Graph Algorithms

Imran Rashid

University of Washington

Jan 11, 2008

Lecture Outline

Lecture Outline

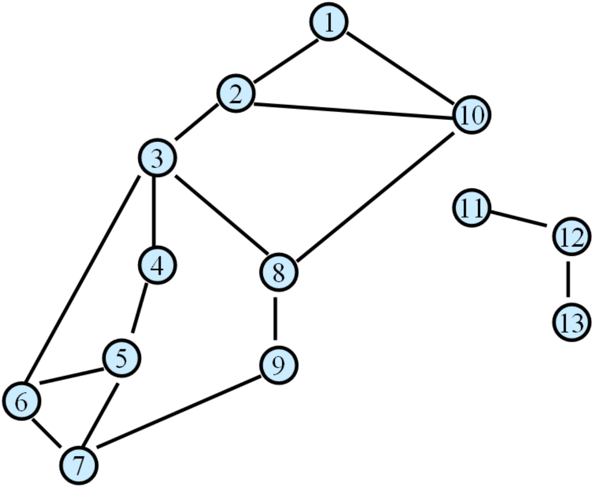
Objects & Relationships

- 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

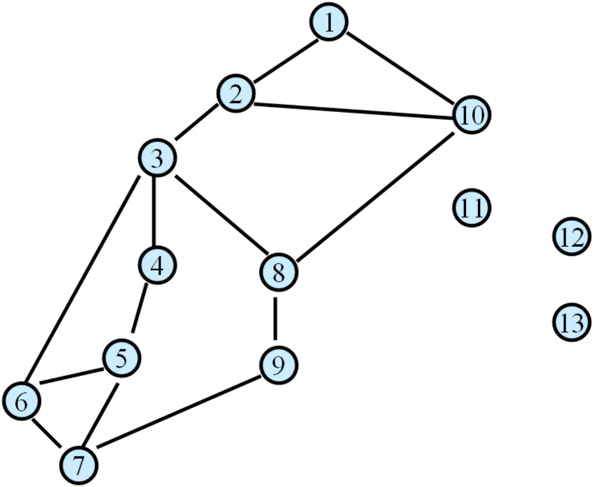
- An extremely important formalism for representing (binary) relationships
- Objects: "vertices", aka "nodes"
- Relationships between pairs: "edges", aka "arcs"
- Formally, a graph $G = (V, E)$ is a pair of sets, V the vertices and E the edges

Undirected Graph G = (V,E)



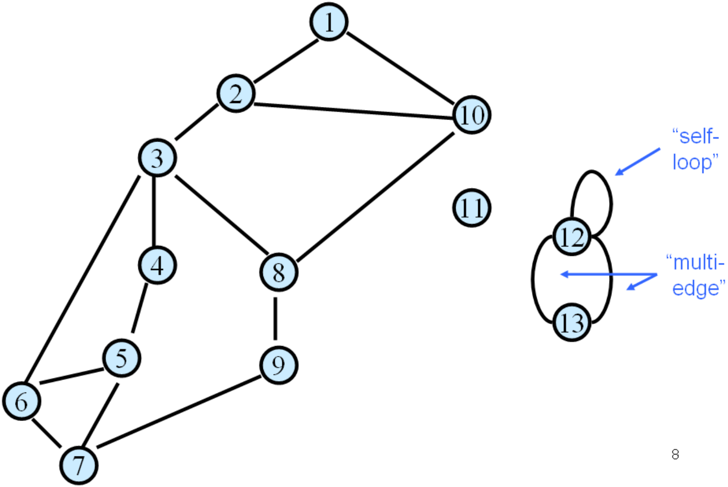
6

Undirected Graph G = (V,E)



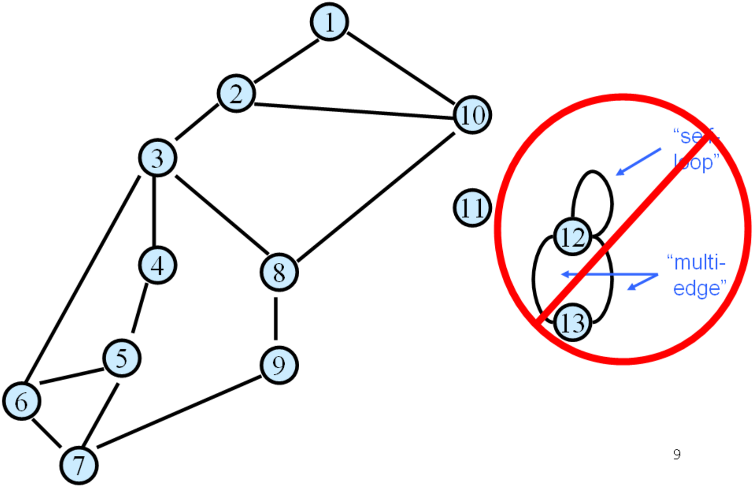
7

Undirected Graph G = (V,E)



8

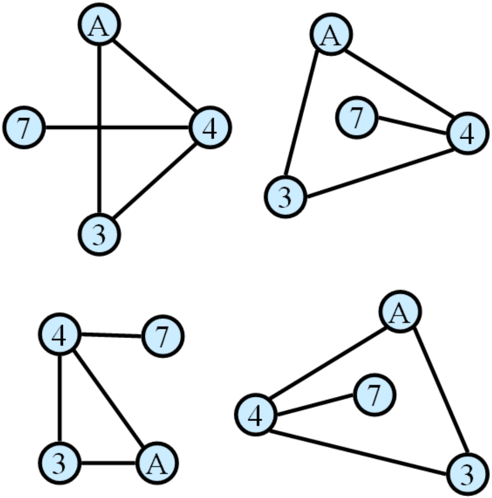
Undirected Graph G = (V,E)



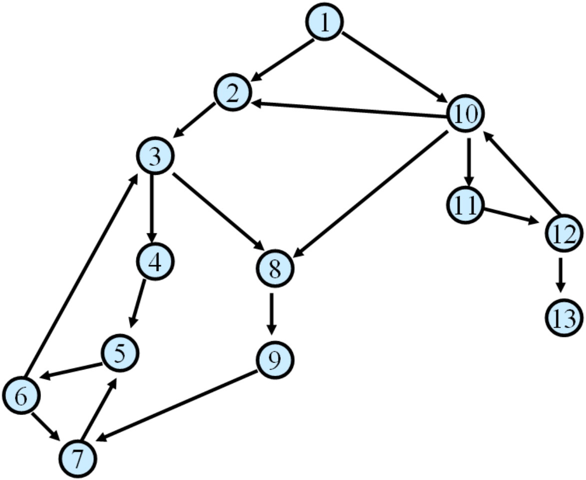
9

Graphs don't live in Flatland

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

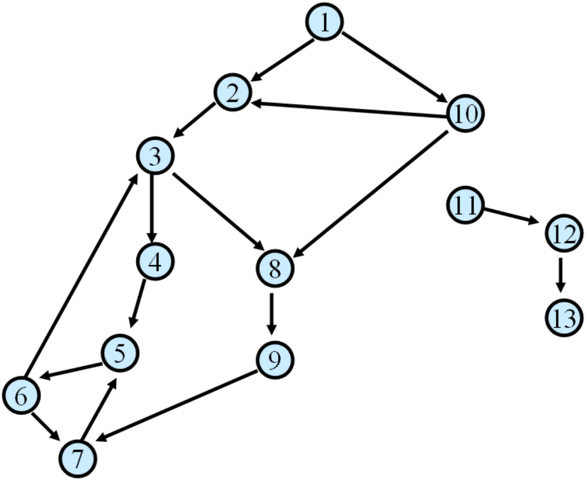


Directed Graph G = (V,E)



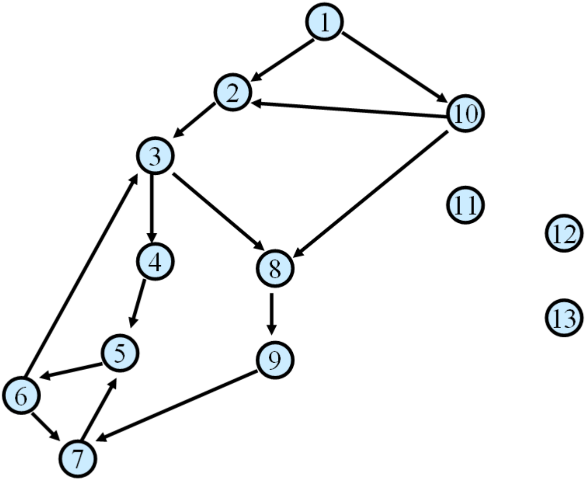
11

Directed Graph G = (V,E)



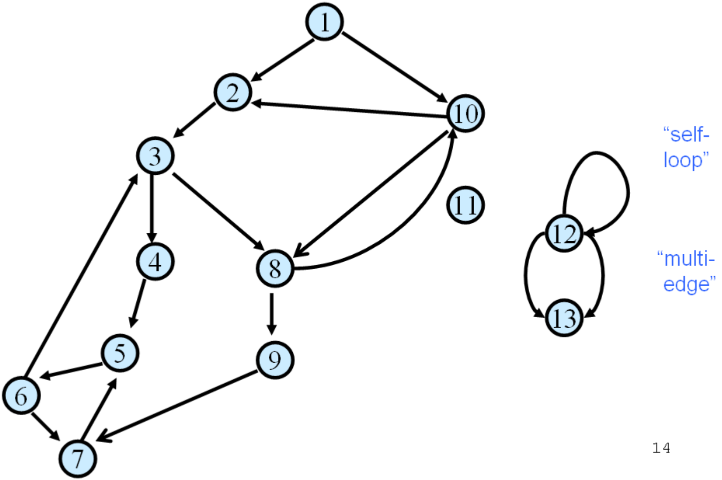
12

Directed Graph G = (V,E)



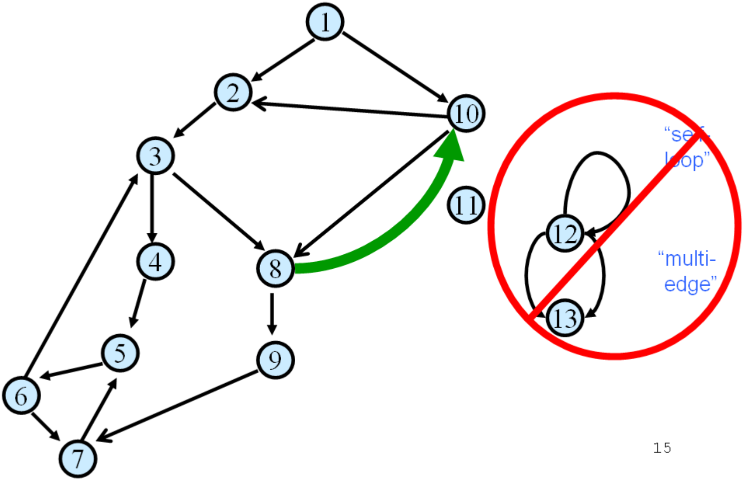
13

Directed Graph G = (V,E)



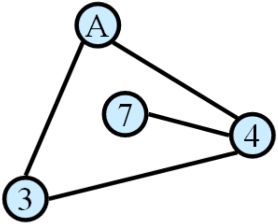
14

Directed Graph G = (V,E)



15

Specifying undirected graphs as input

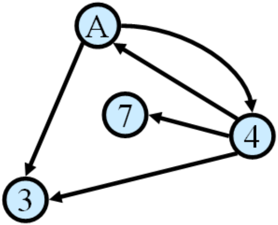


- What are the vertices?
  - Explicitly list them
  - {"A", "7", "3", "4"}
- What are the edges?
  - Either, set of edges
  - {{A,3}, {7,4}, {4,3}, {4,A}}
  - Or, (symmetric) adjacency matrix

	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



		to			
		A	7	3	4
from	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:

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

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

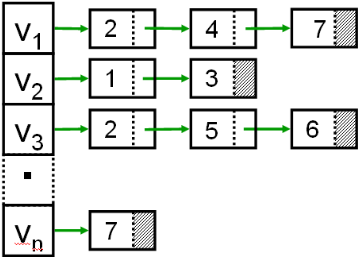
Adjacency Matrix Representation

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

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

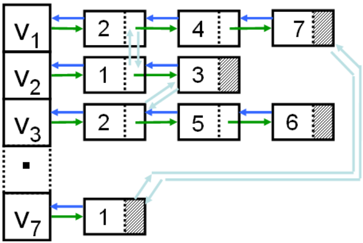
Ajacency List Representation

- Space:
  - n vertices, m edges
  - $O(n+m)$ 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)

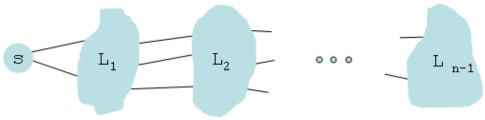


Graph Traversal

- 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

- Idea: Explore from start s, 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.
- Corollary: There is a path from s to t iff t appears in

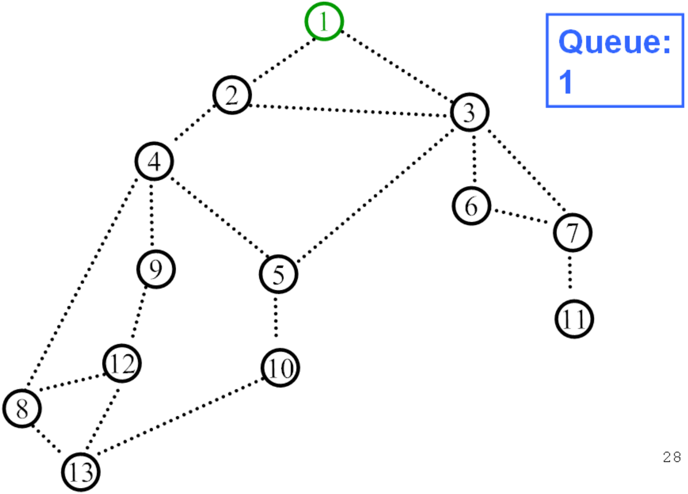
Graph Traversal: Implementation

- 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

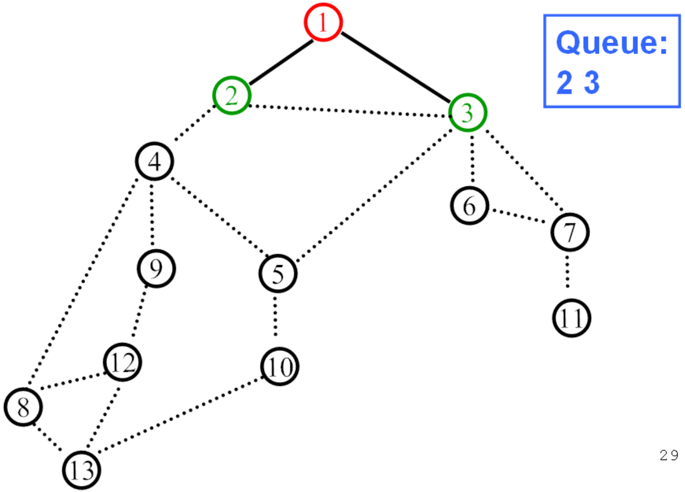
Algorithm: $BFS(s)$

```
Initialize: All vertices marked "undiscovered"
Mark s discovered
queue ← {s}
while queue not empty do
    u ← removeFront(queue)
    for all edge (u, x) do
        if x is "undiscovered" then
            Mark x "discovered"
            Append x on queue
        end if
        Mark u "fully explored"
    end for
end while
```

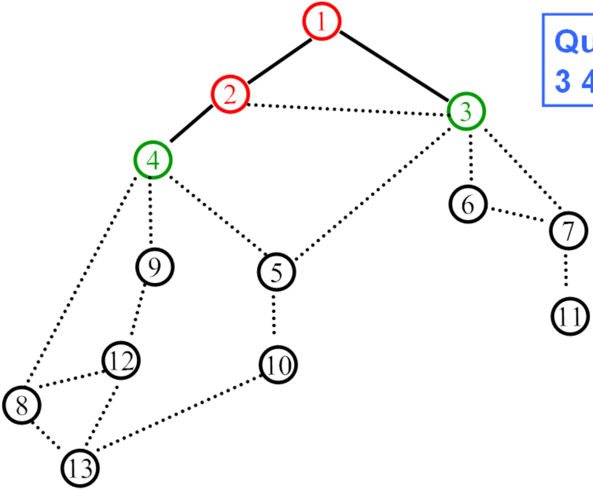
BFS in action



BFS in action



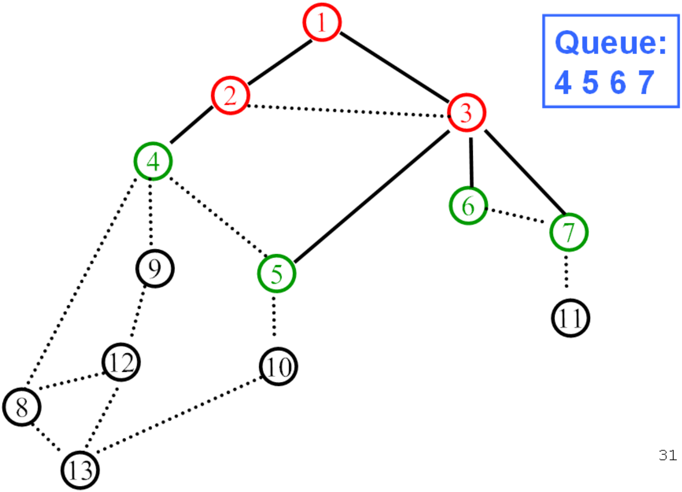
BFS in action



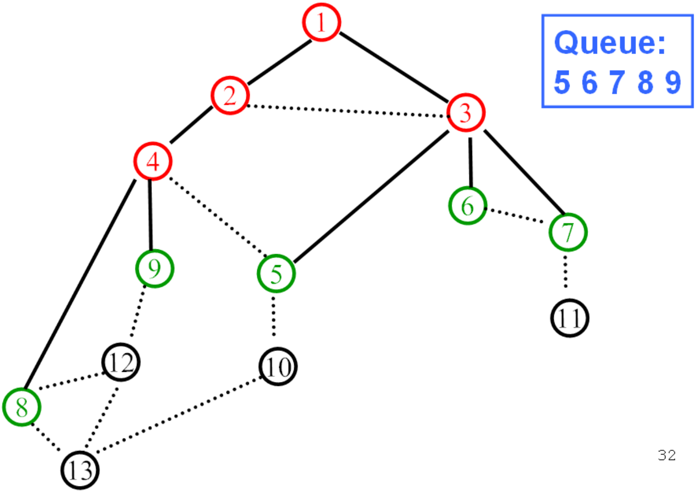
Queue:
3 4

30

BFS in action

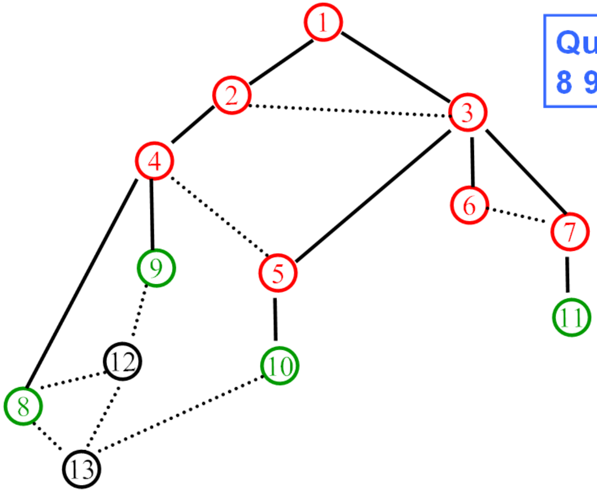


BFS in action



32

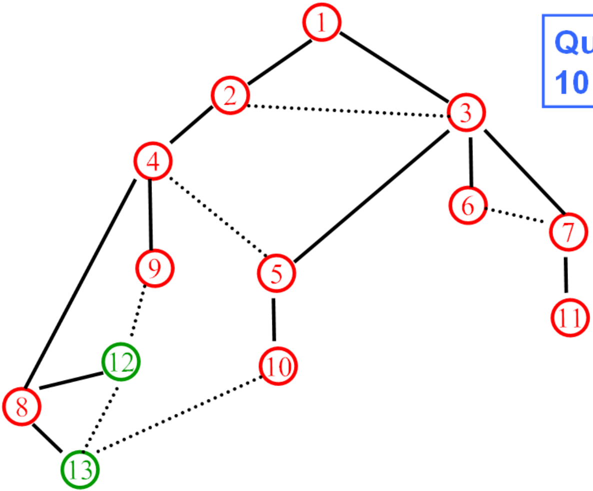
BFS in action



Queue:
8 9 10 11

33

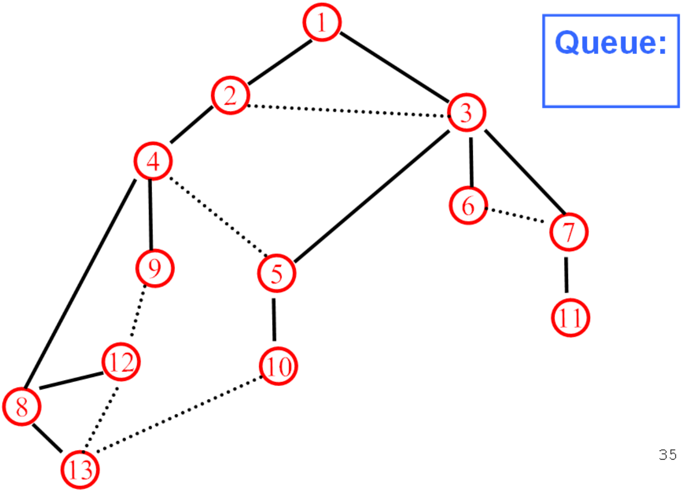
BFS in action



Queue:
10 11 12 13

34

BFS in action



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

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

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 $A[u] = A[v]$?

Algorithm: Find Connected Components

```
Iniitalize all nodes "undiscovered"
for v = 1 to n do
    if v ≠ "fully-explored" then
        BFS(v), setting A[u] ← v for each u found
            ▷ (This will mark u "discovered" / "fully-explored")
    end if
end for
```

- Total cost: O(n+m)
  - each edge is touched a constant number of times (twice)
  - works also with DFS