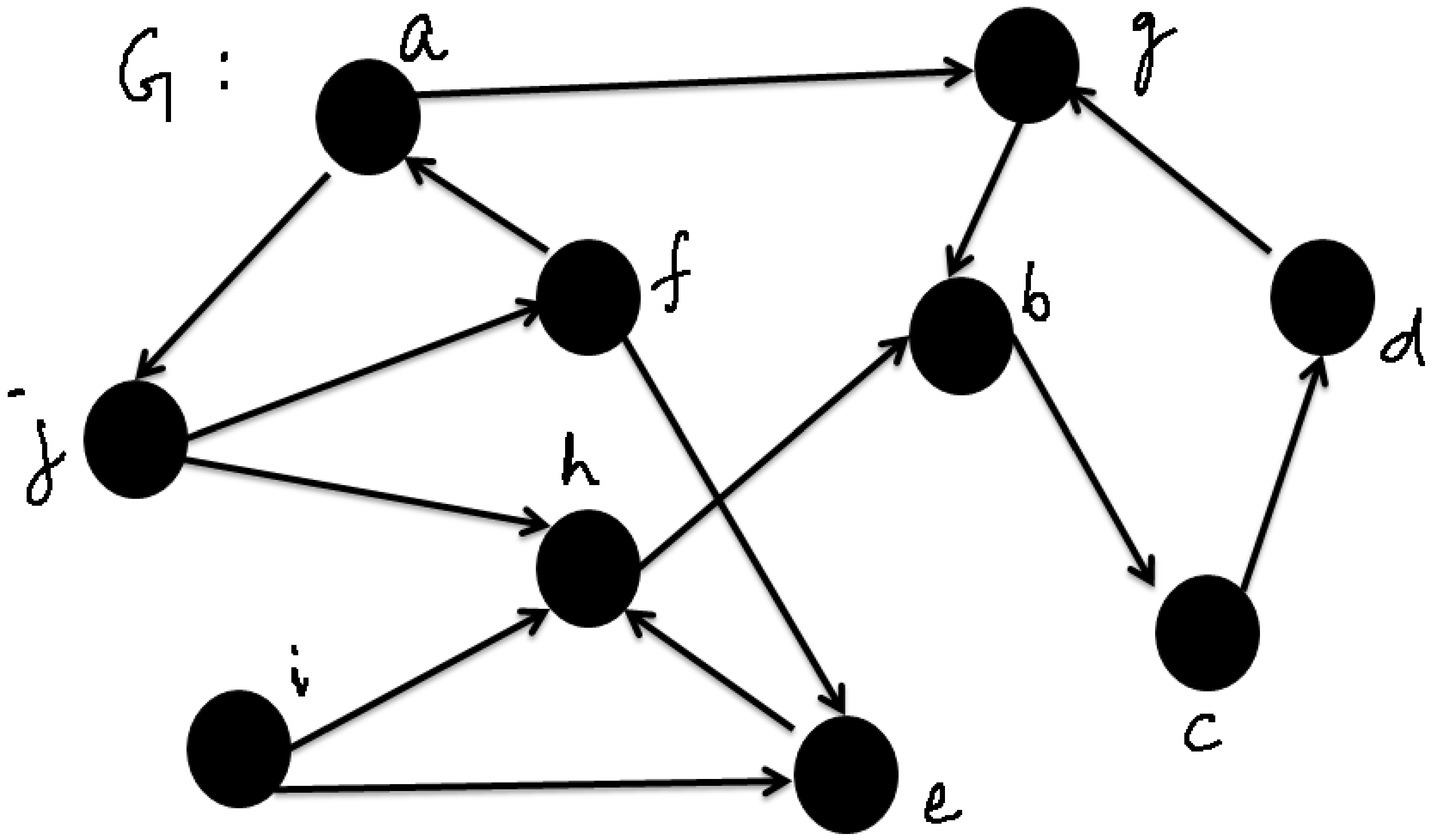


- Suggests that we should pick the start vertex for DFS in G^T such that it has the largest finish time among all vertices according to DFS in G.
- Indeed, that is the algorithm for finding SCCs also.

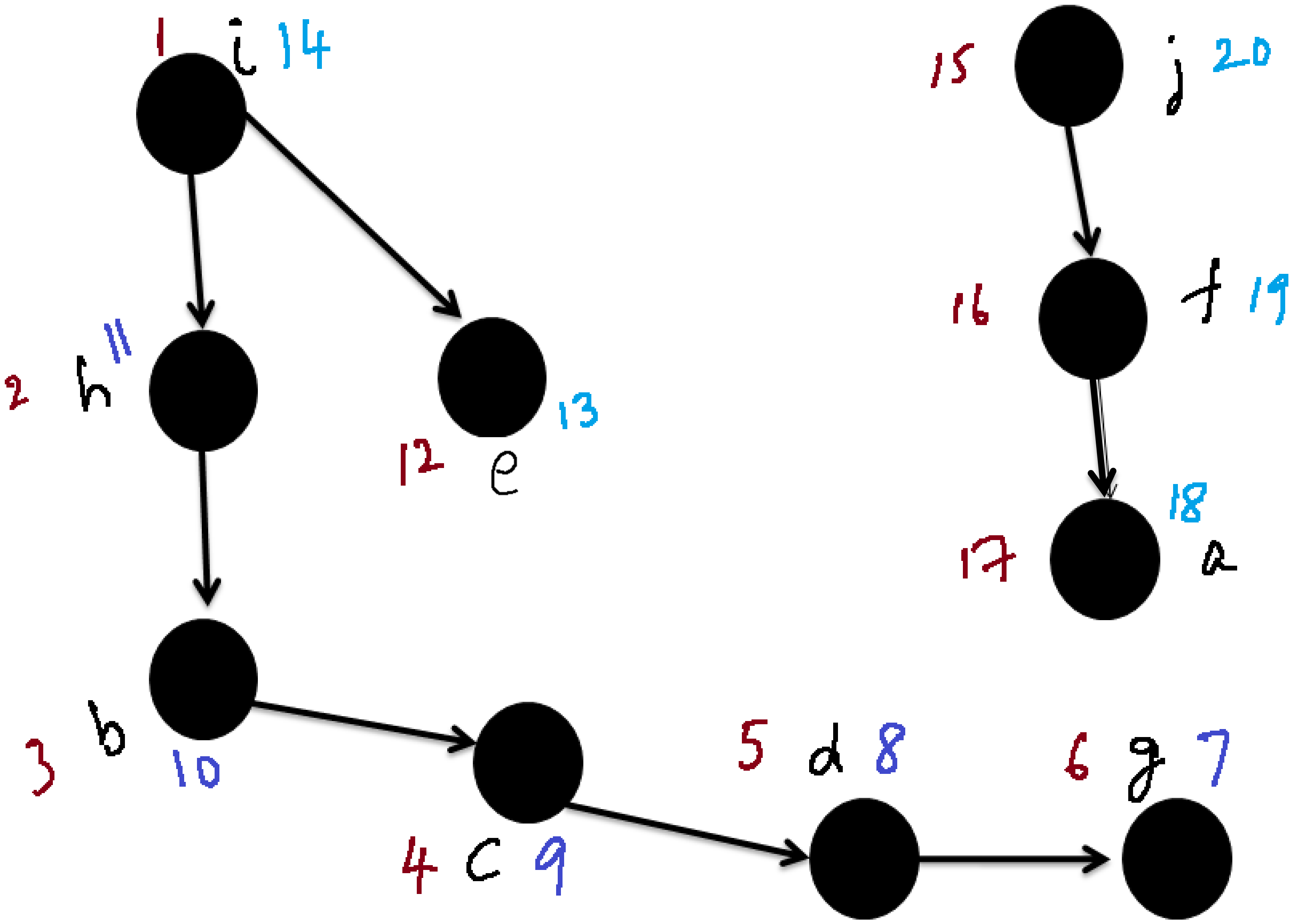
Algorithm for SCC

```
Algorithm SCC(G)
begin
    – perform a DFS in G and record the d() and f() times of all vertices.
    – construct the graph $G^T$ from G.
    – Perform DFS in $G^T$ picking vertices in decreasing order of finish time according to DFS in G
end
```

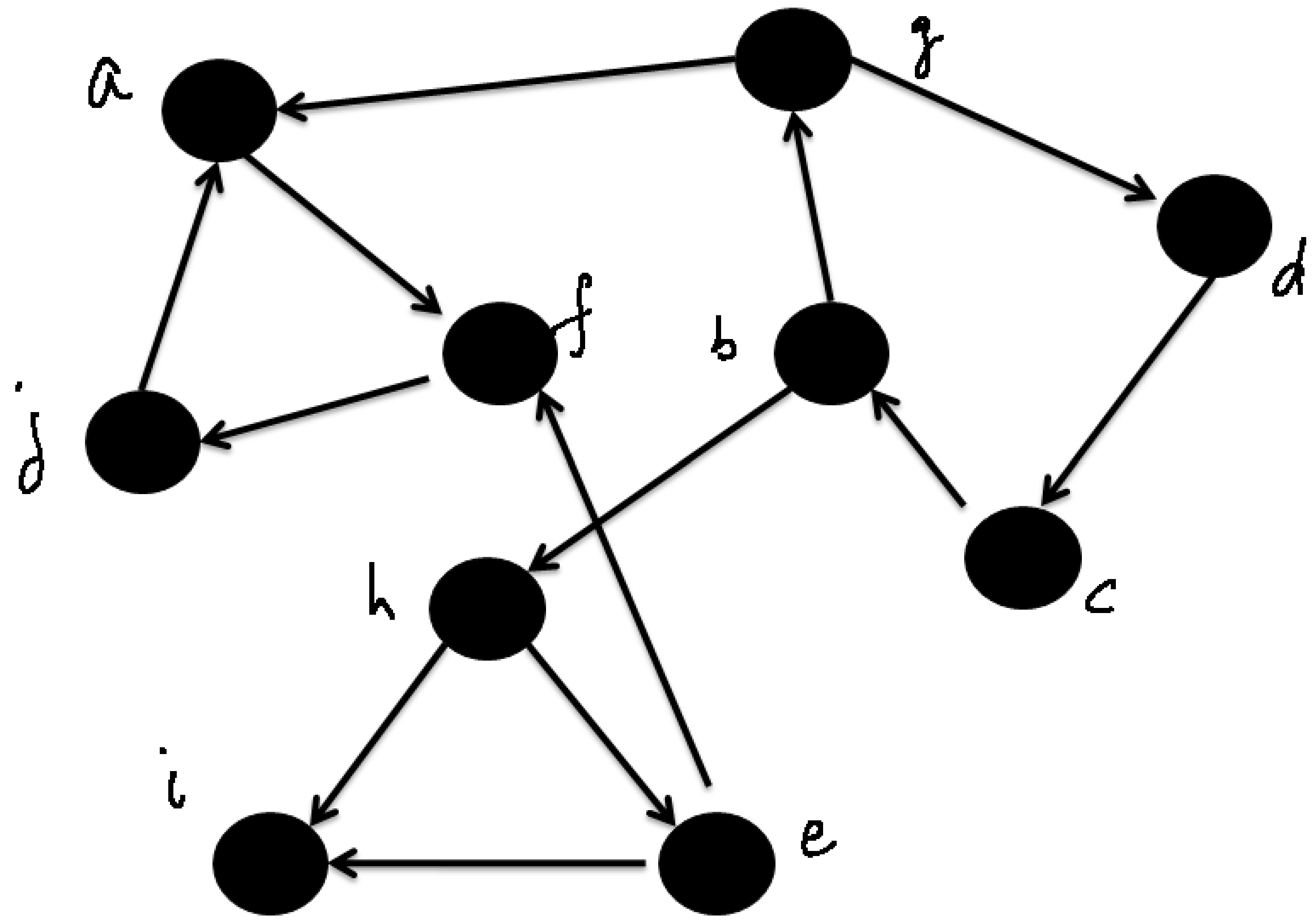
Practice Problem



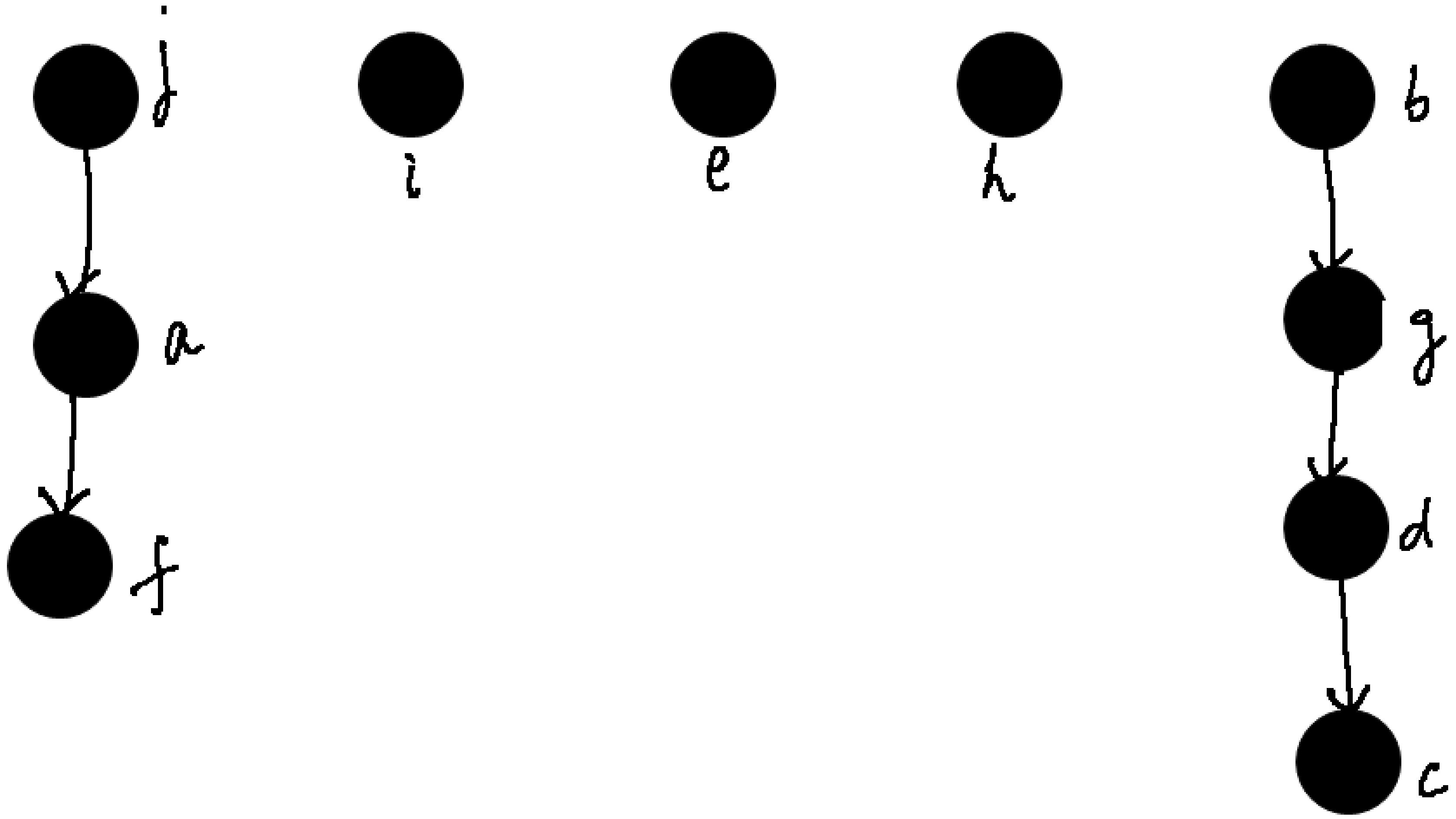
Example SCC: DFS on G



The Graph G^T

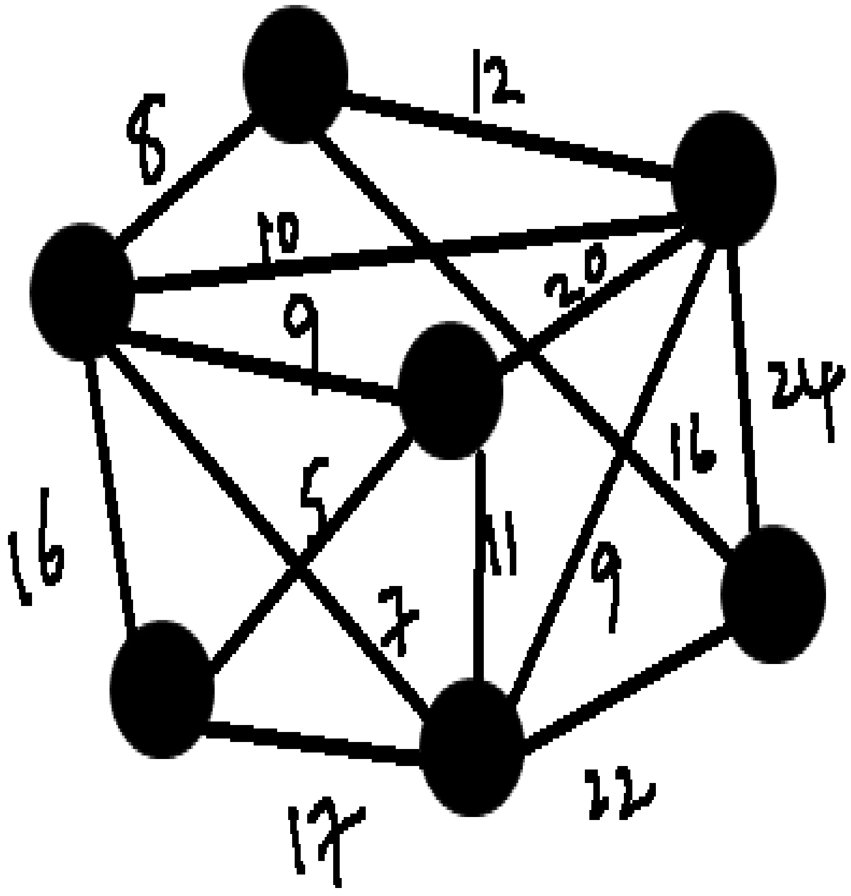


DFS on G^T With Specific Start Vertices



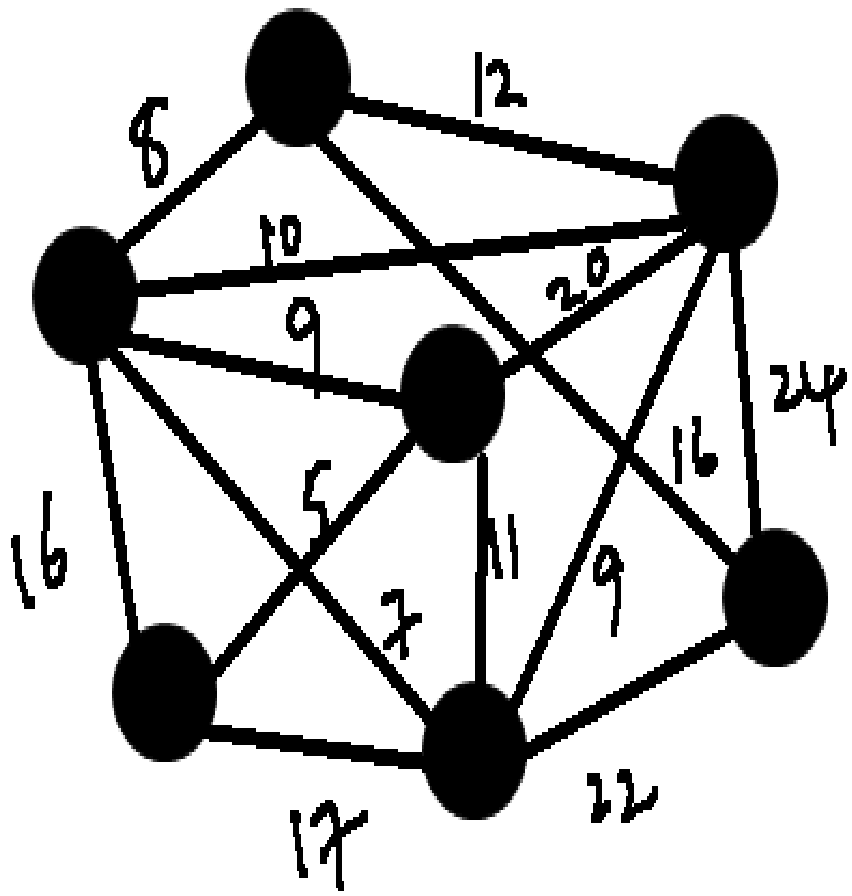
Spanning Trees

- We will now consider another famous problem in graphs.
- Imagine providing connectivity to a set of cities.
- Each highway connects two cities
- In reality, each highway requires a certain cost to be built.



Spanning Trees

- So, there is a trade-off here.
- How to provide connectivity while minimizing the total cost of building the highways.
- The weights on the edges indicate the cost of building that highway.
- The total cost of connectivity = sum of all the built up highway
- Minimize this cost.



Spanning Trees

- Formalize the problem as follows.
- Let G = (V, E, W) be a weighted graph.
- Find a subgraph G' of G that is connected and has the smallest cost
  - Cost is defined as the sum of the edge weights of edges in G'.

Spanning Trees

- Observation I : If G' has a cycle and is connected, then there exists a G'', which is also a subgraph of G and is connected so that

  cost(G'') < cost(G')

- To get G'', simply break at least one cycle of G'.

- Hence, the optimal G' shall have no cycles and is connected.
  - Suggests that G' is a tree.

Spanning Trees

- Two keywords : spanning and tree.
- Some notation: A subgraph G' of G is called a spanning subgraph if V(G') = V(G).
- A spanning subgraph G' of G that is also a tree is called as a spanning tree of G.

Spanning Trees

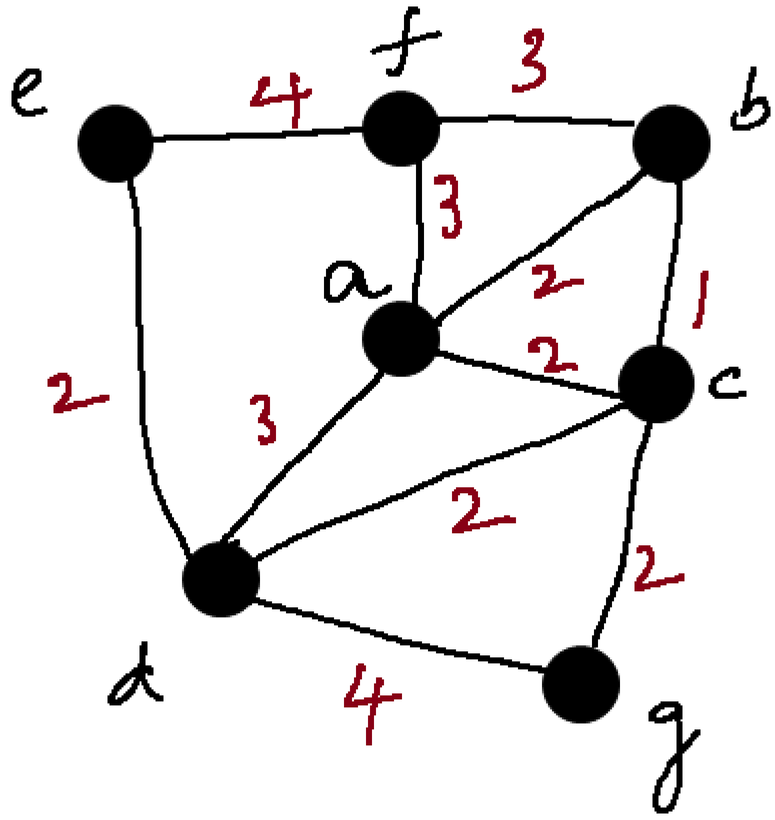
- Consider the problem: Find a spanning tree of G that has the least cost.
- Such a spanning tree is also called as a minimum cost spanning tree of G. Often one refers to this as the minimum spanning tree, or MST for short.

MST Algorithm

```
Algorithm MST(G)
begin
    sort the edges of G in increasing order of   weight as
    $e_1, e_2, ..., e_m$
     k = 1; V(T) = V(G); E(T) = $\Phi$
     while |E(T)| < n-1 do
          if **$E(T)$ U $e_k$ does not have a cycle** then
               E(T) = E(T) U $e_k$
          end-if
          k = k + 1;
    end-while
End.
```

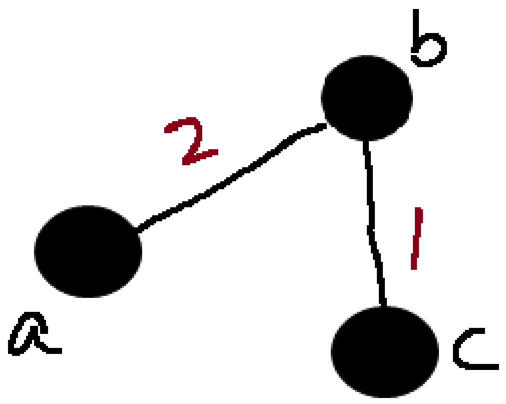
MST Practice Problem

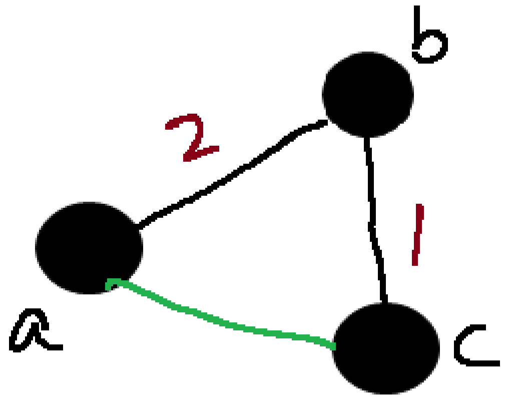
```
Algorithm MST(G)
begin
        sort the edges of G in
        increasing order of   weight
        as $e_1, e_2, ..., e_m$
         k = 1; V(T) = V(G); E(T) = $\Phi$
          while |E(T)| < n-1 do
              if **E(T) U $e_k$ does not
               have a cycle** then
                  E(T) = E(T) U $e_k$
              end-if
              k = k + 1;
        end-while
End.
```

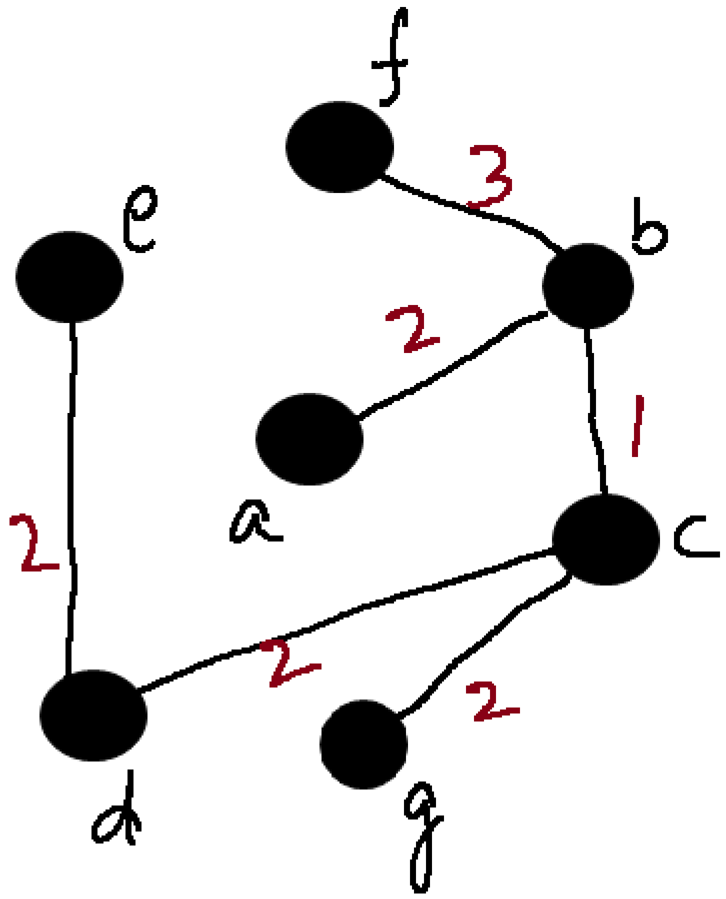


MST Example

- List of edges by weight
  - bc, ab, ac, cg, cd, de, bf, af, ad, ef, dg







MST

- Let us now think of devising an algorithm to construct an MST of a given weighted graph G.
- There are several approaches, but let us consider a bottom-up approach.
- Let us start with a graph (tree) that has no edges and add edges successively.
- Every new edge we add should not create a cycle.
- Further, the total cost of the final tree should be the least possible.

MST

- Suggests that we should prefer edges of smaller weight.

MST

- Suggests that we should prefer edges of smaller weight.
  - But should not add edges that create cycles.

MST

- Suggests that we should prefer edges of smaller weight.
  - But should not add edges that create cycles.
- Indeed, that is intuitive and turns out that is correct too.
  - we will skip the proof of this.

MST Example

- List of edges by weight
  - bc, ab, ac, cg, cd, de, bf, af, ad, ef, dg

MST Algorithm Analysis

- The algorithm we devised is called the Kruskal's algorithm.
- Belongs to a class of algorithms called greedy algorithms.
- How do we analyze our algorithm?
  - Need to know how to implement the cycle checker.

MST Algorithm Analysis

- How quickly can we find if a given graph has a cycle?
  - O(m+n) is possible using DFS.
- Notice that if the graph is a forest, then m = O(n).
- So, can be done in O(n) time.
- Also, need to try all m edges in the worst case.
- So the time required in this case is O(mn).

MST Algorithm Analysis

- Too high in general.
- But, advanced data structures exist to bring the time down very close to O(m+n).
  - Cannot be covered in this class.
  - We will show an approach that takes us almost there.

Advanced Data Structures

- An abstract problem:
- Given n elements, grouped into a collection of disjoint sets S_1, S_2, …, S_k, design a data structure to:
  - Find the set to which an element belongs
  - Combine two sets
- The abstract problem finds applications in several settings:
  - Spanning tree algorithm of Kruskal
  - Graph connected components
  - Least common ancestors
  - …

Some Notation

- Imagine a collection S = {S_1, S_2, …, S_k} of sets.
- Each set has a **representative** element
  - Some member of the set, typically.
  - Depending on application, can be
    - The smallest numbered element
    - A number
    - Or other
- Typical operations
  - MakeSet(x)
  - Union(x, y)
  - FindSet(x)

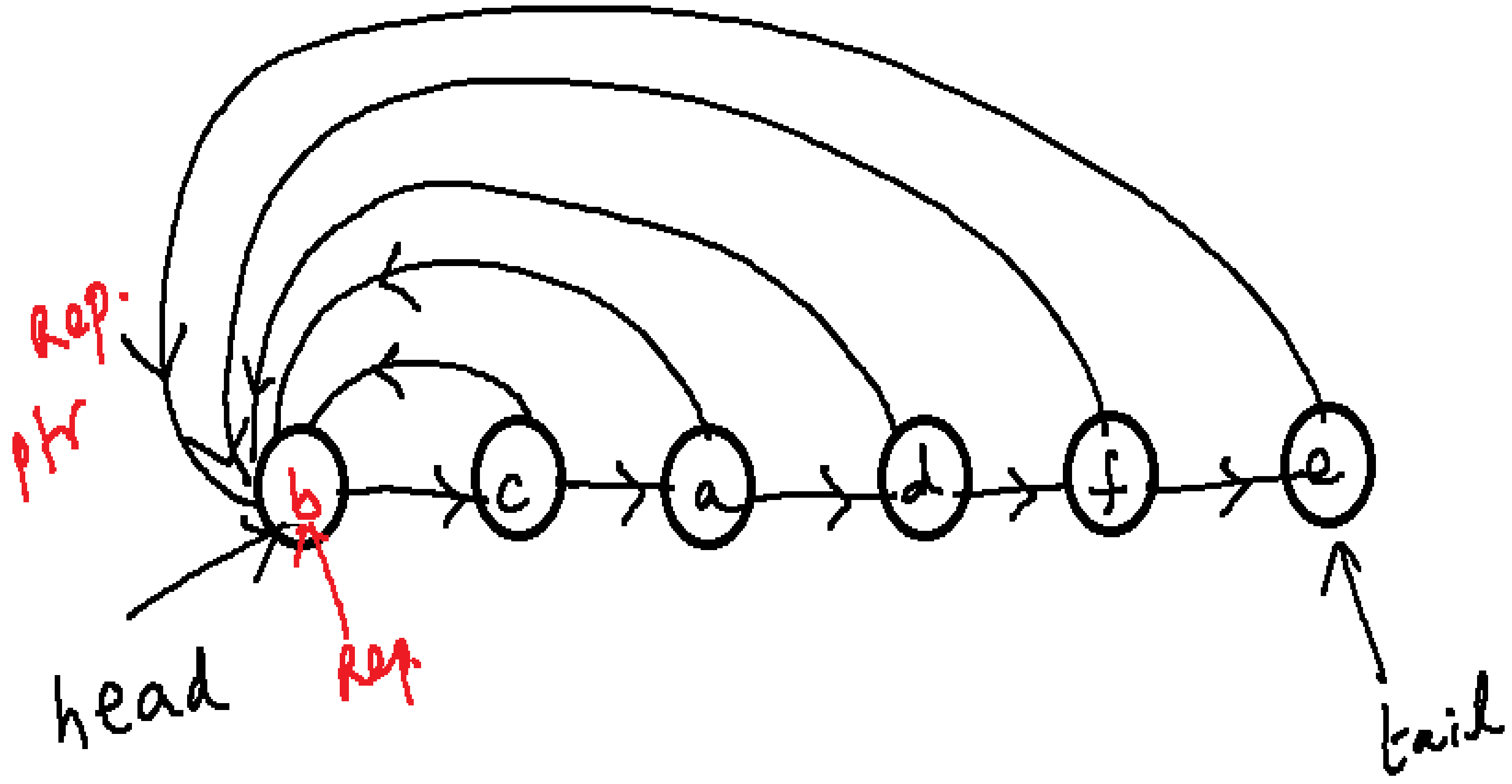
Some Notations

- Two parameters
  - n : The number of MakeSet operations.
  - m : The total number of MaketSet, Union, and Find operations.
- Some observations
  - Each Union operation reduces the number of sets by 1.
  - When starting with n elements, at most n-1 Union operations.
  - Also, m >= n.
- Assume that the n MakeSet operations are the first n operations.

How to Implement the Operations?

- Option 1 : Use linked lists.
- For every set, there is a linked list.
- The representative of a set is the head of the list.
- Every element also stores a pointer to the representative.
- There is a tail pointer indicating where to append.

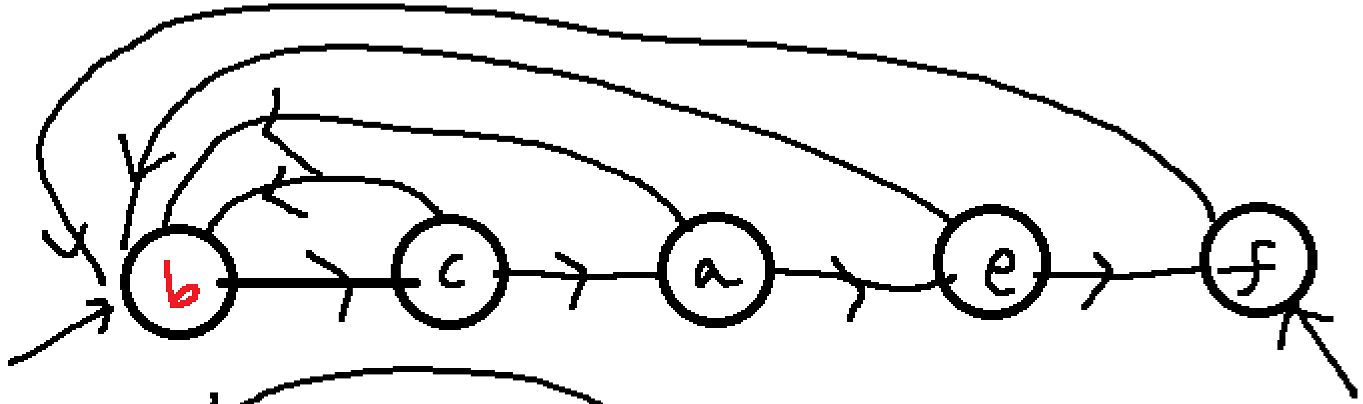
Example

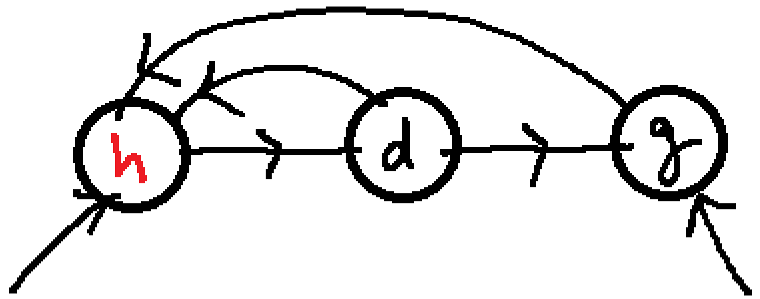


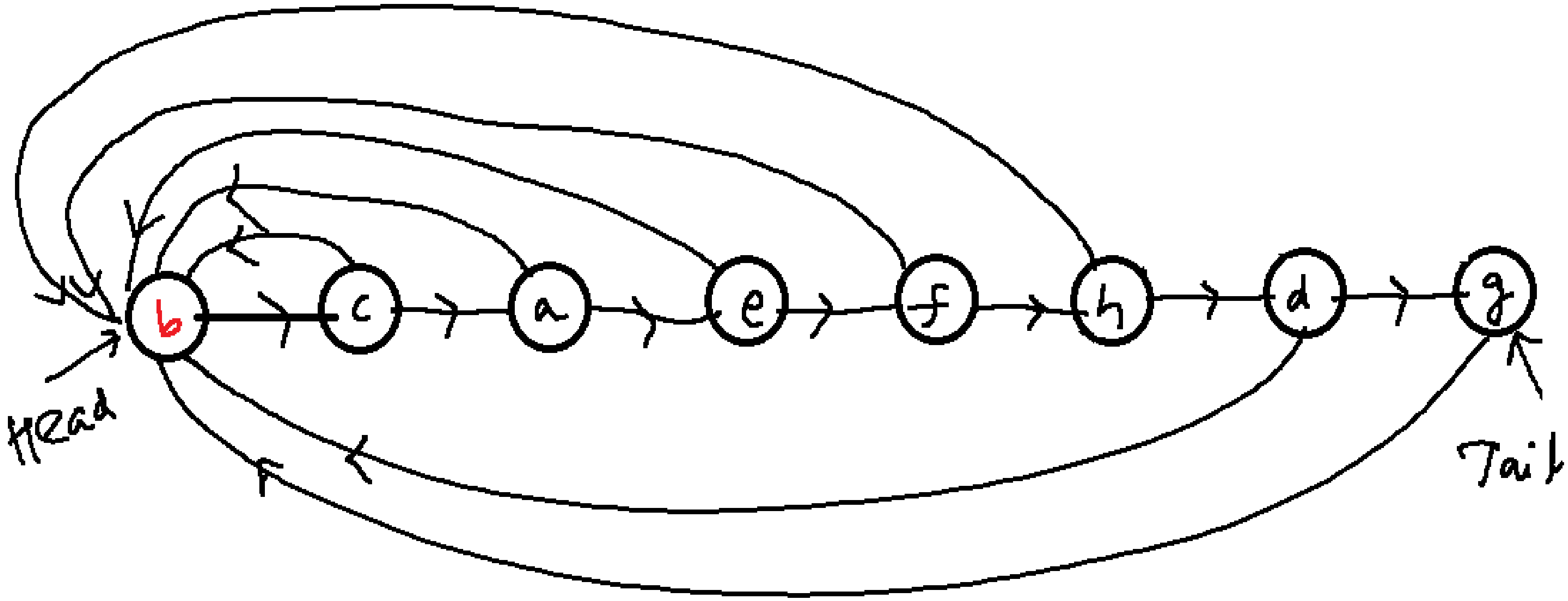
Operations

- MakeSet(x): Create a new linked list.
- FindSet(x) : Can be answered via the direct pointer

- Union(x, y) : Can append the list of x to the list of y.
- But have to update the pointer for each element in the list of x.


b
c
a
e
f

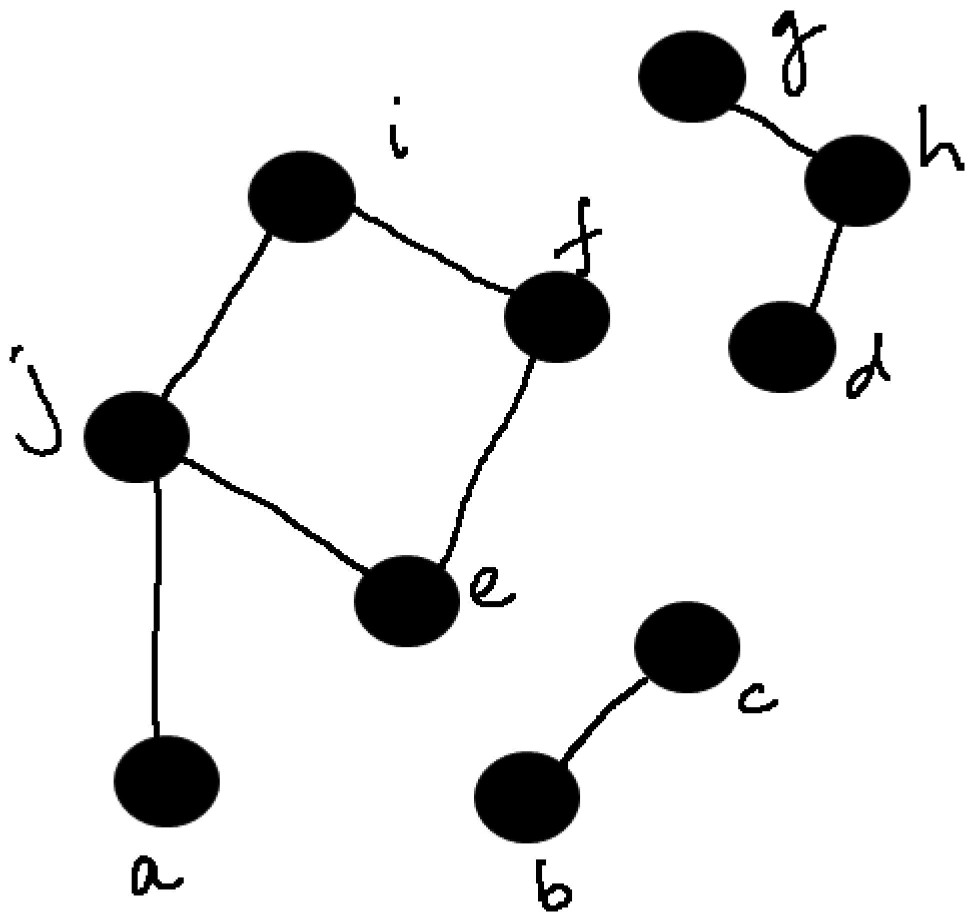

h
d
g


b
c
a
e
f
h
d
g
Head
Tail

Application to Connected Components

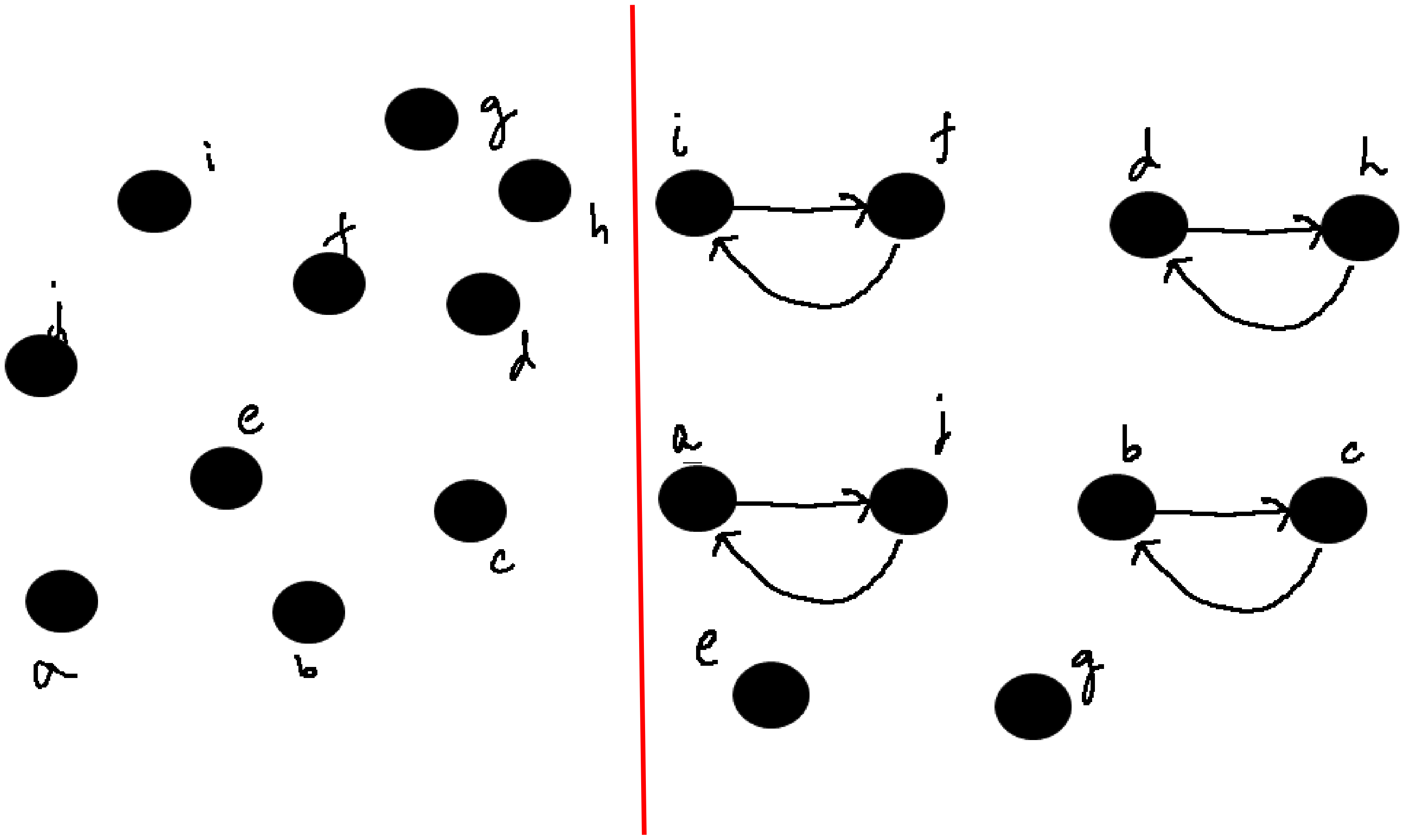
- Problem: Given an undirected graph G = (V, E), partition V into disjoint sets V_1, V_2, …, V_k, so that two vertices u and v are in the same partition if and only if there is a path between u and v.
- Several ways to solve this problem
  - This may not be the best way!
- Example follows.

Example

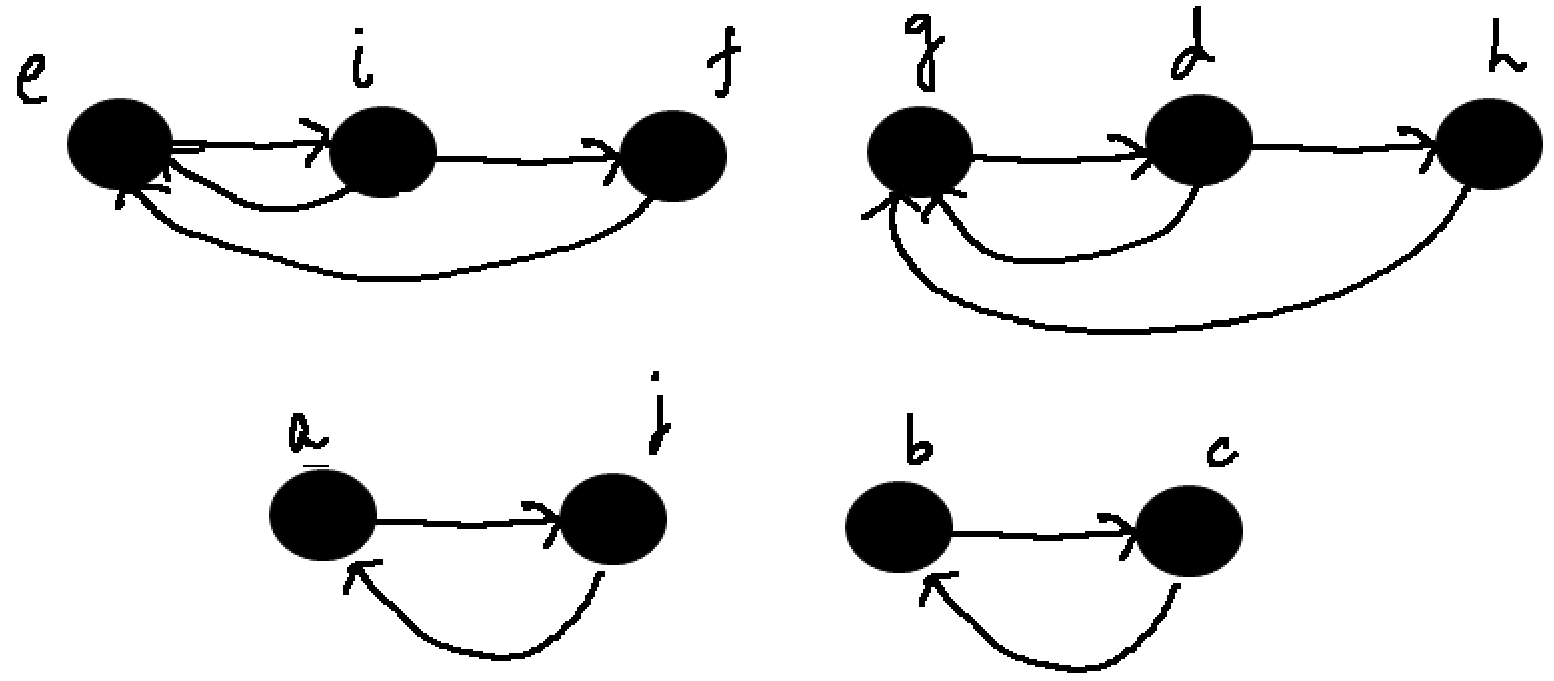


- Algorithm:
  - For each vertex v
    - MakeSet(v)
  - For each edge vw
    - Union(v,w)

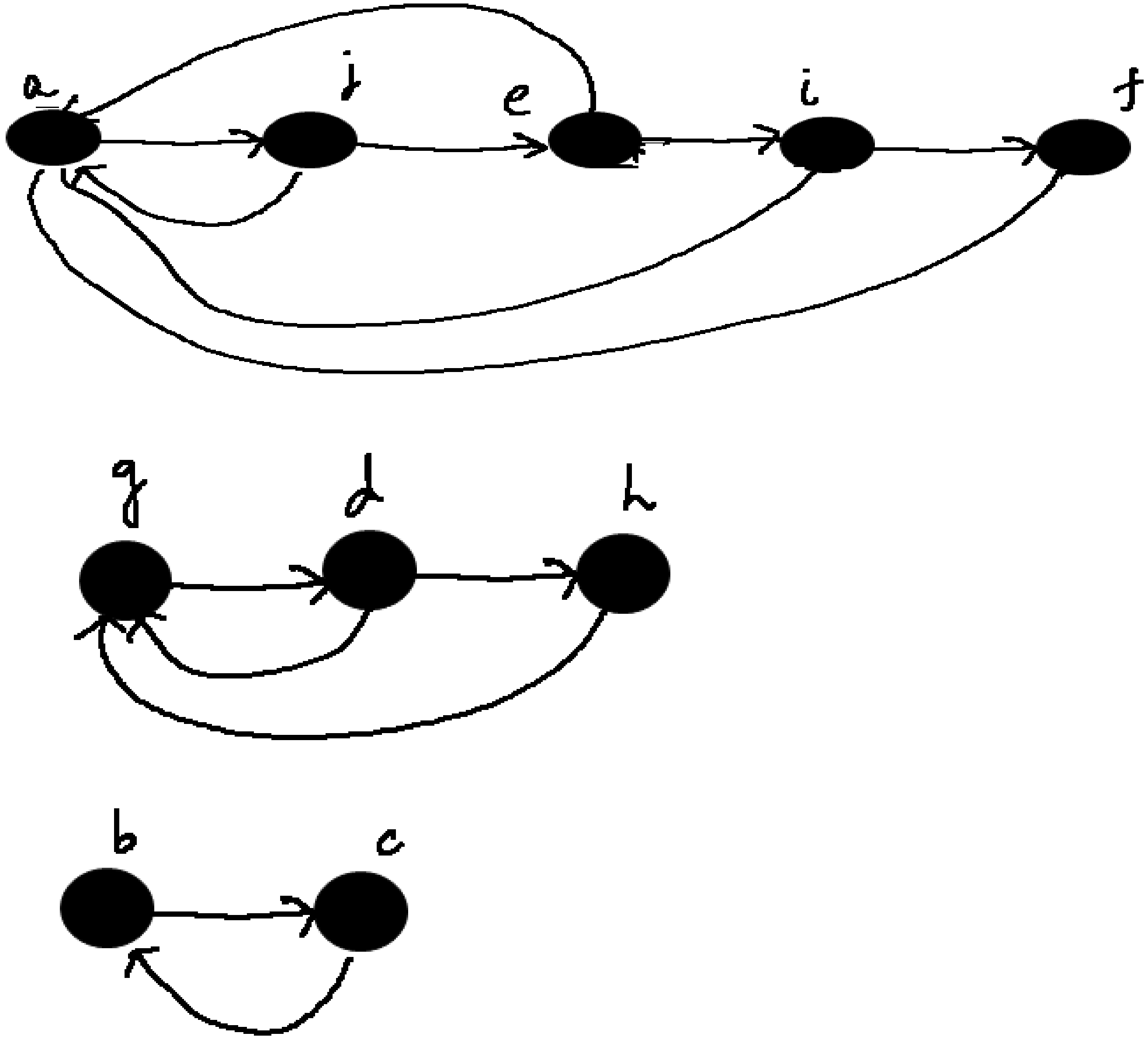
Example



Example



Example



Operations

- How difficult is it to append the lists?
- Claim: There exists a sequence of m operations on n objects so that the total time required for the entire sequence of operations is $O(n^2)$.

Operations

- How difficult is it to append the lists?
- Claim: There exists a sequence of m operations on n objects so that the total time required for the entire sequence of operations is $O(n^2)$.

- After the first n MakeSet operations, call Union(x1,x2), Union(x2, x3), Union(x3, x4), …, Union(x_{n-1}, x_n).
- The kth union call takes time proportional to k.
- Total time is therefore $O(n^2)$.

- The average time per operation is also O(n).

Application to Kruskal’s Algorithm

- An average time of O(n) is not helpful for Kruskal’s algorithm.
- We have several Union calls and several FindSet calls.

Better Solution

- Most of the time spent is in the Union operation.
- Can we modify the operation slightly?
- Intuitively, it is easier to append a smaller list to a larger list.
  - Requires fewer updates.
  - Will the overall time decrease?
- We will show that indeed it does.

The Weighted Union Heuristic

- Maintain the length of each list. Corresponds to the size of the set.
- To perform Union(x, y):
  - Append the list of x to the list of y if len(x) < len(y)
  - Append the list of y to the list of x otherwise.
- A single Union operation can still take lot of time.
  - Union of two large lists, say of size n/10 each.
- But, a sequence of operations may be not so expensive.
  - Hopefully.

Analysis

- How many times can an element change its representative?
- Consider any element x.
- If in an Union operation, the representative of x changes, then x is in the smaller list.
  - Why?

Analysis

- How many times can an element change its representative?
- Consider any element x.
- If in an Union operation, the representative of x changes, then x is in the smaller list.
  - Why?
- The first time this happens, the resulting list has at least 2 elements.

Analysis

- How many times can an element change its representative?
- Consider any element x.
- If in an Union operation, the representative of x changes, then x is in the smaller list.
  - Why?
- The first time this happens, the resulting list has at least 2 elements.
- Next time, the resulting list has at least 4 elements.

Analysis

- In general, if the representative of x changes k times, then the resulting list has size at least 2^k.
- The largest set can have a size of n.
- Therefore, the representative of x cannot change more than log n times, over all the Union operations.
- This applies to every element.
- Therefore, over all Union operations, the total time spent is O(n log n).

Analysis

- Now, consider a sequence of m operations.
- MakeSet and Find are O(1) time operations.
- Therefore, the total time is O(m + nlog n).
- The average time per operation is O(log n).

Application to Kruskal's Algorithm

- How does the above apply?

Application to Kruskal’s Algorithm

- Do n MakeSet operations indicating that each vertex is in its own tree/set.
- To check if e = uv creates a cycle, check if FindSet(u) = FindSet(v).
- If not, add e to the current tree. Perform Union(u, v) to merge the trees of u and v.

- There are at most m FindSet operations.
- Overall time is therefore bound by O(m+nlog n).

MST – Another Approach

- The previous approach has to check for cycles every iteration.
- Another approach that has a smaller runtime even with basic data structures.
- Largely simplifies the solution.

MST – Another Approach

- The previous approach is characterized by having a single tree T at any time.
- In each iteration, T is extended by adding one vertex v not in T and one edge from v to some vertex in T.
- Starting from a tree of one node, this process is repeated n-1 times.

MST – Another Approach

- Two questions:
  - How to pick the new vertex v?
  - How to pick the edge to be added from v to some other vertex in T?

MST – Another Approach

- The answers are provided by the following claims.

- Claim 1: Let G = (V, E, W) be a weighted undirected graph. Let v be any vertex in G. Let vw be the edge of smallest weight amongst all edges with one endpoint as v. Then vw is always contained in any MST of G.

MST – Another Approach

- Claim 1 can be shown in the following way.
- For each vertex v in G, there must be at least one edge in any MST.
- Considering the edge of the smallest weight is useful as it can decrease the cost of the spanning tree.

Generalizing Claim 1.

- Claim 2 can be generalized further.
- Let T be a subtree of some MST of an undirected weighted graph G. Consider edges uv in G such that u is in T and v is not in T. Of all such edges, let e = xy be the edge with the smallest weight. Then T U {e} is also a subtree of some MST of G.

Generalizing Claim 1.

- Claim 2 allows us to expand a given sub-MST T.
- We can use Claim 2 to expand the current tree T.
- How to start?

Towards an Algorithm

- Let v be any vertex in the graph G. Pick v as the starting vertex to be added to T.
- T now contains one vertex and no edges.
- T is a subtree of some MST of G.
- Now, apply Claim 2 and extend T.

Towards an Algorithm

```
Algorithm MST(G, v)
Begin
    Add v to T;
    While T has less than n – 1 edges do
        w = vertex s.t. vw has the smallest weight
        amongst edges with one endpoint in T and
    another not in T.
        Add vw to T.
    End
End
```

Towards an Algorithm

- How to find w in the algorithm?
- Need to maintain the weight of edges that satisfy the criteria.
- A better approach:
  - Associate a key to every vertex
  - key[v] is the smallest weight of edges with v as one endpoint and another in the current tree T.
  - key[v] changes only when some vertex is added to T.
  - Vertex with the smallest key[v] is the one to be added to T.

Towards an Algorithm

- Suggests that key[v] need to be updated only when a new vertex is added to T.
- Further, not all key[v] may change in every iteration.
  - Only the neighbors of the vertex added to T.
  - Similar to Dijkstra's algorithm.

Towards an Algorithm

- Therefore, can maintain a heap of vertices with their key[ ] values.
- Initially, key[v] = infinity for every vertex except the start vertex for which key value can be 0.
- Perform DeleteMin on the heap. Let v be the result.
- Update the key[ ] value for neighbors w of v as:
  - key[w] = min{key[w], W(vw)}

Algorithm Using a Heap

```
Algorithm MST(G, u)
begin
    for each vertex v do key[v] = infty.
    key[u] = 0;
    Add all vertices to a heap H.
    While T has less than n-1 edges do
        v = deleteMin();
        Add v to T via uv s.t. u is in T
        For each neighbor w of v do
            if W(vw) > key[w] then DecreaseKey(w)
        end
   end
end
```

Algorithm Using a Heap

- The algorithm is called as Prim's algorithm.
- Runtime easy to analyze;
  - Each vertex deleted once from the heap. Each DeleteMin() takes O(log n) time. So, this accounts for a time of O(nlog n).
  - Each edge may result in one call to DecreaseKey(). Over m edges, this accounts for a time of O(mlog n).
  - Total time = O((n+m)log n).