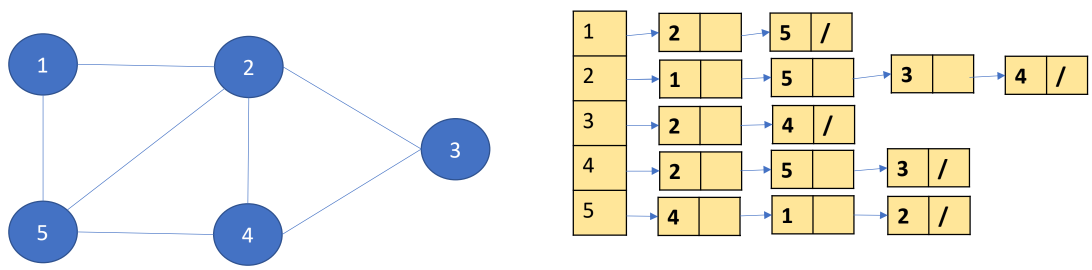
Today's topics

- BFS
- DFS

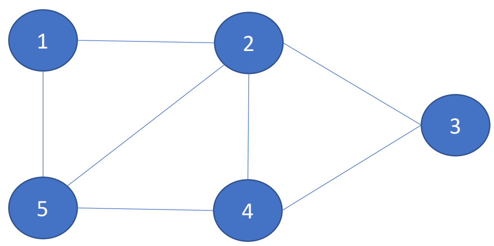
References

- Read chapter-20 of the CLRS book
- Read chapter-6 from Goodrich and Tamassia book
- https://en.wikipedia.org/wiki/Depth-first_search

Adjacency list (undirected)

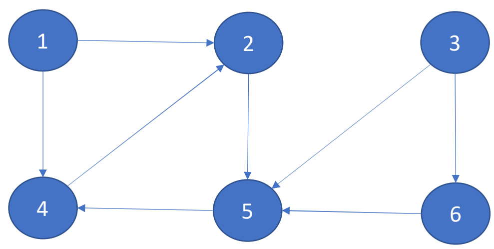


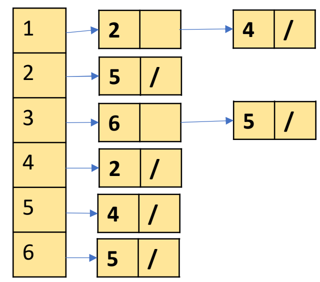
Adjacency matrix (undirected)



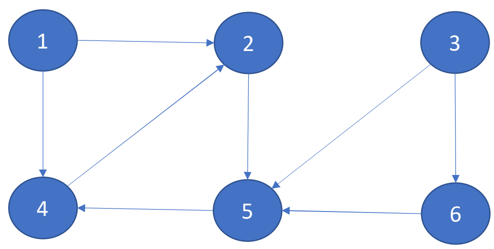
	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

Adjacency list (directed)





Adjacency matrix (directed)



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

Adjacency matrix vs. adjacency list

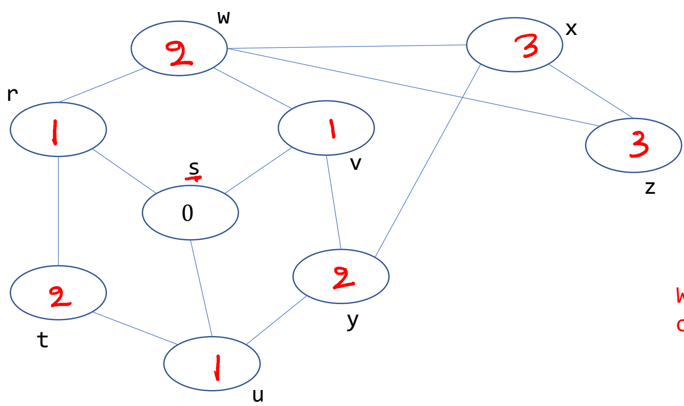
- Checking if there is an edge between two vertices can be done in O(1)
- Adjacency-matrix is beneficial for dense graphs in which $|E|$ is close to $|V|^2$
- Space requirement is $O(|V|^2)$
- Iterating all outgoing edges from a vertex v requires $O(|V|)$ operations

- Checking if there is an edge between two vertices u, v can be done in O(out_degree(u)+out_degree(v)), in a directed graph and min(degree(u), degree(v)) in an undirected graph
- Adjacency-list is beneficial for sparse graphs in which the $|E|$ is much less than $|V|^2$
- Space requirement is $O(|V|+|E|)$
- Iterating all outgoing edges from a vertex v requires O(degree(v)) operations

Shortest path

Distance

distance(s, v) is defined as the minimum number of edges needed to go from s to v.

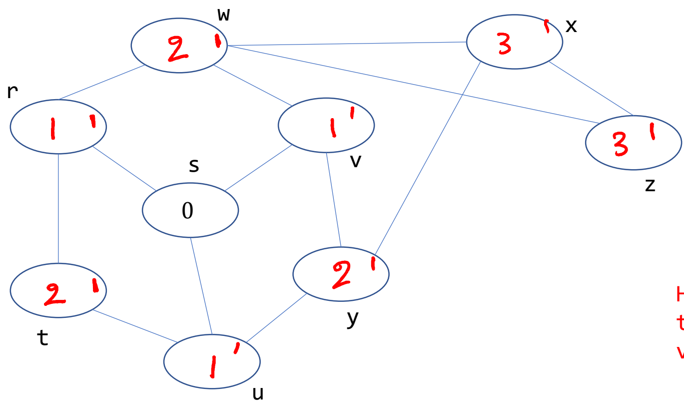


What is the distance of all nodes from s?

The distance of all vertices form s is shown on this slide.

Distance

distance(s, v) is defined as the minimum number of edges needed to go from s to v.

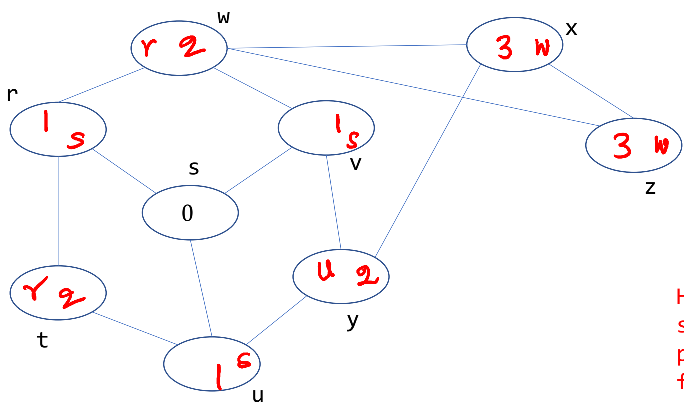


How can we compute the distance of all vertices from s?

To compute the distance of each vertex from s, first, we can find all adjacent vertices of s and update the distance of those vertices to 1. Next, we find the adjacent vertices of the set of vertices at a distance of 1 (as computed in the previous step) and set the distance of these vertices to 2. Next, we can identify the set of vertices adjacent to the set of vertices at a distance of 2 and set the distance of these vertices to 3. We can continue in this manner until we update the distance of each vertex reachable from s. Notice that we need to process all vertices with distance 1 before we can process the vertices at a distance 2 and so on. We can keep the vertices that still need to be processed in a queue. Because we are discovering vertices with distance 1 before distance 2 and so on, we want to process a vertex at the larger distance after the vertices at a smaller distance; we can add a vertex to the queue when it is discovered and process them in the order in which they are added to the queue. We need to update the distance only once when a vertex is added to the queue. If we reach a vertex that has already been added to the queue (or discovered) while traversing the neighbors, we ignore the vertex.

Distance

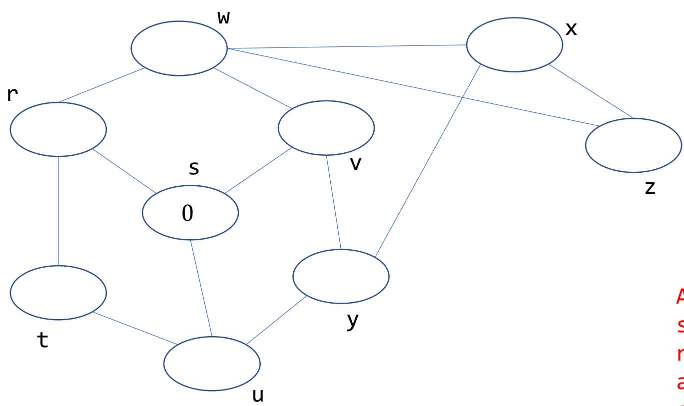
distance(s, v) is defined as the minimum number of edges needed to go from s to v.



How can we also store the shortest path to every node from vertex s, along with the distance?

To store the shortest path whenever a new vertex is discovered, in addition to distance, we also store the predecessor vertex information in the node corresponding to the vertex.

Distance



distance(s, v) is defined as the minimum number of edges needed to go from s to v.

Shortest path to z

s r w z

s v w z

Apart from the shortest paths reported by the algorithm, are other shortest paths possible?

Yes, multiple shortest paths are possible. For example, there are two shortest paths to reach z from s listed on this slide. The algorithm we discussed identifies one of the shortest paths.

Breadth-first search (BFS)

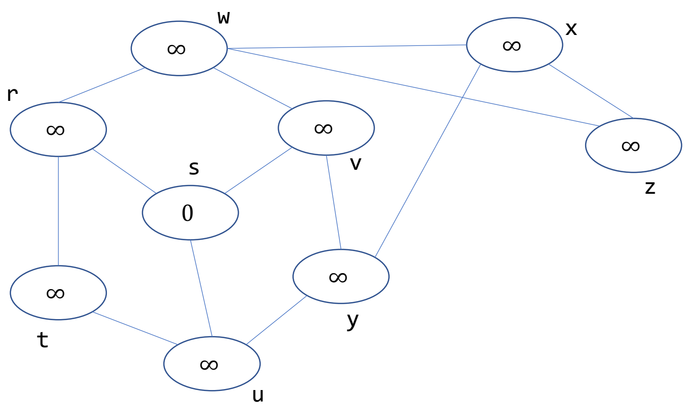
Breadth-first search (BFS)

- Given a graph G = (V, E) and a source vertex s, BFS explores all the vertices that can be reached from s
  - BFS also computes the distance from s to every other reachable vertices v, where distance is the smallest number of edges needed to go from s to v
  - BFS also produces a breadth-first tree which is a spanning tree that connects all the vertices reachable from s
    - In the breadth-first tree, a simple path from s to any other vertex v is also a shortest path from s to v

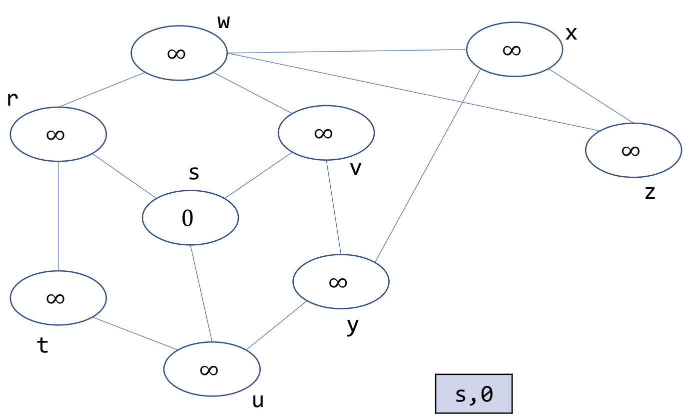
BFS

- The BFS(G, s) works as follows
  - First, we identify all vertices that are at a distance 1 from the source vertex s
    - We also call them vertices at level 1
    - The predecessor of all these vertices in the BFS tree is s
  - Next, we identify all vertices that are at a distance 2 from the source vertex s
    - We also call them vertices at level 2
    - The predecessors of all these vertices in the BFS tree are one of the vertices at level 1
  - Next, we identify all vertices that are at a distance 3 from the source vertex s
    - We also call them vertices at level 3
    - The predecessors of all these vertices in the BFS tree are one of the vertices at level 2
  - And so on
  - until we have discovered all vertices that can be reached from vertex s

BFS

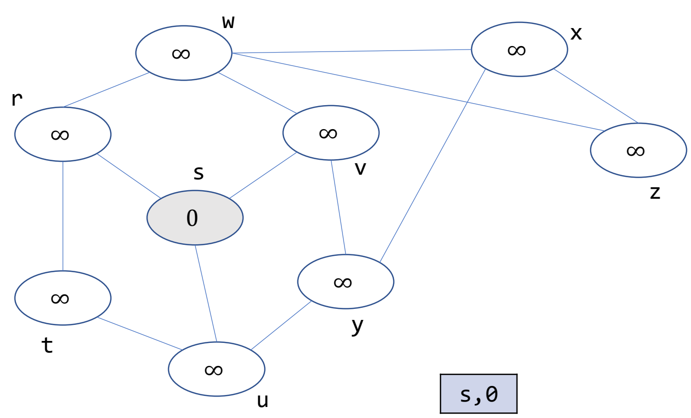


BFS

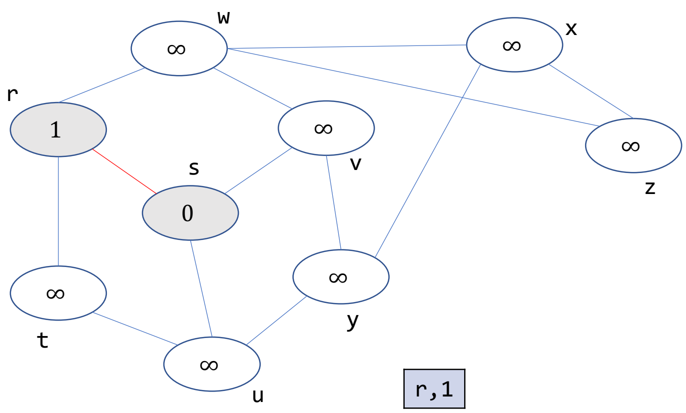


The following slides show the working of the BFS algorithm in detail. Whenever a vertex is added to the queue, it's marked gray. Whenever a vertex is removed from the queue, it's marked black. The edges on the shortest path (discovered using BFS) are shown in red. The edges traversed during the BFS algorithm but not on the shortest path are shown in black.

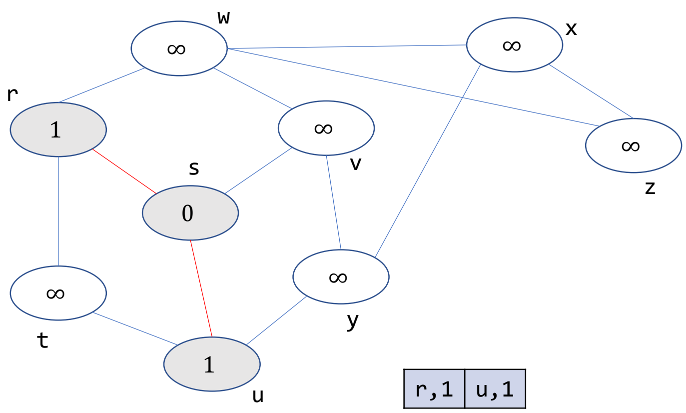
BFS



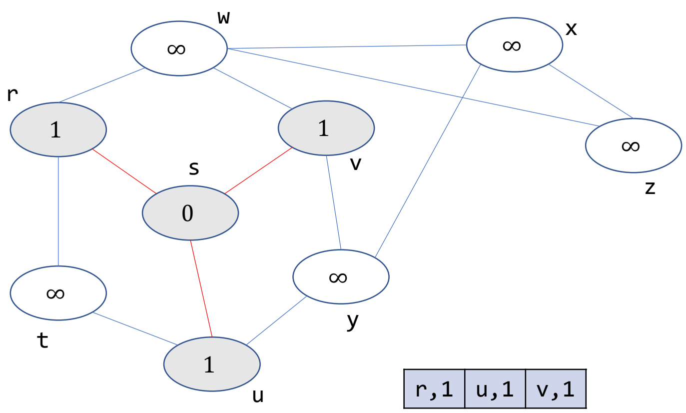
BFS



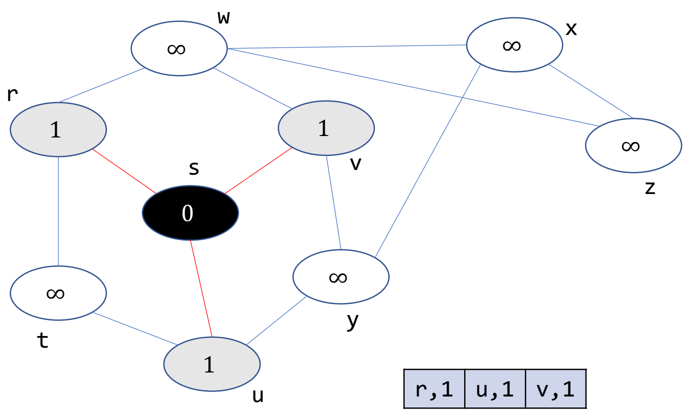
BFS



BFS

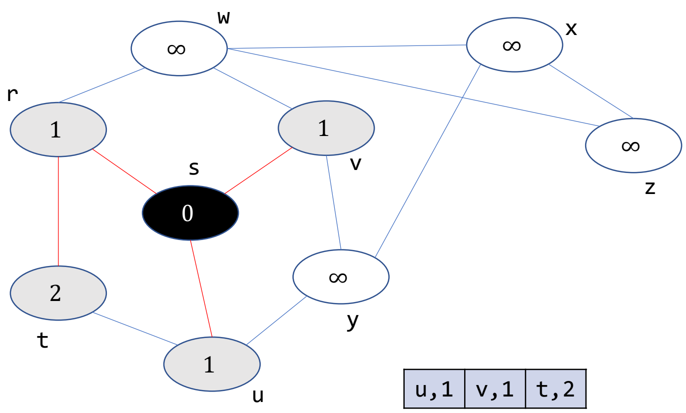


BFS



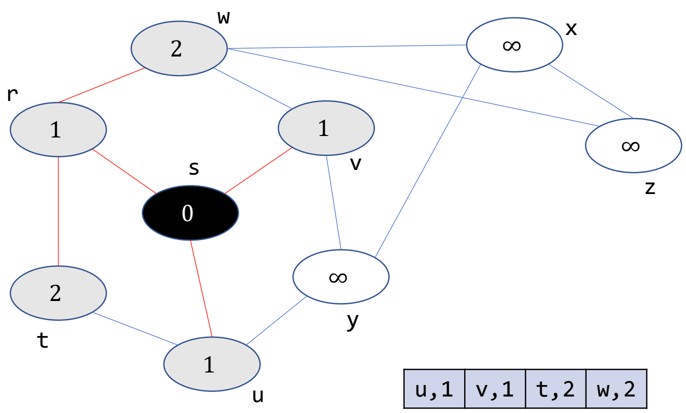
r,1	u,1	v,1

BFS

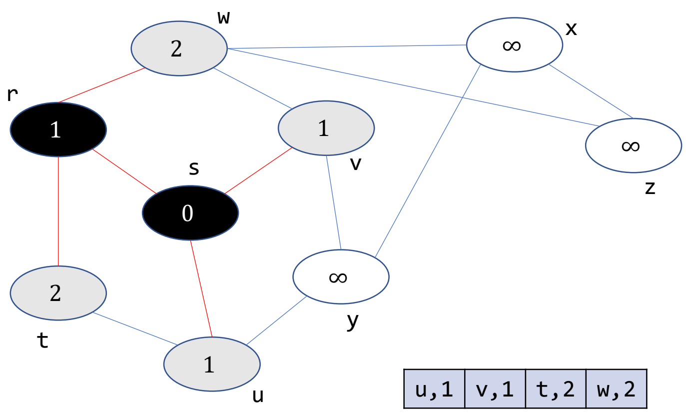


u,1	v,1	t,2

BFS

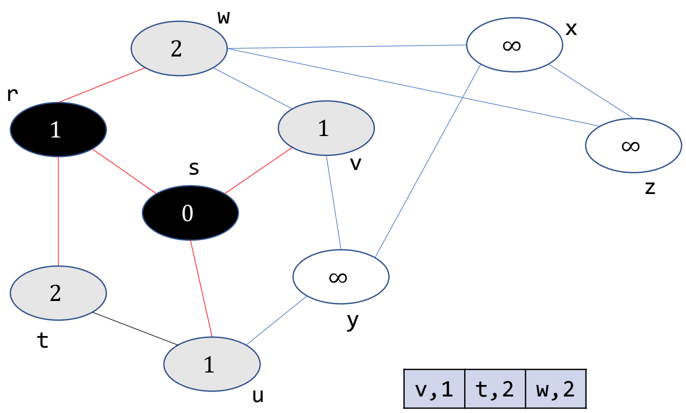


BFS

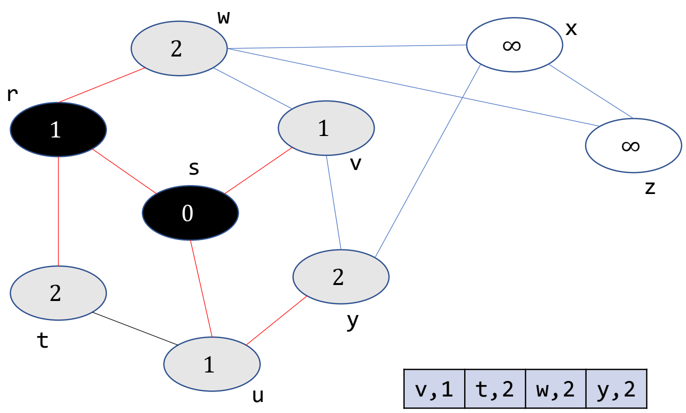


u,1	v,1	t,2	w,2

BFS

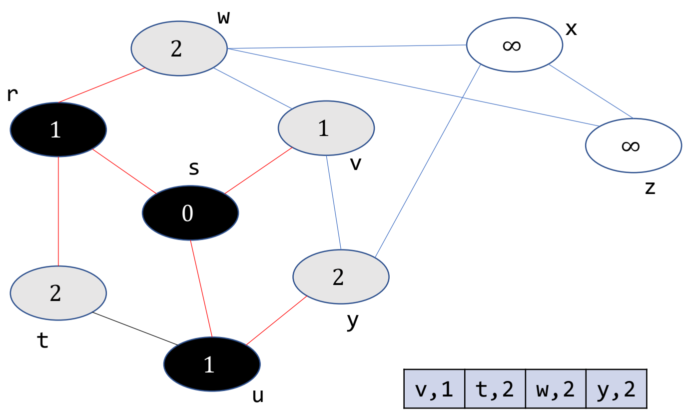


BFS



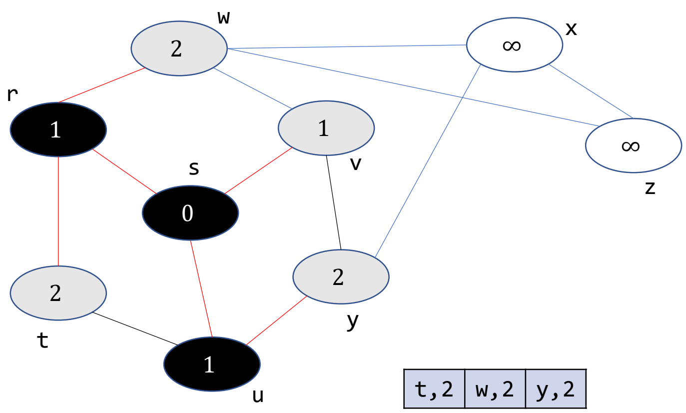
v,1	t,2	w,2	y,2

BFS

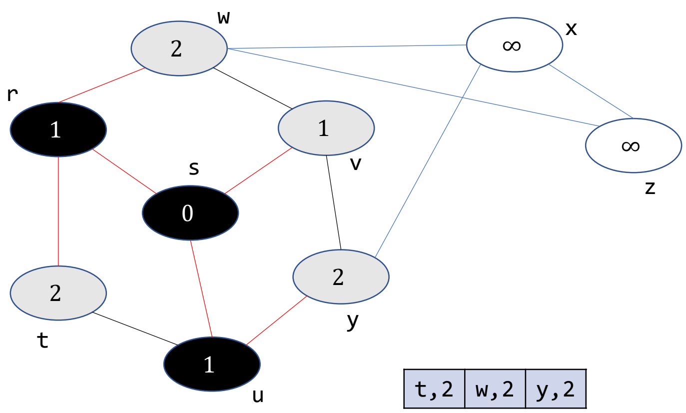


v,1	t,2	w,2	y,2

BFS

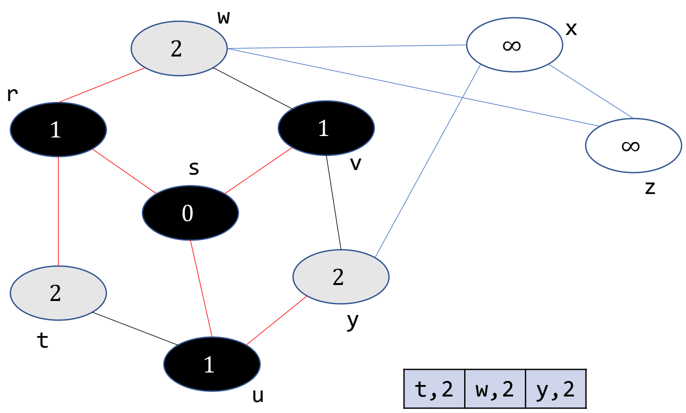


BFS



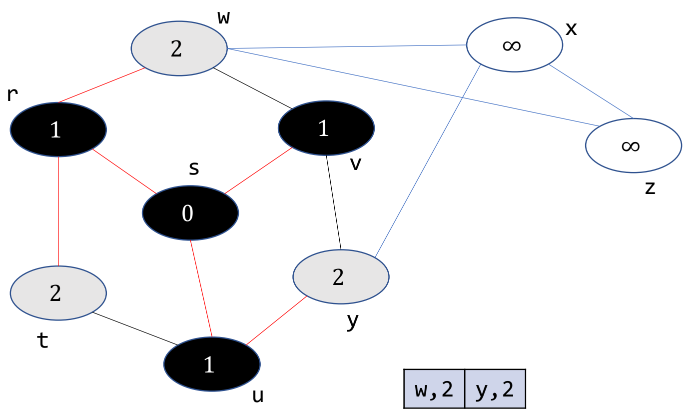
t,2	w,2	y,2

BFS

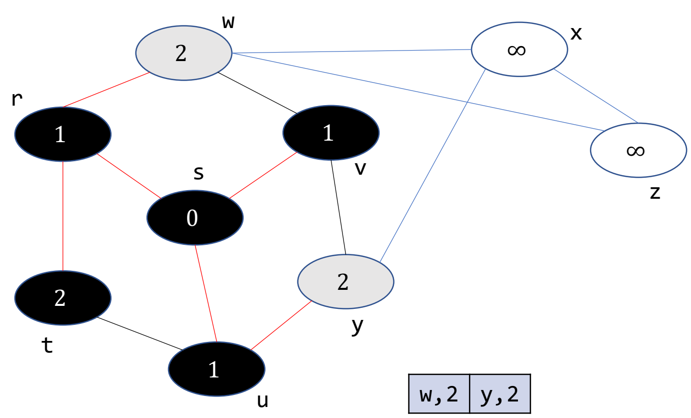


t,2	w,2	y,2

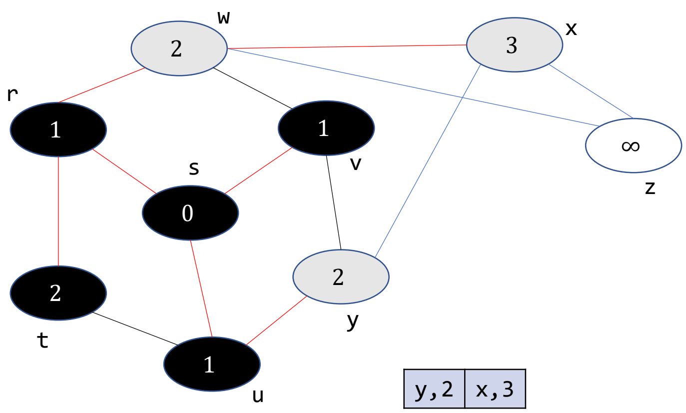
BFS



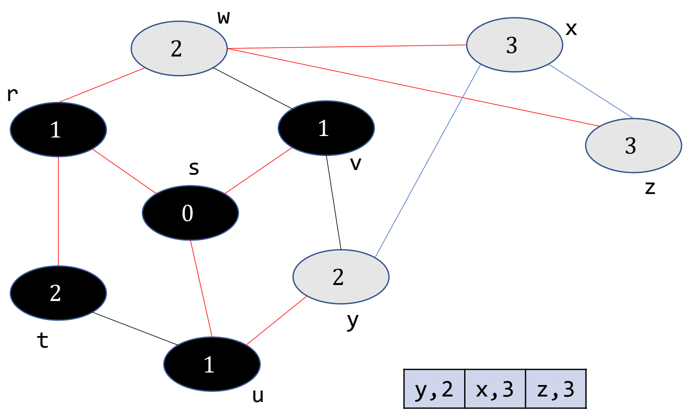
BFS



BFS

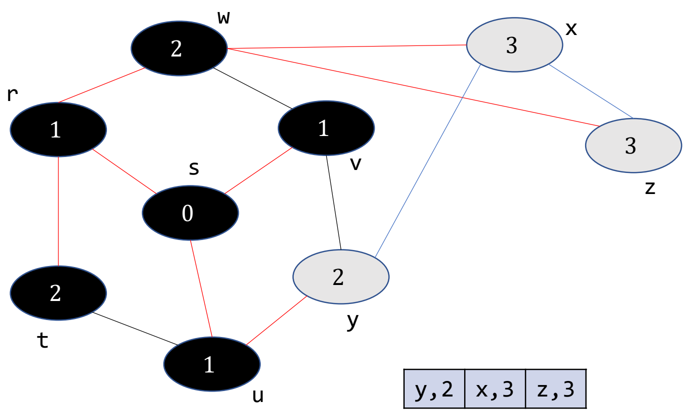


BFS

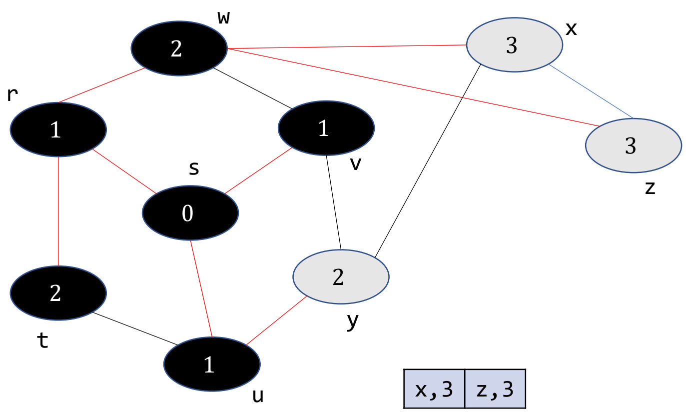


y,2	x,3	z,3

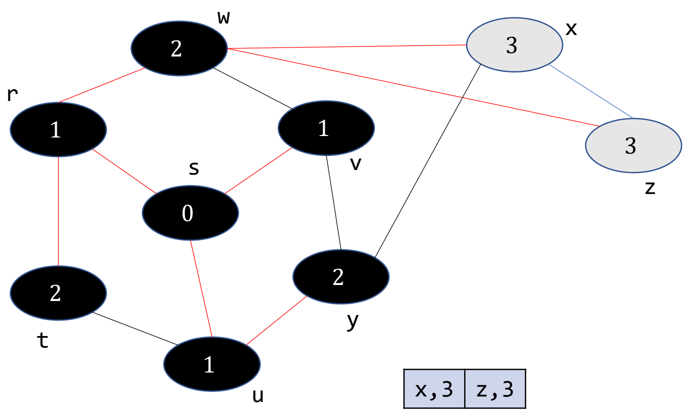
BFS



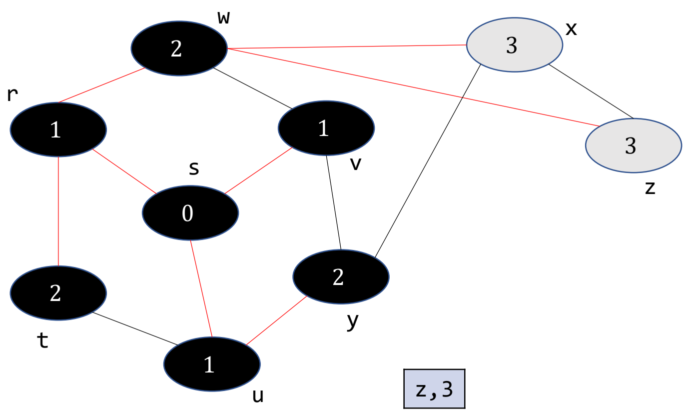
BFS



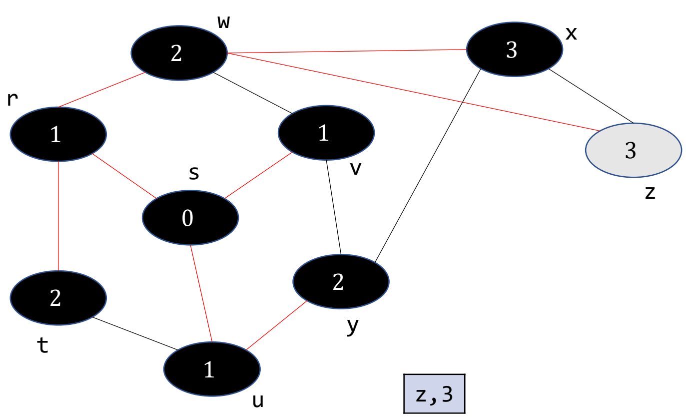
BFS



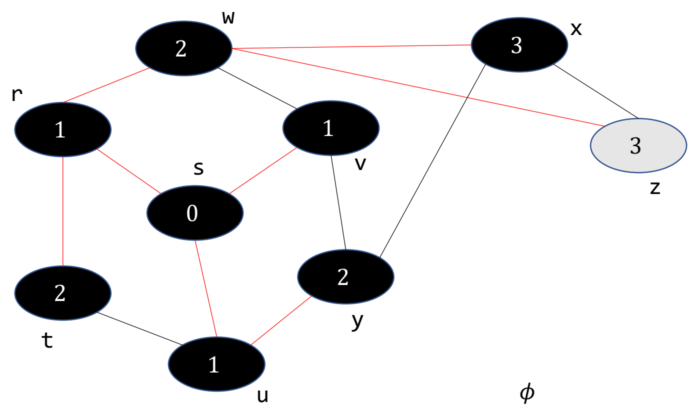
BFS



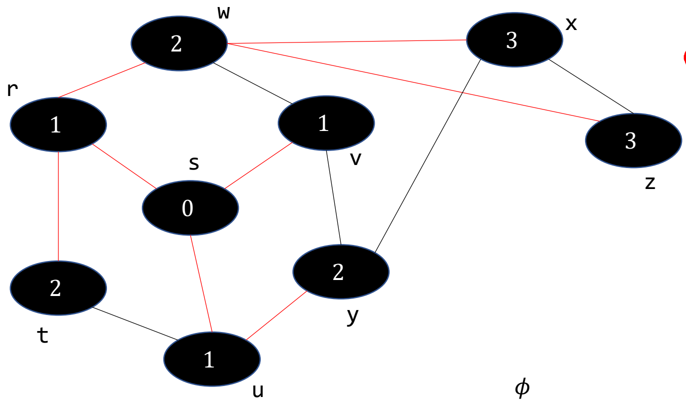
BFS

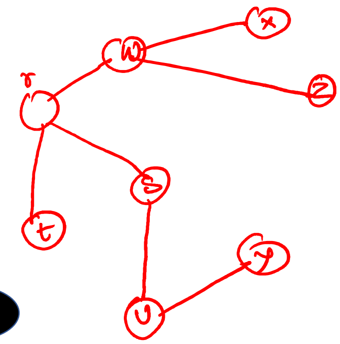


BFS



BFS





BFS

```
BFS(G, s)

// G is a graph (V, E)
// s is the source vertex
// each vertex contains three
fields, color, d, π

// Output: shortest distance to
every   vertex v reachable from u
in v.d

// Output: breath-first tree
rooted at u, every vertex v
contains the predecessor node in
field π on the shortest path from
u to v
```

```
BFS(G, s)

for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL

s.color = GRAY
s.d = 0
s.π = NIL
Q = ϕ
ENQUEUE(Q, s)
while Q ≠ ϕ
    u = DEQUEUE(Q)
    for each vertex v in G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            ENQUEUE(Q, v)
    u.color = BLACK
```

The color field in vertex stores the color of a node (we can use an integer to represent a color). Field π contains a reference to the predecessor on the shortest path. Field d stores the shortest distance.

BFS

- Time complexity

BFS

- Time complexity
  - After the initialization, BFS never whitens a vertex
  - At line-18, it only enqueues a white vertex
  - It marks a vertex as gray before enqueuing
  - Therefore, the dequeue operation at line-12 can take place at most |V| times
  - After dequeuing a vertex v, the for loop at line-13 traverse all outgoing edges from v, i.e., out_degree(v) times
    - Therefore, the for loop runs at most the sum of out degrees of all vertices, which is |E| for directed graph and 2*|E| for undirected graph
    - The algorithm is doing a constant number of operations inside the loop
  - Thus, the time complexity is O(|V| + |E|)

BFS

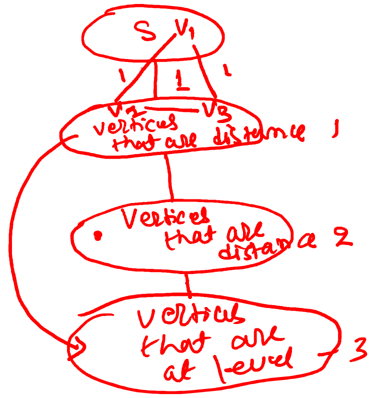
- Let's BFS is performed on graph G = (V, E) for the source vertex s
- Why is a BFS tree also a spanning tree that contains all the vertices reachable from s?
  - Let's say BFS tree is graph G1 = (V1, E1), where V1 are the vertices reachable from s

BFS

- Let's BFS is performed on graph G = (V, E) for the source vertex s
- Why is a BFS tree also a spanning tree that contains all the vertices reachable from s?
  - Let's say BFS tree is graph G1 = (V1, E1), where V1 are the vertices reachable from s
    - Each vertex in V1 has exactly one predecessor except s
    - Therefore, the number of edges is |V1|-1
    - The way we are constructing G1 is that every vertex in G1 has a path to s
    - Therefore, G1 is a connected graph
    - G1 is a tree because it's a connected graph with |V1| vertices and |V1|-1 edges
    - G1 it's also a spanning tree because it contains all vertices

BFS

- Why does BFS(G, s) always give a shortest path from s?

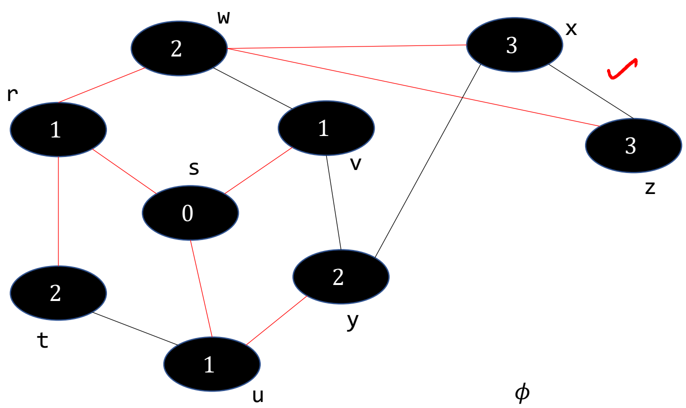


BFS

- Why does BFS(G, s) always give a shortest path from s?
  - In the first step, we first discover all the vertices are at a distance of 1
  - In the second step, we discover all vertices that are at a distance 2
    - Is it possible that we may discover a vertex with a distance < 2 during this step
      - No, because if that is the case, we may have discovered that vertex during the previous step
  - In the third step, we discover all vertices that are at a distance of 3
    - Is it possible that we may discover a vertex with a distance < 3 during this step
      - No, because if that is the case, we may have discovered that vertex during the previous steps
  - Intuitively, during i^{th} step, we can't discover a vertex that is at a distance <i from the source vertex; therefore, BFS gives a shortest path to each vertex

BFS

Tree edges are shown in red
Non-tree edges are shown in black



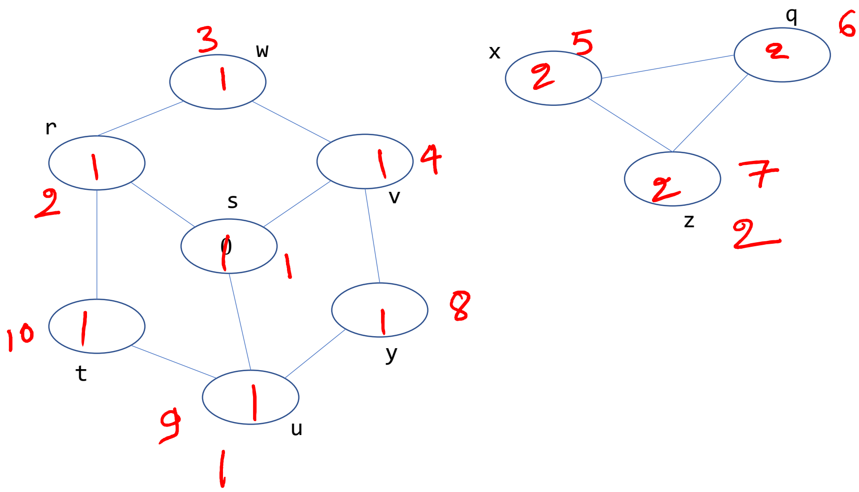
BFS

- If (u, v) is an edge that is not in the BFS tree, then the level numbers of u and v differ by at most 1
  - In other words, non-tree edges are between the vertices of the same level or adjacent levels

The endpoints of a tree edge always belong to two adjacent levels. The endpoints of a non-tree edge are either at the adjacent levels or at the same level. If the levels of a non-tree edge (u, v) are (l1, l2), where l2 > l1, then l1 and l2 can differ by at most 1. This is because if we can reach u in l1 steps, we can reach v in l1+1 steps, and therefore the distance of v can’t be more than l1+1.

Finding all connected components

Finding connected components



Finding connected components

```
ConnectedComponents(G)

// G is a graph (V, E)
// each vertex contains four
fields, color, d, π, comp_id

// Output: connected components id
in comp_id
// The connected components are
numbered from 1 to count, where
count is the number of connected
components
// the component number is the
component id
```

```
1. ConnectedComponents(G)
2. // G is a graph (V, E)
3. // Output: set the component_id in
   each vertex to its component number

4. for each vertex u ∈ G.V
5.     u.component_id = 0

6. component_id = 0
7. for each vertex u ∈ G.V
8.    if u.color == WHITE
9.      u.comp_id = component_id + 1
10.     BFS(G, u)
```

We can add an additional field to a vertex called component id (comp_id) to find all connected components. The BFS algorithm identifies all vertices of a component connected to a given vertex. We can store the component id in the vertex when it is discovered, as shown in line-18 on the next slide. We call BFS for each vertex in a loop if its component id hasn't been identified yet, as shown in lines-7,8.

BFS

```
BFS(G, s)

// G is a graph (V, E)
// s is the source vertex
// each vertex contains three
fields, color, d, π

// Output: shortest distance to
every   vertex v reachable from u
in v.d

// Output: breath-first tree
rooted at u, every vertex v
contains the predecessor node in
field π on the shortest path from
u to v
```

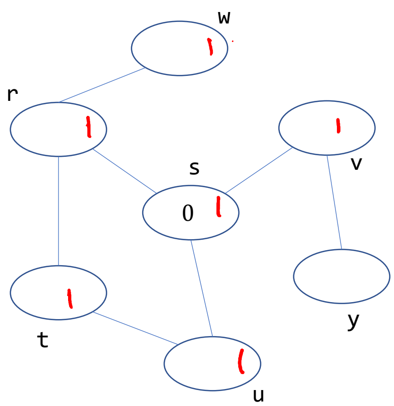
```
BFS(G, s)

for each vertex u ∈ G.V – {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL

s.color = GRAY
s.d = 0
s.π = NIL
Q = ϕ
ENQUEUE(Q, s)
while Q ≠ ϕ
    u = DEQUEUE(Q)
    for each vertex v in G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            v.comp_id = u.comp_id
            ENQUEUE(Q, v)
    u.color = BLACK
```

Finding cycles

Finding cycle



$\boxed{v}$

Notice that BFS generates a BFS spanning tree that connects all the vertices reachable via a given node. If, during the BFS algorithm, we identify a non-tree edge, it means that the graph has at least one cycle because the number of edges is more than what is needed for a spanning tree. If, during the BFS algorithm, we reach a vertex v from u and v has already been discovered, then it’s a tree edge if v is not the predecessor of u. The corresponding logic is shown in lines-16,17 on the next slide. Another quick way to check if the graph has a cycle is to find the number of edges. If the number of edges is more than or equal to the number of vertices, then it’s definitely not a tree and thus has cycle(s). However, if the number of edges is less than the number of vertices, then we need to check if the graph has a non-tree edge because the graph may not be connected.

BFS

```
BFS_Cycle(G, s)

// G is a graph (V, E)
// s is the source vertex
// each vertex contains three
fields, color, d, π

// Output: return 1 if the part of
graph reachable via s has a cycle;
otherwise, return 0
```

```
BFS_cycle(G, s)

for each vertex u ∈ G.V – {s}
    u.color = WHITE
    u.π = NIL

s.color = GRAY
s.π = NIL
Q = ϕ
ENQUEUE(Q, s)
while Q ≠ ϕ
    u = DEQUEUE(Q)
    for each vertex v in G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.π = u
            ENQUEUE(Q, v)
        else if u.π != v
            return 1
    u.color = BLACK
return 0
```