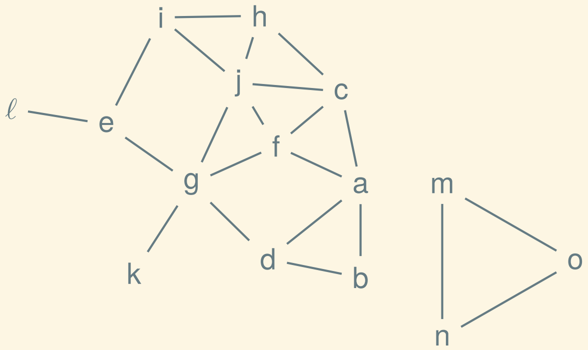
Graphs and their representations

EECS 214, Fall 2017

Kinds of graphs

A graph (undirected)

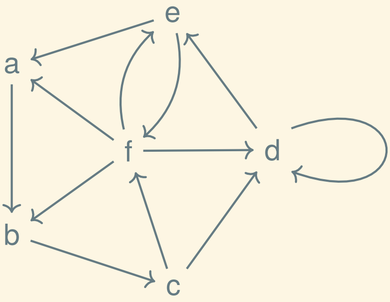


$G = (V, E)$

$V = \{a, b, c, d, e, f, g, h, i, j, k, \ell\}$

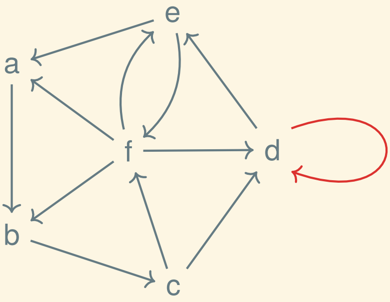
$E = \{\{a, b\}, \{a, c\}, \{a, d\}, \{a, f\}, \{b, d\}, \{c, f\},$
$\{c, h\}, \{c, j\}, \{d, g\}, \{e, g\}, \{e, i\}, \{e, m\},$
$\{f, g\}, \{f, j\}, \{g, j\}, \{g, k\}, \{h, i\}, \{h, j\}, \{i, j\}\}$

A directed graph



$$G = (V, E)$$
$$V = \{a, b, c, d, e, f\}$$
$$E = \{(a, b), (b, c), (c, d), (c, f), (d, d), (d, e), (e, f), (f, e)\}$$

A directed graph

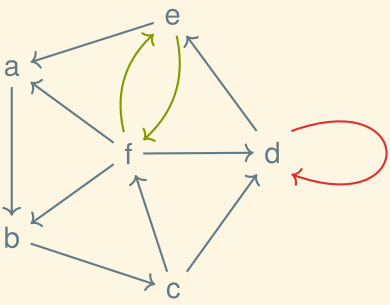


$$G = (V, E)$$

$$V = \{a, b, c, d, e, f\}$$

$$E = \{(a, b), (b, c), (c, d), (c, f), (d, d), (d, e), (e, f), (f, e)\}$$

A directed graph with cycles

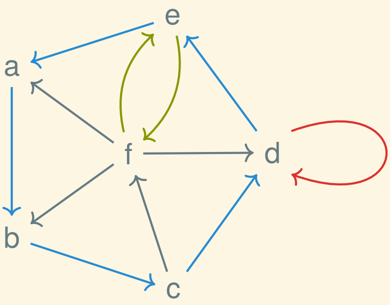


$$G = (V, E)$$

$$V = \{a, b, c, d, e, f\}$$

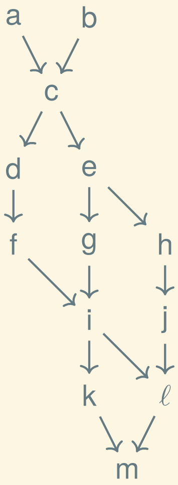
$$E = \{(a, b), (b, c), (c, d), (c, f), (d, d), (d, e), (e, f), (f, e)\}$$

A directed graph with cycles

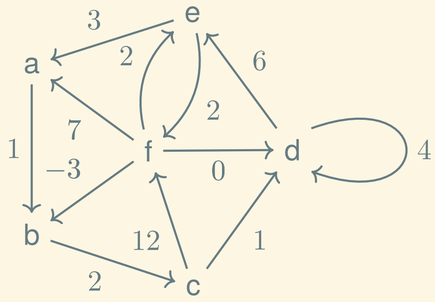


$$G = (V, E)$$
$$V = \{a, b, c, d, e, f\}$$
$$E = \{(a, b), (b, c), (c, d), (c, f), (d, d), (d, e), (e, f), (f, e)\}$$

A DAG (directed acyclic graph)



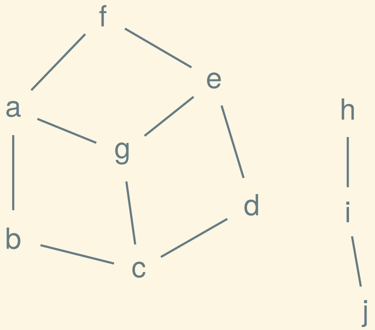
A weighted, directed graph



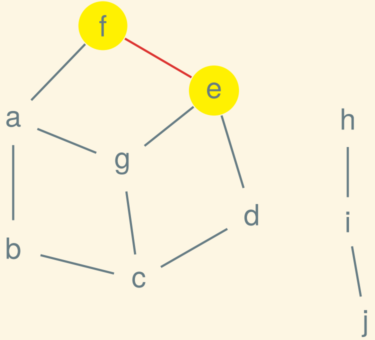
$$G = (V, E, w)$$
$$V = \{a, b, c, d, e, f\}$$
$$E = \{(a, b), (b, c), (c, d), (c, f), (d, d), (d, e), (e, f), (f, e)\}$$
$$w = \{(a, b) \mapsto 1, (b, c) \mapsto 2, (c, d) \mapsto 1, (c, f) \mapsto 12, \dots\}$$

A little graph theory

Some graph definitions

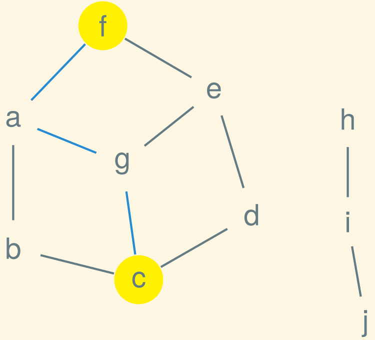


Some graph definitions



If $\{v, u\} \in E$ then v and u are *adjacent*

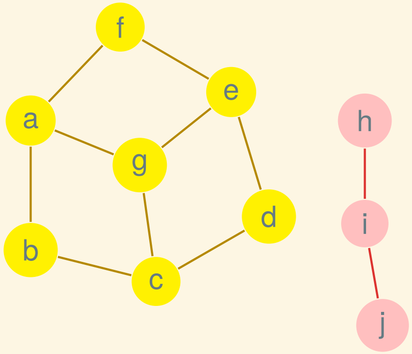
Some graph definitions



If $\{v, u\} \in E$ then v and u are *adjacent*

If $\{v_0, v_1\}, \{v_1, v_2\}, \ldots, \{v_{k-1}, v_k\} \in E$ then there is a *path* from v_0 to v_k, and we say v_0 and v_k are *connected*

Components



A subgraph of nodes all connected to each other is a *connected component*; here we have two

Degree

The degree of a vertex is the number of adjacent vertices:

$$\mathrm{degree}(v, G) = \left|\{u \in V : \{u, v\} \in E\}\right| \text{ where } G = (V, E)$$

Degree

The degree of a vertex is the number of adjacent vertices:

$$\mathrm{degree}(v, G) = \Big|\{u \in V : \{u, v\} \in E\}\Big| \text{ where } G = (V, E)$$

The degree of a graph is the maximum degree of any vertex:

$$\mathrm{degree}(G) = \max_{v \in V} \mathrm{degree}(v, G) \text{ where } G = (V, E)$$

Degree

The degree of a vertex is the number of adjacent vertices:

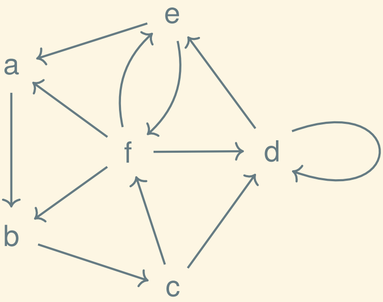
$$\mathrm{degree}(v, G) = \Big|\{u \in V : \{u, v\} \in E\}\Big| \text{ where } G = (V, E)$$

The degree of a graph is the maximum degree of any vertex:

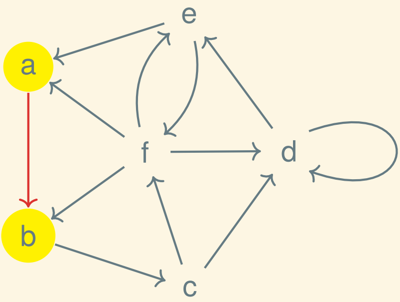
$$\mathrm{degree}(G) = \max_{v \in V} \mathrm{degree}(v, G) \text{ where } G = (V, E)$$

Sometimes we will refer to the degree as d, such as when we say that a particular operation is $\mathcal{O}(d)$.

Some digraph definitions

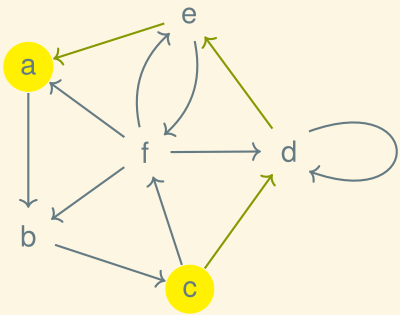


Some digraph definitions



If $(v, u) \in E$, v is the *direct predecessor* of u and u is the *direct successor* of v

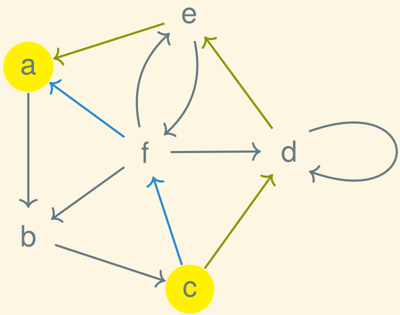
Some digraph definitions



If $(v, u) \in E$, v is the *direct predecessor* of u and u is the *direct successor* of v

If $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k) \in E$ then there is a *path* from v_0 to v_k; we say that v_k is *reachable* from v_0

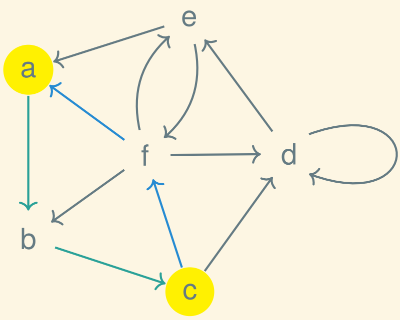
Some digraph definitions



If $(v, u) \in E$, v is the *direct predecessor* of u and u is the *direct successor* of v

If $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k) \in E$ then there is a *path* from v_0 to v_k; we say that v_k is *reachable* from v_0

Some digraph definitions

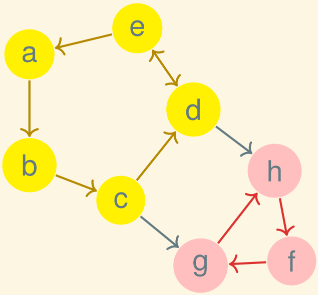


If $(v, u) \in E$, v is the *direct predecessor* of u and u is the *direct successor* of v

If $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k) \in E$ then there is a *path* from v_0 to v_k; we say that v_k is *reachable* from v_0

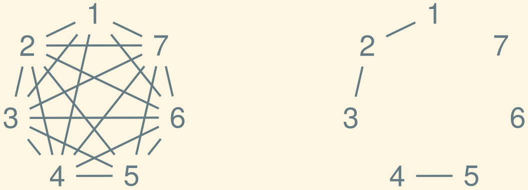
If v_k and v_0 are mutually reachable from each other, they are *strongly connected*

Strongly connected components



In a digraph, a subgraph of vertices all strongly connected to each other is a *strongly connected component*; here we have a connected graph with two SCCs

Dense versus sparse



Programming with graphs

A graph ADT

Looks like (V, E) (as above)

Operations:

- *newVertex*(Graph): Integer
- *addEdge*(Graph, Integer, Integer): Void
- *hasEdge*(Graph, Integer, Integer): Bool
- *getVertices*(Graph): IntegerSet
- *getNeighbors*(Graph, Integer): IntegerSet

A graph ADT

Looks like (V, E) (as above)

Operations:

- *newVertex*(Graph): Integer
- *addEdge*(Graph, Integer, Integer): Void
- *hasEdge*(Graph, Integer, Integer): Bool
- *getVertices*(Graph): IntegerSet
- *getNeighbors*(Graph, Integer): IntegerSet

Invariants:

- $V = \{0, 1, \ldots, |V| - 1\}$
- $\bigcup E \subseteq V$

Graph ADT laws

1. $\{g = (V, E)\}$ $\mathit{newVertex}(g) = n$ $\{g = (V \cup \{n\}, E)\}$ where $n = \max(V) + 1$
2. $\{g = (V, E) \wedge n, m \in V\}$ $\mathit{addEdge}(g, n, m)$ $\{g = (V, E \cup \{\{n, m\}\})\}$
3. $\{g = (V, E) \wedge \{n, m\} \in E\}$ $\mathit{hasEdge}(g, n, m) = \top$
4. $\{g = (V, E) \wedge \{n, m\} \notin E\}$ $\mathit{hasEdge}(g, n, m) = \bot$
5. $\{g = (V, E)\}$ $\mathit{getVertices}(g) = V$
6. $\{g = (V, E)\}$ $\mathit{getNeighbors}(g, n) = \{m \in V : \{m, n\} \in E\}$

A digraph ADT

Looks like (V, E) (as above, E contains ordered pairs of vertices)

Operations:

- *newVertex*(Graph): Integer
- *addEdge*(Graph, Integer, Integer): Void
- *hasEdge*(Graph, Integer, Integer): Bool
- *getVertices*(Graph): IntegerSet
- *getSuccessors*(Graph, Integer): IntegerSet
- *getPredecessors*(Graph, Integer): IntegerSet

Invariants:

- $V = \{0, 1, \ldots, |V| - 1\}$
- $\forall (v, u) \in E.\ v \in V \land u \in V$

Digraph ADT laws

1. $\{g = (V, E)\}$ *newVertex*$(g) = n$ $\{g = (V \cup \{n\}, E)\}$ where $n = \max(V) + 1$
2. $\{g = (V, E) \wedge n, m \in V\}$ *addEdge*(g, n, m) $\{g = (V, E \cup \{(n, m)\})\}$
3. $\{g = (V, E) \wedge (n, m) \in E\}$ *hasEdge*$(g, n, m) = \top$
4. $\{g = (V, E) \wedge (n, m) \notin E\}$ *hasEdge*$(g, n, m) = \bot$
5. $\{g = (V, E)\}$ *getVertices*$(g) = V$
6. $\{g = (V, E)\}$ *getSuccessors*$(g, n) = \{m \in V : (n, m) \in E\}$
7. $\{g = (V, E)\}$ *getPredecessors*$(g, n) = \{m \in V : (m, n) \in E\}$

A weighted digraph ADT

Looks like (V, E, w) (as above)

Operations:

- *newVertex*(Graph): Integer
- *setEdge*(Graph, Integer, Weight_∞, Integer): Void
- *getEdge*(Graph, Integer, Integer): Weight_∞
- *getVertices*(Graph): IntegerSet
- *getSuccessors*(Graph, Integer): IntegerSet
- *getPredecessors*(Graph, Integer): IntegerSet

where Weight_∞ is either a numeric weight or infinity

A weighted digraph ADT

Looks like (V, E, w) (as above)

Operations:

- *newVertex*(Graph): Integer
- *setEdge*(Graph, Integer, Weight_∞, Integer): Void
- *getEdge*(Graph, Integer, Integer): Weight_∞
- *getVertices*(Graph): IntegerSet
- *getSuccessors*(Graph, Integer): IntegerSet
- *getPredecessors*(Graph, Integer): IntegerSet

where Weight_∞ is either a numeric weight or infinity

Additional invariant:

- $\forall v, u \in V$:
    - If $(v, u) \in E$ then $w(v, u) < \infty$
    - If $(v, u) \notin E$ then $w(v, u) = \infty$

Weighted digraph ADT laws

1. $\{g = (V, E, w)\}$ $newVertex(g) = n$ $\{g = (V \cup \{n\}, E, w)\}$ where $n = \max(V) + 1$
2. $\{g = (V, E, w) \wedge n, m \in V\}$ $setEdge(g, n, a, m)$ $\{g = (V, E \cup \{(n, m)\}, w\{(n, m) \mapsto a\})\}$
3. $\{g = (V, E, w) \wedge (n, m) \in E\}$ $getEdge(g, n, m) = w(n, m)$
4. $\{g = (V, E, w) \wedge (n, m) \notin E\}$ $getEdge(g, n, m) = \infty$
5. $\{g = (V, E, w)\}$ $getVertices(g) = V$
6. $\{g = (V, E, w)\}$ $getSuccessors(g, n) = \{m \in V : (n, m) \in E\}$
7. $\{g = (V, E, w)\}$ $getPredecessors(g, n) = \{m \in V : (m, n) \in E\}$

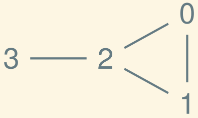
Graph representation

Two graph representations

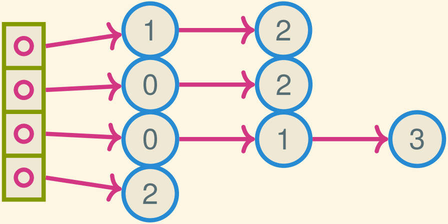
There are two common ways that graphs are represented on a computer:

- adjacency list
- adjacency matrix

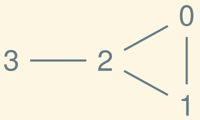
Adjacency list



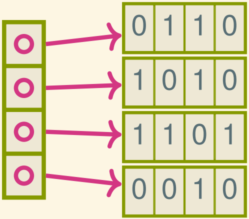
In an array, store a list of neighbors (or successors) for each vertex:



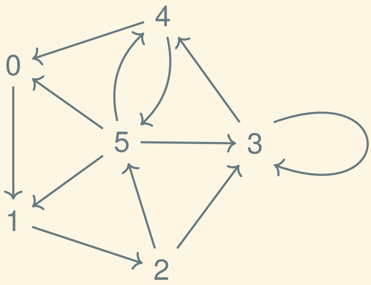
Adjacency matrix



Store a $|V|$-by-$|V|$ matrix of Booleans indicating where edges are present:

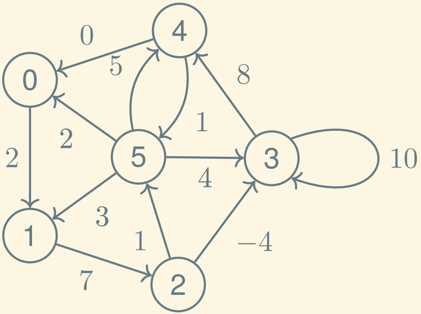


A directed adjacency matrix example



	0	1	2	3	4	5
0	0	1	0	0	0	0
1	0	0	1	0	0	0
2	0	0	0	1	0	1
3	0	0	0	1	1	0
4	1	0	0	0	0	1
5	1	1	0	1	1	0

With weights



	0	1	2	3	4	5
0	∞	2	∞	∞	∞	∞
1	∞	∞	7	∞	∞	∞
2	∞	∞	∞	-4	∞	1
3	∞	∞	∞	10	8	∞
4	1	∞	∞	∞	∞	0
5	2	3	∞	4	5	∞

Space comparison

Adjacency list—has a list for each vertex, and the total length of all the lists is the number of edges: $\mathcal{O}(V + E)$

Adjacency matrix—is $|V|$ by $|V|$ regardless of the number of edges: $\mathcal{O}(V^2)$

Space comparison

Adjacency list—has a list for each vertex, and the total length of all the lists is the number of edges: $\mathcal{O}(V + E)$

Adjacency matrix—is $|V|$ by $|V|$ regardless of the number of edges: $\mathcal{O}(V^2)$

When might we want to use one or the other?

Time comparison

	adj. list	adj. matrix
addEdge/setEdge		

Time comparison

	adj. list	adj. matrix
addEdge/setEdge	$\mathcal{O}(\textit{setInsert}(d))$	$\mathcal{O}(1)$

Time comparison

	adj. list	adj. matrix
addEdge/*setEdge*	$\mathcal{O}(\textit{setInsert}(d))$	$\mathcal{O}(1)$
getEdge/*hasEdge*		

Time comparison

	adj. list	adj. matrix
addEdge/*setEdge*	$\mathcal{O}(\textit{setInsert}(d))$	$\mathcal{O}(1)$
getEdge/*hasEdge*	$\mathcal{O}(\textit{setLookup}(d))$	$\mathcal{O}(1)$

Time comparison

	adj. list	adj. matrix
addEdge/setEdge	$\mathcal{O}(\textit{setInsert}(d))$	$\mathcal{O}(1)$
getEdge/hasEdge	$\mathcal{O}(\textit{setLookup}(d))$	$\mathcal{O}(1)$
getSuccessors		

Time comparison

	adj. list	adj. matrix
addEdge/setEdge	$\mathcal{O}(\textit{setInsert}(d))$	$\mathcal{O}(1)$
getEdge/hasEdge	$\mathcal{O}(\textit{setLookup}(d))$	$\mathcal{O}(1)$
getSuccessors	$\mathcal{O}(\lvert \textit{Result} \rvert)$	$\mathcal{O}(V)$

Time comparison

	adj. list	adj. matrix
addEdge/setEdge	$\mathcal{O}(\textit{setInsert}(d))$	$\mathcal{O}(1)$
getEdge/hasEdge	$\mathcal{O}(\textit{setLookup}(d))$	$\mathcal{O}(1)$
getSuccessors	$\mathcal{O}(\lvert \textit{Result} \rvert)$	$\mathcal{O}(V)$
getPredecessors		

Time comparison

	adj. list	adj. matrix
addEdge/*setEdge*	$\mathcal{O}(\mathit{setInsert}(d))$	$\mathcal{O}(1)$
getEdge/*hasEdge*	$\mathcal{O}(\mathit{setLookup}(d))$	$\mathcal{O}(1)$
getSuccessors	$\mathcal{O}(\lvert\mathit{Result}\rvert)$	$\mathcal{O}(V)$
getPredecessors	$\mathcal{O}(V+E)$	$\mathcal{O}(V)$

Next time: graph search