Graphs and Graph Algorithms

Slides by Larry Ruzzo

Goals

Graphs: defns, examples, utility, terminology

Representation: input, internal

Traversal: Breadth- & Depth-first search

Three Algorithms:

- Connected components
- Bipartiteness
- Topological sort

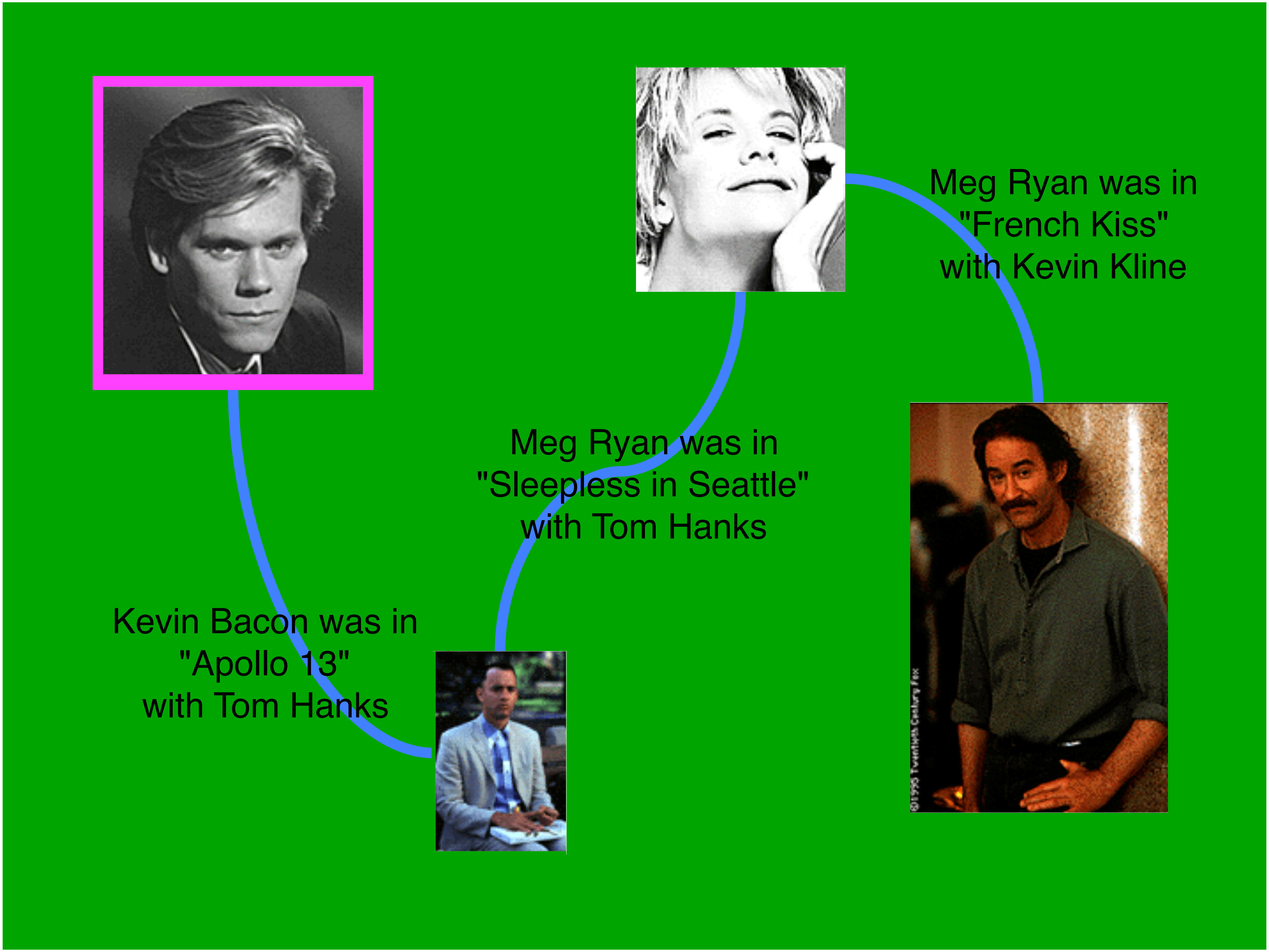
Graphs

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


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:

Obj: Actors

Rel: Two are related if they've been in a movie together

Exam Scheduling:

Obj: Classes

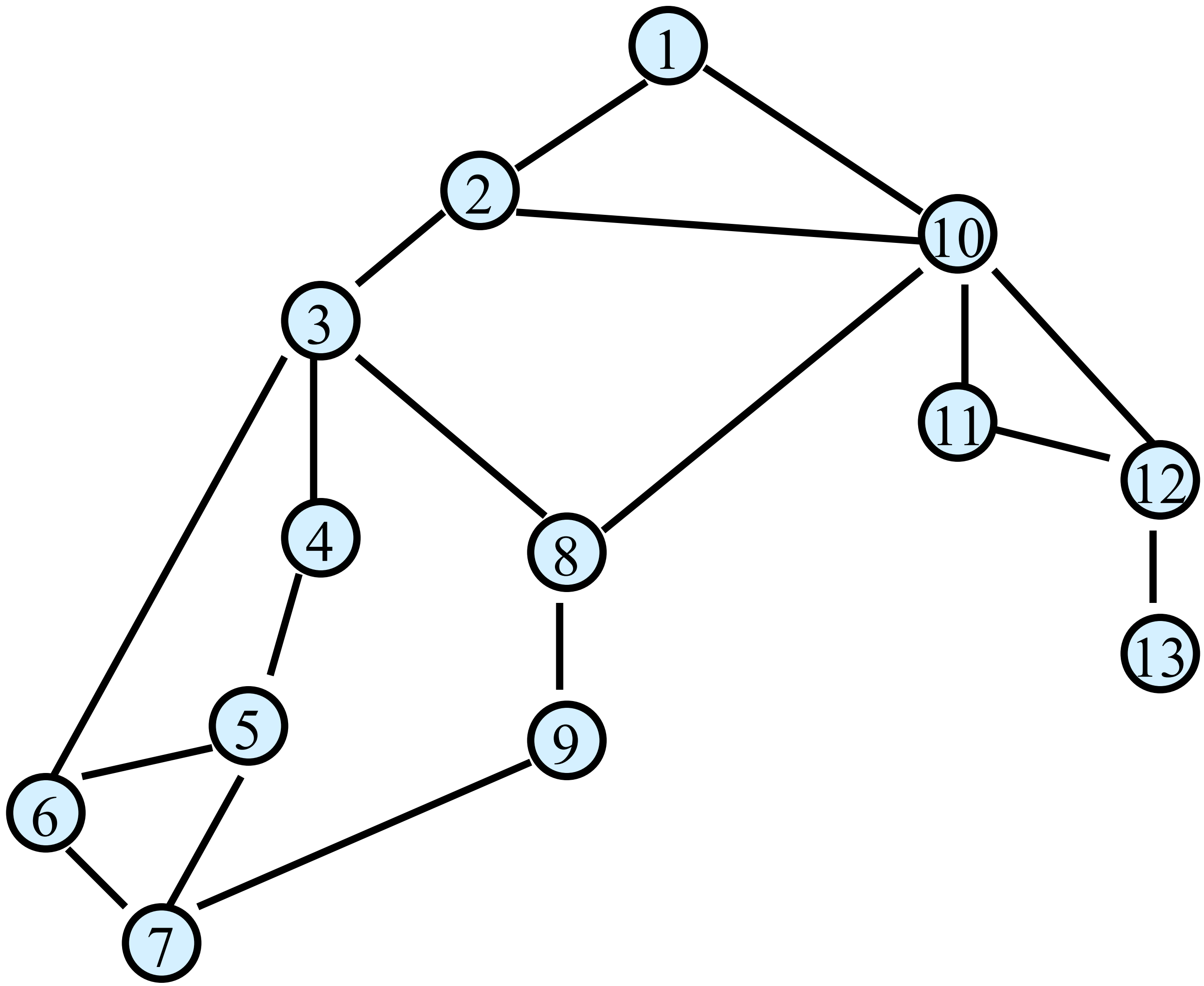
Rel: Two are related if they have students in common

Traveling Salesperson Problem:

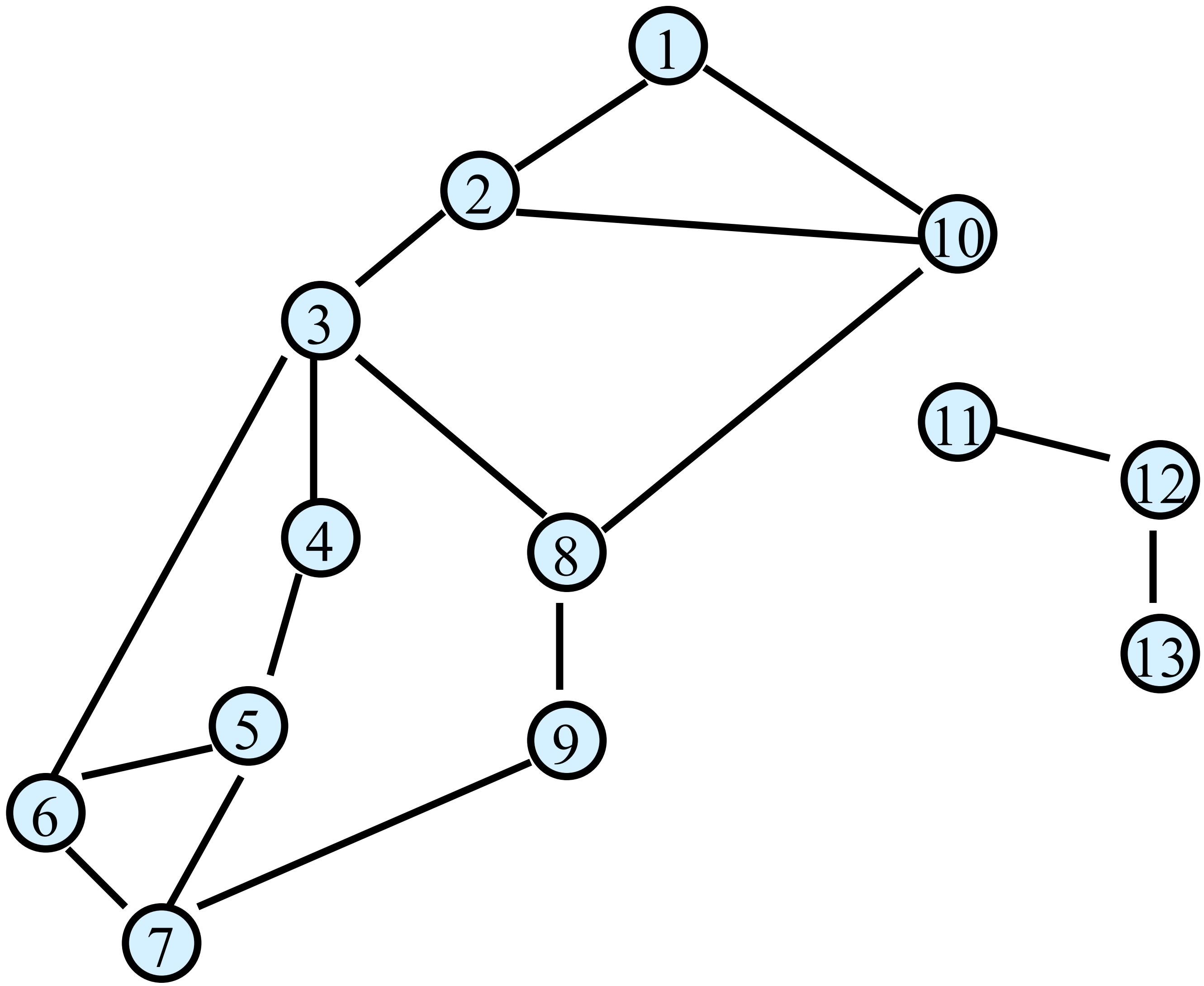
Obj: Cities

Rel: Two are related if can travel *directly* between them

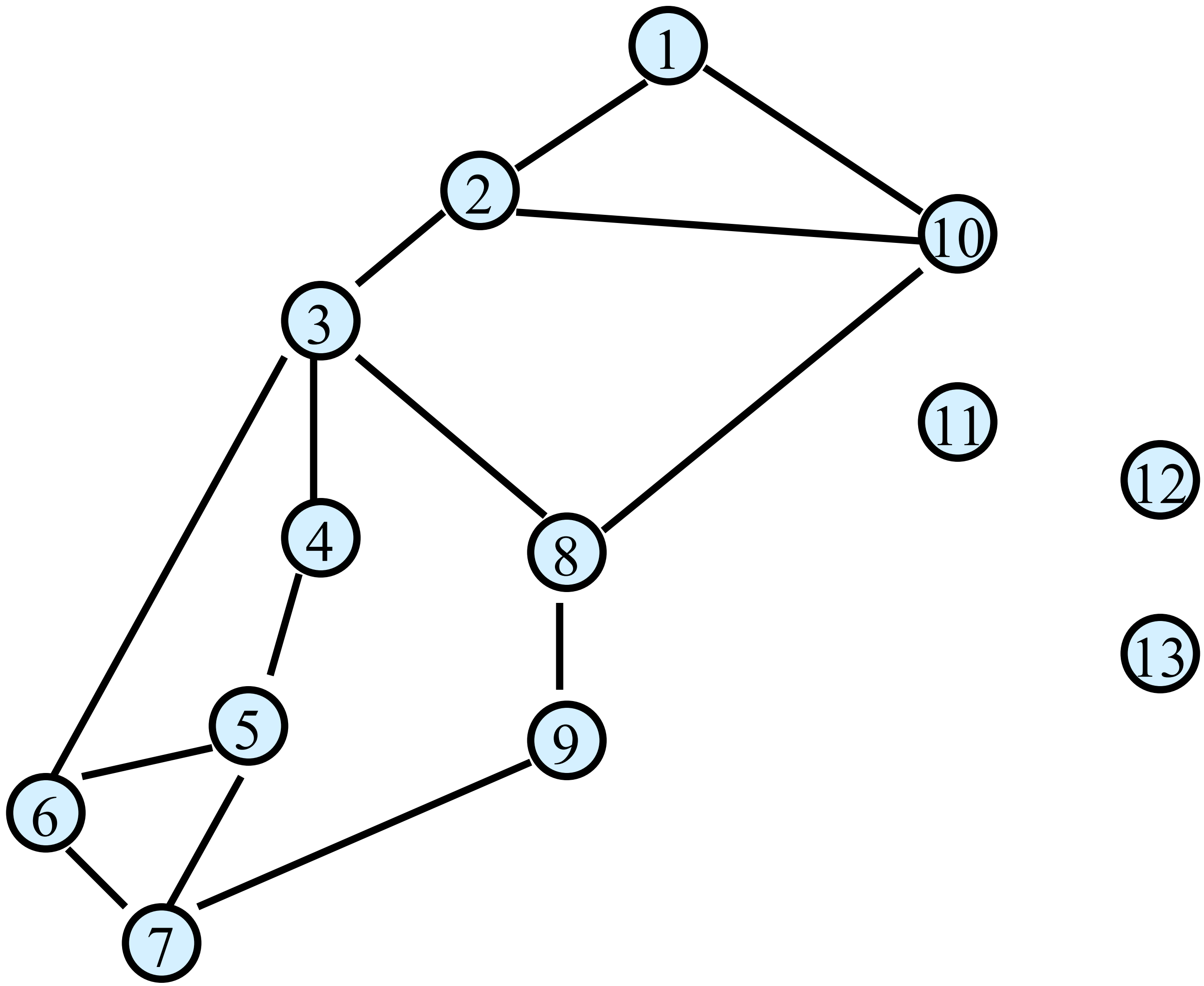
Undirected Graph   G = (V,E)



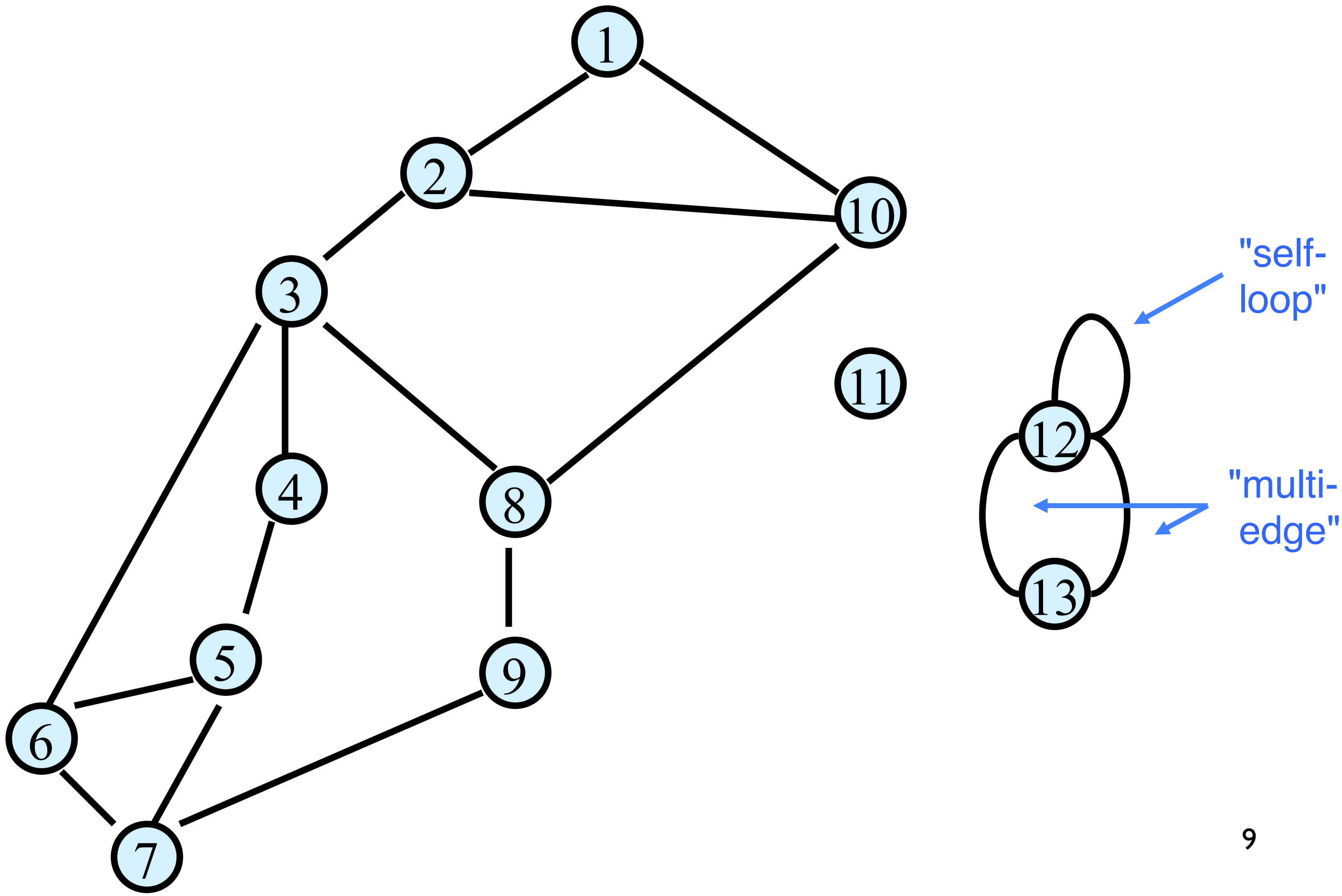
Undirected Graph   G = (V,E)



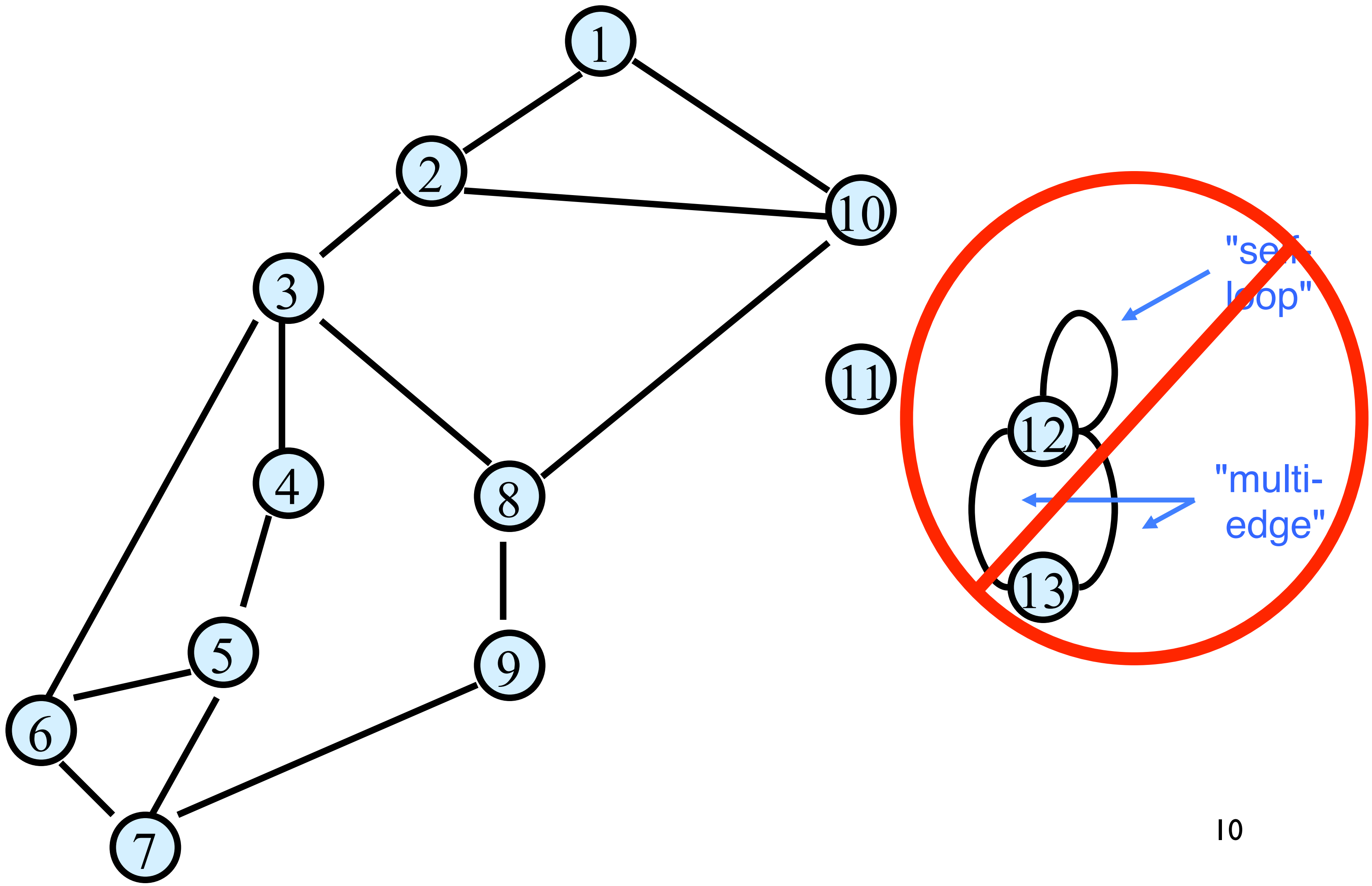
Undirected Graph G = (V,E)



Undirected Graph G = (V,E)

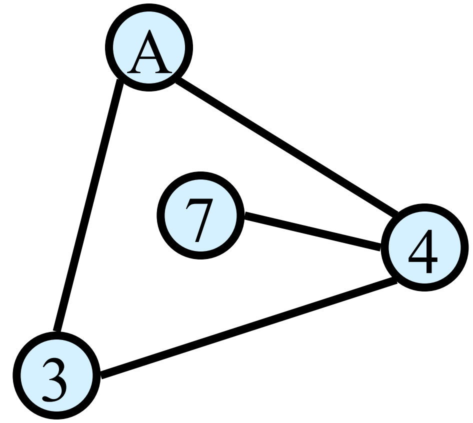


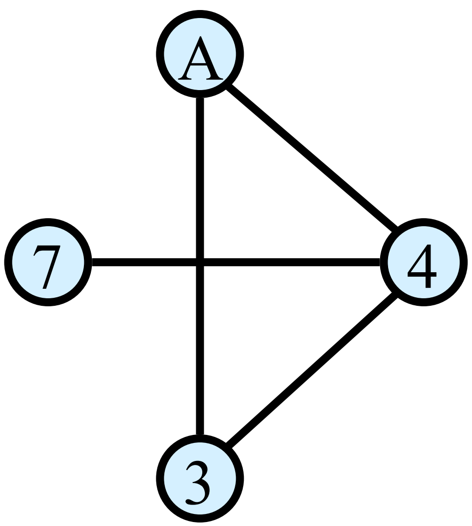
Undirected Graph   G = (V,E)

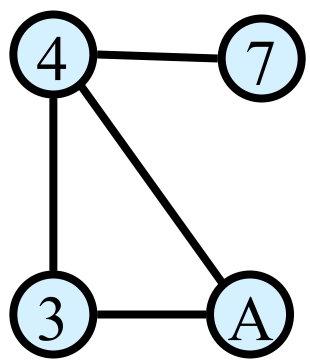


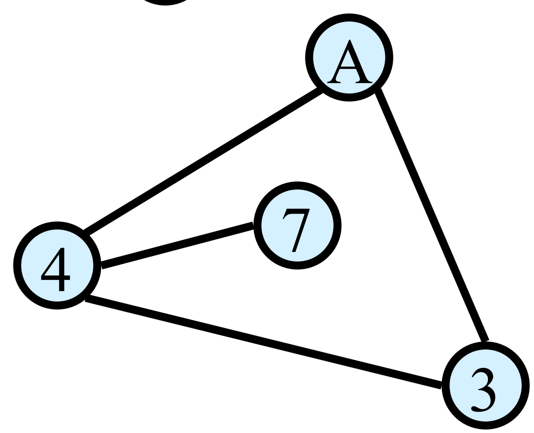
Graphs don't live in Flatland

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

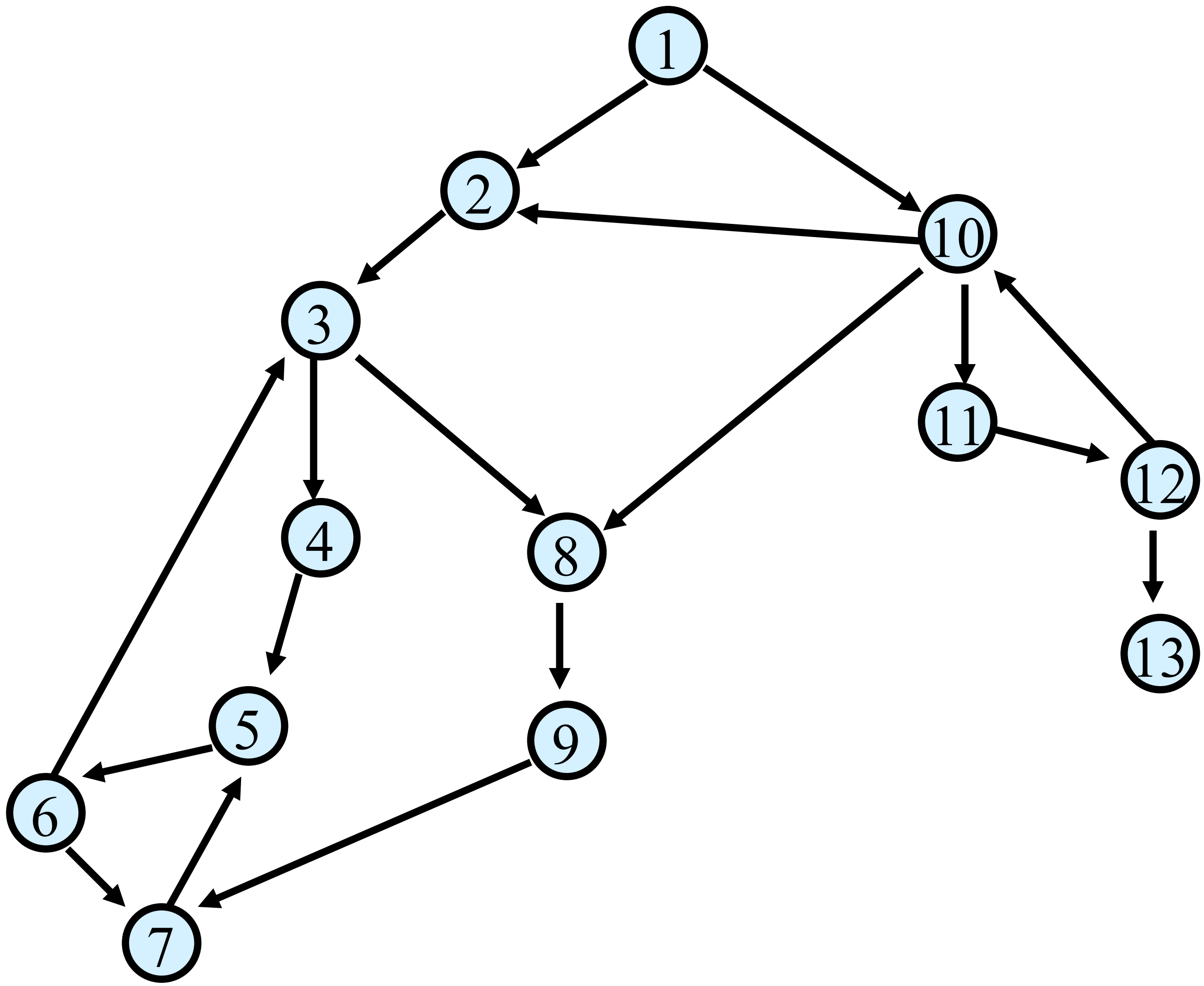




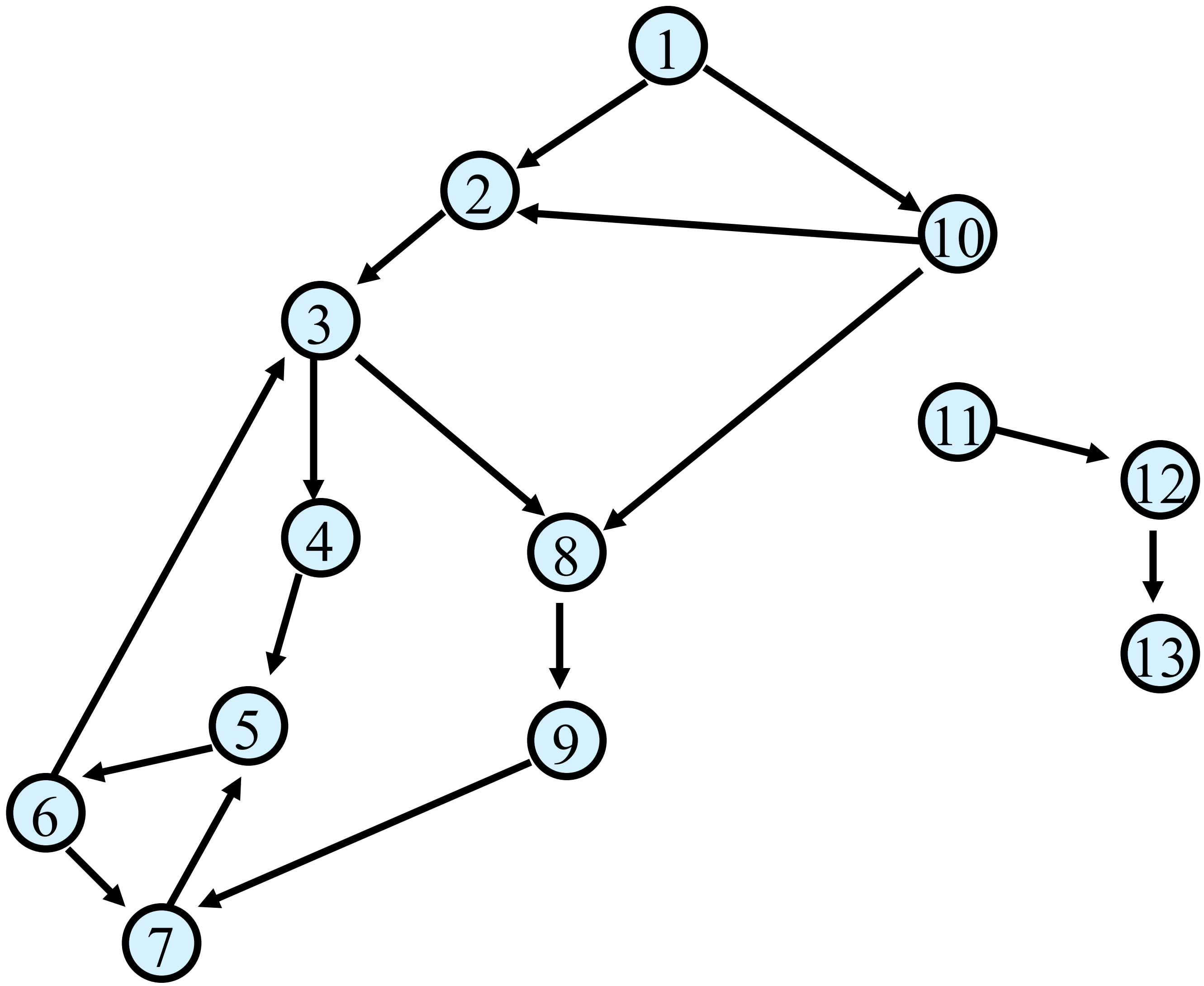




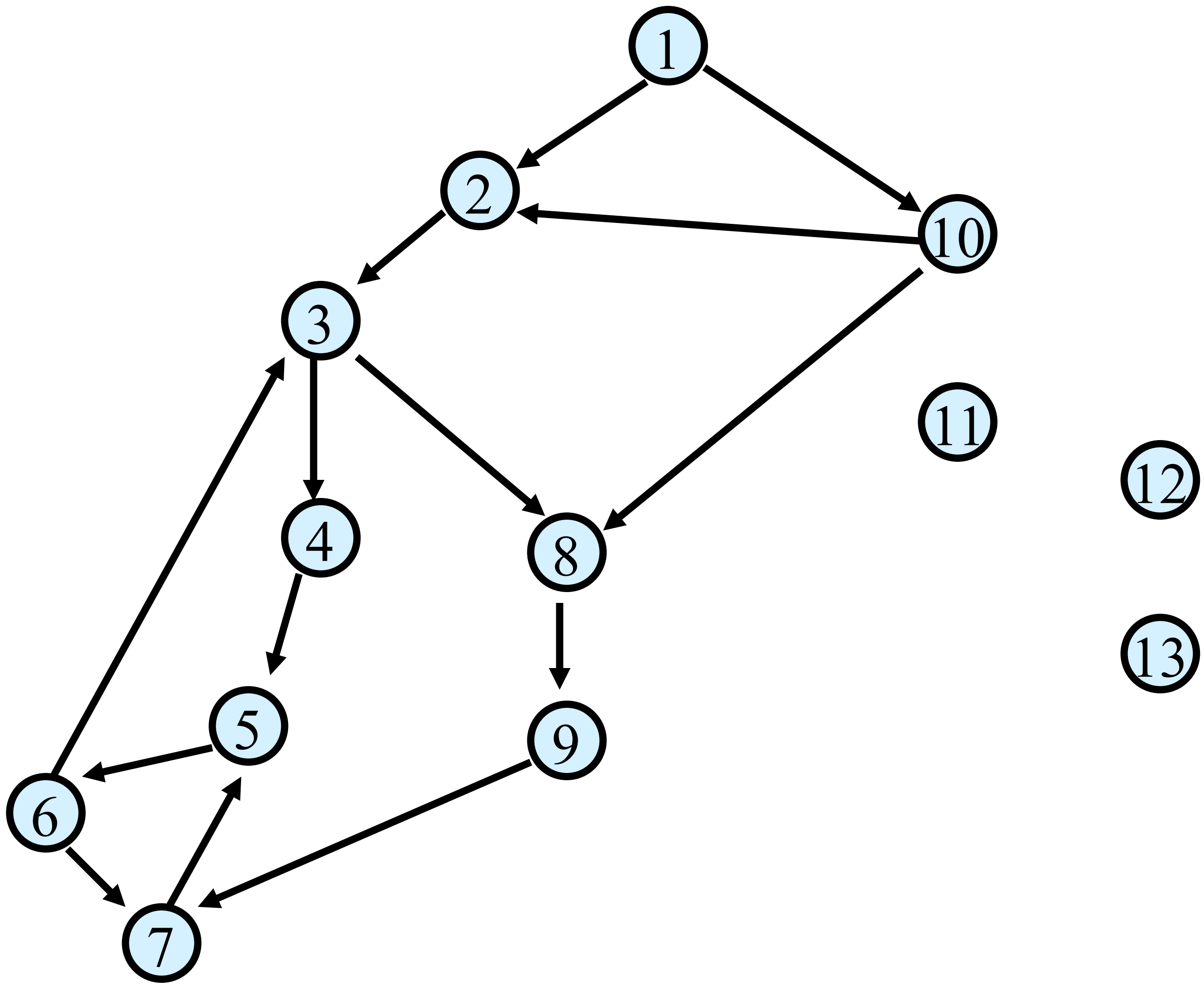
Directed Graph G = (V,E)



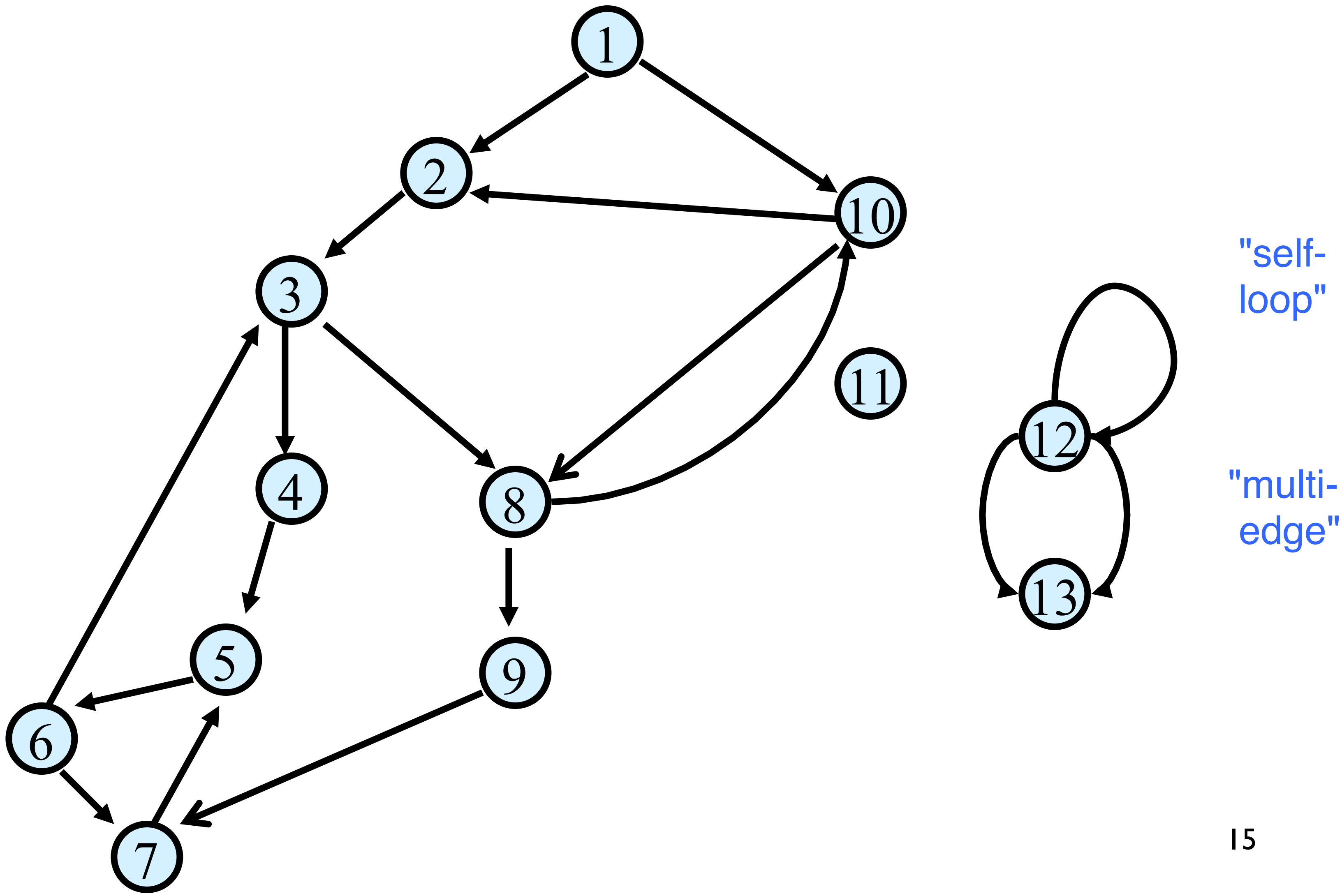
Directed Graph G = (V,E)



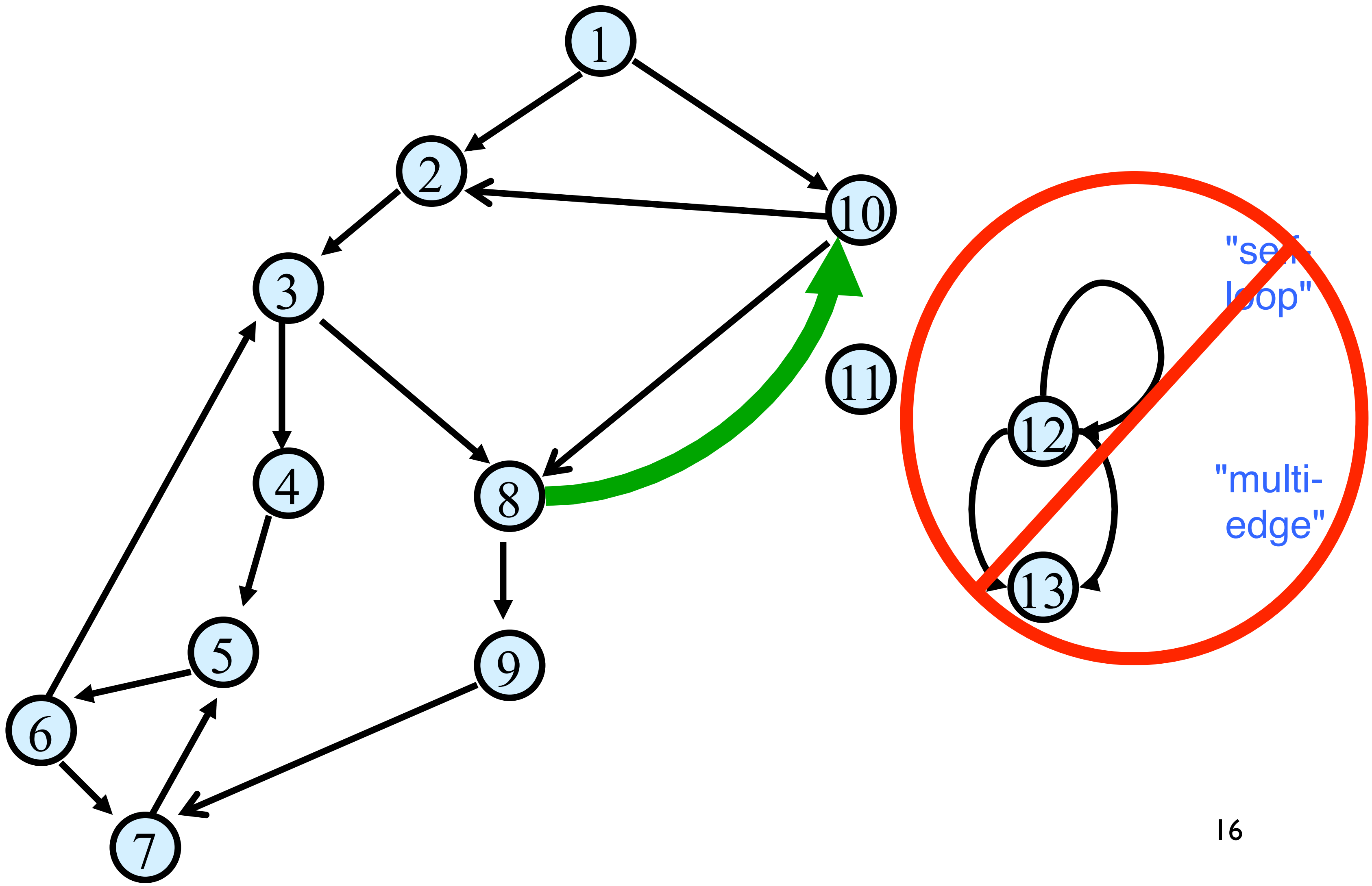
Directed Graph G = (V,E)



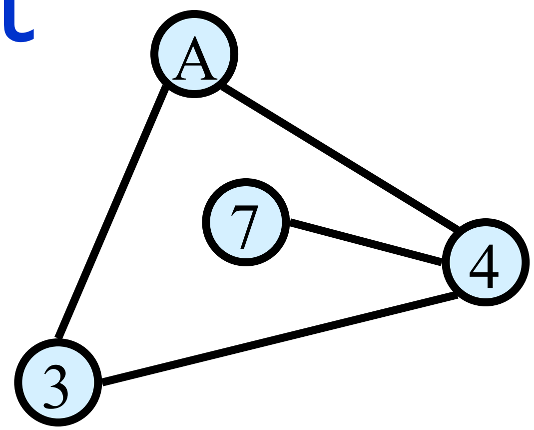
Directed Graph G = (V,E)



Directed Graph G = (V,E)



Specifying undirected graphs as input



What are the vertices?

Explicitly list them: {"A", "7", "3", "4"}

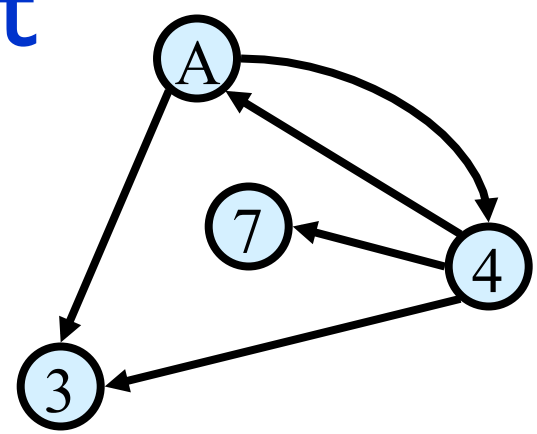
What are the edges?

One possibility:

(symmetric) adjacency matrix

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

Specifying directed graphs as input



What are the vertices?

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

What are the edges?

- (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?

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:

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

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 ($m \leq 3n-6$, for $n \geq 3$)

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

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

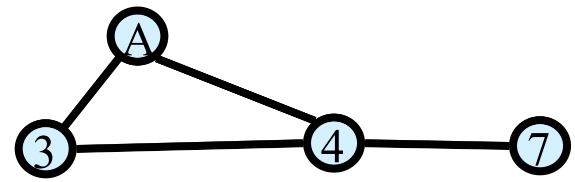
E.g., all planar graphs are sparse ($m \le 3n-6$, for $n \ge 3$)

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?

Disadvantages?

Representing Graph $G = (V,E)$

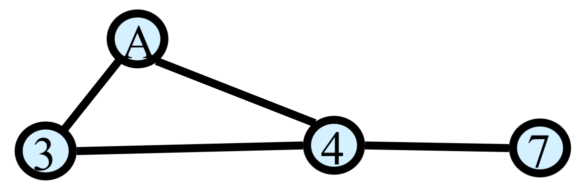
internally, indp of input format

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

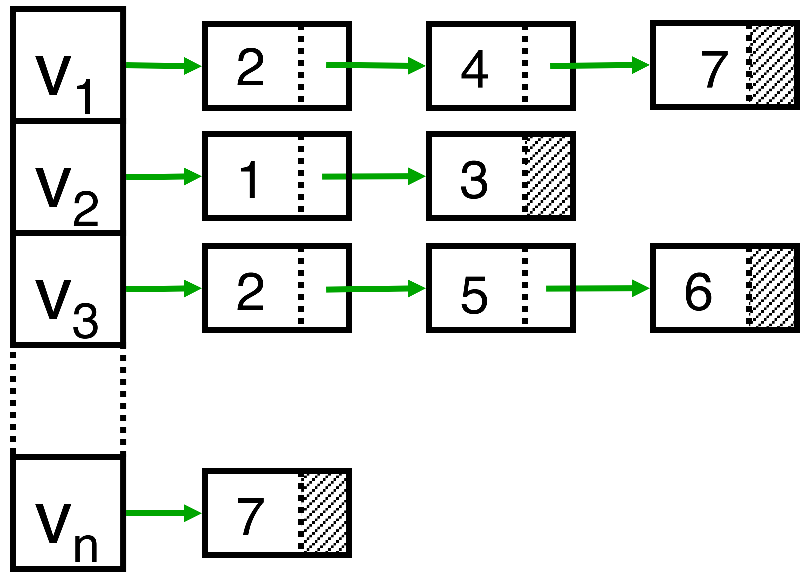
$m \ll n^2$

Representing Graph G=(V,E)

n vertices, m edges

Adjacency List:

O(n+m) words



Advantages?

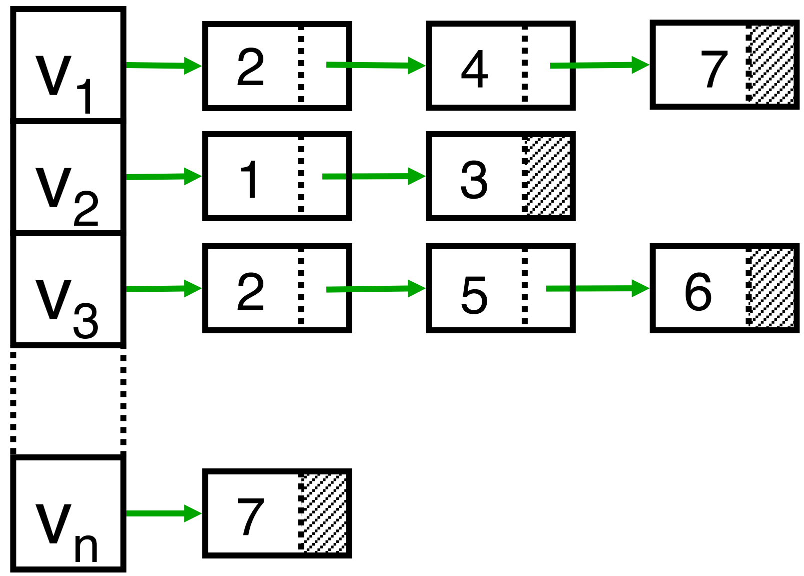
Disadvantages?

Representing Graph G=(V,E)

n vertices, m edges

Adjacency List:

O(n+m) words



Advantages:

Compact for sparse graphs

Disadvantages

More complex data structure

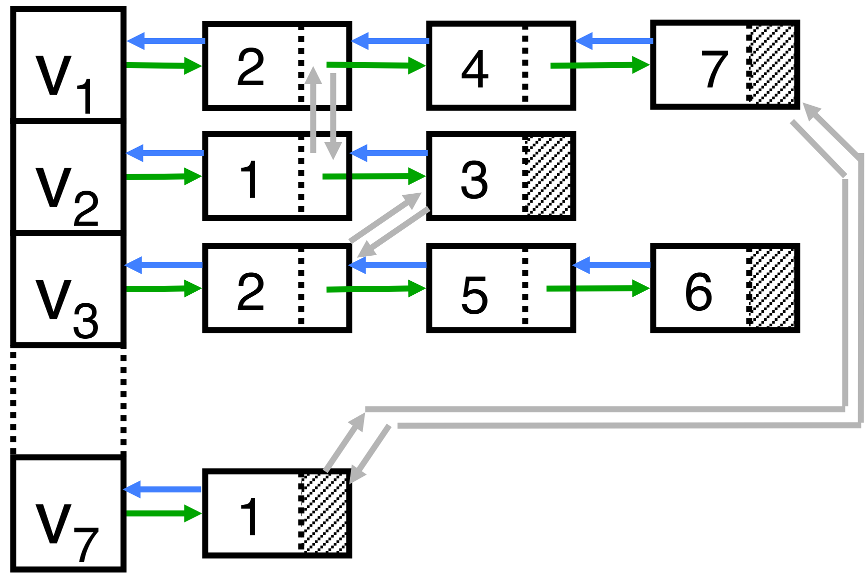
no O(1) edge test

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: order the nodes in successive layers based on distance from s
- Depth-First Search: more natural approach for exploring a maze; many efficient algs build on it.

Breadth-First Search

Completely explore the vertices in order of their distance from s

Naturally implemented using a queue

Graph Traversal: Implementation

Learn the basic structure of a graph

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

Three states of vertices

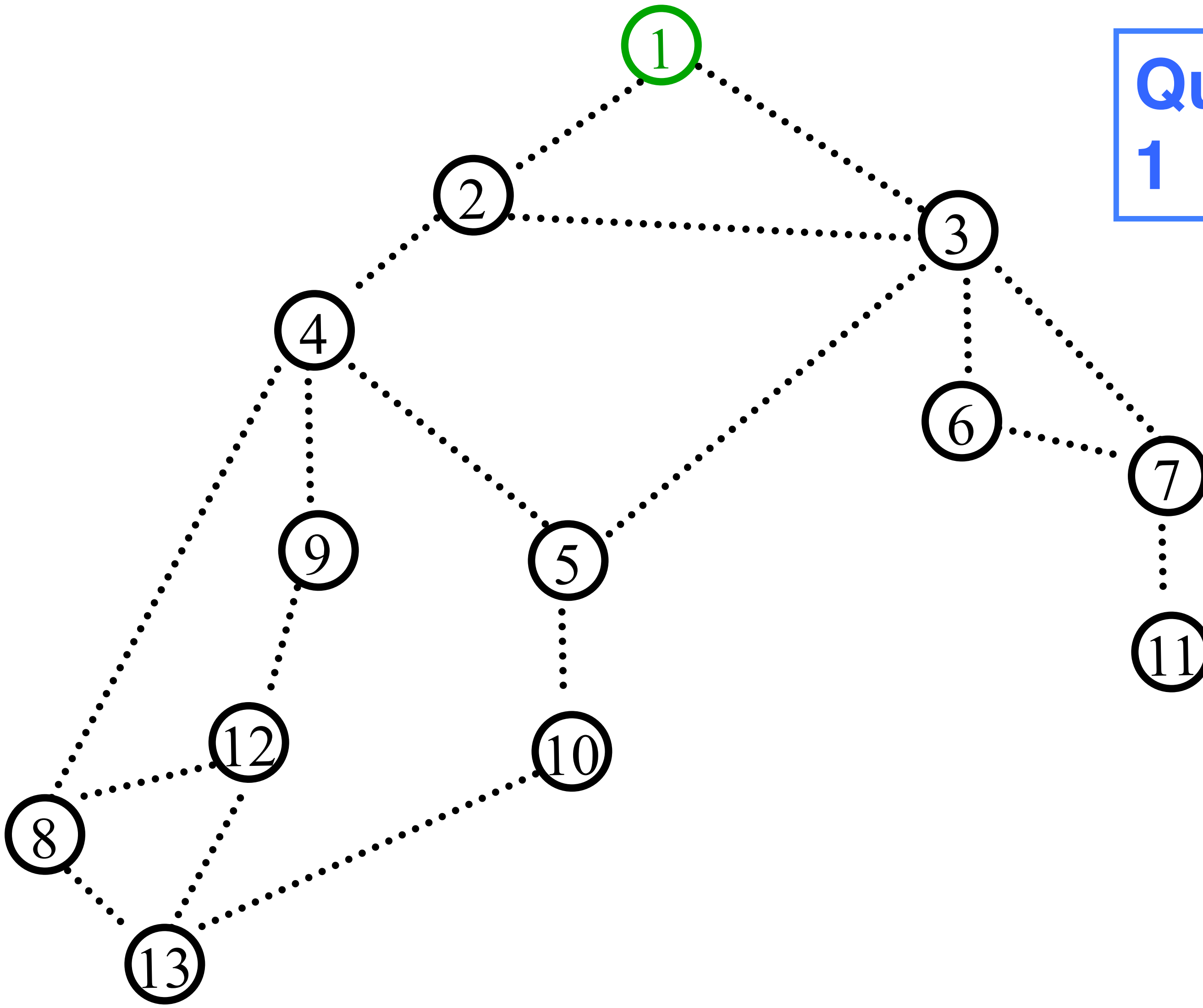
- *undiscovered*
- *discovered*
- *fully-explored*

BFS(s) Implementation

Global initialization: mark all vertices **"undiscovered"**

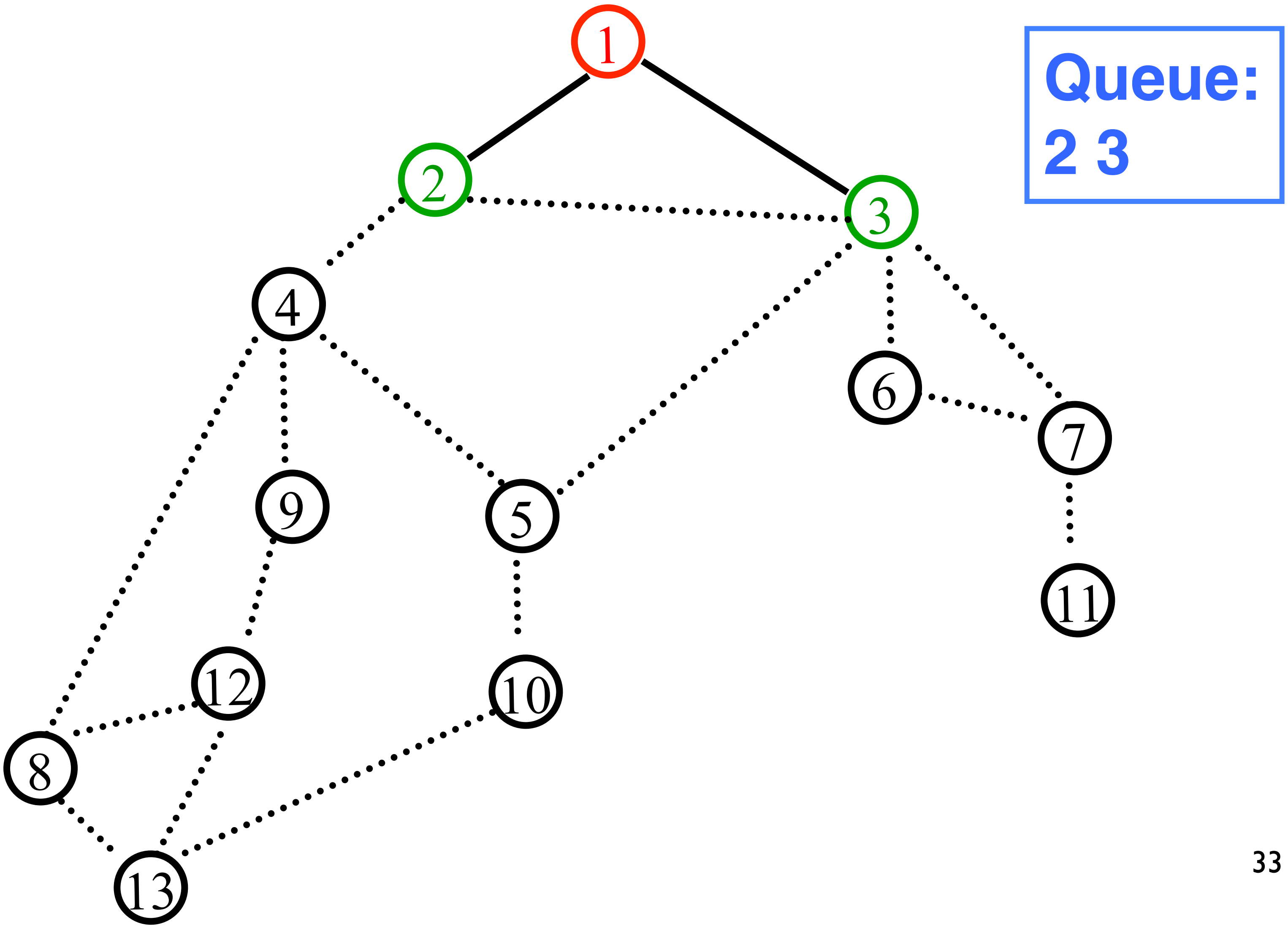
```
BFS(s)
    mark  s "discovered"
    queue = { s }
    while queue not empty
        u = remove_first(queue)
        for each edge {u,x}
            if (x is undiscovered)
                mark x discovered
                append x on queue
        mark u fully explored
```

BFS(v)

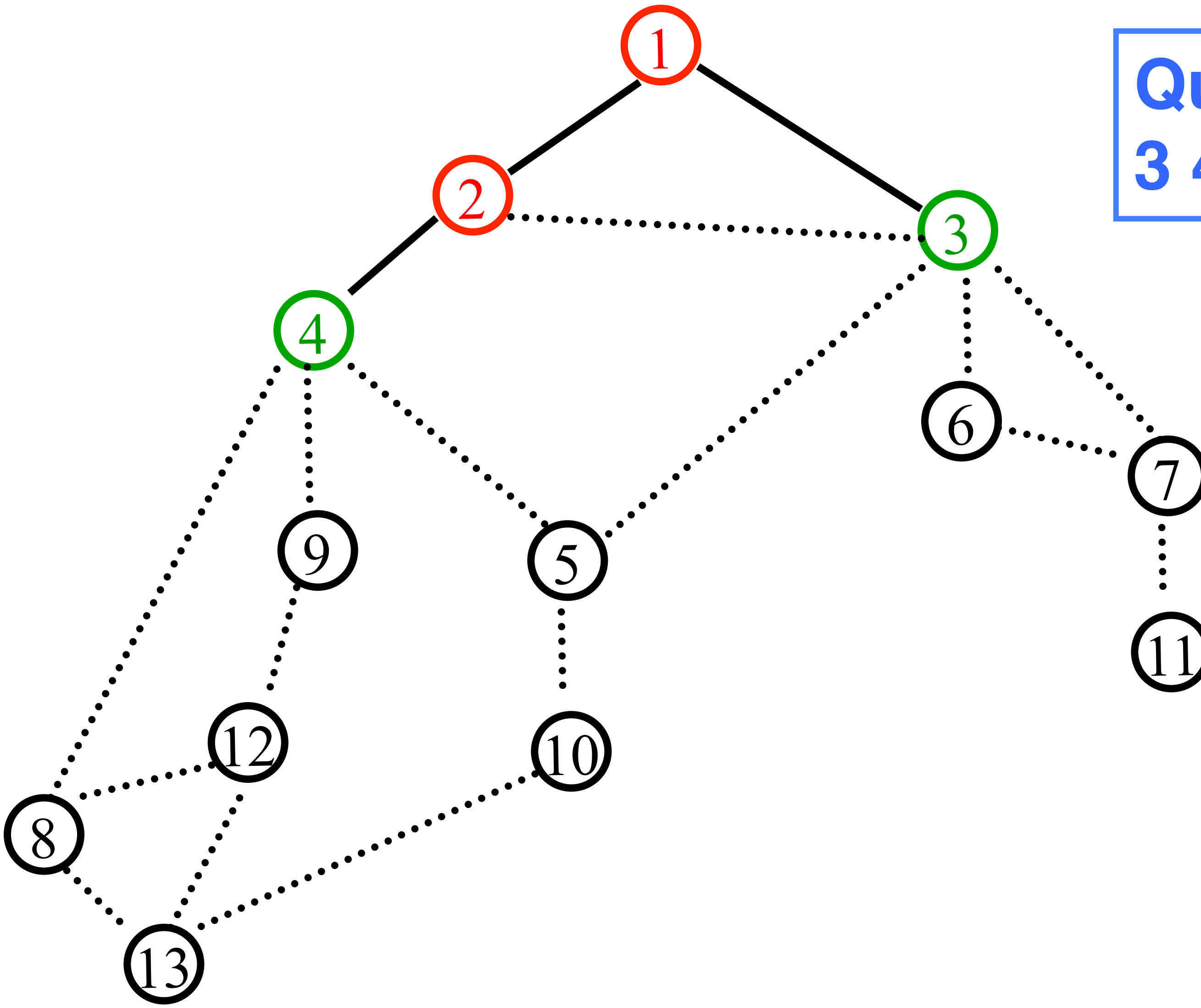


Queue:
1

BFS(v)

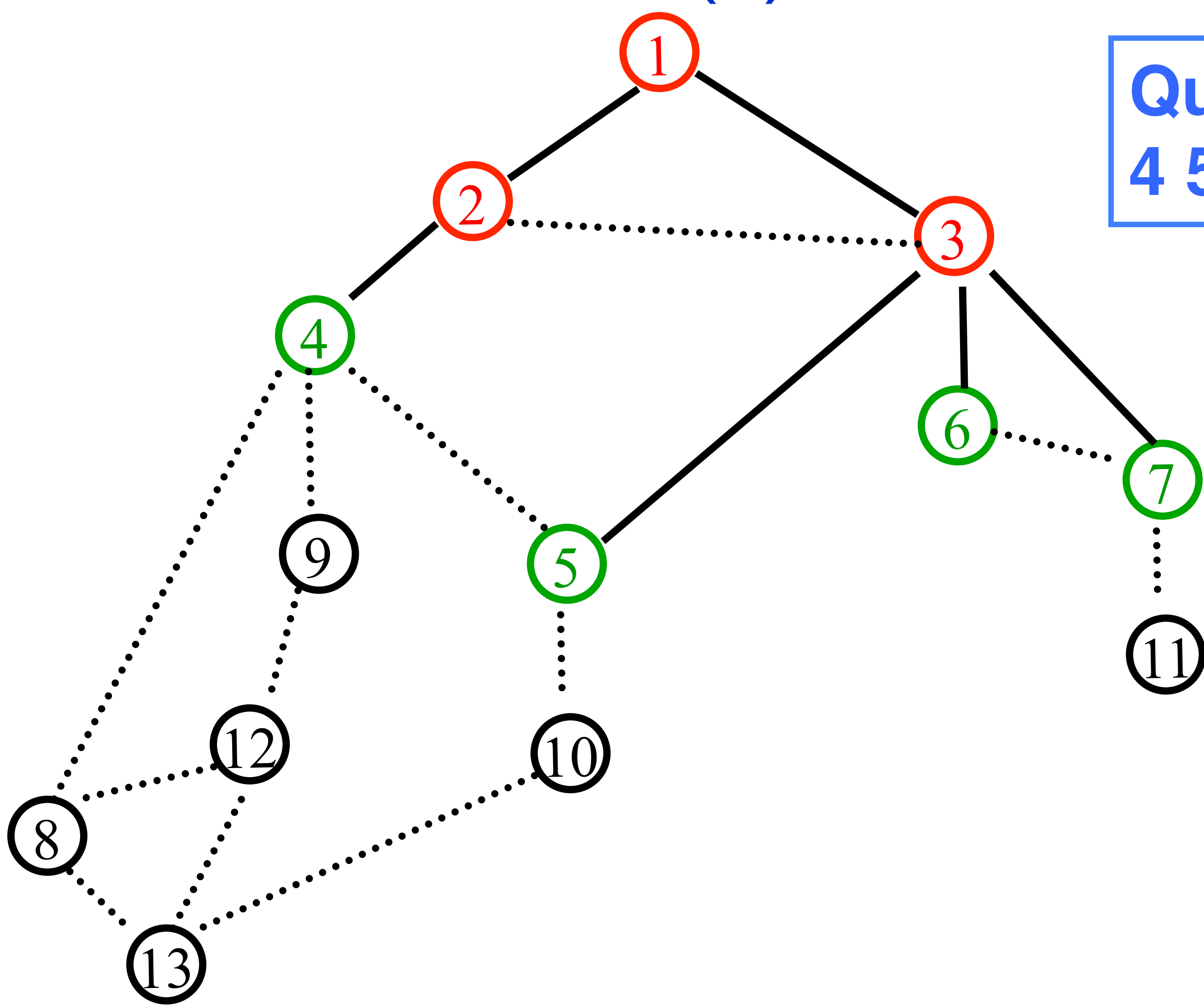


BFS(v)



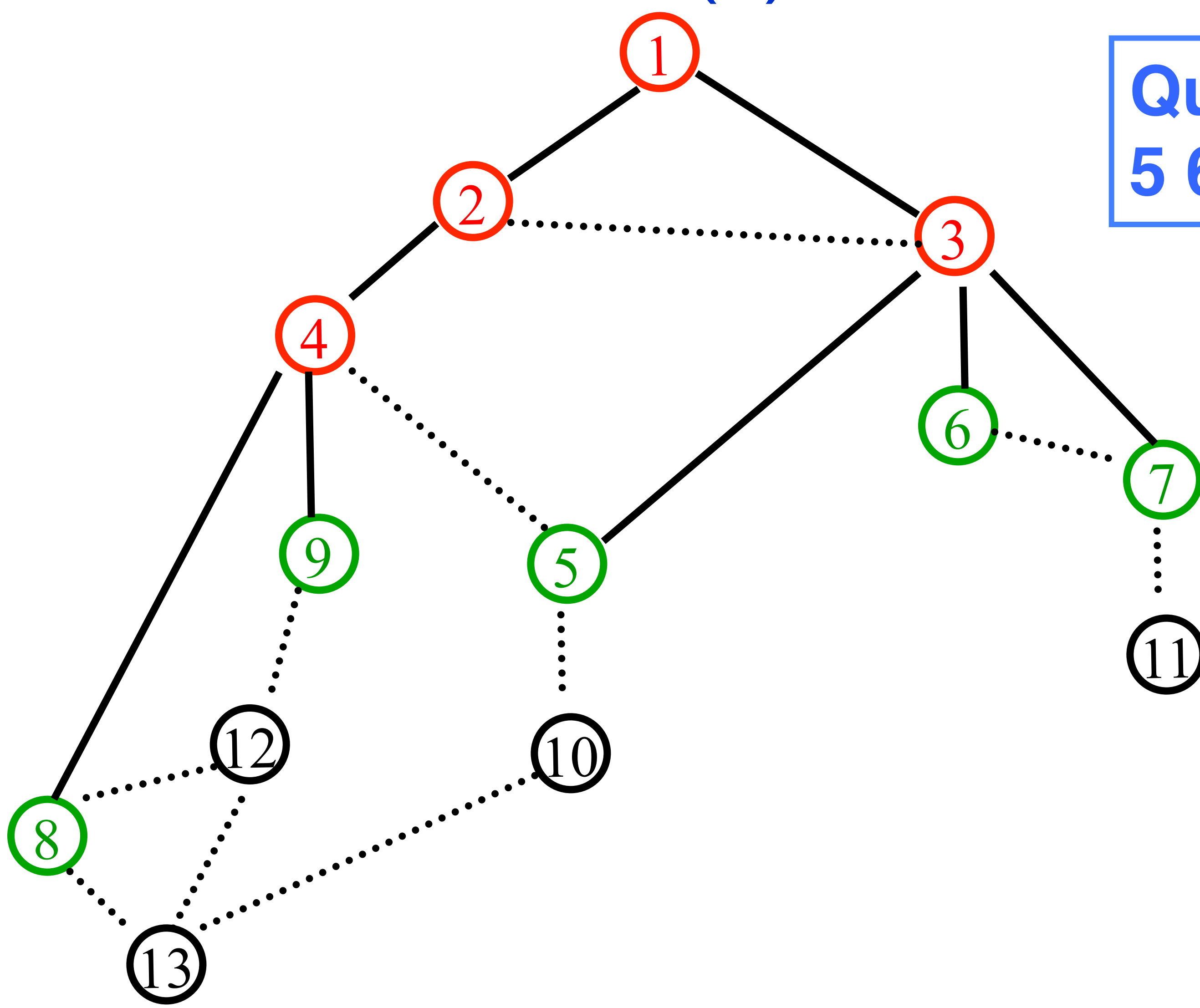
Queue:
3 4

BFS(v)



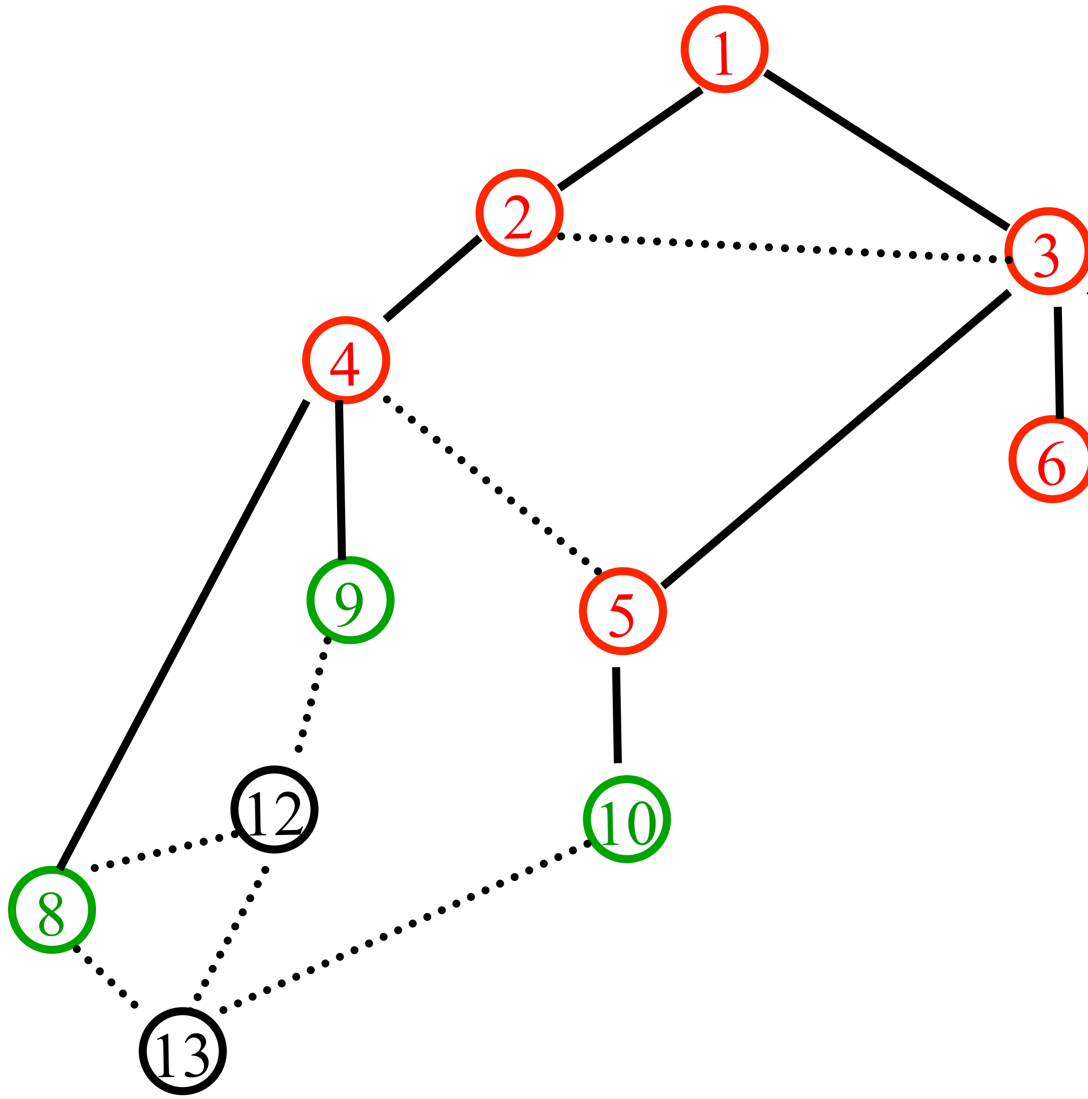
Queue:
4 5 6 7

BFS(v)



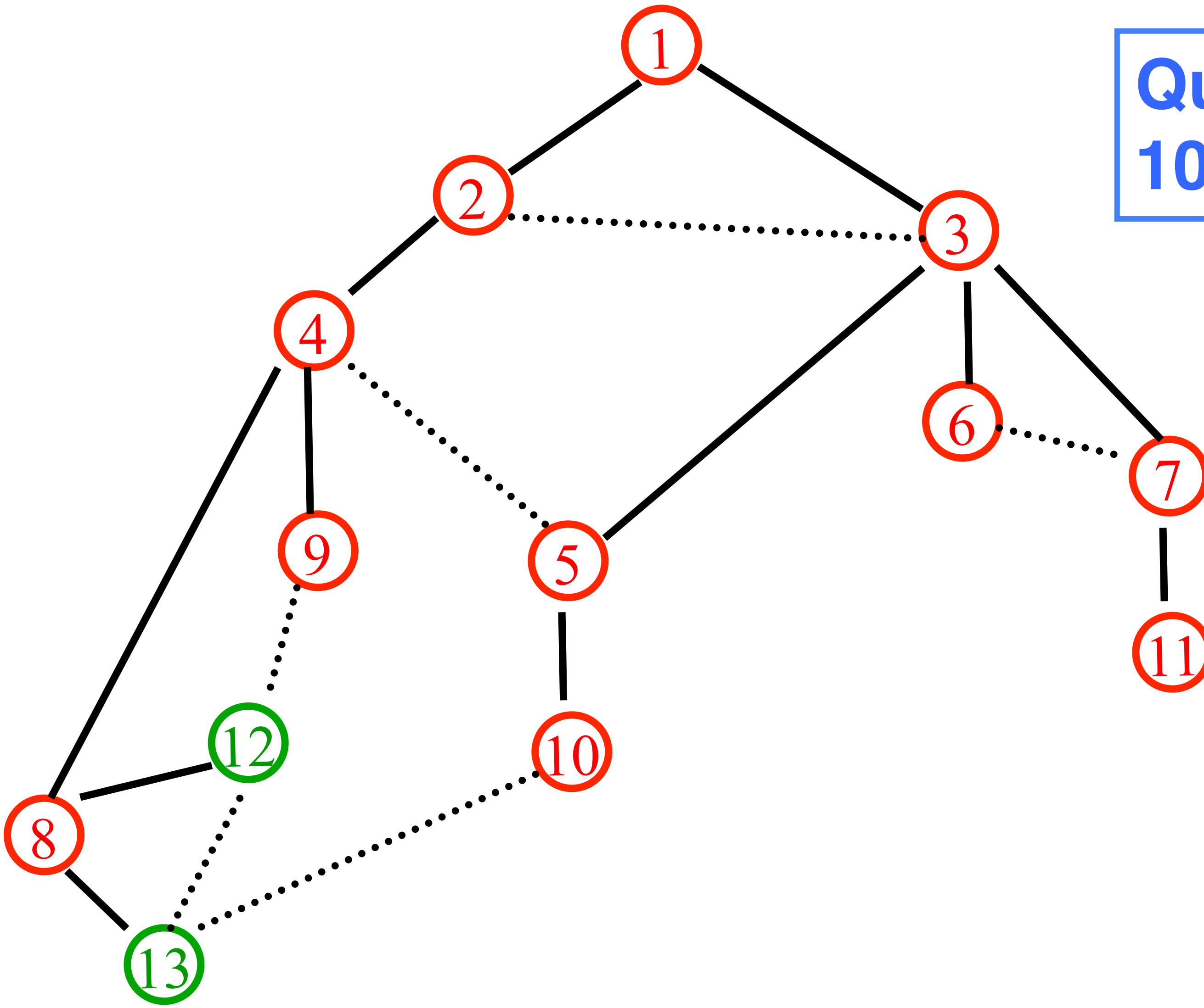
Queue:
5 6 7 8 9

BFS(v)



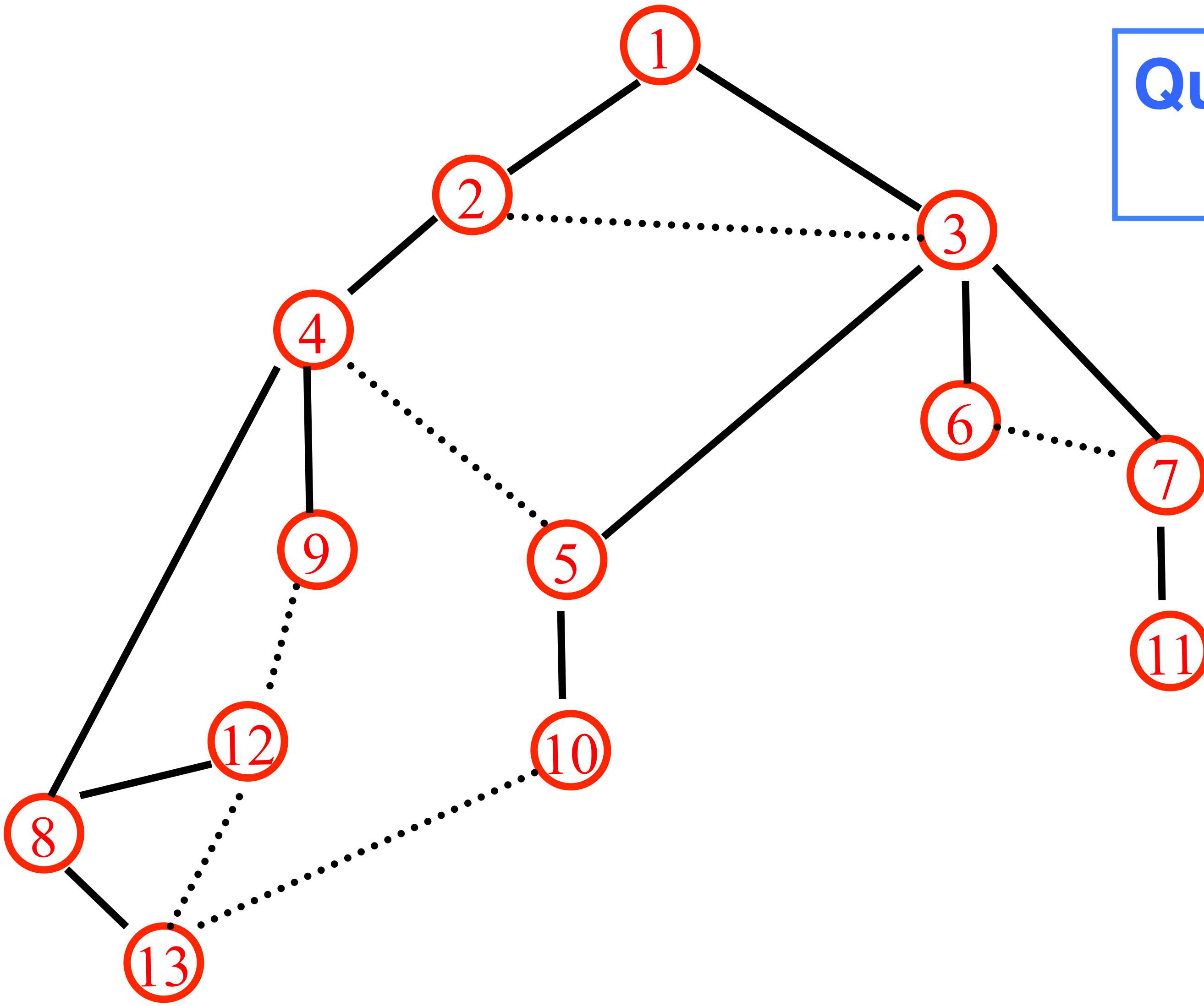
Queue:
8 9 10 11

BFS(v)



Queue:
10 11 12 13

BFS(v)



Queue:

BFS: Analysis, I

$O(n)$	Global initialization: mark all vertices "undiscovered"
+	BFS(s)
$O(1)$	mark s "discovered"
+	queue = { s }
$O(n)$	while queue not empty
x	u = remove_first(queue)
$O(n)$	for each edge {u,x}
	if (x is undiscovered)
	mark x discovered
	append x on queue
=	mark u fully explored
$O(n^2)$	

```
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
```

Simple analysis: 2 nested loops. Get worst-case number of iterations of each; multiply.

BFS: Analysis, II

Above analysis correct, but pessimistic (can't have $\Omega(n)$ edges incident to each of $\Omega(n)$ distinct "u" vertices if G is sparse). Alt, more global analysis:

Each edge is explored once from each end-point, so *total* runtime of inner loop is O(m).

Exercise: extend algorithm and analysis to non-connected graphs

Total O(n+m), n = # nodes, m = # edges

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

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

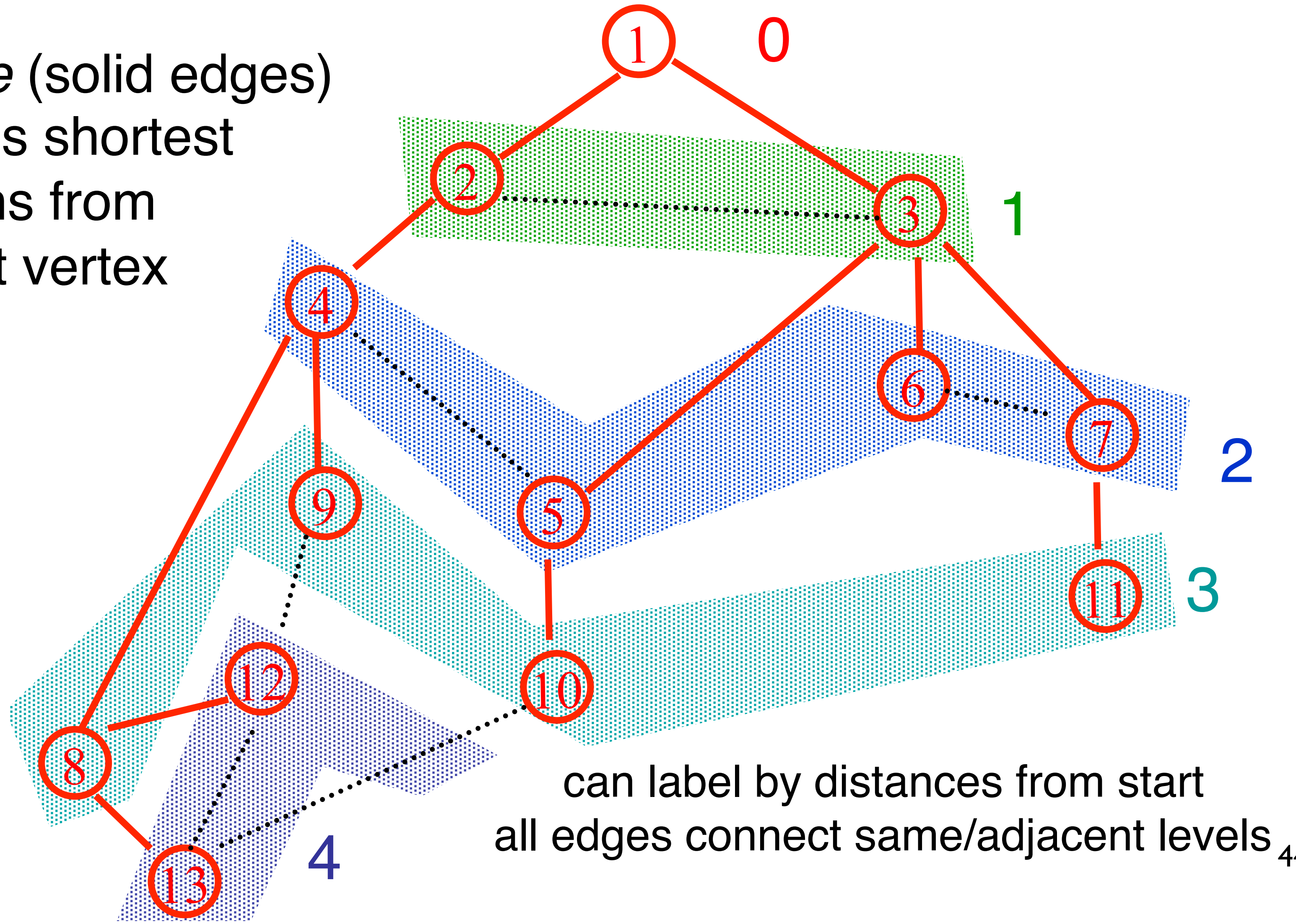
Level i in this tree are exactly those vertices *u* such that the shortest path (in G, not just the tree) from the root v is of length i.

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

not true of every spanning tree!

BFS Application: Shortest Paths

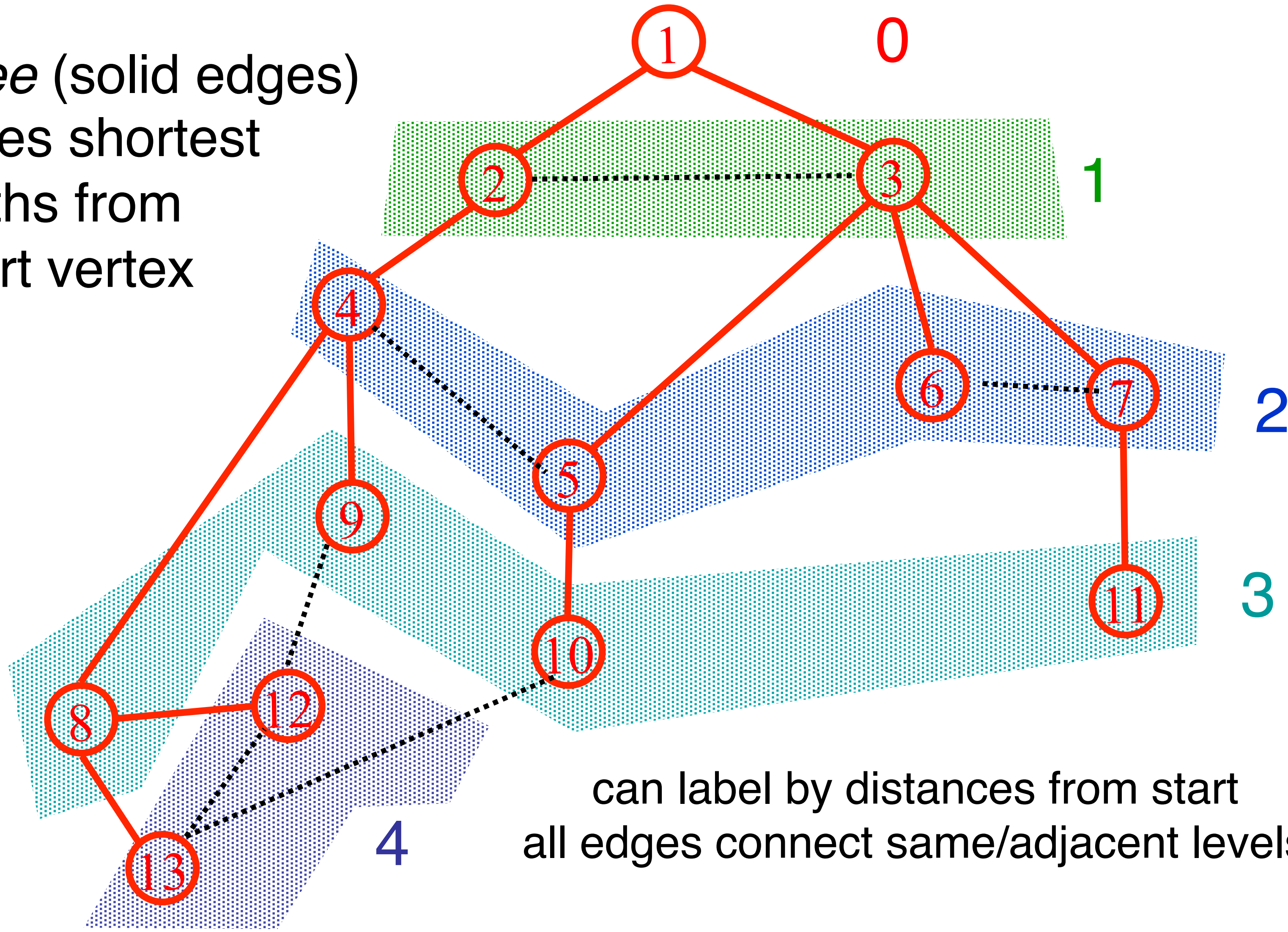
Tree (solid edges) gives shortest paths from start vertex



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

BFS Application: Shortest Paths

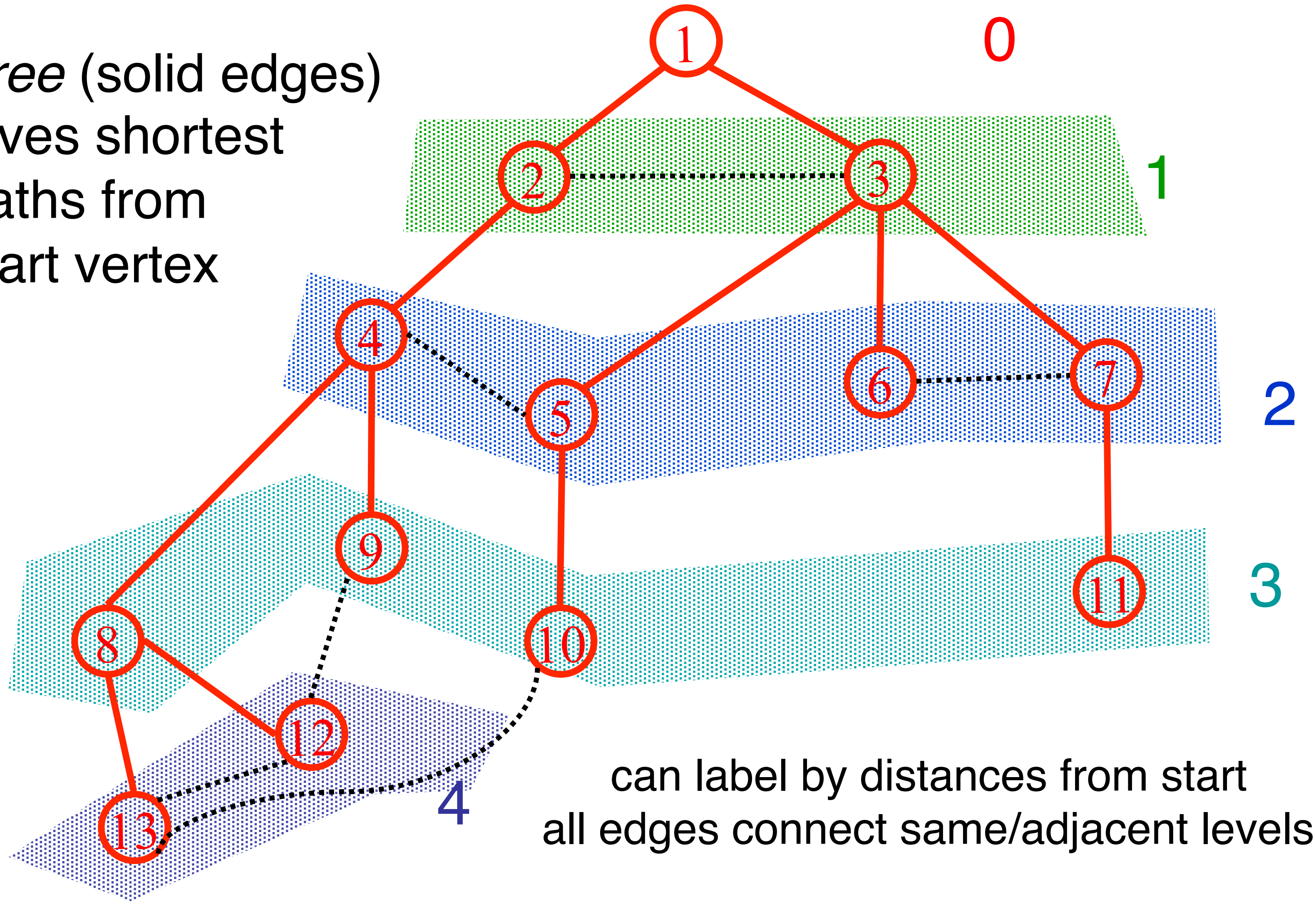
Tree (solid edges) gives shortest paths from start vertex



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

BFS Application: Shortest Paths

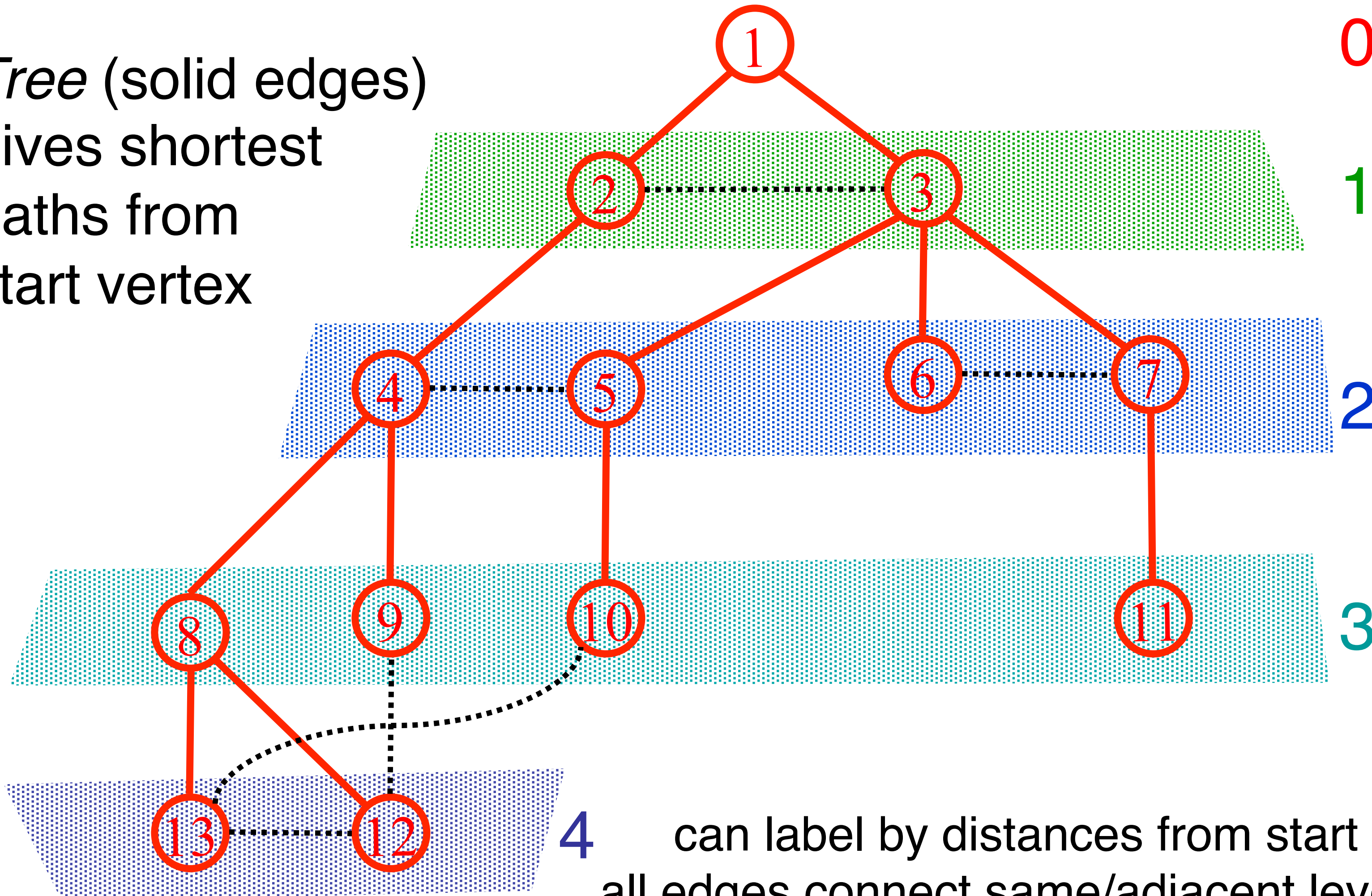
Tree (solid edges) gives shortest paths from start vertex



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

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 compute level numbers

Label edges as tree edges or non-tree edges (within/between)

Number of distinct shortest paths

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?

Set up one-time data structure to answer such questions efficiently.

Graph Search Application: Connected Components

```
initial state: all v undiscovered
for v = 1 to n do
    if state(v) != fully-explored then
        BFS(v)
    endif
endfor
```

Exercise: modify code to answer CC queries

Total cost: O(n+m)

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

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

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