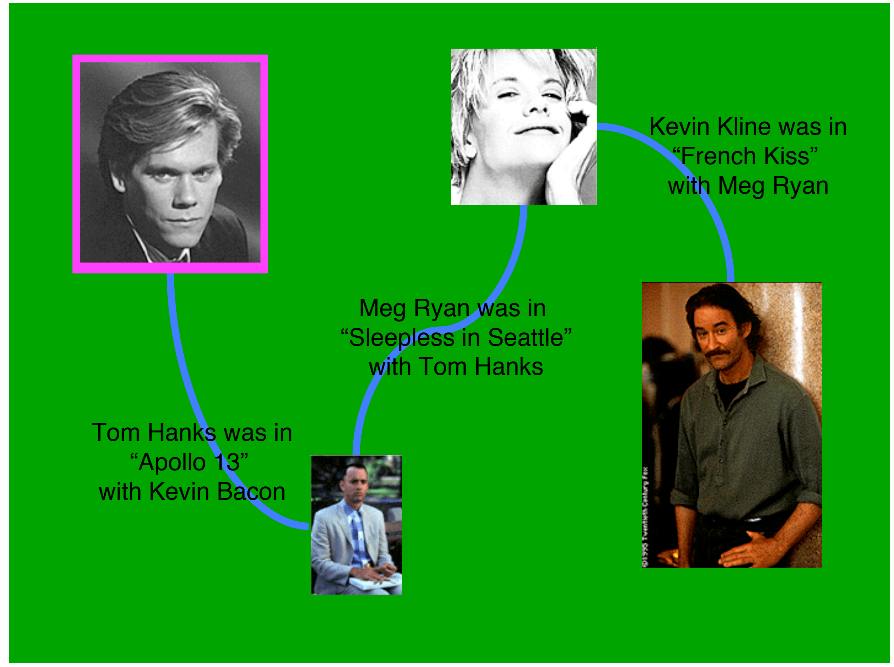
CSE 417: Algorithms and Computational Complexity

Winter 2006

Graphs and Graph Algorithms

Larry Ruzzo



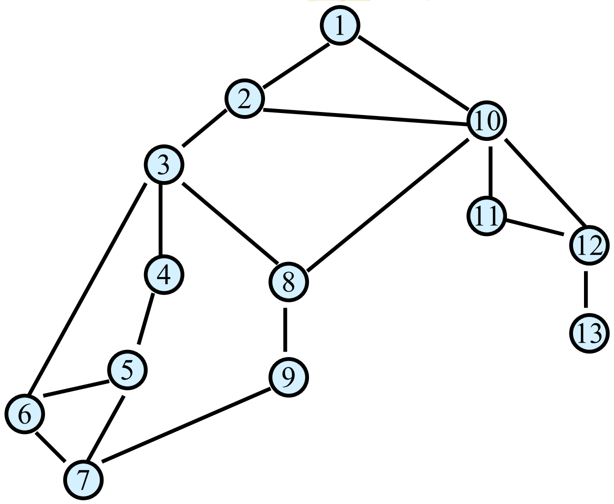
Objects & Relationships

- 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

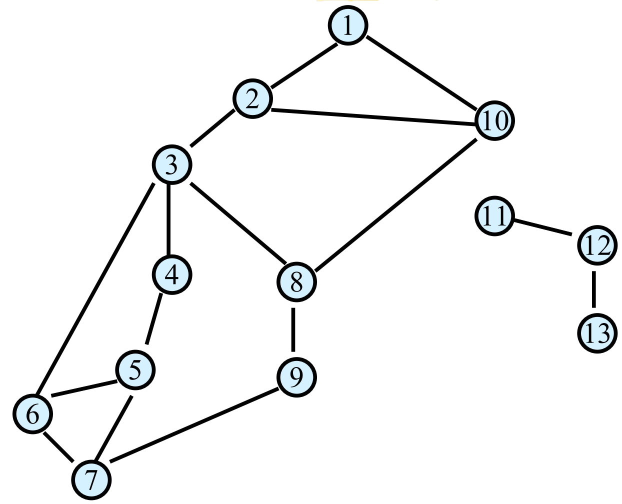
Graphs

- 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

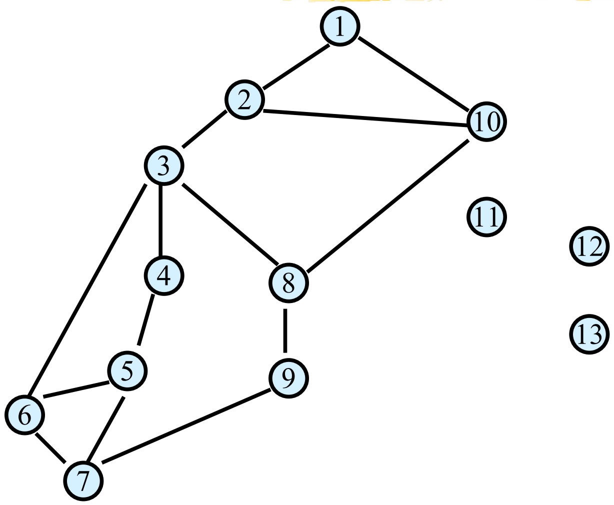
Undirected Graph G = (V,E)



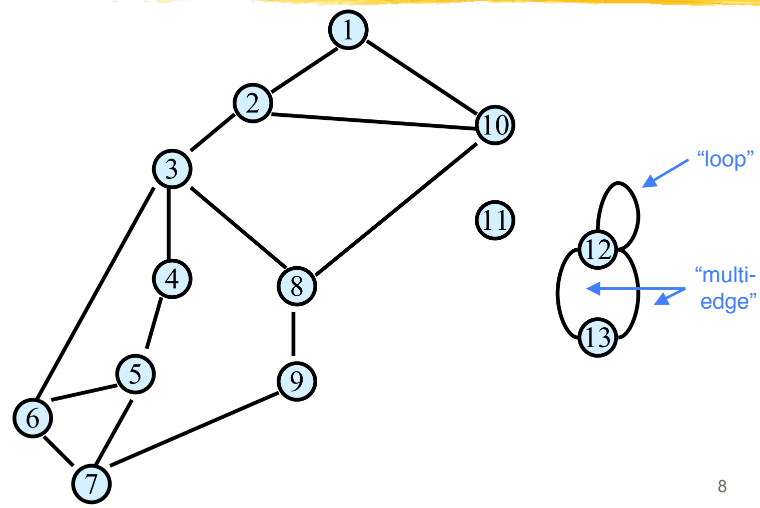
Undirected Graph G = (V,E)



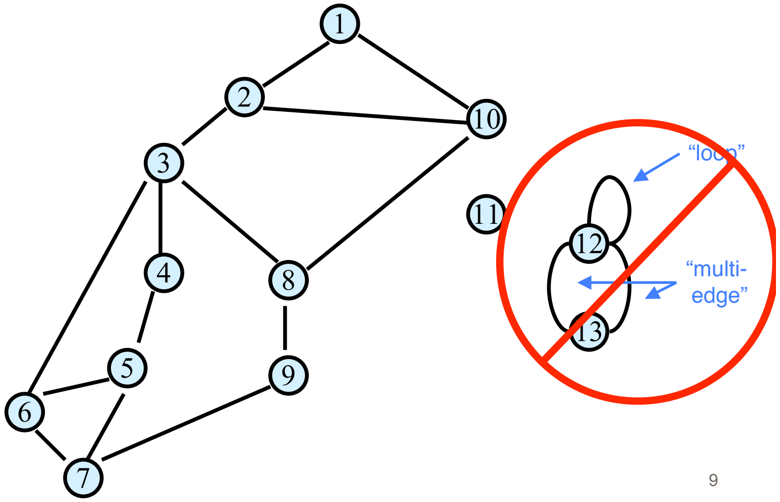
Undirected Graph G = (V,E)



Undirected Graph G = (V,E)

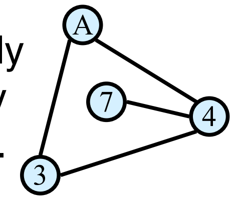


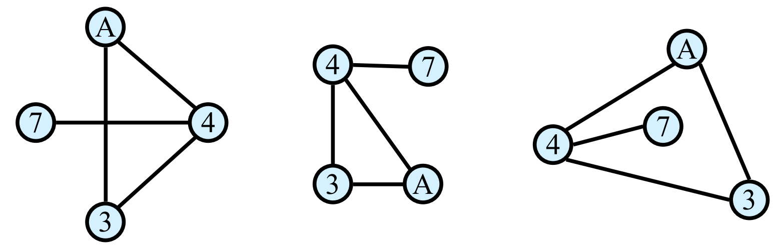
Undirected Graph G = (V,E)



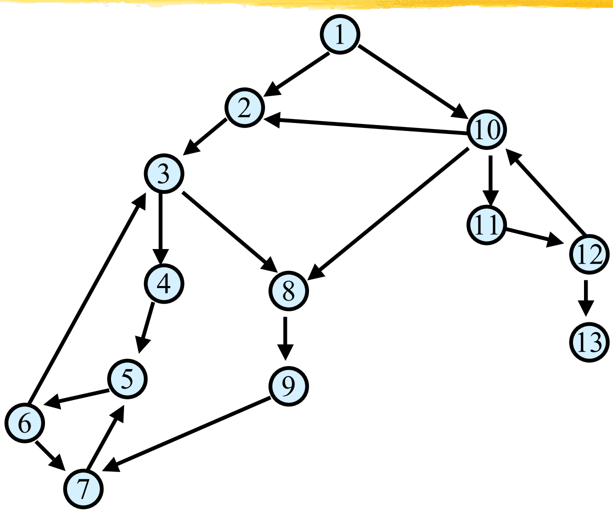
Graphs don’t live in Flatland

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

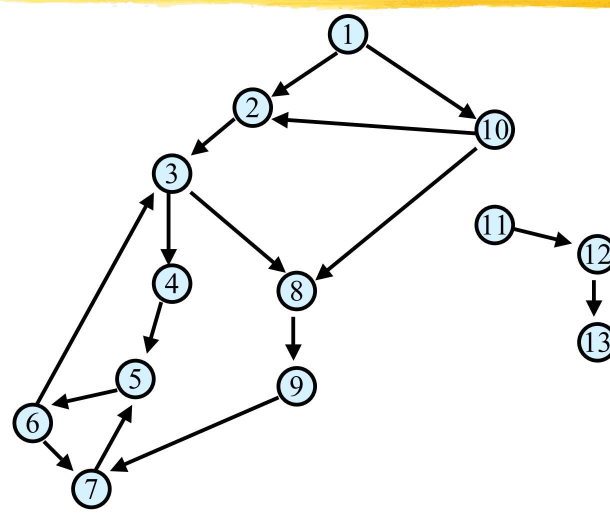




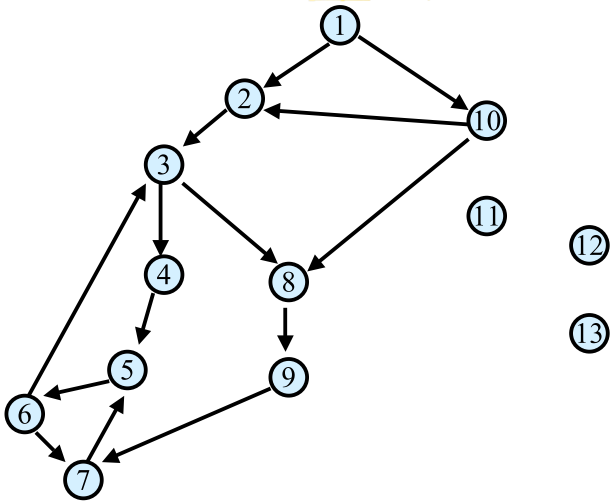
Directed Graph G = (V,E)



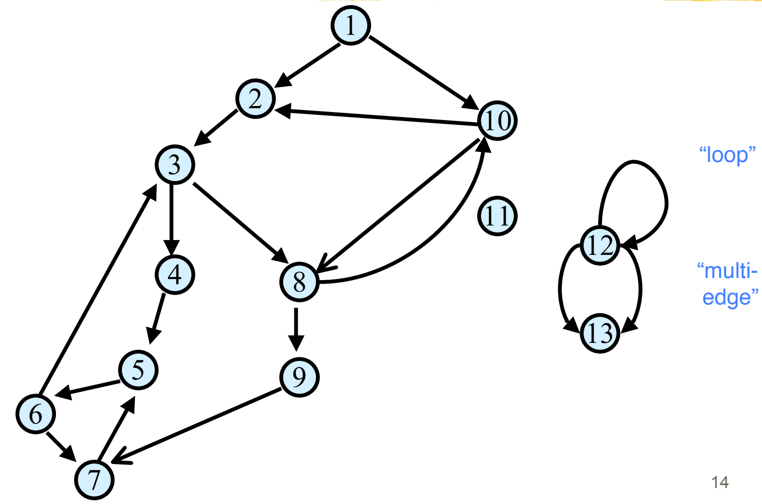
Directed Graph G = (V,E)



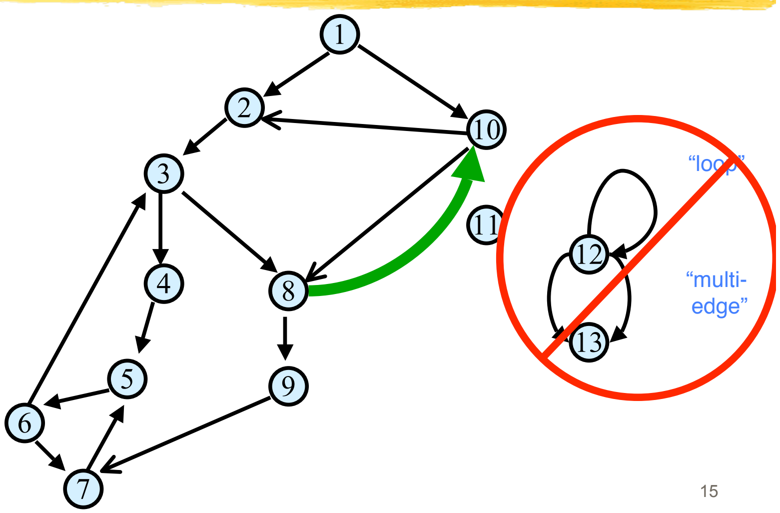
Directed Graph G = (V,E)



Directed Graph G = (V,E)

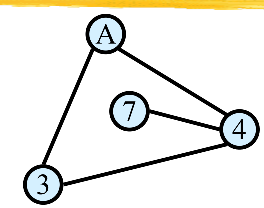


Directed Graph G = (V,E)



Specifying undirected graphs as input

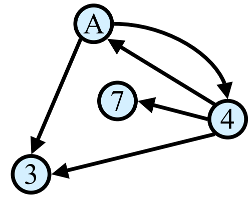
- What are the vertices?
  - Explicitly list them: {“A”, “7”, “3”, “4”}
- What are the edges?
  - Either, set of edges {{A,3}, {7,4}, {4,3}, {4,A}}
  - Or, (symmetric) adjacency matrix:



	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
  - 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: $0 \le m \le n(n-1)/2 = O(n^2)$

More Cool Graph Lingo

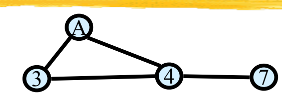
- A graph is called *sparse* if $m << n^2$, otherwise it is *dense*
  - Boundary is somewhat fuzzy; O(n) edges is certainly sparse, $\Omega(n^2)$ edges is dense.
- Sparse graphs are common in practice
  - E.g., all planar graphs are sparse
- Q: which is a better run time, O(n+m) or $O(n^2)$?

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



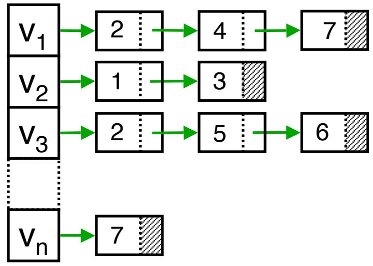
	A	7	3	4
A	0	0	1	1
7	0	0	0	1
3	1	0	0	1
4	1	1	1	0

- Advantages:
  - O(1) test for presence or absence of edges.
- Disadvantages: inefficient for sparse graphs, both in storage and access

$m << n^2$

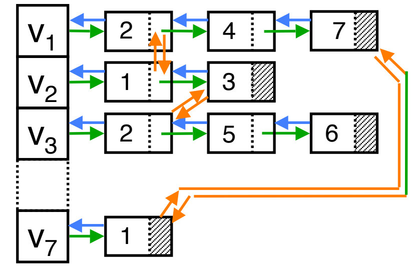
Representing Graph G=(V,E)
n vertices, m edges

- 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 v to all vertices reachable from v

- Three states of vertices
  - **undiscovered**
  - **discovered**
  - **fully-explored**

Breadth-First Search

- Completely explore the vertices in order of their distance from v

- Naturally implemented using a queue

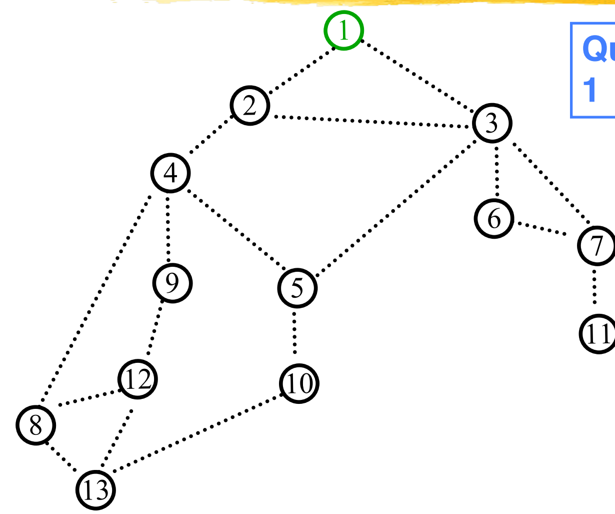
BFS(v)

Global initialization: mark all vertices "undiscovered"

```
BFS(v)
    mark  v "discovered"
    queue = v
    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 completed
```

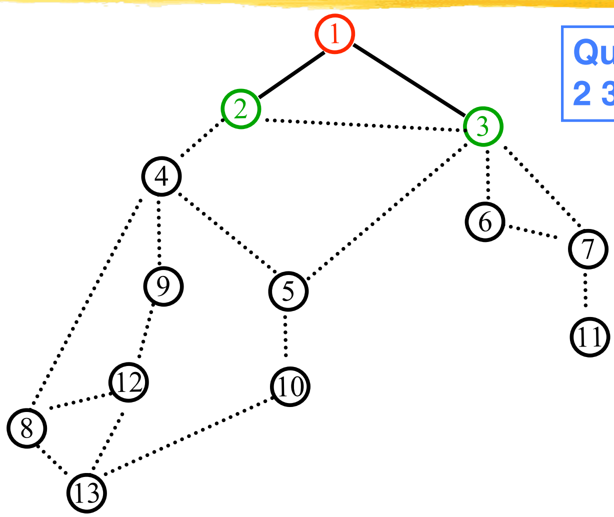
Exercise: modify code to number vertices & compute level numbers

BFS(v)



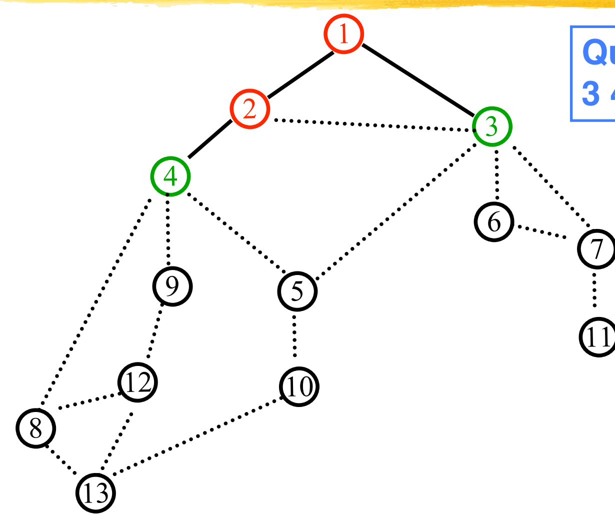
Queue: 1

BFS(v)



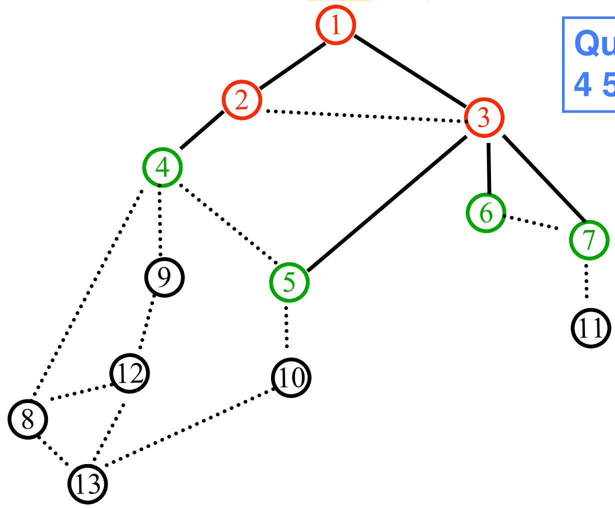
Queue: 2 3

BFS(v)



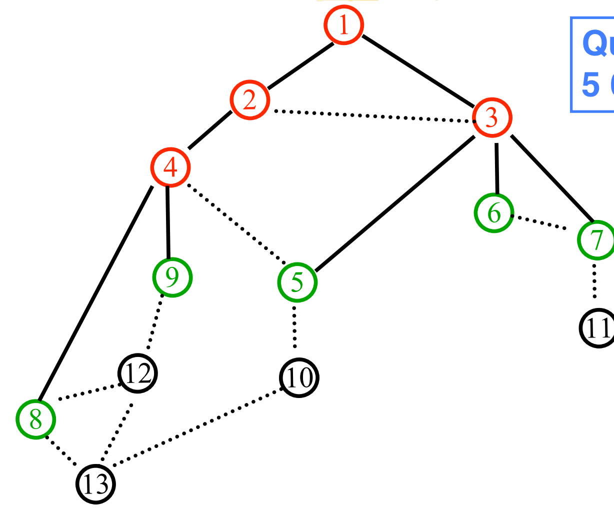
Queue: 3 4

BFS(v)



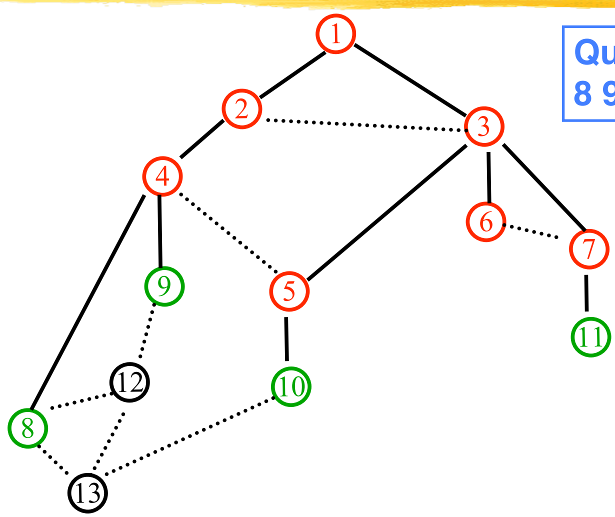
Queue:
4 5 6 7

BFS(v)



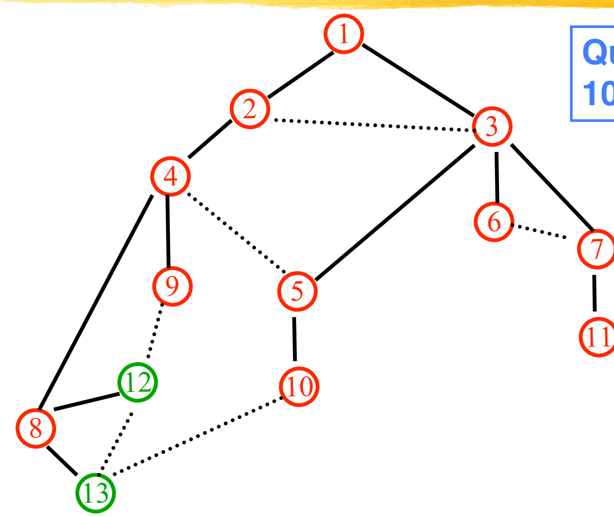
Queue:
5 6 7 8 9

BFS(v)



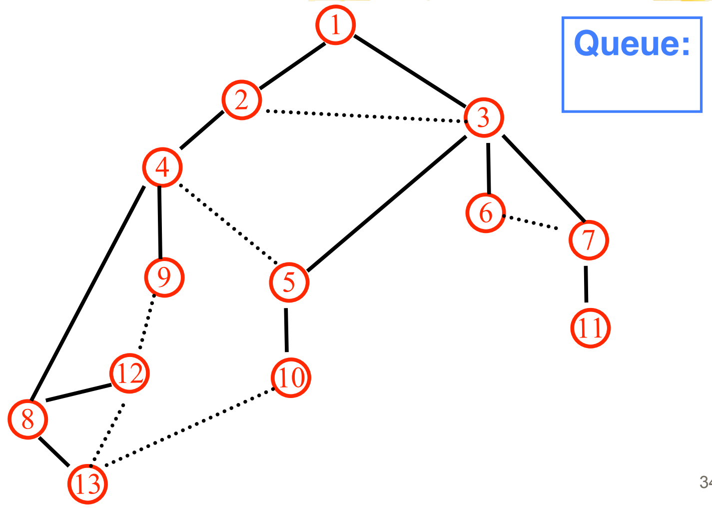
Queue:
8 9 10 11

BFS(v)



Queue:
10 11 12 13

BFS(v)



BFS analysis

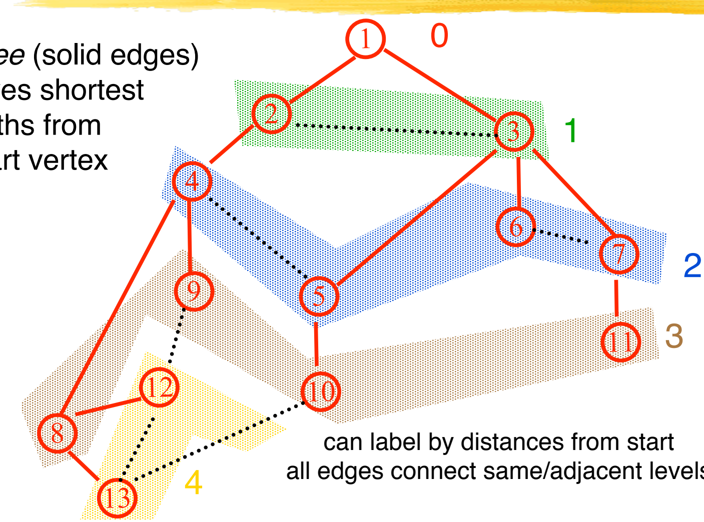
- Each edge is explored once from each end-point (at most)
- Each vertex is discovered by following a different edge
- Total cost O(m) where m=# of edges

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
- 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

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 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
  - works also with DFS

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