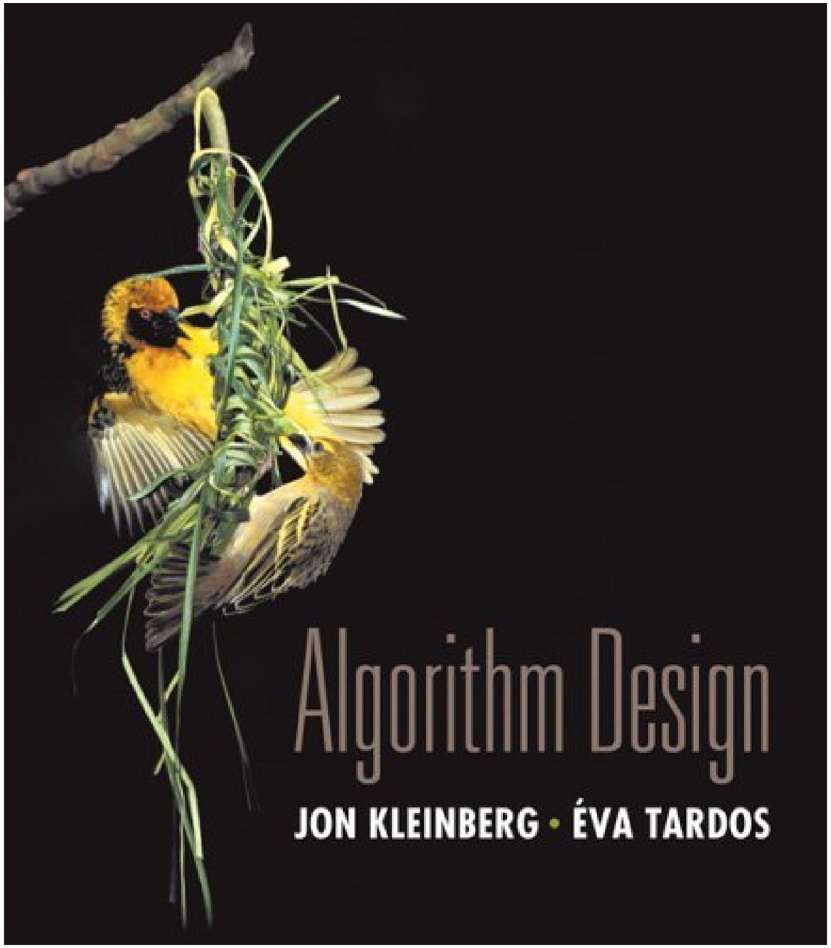
CS 381 – FALL 2019

Week 8.3, Friday, Oct 11

Homework 4 Due: October 14th @ 11:59PM (Gradescope)
Reminder: You must include a collaborator/resource (CR) statement for every problem.



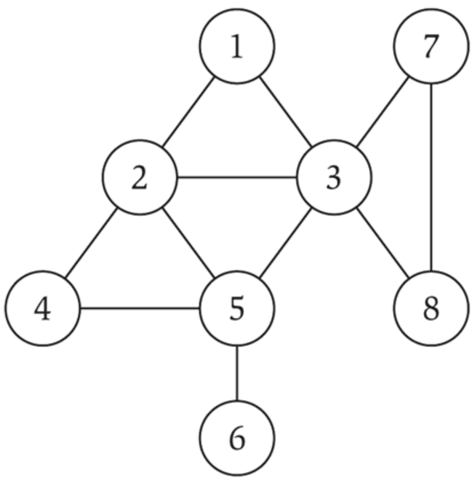
Graphs

3.1 Basic Definitions and Applications

Undirected Graphs

Undirected graph. G = (V, E)

- V = nodes.
- E = edges between pairs of nodes.
- Captures pairwise relationship between objects.
- Graph size parameters: n = |V|, m = |E|.



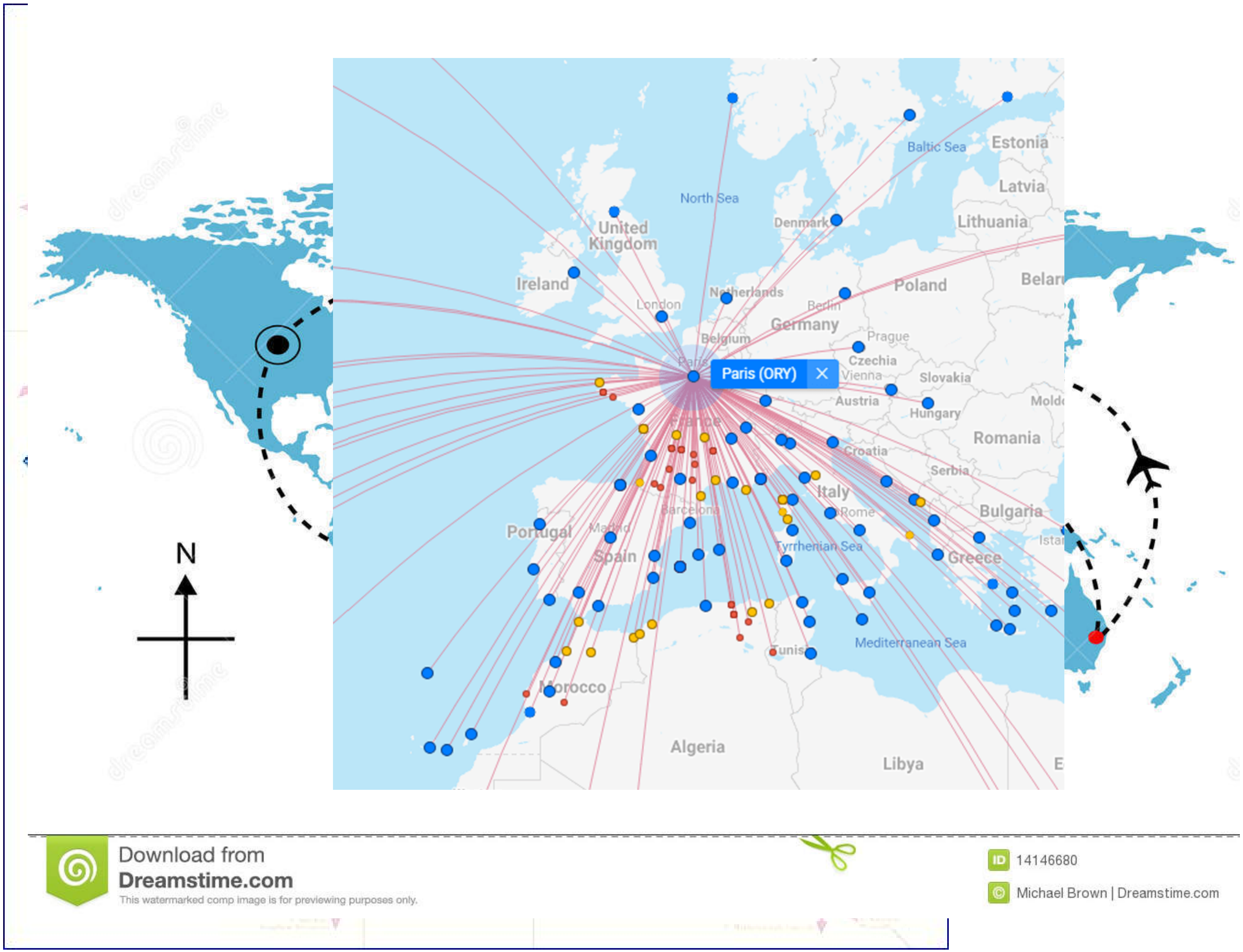
V = { 1, 2, 3, 4, 5, 6, 7, 8 }

E = { 1-2, 1-3, 2-3, 2-4, 2-5, 3-5, 3-7, 3-8, 4-5, 5-6 }

n = 8

m = 11

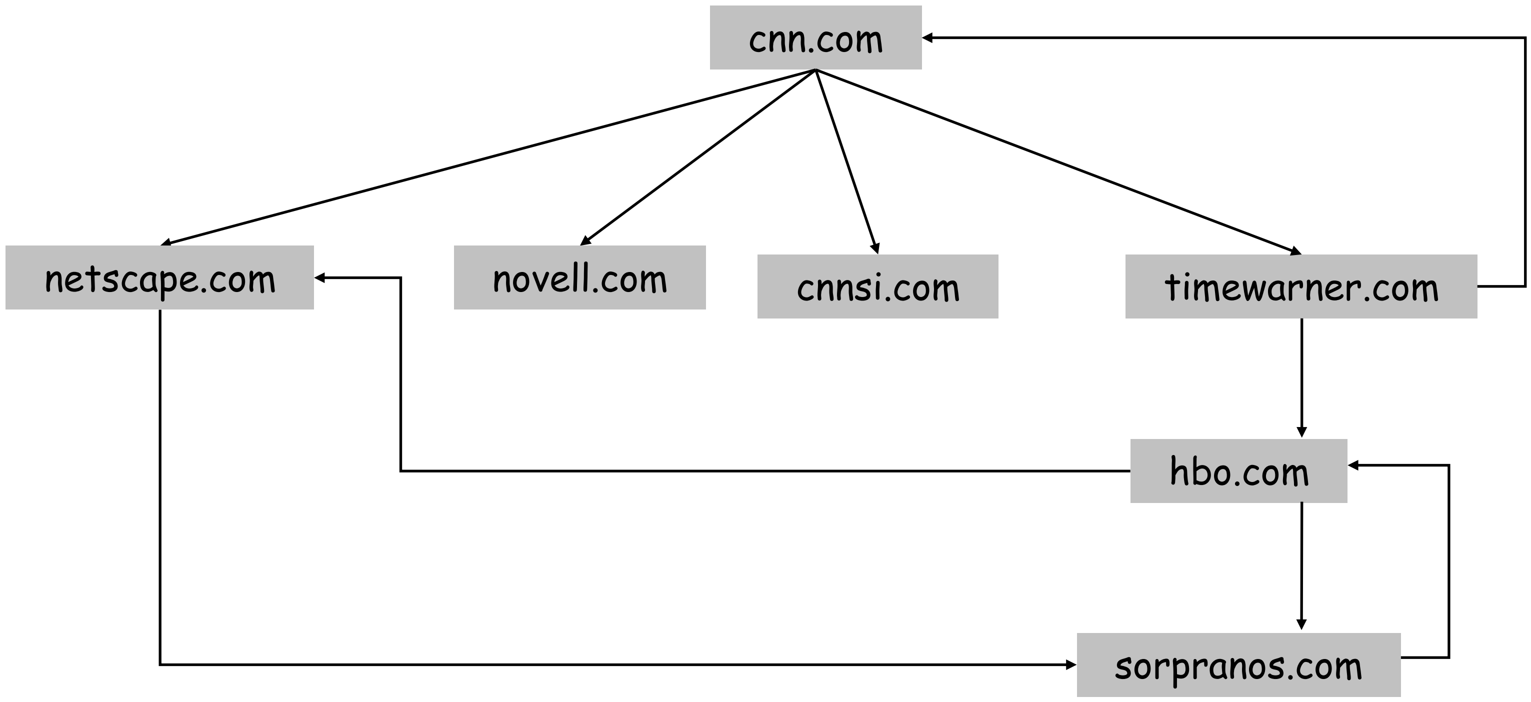
Transportation Networks (Planes, Trains, Highways etc…)



World Wide Web

Web graph.

- Node: web page.
- Edge: hyperlink from one page to another.



Social Network

9-11 Terrorist Network

Social network graph.

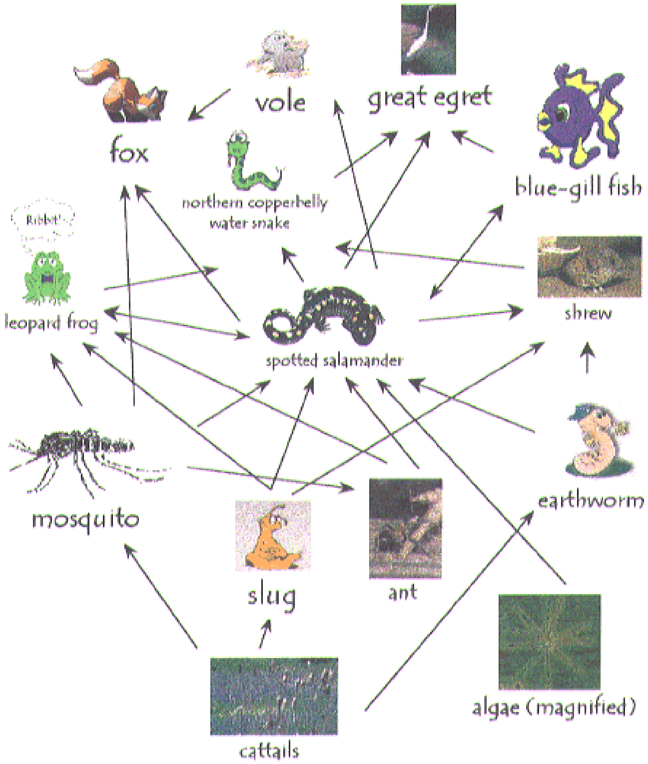
- Node: people.
- Edge: relationship between two people.

Reference: Valdis Krebs, http://www.firstmonday.org/issues/issue7_4/krebs

Ecological Food Web

Food web graph.

- Node = species.
- Edge = from prey to predator.

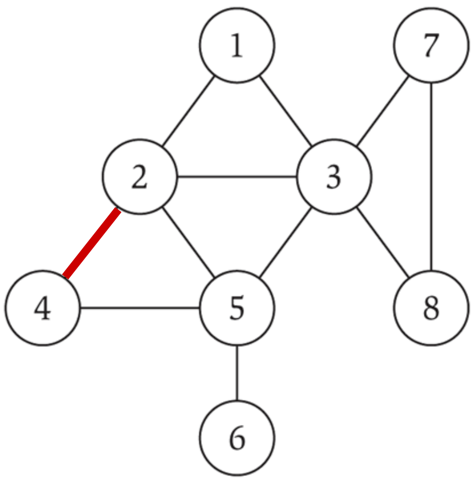


Reference: http://www.twingroves.district96.k12.il.us/Wetlands/Salamander/SalGraphics/salfoodweb.giff

Graph Representation: Adjacency Matrix

Adjacency matrix. n-by-n matrix with A_{uv} = 1 if (u, v) is an edge.

- Two representations of each edge.
- Space proportional to n^2.
- Checking if (u, v) is an edge takes $\Theta(1)$ time.
- Identifying all edges takes $\Theta(n^2)$ time.



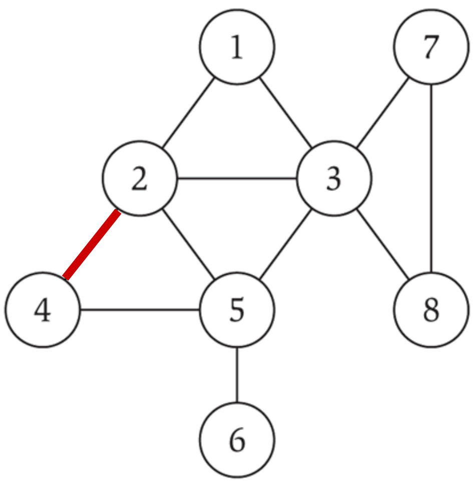
	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	1	1	0	0	0
3	1	1	0	0	1	0	1	1
4	0	1	0	0	1	0	0	0
5	0	1	1	1	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0

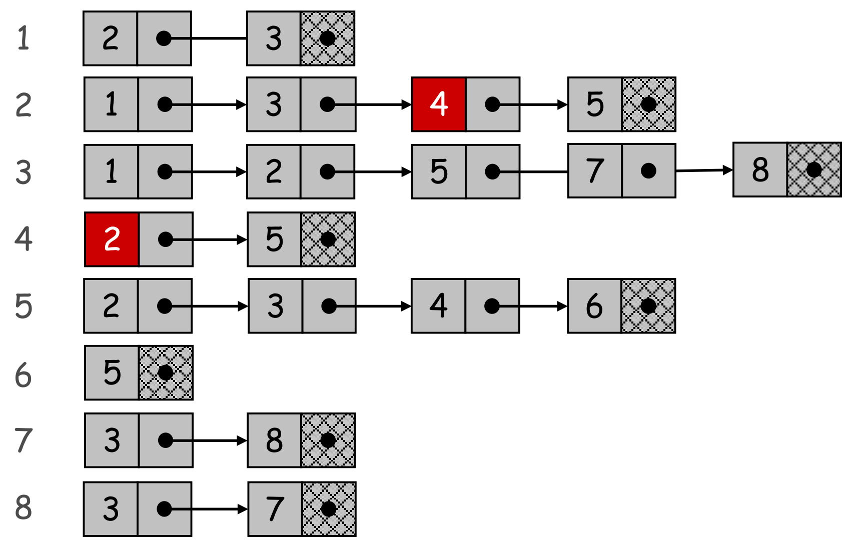
Graph Representation: Adjacency List

Adjacency list. Node indexed array of lists.

- Two representations of each edge.
- Space proportional to m + n.
- Checking if (u, v) is an edge takes O(deg(u)) time.
- Identifying all edges takes Θ(m + n) time.

degree = number of neighbors of u



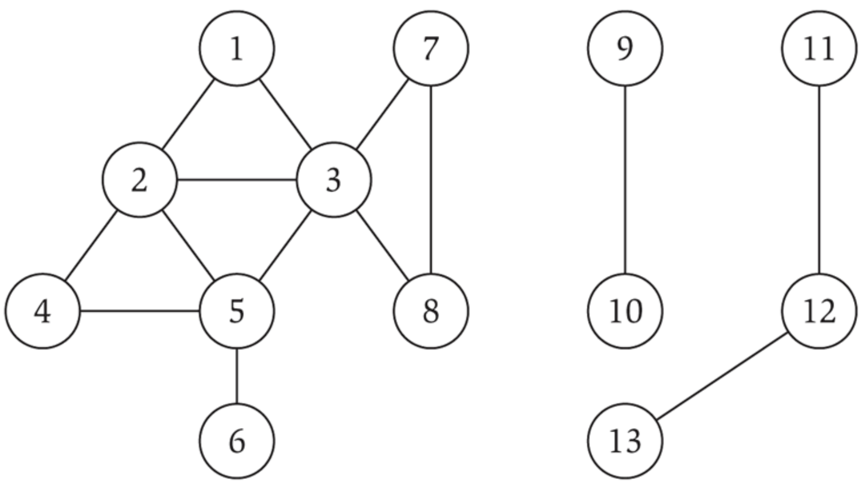


Paths and Connectivity

Def. A path in an undirected graph $G = (V, E)$ is a sequence P of nodes $v_1, v_2, \ldots, v_{k-1}, v_k$ with the property that each consecutive pair v_i, v_{i+1} is joined by an edge in E.

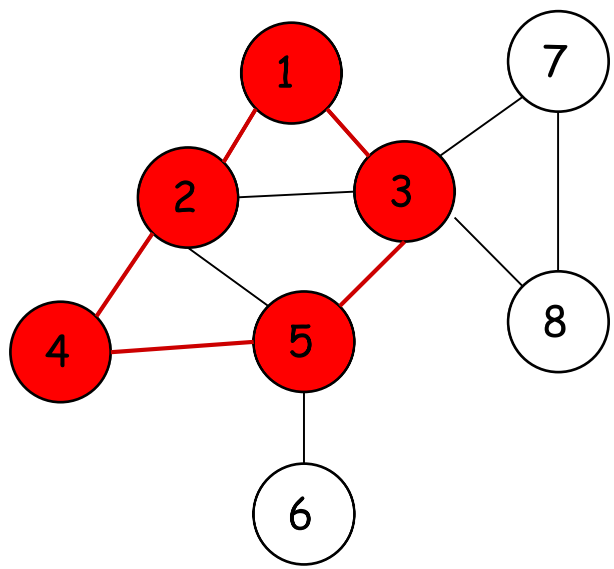
Def. A path is simple if all nodes are distinct (e.g., 1,2,5,6 below)

Def. An undirected graph is connected if for every pair of nodes u and v, there is a path between u and v.



Cycles

Def. A cycle is a path $v_1, v_2, \ldots, v_{k-1}, v_k$ in which $v_1 = v_k$, $k > 2$, and the first k-1 nodes are all distinct.



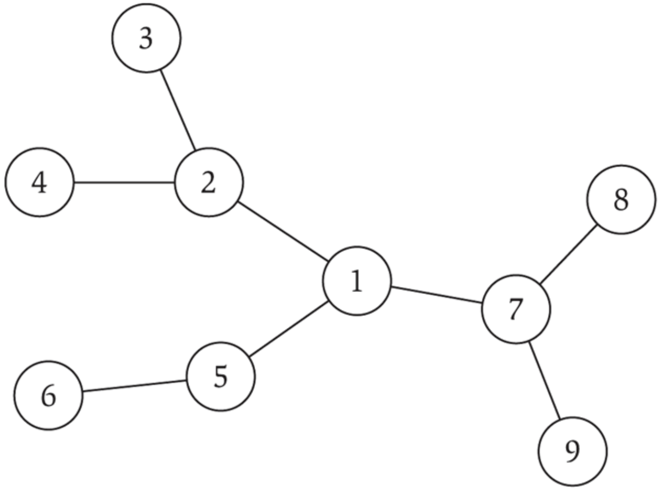
cycle C = 1-2-4-5-3-1

Trees

Def. An undirected graph is a **tree** if it is connected and does not contain a cycle.

Theorem. Let G be an undirected graph on n nodes. Any two of the following statements imply the third.

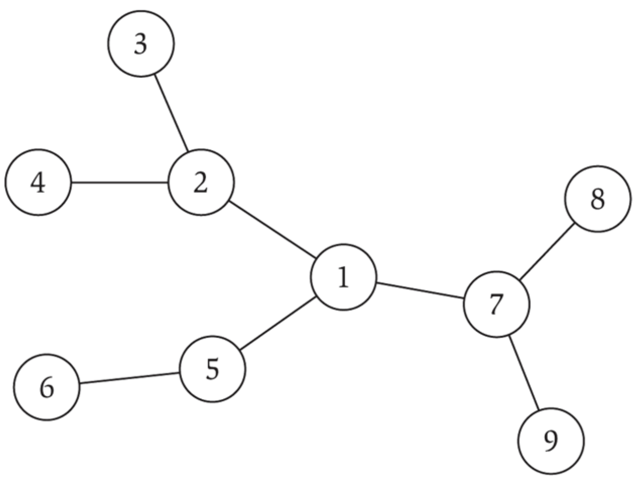
- G is connected.
- G does not contain a cycle.
- G has n-1 edges.



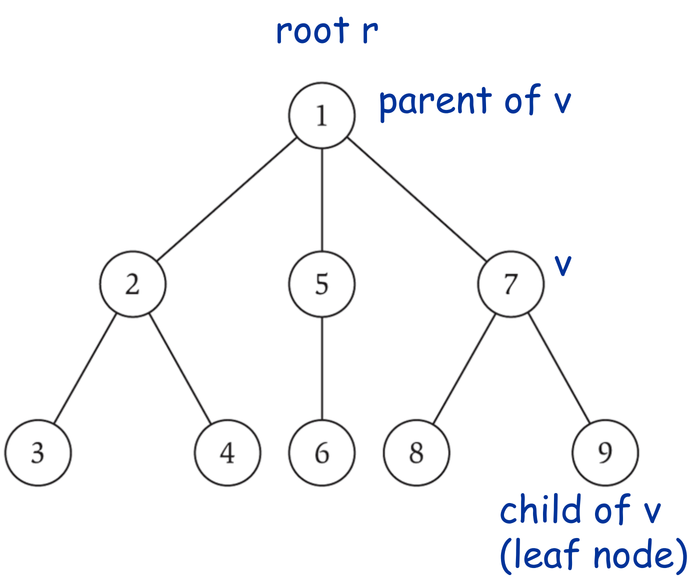
Rooted Trees

Rooted tree. Given a tree T, choose a root node r and orient each edge away from r.

Importance. Models hierarchical structure.



a tree

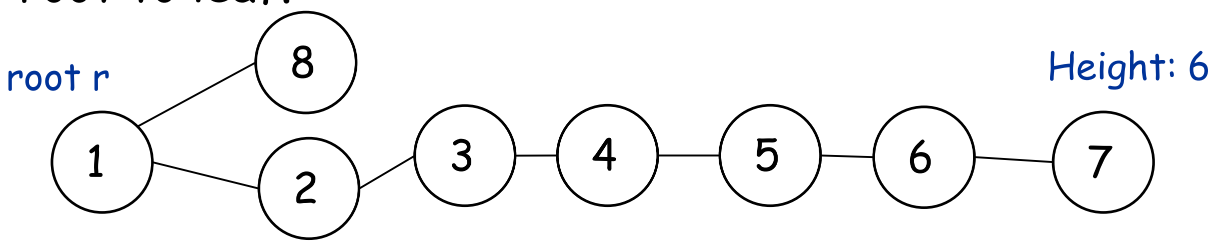


the same tree, rooted at 1

Binary Tree

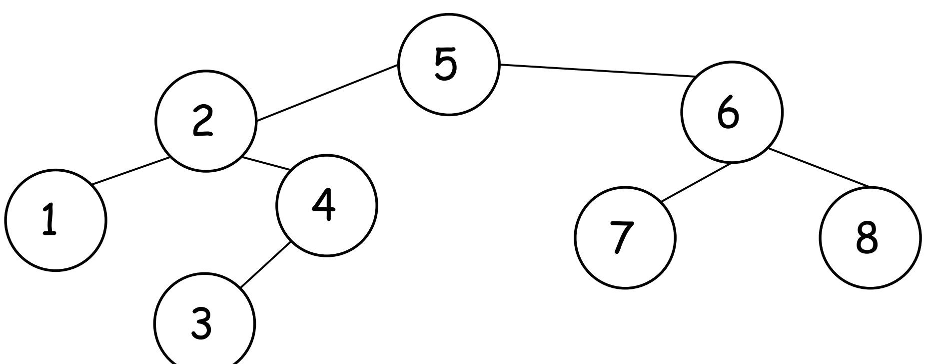
Def. A rooted tree in which each node has at most 2 children

Def. Height of a tree is the number of edges in the longest path from root to leaf.



Thm. Number of nodes in binary tree of height h is $n \le 2^{h+1} - 1 \ \ (= 2^0 + 2^1 + 2^2 + \cdots + 2^h)$.

Balanced Binary Tree. Height $h = O(\log n)$



iClicker Quiz

Let $G=(V,E)$ be an simple undirected graph. Which of the following claims are <u>false</u>?

A. If $|E| > n-1$ then G contains a cycle

B. If $|E|=n-1$ then G is a tree

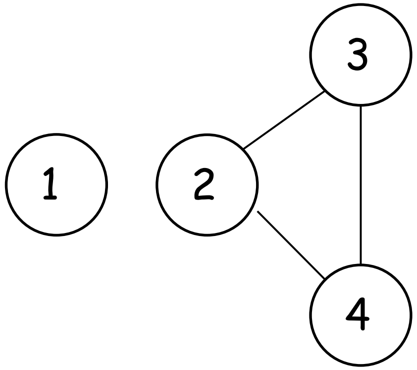
C. If $|E| = n(n+1)/2$ then G is connected

D. If G is represented as an adjacency matrix then we can test whether or not a particular edge (u,v) exists in time $O(1)$

E. We can store G using $O(m+n)$ space using adjacency list representation.

iClicker Quiz

Let G=(V,E) be an simple undirected graph. Which of the following claims are false?



A. If |E|> n-1 then G contains a cycle

B. If |E|=n-1 then G is a tree

C. If |E| = n(n+1)/2 then G is connected
(G=Kn is complete graph & contains all possible edges)

D. If G is represented as an adjacency matrix then we can test whether or not a particular edge (u,v) exists in time O(1)

E. We can store G using O(m+n) space using adjacency list representation.

3.2 Graph Traversal

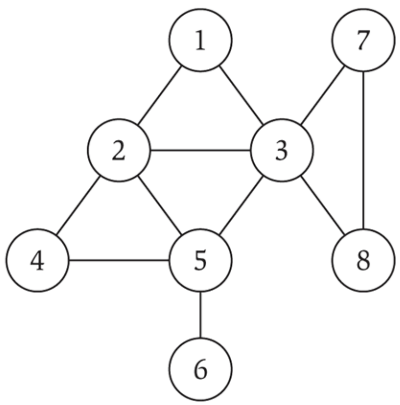
Connectivity

s-t connectivity problem. Given two node s and t, is there a path between s and t?

s-t shortest path problem. Given two node s and t, what is the length of the shortest path between s and t?

Applications.

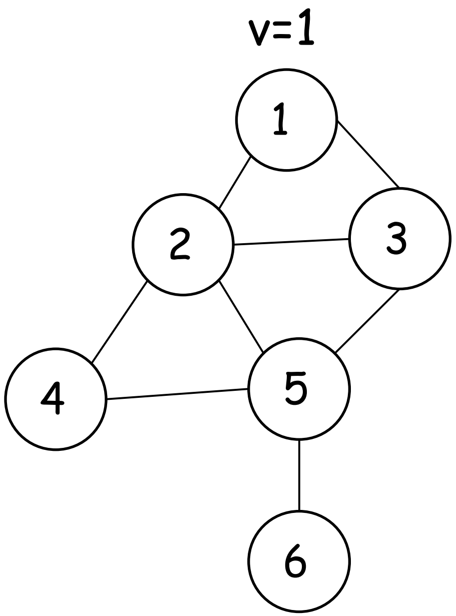
- Navigation (Google Maps).
- Maze traversal.
- Kevin Bacon number (or Erdős Number).
- Fewest number of hops in a communication network.



Breadth First Search: **BFS**(v, G=(V,E))

Input: Start node v, graph G (adjacency list) with n node V={1,…,n}
Output: Array Level[u] = distance from v to u

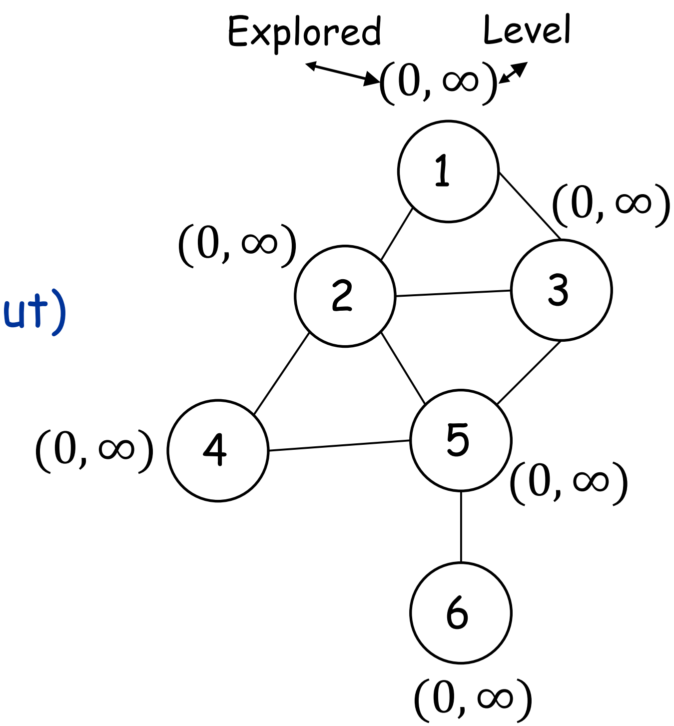
```
 For each node u in V
    Explored[u]=0
    Level[u] = ∞          // No v→u path found yet
 Q.Enqueue(v)             // Queue Q (First In, First Out)
 Level[v]=0
 while(Q is not empty)
   u = Q.Dequeue()
   if (Explored[u] = 0)
     Explored[u]=1
     foreach node w in u.AdjList
       if (Level[w] = ∞)
          Q.Enqueue(w)
          Level[w] = Level[u]+1
   end if
```



Breadth First Search: **BFS**(v, G=(V,E))

Input: Start node v, graph G (adjacency list) with n node V={1,…,n}
Output: Array Level[u] = distance from v to u

```
 For each node u in V
    Explored[u]=0
    Level[u] = ∞        // No v→u path found yet
Q.Enqueue(v)            // Queue Q (First In, First Out)
Level[v]=0
while(Q is not empty)
   u = Q.Dequeue()
   if (Explored[u] = 0)
      Explored[u]=1
      foreach node w in u.AdjList
         if (Level[w] = ∞)
            Q.Enqueue(w)
            Level[w] = Level[u]+1
   end if
```

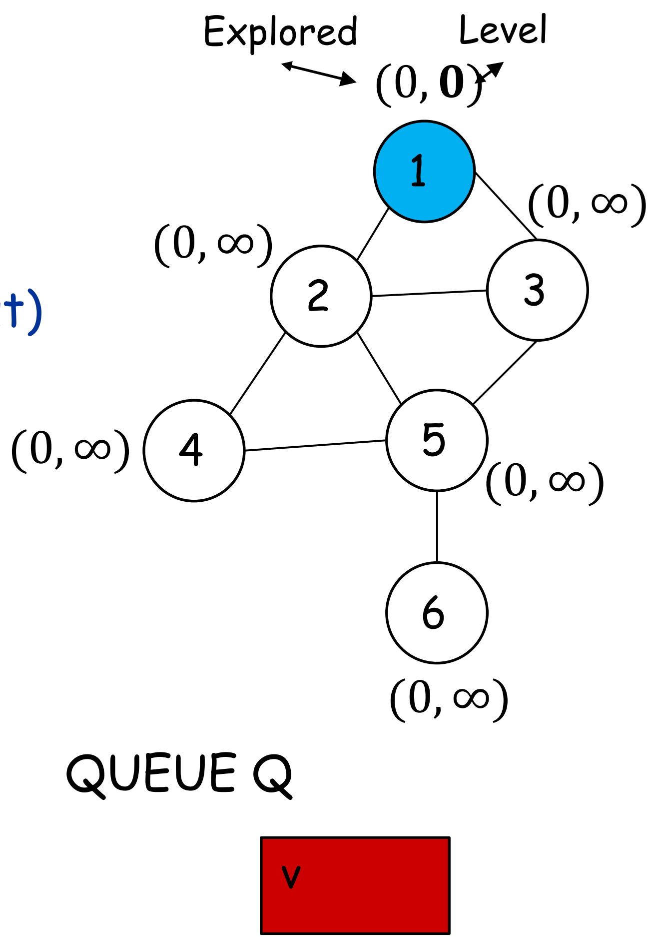


Breadth First Search: **BFS**(v, G=(V,E))

Input: Start node v, graph G (adjacency list) with n node V={1,…,n}

Output: Array Level[u] = distance from v to u

```
 For each node u in V
    Explored[u]=0
    Level[u] = ∞        // No v→u path found yet
 Q.Enqueue(v)           // Queue Q (First In, First Out)
 Level[v]=0
 while(Q is not empty)
   u = Q.Dequeue()
   if (Explored[u] = 0)
     Explored[u]=1
     foreach node w in u.AdjList
        if (Level[w] = ∞)
            Q.Enqueue(w)
            Level[w] = Level[u]+1
   end if
```

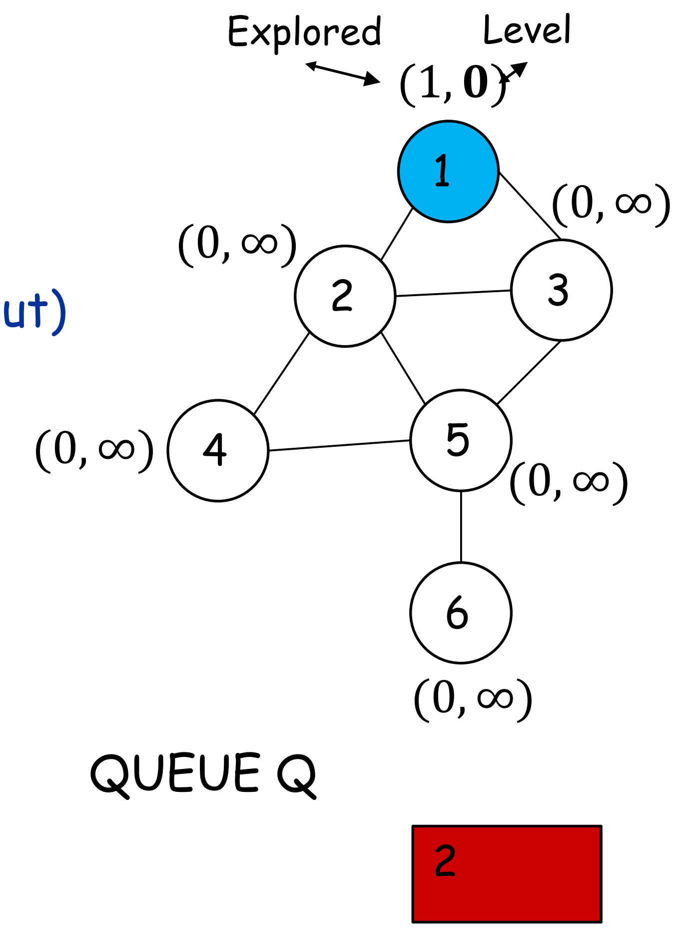


Breadth First Search: **BFS**(v, G=(V,E))

Input: Start node v, graph G (adjacency list) with n node V={1,…,n}

Output: Array Level[u] = distance from v to u

```
 For each node u in V
    Explored[u]=0
    Level[u] = ∞        // No v→u path found yet
 Q.Enqueue(v)           // Queue Q (First In, First Out)
 Level[v]=0
 while(Q is not empty)
   u = Q.Dequeue()
   if (Explored[u] = 0)
→    Explored[u]=1
→    foreach node w in u.AdjList
        if (Level[w] = ∞)
→          Q.Enqueue(w)
→          Level[w] = Level[u]+1
   end if
```

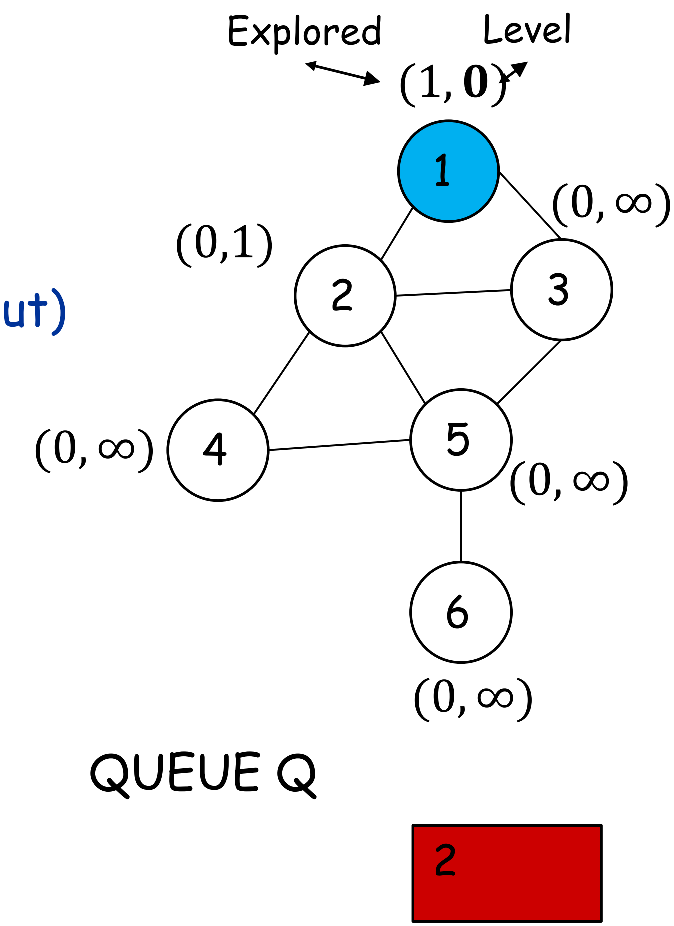


Breadth First Search: **BFS**(v, G=(V,E))

Input: Start node v, graph G (adjacency list) with n node V={1,…,n}

Output: Array Level[u] = distance from v to u

```
 For each node u in V
     Explored[u]=0
     Level[u] = ∞         // No v→u path found yet
 Q.Enqueue(v)             // Queue Q (First In, First Out)
 Level[v]=0
 while(Q is not empty)
    u = Q.Dequeue()
    if (Explored[u] = 0)
      Explored[u]=1
      foreach node w in u.AdjList
          if (Level[w] = ∞)
             Q.Enqueue(w)
             Level[w] = Level[u]+1
    end if
```

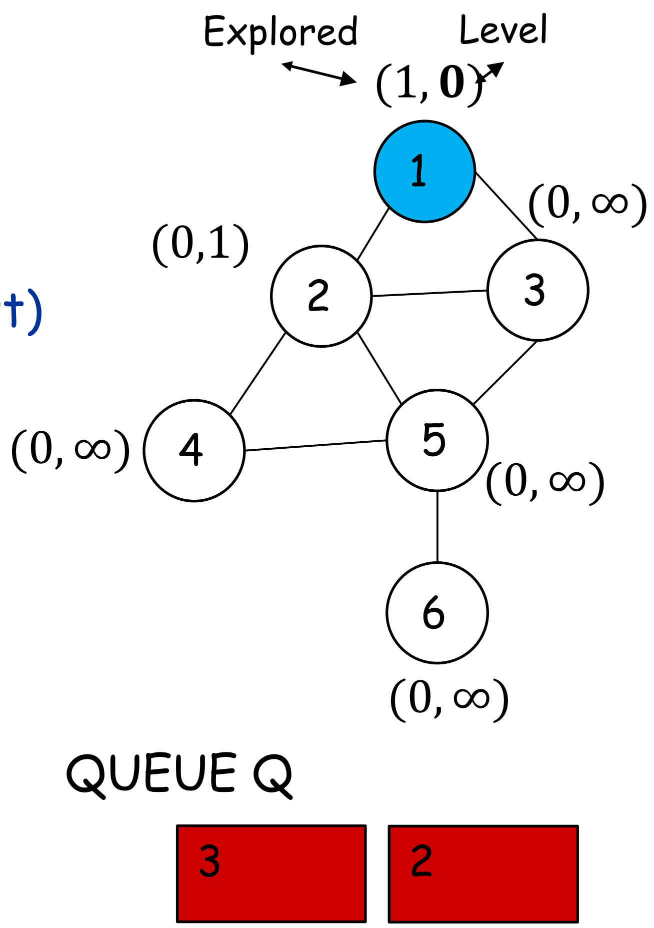


Breadth First Search: **BFS**(v, G=(V,E))

Input: Start node v, graph G (adjacency list) with n node V={1,…,n}

Output: Array Level[u] = distance from v to u

```
 For each node u in V
    Explored[u]=0
    Level[u] = ∞        // No v→u path found yet
 Q.Enqueue(v)           // Queue Q (First In, First Out)
 Level[v]=0
 while(Q is not empty)
   u = Q.Dequeue()
   if (Explored[u] = 0)
     Explored[u]=1
→    foreach node w in u.AdjList
  →    if (Level[w] = ∞)
           Q.Enqueue(w)
           Level[w] = Level[u]+1
   end if
```

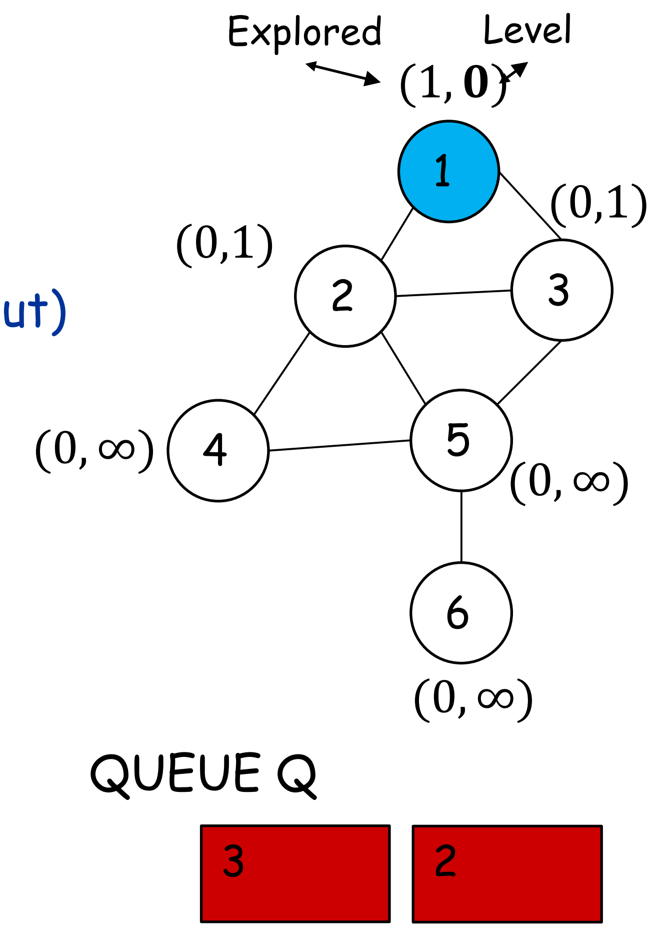


Breadth First Search: **BFS**(v, G=(V,E))

Input: Start node v, graph G (adjacency list) with n node V={1,…,n}

Output: Array Level[u] = distance from v to u

```
 For each node u in V
     Explored[u]=0
     Level[u] = ∞          // No v→u path found yet
 Q.Enqueue(v)              // Queue Q (First In, First Out)
 Level[v]=0
 while(Q is not empty)
   u = Q.Dequeue()
   if (Explored[u] = 0)
     Explored[u]=1
     foreach node w in u.AdjList
        if (Level[w] = ∞)
          Q.Enqueue(w)
          Level[w] = Level[u]+1
   end if
```

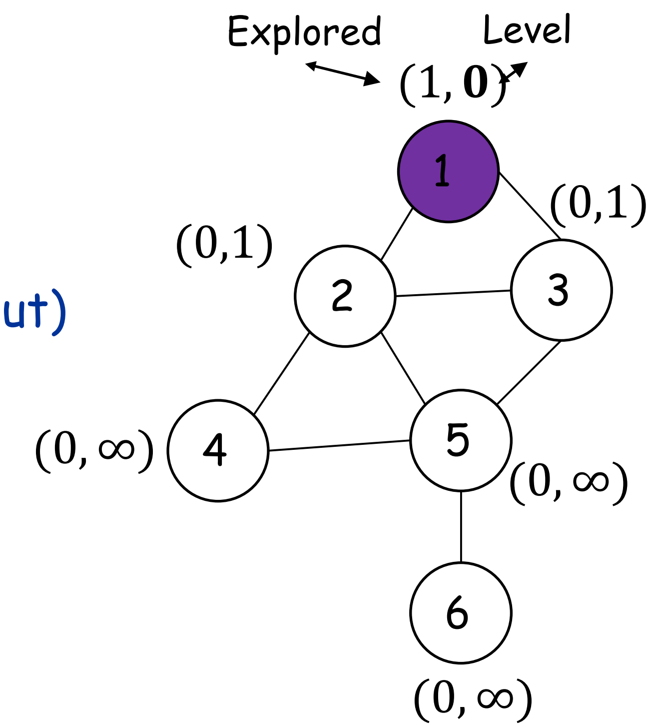


Breadth First Search: **BFS**(v, G=(V,E))

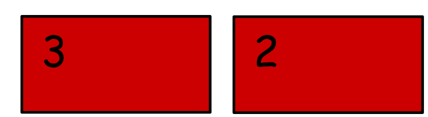
Input: Start node v, graph G (adjacency list) with n node V={1,…,n}

Output: Array Level[u] = distance from v to u

```
 For each node u in V
    Explored[u]=0
    Level[u] = ∞          // No v→u path found yet
 Q.Enqueue(v)            // Queue Q (First In, First Out)
 Level[v]=0
 while(Q is not empty)
   u = Q.Dequeue()
   if (Explored[u] = 0)
     Explored[u]=1
     foreach node w in u.AdjList
        if (Level[w] = ∞)
           Q.Enqueue(w)
           Level[w] = Level[u]+1
   end if
```



QUEUE Q

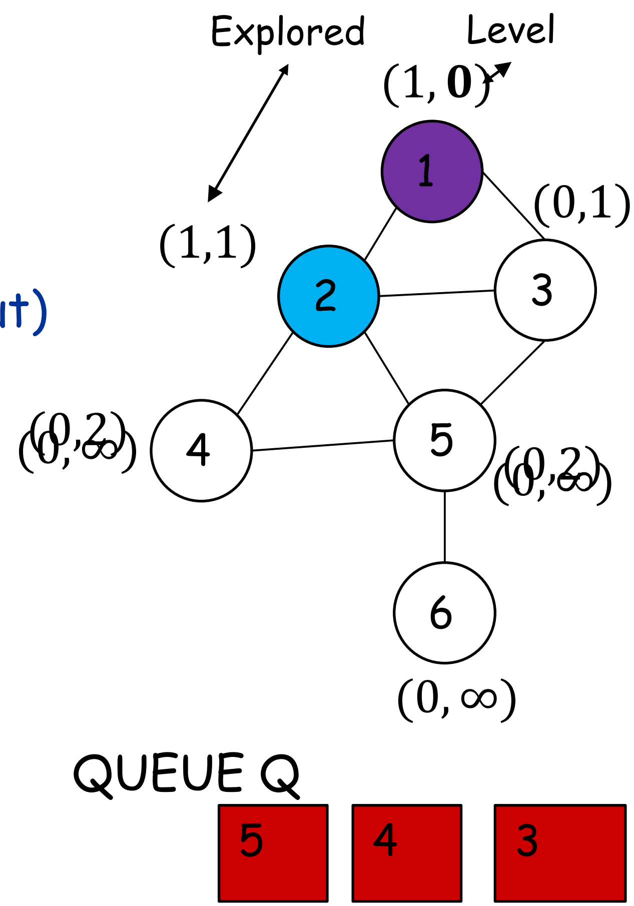


Breadth First Search: **BFS**(v, G=(V,E))

Input: Start node v, graph G (adjacency list) with n node V={1,…,n}

Output: Array Level[u] = distance from v to u

```
  For each node u in V
     Explored[u]=0
     Level[u] = ∞          // No v→u path found yet
  Q.Enqueue(v)            // Queue Q (First In, First Out)
  Level[v]=0
  while(Q is not empty)
    u = Q.Dequeue()
    if (Explored[u] = 0)
      Explored[u]=1
→     foreach node w in u.AdjList
        if (Level[w] = ∞)
          Q.Enqueue(w)
          Level[w] = Level[u]+1
    end if
```

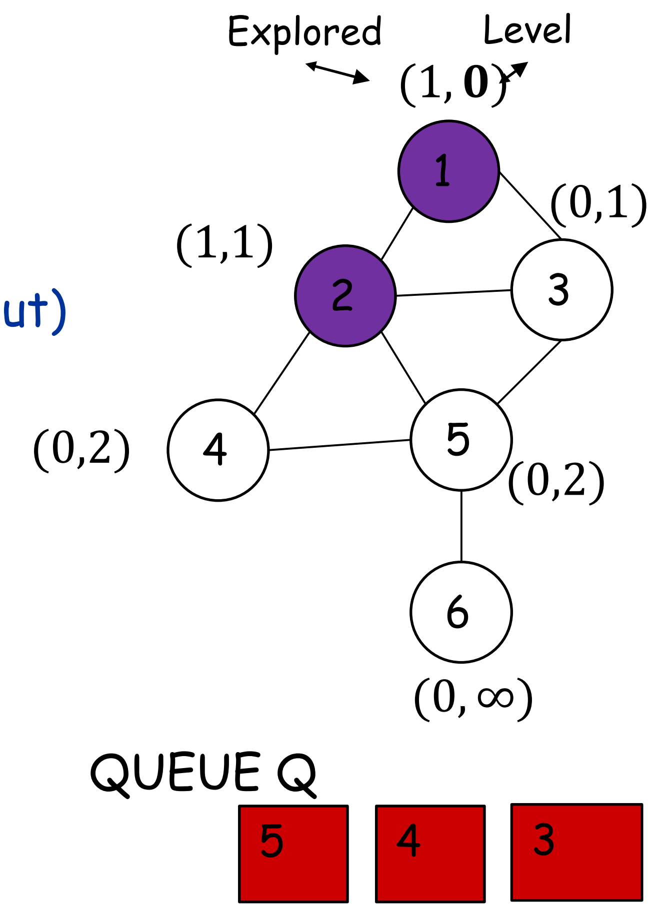


Breadth First Search: **BFS**(v, G=(V,E))

Input: Start node v, graph G (adjacency list) with n node V={1,…,n}

Output: Array Level[u] = distance from v to u

```
 For each node u in V
    Explored[u]=0
    Level[u] = ∞        // No v→u path found yet
 Q.Enqueue(v)           // Queue Q (First In, First Out)
 Level[v]=0
 while(Q is not empty)
   u = Q.Dequeue()
   if (Explored[u] = 0)
     Explored[u]=1
     foreach node w in u.AdjList
        if (Level[w] = ∞)
           Q.Enqueue(w)
           Level[w] = Level[u]+1
   end if
```

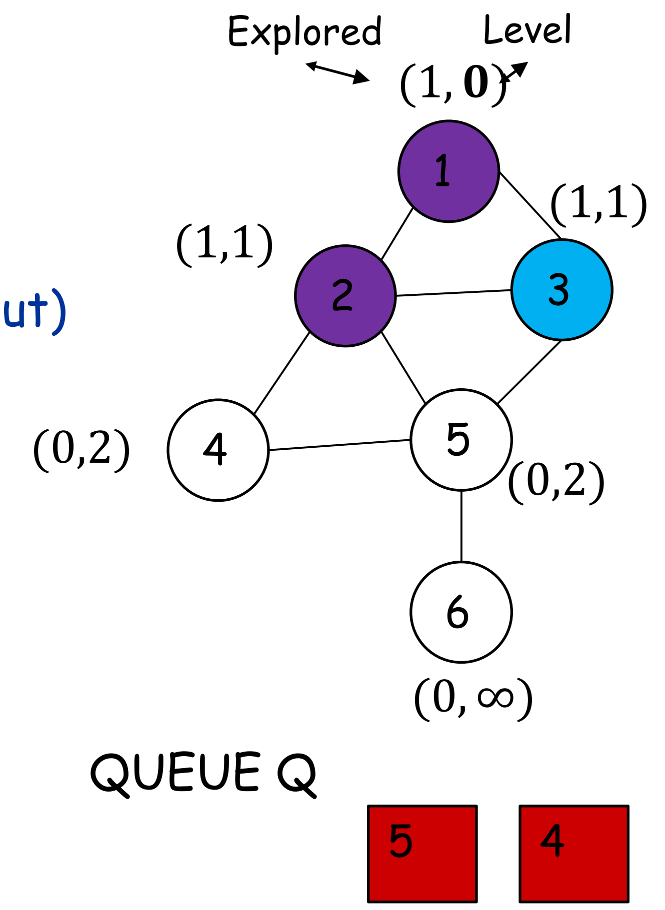


Breadth First Search: **BFS**(v, G=(V,E))

Input: Start node v, graph G (adjacency list) with n node V={1,…,n}

Output: Array Level[u] = distance from v to u

```
 For each node u in V
    Explored[u]=0
    Level[u] = ∞        // No v→u path found yet
 Q.Enqueue(v)           // Queue Q (First In, First Out)
 Level[v]=0
→ while(Q is not empty)
→   u = Q.Dequeue()
    if (Explored[u] = 0)
      Explored[u]=1
→     foreach node w in u.AdjList
        if (Level[w] = ∞)
          Q.Enqueue(w)
          Level[w] = Level[u]+1
    end if
```

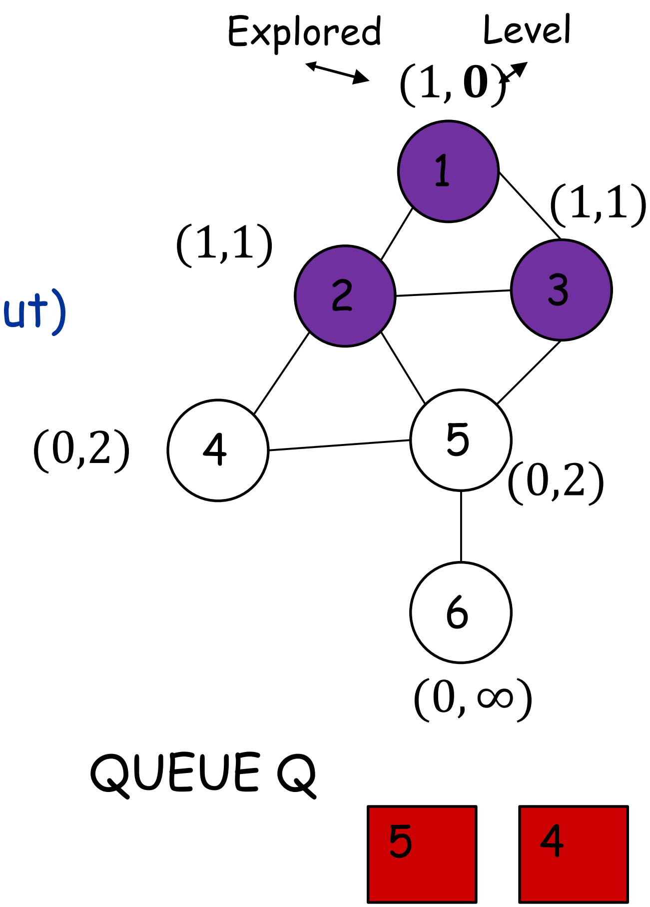


Breadth First Search: **BFS**(v, G=(V,E))

Input: Start node v, graph G (adjacency list) with n node V={1,…,n}

Output: Array Level[u] = distance from v to u

```
 For each node u in V
    Explored[u]=0
    Level[u] = ∞         // No v→u path found yet
Q.Enqueue(v)             // Queue Q (First In, First Out)
Level[v]=0
while(Q is not empty)
  u = Q.Dequeue()
  if (Explored[u] = 0)
    Explored[u]=1
    foreach node w in u.AdjList
       if (Level[w] = ∞)
           Q.Enqueue(w)
           Level[w] = Level[u]+1
  end if
```

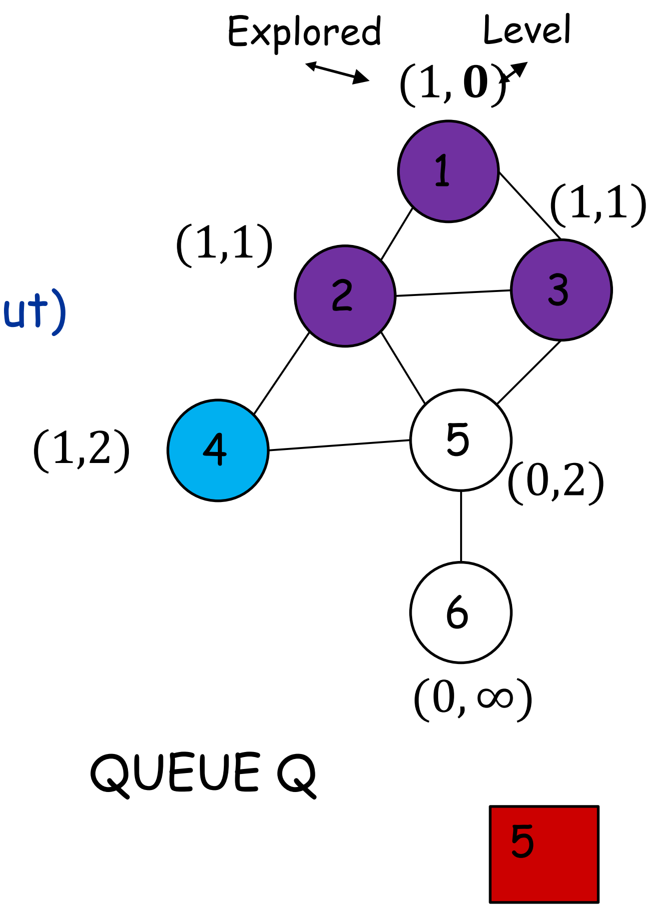


Breadth First Search: **BFS**(v, G=(V,E))

Input: Start node v, graph G (adjacency list) with n node V={1,…,n}

Output: Array Level[u] = distance from v to u

```
For each node u in V
    Explored[u]=0
    Level[u] = ∞          // No v→u path found yet
Q.Enqueue(v)              // Queue Q (First In, First Out)
Level[v]=0
while(Q is not empty)
    u = Q.Dequeue()
    if (Explored[u] = 0)
        Explored[u]=1
        foreach node w in u.AdjList
            if (Level[w] = ∞)
                Q.Enqueue(w)
                Level[w] = Level[u]+1
    end if
```

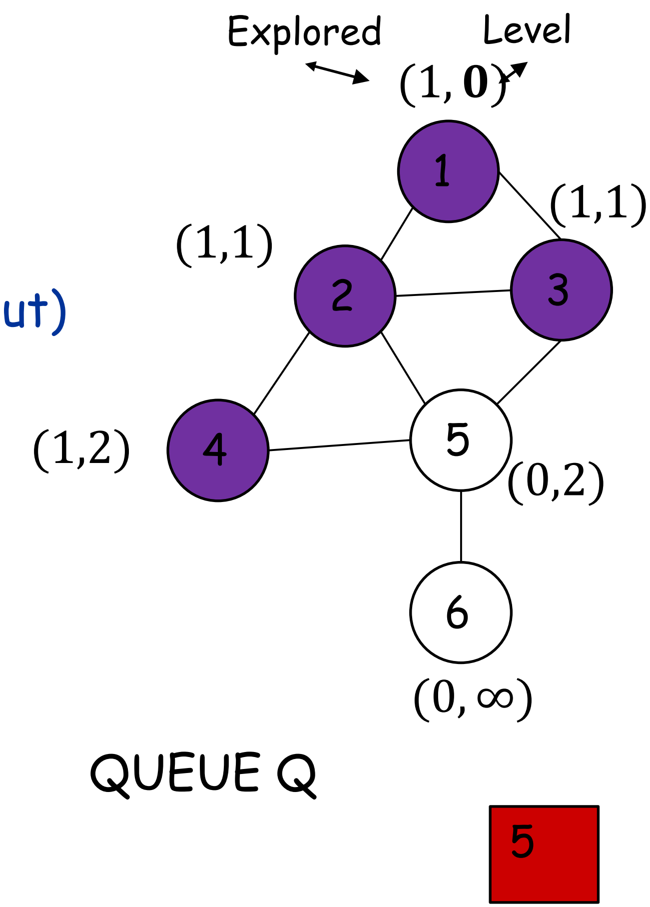


Breadth First Search: **BFS**(v, G=(V,E))

Input: Start node v, graph G (adjacency list) with n node V={1,…,n}

Output: Array Level[u] = distance from v to u

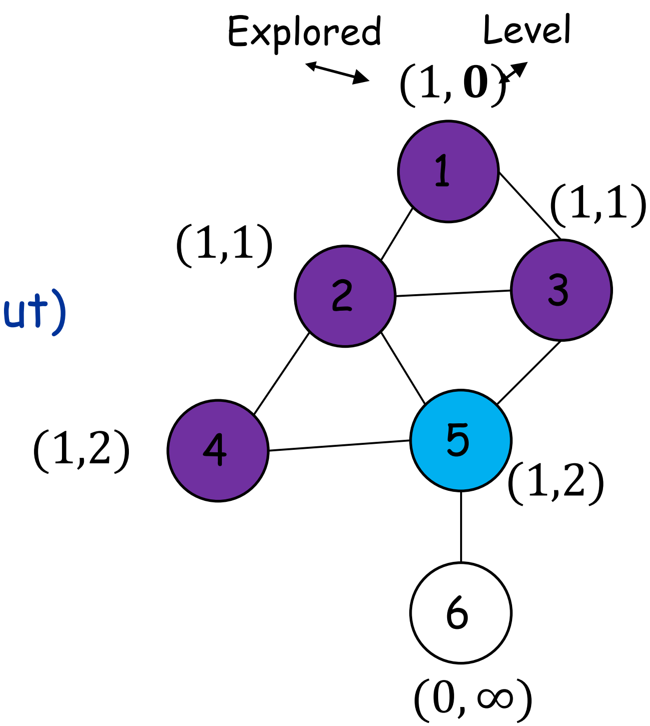
```
  For each node u in V
     Explored[u]=0
     Level[u] = ∞          // No v→u path found yet
  Q.Enqueue(v)             // Queue Q (First In, First Out)
  Level[v]=0
→ while(Q is not empty)
→   u = Q.Dequeue()
    if (Explored[u] = 0)
       Explored[u]=1
       foreach node w in u.AdjList
          if (Level[w] = ∞)
              Q.Enqueue(w)
              Level[w] = Level[u]+1
    end if
```



Breadth First Search: **BFS**(v, G=(V,E))

Input: Start node v, graph G (adjacency list) with n node V={1,…,n}
Output: Array Level[u] = distance from v to u

```
 For each node u in V
    Explored[u]=0
    Level[u] = ∞        // No v→u path found yet
 Q.Enqueue(v)           // Queue Q (First In, First Out)
 Level[v]=0
→ while(Q is not empty)
→   u = Q.Dequeue()
    if (Explored[u] = 0)
      Explored[u]=1
→     foreach node w in u.AdjList
         if (Level[w] = ∞)
            Q.Enqueue(w)
            Level[w] = Level[u]+1
    end if
```



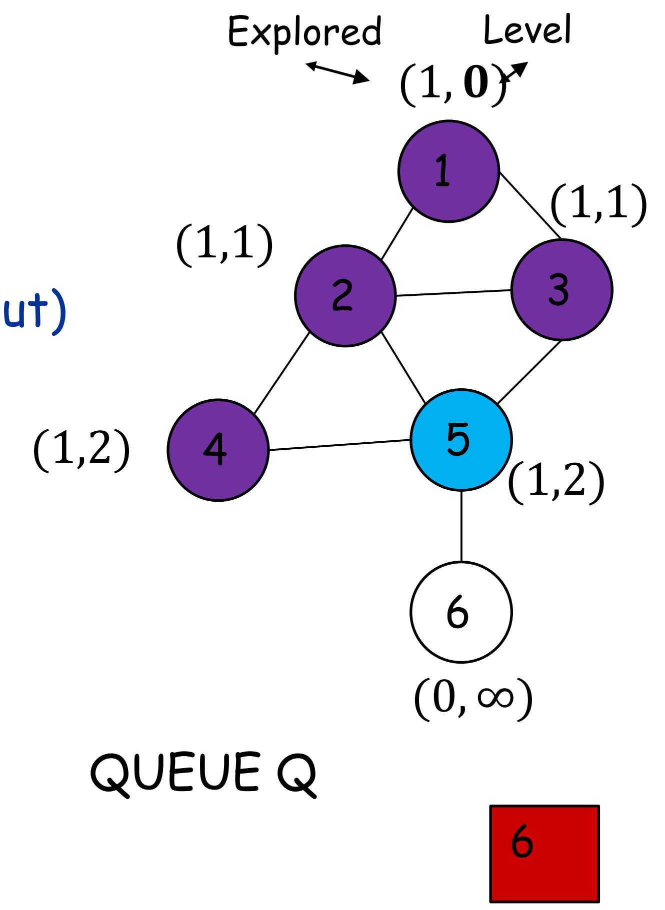
QUEUE Q

Breadth First Search: **BFS**(v, G=(V,E))

Input: Start node v, graph G (adjacency list) with n node V={1,…,n}

Output: Array Level[u] = distance from v to u

```
For each node u in V
    Explored[u]=0
    Level[u] = ∞        // No v→u path found yet
Q.Enqueue(v)            // Queue Q (First In, First Out)
Level[v]=0
while(Q is not empty)
  u = Q.Dequeue()
  if (Explored[u] = 0)
    Explored[u]=1
    foreach node w in u.AdjList
        if (Level[w] = ∞)
            Q.Enqueue(w)
            Level[w] = Level[u]+1
  end if
```

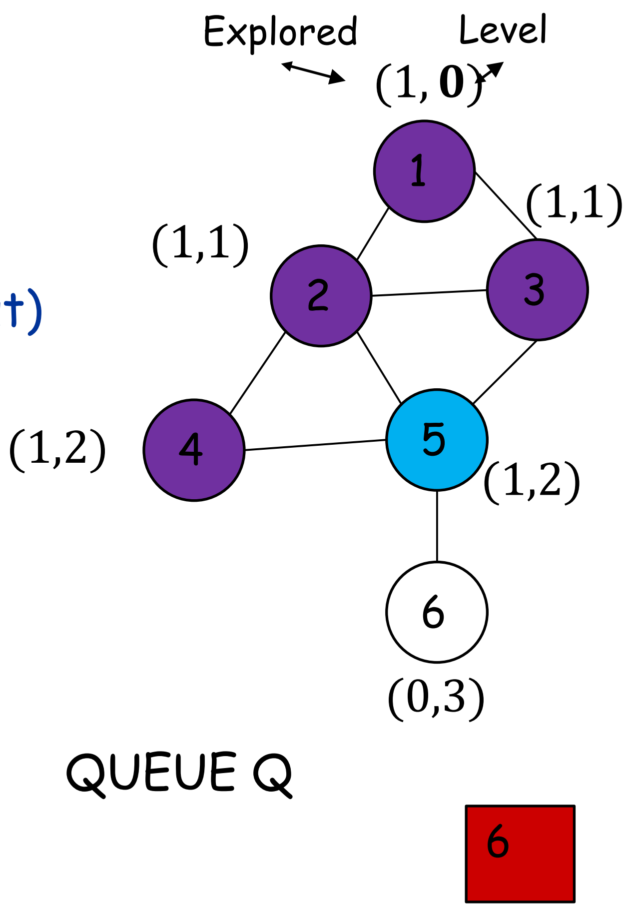


Breadth First Search: **BFS**(v, G=(V,E))

Input: Start node v, graph G (adjacency list) with n node V={1,…,n}

Output: Array Level[u] = distance from v to u

```
  For each node u in V
     Explored[u]=0
     Level[u] = ∞        // No v→u path found yet
  Q.Enqueue(v)           // Queue Q (First In, First Out)
  Level[v]=0
→ while(Q is not empty)
→   u = Q.Dequeue()
    if (Explored[u] = 0)
      Explored[u]=1
→     foreach node w in u.AdjList
          if (Level[w] = ∞)
             Q.Enqueue(w)
             Level[w] = Level[u]+1
    end if
```

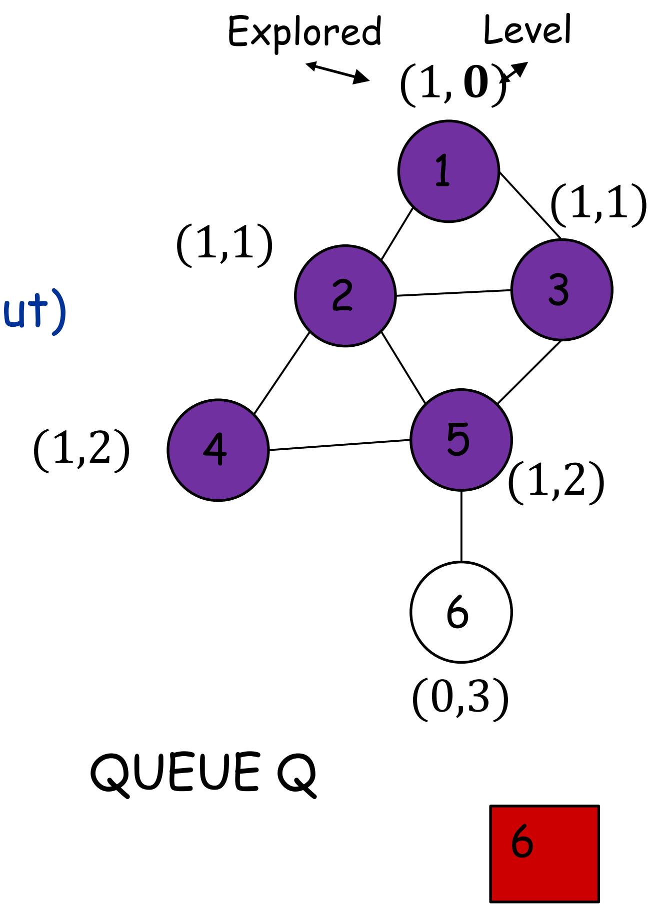


Breadth First Search: **BFS**(v, G=(V,E))

Input: Start node v, graph G (adjacency list) with n node V={1,…,n}

Output: Array Level[u] = distance from v to u

```
  For each node u in V
     Explored[u]=0
     Level[u] = ∞          // No v→u path found yet
  Q.Enqueue(v)             // Queue Q (First In, First Out)
  Level[v]=0
→ while(Q is not empty)
→   u = Q.Dequeue()
    if (Explored[u] = 0)
       Explored[u]=1
→      foreach node w in u.AdjList
          if (Level[w] = ∞)
             Q.Enqueue(w)
             Level[w] = Level[u]+1
    end if
```

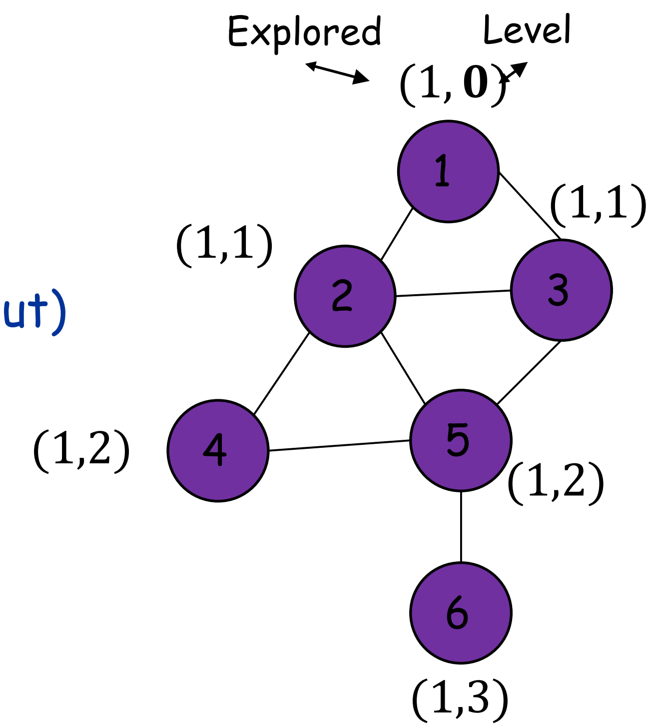


Breadth First Search: **BFS**(v, G=(V,E))

Input: Start node v, graph G (adjacency list) with n node V={1,…,n}

Output: Array Level[u] = distance from v to u

```
 For each node u in V
    Explored[u]=0
    Level[u] = ∞          // No v→u path found yet
 Q.Enqueue(v)             // Queue Q (First In, First Out)
 Level[v]=0
 while(Q is not empty)
   u = Q.Dequeue()
   if (Explored[u] = 0)
      Explored[u]=1
      foreach node w in u.AdjList
         if (Level[w] = ∞)
            Q.Enqueue(w)
            Level[w] = Level[u]+1
   end if
```

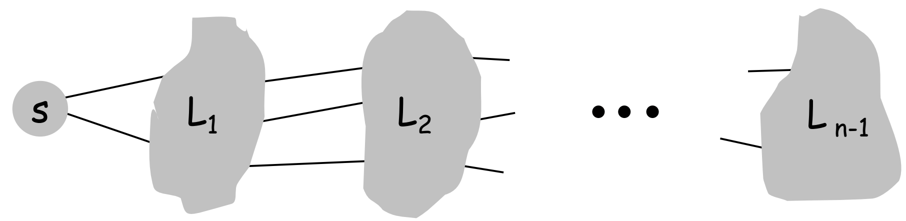


QUEUE Q

Done!

Breadth First Search

BFS intuition. Explore outward from s in all possible directions, adding nodes one "layer" at a time.



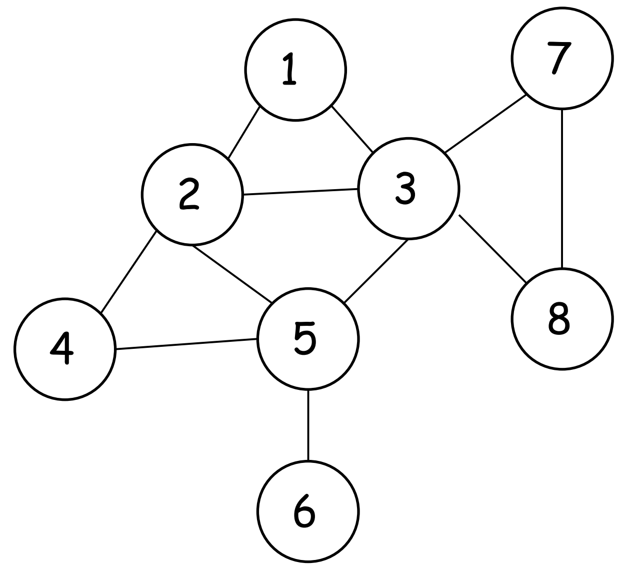
BFS algorithm.

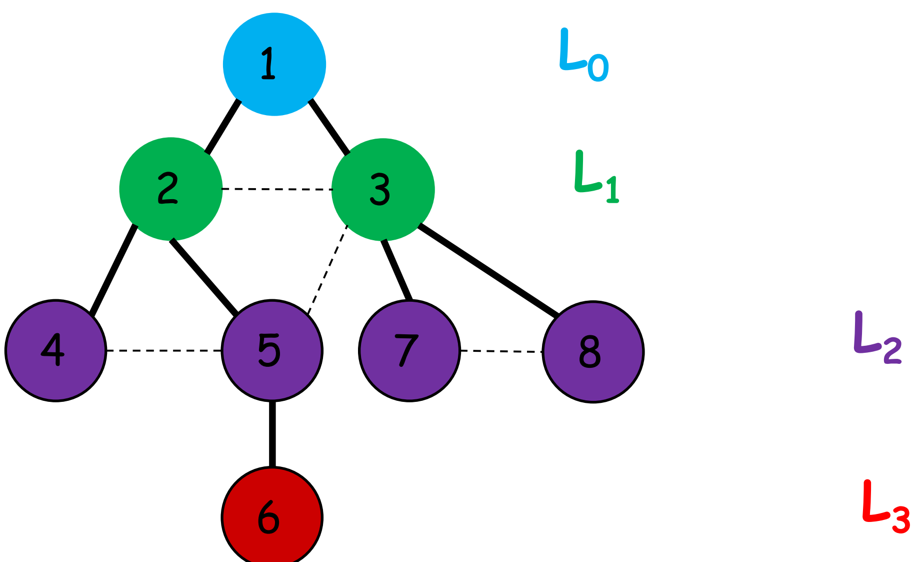
- $L_0 = \{ s \}$.
- L_1 = all neighbors of L_0.
- L_2 = all nodes that do not belong to L_0 or L_1, and that have an edge to a node in L_1.
- L_{i+1} = all nodes that do not belong to an earlier layer, and that have an edge to a node in L_i.

Theorem. For each i, L_i consists of all nodes at distance exactly i from s. There is a path from s to t iff t appears in some layer.

Breadth First Search

Property. Let T be a BFS tree of G = (V, E), and let (x, y) be an edge of G. Then the level of x and y differ by at most 1.





Breadth First Search: Analysis

Theorem. The above implementation of BFS runs in $O(m + n)$ time if the graph is given by its adjacency representation.

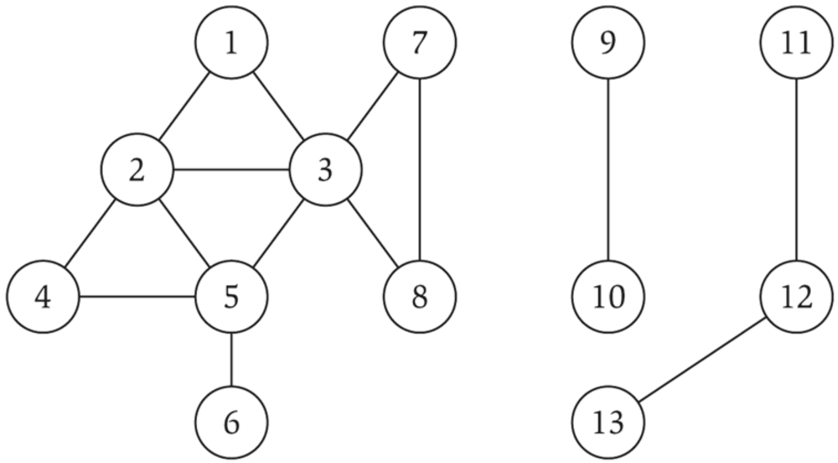
Pf.

- Easy to prove $O(n^2)$ running time:
  - at most n lists L[i]
  - each node occurs on at most one list; for loop runs $\leq$ n times
  - when we consider node u, there are $\leq$ n incident edges (u, v), and we spend $O(1)$ processing each edge
- Actually runs in $O(m + n)$ time:
  - when we consider node u, there are deg(u) incident edges (u,v)
  - total time processing edges is $\Sigma_{u \in V} \deg(u) = 2m$ ▪

each edge (u, v) is counted exactly twice
in sum: once in deg(u) and once in deg(v)

Connected Component

Connected component. Find all nodes reachable from s.



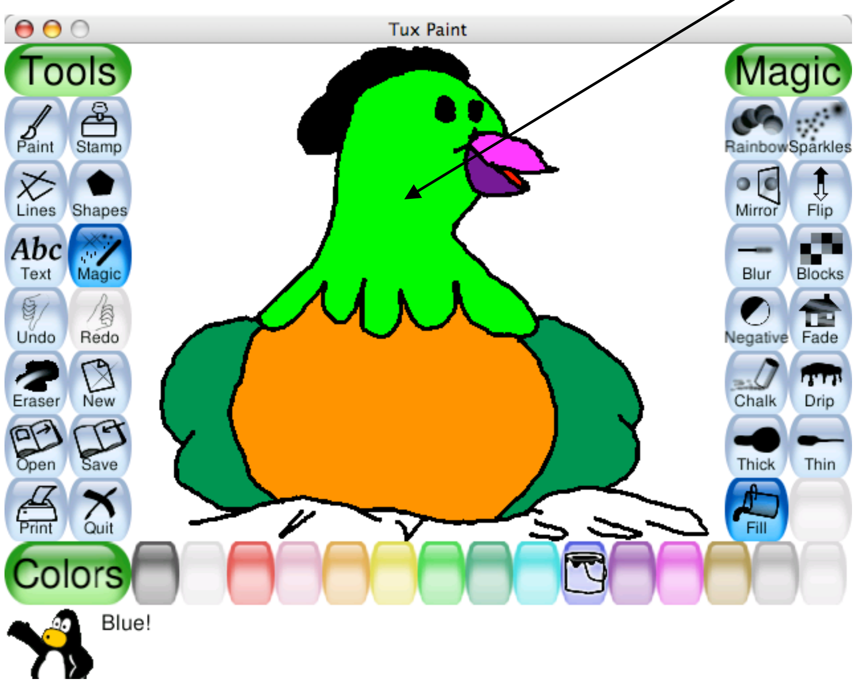
Connected component containing node 1 = { 1, 2, 3, 4, 5, 6, 7, 8 }.

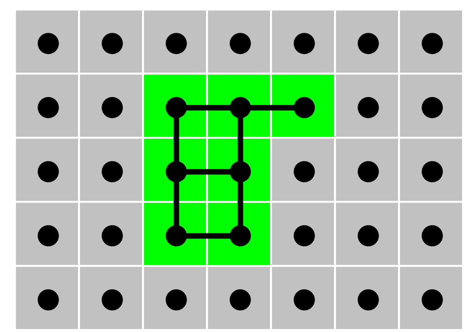
Flood Fill

Flood fill. Given lime green pixel in an image, change color of entire blob of neighboring lime pixels to blue.

- Node: pixel.
- Edge: two neighboring lime pixels.
- Blob: connected component of lime pixels.

recolor lime green blob to blue



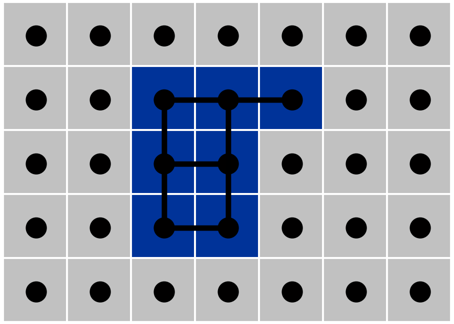


Flood Fill

Flood fill. Given lime green pixel in an image, change color of entire blob of neighboring lime pixels to blue.

- Node: pixel.
- Edge: two neighboring lime pixels.
- Blob: connected component of lime pixels.

recolor lime green blob to blue

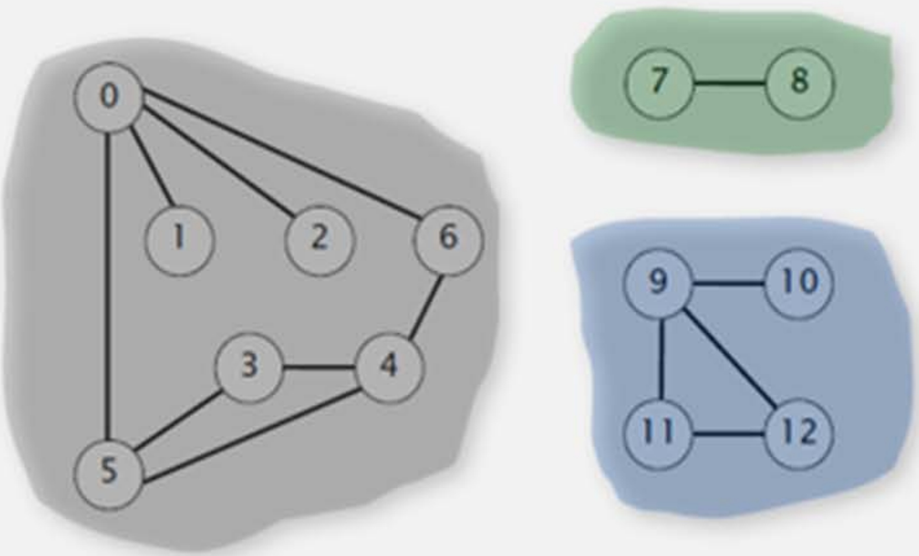


What problems can BFS solve?

- Check if an undirected graph is connected
- Determine *connected components* in an undirected graph
- Identify shortest path from source node to a destination node
- Is G a tree? (Does G have a cycle)

Finding connected components

Def. A connected component is a maximal set of connected vertices.



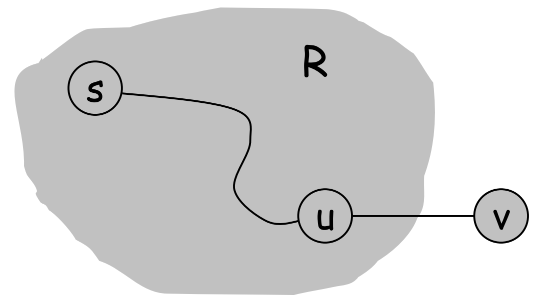
3 connected components

v	id[]
0	0
1	0
2	0
3	0
4	0
5	0
6	0
7	1
8	1
9	2
10	2
11	2
12	2

Sedgewick-Wayne 251 text

Connected component. Find all nodes reachable from s.

```
R will consist of nodes to which s has a path
Initially R = {s}
While there is an edge (u, v) where u ∈ R and v ∉ R
  Add v to R
Endwhile
```



it's safe to add v

Theorem. Upon termination, R is the connected component containing s.

- BFS = explore in order of distance from s.
- DFS = explore in a different way.

Challenge Puzzle

Suppose the shortest path from u to v has length t > n/2. Show that there exists some node $\mathbf{w} \neq \boldsymbol{u}, \boldsymbol{v}$ s.t. deleting w from G disconnects u and v.

Hint: Consider the BFS tree!

Solution: Run BFS(u) to obtain levels $L_0, L_1, \ldots$

Observation 1: v is in level L_t

Observation 2: $|L_0|+\ldots+|L_t| <= n$

Observation 3: Must have some $0 < k < t$ with $|L_k|=1$ by observation 2

Otherwise $|L_0|+|L_t|+(|L_1|+\ldots+|L_{t-1}|) \geq 2 + 2(t-1) > n$

Observation 4: Removing level L_k disconnects u and v

(BFS tree property → if $i < k < j$ then no edge connects L_i and L_j)

Depth First Search (22.3+22.5; 251 text)

- **DFS performs a recursive exploration of a graph**
  - DFS follows a path until it is forced to back up – “backtracking”
- DFS operates on an adjacency list representation
- Most uses of DFS result in O(n+m) time
- DFS be used on directed and undirected graphs
- **Undirected graph**
  - DFS partitions the edges into tree and back edges
  - Assigns numbers to the vertices during exploration (e.g. DFS number, discovery number, finish number)