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

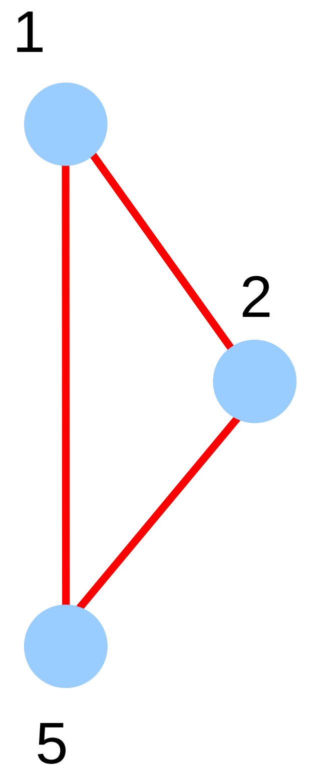
- In a tree, using level order traversal, each vertex is visited also exactly once.
  - Why?

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.

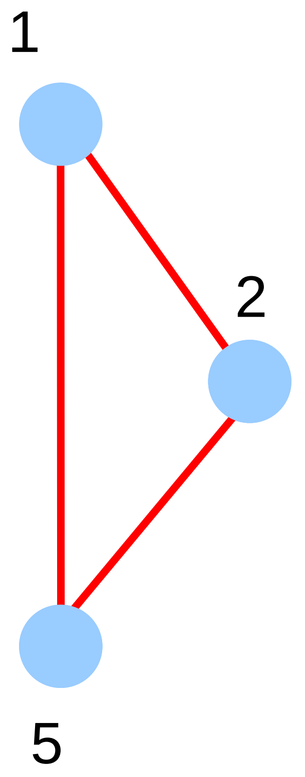
Breadth First Search

- 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.



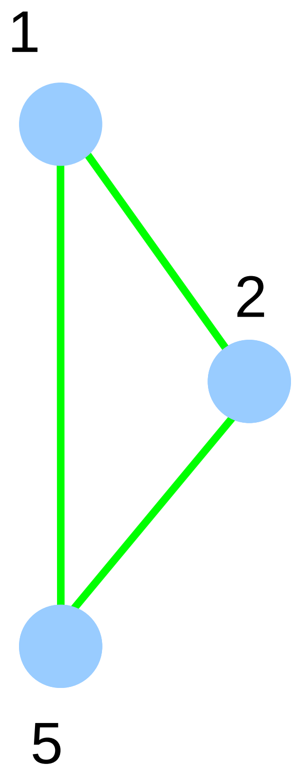
Breadth First Search

- 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.



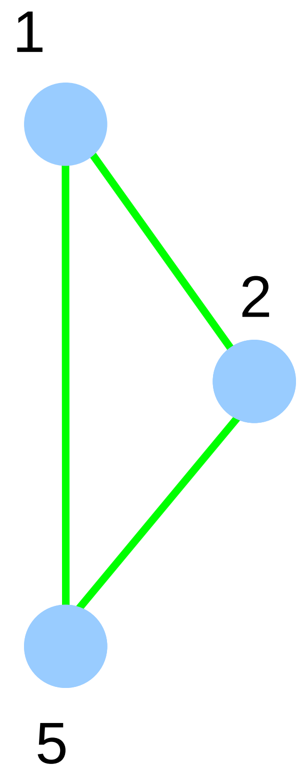
Breadth First Search

- 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.

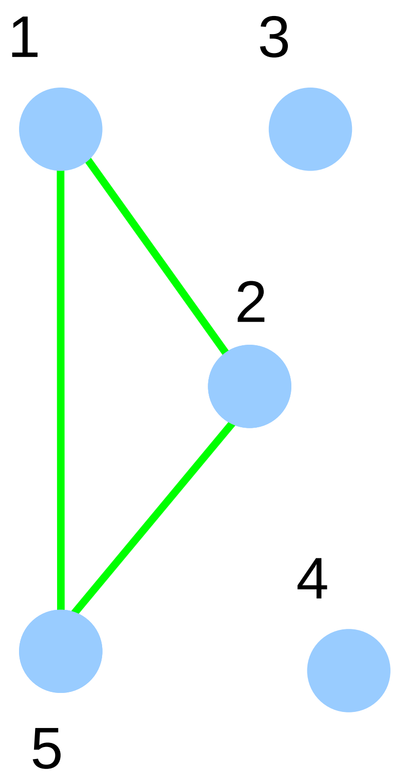


Breadth First Search

- 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

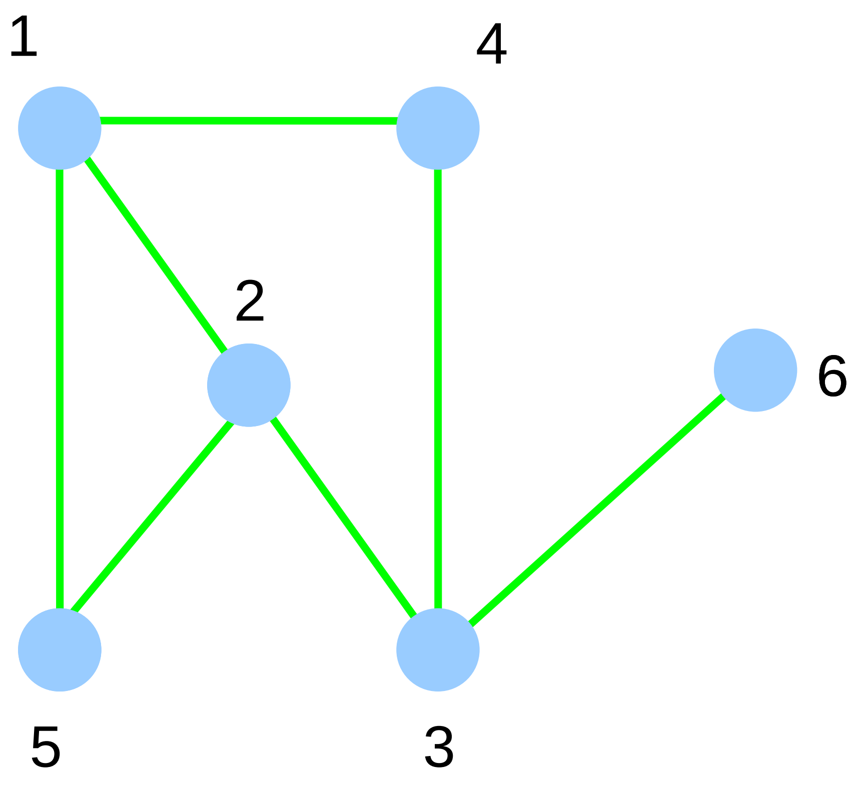
- So, for vertices v that are not reachable from s, can say that $d_s(v)$ is ∞
- Alike a level order traversal of a tree, can use a queue to store vertices in progress.

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?

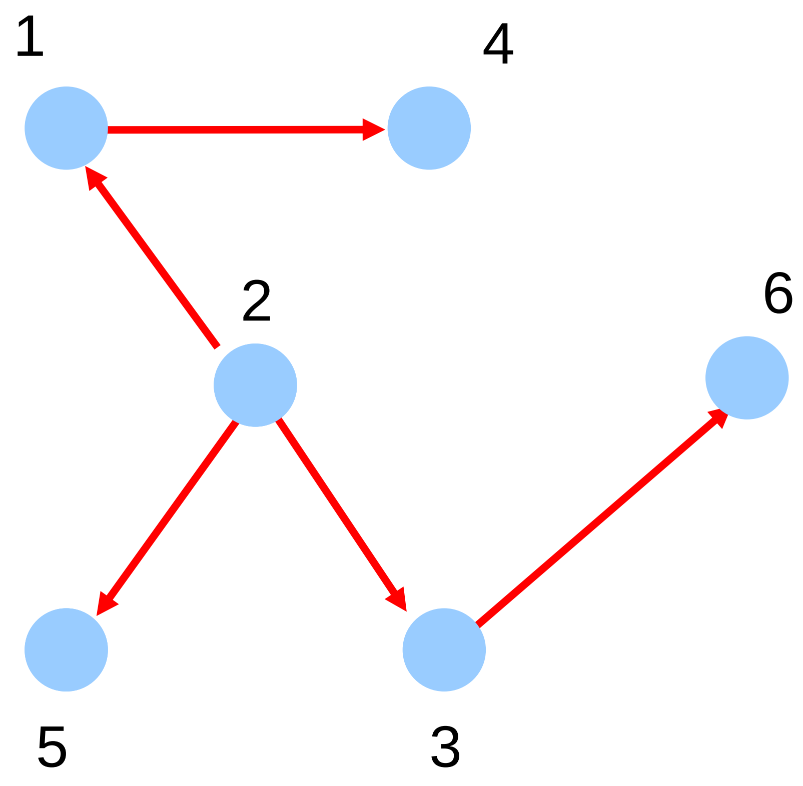
BFS – Additional Details

- 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
  - $V_\pi = \{v \in V : \pi(v) \,!\!= \text{NULL}\} \cup \{s\}$, 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

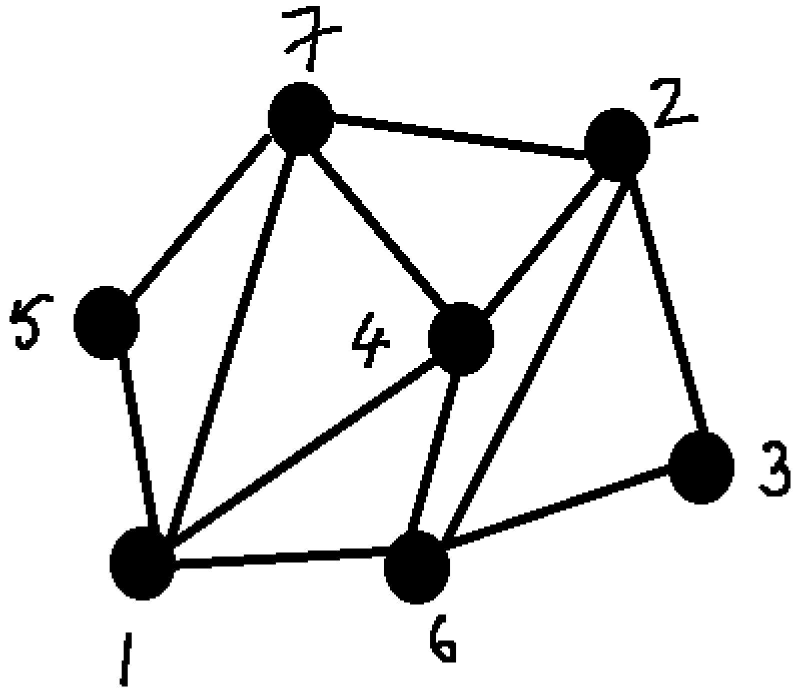
- Consider the time at which a vertex v has entered the queue.
- The state of v at that instant changes from NOT_VISITED to IN_PROGRESS.
- $d_s(v)$ changes to a finite value, and
- $d_s(v)$ can never change after that instant.

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

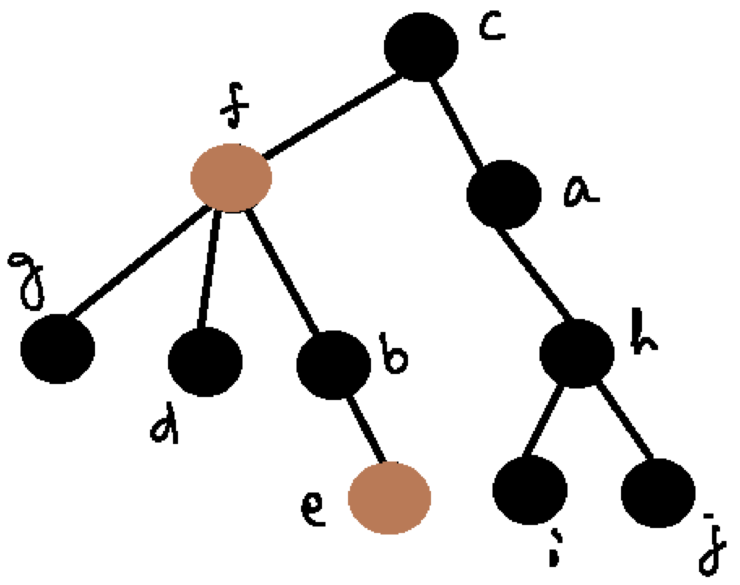
- Identify the tree- and the non-tree edges according to a BFS on the following graph. Choose vertex number 3 as the start vertex.

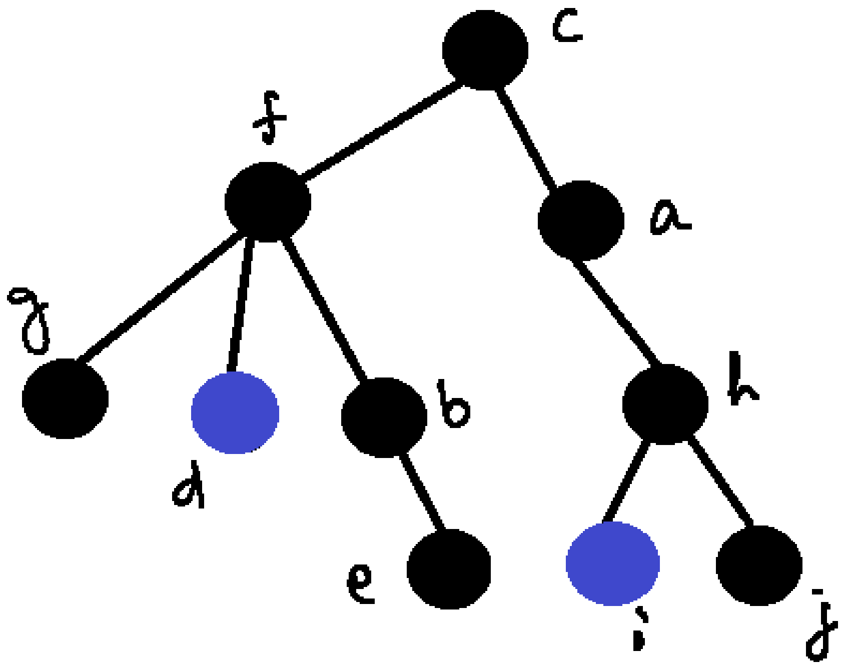


- Pick vertices in their order.

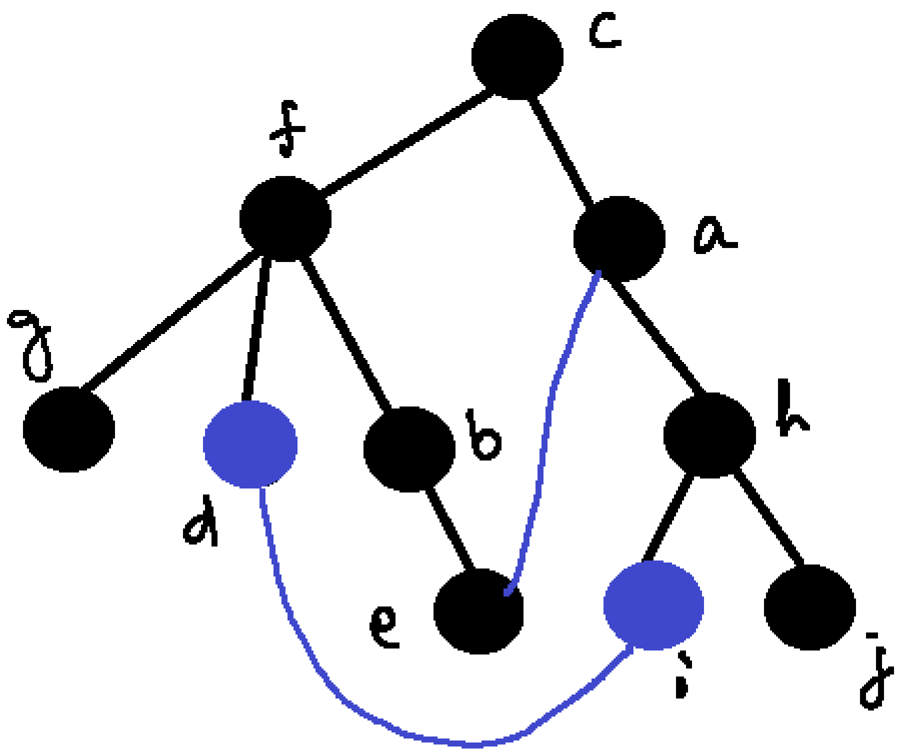
Classifying Edges – The Non-Tree Edges

- 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.



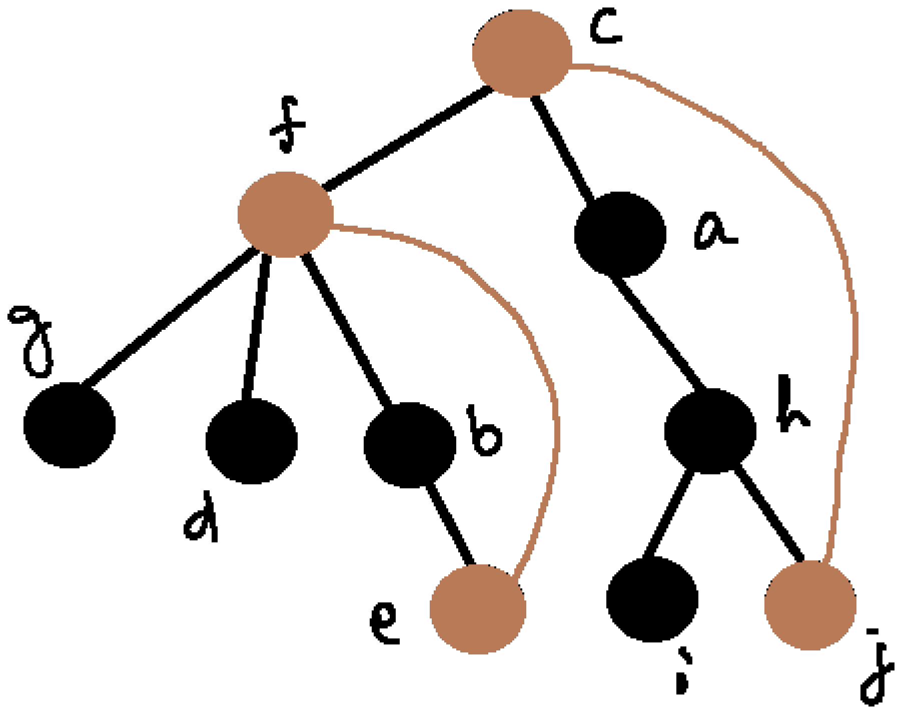


Classifying Edges – The Non-Tree Edges



- 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



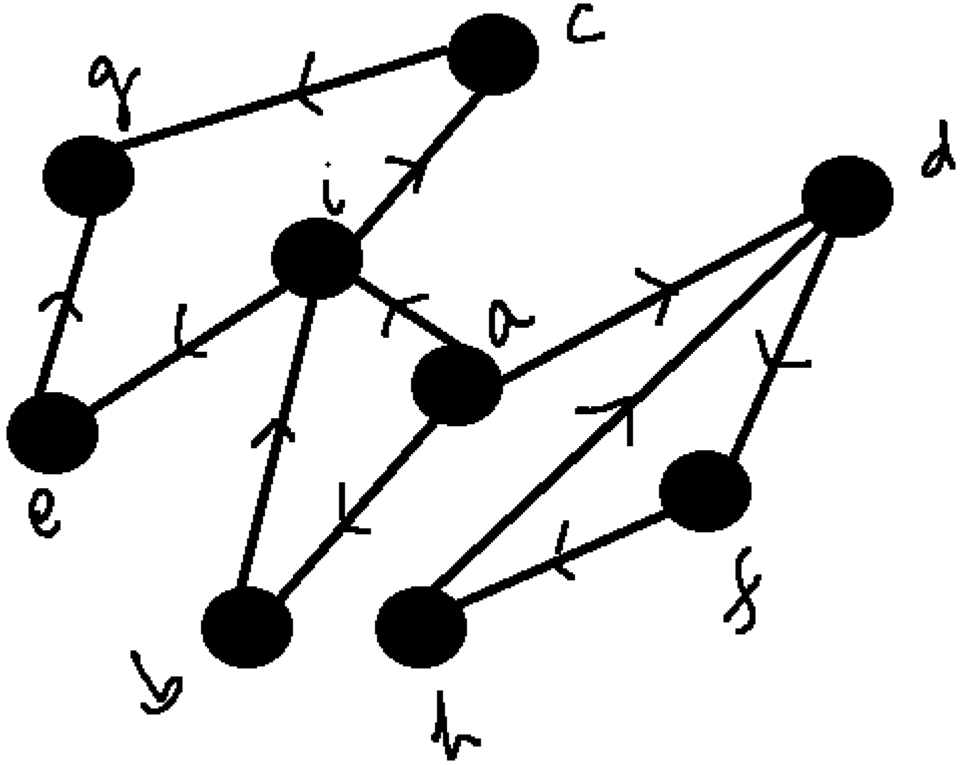
- For any pair of vertices u, v with (u,v) an edge in G:
  - Either they share an ancestor-descendant relation in T_{BFS}.
  - If u is an ancestor of v, then (u,v) is a forward edge.
  - If u is a descendant of v, then (u,v) is a back edge.

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.

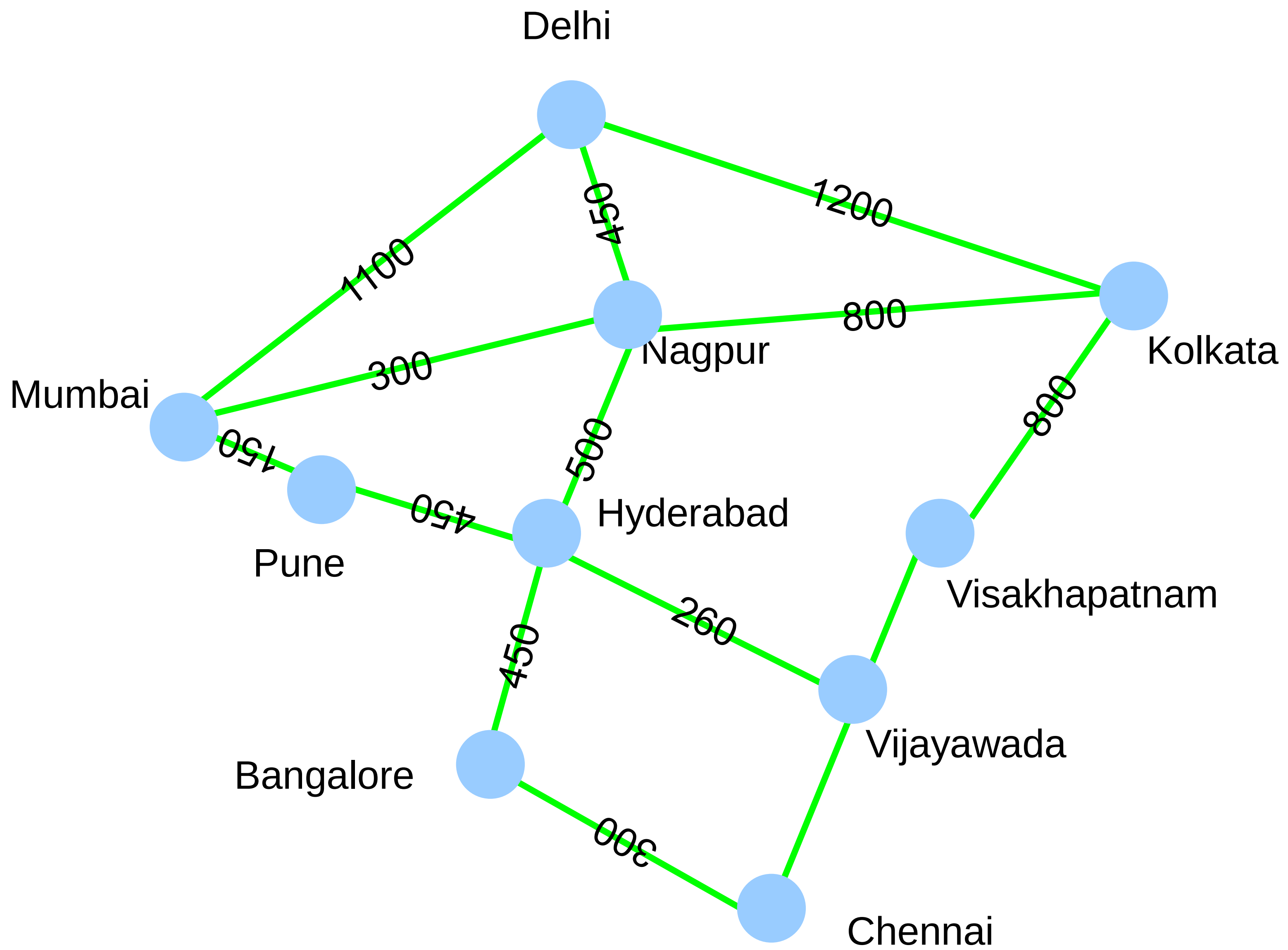
Towards Weighted BFS

- 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

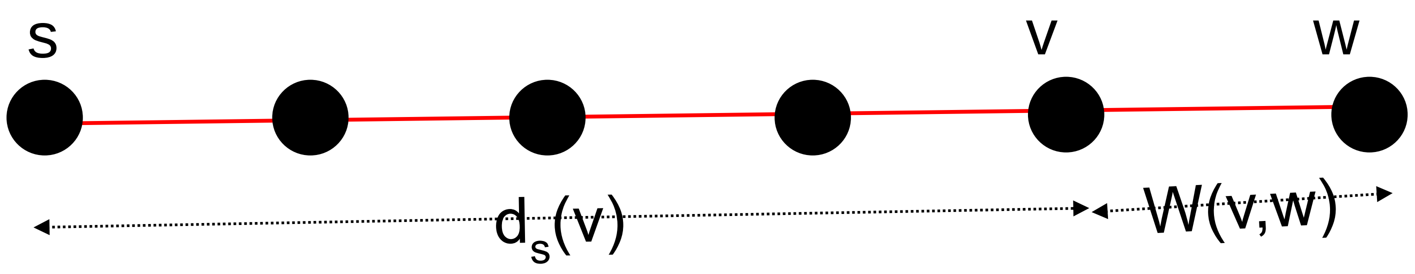
SSSP

- 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



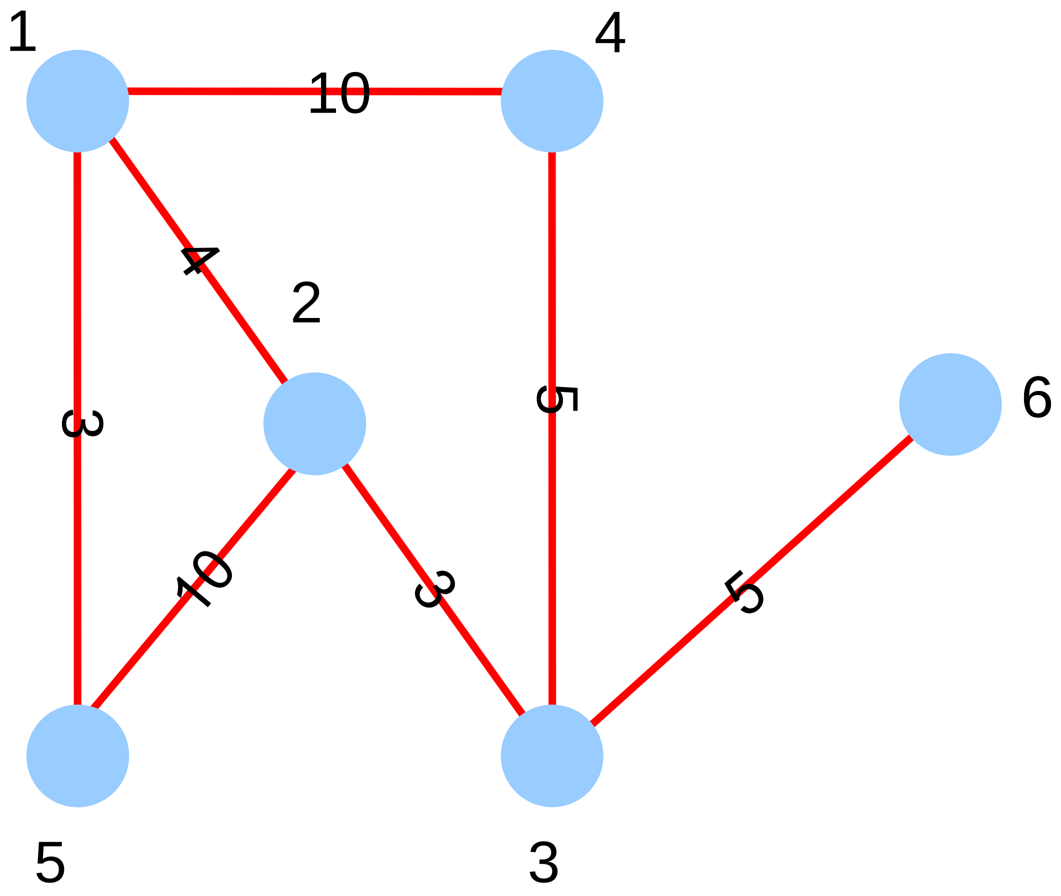
- Update $d_s(v)$ with weights.
- Also, weights on edges mean that if v is a neighbor of w in the shortest path from s to w, then $d_s(w) = d_s(v) + W(v,w)$.
  - Instead of $d_s(w) = d_s(v) + 1$ as in BFS.
- We will call this as the first change to BFS.

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.

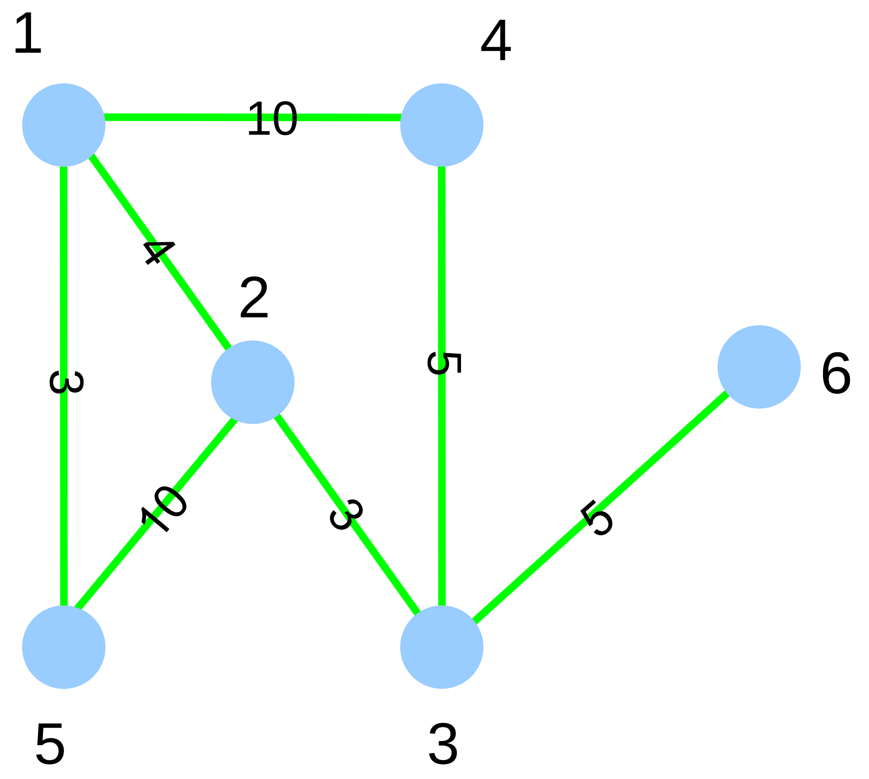
Weighted BFS

- Consider s = 2 and perform weighted BFS.



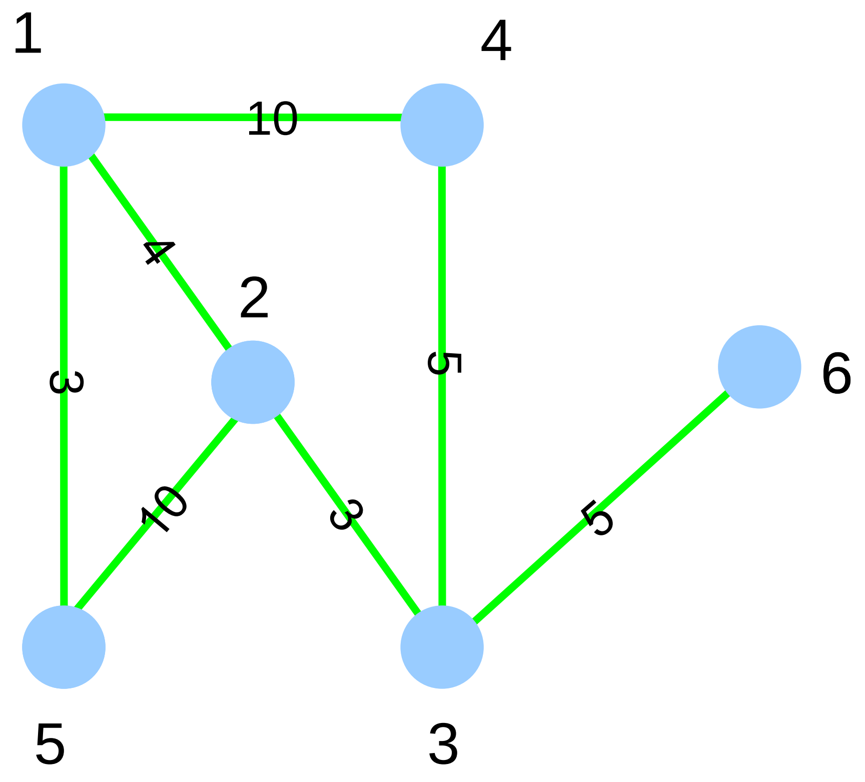
Weighted BFS

- 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.



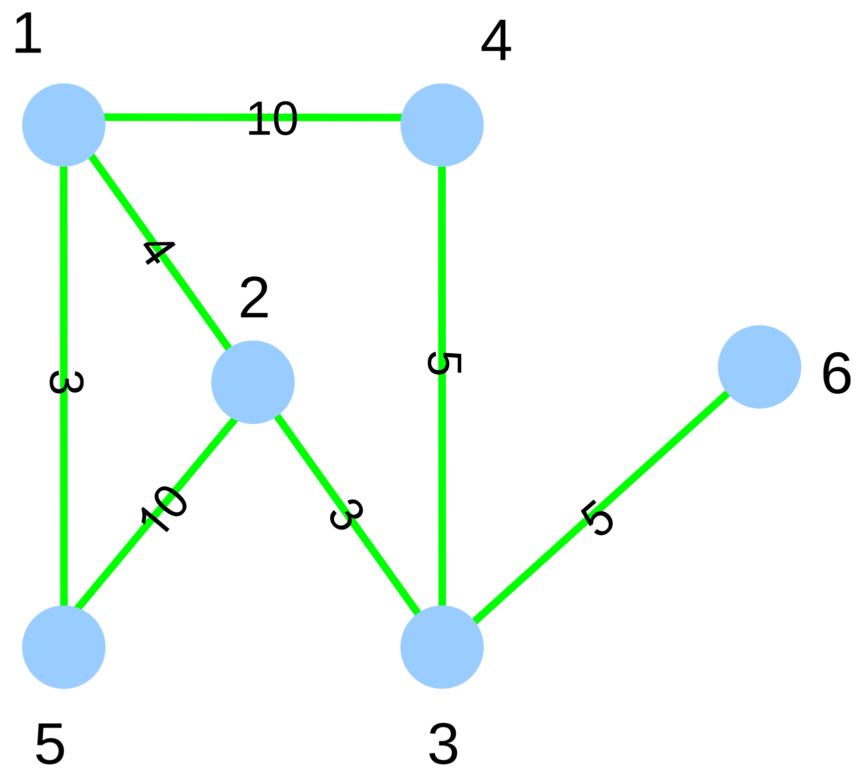
Weighted BFS

- 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.



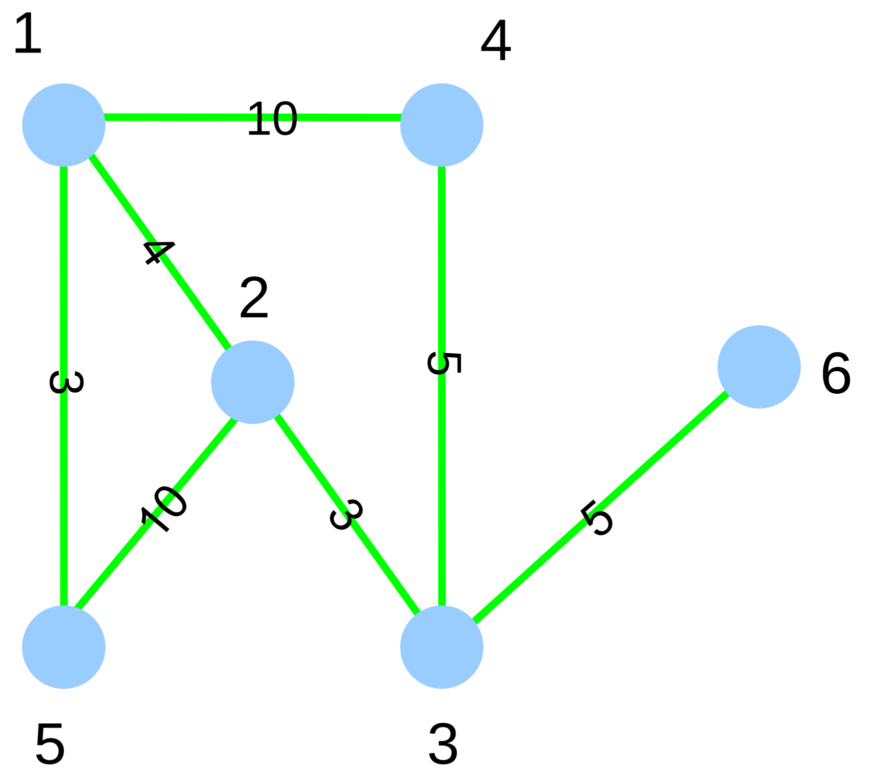
Weighted BFS

- 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

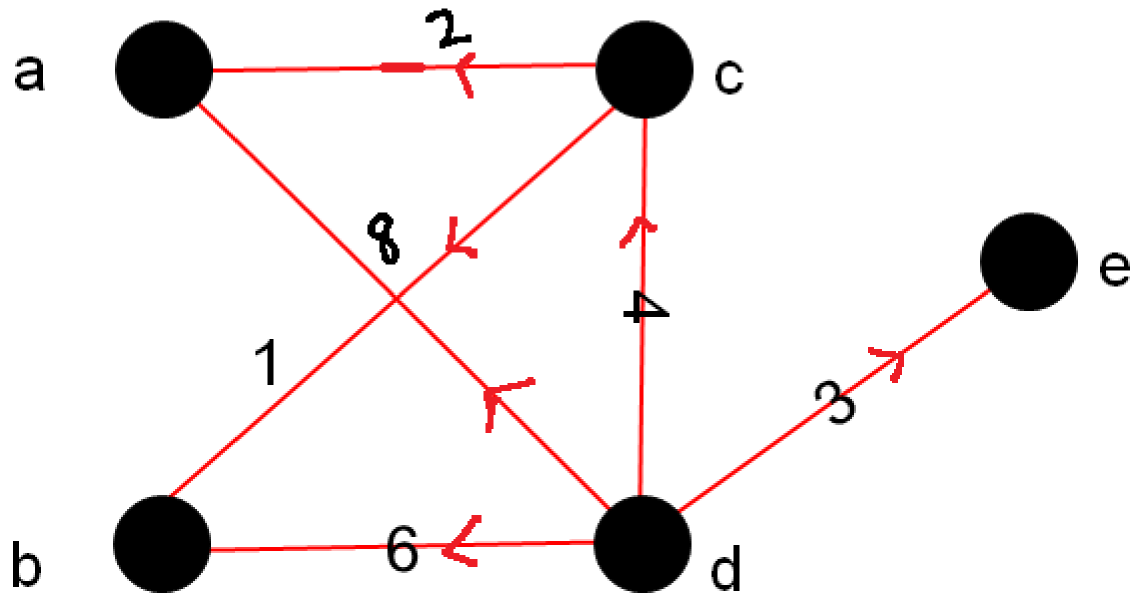
```
Algorithm SSSP(G,s)
begin
    for all vertices v do
        $d(v) = \infty$; $\pi(v) = NIL$;
    end-for
    $d(s) = 0$;
    for n-1 iterations do
        for each edge (v,w) do
            if $d(w) > d(v) + W(v,w)$ then
                $d(w) = d(v) + W(v,w)$; $\pi(w) = v$;
            end-if
  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



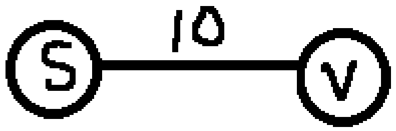
Thinking about the Bellman-Ford Algorithm

```
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

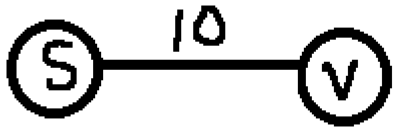
- Consider the source vertex s.
- For s, d(s) = 0 is the best possible result.
- So, s is FINISHED.



- 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.



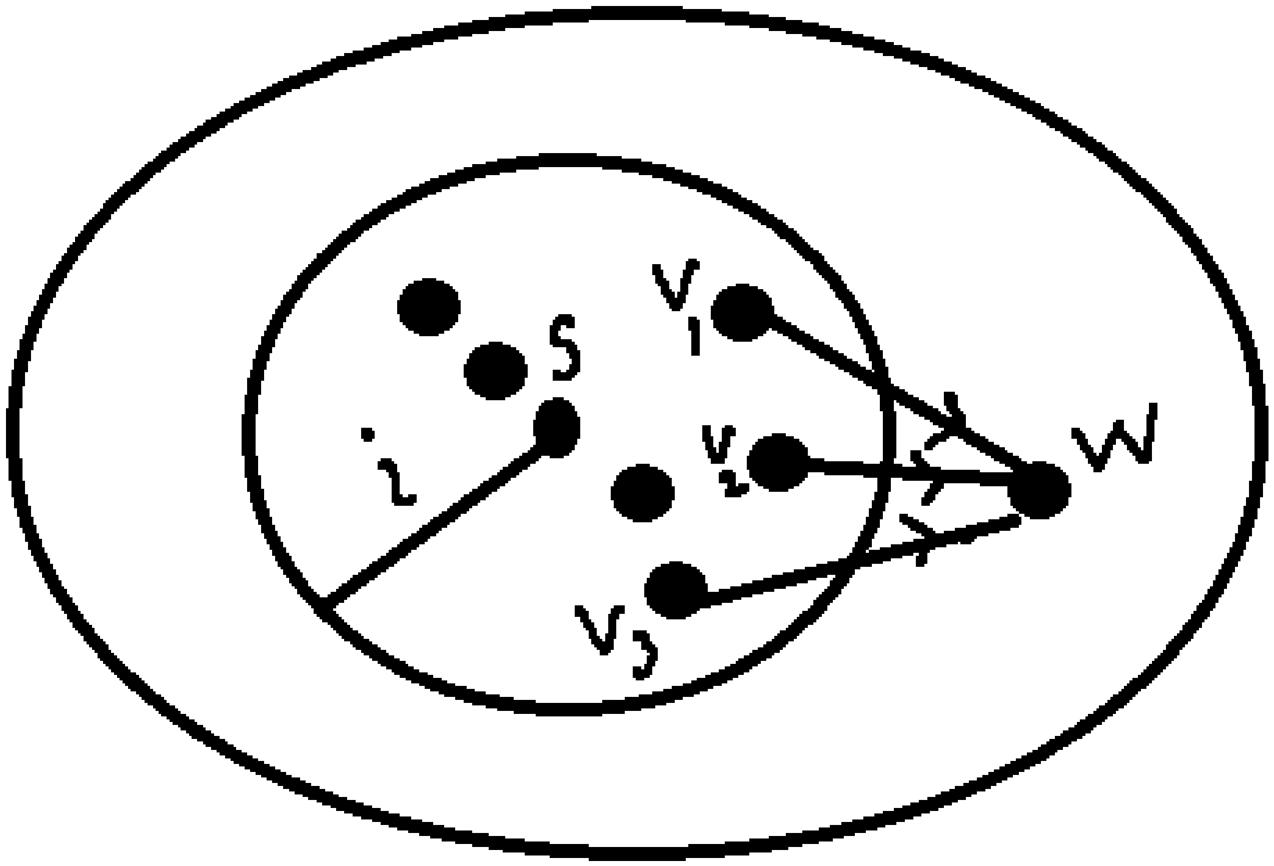
- Now consider a vertex v such that the shortest path from s to v contains only one edge, say (s,v).
- The edge (s,v) appears at some iteration of the second for loop in the first iteration of the main loop.
- At that point, d(v) is set correctly.
- Does that mean that all neighbors of s FINISH in one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



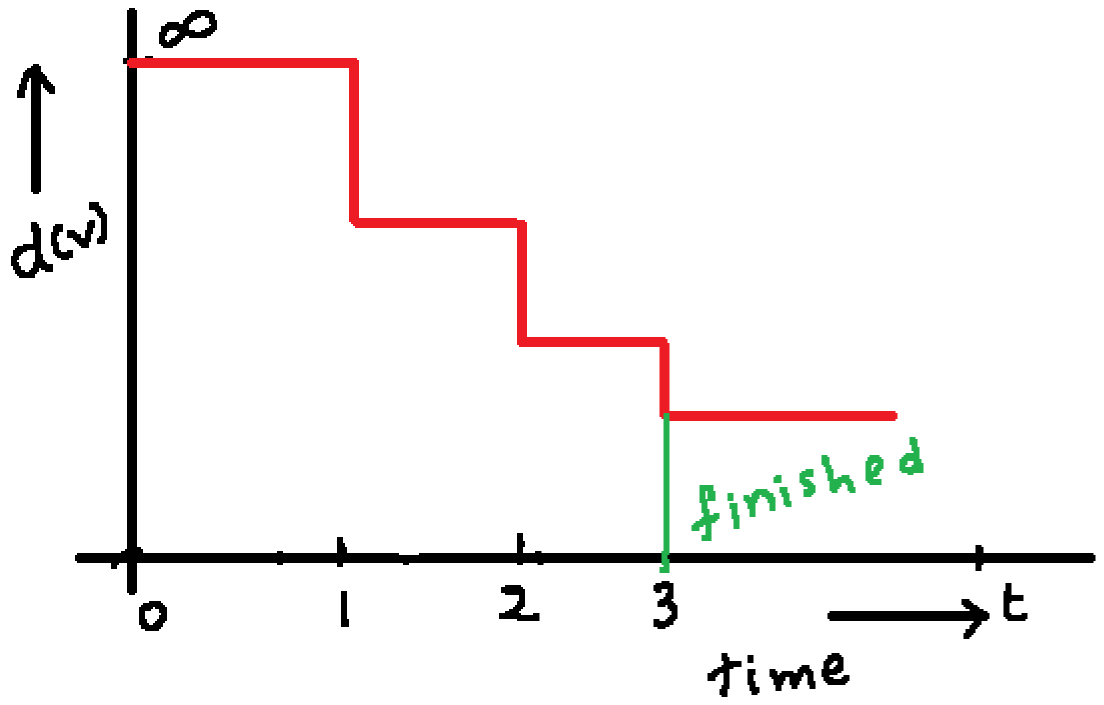
Algorithm SSSP

- 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

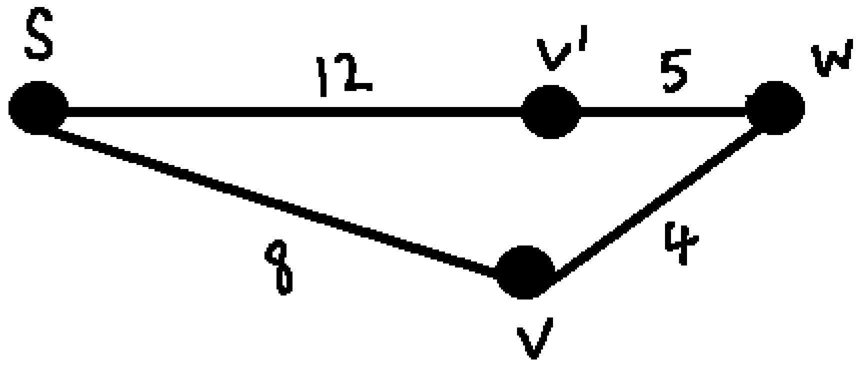
- The time taken by the Bellman-Ford algorithm is too high compared to that of BFS.
- Can we improve on the time requirement?
- Most of the time is due to
  - Repeatedly considering edges, and as a result
  - Updating d(v) possibly many times
- Need to know how to stop updating d(v) for any vertex v.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

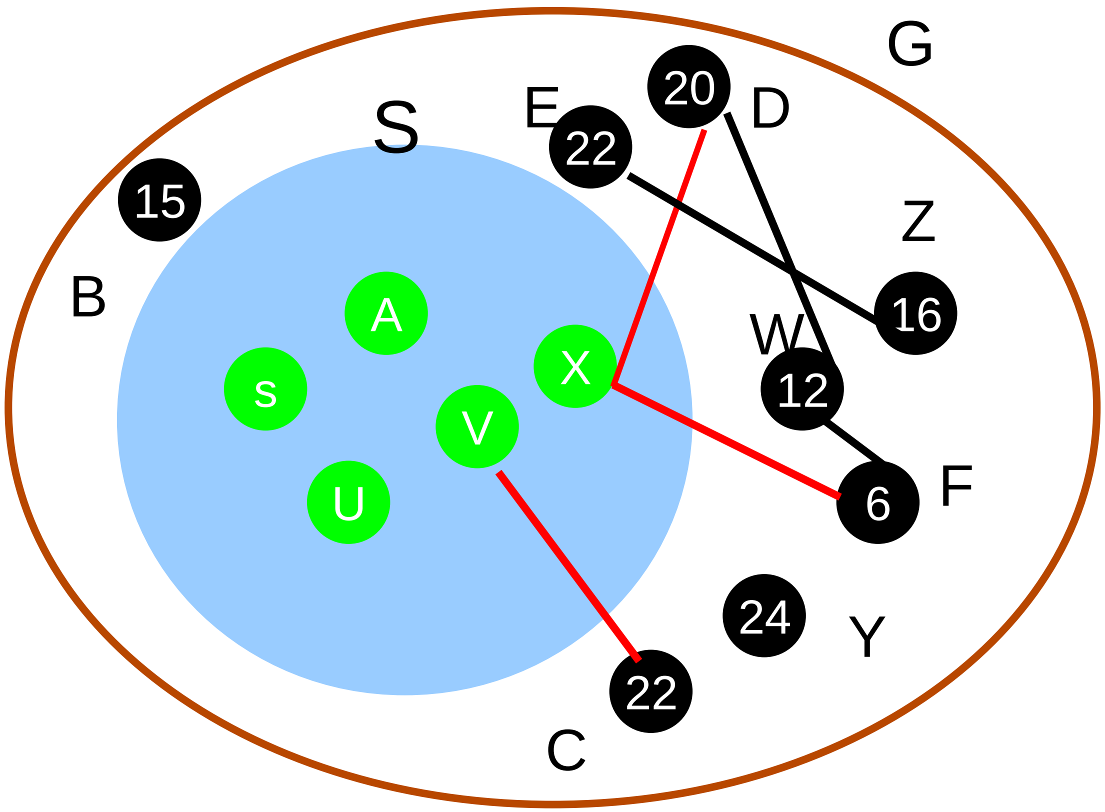
To Improve the Runtime

- For this to happen, consider the following.
  - A few vertices, say S, are FINISHED.
  - 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

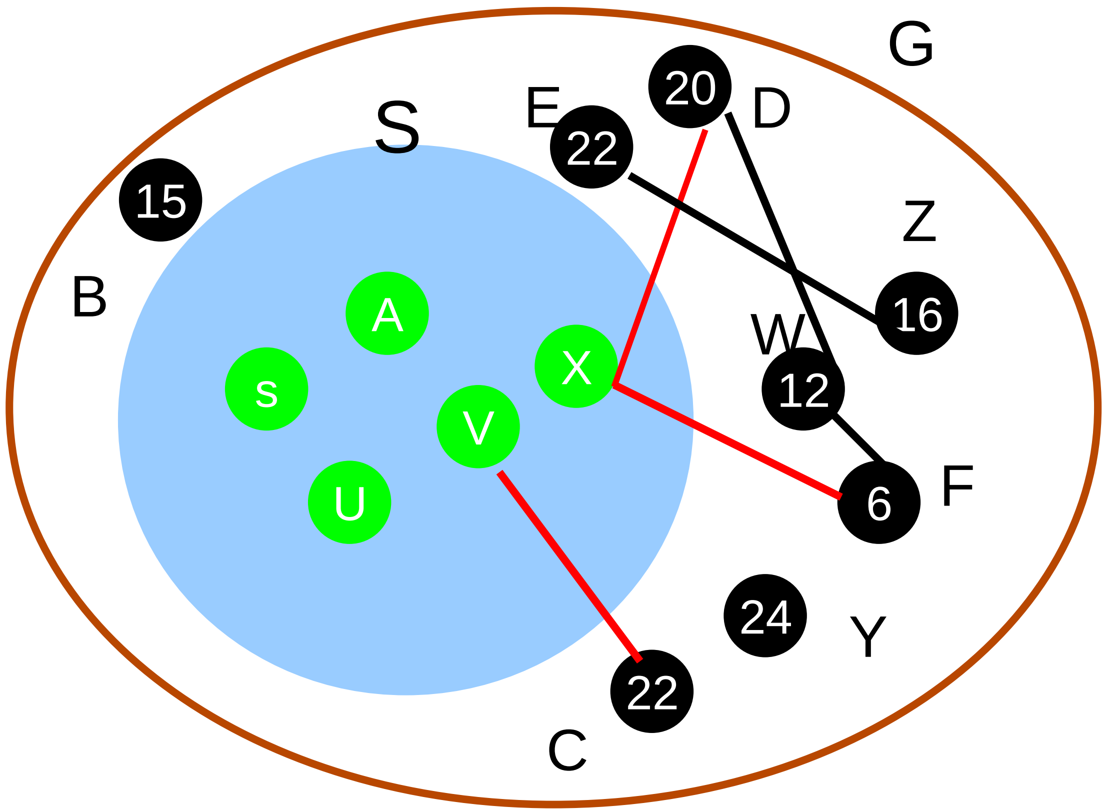
- For this to happen, consider the following.
  - Let each edge have a positive weight.
  - A few vertices, say S, are FINISHED.
  - Plus, all the edges with at least one endpoint in S are the only edges considered.
  - Other vertices in V \ S, have some d() value.
  - Which of these cannot improve d()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

The Setting



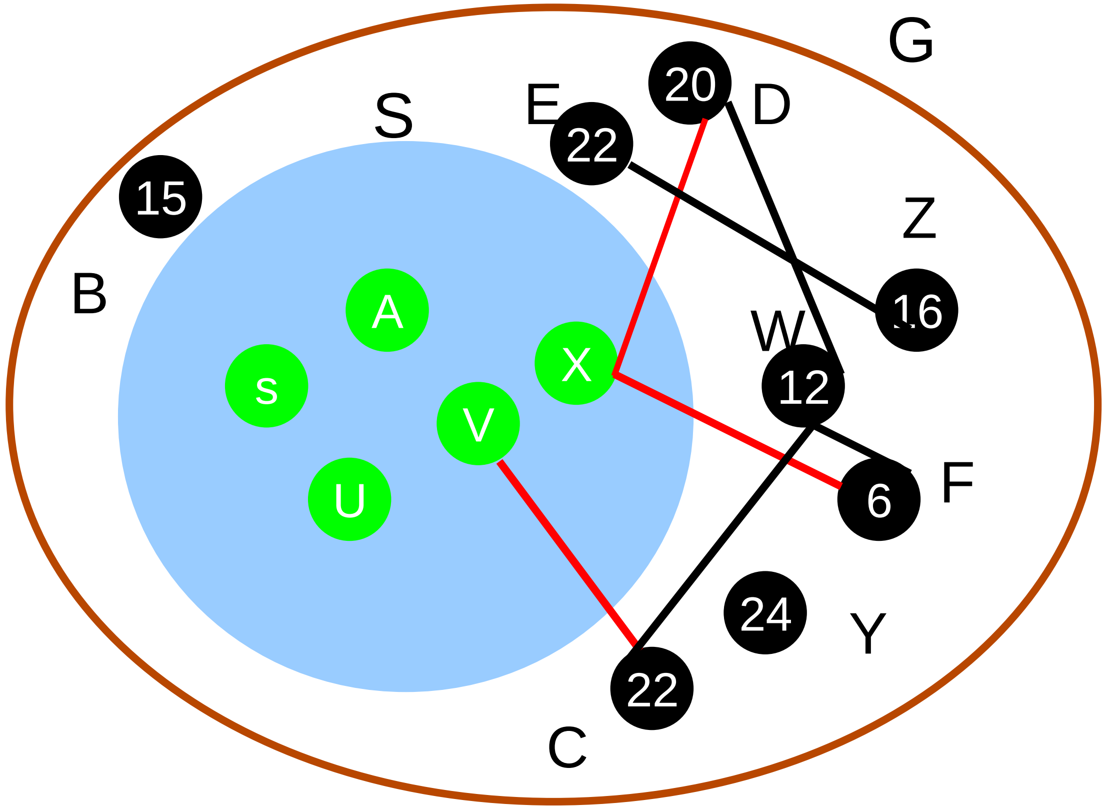
- Green vertices are FINISHED.
- Red edges, edges with at least one end point as a green vertex are the ONLY edges PROCESSED.
- Numbers on black vertices indicate their d() value using only green vertices as intermediate vertices.

The Setting



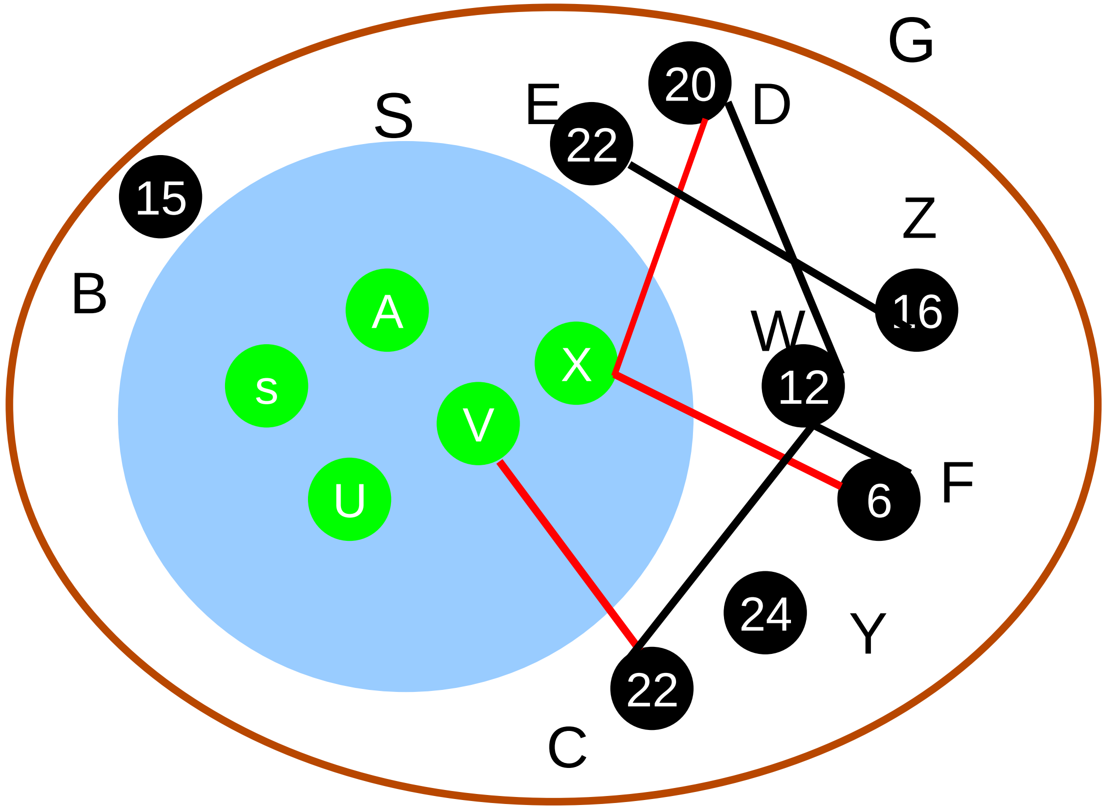
- Suppose we want to add one more vertex to the set S.
- Which of the black vertices is FINISHED?

The Setting



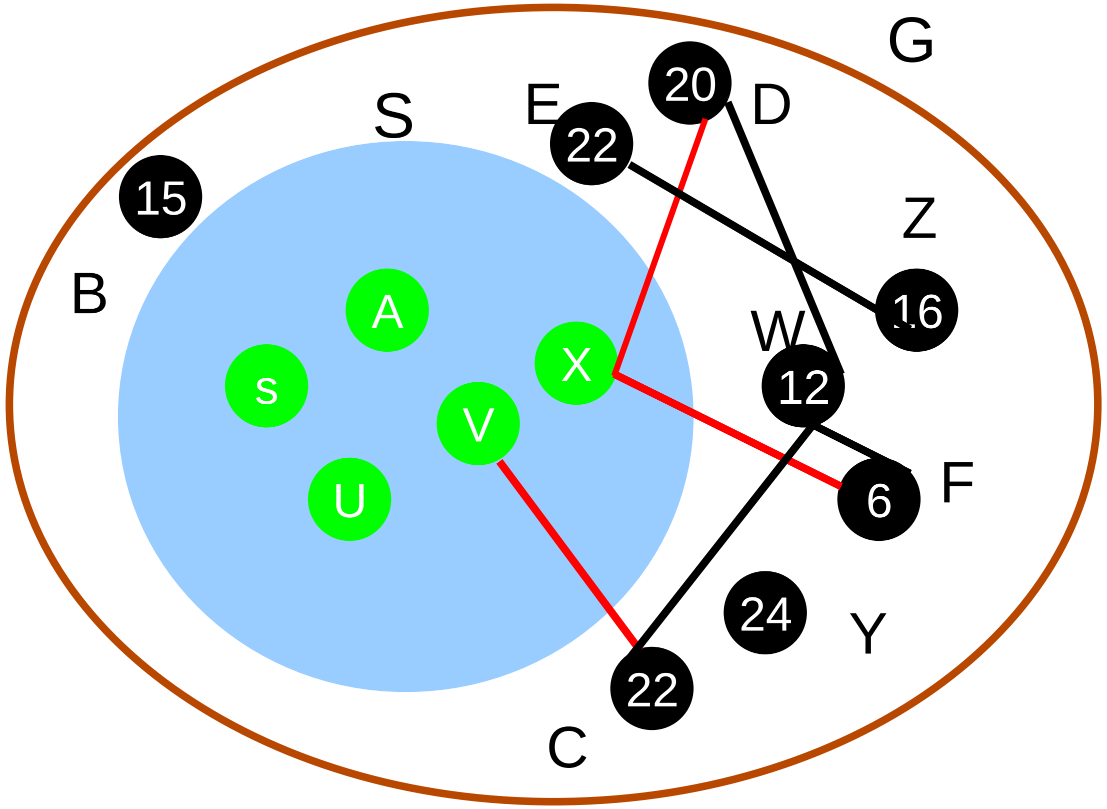
- Notice that there could be edges between the black vertices also.
  - None of them are processed so far.

The Setting



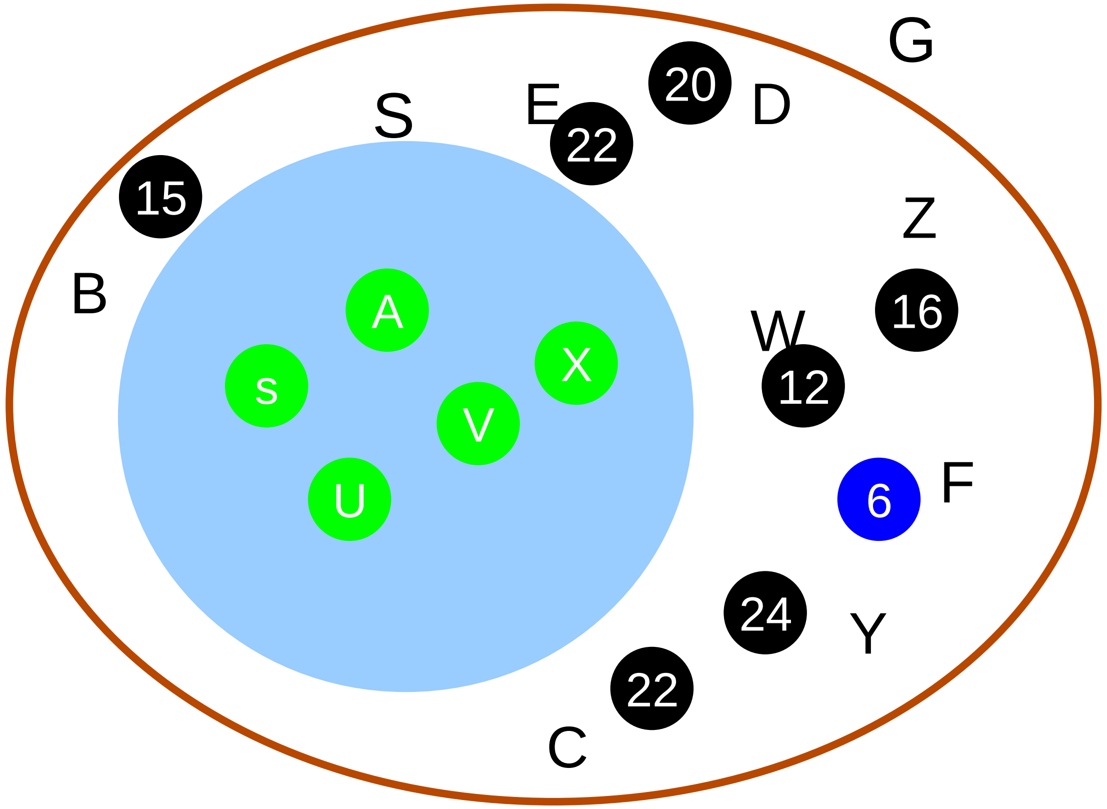
- Consider the vertex v with the smallest d() value among the black vertices.
- Any more decrease to d(v) would involve using at least one more edge between two black vertices.

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.