Graphs – Basic Review and BFS

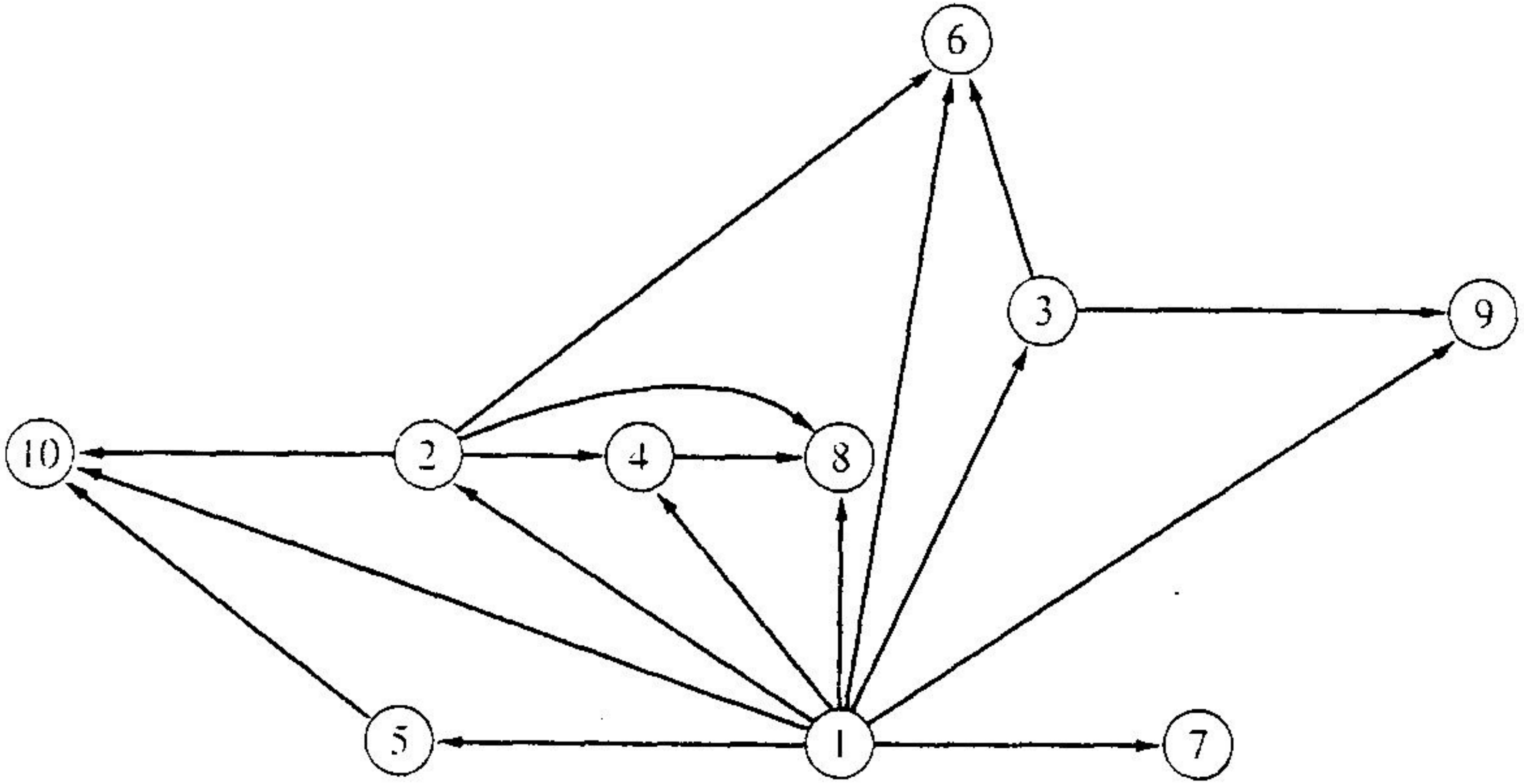
Tom Horton, Mark Floryan

CLRS Chapter 22.1 and 22.2

Graphs Review

Problems: e.g. Binary relation

- x is a proper factor of y



Definition: Directed graph

- Directed Graph
  - A directed graph, or digraph, is a pair
  - G = (V, E)
  - where V is a set whose elements are called vertices, and
  - E is a set of ordered pairs of elements of V.

    - Vertices are often also called nodes.
    - Elements of E are called edges, or directed edges, or arcs.
    - For directed edge (v, w) in E, v is its tail and w its head;
    - (v, w) is represented in the diagrams as the arrow, v -> w.
    - In text we simple write vw.

Definition: Undirected graph

- Undirected Graph
  - A undirected graph is a pair
  - G = (V, E)
  - where V is a set whose elements are called vertices, and
  - E is a set of *unordered* pairs of *distinct* elements of V.
    - Vertices are often also called nodes.
    - Elements of E are called edges, or undirected edges.
    - Each edge may be considered as a subset of V containing two elements,
    - {v, w} denotes an undirected edge
    - In diagrams this edge is the line v---w.
    - In text we simple write vw, or wv
    - vw is said to be *incident* upon the vertices v and w

Terms You Should Know

- Vertex (plural *vertices*) or Node
- Edge (sometimes referred to as an *arc*)
  - Note the meaning of *incident*
- Degree of a vertex: how many adjacent vertices
  - Digraph: in-degree (num. of incoming edges) vs. out-degree
- Graphs can be:
  - Directed or undirected
  - Weighted or not weighted
    - weights can be reals, integers, etc.
    - weight also known as: cost, length, distance, capacity,…
- Undirected graphs:
  - Normally an edge can't connect a vertex to itself
- A directed graph (also known as a *digraph*)
  - "Originating" node is the *head*, the target the *tail*
  - An edge may connect a vertex to itself

Terms You Should Know or Learn Now

- Size of graph? Two measures:
  - Number of nodes. Usually ‘V’
  - Number of edges: usually ‘E’
- Dense graph: many edges
  - Maximally dense?
  - Undirected: each node connects to all others, so
    $e = v(v-1)/2$
    Called a *complete graph*
  - Directed: $e = v(v-1)$ *why?*
- Sparse graph: fewer edges
  - Could be zero edges…

Terms You Should Know or Learn Now

- Path vs. simple path
  - One vertex is *reachable* from another vertex
- A *connected graph*
  - undirected graph, where each vertex is reachable from all others
- A *strongly connected digraph:*
  - direction affects this!
  - node u may be reachable from v, but not v from u
  - Strongly connected means both directions
- Connected components for undirected graphs

Terms You Should Know or Learn Now

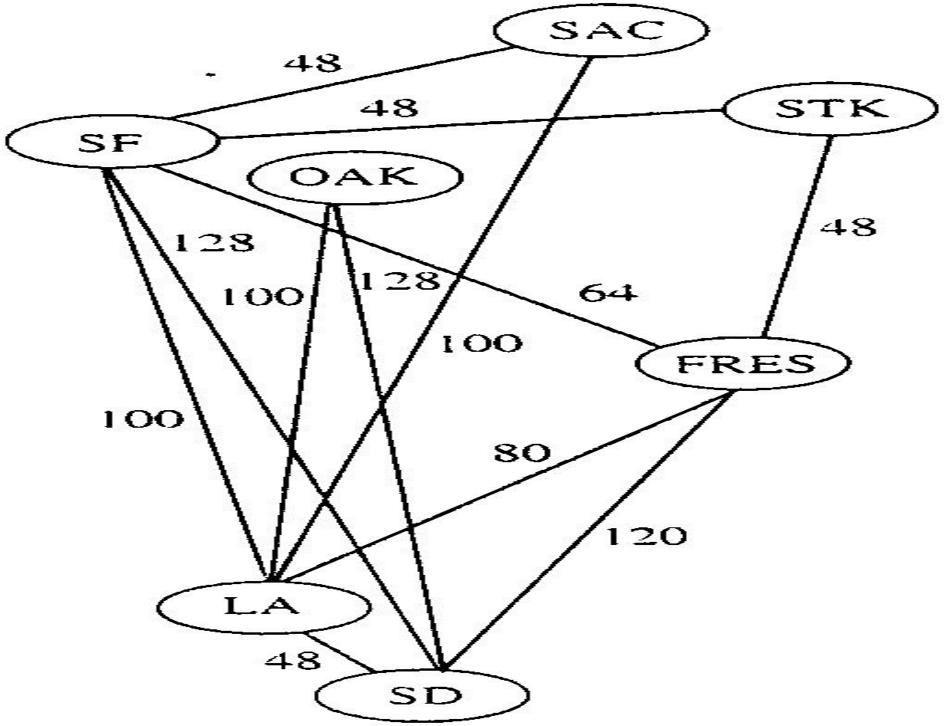
- Cycle
  - Directed graph: non-empty path with same starting and ending node
  - An edge may appear more than once (but why?)
    - **Simple cycle**: no node repeated except start and end
  - Undirected graph: same idea
    - If an edge appears more than once (I.e. non-simple) then we traverse it in the same direction
- Acyclic: no-cycles
- A connected, acyclic undirected graph: *free tree*
  - If we specificy a root, it's a *rooted tree*
  - Acyclic but not connected? a undirected *forest*
- Directed acyclic graph: a DAG

Self-test: Understand these Terms?

- Subgraph
- Symmetric digraph
- complete graph
- Adjacency relation
- Path, simple path, reachable
- Connected, Strongly Connected
- Cycle, simple cycle
- acyclic
- undirected forest
- free tree, undirected tree
- rooted tree
- Connected component

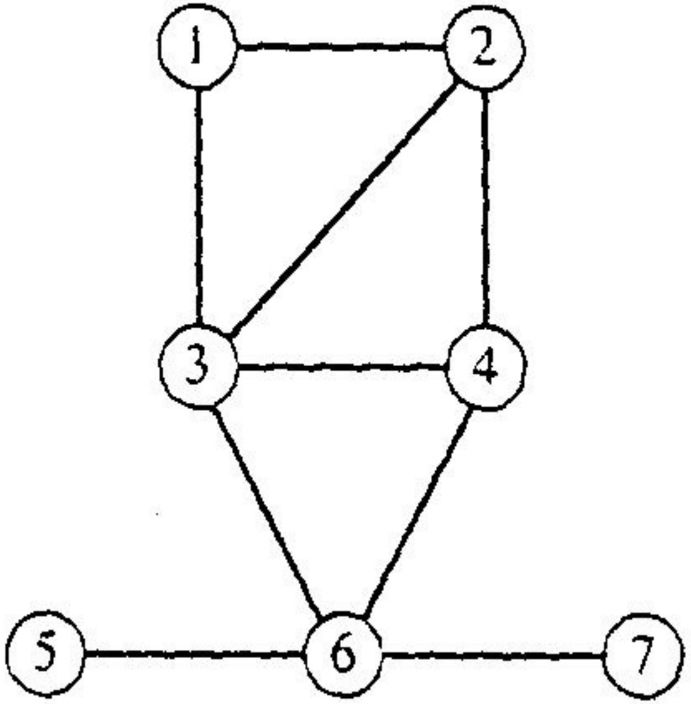
Definitions: Weighted Graph

- A weighted graph is a triple (V, E, W)
  - where (V, E) is a graph (directed or undirected) and
  - W is a function from E into R, the reals (integer or rationals).
  - For an edge e, W(e) is called the weight of e.



Graph Representations using Data Structures

- Adjacency Matrix Representation
  - Let G = (V,E), n = |V|, m = |E|, V = {v1, v2, …, vn)
  - G can be represented by an n × n matrix

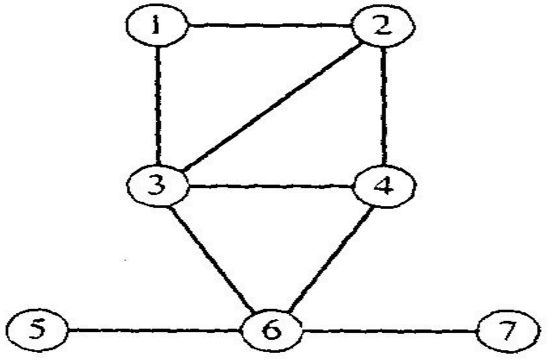


(a) An undirected graph

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

(b) Its adjacency matrix

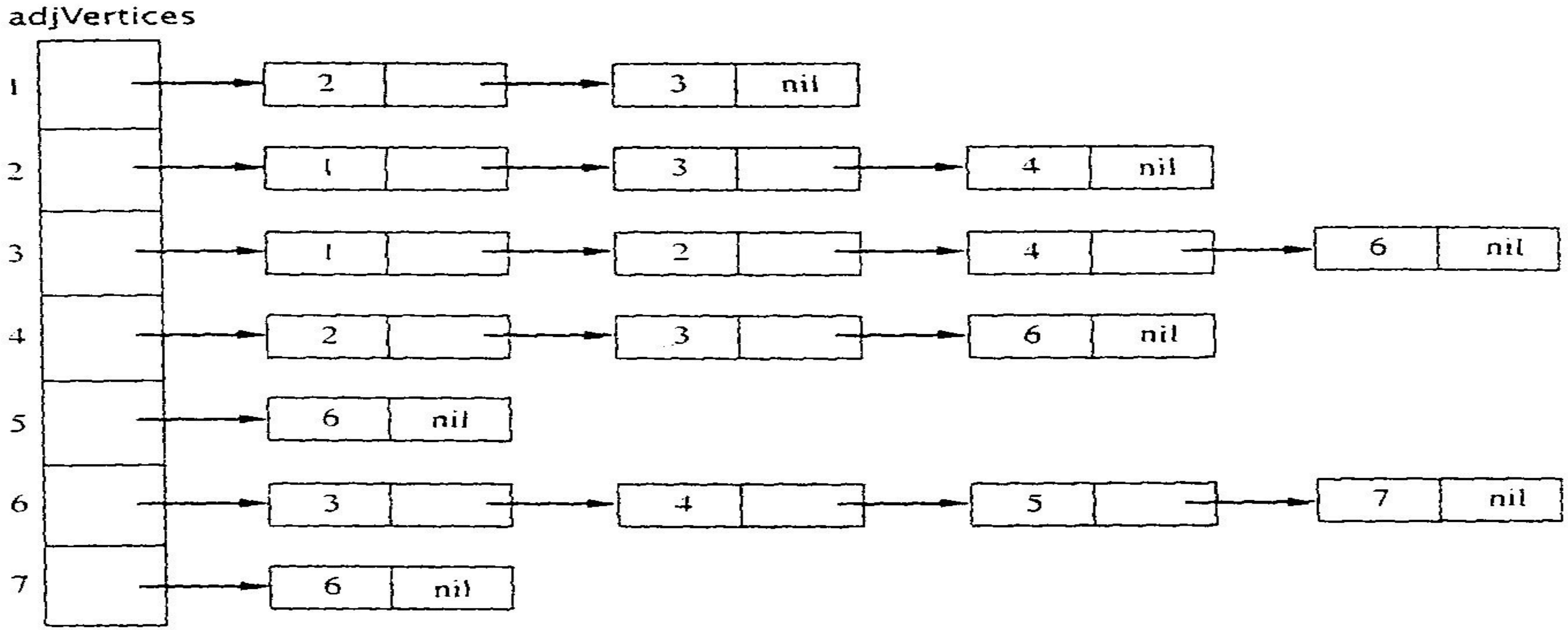
Array of Adjacency Lists Representation



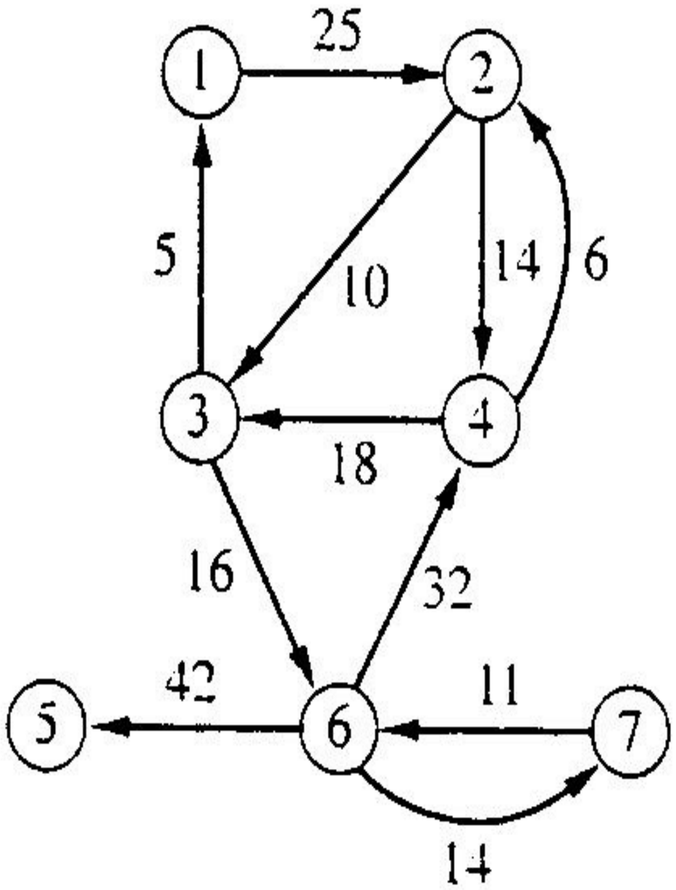
(a) An undirected graph

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

(b) Its adjacency matrix



Adjacency Matrix for weight digraph

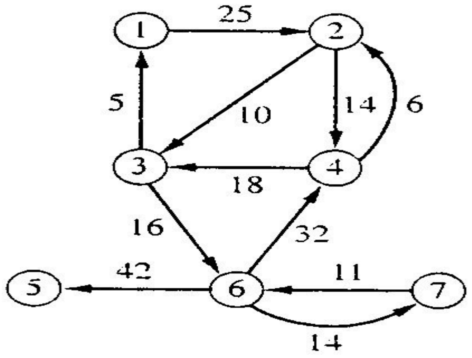


(a) A weighted digraph

$$
\begin{pmatrix}
0 & 25.0 & \infty & \infty & \infty & \infty & \infty \\
\infty & 0 & 10.0 & 14.0 & \infty & \infty & \infty \\
5.0 & \infty & 0 & \infty & \infty & 16.0 & \infty \\
\infty & 6.0 & 18.0 & 0 & \infty & \infty & \infty \\
\infty & \infty & \infty & \infty & 0 & \infty & \infty \\
\infty & \infty & \infty & 32.0 & 42.0 & 0 & 14.0 \\
\infty & \infty & \infty & \infty & \infty & 11.0 & 0
\end{pmatrix}
$$

(b) Its adjacency matrix

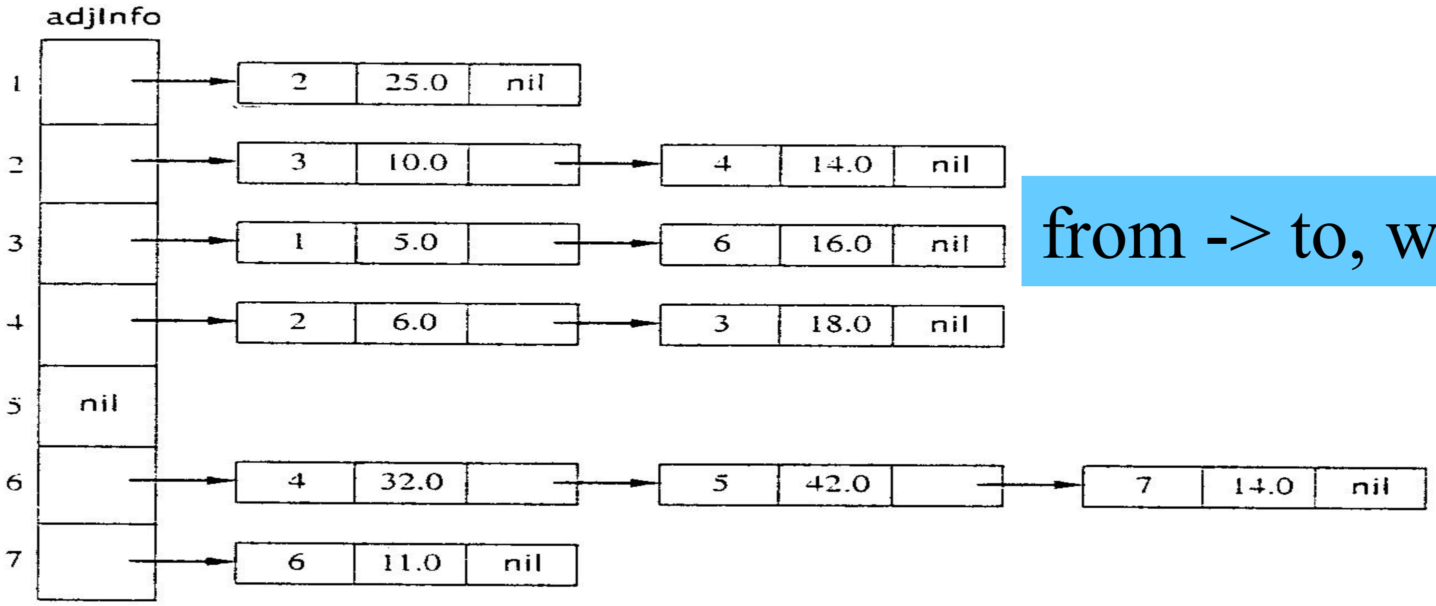
Array of Adjacency Lists Representation



(a) A weighted digraph

$$\begin{pmatrix} 0 & 25.0 & \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 10.0 & 14.0 & \infty & \infty & \infty \\ 5.0 & \infty & 0 & \infty & \infty & 16.0 & \infty \\ \infty & 6.0 & 18.0 & 0 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 & \infty & \infty \\ \infty & \infty & \infty & 32.0 & 42.0 & 0 & 14.0 \\ \infty & \infty & \infty & \infty & \infty & 11.0 & 0 \end{pmatrix}$$

(b) Its adjacency matrix



from -> to, weight

(c) Its adjacency-list structure

Breadth-First Search

Traversing Graphs

- "Traversing" means processing each vertex edge in some organized fashion by following edges between vertices
  - We speak of *visiting* a vertex. Might do something while there.
- Recall traversal of binary trees:
  - Several strategies: In-order, pre-order, post-order
  - Traversal strategy implies an <u>order</u> of visits
  - We used recursion to describe and implement these
- Graphs can be used to model interesting, complex relationships
  - Often traversal used just to process the set of vertices or edges
  - Sometimes traversal can identify interesting properties of the graph
  - Sometimes traversal (perhaps modified, enhanced) can answer interesting questions about the problem-instance that the graph models

BFS: Overall Strategy

- Breadth-first search: Strategy
  - choose a starting vertex, distance d = 0
  - vertices are visited in order of increasing distance from the starting vertex,
  - examine all edges leading from vertices (at distance d) to adjacent vertices (at distance d+1)
  - then, examine all edges leading from vertices at distance d+1 to distance d+2, and so on,
  - until no new vertex is discovered

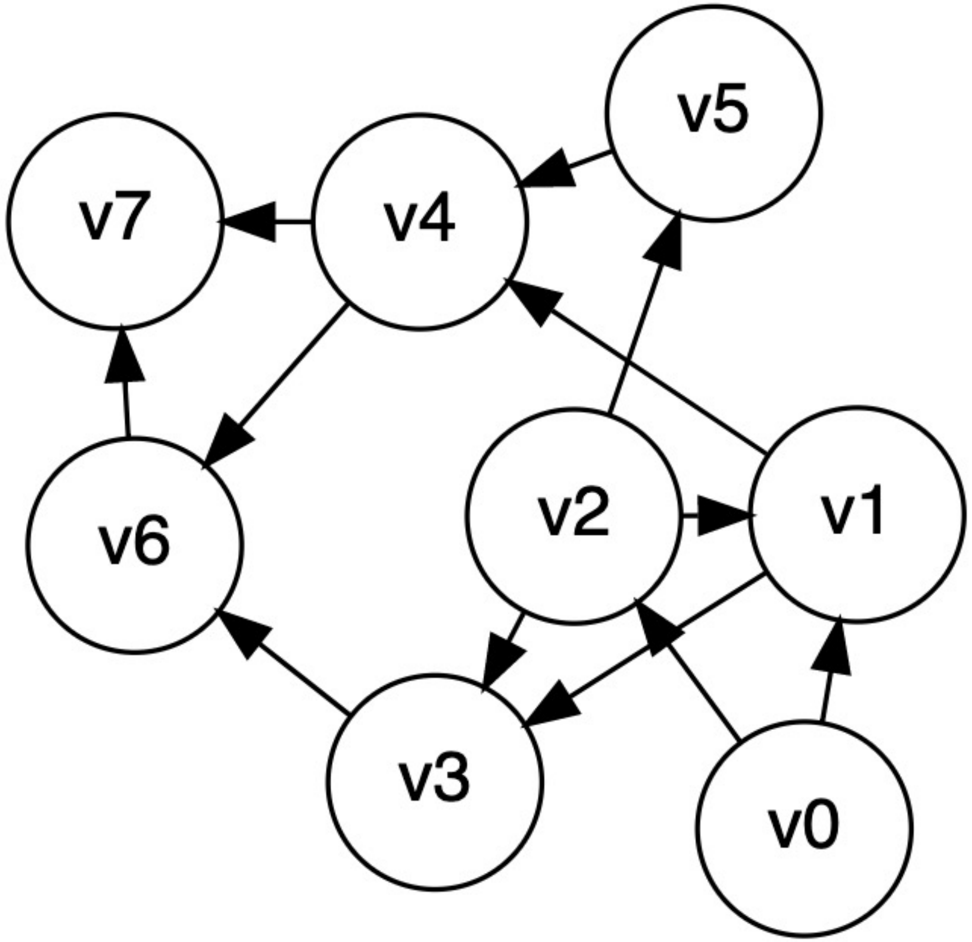
BFS: Specific Input/Output

- Input:
  - A graph ***G***
  - single start vertex ***s***

- Output:
  - Shortest distance from ***s*** to each node in ***G*** (distance = number of edges)
  - Breadth-First Tree of ***G*** with root ***s***
    - *Note: The paths in this BFS tree represent the shortest paths from s to each node in G*

Breadth-first search, quick example

- Let's start at V0



Breadth-first search implementation

```
BFS(G, s)
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            ENQUEUE(Q, v)
    u.color = BLACK
```

- Vertices here have some properties:
  - *color = white/gray/black*
  - *d = distance from start node*
  - *pi = node through which d is achieved*

Breadth-first search: Analysis

- For a digraph having V vertices and E edges
  - Each edge is processed once in the while loop for a cost of $\theta(E)$
  - Each vertex is put into the queue once and removed from the queue and processed once, for a cost $\theta(V)$
  - Total: $\theta(V+E)$
  - Extra space is used for color array and queue, there are $\theta(V)$
- From a *tree* (breadth-first spanning tree)
  - the path in the tree from start vertex to any vertex contains the *minimum* possible number of edges
- Not all vertices are necessarily reachable from a selected starting vertex

Breadth-first search: Some Properties

- Does BFS always compute δ(s,v) correctly, where δ(s,v) is the shortest path (number of edges) from s to any vertex v?

- Lemma 1:

Let G=(V,E) be a directed or undirected graph, and let $s \epsilon$ V be an arbitrary vertex. Then, for any edge $(u, v) \epsilon\ E$

δ(s,v) ≤ δ(s,u) + 1

Breadth-first search: Some Properties

- Lemma 2:

Let G = (V,E) be a directed or undirected graph, and suppose BFS is run on G from a given source vertex $s \in V$, Then upon termination, for each vertex $v \in V$, the value v.d computed by BFS satisfies $v.d \geq \delta(s, v)$

^^^^This is a weak bound! Just says distance will not be better than best path.

$$\begin{aligned} v.d &= u.d + 1 \\ &\geq \delta(s, u) + 1 \\ &\geq \delta(s, v) . \end{aligned}$$

//By how code updates v.d
//By inductive hypothesis
//By Lemma 1 on previous slide

Breadth-first search: Some Properties

- Lemma 3:

Suppose during BFS execution, the Queue contains vertices $\{v_1, v_2, \ldots. v_n\}$ where v_1 is at head of queue and v_n is at tail of queue. Then:

$v_n.d \leq v_1.d + 1$ *//all nodes on Q differ by at most 1*

$v_i.d \leq v_{i+1}.d$ *//nodes on Q are non-decreasing distances*

for i = 1,2,3,….,n-1

Why?

Correctness of BFS

Proof of Correctness

- Claim:

- Let G=(V,E) be a directed or undirected graph, and suppose that BFS is run on G from a given source vertex $s \in V$. Then, during its execution, BFS discovers every vertex $v \in V$ that is reachable from s, and upon termination $v.d = \delta(s, v)$ for all $v \in V$.

Proof of Correctness

- Proof by Contradiction:

- Assume that BFS does NOT work.
- Then…there MUST exist at least one node v such that $v.d \neq \delta(s, v)$

- There might be more, but let v be such a node with the smallest v.d value
  - Meaning the "first one" that BFS incorrectly calculates.
  - This is a good choice because we can assume all nodes with smaller d value were computed correctly! Nice!

Proof of Correctness

- So, this incorrectly calculated node v has the following property:

$$v.d > \delta(s,v) = \delta(s,u) + 1 = u.d + 1$$

Because of Lemma 2!

By definition of optimal path

By how we chose v

Proof of Correctness

$$v.d > \delta(s,v) = \delta(s,u) + 1 = u.d + 1$$

So…at some point during execution. The node u is popped off the queue and the edge e=(u,v) is followed and node v is processed. Three cases:

Case 1: v is white

Case 2: v is gray

Case 3: v is black

Proof of Correctness

$$v.d > \delta(s, v) = \delta(s, u) + 1 = u.d + 1$$

Case 1: v is white

If v is white, algorithm sets $v.d = u.d + 1$ (line 15).

Contradiction! above formula shows $v.d > u.d + 1$

Proof of Correctness

$$v.d > \delta(s, v) = \delta(s, u) + 1 = u.d + 1$$

Case 2: v is gray

if v is gray, then v is currently on the queue.
v was turned gray by dequeuing some other node w, setting $v.d = w.d + 1$
Order on queue: w, then u, then v, Lemma 3 gives $w.d \leq u.d \leq v.d$
So: $v.d = w.d + 1 \leq u.d + 1$

^^Contradiction!

Proof of Correctness

$$v.d > \delta(s, v) = \delta(s, u) + 1 = u.d + 1$$

Case 3: v is black

if v is black, then v was previously on queue ahead of u
queue distance values monotonically increasing, so $v.d \leq u.d$ (Lemma 3)
Thus $v.d \leq u.d < u.d + 1$
^^Contradiction!!

Proof of Correctness

Finishing out the proof!

If BFS is wrong then either:

$$v.d < \delta(s, v)$$

No! By Lemma 2

$$v.d > \delta(s, v)$$

No! By proof by contradiction / 3 cases

Thus, $v.d = \delta(s, v)$