Graph Algorithms

Sridhar Alagar

Basic Definitions

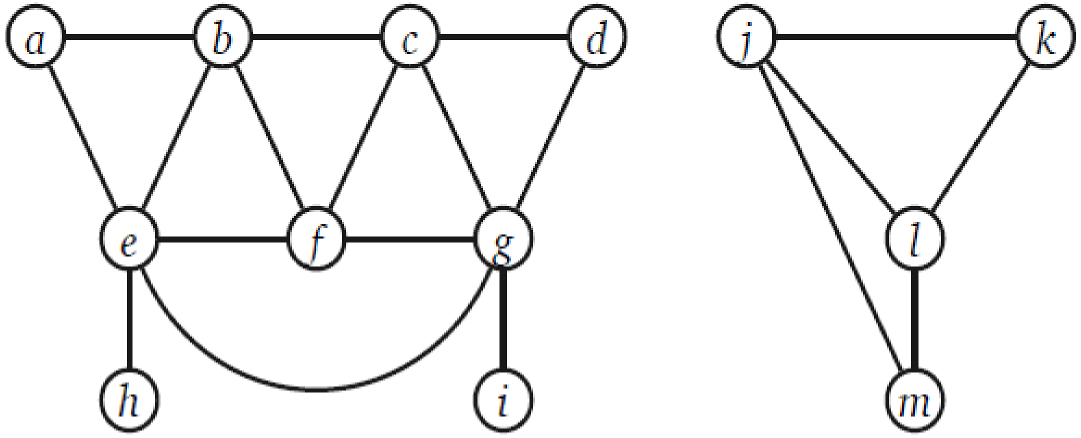
- $G = (V, E)$
  - V is a non-empty, finite set of vertices/nodes
  - E is a set of edges; $E \subseteq V \times V$
- Undirected graph: edges are unordered pairs
  - $(u, v) \Rightarrow (v, u)$
- Directed graph: edges are ordered

- Graph is simple if there are no self loops and no parallel edges; otherwise, it is multi-graph

- Graph is mixed if it has both directed and undirected edges

Basic Definitions

- u and v are neighbors if there is an edge (u, v) in E
- Degree of a node is the number of neighbors or edges incident on the node
  - In-degree is number of incoming edges
  - Out-degree is number of outgoing edges
  - In an undirected graph in-deg(u) = out-deg(v)
- Walk is a sequence of vertices where each successive pairs are adjacent/neighbors
- A path is a walk where no vertices are visited more than once

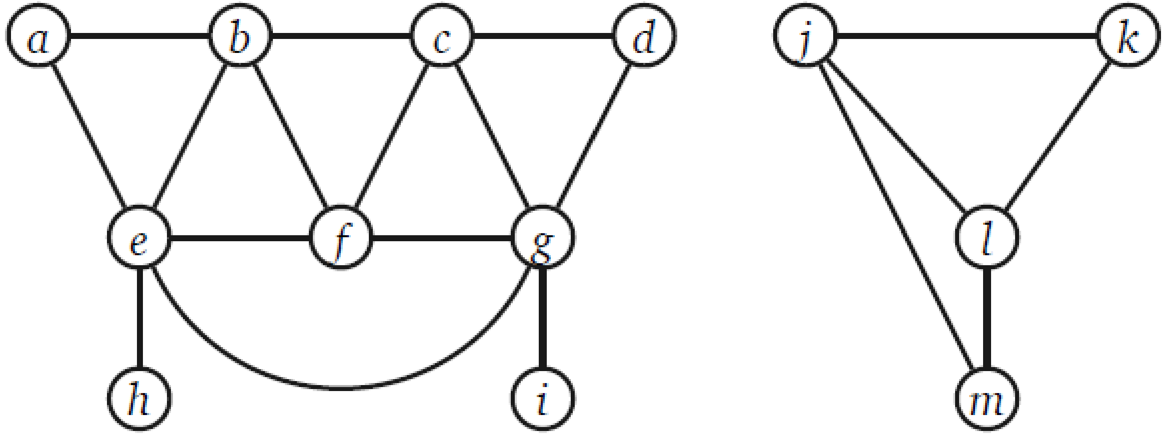


Basic Definitions

- v is reachable from u if there is a path from u to v

- A graph is connected if every vertex is reachable from every other vertex

- A component is a maximally connected sub-graph

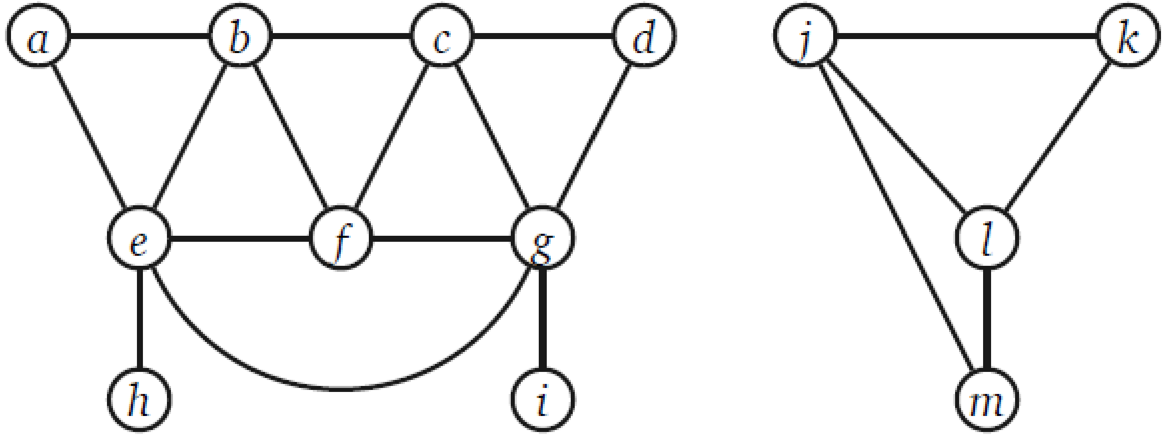


Data Structures: Adjacency Matrix

- A[i,j] = 1 if (i, j) is in E

- Advantages?

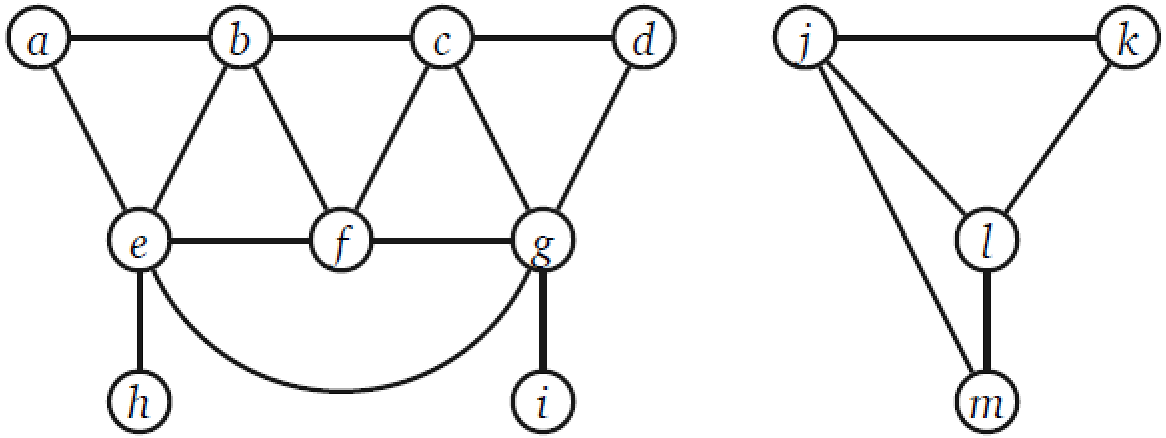
- Disadvantages?



Data Structures: Adjacency List

- Array of list of vertices?

List<Vertex> []



- Array of list of edges?

List<Edge>[]

Comparison: Adj. list vs Adj. matrix

Operations	Adj. list	Adj. matrix
Space		
Test (u,v) in E		
List neighbors of u		
List all edges		
Insert edge		
Delete edge		

Whatever-First Search

Reachability Problem: Given G and start vertex s, which vertices are reachable from s? Assume G is undirected

```
wfs(s){
     put s into bag // bag is a generic data structure
     while bag not empty
          take v from bag
          if v is unmarked
               mark v
               for each (v, w) in G do
                    put w in bag
}
```

RT: $O(V + ET)$
T is time taken to add/delete from a bag

Variants: Based on data structure used

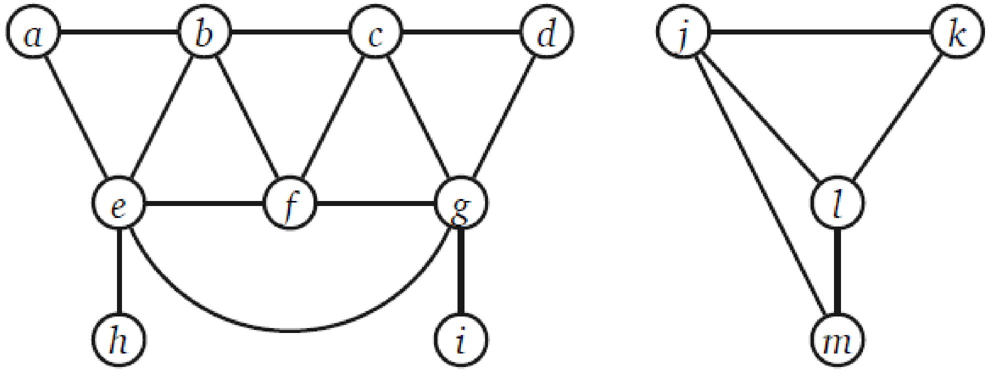
Queue	Stack	Priority Queue
add instead of put	push instead of put	add instead of put
remove instead of take	pop instead of take	remove instead of take
O(V + E)	O(V + E)	O(V + E log E)
Breadth first tree	Depth first tree	Best first spanning tree
		Family of algorithms depending on the priority

Best First Search

- G is undirected and weight of edge is priority,
  - it is MST
  - Commonly called as Kruskal's algorithm

- Use distance from s as priority,
  - it is SPT from s
  - dist(s) = 0
  - dist(v) = dist(p) + w(p, v)
  - Update dist(v) whenever parent(v) <- p
  - when (v, w) added to PQ, use priority dist(v) + w(v, w)

WFS - visit all nodes



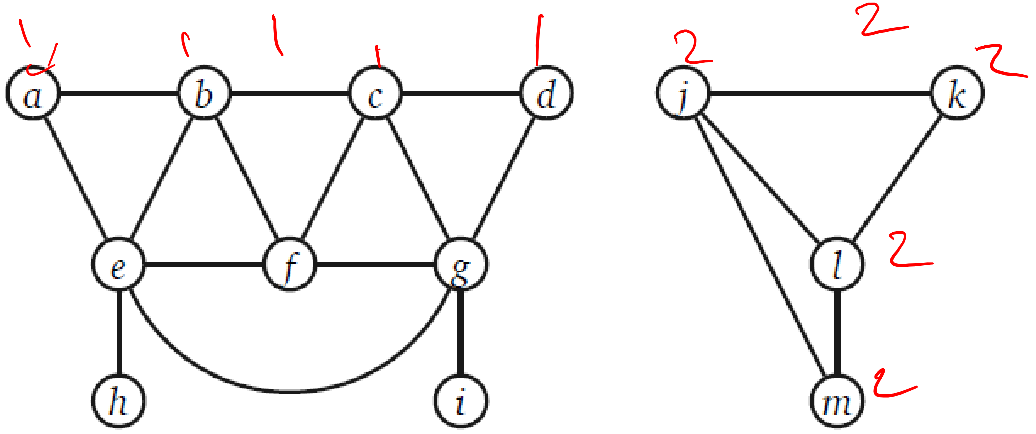
May not visit all nodes

wfs(a) will visit only all the nodes in the component of 'a'

```
wfsAll(G){
      for each v in V do
            unmark v
      for each v in V do
            if v is unmarked
                  wfs(v)
}
```

```
wfs(s){
   put s into bag
   while bag not empty
         take v from bag
         if v is unmarked
            mark v
            for each (v, w) in G do
               put w in bag
}
```

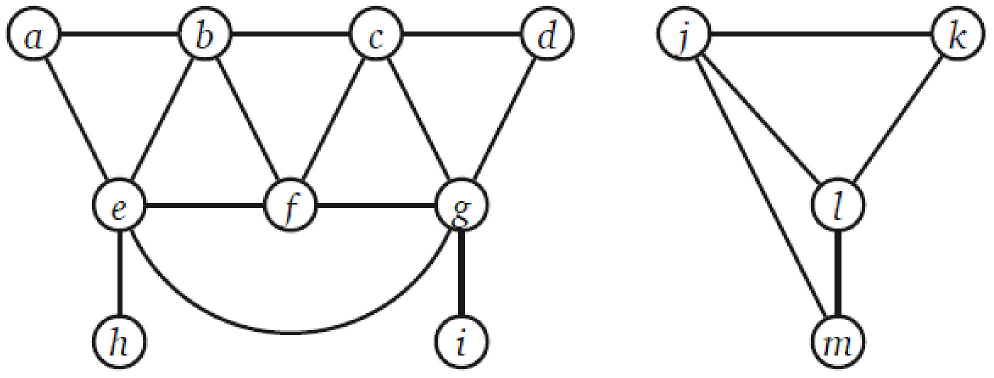
Count Components




```
countComponents(G){
        count = 0
        for each v in V do
                unmark v
        for each v in V do
                if v is unmarked
                        count++
                        wfs(v)
}
```

```
wfs(s){
    put s into bag
    while bag not empty
        take v from bag
        if v is unmarked
            mark v
            for each (v, w) in G do
                put w in bag
}
```

Label Vertex with Component num




```
countandLabel(G){
      count = 0
      for each v in V do
            unmark v
      for each v in V do
            if v is unmarked
                  count++
                  Label(v, count)
}
```

```
Label(s, count){//label one component
      put s into bag
      while bag not empty
            take v from bag
            if v is unmarked
               mark v
               comp(v) = count
               for each (v, w) in G do
                  put w in bag
}
```

Problem of the day: Snakes and Ladders

Snakes and Ladders is a classic board game, originating in India no later than the 16th century. The board consists of an $n \times n$ grid of squares, numbered consecutively from 1 to n^2, starting in the bottom left corner and proceeding row by row from bottom to top, with rows alternating to the left and right. Certain pairs of squares in this grid, always in different rows, are connected by either "snakes" (leading down) or "ladders" (leading up). Each square can be an endpoint of at most one snake or ladder.

You start with a token in cell 1, in the bottom left corner. In each move, you advance your token up to k positions, for some fixed constant k. If the token ends the move at the *top* end of a snake, it slides down to the bottom of that snake. Similarly, if the token ends the move at the *bottom* end of a ladder, it climbs up to the top of that ladder.

Describe and analyze an algorithm to compute the smallest number of moves required for the token to reach the last square of the grid.