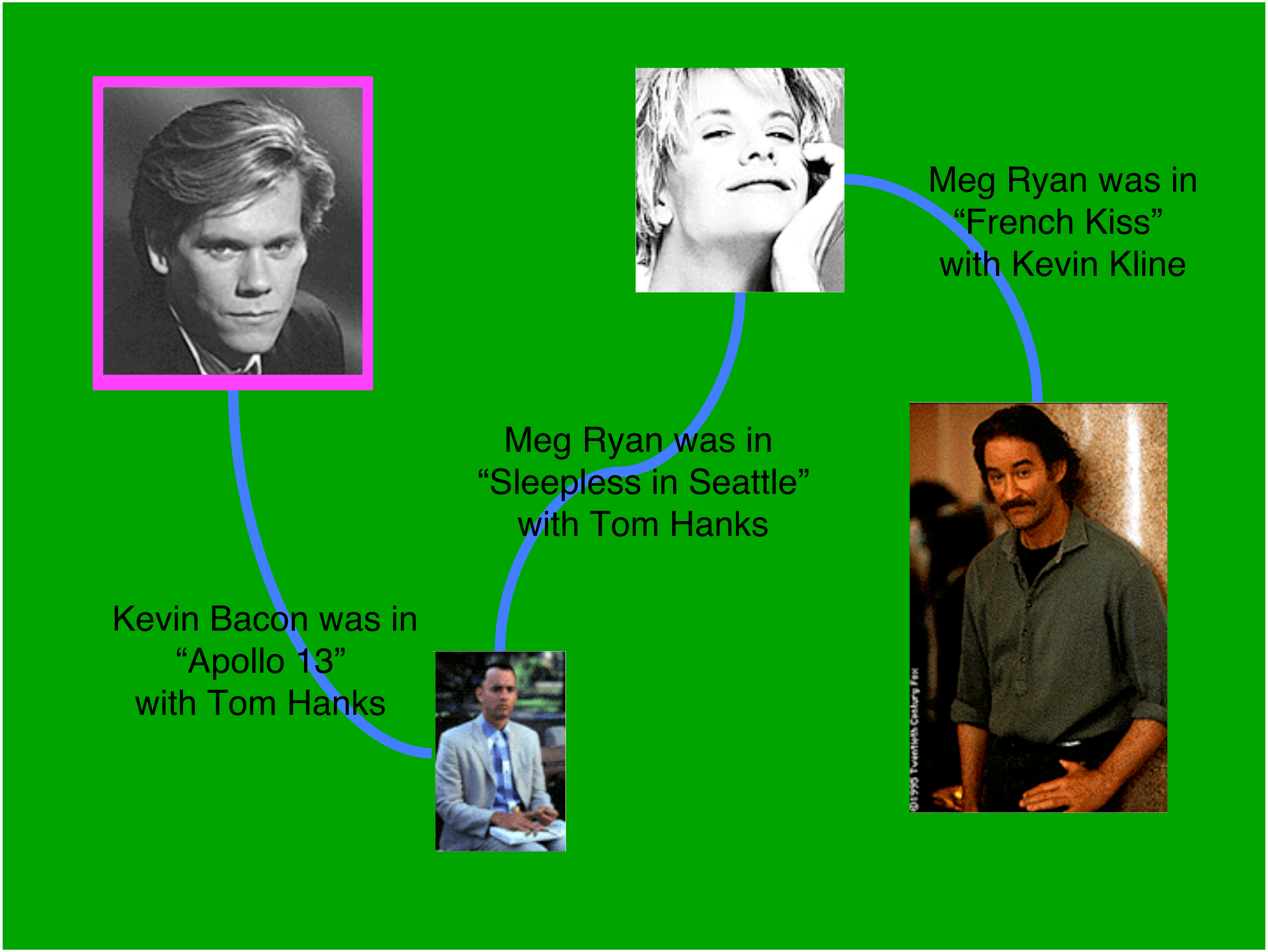
CSE 417: Algorithms and Computational Complexity

Winter 2007

Graphs and Graph Algorithms

Larry Ruzzo


Meg Ryan was in
“French Kiss”
with Kevin Kline
Meg Ryan was in
“Sleepless in Seattle”
with Tom Hanks
Kevin Bacon was in
“Apollo 13”
with Tom Hanks

Objects & Relationships

The Kevin Bacon Game:

- Actors
- Two are related if they've been in a movie together

Exam Scheduling:

- Classes
- Two are related if they have students in common

Traveling Salesperson Problem:

- Cities
- Two are related if can travel *directly* between them

Graphs

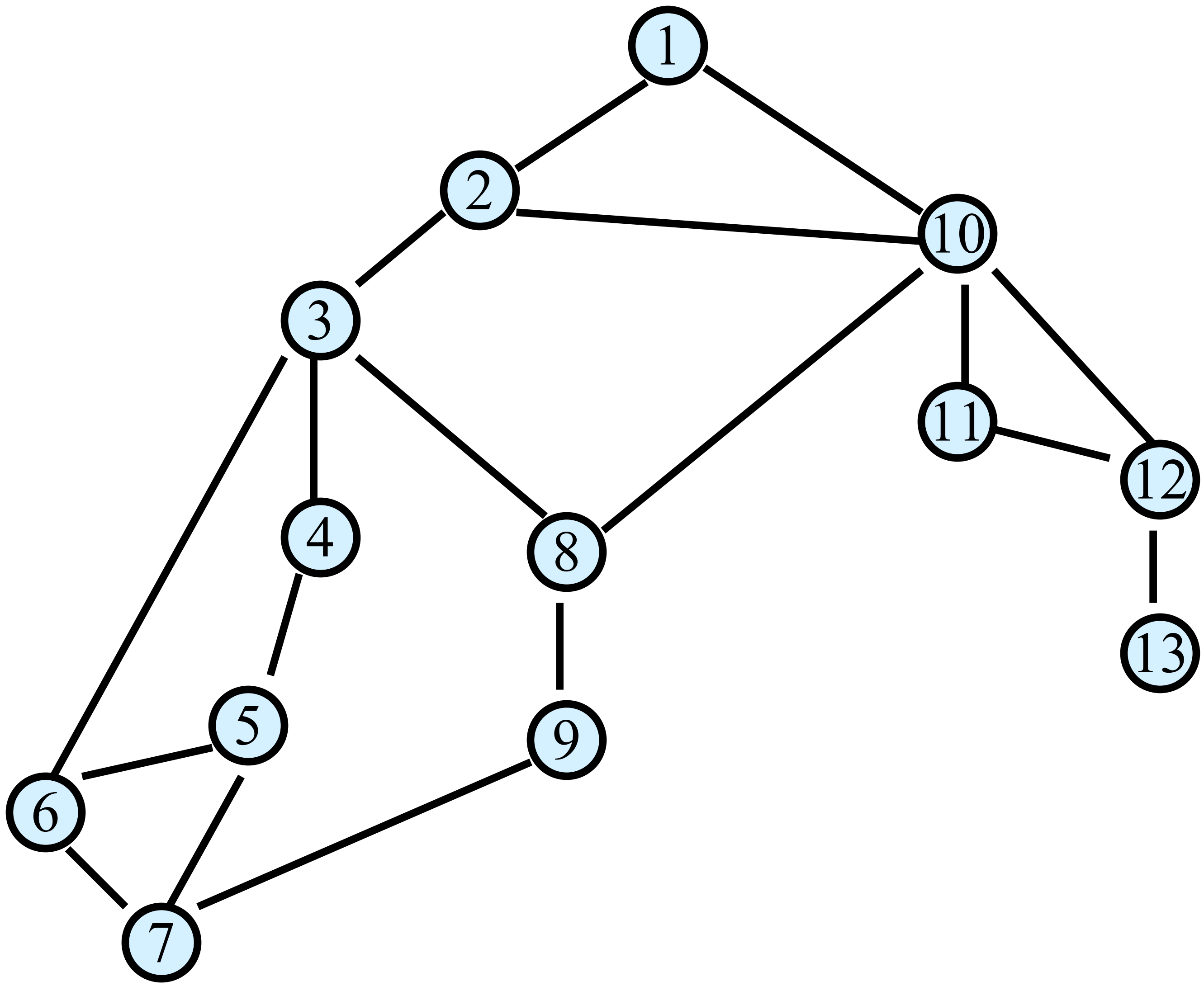
An extremely important formalism for representing (binary) relationships

Objects: "vertices", aka "nodes"

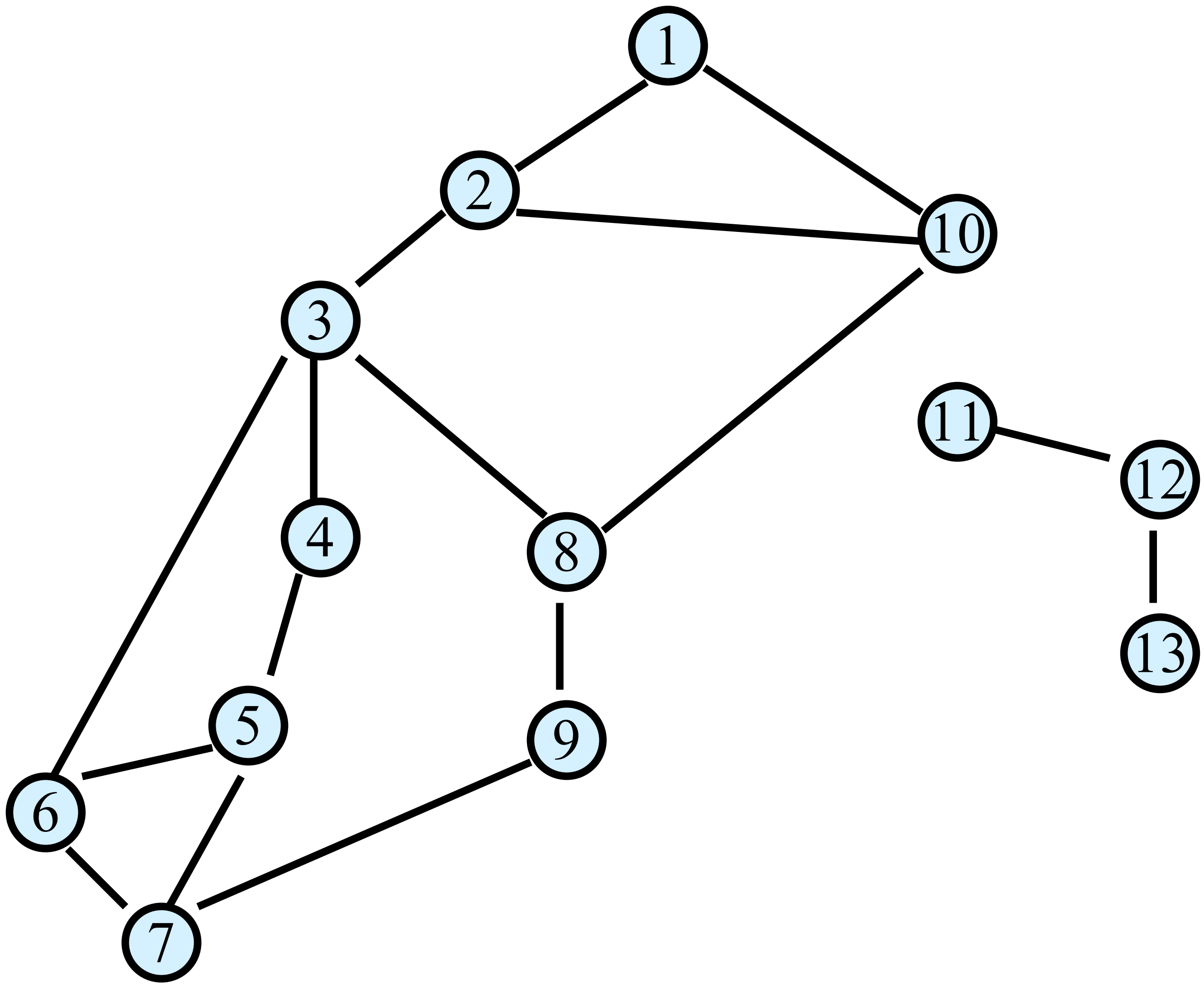
Relationships between pairs: "edges", aka "arcs"

Formally, a graph $G = (V, E)$ is a pair of sets, V the vertices and E the edges

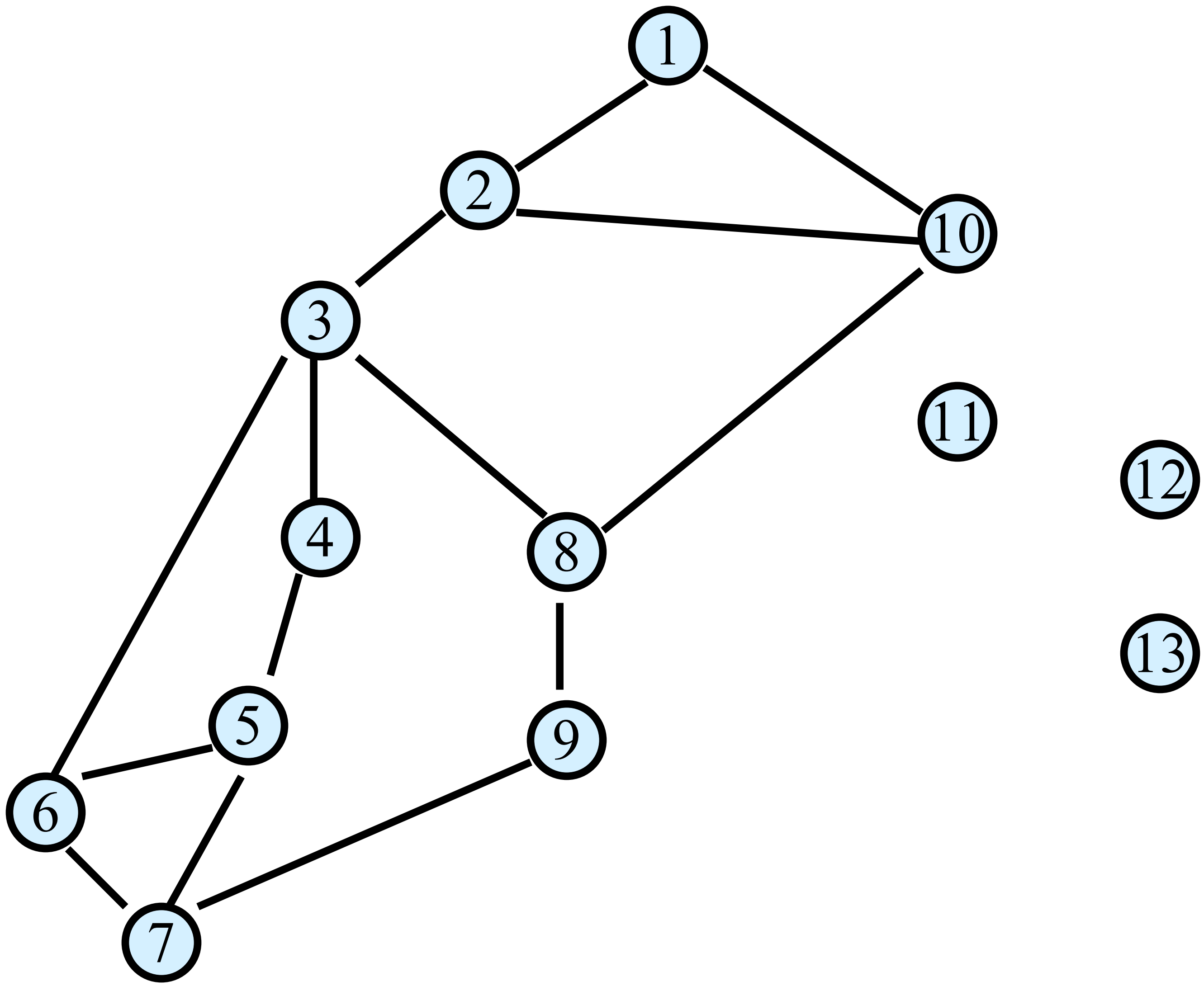
Undirected Graph $G = (V,E)$



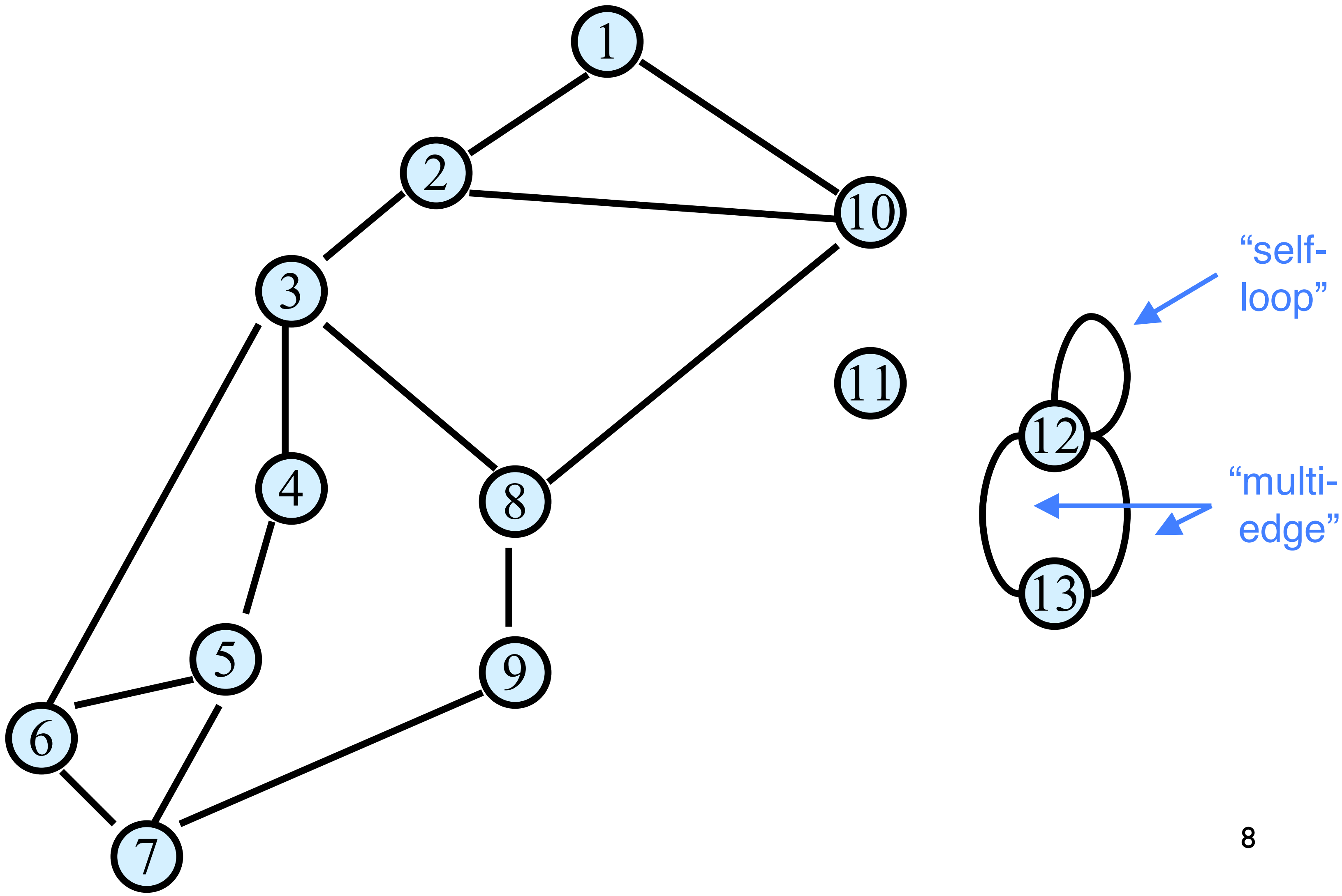
Undirected Graph $G = (V,E)$



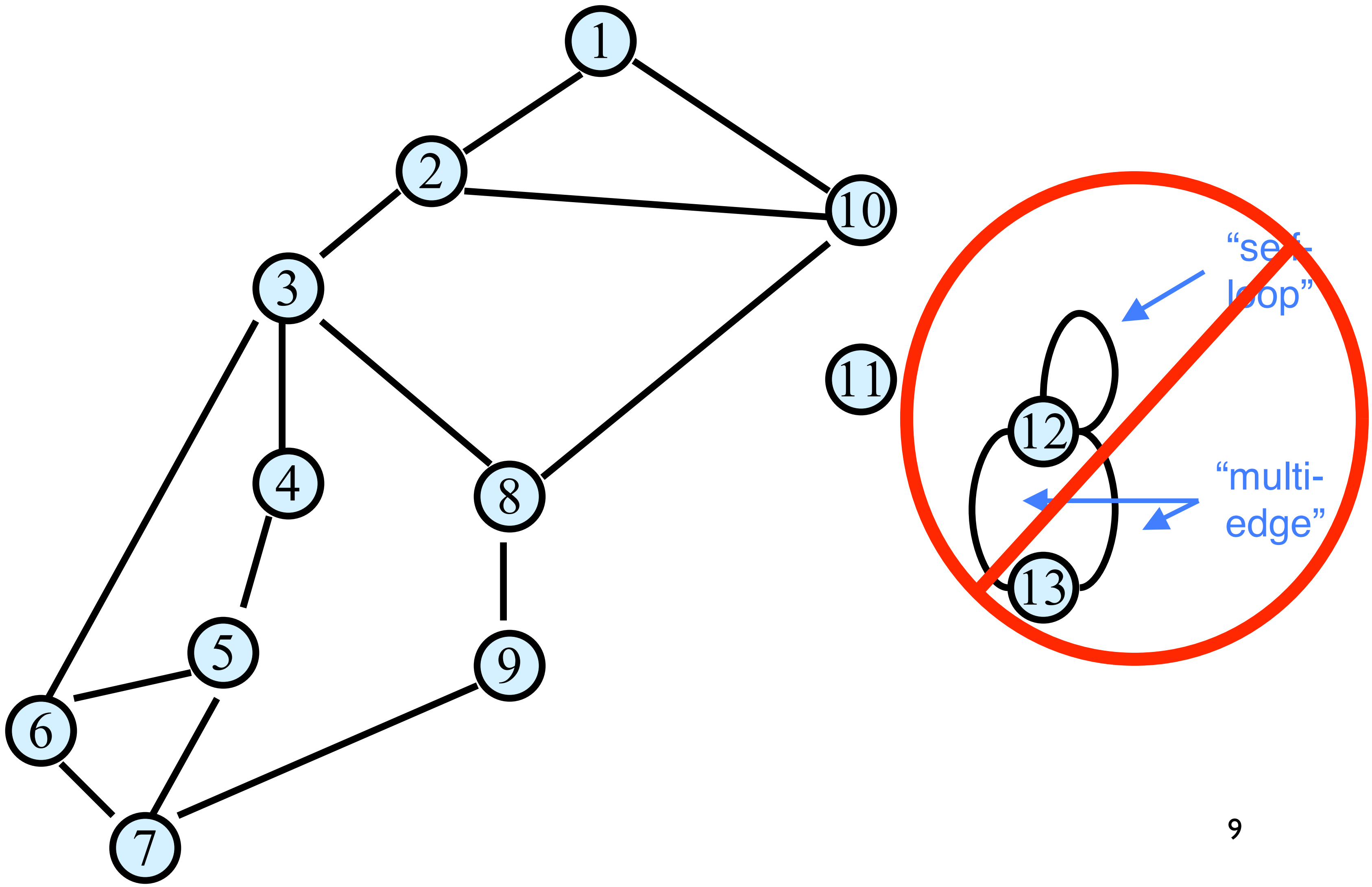
Undirected Graph G = (V,E)



Undirected Graph G = (V,E)

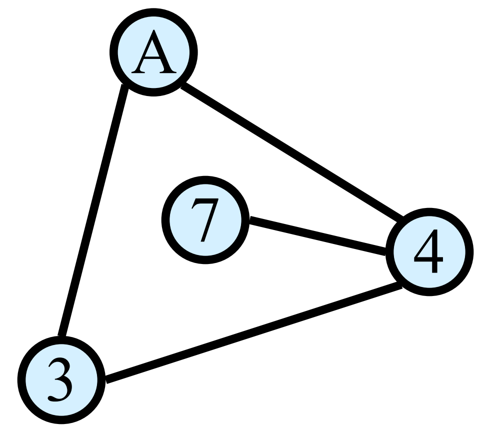


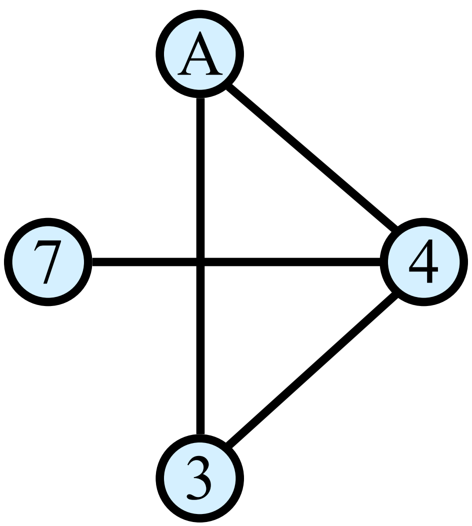
Undirected Graph G = (V,E)

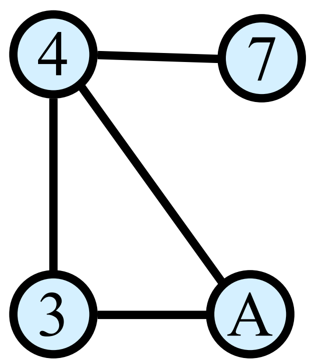


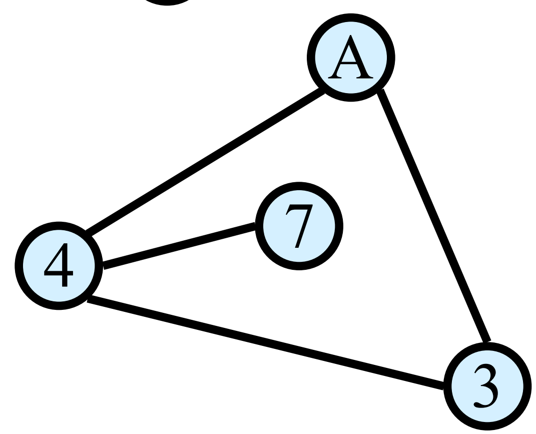
Graphs don’t live in Flatland

Geometrical drawing is mentally convenient, but mathematically irrelevant: 4 drawings, 1 graph.

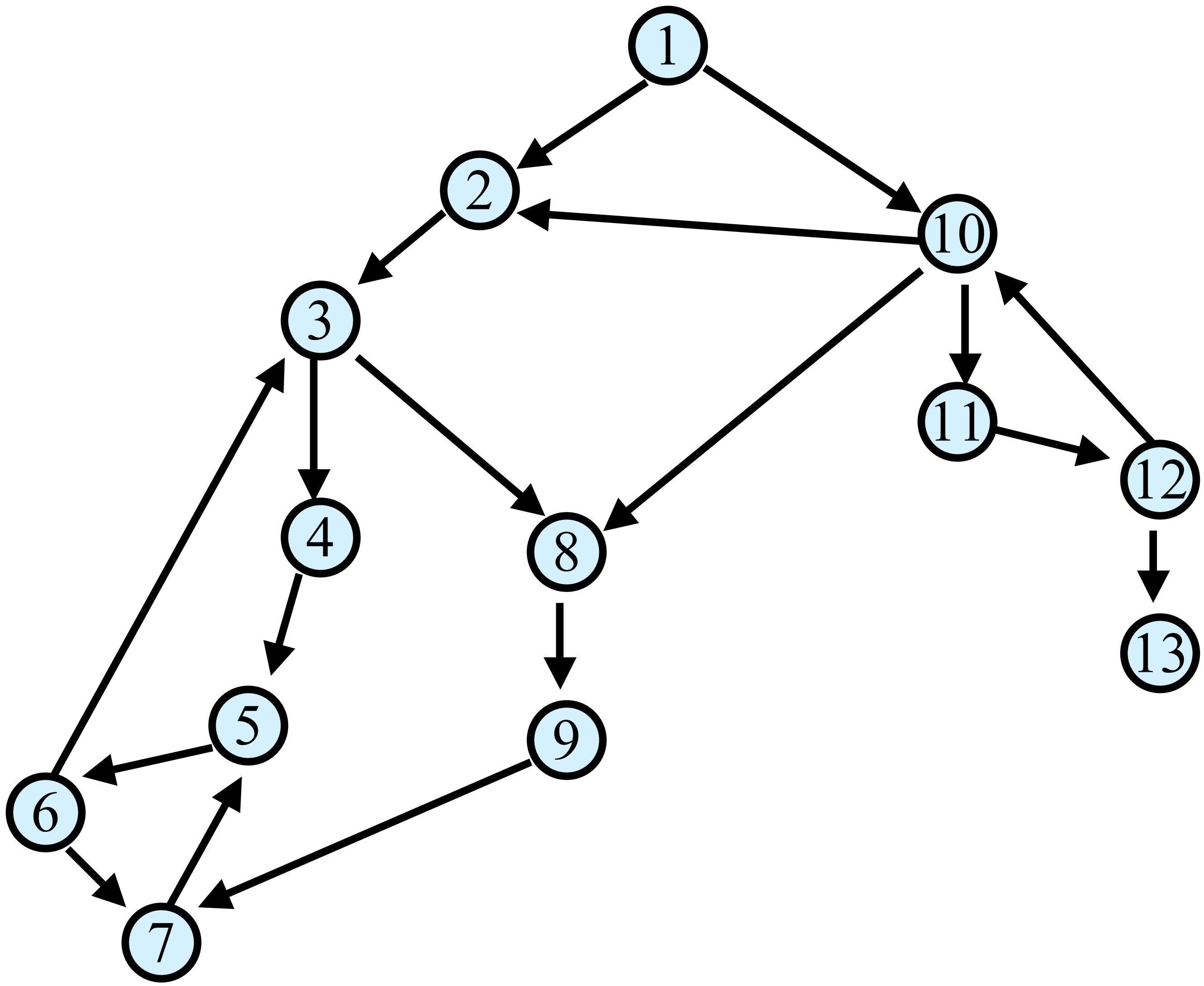




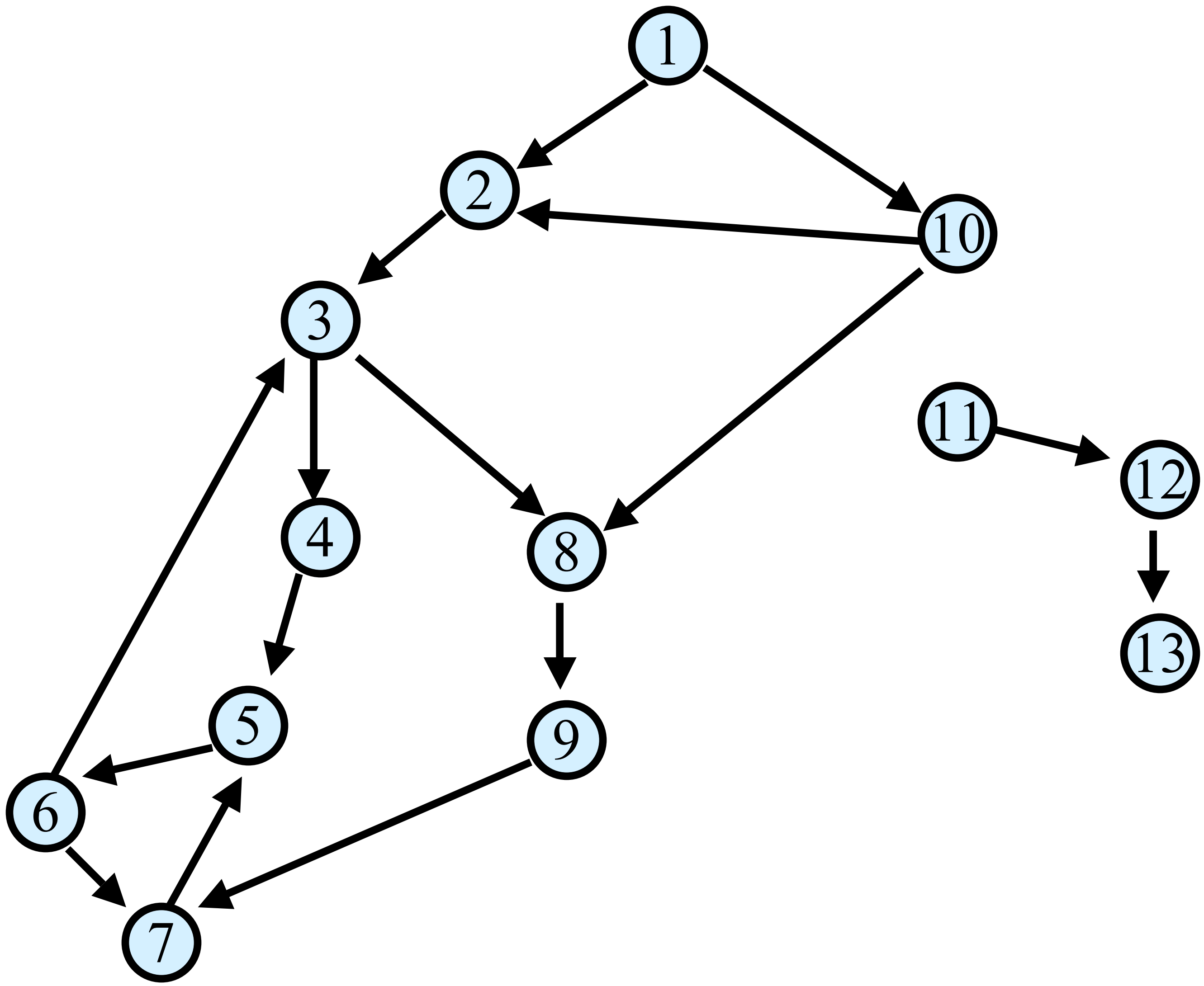




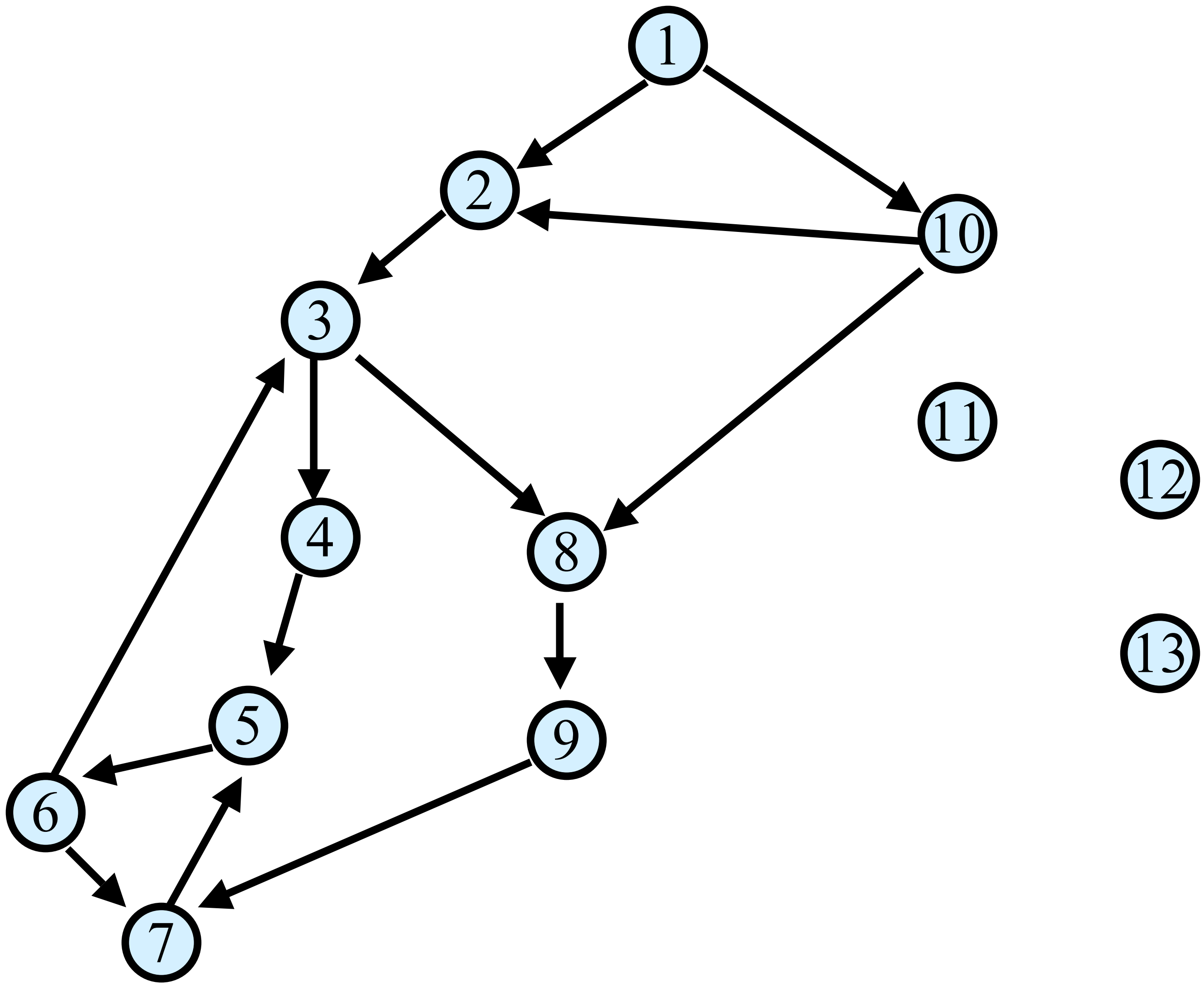
Directed Graph G = (V,E)



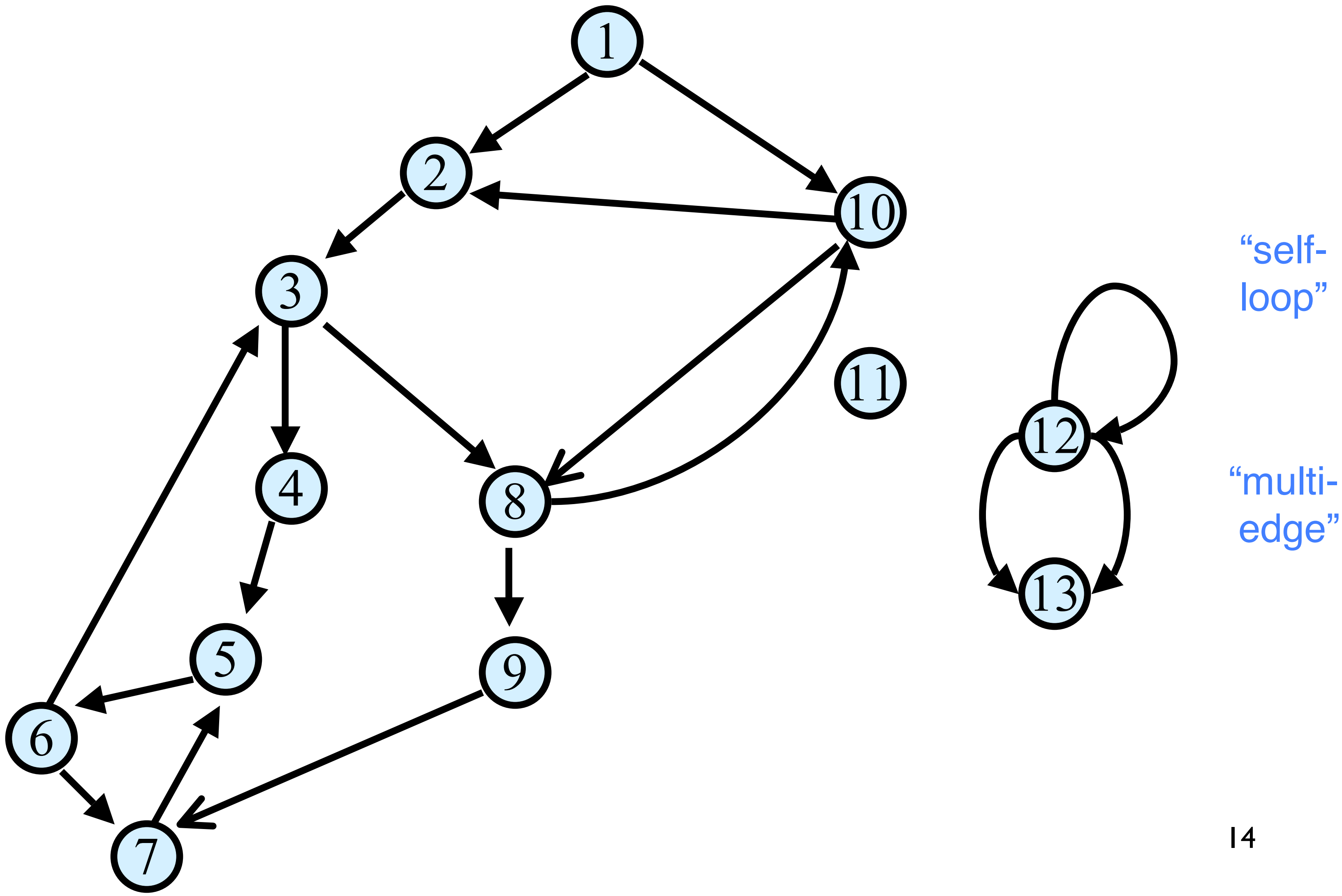
Directed Graph G = (V,E)



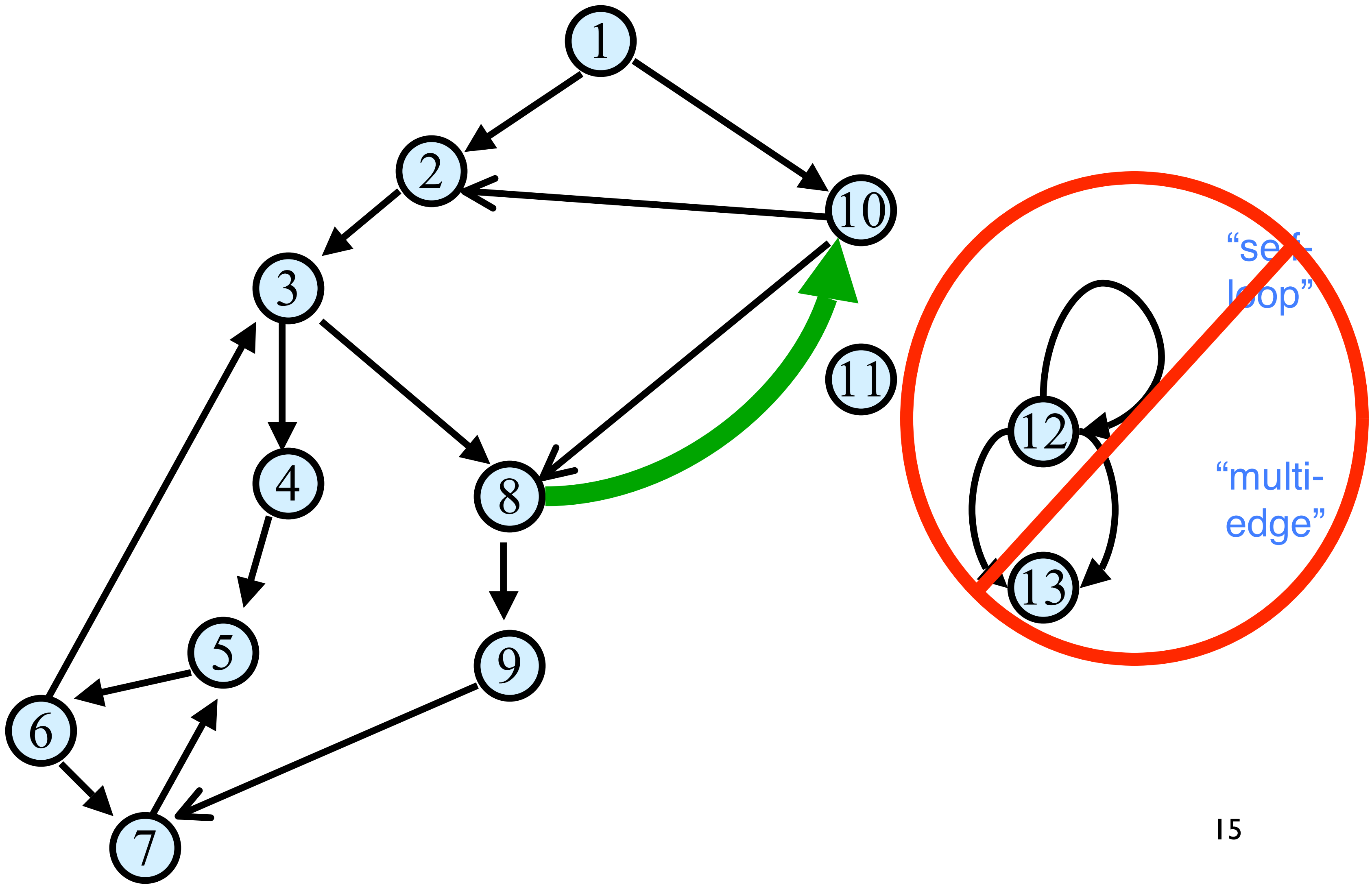
Directed Graph G = (V,E)



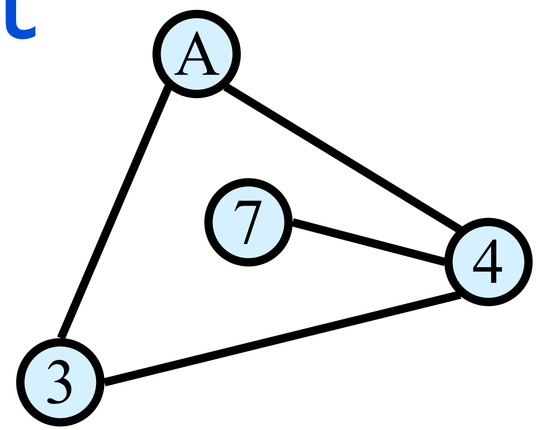
Directed Graph G = (V,E)



Directed Graph G = (V,E)



Specifying undirected graphs as input



What are the vertices?

Explicitly list them:
{“A”, “7”, “3”, “4”}

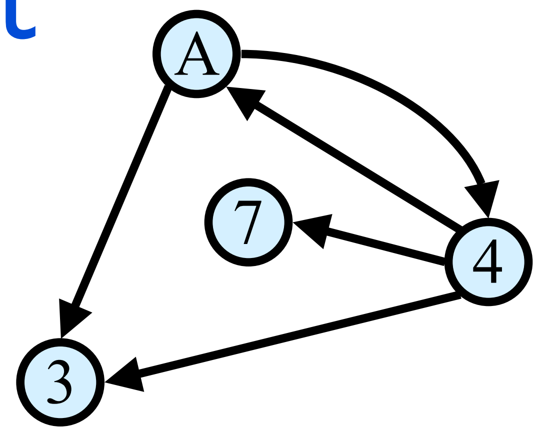
What are the edges?

Either, set of edges
{{A,3}, {7,4}, {4,3}, {4,A}}

Or, (symmetric) adjacency matrix:

$$
\begin{array}{c|cccc}
 & A & 7 & 3 & 4 \\
\hline
A & 0 & 0 & 1 & 1 \\
7 & 0 & 0 & 0 & 1 \\
3 & 1 & 0 & 0 & 1 \\
4 & 1 & 1 & 1 & 0
\end{array}
$$

Specifying directed graphs as input



What are the vertices?

- Explicitly list them: {“A”, “7”, “3”, “4”}

What are the edges?

- Either, set of directed edges: {(A,4), (4,7), (4,3), (4,A), (A,3)}
- Or, (nonsymmetric) adjacency matrix:

	A	7	3	4
A	0	0	1	1
7	0	0	0	0
3	0	0	0	0
4	1	1	1	0

Vertices vs # Edges

Let G be an undirected graph with n vertices and m edges. How are n and m related?

Since

> every edge connects two different vertices (no loops), and no two edges connect the same two vertices (no multi-edges),

it must be true that:

$$\boxed{0 \leq m \leq n(n-1)/2 = O(n^2)}$$

More Cool Graph Lingo

A graph is called *sparse* if $m \ll n^2$, otherwise it is *dense*

- Boundary is somewhat fuzzy; $O(n)$ edges is certainly sparse, $\Omega(n^2)$ edges is dense.

Sparse graphs are common in practice

- E.g., all planar graphs are sparse ($m \le 3n-6$, for $n \ge 3$)

Q: which is a better run time, $O(n+m)$ or $O(n^2)$?

A: $O(n+m) = O(n^2)$, but $n+m$ usually way better!

Representing Graph $G = (V,E)$

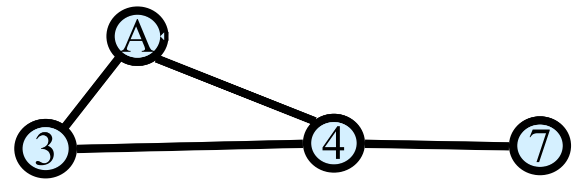
internally, indp of input format

Vertex set $V = \{v_1, \ldots, v_n\}$

Adjacency Matrix A

$A[i,j] = 1$ iff $(v_i,v_j) \in E$

Space is n^2 bits



	A	7	3	4
A	0	0	1	1
7	0	0	0	1
3	1	0	0	1
4	1	1	1	0

Advantages:

$O(1)$ test for presence or absence of edges.

Disadvantages: inefficient for sparse graphs, both in storage and access

$m << n^2$

Representing Graph G=(V,E)

n vertices, m edges

Adjacency List:

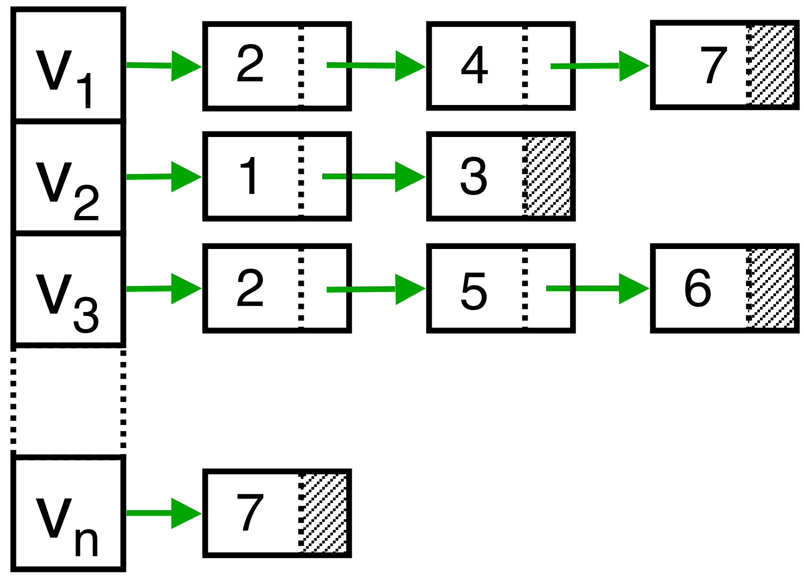
- O(n+m) words

Advantages:

- Compact for sparse graphs
- Easily see all edges

Disadvantages

- More complex data structure
- no O(1) edge test

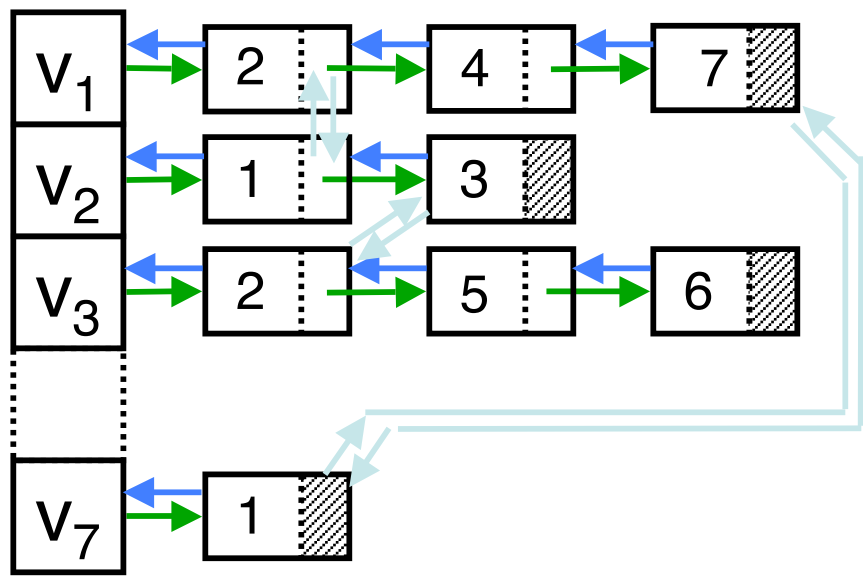


Representing Graph G=(V,E)

n vertices, m edges

Adjacency List:

O(n+m) words



Back- and cross pointers more work to build, but allow easier traversal and deletion of edges, *if needed*, (don’t bother if not)

Graph Traversal

Learn the basic structure of a graph

"Walk," *via edges*, from a fixed starting vertex s to all vertices reachable from s

Being *orderly* helps. Two common ways:

- Breadth-First Search
- Depth-First Search

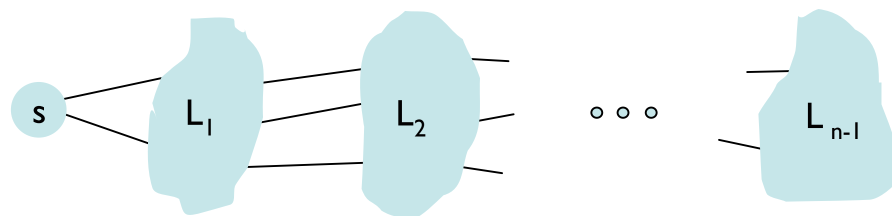
Breadth-First Search

Completely explore the vertices in order of their distance from s

Naturally implemented using a queue

Breadth-First Search

Idea: Explore from s in all possible directions, layer by layer.



BFS algorithm.

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

Theorem. For each i, L_i consists of all nodes at distance (i.e., min path length) exactly i from s.

Cor: There is a path from s to t iff t appears in some layer.

Graph Traversal: Implementation

Learn the basic structure of a graph

"Walk," via edges, from a fixed starting vertex *s* to all vertices reachable from *s*

Three states of vertices

- *undiscovered*
- *discovered*
- *fully-explored*

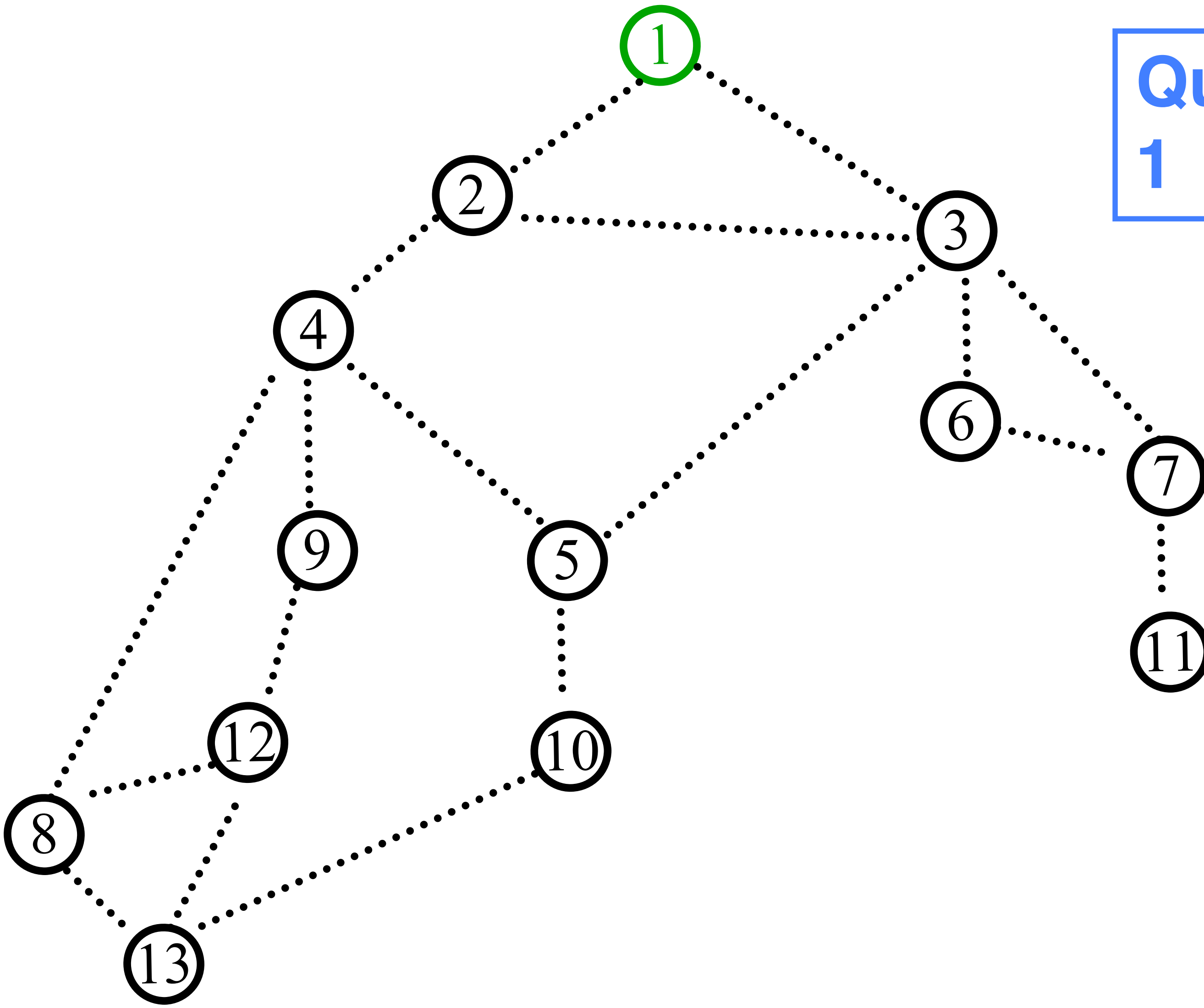
BFS(s) Implementation

Global initialization: mark all vertices "undiscovered"

```
BFS(s)
    mark  s "discovered"
    queue = { s }
    while queue not empty
        u = remove_first(queue)
        for each edge {u,x}
            if (x is undiscovered)
                mark x discovered
                append x on queue
        mark u fully explored
```

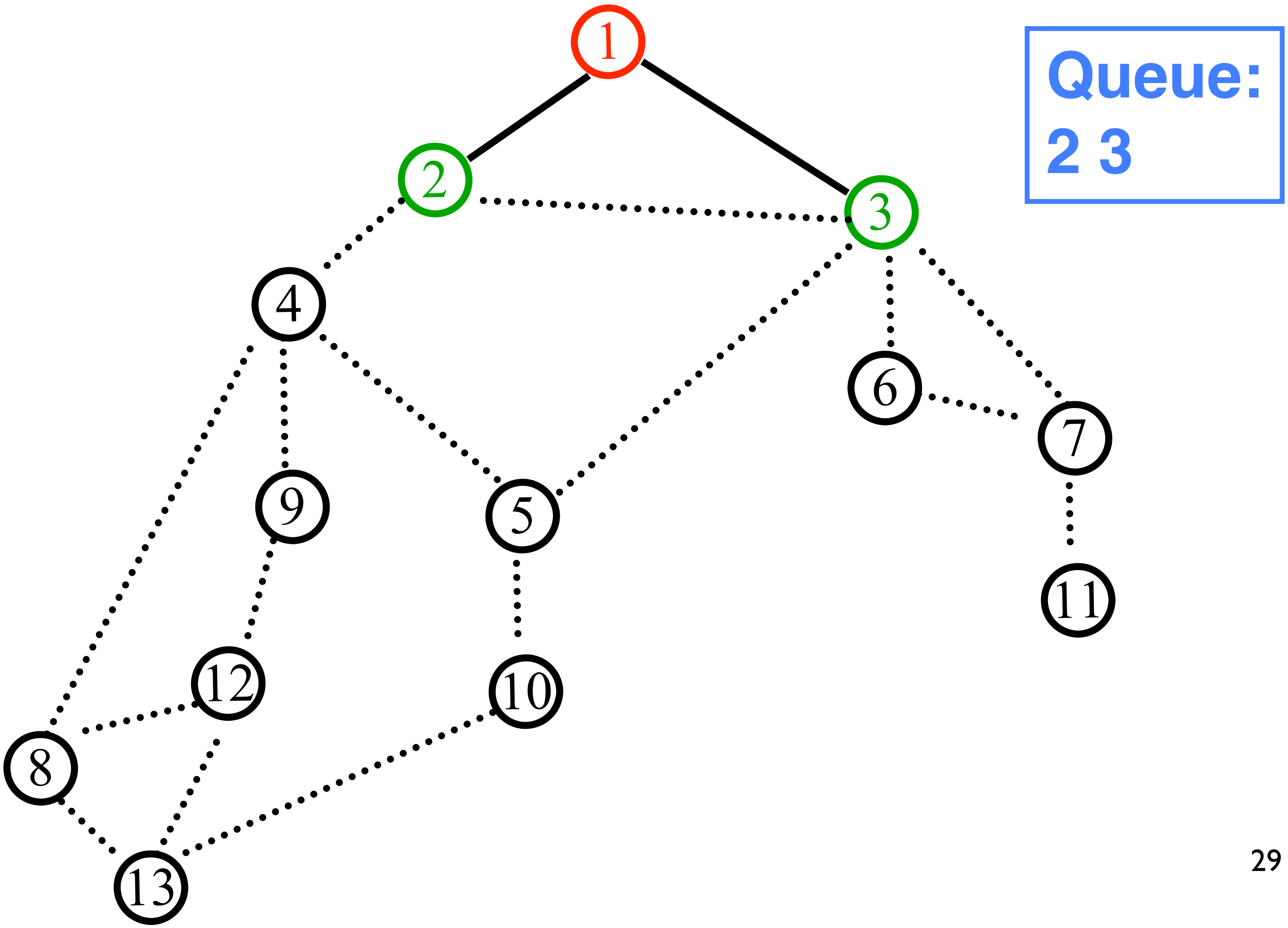
Exercise: modify code to number vertices & compute level numbers

BFS(v)

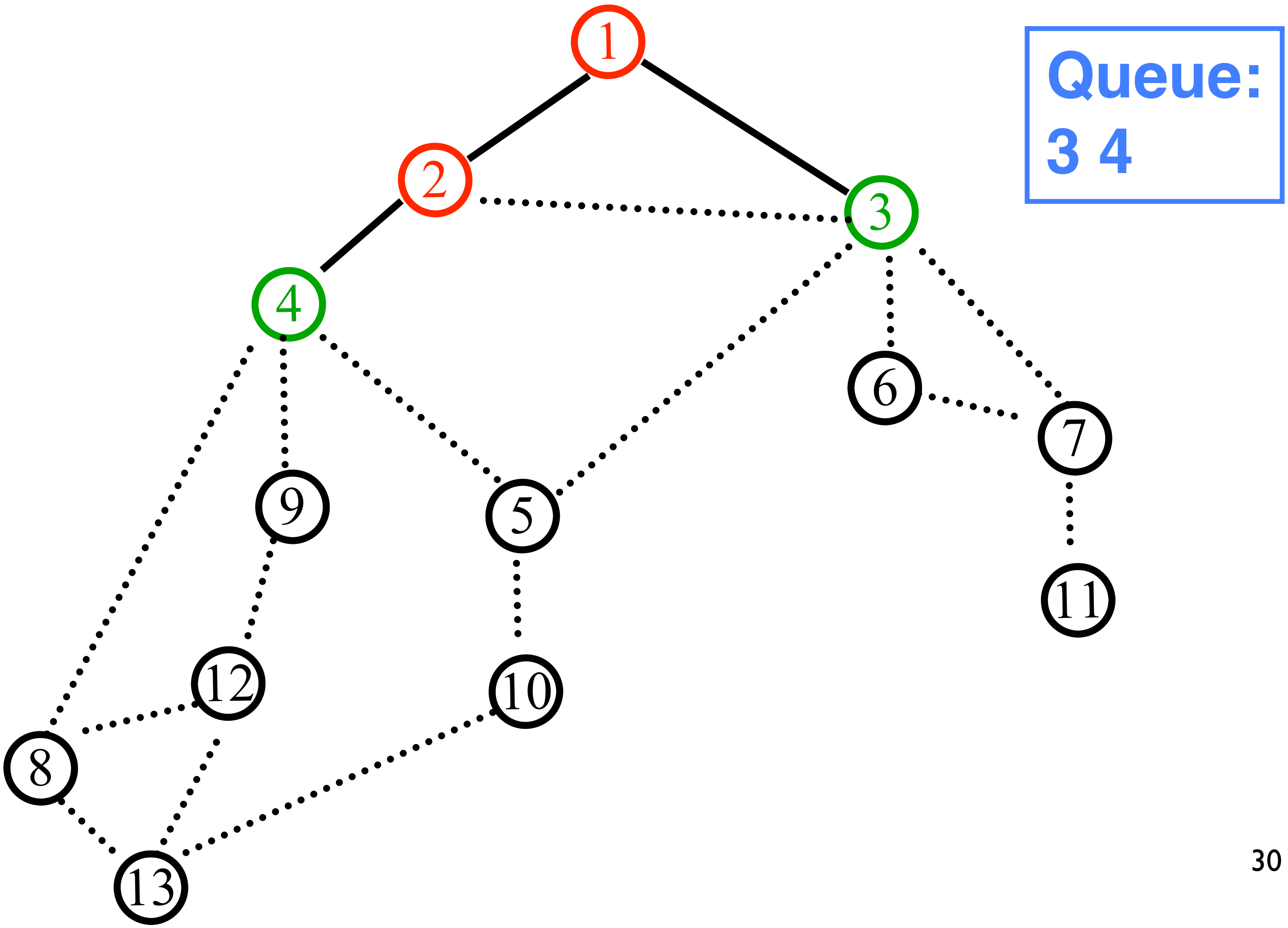


Queue:
1

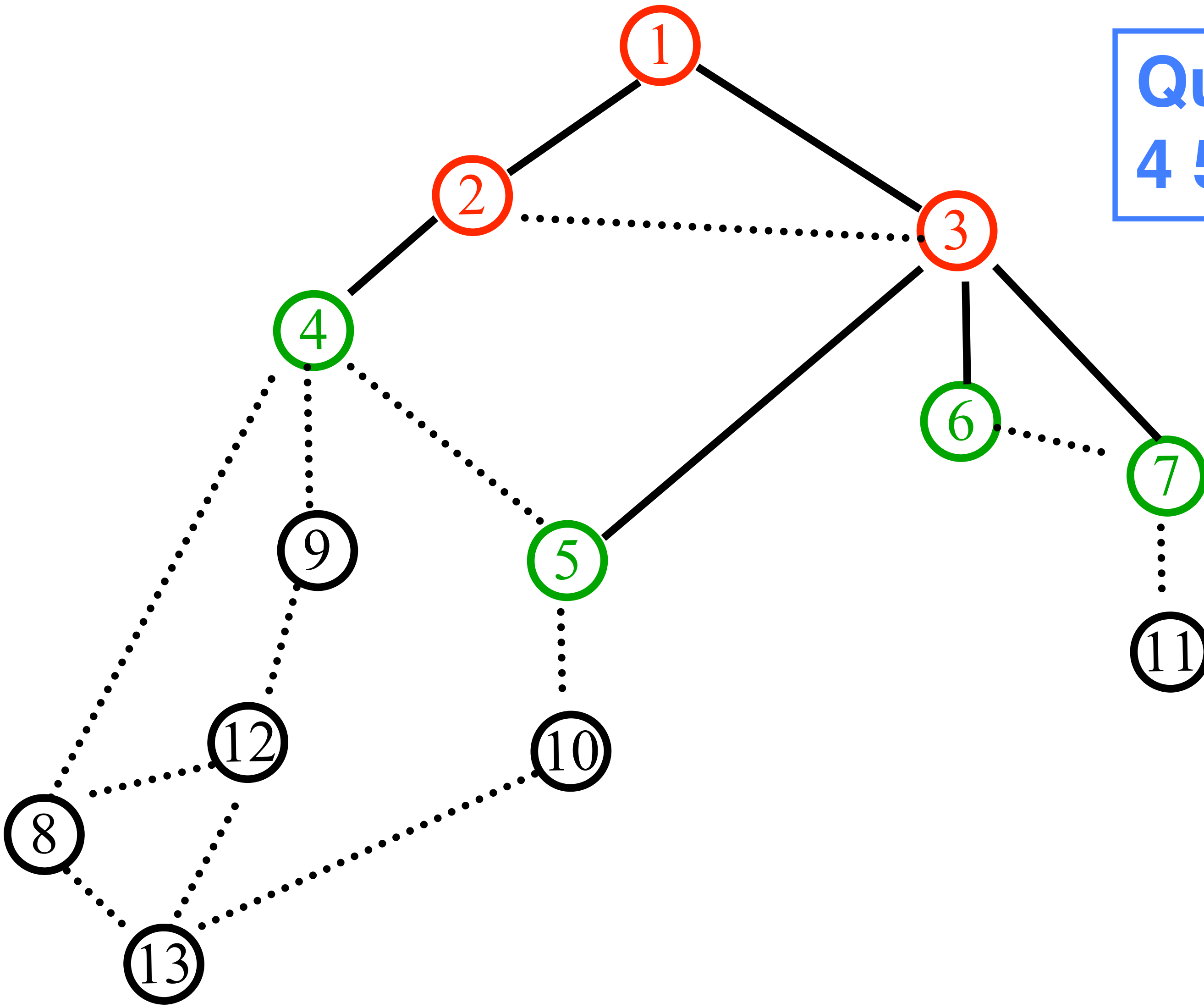
BFS(v)



BFS(v)

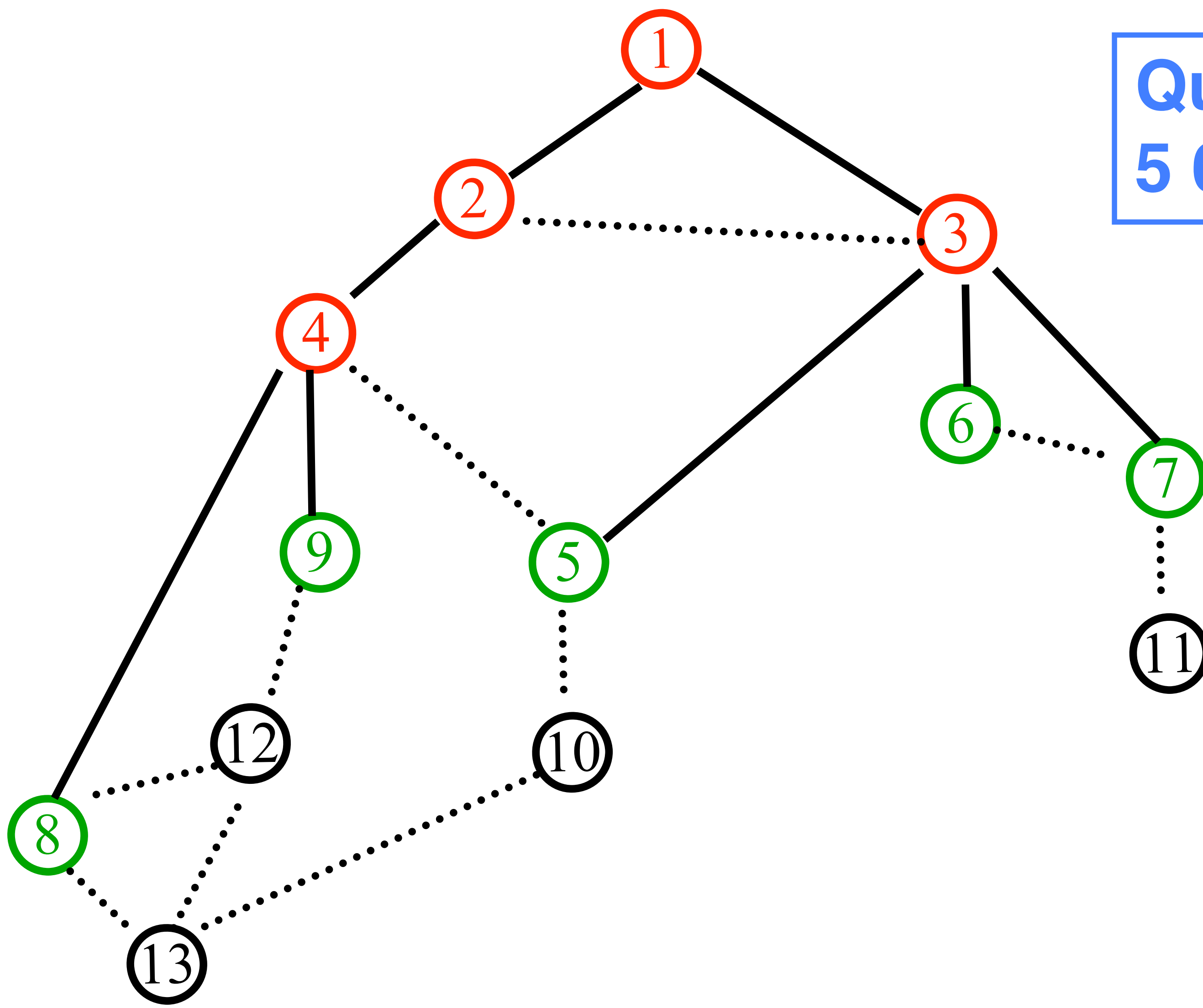


BFS(v)



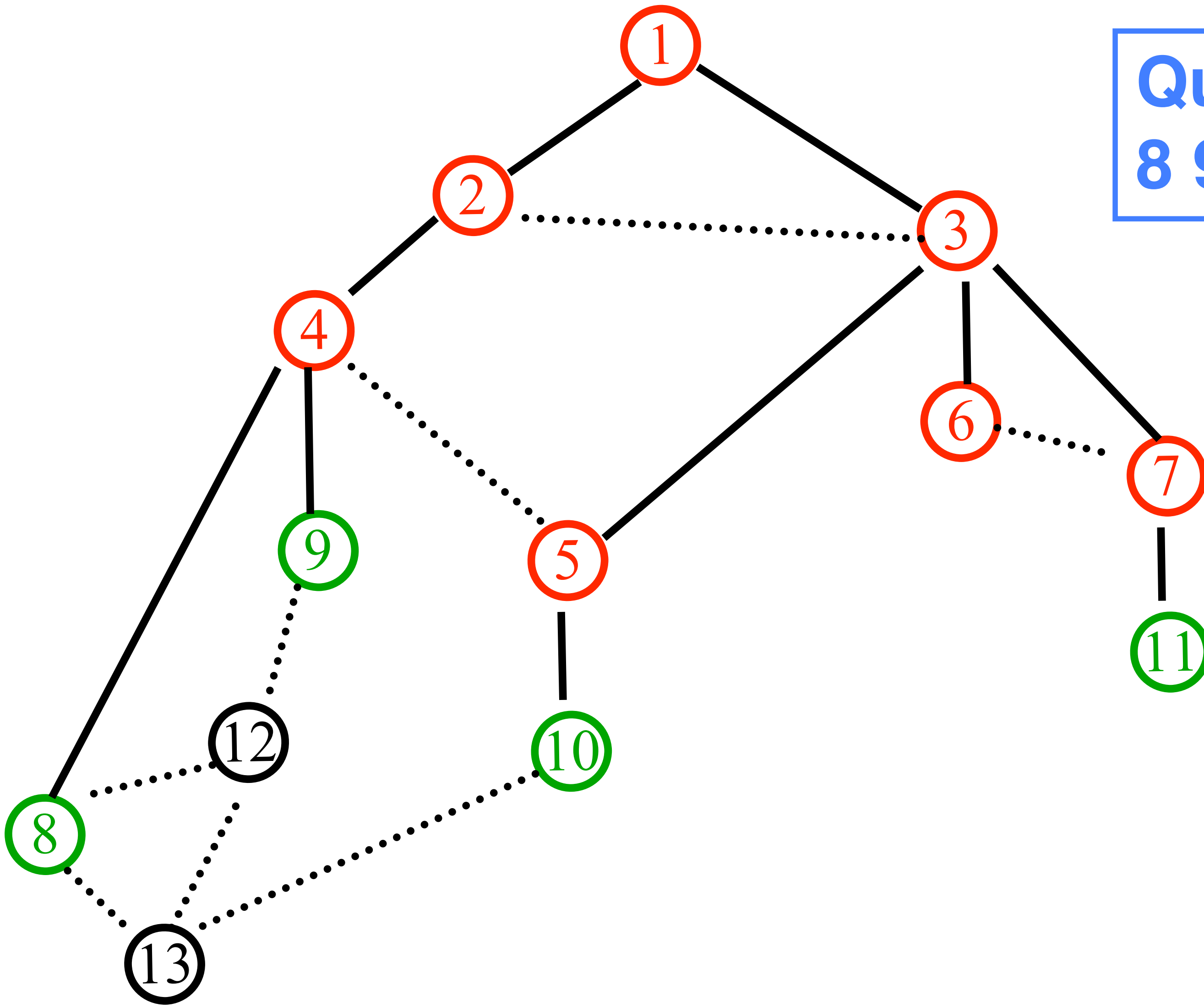
Queue:
4 5 6 7

BFS(v)



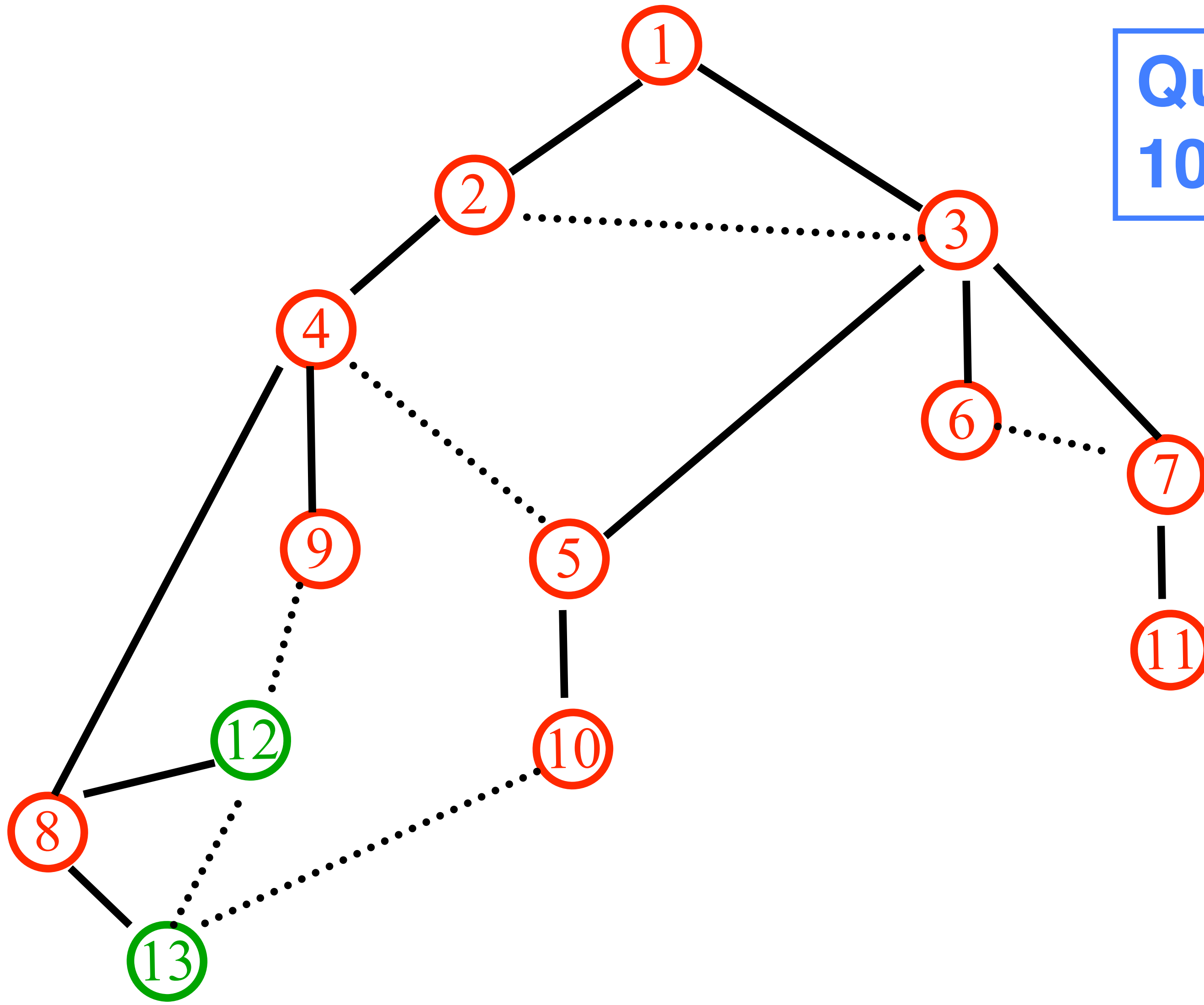
Queue:
5 6 7 8 9

BFS(v)



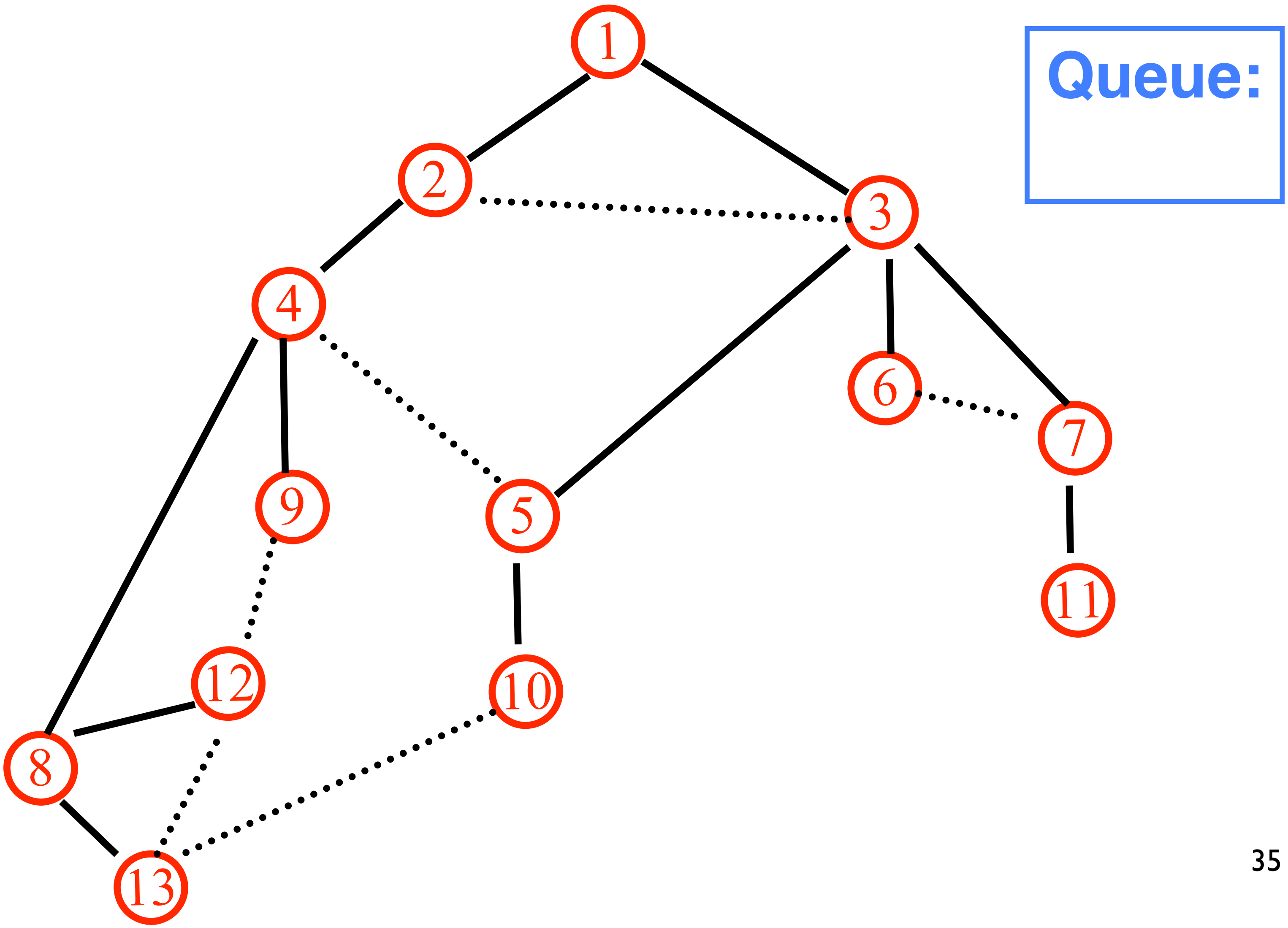
Queue:
8 9 10 11

BFS(v)



Queue:
10 11 12 13

BFS(v)



BFS(s) Implementation

Global initialization: mark all vertices "undiscovered"

```
BFS(s)
    mark  s "discovered"
    queue = { s }
    while queue not empty
        u = remove_first(queue)
        for each edge {u,x}
            if (x is undiscovered)
                mark x discovered
                append x on queue
        mark u fully explored
```

Exercise: modify code to number vertices & compute level numbers

BFS analysis

Each edge is explored once from each end-point

Each vertex is discovered by following a different edge

Total cost $O(m)$, m = # of edges

Exercise: extend algorithm and analysis to non-connected graphs

Properties of (Undirected) BFS(v)

BFS(v) visits x if and only if there is a path in G from v to x.

Edges into then-undiscovered vertices define a ***tree*** – the "breadth first spanning tree" of G

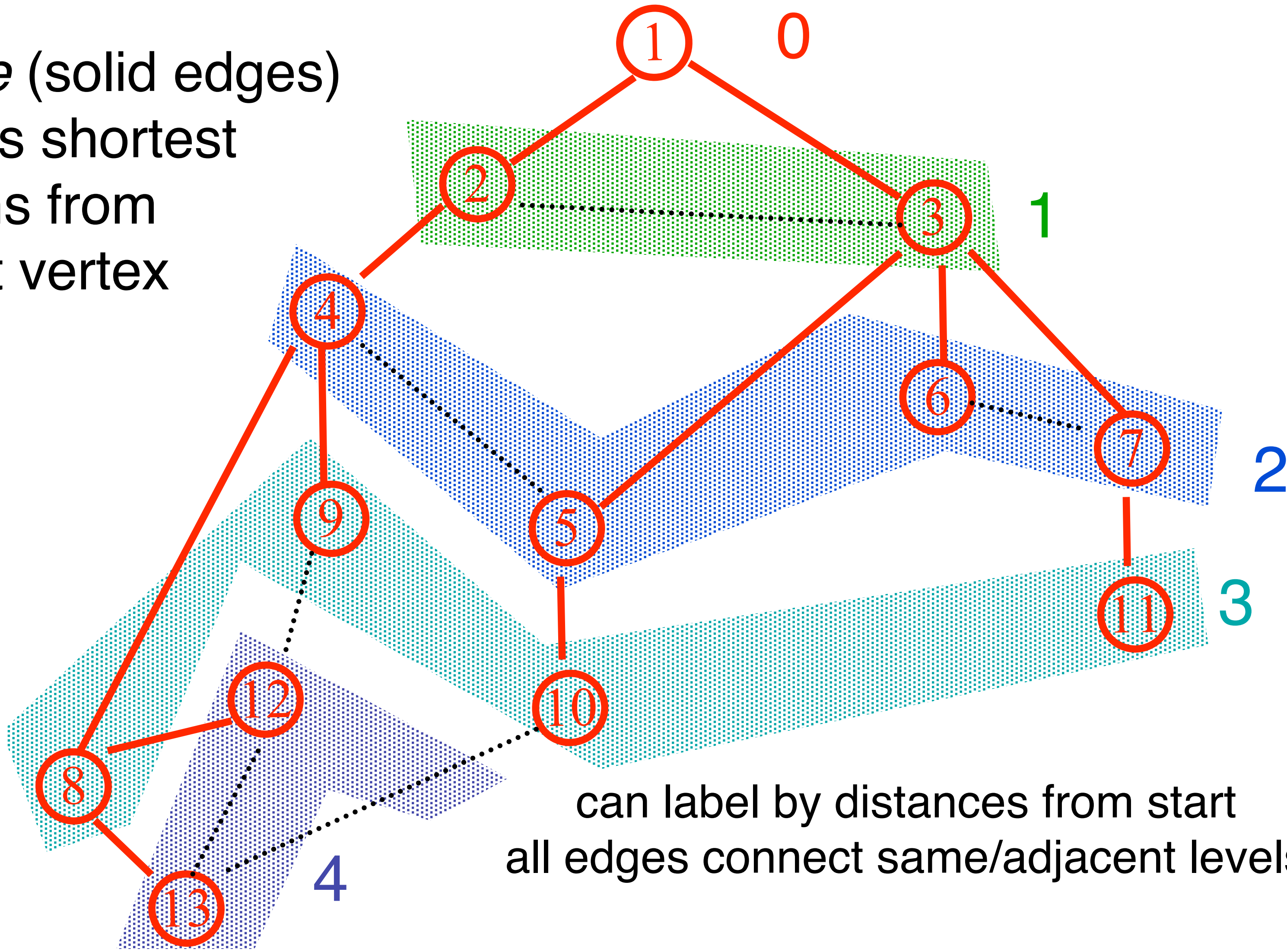
Level i in this tree are exactly those vertices *u* such that the shortest path (in G, not just the tree) from the root v is of length i.

All non-tree edges join vertices on the same or adjacent levels

not true of every spanning tree!

BFS Application: Shortest Paths

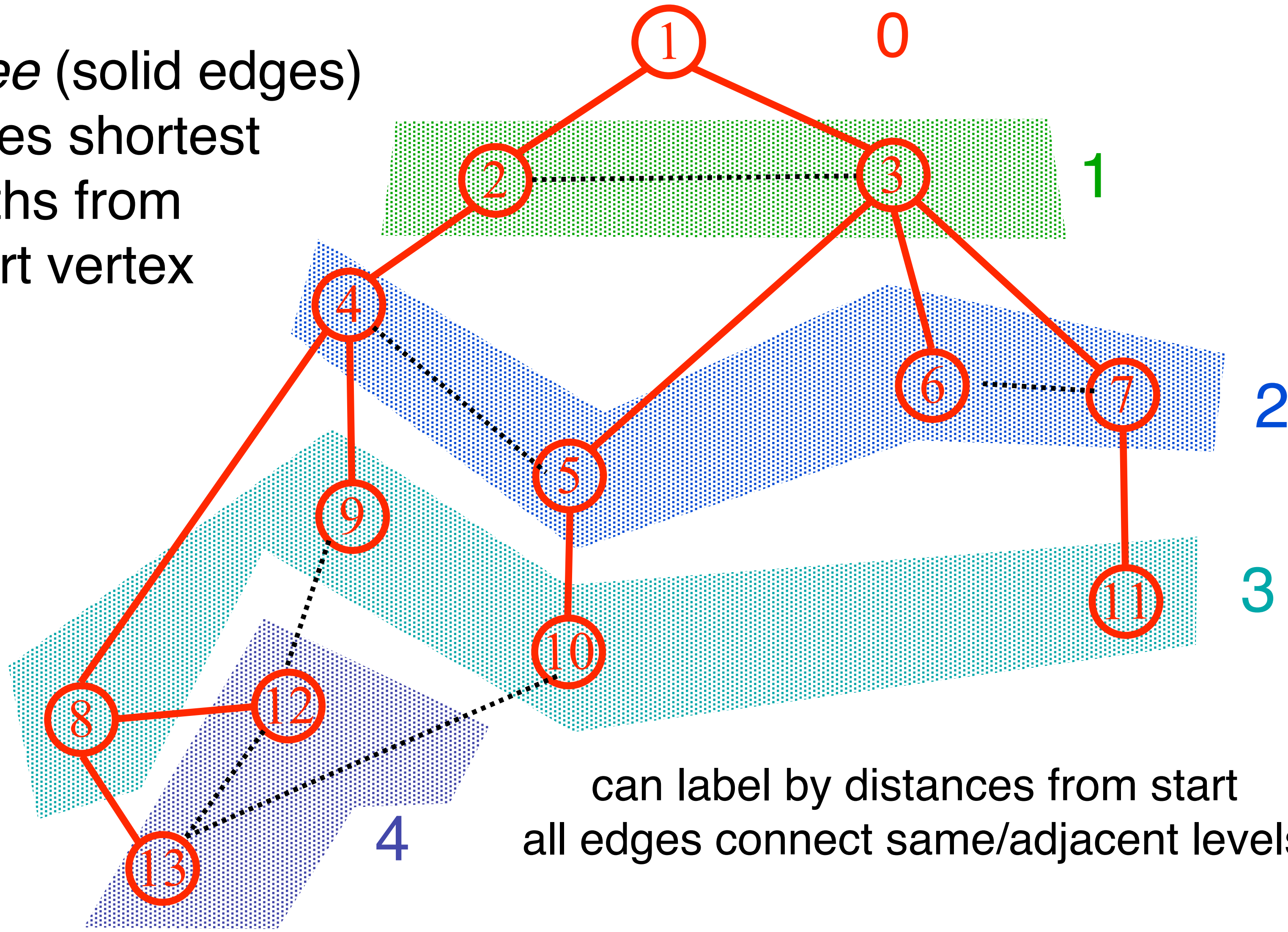
Tree (solid edges) gives shortest paths from start vertex



can label by distances from start
all edges connect same/adjacent levels

BFS Application: Shortest Paths

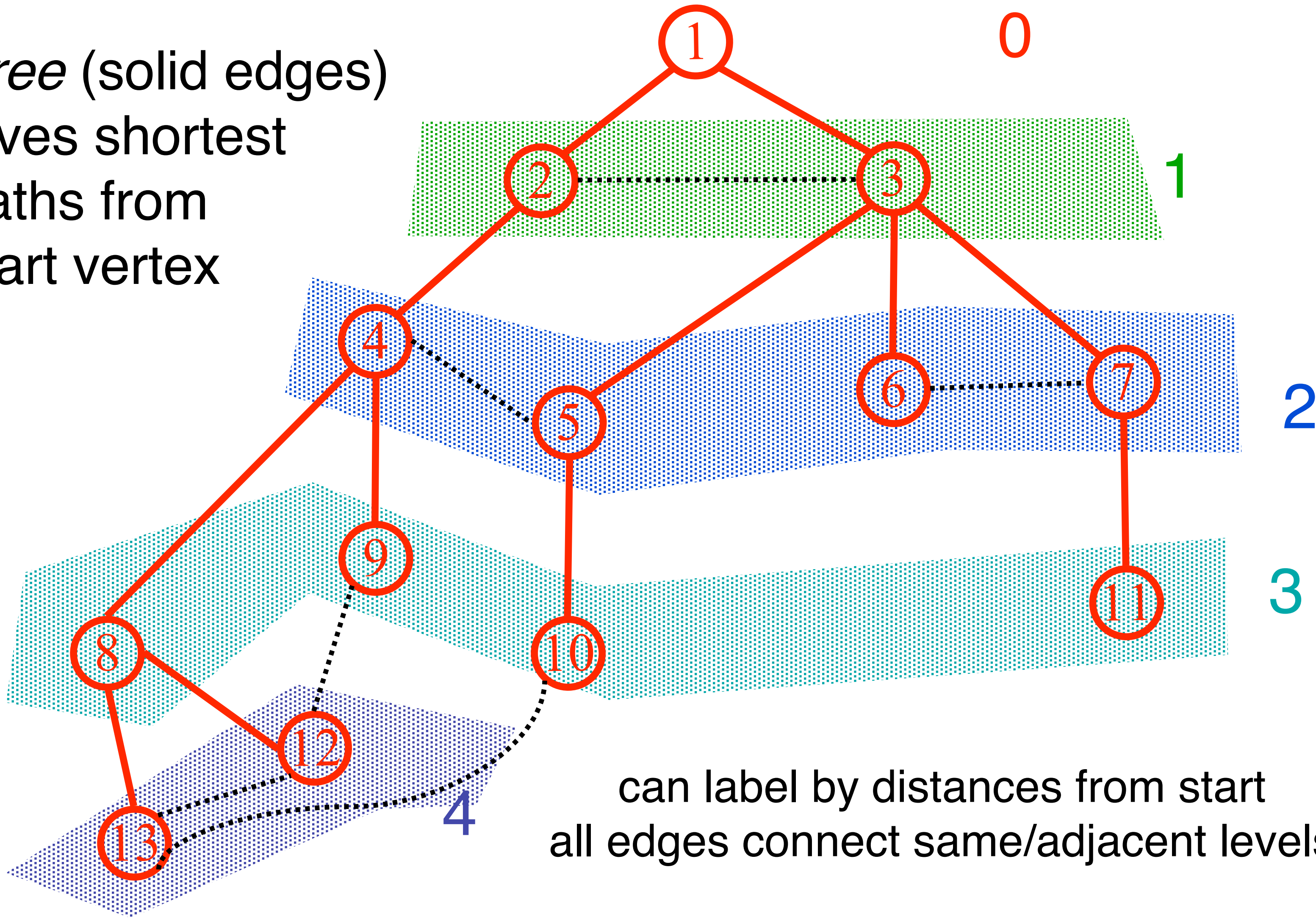
Tree (solid edges) gives shortest paths from start vertex



can label by distances from start
all edges connect same/adjacent levels

BFS Application: Shortest Paths

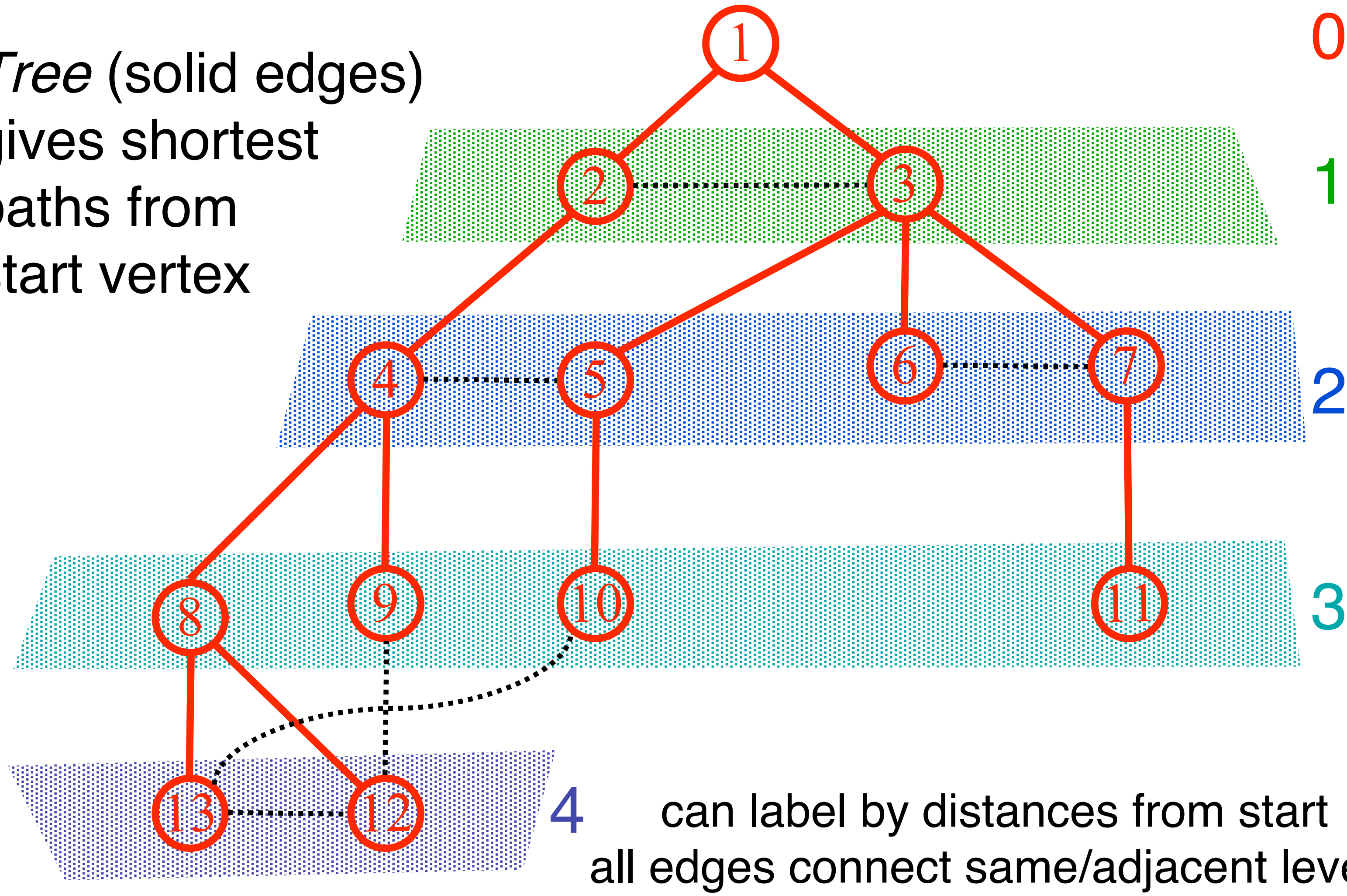
Tree (solid edges) gives shortest paths from start vertex



can label by distances from start
all edges connect same/adjacent levels

BFS Application: Shortest Paths

Tree (solid edges) gives shortest paths from start vertex



can label by distances from start
all edges connect same/adjacent levels

Why fuss about trees?

Trees are simpler than graphs

Ditto for algorithms on trees vs algs on graphs

So, this is often a good way to approach a graph problem: find a "nice" tree in the graph, i.e., one such that non-tree edges have some simplifying structure

E.g., BFS finds a tree s.t. level-jumps are minimized

DFS (next) finds a different tree, but it also has interesting structure…

Graph Search Application: Connected Components

Want to answer questions of the form:

given vertices u and v, is there a path from u to v?

Idea: create array A such that

A[u] = smallest numbered vertex that is connected to u. Question reduces to whether A[u]=A[v]?

Q: Why not create 2-d array Path[u,v]?

Graph Search Application: Connected Components

```
initial state: all v undiscovered
for v = 1 to n do
    if state(v) != fully-explored then
        BFS(v): setting A[u] ←v for each u found
        (and marking u discovered/fully-explored)
    endif
endfor
```

Total cost: O(n+m)

- each edge is touched a constant number of times (twice)
- works also with DFS

3.4 Testing Bipartiteness

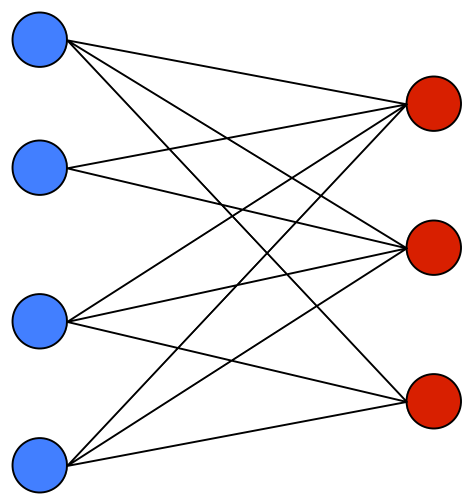
Bipartite Graphs

Def. An undirected graph G = (V, E) is *bipartite* if the nodes can be colored red or blue such that every edge has one red and one blue end.

Applications.

Stable marriage: men = red, women = blue

Scheduling: machines = red, jobs = blue



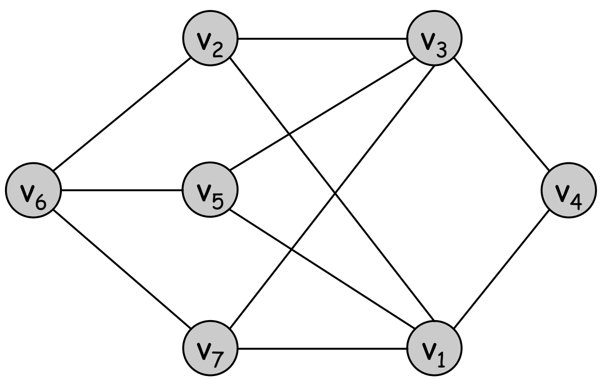
a bipartite graph

"bi-partite" means "two parts." An equivalent definition: G is bipartitite if you can partition the node set into 2 parts (say, blue/red or left/right) so that all edges join nodes in different parts/no edge has both ends in the same part.

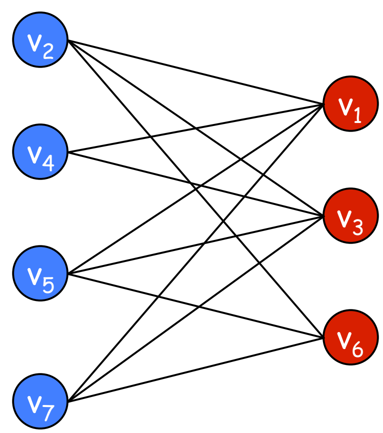
Testing Bipartiteness

Testing bipartiteness. Given a graph G, is it bipartite?

- Many graph problems become:
  - easier if the underlying graph is bipartite (matching)
  - tractable if the underlying graph is bipartite (independent set)
- Before attempting to design an algorithm, we need to understand structure of bipartite graphs.



a bipartite graph G

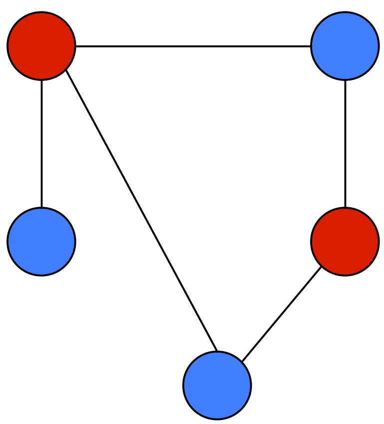


another drawing of G

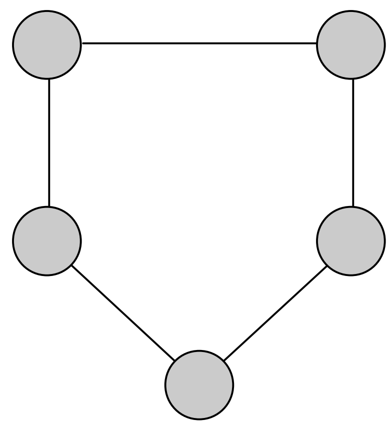
An Obstruction to Bipartiteness

Lemma. If a graph G is bipartite, it cannot contain an odd length cycle.

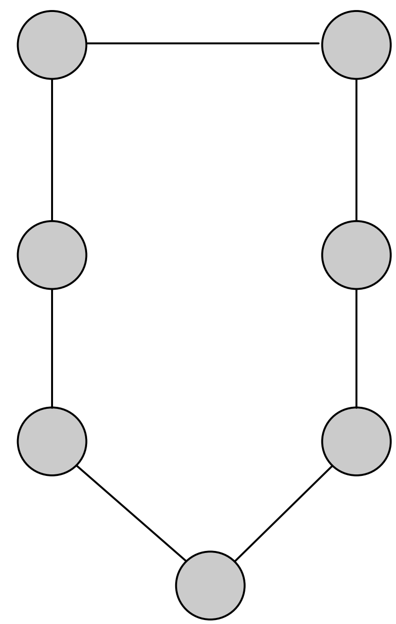
Pf. Impossible to 2-color the odd cycle, let alone G.



bipartite
(2-colorable)



not bipartite
(not 2-colorable)



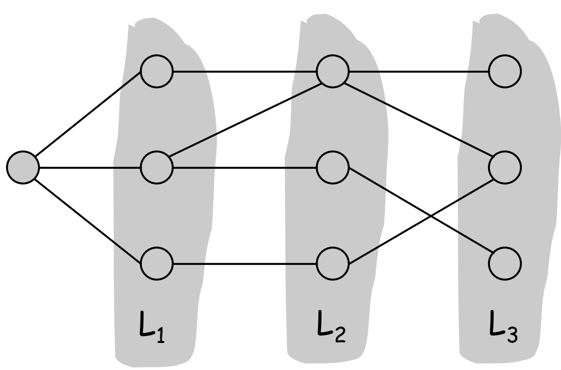
not bipartite
(not 2-colorable)

Bipartite Graphs

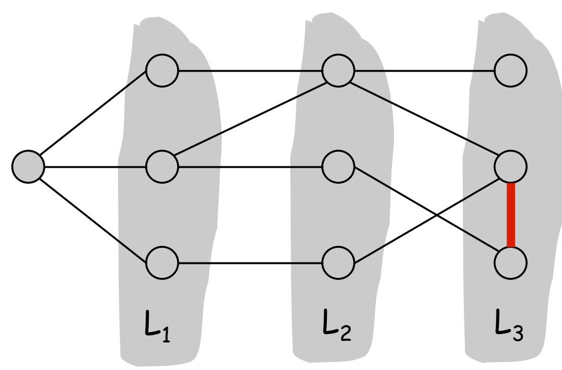
Lemma. Let G be a connected graph, and let $L_0, \ldots, L_k$ be the layers produced by BFS starting at node s. Exactly one of the following holds.

(i) No edge of G joins two nodes of the same layer, and G is bipartite.

(ii) An edge of G joins two nodes of the same layer, and G contains an odd-length cycle (and hence is not bipartite).



Case (i)



Case (ii)

Bipartite Graphs

Lemma. Let G be a connected graph, and let L_0, …, L_k be the layers produced by BFS starting at node s. Exactly one of the following holds.

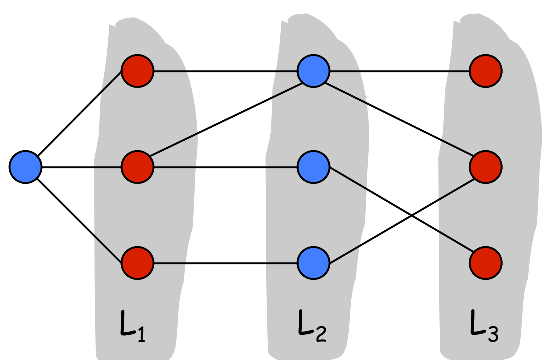
(i) No edge of G joins two nodes of the same layer, and G is bipartite.

(ii) An edge of G joins two nodes of the same layer, and G contains an odd-length cycle (and hence is not bipartite).

Pf. (i)

Suppose no edge joins two nodes in the same layer.

By previous lemma, all edges join nodes on adjacent levels.



Case (i)

Bipartition:
red = nodes on odd levels,
blue = nodes on even levels.

Bipartite Graphs

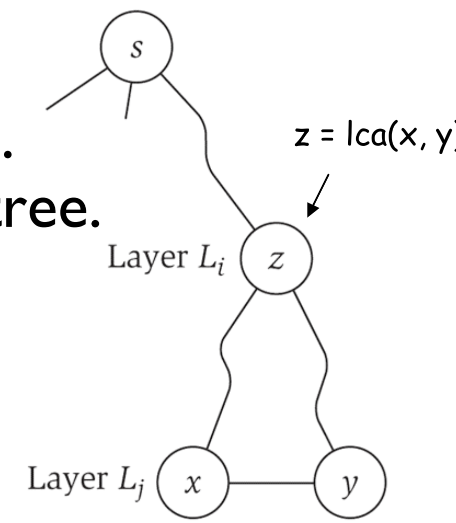
Lemma. Let G be a connected graph, and let $L_0, \ldots, L_k$ be the layers produced by BFS starting at node s. Exactly one of the following holds.

(i) No edge of G joins two nodes of the same layer, and G is bipartite.

(ii) An edge of G joins two nodes of the same layer, and G contains an odd-length cycle (and hence is not bipartite).

Pf. (ii)

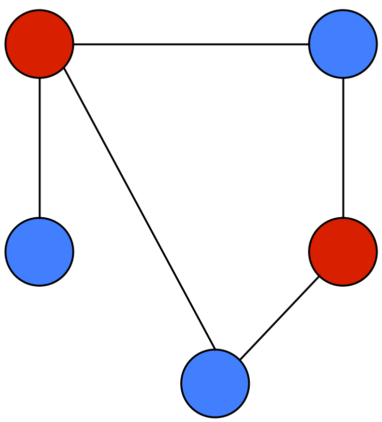
Suppose (x, y) is an edge & x, y in same level Lj.
Let z = their lowest common ancestor in BFS tree.
Let Li be level containing z.
Consider cycle that takes edge from x to y,
then tree from y to z, then tree from z to x.
Its length is $\underbrace{1}_{(x,\, y)} + \underbrace{(j-i)}_{\text{path from } y \text{ to } z} + \underbrace{(j-i)}_{\text{path from } z \text{ to } x}$, which is odd.



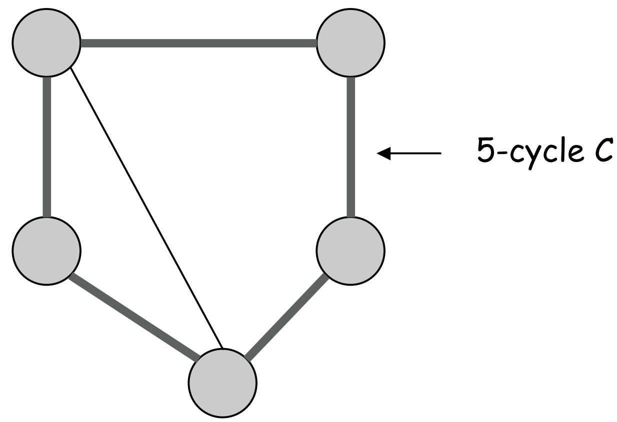
Obstruction to Bipartiteness

Cor: A graph G is bipartite iff it contains no odd length cycle.

NB: the proof is algorithmic-- in a non-bipartite graph, it *finds* an odd cycle.



bipartite
(2-colorable)



not bipartite
(not 2-colorable)

3.6 DAGs and Topological Ordering

Precedence Constraints

Precedence constraints. Edge (v_i, v_j) means task v_i must occur before v_j.

Applications

Course prerequisite graph: course v_i must be taken before v_j

Compilation: must compile module v_i before v_j

Pipeline of computing jobs: output of job v_i is part of input to job v_j

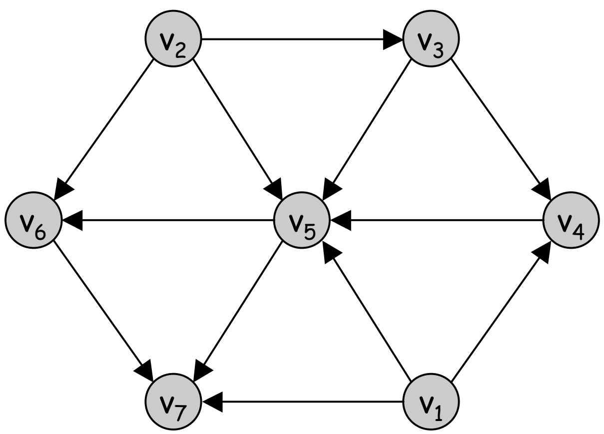
Manufacturing or assembly: sand it before you paint it…

Directed Acyclic Graphs

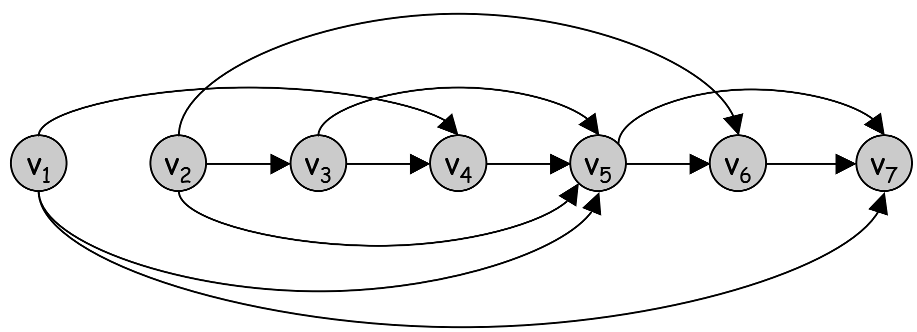
Def. A DAG is a directed acyclic graph, i.e., one that contains no directed cycles.

Ex. Precedence constraints: edge (v_i, v_j) means v_i must precede v_j.

Def. A <u>topological order</u> of a directed graph $G = (V, E)$ is an ordering of its nodes as $v_1, v_2, \ldots, v_n$ so that for every edge (v_i, v_j) we have $i < j$.



a DAG



a topological ordering of that DAG

Directed Acyclic Graphs

Lemma. If G has a topological order, then G is a DAG.

if all edges go L->R, you can't loop back to close a cycle

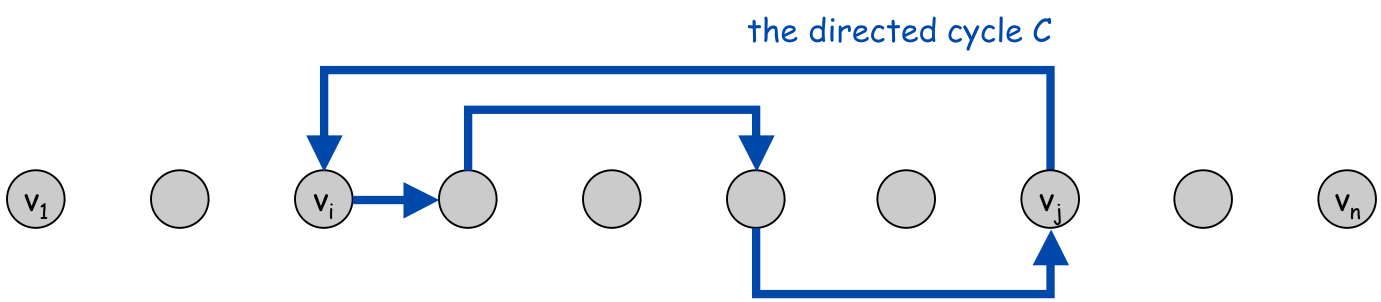
Pf. (by contradiction)

Suppose that G has a topological order $v_1, \ldots, v_n$ and that G also has a directed cycle C.

Let v_i be the lowest-indexed node in C, and let v_j be the node just before v_i; thus (v_j, v_i) is an edge.

By our choice of i, we have $i < j$.

On the other hand, since (v_j, v_i) is an edge and $v_1, \ldots, v_n$ is a topological order, we must have $j < i$, a contradiction. ▪



the supposed topological order: $v_1, \ldots, v_n$

Directed Acyclic Graphs

Lemma.
  If G has a topological order, then G is a DAG.

Q. Does every DAG have a topological ordering?

Q. If so, how do we compute one?

Directed Acyclic Graphs

Lemma. If G is a DAG, then G has a node with no incoming edges.

Pf. (by contradiction)

Suppose that G is a DAG and every node has at least one incoming edge. Let's see what happens.

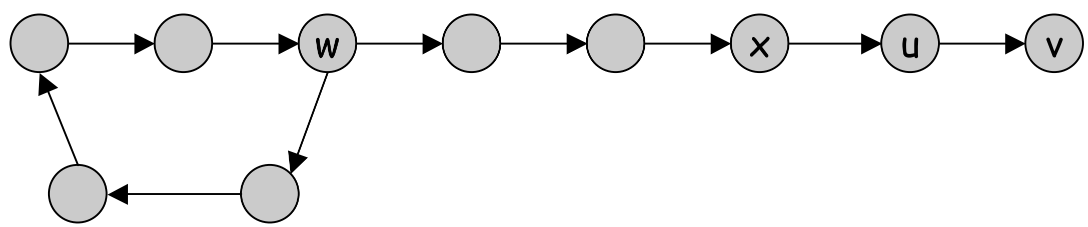
Pick any node v, and begin following edges backward from v. Since v has at least one incoming edge (u, v) we can walk backward to u.

Then, since u has at least one incoming edge (x, u), we can walk backward to x.

Repeat until we visit a node, say w, twice.

Why must this happen?

Let C be the sequence of nodes encountered between successive visits to w. C is a cycle.



Directed Acyclic Graphs

Lemma. If G is a DAG, then G has a topological ordering.

Pf. (by induction on n)

Base case: true if n = 1.

Given DAG on n > 1 nodes, find a node v with no incoming edges.

G - { v } is a DAG, since deleting v cannot create cycles.

By inductive hypothesis, G - { v } has a topological ordering.

Place v first in topological ordering; then append nodes of G - { v } in topological order. This is valid since v has no incoming edges. ▪

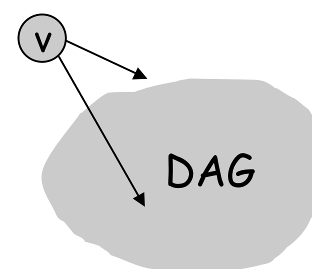
To compute a topological ordering of G:

Find a node v with no incoming edges and order it first

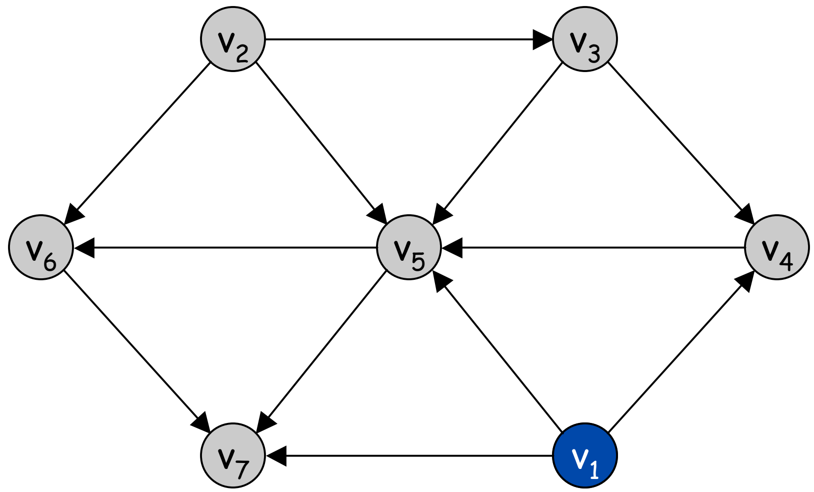
Delete v from G

Recursively compute a topological ordering of $G-\{v\}$

  and append this order after v

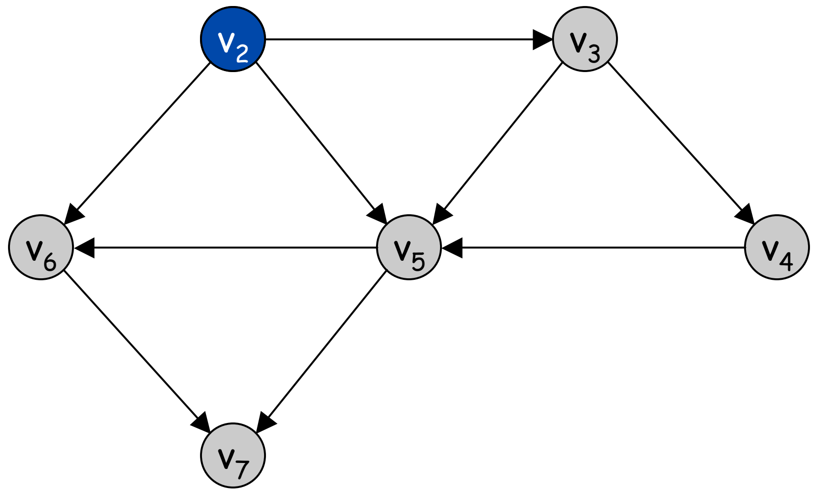


Topological Ordering Algorithm: Example



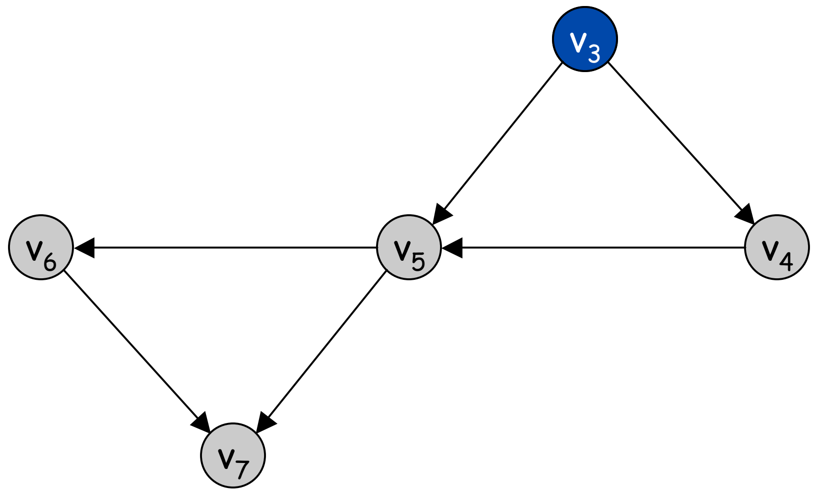
Topological order:

Topological Ordering Algorithm: Example



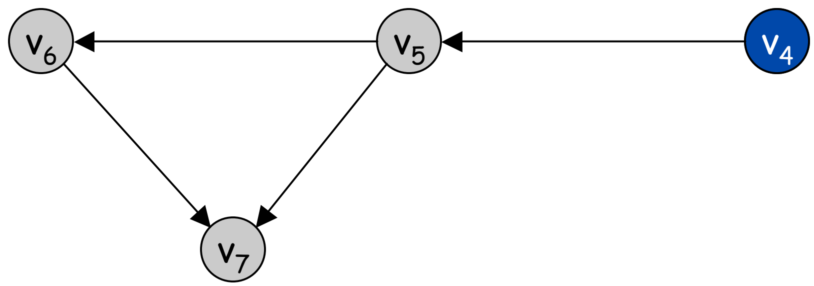
Topological order: v_1

Topological Ordering Algorithm: Example



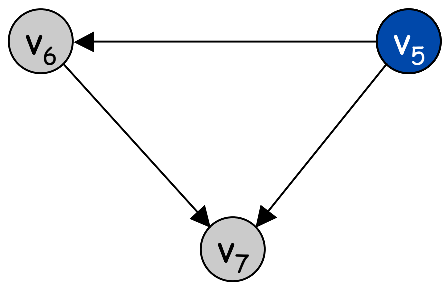
Topological order: v_1, v_2

Topological Ordering Algorithm: Example



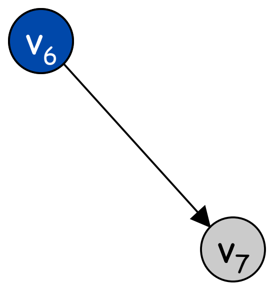
Topological order: v_1, v_2, v_3

Topological Ordering Algorithm: Example



Topological order: v_1, v_2, v_3, v_4

Topological Ordering Algorithm: Example

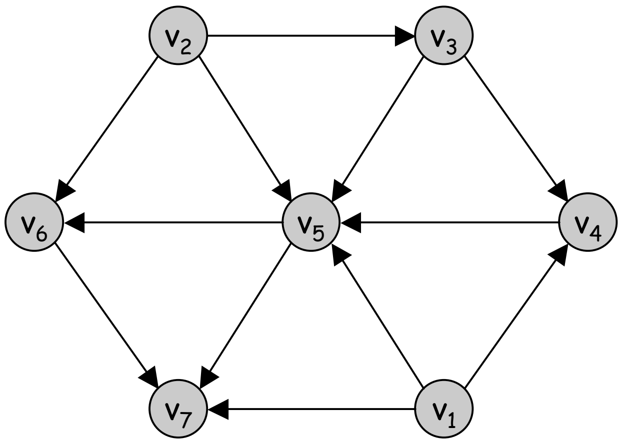


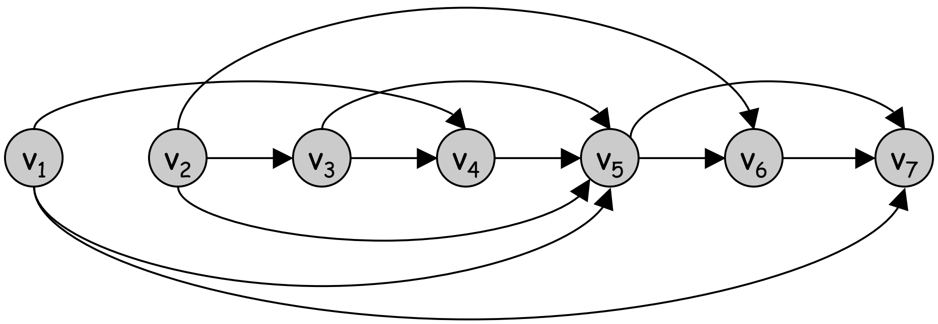
Topological order: v_1, v_2, v_3, v_4, v_5

Topological Ordering Algorithm: Example

Topological order: v_1, v_2, v_3, v_4, v_5, v_6

Topological Ordering Algorithm: Example





Topological order: $v_1, v_2, v_3, v_4, v_5, v_6, v_7$.

Topological Sorting Algorithm

Maintain the following:

count[w] = (remaining) number of incoming edges to node w

S = set of (remaining) nodes with no incoming edges

Initialization:

count[w] = 0 for all w

count[w]++ for all edges (v,w)

S = S ∪ {w} for all w with count[w]==0

O(m + n)

Main loop:

while S not empty

  remove some v from S

  make v next in topo order

  for all edges from v to some w

  decrement count[w]

  add w to S if count[w] hits 0

O(1) per node

O(1) per edge

Correctness: clear, I hope

Time: O(m + n) (assuming edge-list representation of graph)

Depth-First Search

Follow the first path you find as far as you can go

Back up to last unexplored edge when you reach a dead end, then go as far you can

Naturally implemented using recursive calls or a stack

DFS(v) – Recursive version

Global Initialization:

```
    for all nodes v, v.dfs# = -1   // mark v "undiscovered"
    dfscounter = 0

DFS(v)
    v.dfs# = dfscounter++         // v "discovered", number it
    for each edge (v,x)
        if (x.dfs# = -1)          // tree edge (x previously undiscovered)
            DFS(x)
        else …                    // code for back-, fwd-, parent,
                                  // edges, if needed
                                  // mark v "completed," if needed
```