Data Structures

November 2

Graphs

Objectives

Discuss the following topics:

- Graphs; Graphs as ADT
- Graph Representation
- Graph Traversals (breadth first, depth first)
- Connectivity
- Bipartiteness
- Topological Sort (aka topological ordering)
- Minimum Spanning Trees (Kruskal’s and Prim’s algorithms)
- Shortest Paths

Graphs

- A **graph** is a collection of vertices (or nodes) and the connections between them
- A **simple graph** *G = (V, E)* consists of a nonempty set *V* of **vertices** and a possibly empty set *E* of **edges**, each edge being a set of two vertices from *V*
- A **directed graph**, or a **digraph**, *G* = (*V*, *E*) consists of a nonempty set *V* of vertices and a set *E* of edges (also called **arcs**), where each edge is a pair of vertices from *V*

4

Graphs (continued)

- A **multigraph** is a graph in which two vertices can be joined by multiple edges
- A **pseudograph** is a multigraph with the condition $v_i \neq v_j$ removed, which allows for loops to occur
- A graph is called a **weighted graph** if each edge has an assigned number

5

Graphs (continued)

- A **path** from v_1 to v_n is a sequence of edges *edge*(v_1,v_2), *edge*(v_2,v_3), …, *edge*(v_{n-1},v_n)
- If v_1=v_n, and no edge is repeated, then the path is called a **circuit**
- If all vertices in a circuit are different, then it is called a **cycle**.

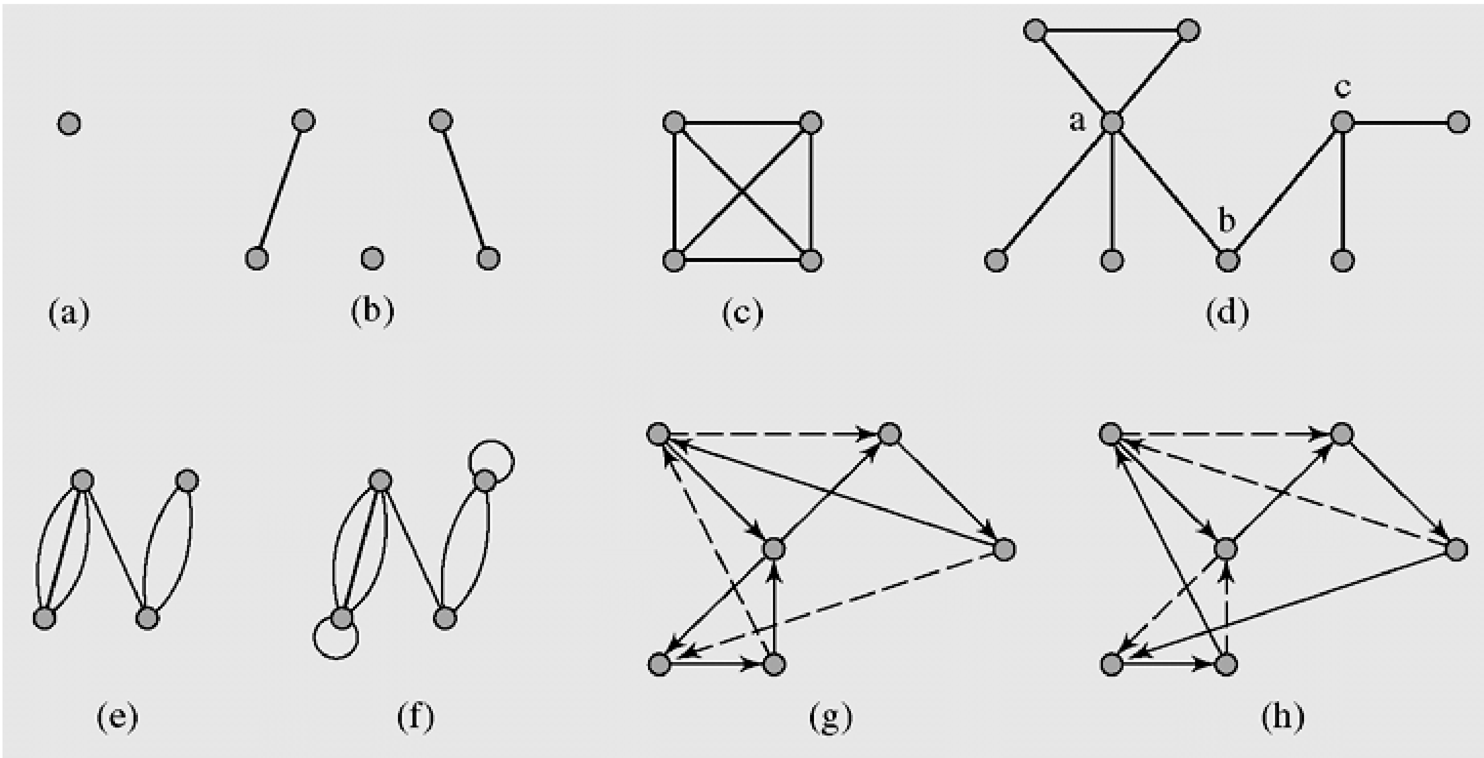
6

Graphs (continued)

- An undirected graph is **connected** if for every pair of nodes u and v, there is a path from u to v
- Any two connected components of an undirected graph either coincide or are disjoint.
- A directed graph is **strongly connected** if for every pair of nodes u and v, there is a path from u to v and a path from v to u
- Any two strongly connected components coincide or are disjoint

7

Graphs (continued)



Examples of graphs: (a–d) simple graphs; (c) a complete graph K_4; (e) a multigraph; (f) a pseudograph; (g) a circuit in a digraph; (h) a cycle in the digraph

8

Graph as an ADT

- Insertion and deletion somewhat different for graphs than for other ADTs: they can either apply to edges or vertices
- Can define the ADT graph so that its vertices contain or don't contain any values
- Not uncommon: graph representing only relationships among vertices – vertices don't contain values
- Our definition of ADT graph operations do assume that the graph's vertices contain values

Graph as an ADT (cont'd)

- **createGraph(G)** // creates empty //graph
- **destroyGraph(G)** // destroys the //graph
- **graphIsEmpty(G)** // returns true if //the graph is empty; otherwise //false
- **insertVertex(G, v, success)** // //inserts a vertex v into the graph //G whose vertices have distinct //search keys that differ from v's //search key. Success indicates //whether the insertion was //successful

Graph as an ADT (cont'd)

- **insertEdge(G, v1, v2, success)** //
  //inserts an edge between vertices
  //v1 and v2 in the graph G and sets
  //success to true. However, if an
  //edge already exists between
  //specified vertices, sets success
  //to false
- **deleteVertex(G,v, success)** //
  //Deletes the vertex v from the
  //graph G, and sets success to true.
  //However, if no such vertex exists,
  //sets success to false

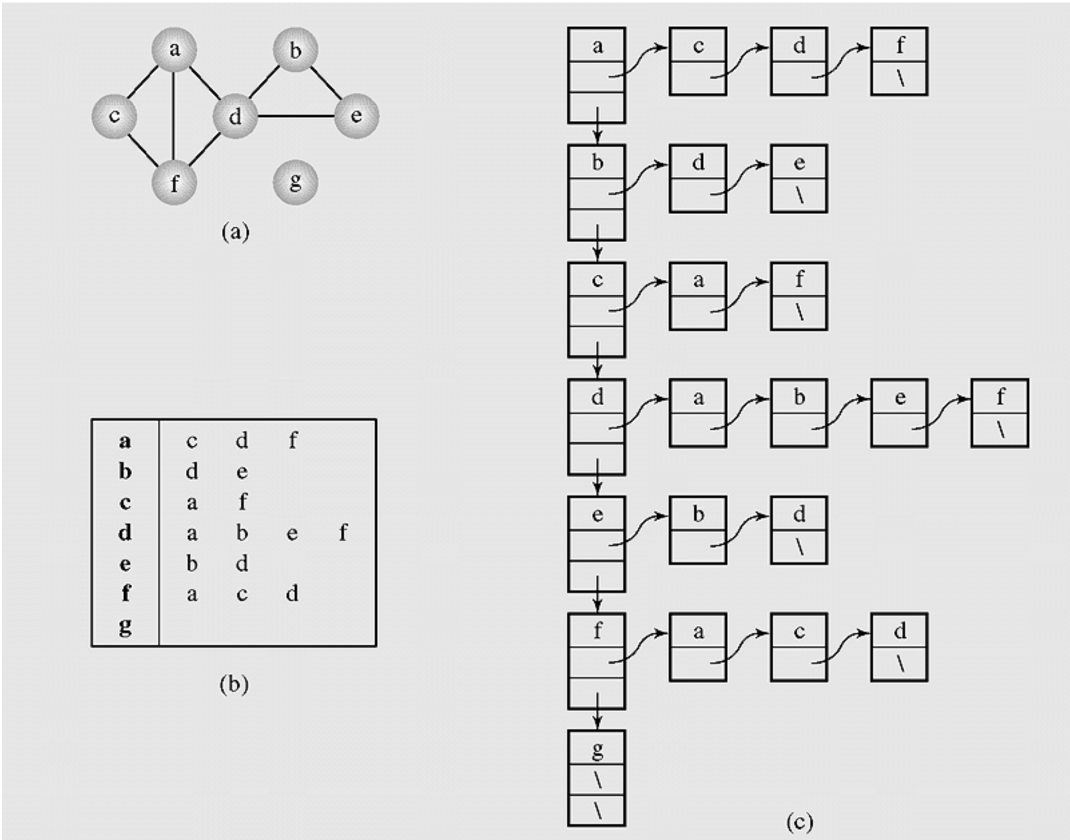
Graph as an ADT (cont'd)

- **deleteEdge(G, v1,v2,success**) //
  //deletes the edge between vertices
  //v1 and v2 in the graph G and sets
  //success to true. However, if no
  //edge exists between the specified
  //vertices, sets success to false
- **retrieveVertex(G, searchKey, v,**
  success) // copies into v the
  //vertex, if any, of G that contains
  //the searchKey. Sets success to
  //true if the vertex was found;
  //otherwise sets it to false

Graph as an ADT (cont'd)

- **`replaceVertex(G,searchKey, v, success)`** `//replaces the vertex that contains searchKey with v. Sets success to true if the vertex was found; otherwise sets it to false.`
- **`isEdge(G, v1, v2)`** `// returns true, if an edge between vertices v1 and v2 exists; otherwise returns false.`
- NB several variations of this ADT are possible, of course. For example, if the graph is directed, you can replace instances of "edges" in the previous specification with "directed edges". Can also add traversal operations. Very often graph is also weighted, so need to deal with retrieving, updating, inserting of weights on edges.

Graph Representation



Graph representations (a) A graph represented as (b–c) an adjacency list

$m \leq n(n-1)/2! \leq n^2$ (m=#edges=|E|; n=#nodes = |V|)

G connected: $n-1 \leq m \leq n(n-1)/2! \leq n^2$

G sparse: $m << n(n-1)/2!$

Adjacency matrix requires $O(n^2)$ space; process neighbors of v needs |V| steps.

Adjacency list: $O(m+n)$ space; steps; process neighbors of v needs deg(v) steps

14

Graph Representation (continued)

	a	b	c	d	e	f	g
a	0	0	1	1	0	1	0
b	0	0	0	1	1	0	0
c	1	0	0	0	0	1	0
d	1	1	0	0	1	1	0
e	0	1	0	1	0	0	0
f	1	0	1	1	0	0	0
g	0	0	0	0	0	0	0

(d)

	ac	ad	af	bd	be	cf	de	df
a	1	1	1	0	0	0	0	0
b	0	0	0	1	1	0	0	0
c	1	0	0	0	0	1	0	0
d	0	1	0	1	0	0	1	1
e	0	0	0	0	1	0	1	0
f	0	0	1	0	0	1	0	1
g	0	0	0	0	0	0	0	0

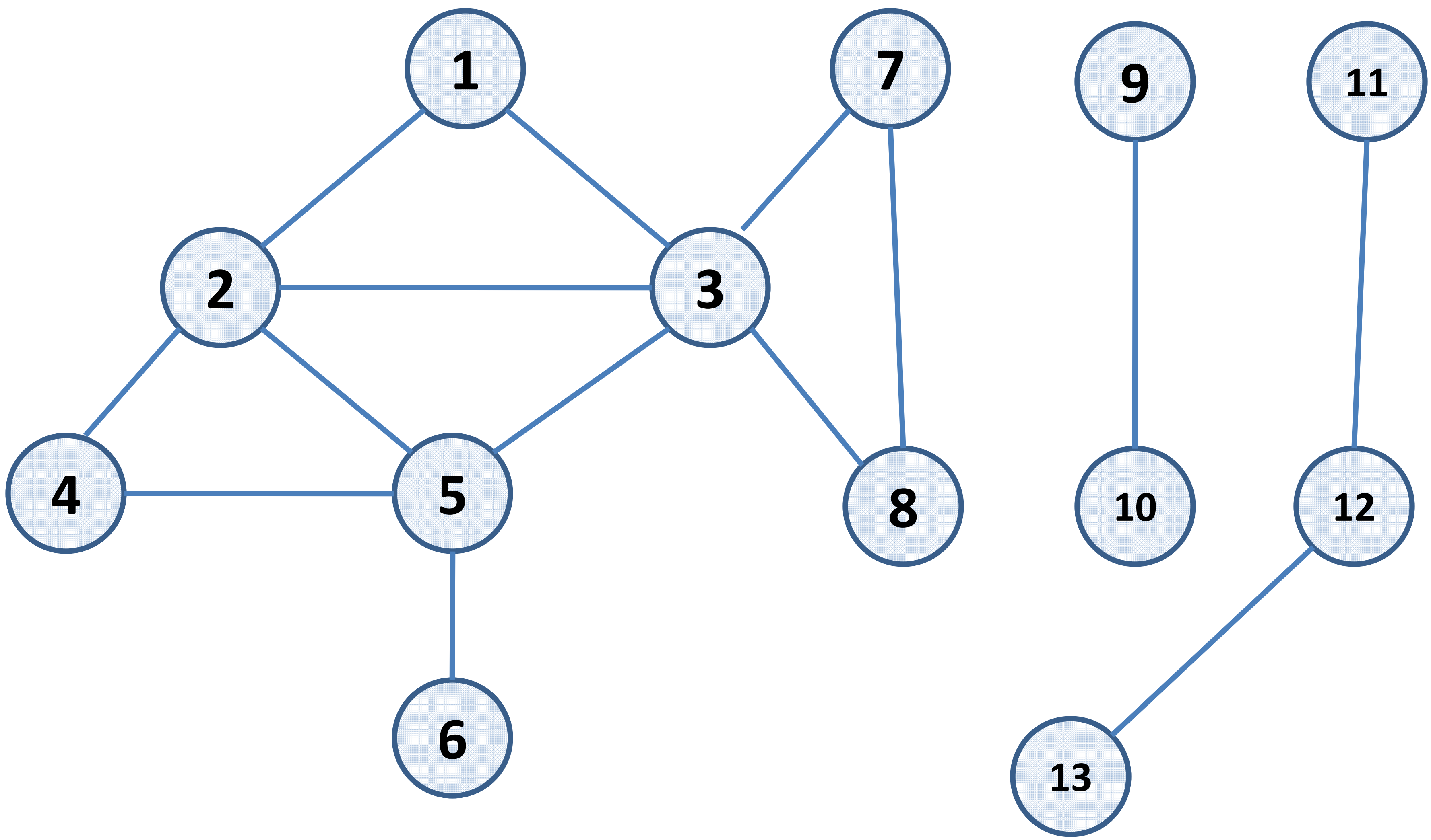
(e)

Graph representations (d) an adjacency matrix, and
(e) an incidence matrix

15

Graph Traversal: breadth first search and depth first search

- Let $G = (V, E)$ be a graph and let s and t be two particular nodes. Is there a path from s to t in G?
- Two high level solutions: breadth first search and depth first search
- *Breadth-first search*:



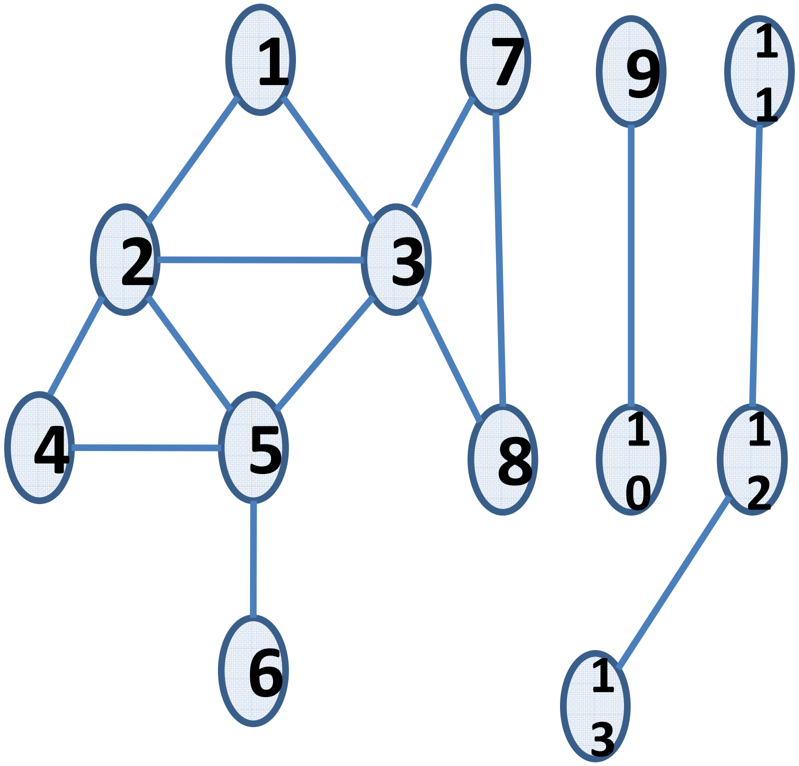
Layers, flooding; more precisely:

Graph traversal: bsf

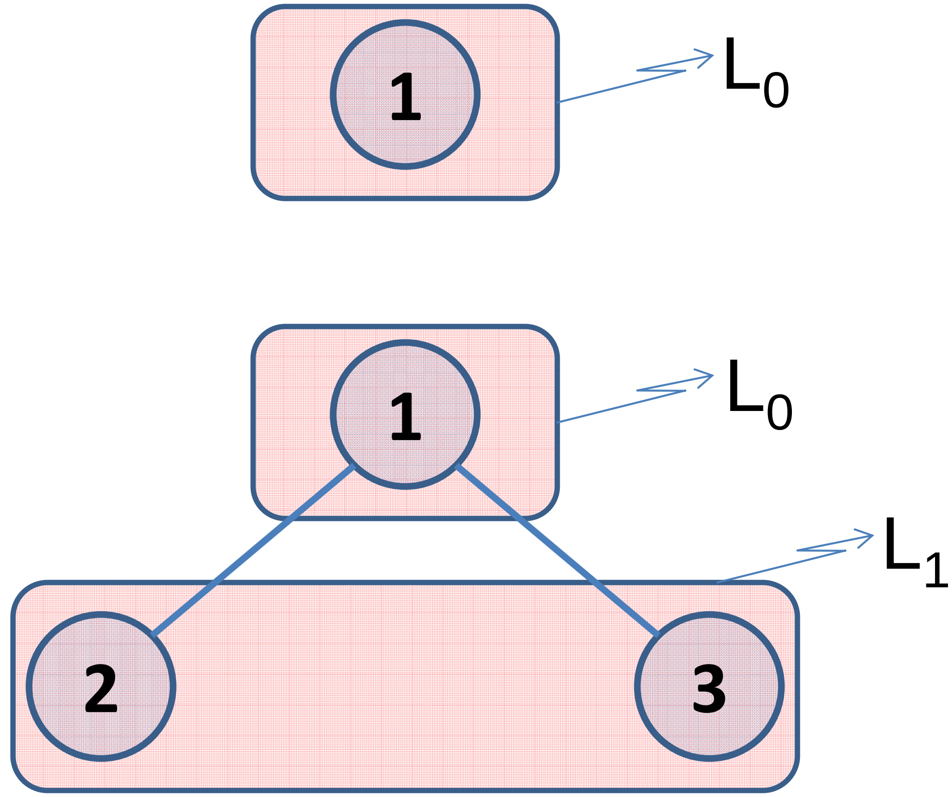
- Define the layers L_1, L_2, L_3, … more precisely
- Layer L_1 consists of all nodes that are neighbors of node s. (Denote the set {s} by L_0)
- Assume we have defined L_1, … , L_j, then layer L_{j+1} consists of all nodes that do not belong to an earlier layer and that have an edge to a node in layer L_j.
- Distance between two nodes: minimum number of edges on a path joining them

Graph traversal: bsf

- *For each $j \geq 1$, layer L_j produced by BFS consists of all nodes at distance exactly **j** from **s**.*
- *There is a path from **s** to **t** if and only if **t** appears in some layer.*
- BFS → a tree T rooted at s on the set of nodes reachable from s. Breadth first search tree.



→Bfs tree starting from node 1 (rooted at node 1).

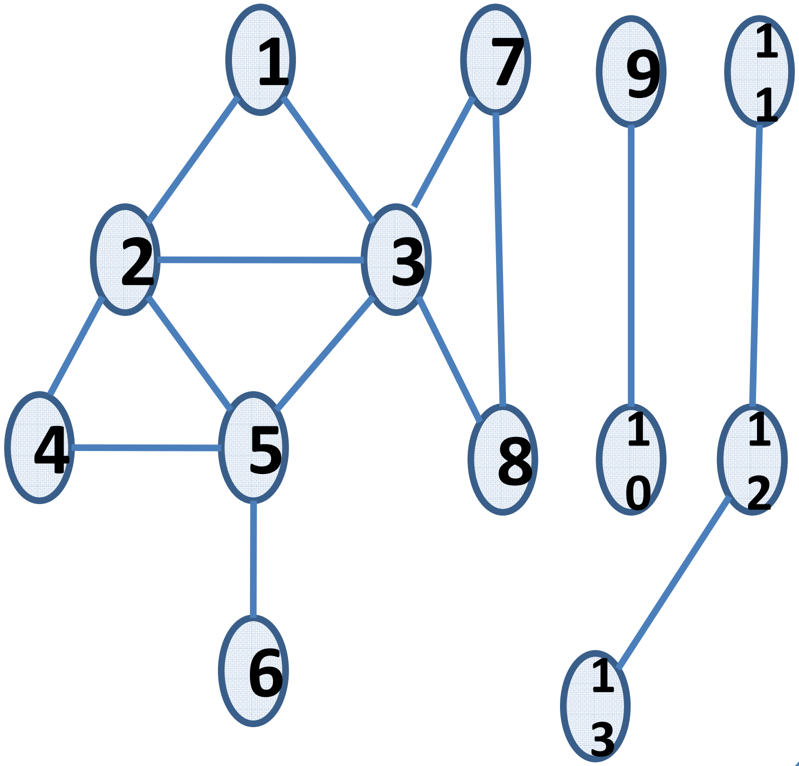


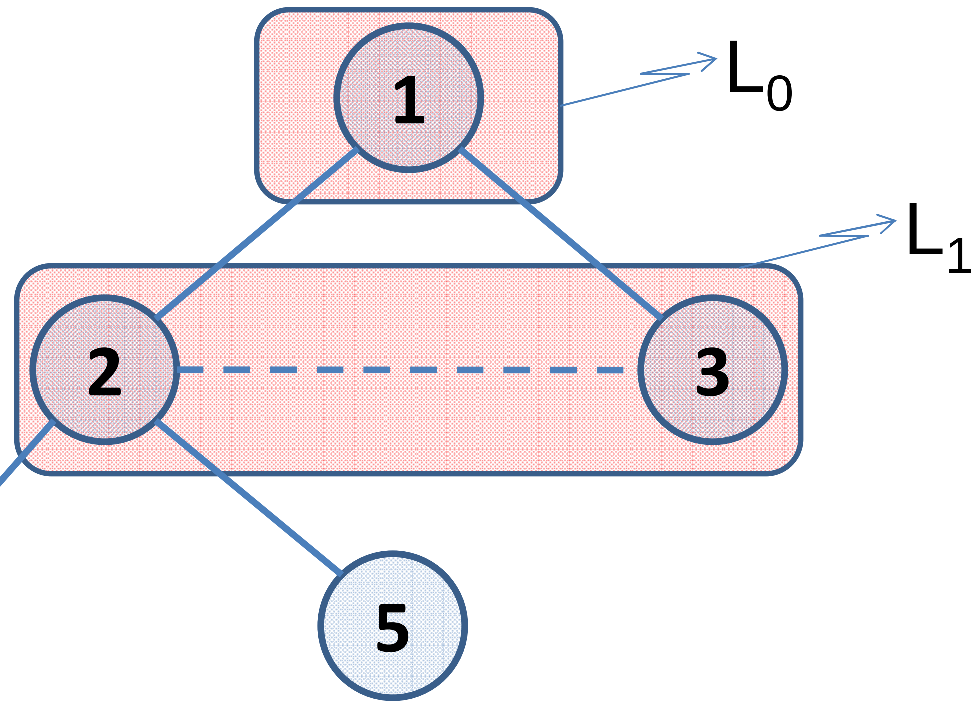
Building up of the layers and BSF tree

We also introduce an array bool discovered[] of size 13
Initialized as follows: discovered[1] = true; discovered[i]=false, for i>1.

After layer L_1 has been built, discovered[2] == true and discovered[3] == true, remaining are still false

→Bfs tree starting from node 1.

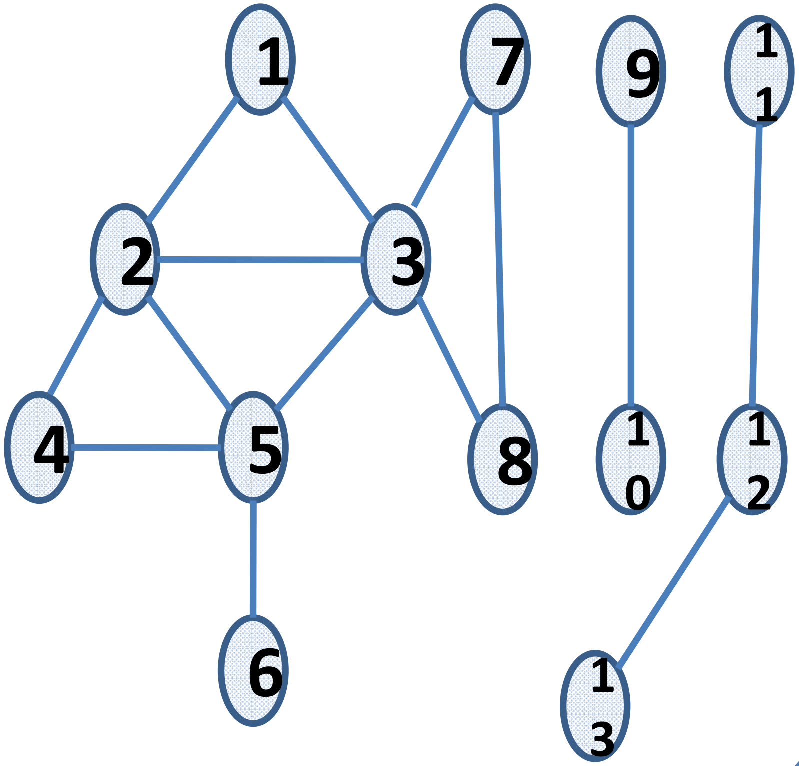




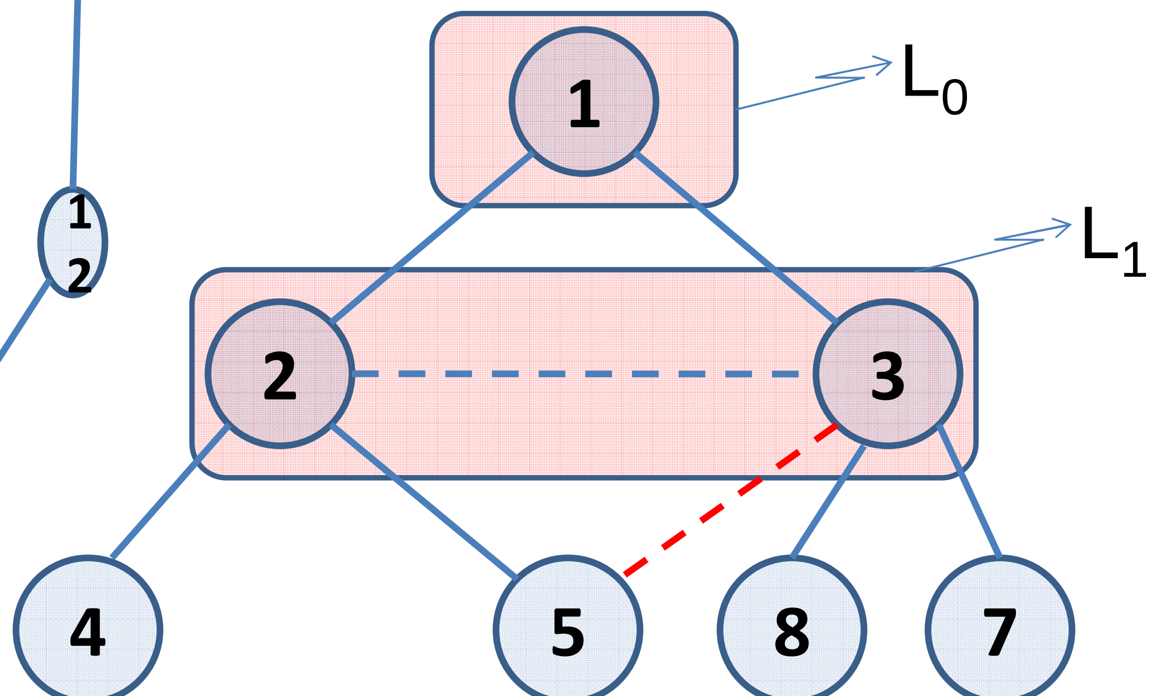
Building up of the layers and BSF tree: the building of layer **L_2**; we start by looking at the edges of node 2: node 3 will not be in layer L_2 since it has been sighted already in layer L_1; for the same reason edge (2,3) will not be part of the bfs-tree; node 4 and 5 are part of L_2, since they have not been sighted yet; for building the bsf-tree it is important to set

discovered[4] = true and discovered[5] = true

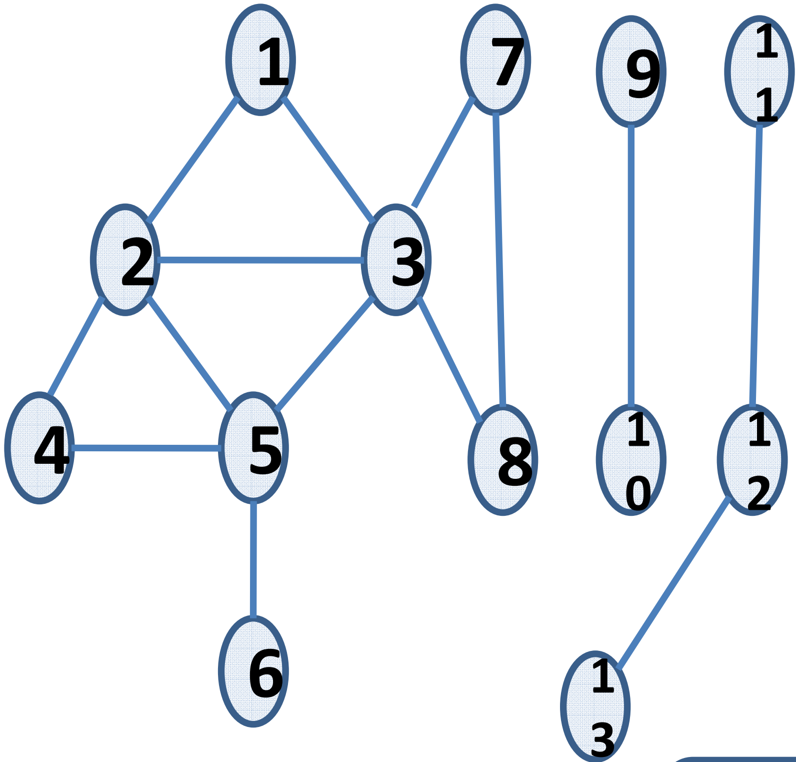
immediately, as we shall on the next slide



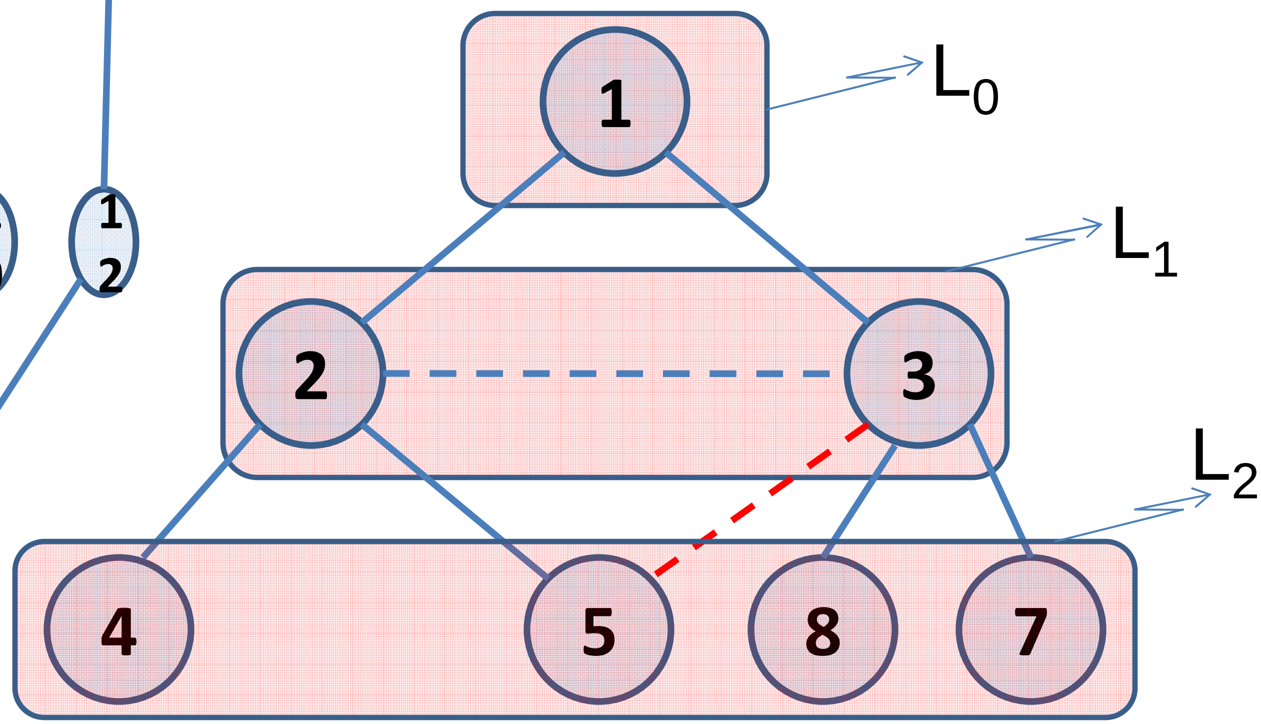
→Bfs tree starting from node 1.

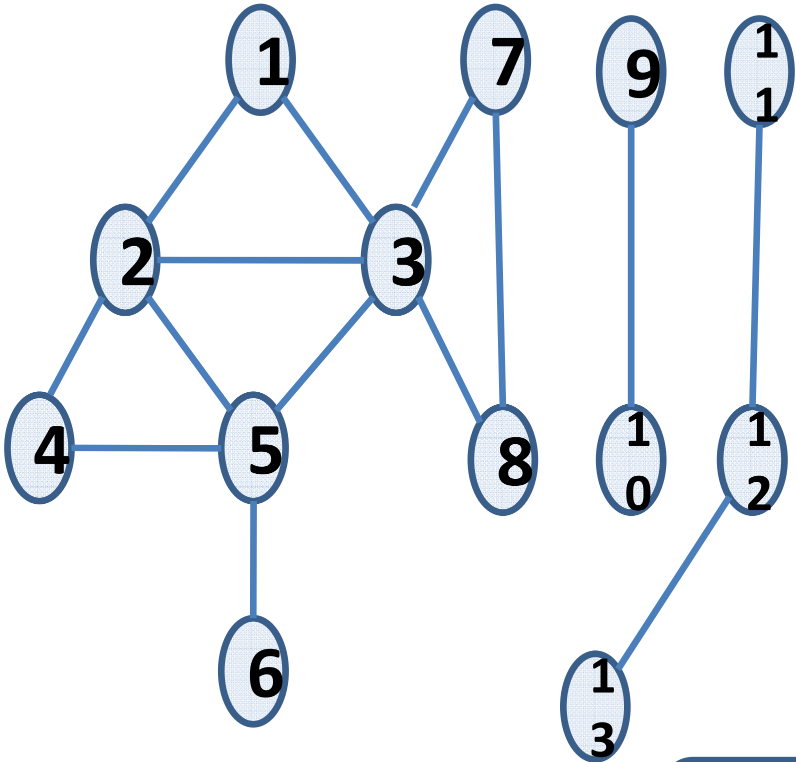


Building up of the layers and BSF tree: the building of layer L_2; we process the next Node in layer L_1: node 3 and look at the edges of this node;
node 5 gets a second reason to be included in layer L_2 but it does not have to be followed up since node 5 is already marked as a sighted node. For the edge (3,5) it is important that node 5 is already marked as sighted, and fortunately does not have to be included in the bsf-tree (it would ***kill*** the ***tree property*** otherwise); nodes 8 and 7 are marked as sighted and included in layer L2, and the edges (3,8)and (3,7) will become part of the bsf-tree . Also the array discovered is updated: discovered[8] = true; discovered[7] = true
Once more: 1) in constructing the bsf-tree it is important to have a sighted node be marked as such IMMEDIATELY 2) for layering the nodes this is not necessary, but it does not hurt to do this either.
NB dashed edges are not included in the bsf-tree; (I marked edge (3,5) in red because it is an example where sighting of a node should be marked without delay.)

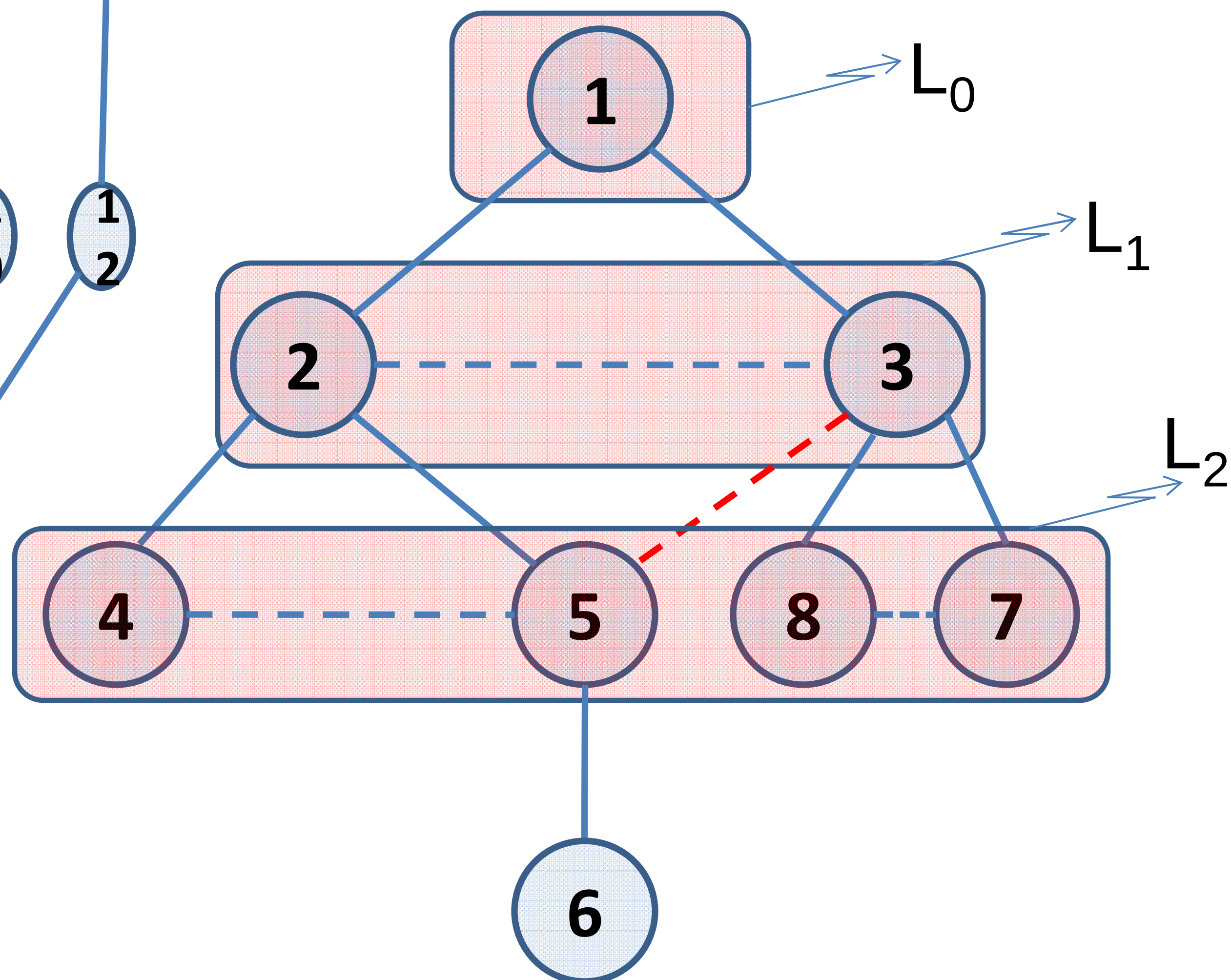


→**Bfs tree starting from node 1.**



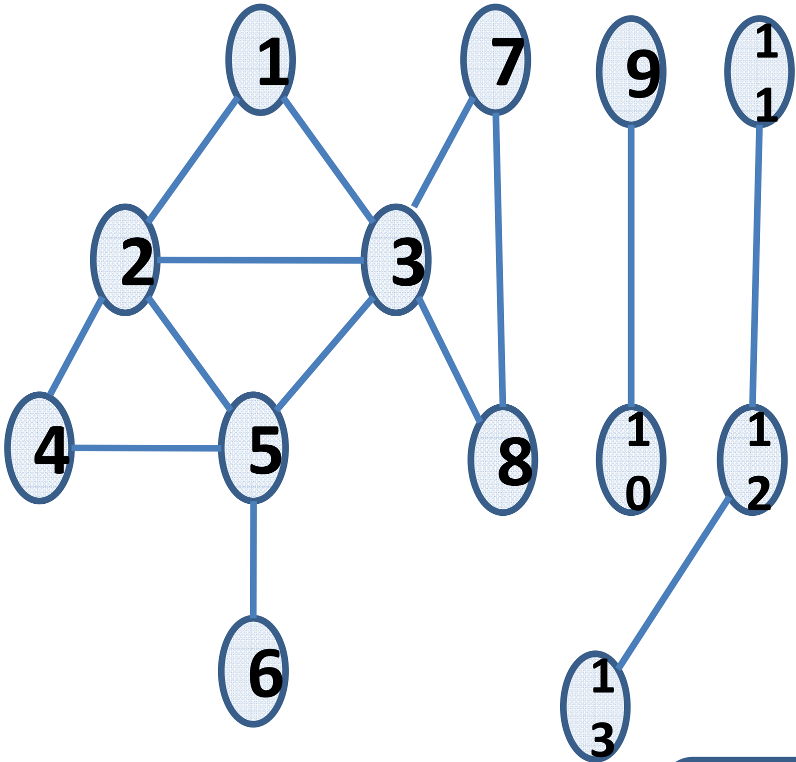


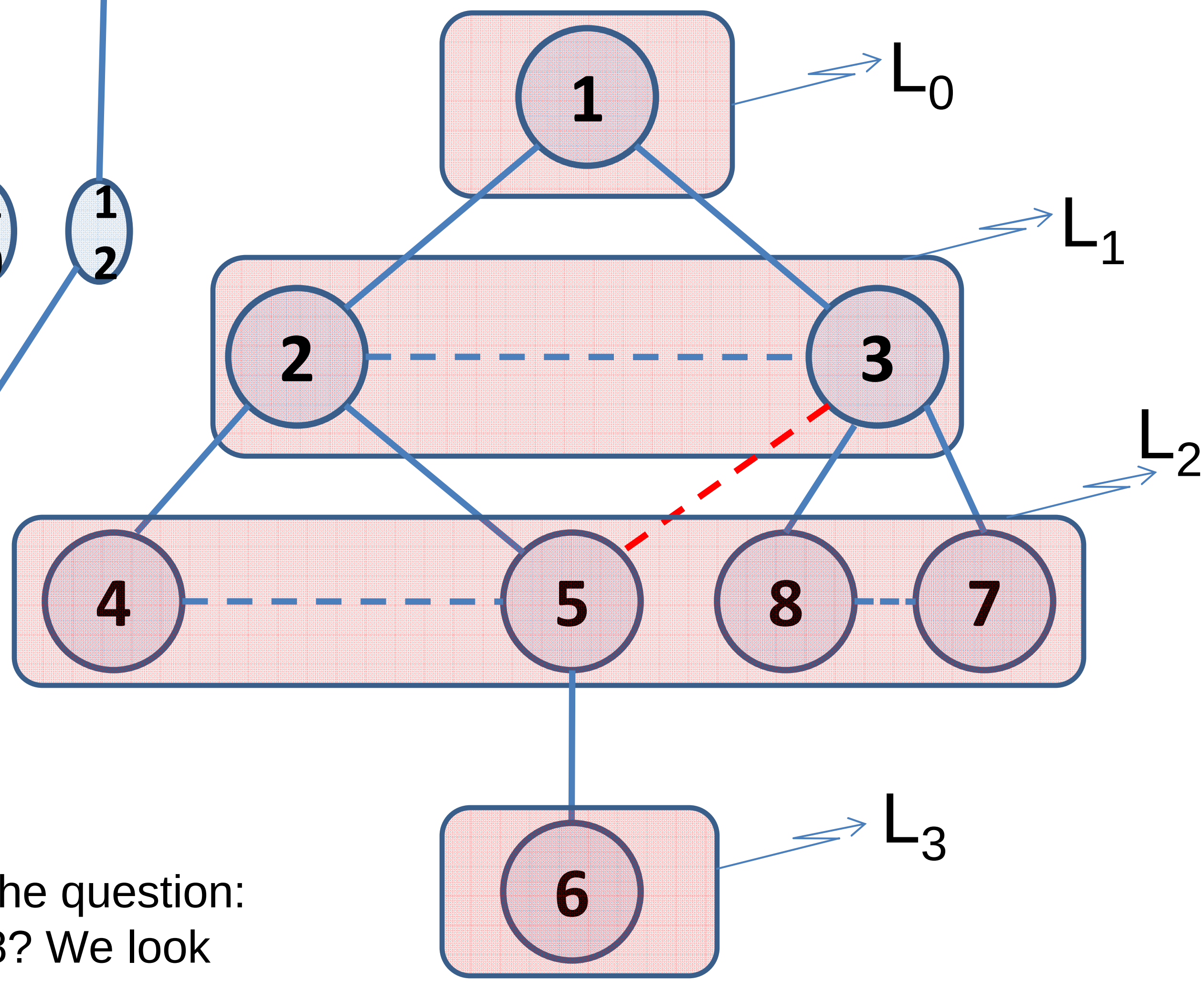
→Bfs tree starting from node 1.



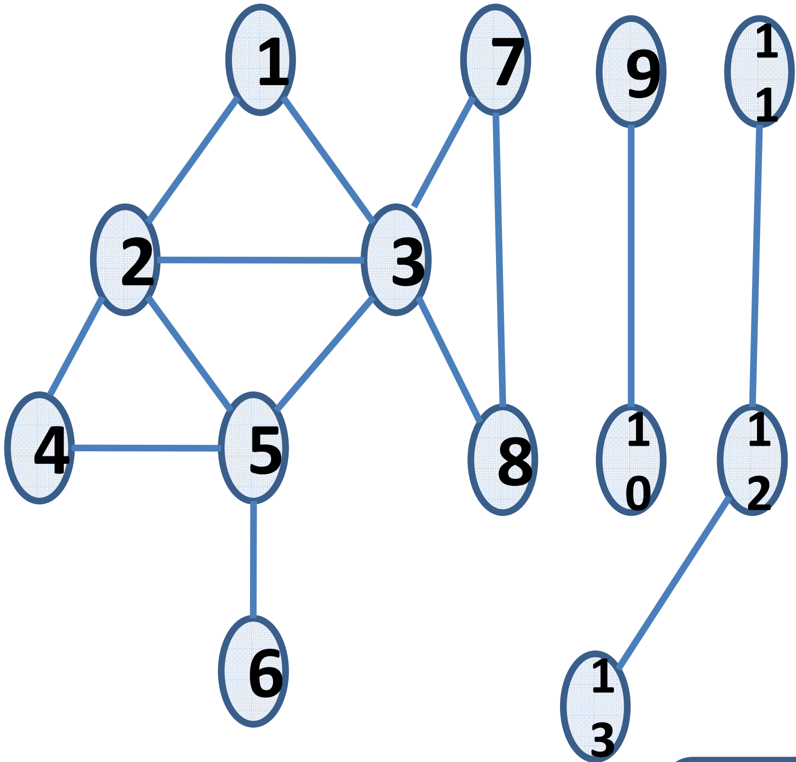
Building up of the layers and BSF tree: the building of layer **L_3**; we process node 4, 5, 8, and 7 in layer L_2: Only node 6 is taken up in layer L_3, others 4,5,8,7 are already sighted; edge (5,6) will also be part of the bsf-tree. The dashed edges are not included.
discovered[6] = true

→**Bfs tree starting from node 1.**

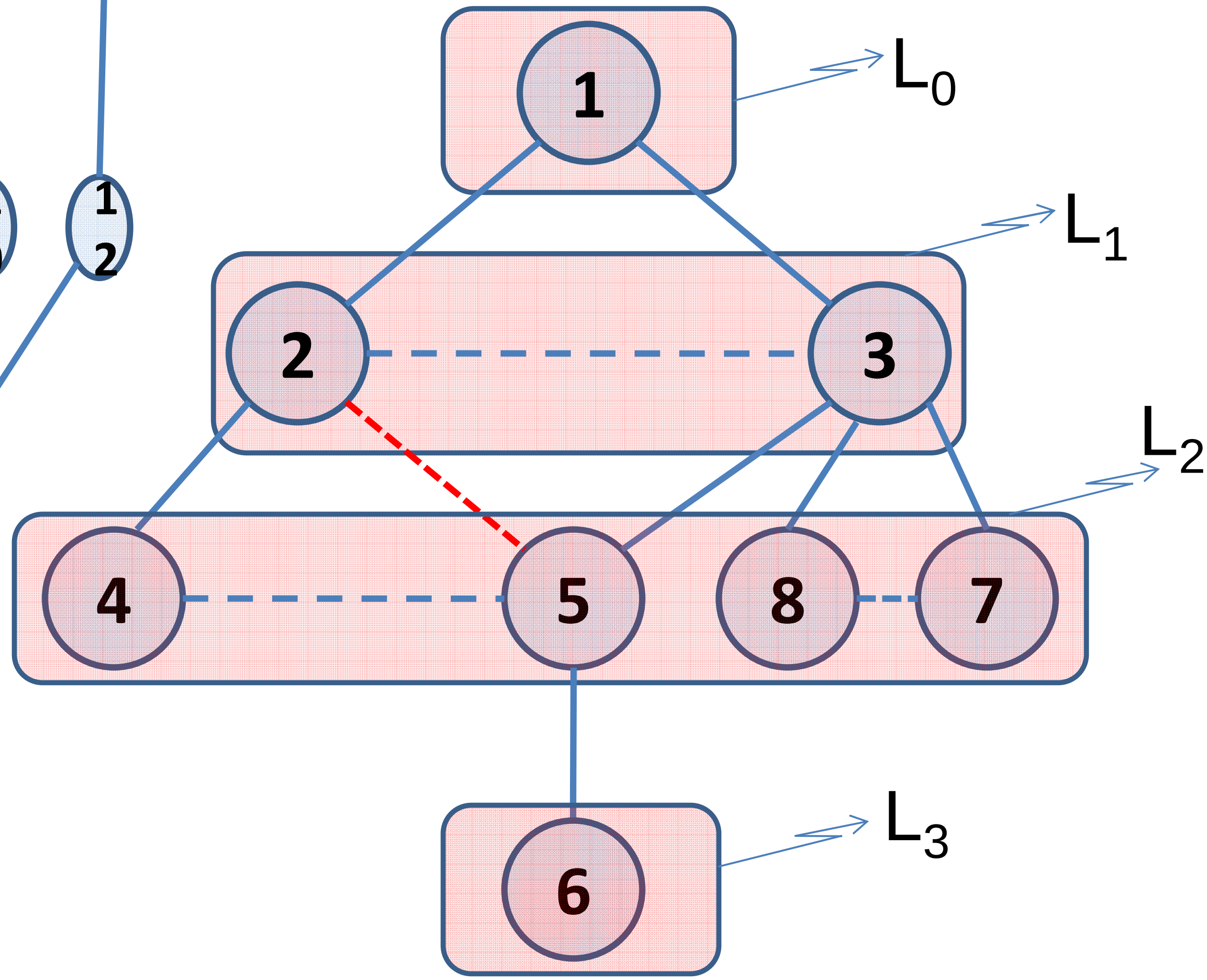




It is now easy to answer the question:
Is there a path from 1 to 8? We look
At the array; discovered[8] is true, thus
There is a path. Discovered[9] is false so
There is no path;



→Another Bfs tree starting from node 1.



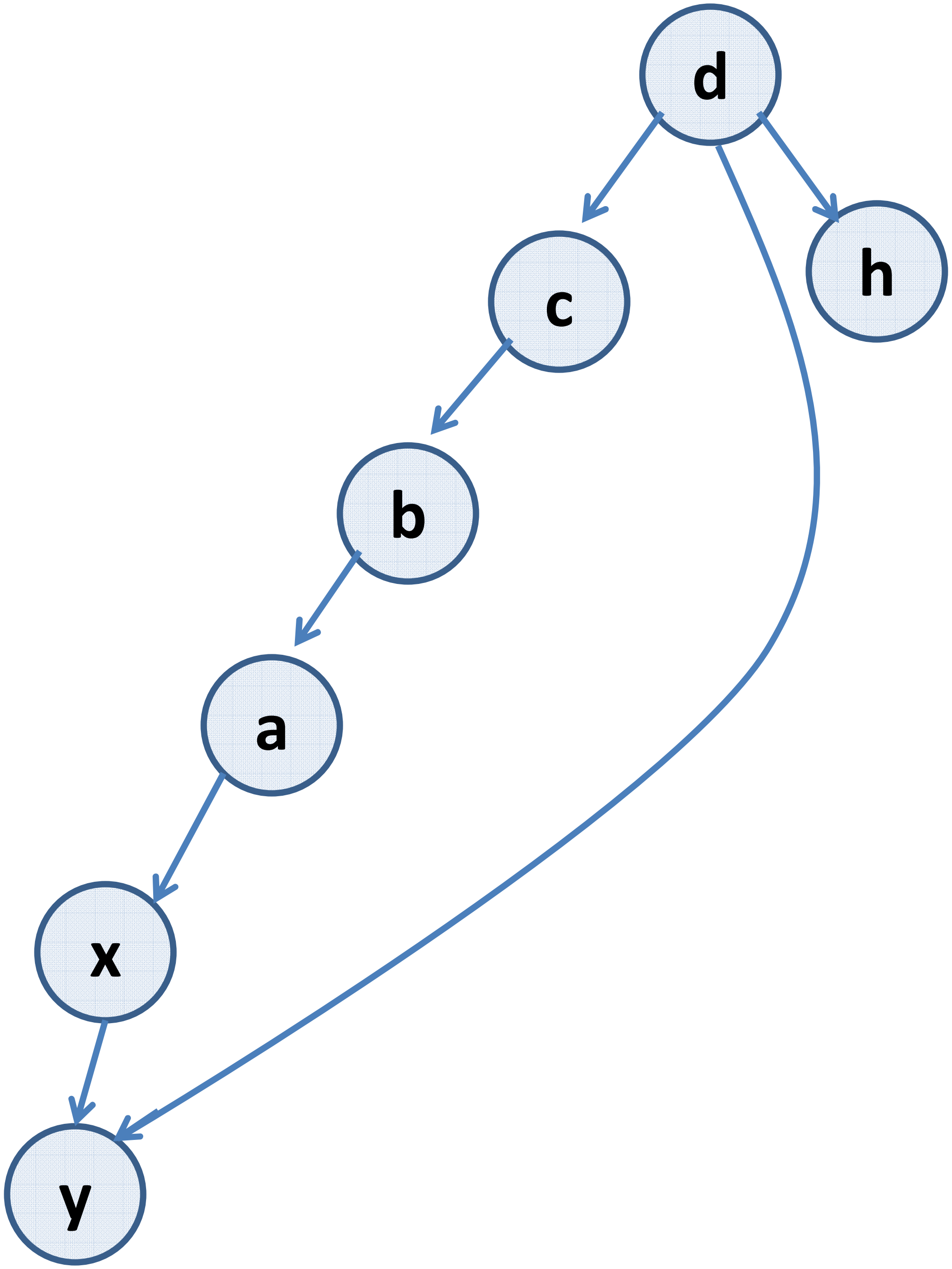
Graph traversal: bsf

- Let T be a breadth-first search tree, let x and y be nodes in T belonging to L_i and L_j, and let (x,y) be an edge of G. Then i and j differ by at most 1.
- Proof: Let x be in L_i . Then it is clear that y is at the latest in L_{i+1} (as (x,y) is an edge of G) or y is in an earlier layer L_k with $k \leq i$; thus $j \leq i+1$. We assume that G is *undirected*: we get by symmetry (since we can consider (y,x) as an edge in G) $i \leq j+1$ (or $i-1 \leq j$). Thus we get

$i-1 \leq j \leq i+1$ (which is equivalent to **$|i-j| \leq 1$** or i and j differ by at most one)

Graph traversal: bsf

- It needs to be stressed that we assumed that the graph G is undirected; for directed graphs the previous statement does not hold: see the following slide for a counter example



Directed graph

BSF for this graph starting in node d:

L0 = { d }
L1 = { c,h,y }
L2 = { b }
L3 = { a }
L4 = { x }

y is in L1
x is in L4
We see that |1-4| = 3 > 1, despite the fact that
there is an edge (x,y).

Graph Traversal: Breadth-first search

Use array discovered[], and for each layer L_i we have a list L[i], i=0, 1, 2, ….

```
BFS(s):
   discovered[s] ← true;
   discovered[v] ← false    // for all other nodes of G
   set layer counter i=0
   initialize L[0] to consist of the single element s
   set current BFS tree T to ø.
   while ( L[i] ≠ ø )
          initialize empty list L[i+1]
             for each node  u ε L[i]
                consider each edge (u,v) incident to u
                if (!discovered[v] ) {
                   discovered[v] ← true;
                   add edge (u,v) to T
                   add v to the list L[i+1]
                }
             endfor
          increment layer counter i
endwhile
```

Can use queue;
Get single list then

Graph Traversal: BFS

The algorithm will visit each node in the connected component of s. In order to visit nodes in the other connected components you need run the above algorithm on a node for which discovered[] is false (by scanning the list after a run of the above algorithm)

Graph Traversal: BFS

- Can implement the algorithm using a single list which is maintained as queue
- Each time a node is discovered it is added to the end of the queue, algorithm will process edges out of the node that is currently first in the queue (see next slide)
- $G=(V,E)$. BFS runs in $O(|E|+|V|)$

Use array discovered[], and for each layer L_i we have a list L[i], i=0, 1, 2, ...

```
BFS(s):
   discovered[s] ← true;
   discovered[v] ← false    // for all other nodes of G
   initialize queue Q to consist of the single element s
   set current BFS tree T to ø.
   while ( Q ≠ ø )
              u=dequeue();
               consider each edge (u,v) incident to u
                if (!discovered[v] ) {
                    discovered[v] ← true;
                    add edge (u,v) to T
                    enqueue(v)
                 }
endwhile
```

Can use queue;
Get single list then

Graph Traversal: Breadth-first search

App of BSF: Testing Bipartiteness

- A graph G=(V,E) is bipartite, if V can be split up into two subsets X and Y such that
  - $X \neq \emptyset$ and $Y \neq \emptyset$
  - V = X U Y
  - Every edge has one end in X and the other in Y.

App of BSF: Testing Bipartiteness

- If a graph G=(V,E) is bipartite, then it cannot contain an odd cycle.
- Containing an odd cycle is the only obstacle to not being bipartite.
- Can assume G is connected (otherwise investigate each connected component separately: each connected component needs to have the bipartite property)
-

App of BSF: Testing Bipartiteness

- Start in arbitrary node and color it red
- Its neighbors blue
- Their neighbors red etc etc. until the whole graph is colored: either we have a valid red-blue coloring of G, in which every edge has ends of opposite colors, or there is an edge with ends of the same color.
- It is essentially the bsf: color L0 red, layer L1 blue, layer L2 red etc, can be implemented on top of bsf: when adding a vertex to an even numbered layer color it red, and when it is added to an odd numbered layer color it blue

App of BSF: Testing Bipartiteness

- Let G be a connected graph, and let L1, L2, … be the layers produced by bsf starting at node s. Then exactly one of the following two things must occur.
  - There is *no* edge of G joining two nodes in the same layer. In this case the graph is bipartite: nodes in even layers are colored red and nodes in odd layers are colored blue.
  - There is an edge of G joining two nodes in the same layer. In this case G, contains an odd length cycle, and so it cannot be bipartite

Graph Traversal: Depth First Search

- Recursive version
- A list (array) which records whether a node has been explored or not: explored[]; a set S of visited nodes;

Initialize: explored[i] ← false, for all i; S ← ø;

```
DSF (s)
   visit(s);
   explored[s] ← true; S← S U {s};
   for ( each v s.t. (s,v) is an edge of the graph G)
      if (!explored[v]) {
         DSF(v)
      }
   endfor  // mark the node as "explored" instead of array
```

Graph Traversal: Depth First Search

- Depth-first search tree of G:
- Initialize: explored[i] ← false, for all i; S ← ø ; T ← ø;

```
DSF (s)
   visit(s);
   explored[s] ← true; S← S U {s};
   for ( each v s.t. (s,v) is an edge of the graph G)
      if (!explored[v]) {
         add edge(s,v) to T
         DSF(v)
      }
   endfor
```

Graph Traversal: Depth First Search, iterative version

- Iterative version of DFS
- A list (array) which records whether a node has been explored or not : list[i].explored

```
DSF (s)
Initialize: list[i].explored ← false, for all i;
Initialize S to be a stack with one element s
  while (S ≠ ø )
     take a node u from S
    if (!explored[u].explored) {
      list[u].explored ← true;
      for (each v s.t. (u,v) edge of G)
         add v to stack S
       endfor
    } // endif
  endwhile
```

- Iterative version includes the building of the *DFS-tree*
- A list (array) which records whether a node has been explored or not and also the parent of the node: list[i].explored, list[i].parent

```
DSF (s)
Initialize: list[i].explored  ← false, for all i;  list[i].parent ← null, , for all i ;
T ← {s} tree; Initialize S to be a stack with one element s
   while  (S ≠ ø )
       take a node u from S
      if (!explored[u]) {
         list[u].explored ← true;
         if (u != s) { add edge (list[u].parent, u) to T}
         for (each v s.t. (u,v) edge of G)
            add v to stack S
            list[v].parent ← u
          endfor
     } // endif
   endwhile
```

Graph traversal for graphs which are not connected

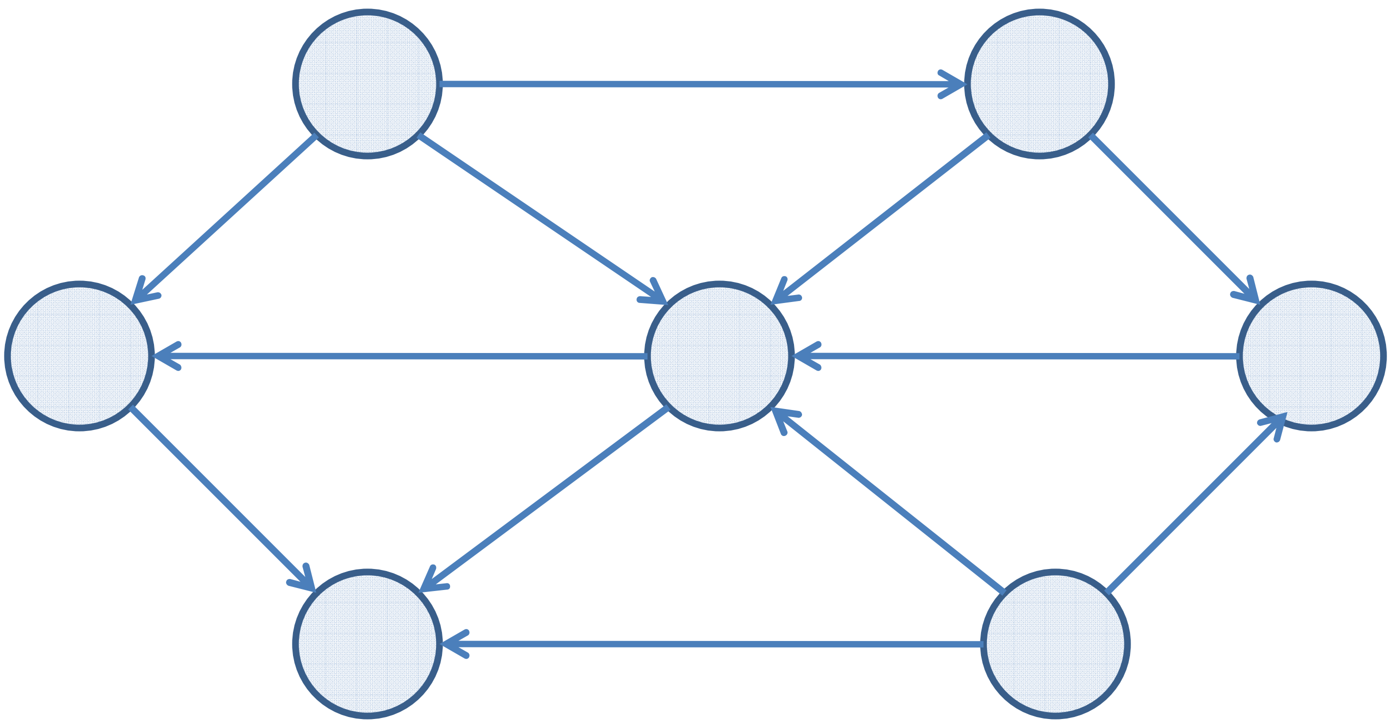
Directed Acyclic Graphs and Topological Ordering

- A directed graph with no cycles is called a **directed acyclic graph** (DAG).
- For instance a node represents a task and a directed edge (i,j) is used to record that job i must be done before job j.
- Precedence relations: given a set of tasks with dependencies it would be natural to seek a valid order in which the tasks could be performed, so that all *dependencies* are respected

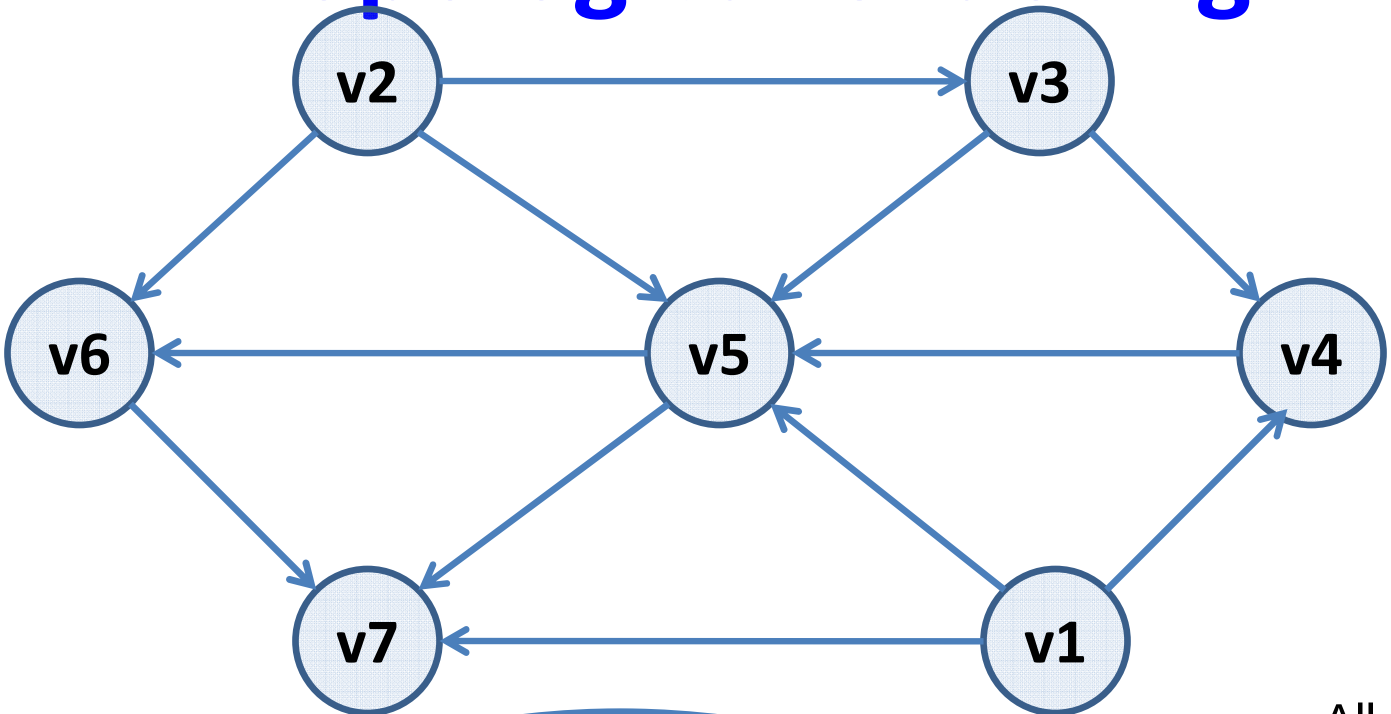
Directed Acyclic Graphs and Topological Ordering

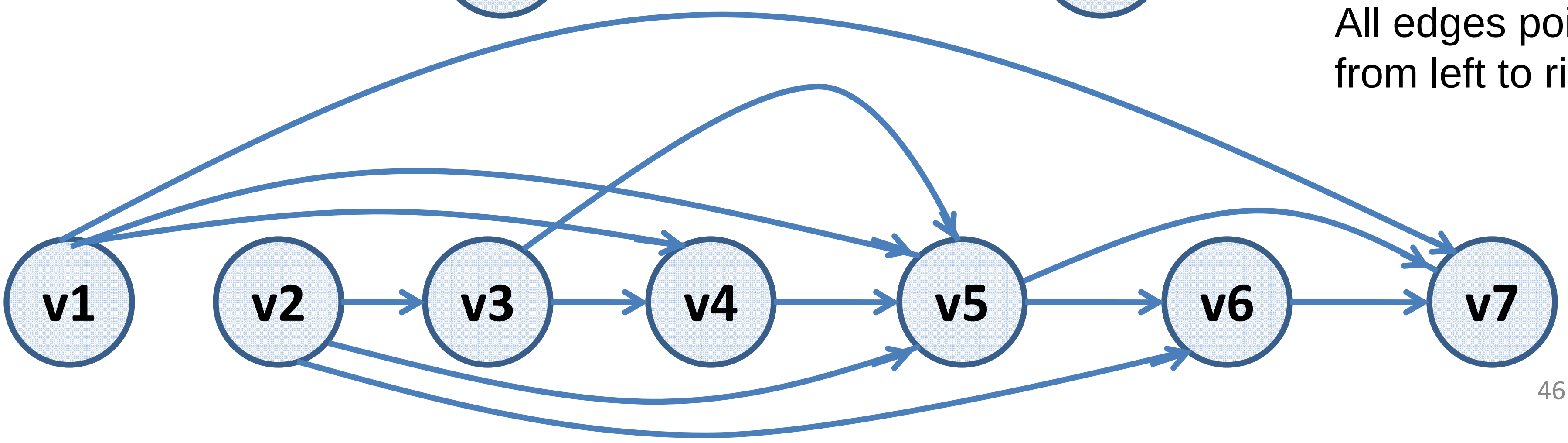
- Definition. Let G be a directed graph. A **topological ordering** of G is an ordering of its nodes as $v_1, v_2, \ldots, v_n$ so that if a pair (v_i, v_j) is an edge of G, then $i<j$.
- a topological ordering on tasks provides an order in which they can be safely performed;

Directed Acyclic Graphs and Topological Ordering



Directed Acyclic Graphs and Topological Ordering





All edges point from left to right

Directed Acyclic Graphs and Topological Ordering

- **Thm.** (G is a directed graph.) If G has a topological ordering, then G is a DAG.
- Proof: Assume G has a topological ordering $v_1, \ldots, v_n$ and G has a cycle. From this we derive a contradiction. Let v_i be the edge on the cycle C with lowest index; now consider v_j on this cycle C which just comes before v_i, in other words (v_j, v_i) is an edge of G; topological sorting implies: $j<i$; on the other hand, i was the lowest index on C: $i<j$; contradiction

Directed Acyclic Graphs and Topological Ordering

- If G has a topological ordering, then G is a DAG.
- The converse of this statement also holds:
- If G is a DAG, then it has a topological ordering.
- Follows from: In every DAG, there is a node v with no incoming edges.
- This latter statement is the basis for an algorithm
- Discussion of efficiency, $O(n^2)$ easily; can achieve $O(m+n)$