Data Structures and Algorithms

(CS210A)

Semester I – **2014-15**

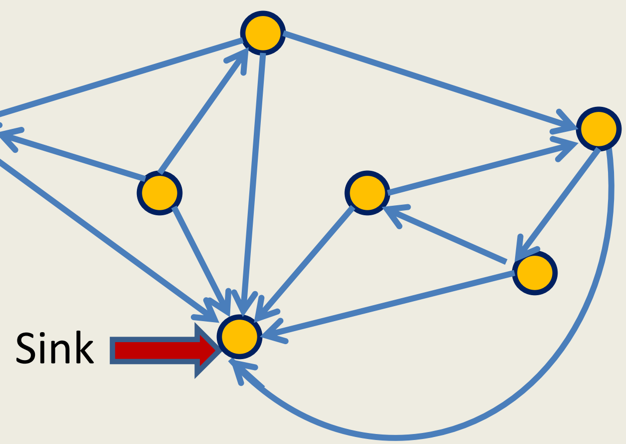
Lecture 21

- **Finding a sink in a directed graph**
- **Graph Traversal**
  - **Breadth First Search Traversal and its simple applications**

An interesting problem
(Finding a sink)

Definition: A vertex x in a given directed graph is said to be a **sink** if

- There is no edge emanating from (leaving) x
- Every other vertex has an edge into x.



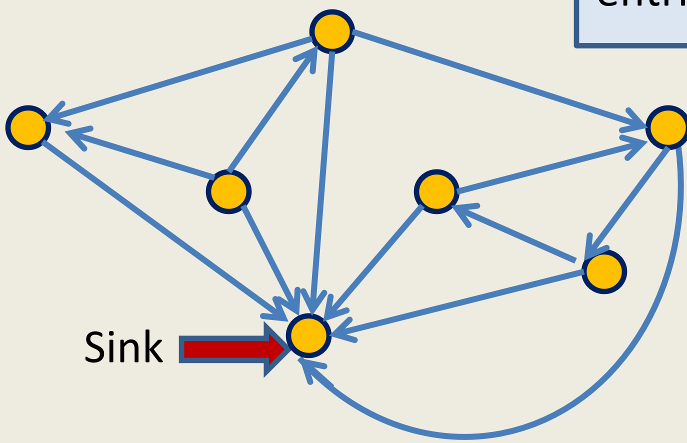
How many sinks can there be in G ?

At most **1**.

An interesting problem
(Finding a sink)

Problem: Given a directed graph $G=(V,E)$ in an **adjacency matrix** representation, design an $O(n)$ time algorithm to determine if there is any **sink** in G.

We are allowed to look into only $O(n)$ entries of the **Adjacency matrix M**.



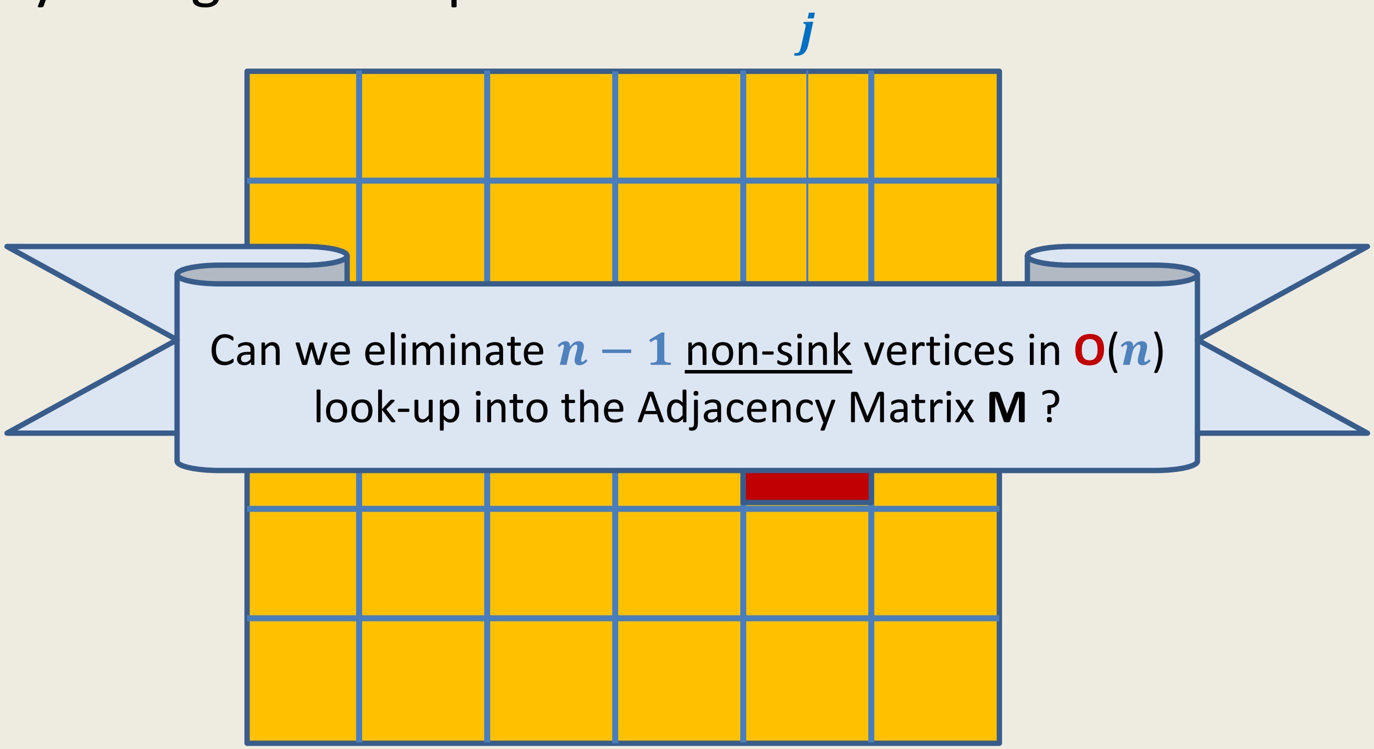
Question: Can we verify efficiently whether any given vertex i is a sink ?

Answer: Yes, in $O(n)$ time only ☺

Look at ith **row** and ith **column** of **M**.

Key idea

Let us try a single look-up in **M**.



Can we eliminate $n-1$ non-sink vertices in **O**(n) look-up into the Adjacency Matrix **M** ?

If **M**[i, j] = **0**, then j can not be sink

If **M**[i, j] = **1** , then i can not be sink

Algorithm to find a sink in a graph

Key ideas:

- Looking at a single entry in **M** allows us to discard one vertex from being a sink.
- It takes **O**(n) time to verify if a vertex i is a sink.

Find-Sink(**M**)   // **M** is the adjacency matrix of the given directed graph.

```
s← 0;
For(i=1 to n − 1)
{
        If (M[s,i] =  ? )  ....?...;
}
Verify if s is a sink and output accordingly.
```

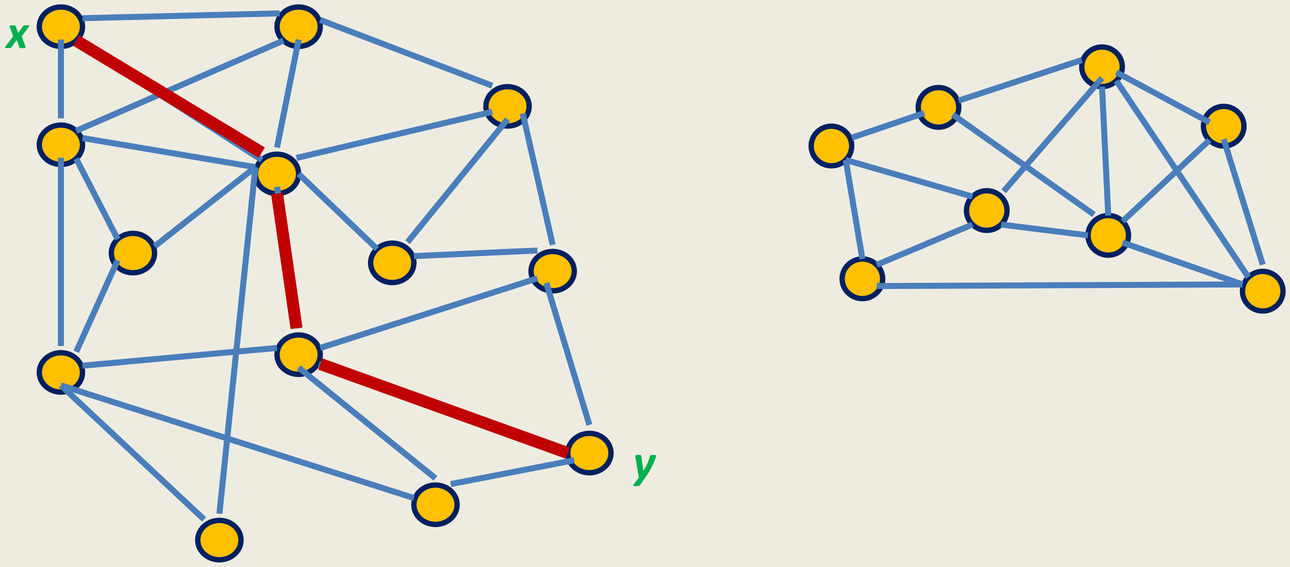
(Fill in the details of this pseudo code as an exercise.)

What is **Graph traversal** ?

Graph traversal

Definition:

A vertex ***y*** is said to be reachable from ***x*** if there is a **path** from ***x*** to ***y***.



Graph traversal from vertex ***x***: Starting from a given vertex ***x***, the aim is to visit all vertices which are reachable from ***x***.

Non-triviality of graph traversal

- **Avoiding loop**:

  How to avoid visiting a vertex multiple times ?

  (keeping track of vertices already visited)

- **Finite number of steps** :

  The traversal **must stop** in finite number of steps.

- **Completeness** :

  We must visit **all** vertices reachable from the start vertex ***x***.

Spend some time on these points and see how can you design such a traversal algorithm

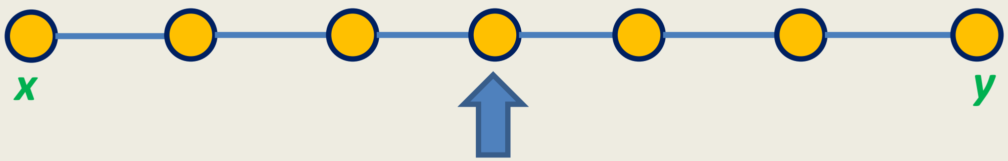
Breadth First Search traversal

We shall introduce this traversal technique through an interesting problem.

computing distances from a vertex.

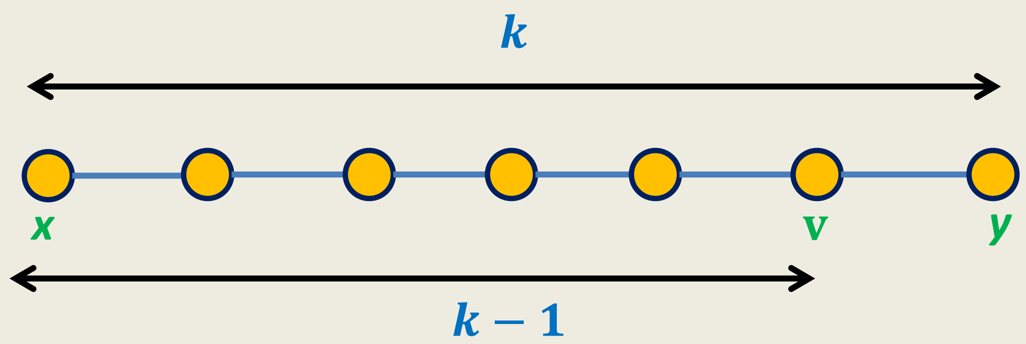
Notations and Observations

Length of a path: the **number of edges** on the path.



A path of length 6 between ***x*** and ***y***

Notations and Observations



Observation:

If <x, …,v,y> is a path of length k from x to y,

then what is the length of the path <x, …,v> ?

Answer: $k-1$

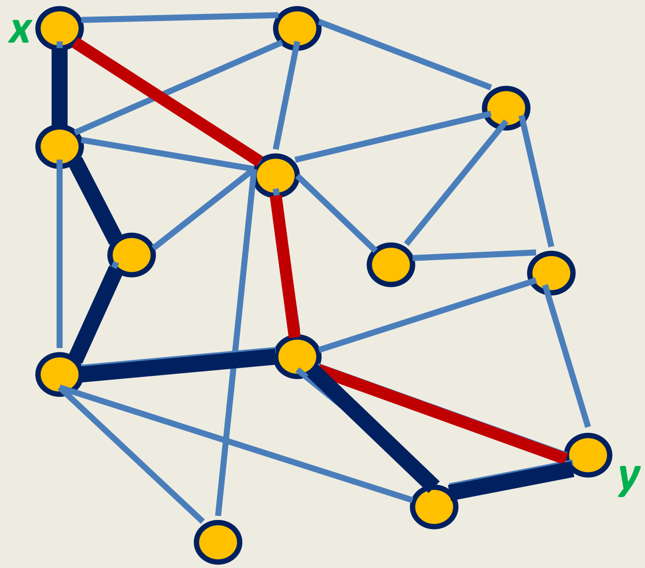
Question: What can be the maximum length of any path in a graph ?

Answer: $n-1$

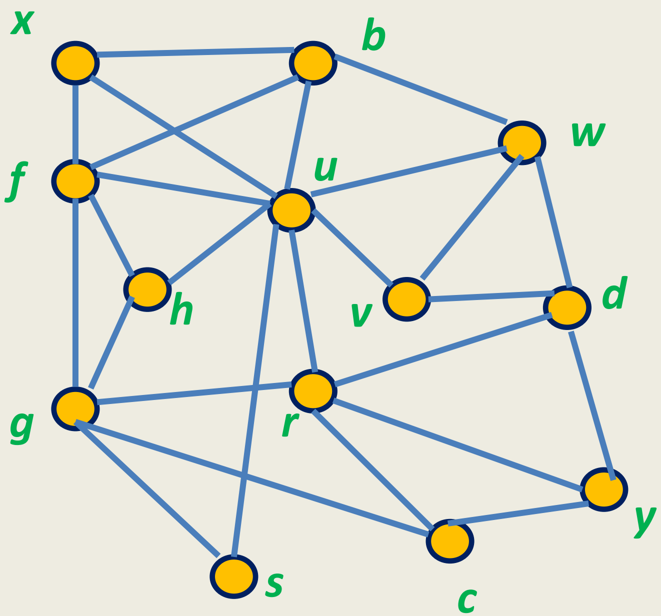
Notations and Observations

Shortest Path from *x* to *y*: A path from ***x*** to ***y*** of **least** length

Distance from ***x*** to ***y***: the **length** of the shortest path from ***x*** to ***y*.**



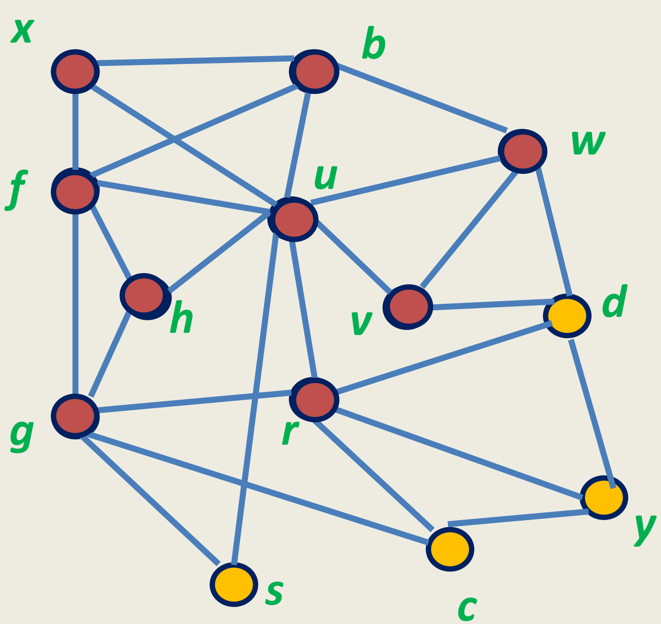
Shortest Paths
in Undirected Graphs



Problem:

How to compute distance to all vertices **reachable** from ***x*** in a given undirected graph ?

Shortest Paths
in Undirected Graphs



V_0 : Vertices at distance **0** from ***x***: {*x*}

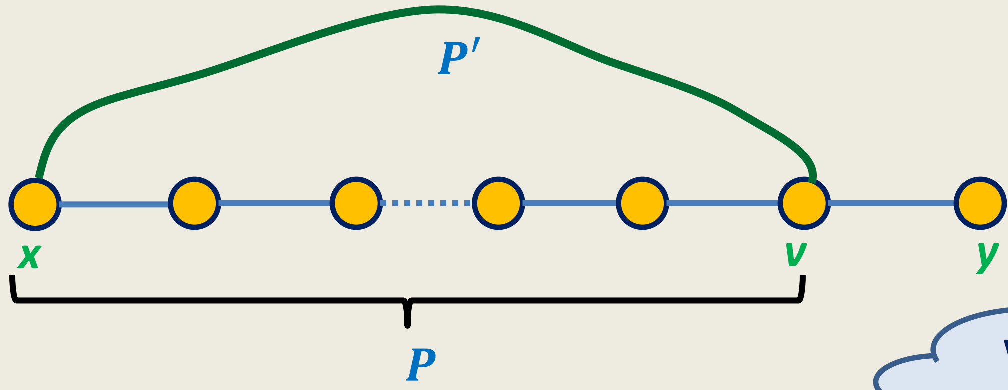
V_1 : Vertices at distance **1** from ***x***: {*f*,*u*,*b*}

V_2 : Vertices at distance **2** from ***x***: {*g*,*h*,*s*,*r*,*v*,*w*}

Why ?

While reporting V_2, you have <u>(sub)consciously</u> used an **important property** of shortest paths.
Can you state this property ?

An important property of shortest paths



A shortest path between ***x*** and ***y***

What can you say about P ?

Observation:

If <x, …,v,y> is a shortest path from x to y, then <x, …,v> is also a shortest path.

Proof:

Suppose P = <x, …,v> is <u>not</u> a shortest path between x and v.

Then let P' be a shortest path between x and v.

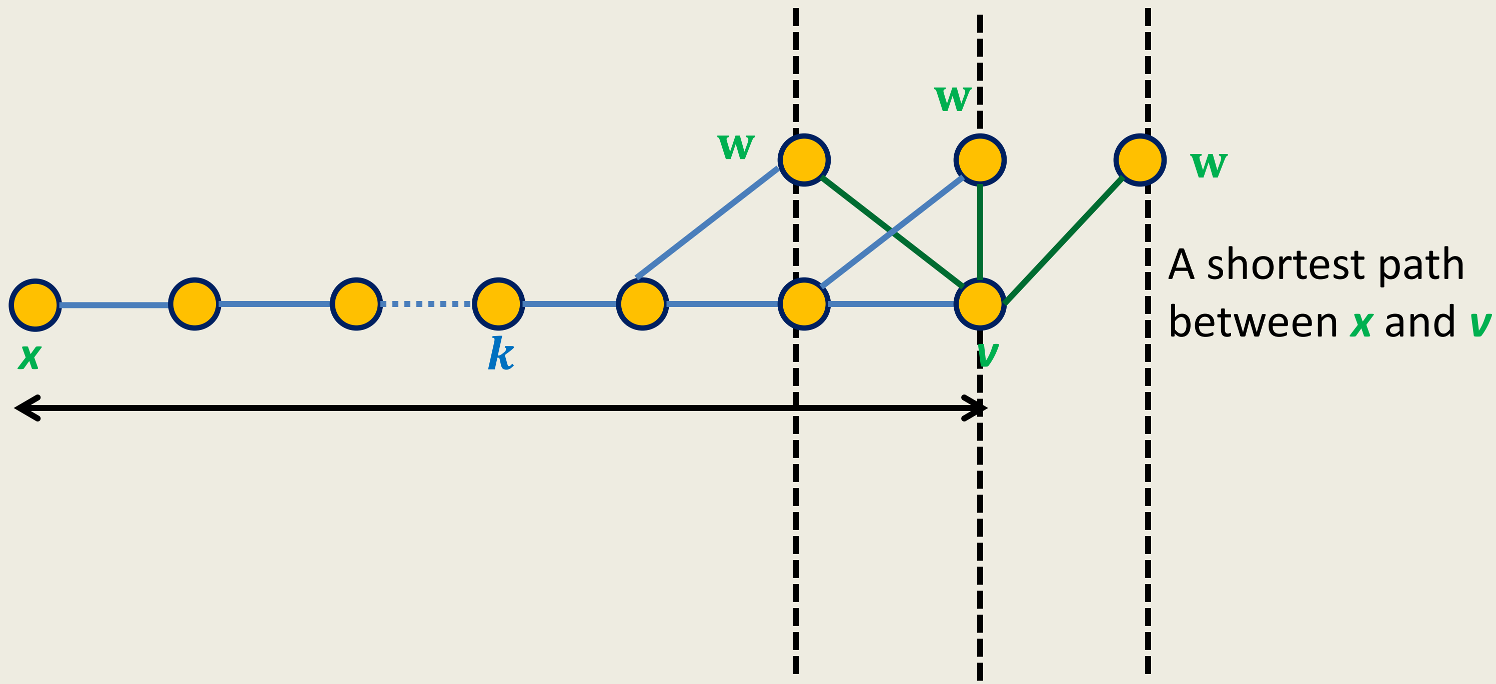
Length(P') < Length(P).

Question: What happens if we concatenate P' with edge (v, y) ?

Answer: a path between x and y <u>shorter</u> than the <u>shortest-path</u> <x, …,v,y> .

➔ Contradiction.

An important question



Question:

Let (v,w) be an edge. If **Distance**(x,v) is k,

then what can be **Distance**(x,w) ?

Answer: an element from the set $\{k-1, k, k+1\}$ only.

Relationship among vertices at different distances from *x*

V_0 : Vertices at distance **0** from ***x*** = {***x***}

V_1 : Vertices at distance **1** from ***x*** =

Neighbors of V_0

V_2 : Vertices at distance **2** from ***x*** =

Those Neighbors of V_1 which **do not** belong to V_0 or V_1

.

.

.

V_{i+1} : Vertices at distance **i+1** from ***x*** =

Those Neighbors of V_i which **do not** belong to $V_i - 1$ or V_i

How to distinguish the neighbors of V_i which belong to V_{i+1} from those which belong to V_j, $j \leq i$?

How can we **compute** V_{i+1} ?

Key idea: compute V_i's in increasing order of i.

Initialize **Distance**[**v**] ← ∞ of each vertex **v** in the graph.

Initialize **Distance**[**x**] ← 0.

- First compute V_0 .
- Then compute V_1 .
- …
- Once we have computed V_i , for every neighbor **v** of a vertex in V_i,

If **v** is in V_j for some $j \in \{i, i-1\}$, then **Distance**[**v**] = a number ≤ i

If **v** is in V_{i+1}, **Distance**[**v**] = ∞

We can thus distinguish the neighbors of V_i which belong to V_{i+1} from those which belong to V_j.

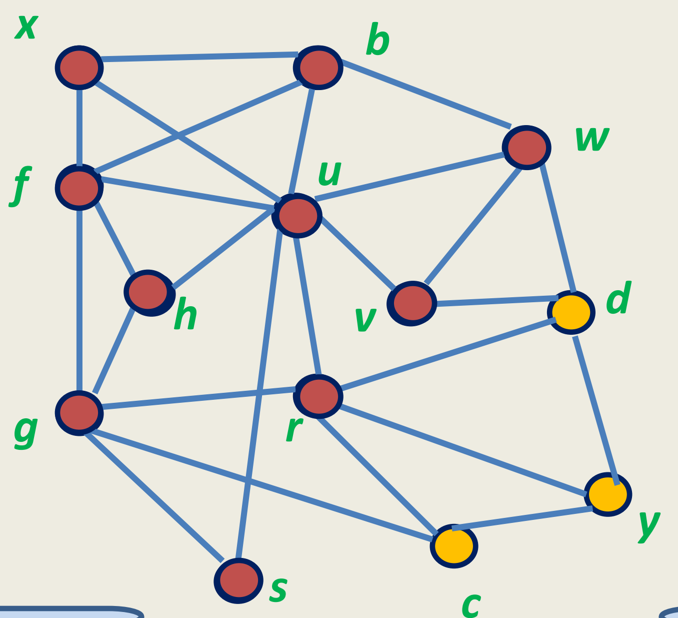
A neat algorithm for computing distances from *x*

we need an algorithm which traverses/visits the vertices in **non-decreasing** order of **distances** from ***x***

This traversal algorithm is called **BFS** (breadth first search) traversal

Using a queue for traversing vertices in non-decreasing order of distances

Compute distance of vertices from *x*:



Queue	
x	V_0
f u b	V_1
u b g h	
b g h s r v w	
g h s r v w	V_2

Remove ***x*** and for each neighbor of ***x*** that

unvisited, mark it visited and p

Remove ***b*** and for each neighbor of ***b*** that was unvisited, mark it visited and put it into queue.

BFS traversal from a vertex

```
BFS(G, x)
    CreateEmptyQueue(Q);
    Distance(x) ← 0;
    Enqueue(x,Q);
    While( Not IsEmptyQueue(Q) )
    {       v ←Dequeue(Q);
            For each neighbor w of v
            {
                    if (Distance(w) = ∞)
                    {   Distance(w) ← Distance(v) +1 ;
                        Enqueue(w, Q); ;
                    }
            }
    }
```

Running time of BFS traversal

```
BFS(G, x)
   CreateEmptyQueue(Q);
   Distance(x) ← 0;
   Enqueue(x,Q);
   While(  Not IsEmptyQueue(Q)  )
   {        v ←Dequeue(Q);
            For each neighbor w of v
            {
                     if (Distance(w) = ∞)
                     {   Distance(w) ←  Distance(v) +1  ;
                         Enqueue(w, Q);  ;
                     }
            }
   }
```

$\Big\}$ O(deg(v))

A vertex can enter queue **at most** once. Prove this claim first.

Running time of BFS(x) = no. of edges in **the connected component** of **x.**

Correctness of BFS traversal

Question: What do we mean by correctness of **BFS** traversal from vertex **x** ?

Answer:

- **All vertices** reachable from **x** **get visited**.
- Vertices get visited in the **non-decreasing order of their distances** from **x**.
- At the end of the algorithm, **Distance**(**v**) **is the distance** of vertex **v** from **x**.

Think about the **proofs** of these statements. We shall discuss them in the next class.

A useful advice

An effective way to master the technique of **proving correctness of an algorithm** is to

Do each home **work exercise**
(about proof of correctness)
that is asked in the class before attending the next class.

There is no escape. There will be question on proof of correctness in the end-sem exam. ☺