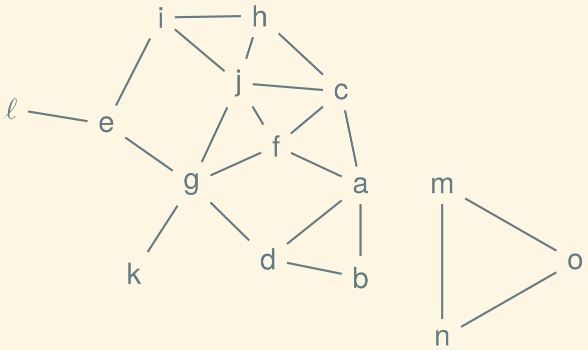
Graphs and their representations

CS 214, Fall 2019

Kinds of graphs

Kinds of graphs, or, What is a graph?

A graph (undirected)

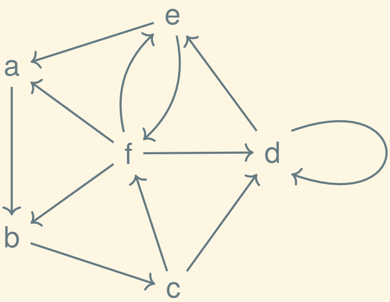


$$G = (V, E)$$

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

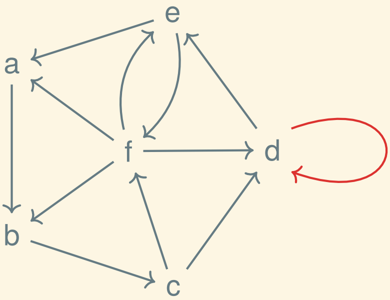
$$E = \{\{a, b\}, \{a, c\}, \{a, d\}, \{a, f\}, \{b, d\}, \{c, f\}, \{c, h\}, \{c, j\}, \{d, g\}, \{e, g\}, \{e, i\}, \{e, m\}, \{f, g\}, \{f, j\}, \{g, j\}, \{g, k\}, \{h, i\}, \{h, j\}, \{i, j\}\}$$

A directed graph



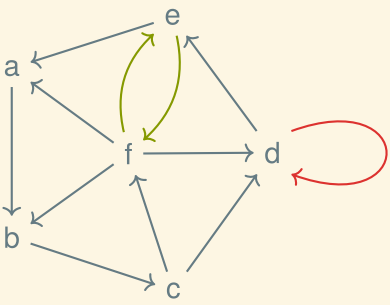
$$G = (V, E)$$
$$V = \{a, b, c, d, e, f\}$$
$$E = \{(a, b), (b, c), (c, d), (c, f), (d, d), (d, e), (e, f), (f, e)\}$$

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

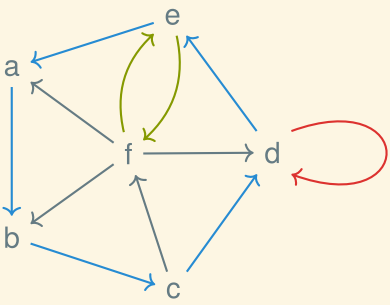


$$G = (V, E)$$

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

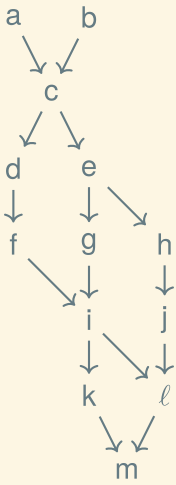
$$E = \{(a, b), (b, c), (c, d), (c, f), (d, d), (d, e), (e, f), (f, e)\}$$

A directed graph with cycles

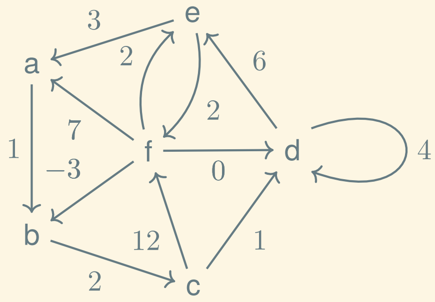


$$G = (V, E)$$
$$V = \{a, b, c, d, e, f\}$$
$$E = \{(a,b), (b,c), (c,d), (c,f), (d,d), (d,e), (e,f), (f,e)\}$$

A DAG (directed acyclic graph)



A weighted, directed graph



$$G = (V, E, w)$$

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

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

$$w = \{(a, b) \mapsto 1, (b, c) \mapsto 2, (c, d) \mapsto 1, (c, f) \mapsto 12, \ldots\}$$

Graphs: What are they good for?

Lots of things!

- spatial graphs

Lots of things!

- spatial graphs
- dependency graphs

Lots of things!

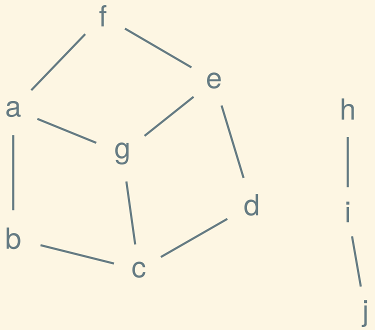
- spatial graphs
- dependency graphs
- interference graphs

Lots of things!

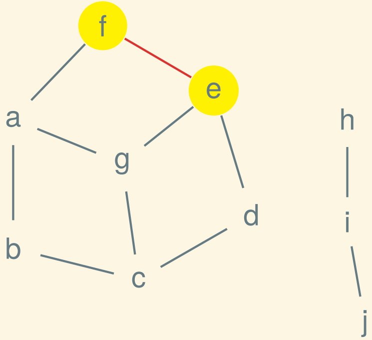
- spatial graphs
- dependency graphs
- interference graphs
- flow graphs

A little graph theory

Some graph definitions

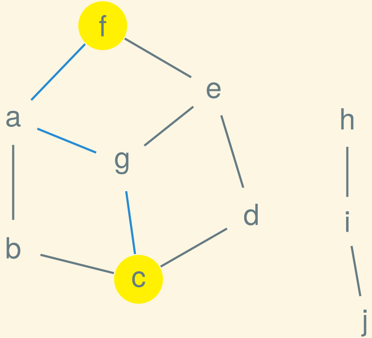


Some graph definitions



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

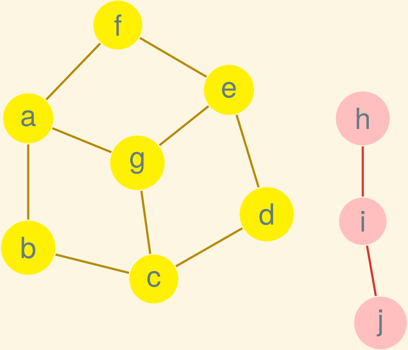
Some graph definitions



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:

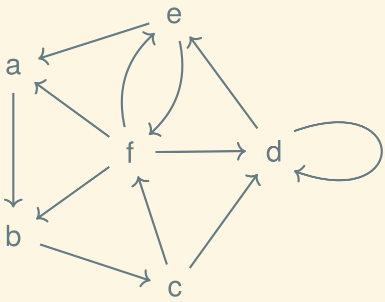
$$\text{degree}(v, G) = \Big|\{u \in V : \{u, v\} \in E\}\Big| \text{ where } G = (V, E)$$

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

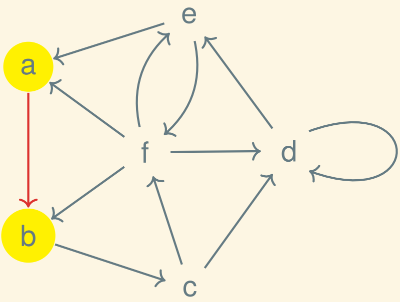
$$\text{degree}(G) = \max_{v \in V} \text{degree}(v, G) \text{ where } G = (V, E)$$

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

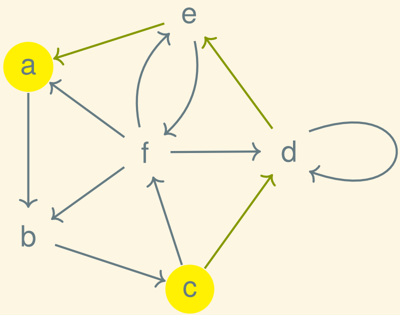


Some digraph definitions



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

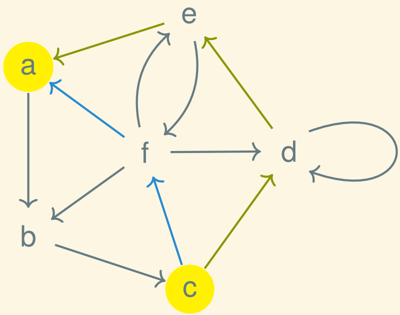
Some digraph definitions



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

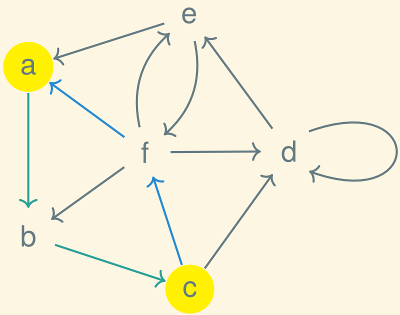
Some digraph definitions



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

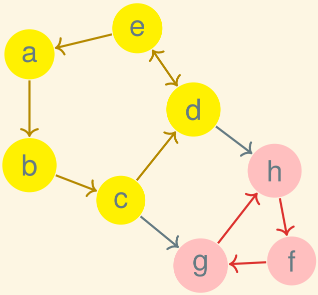


If $(v, u) \in E$, v is the *direct predecessor* of u and u is the *direct successor* of v

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

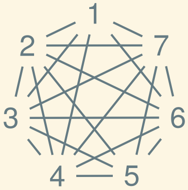
If v_k and v_0 are mutually reachable from each other, they are *strongly connected*

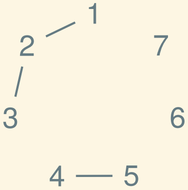
Strongly connected components



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 $\boxed{(V, E)}$ (as above)

Operations:

```
interface GRAPH:
    def new_vertex(self) -> nat?
    def add_edge(self, u: nat?, v: nat?) -> NoneC
    def has_edge?(self, u: nat?, v: nat?) -> bool?
    def get_vertices(self) -> VertexSet
    def get_neighbors(self, v: nat?) -> VertexSet
```

A graph ADT

Looks like $\boxed{(V, E)}$ (as above)

Operations:

```
interface GRAPH:
    def new_vertex(self) -> nat?
    def add_edge(self, u: nat?, v: nat?) -> NoneC
    def has_edge?(self, u: nat?, v: nat?) -> bool?
    def get_vertices(self) -> VertexSet
    def get_neighbors(self, v: nat?) -> VertexSet
```

Invariants:

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

Graph ADT laws

$$\left\{ g = \boxed{(V, E)} \right\} \; g.new\_vertex() \Rightarrow n \; \left\{ g = \boxed{(V \cup \{n\}, E)} \wedge n = \max(V) + 1 \right\}$$

Graph ADT laws

$$\left\{g = \boxed{(V, E)}\right\}\ g.new\_vertex() \Rightarrow n\ \left\{g = \boxed{(V \cup \{n\}, E)} \wedge n = \max(V) + 1\right\}$$

$$\left\{g = \boxed{(V, E)} \wedge n, m \in V\right\}\ g.add\_edge(n, m) \Rightarrow None\ \left\{g = \boxed{(V, E \cup \{\{n, m\}\})}\right\}$$

Graph ADT laws

$$\left\{g = \boxed{(V, E)}\right\}\ g.new\_vertex() \Rightarrow n\ \left\{g = \boxed{(V \cup \{n\}, E)} \wedge n = \max(V) + 1\right\}$$

$$\left\{g = \boxed{(V, E)} \wedge n, m \in V\right\}\ g.add\_edge(n, m) \Rightarrow None\ \left\{g = \boxed{(V, E \cup \{\{n, m\}\})}\right\}$$

$$\left\{g = \boxed{(V, E)} \wedge \{n, m\} \in E\right\}\ g.has\_edge?(n, m) \Rightarrow True\ \{g = g_0\}$$

$$\left\{g = \boxed{(V, E)} \wedge \{n, m\} \notin E\right\}\ g.has\_edge?(n, m) \Rightarrow False\ \{g = g_0\}$$

Graph ADT laws

$$\left\{g = \boxed{(V, E)}\right\}\ g.new\_vertex() \Rightarrow n\ \left\{g = \boxed{(V \cup \{n\}, E)} \wedge n = \max(V) + 1\right\}$$

$$\left\{g = \boxed{(V, E)} \wedge n, m \in V\right\}\ g.add\_edge(n, m) \Rightarrow None\ \left\{g = \boxed{(V, E \cup \{\{n, m\}\})}\right\}$$

$$\left\{g = \boxed{(V, E)} \wedge \{n, m\} \in E\right\}\ g.has\_edge?(n, m) \Rightarrow True\ \{g = g_0\}$$

$$\left\{g = \boxed{(V, E)} \wedge \{n, m\} \notin E\right\}\ g.has\_edge?(n, m) \Rightarrow False\ \{g = g_0\}$$

$$\left\{g = \boxed{(V, E)}\right\}\ g.get\_vertices() \Rightarrow V\ \{g = g_0\}$$

$$\left\{g = \boxed{(V, E)}\right\}\ g.get\_neighbors(n) \Rightarrow \{m \in V : \{m, n\} \in E\}\ \{g = g_0\}$$

A digraph ADT

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

Operations:

```
interface DIGRAPH:
    def new_vertex(self) -> nat?
    def add_edge(self, src: nat?, dst: nat?) -> NoneC
    def has_edge?(self, src: nat?, dst: nat?) -> bool?
    def get_vertices(self) -> VertexSet
    def get_succs(self, v: nat?) -> VertexSet
    def get_preds(self, v: nat?) -> VertexSet
```

Invariants:

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

Digraph ADT laws

$$\left\{g = \boxed{(V, E)}\right\}\ g.new\_vertex() \Rightarrow n\ \left\{g = \boxed{((V \cup \{n\}, E))} \wedge n = \max(V) + 1\right\}$$

$$\left\{g = \boxed{(V, E)} \wedge n, m \in V\right\}\ g.add\_edge(n, m) \Rightarrow None\ \left\{g = \boxed{(V, E \cup \{(n, m)\})}\right\}$$

Digraph ADT laws

$$\left\{g = \boxed{(V, E)}\right\}\ g.new\_vertex() \Rightarrow n\ \left\{g = \boxed{((V \cup \{n\}, E))} \wedge n = \max(V) + 1\right\}$$

$$\left\{g = \boxed{(V, E)} \wedge n, m \in V\right\}\ g.add\_edge(n, m) \Rightarrow None\ \left\{g = \boxed{(V, E \cup \{(n, m)\})}\right\}$$

$$\left\{g = \boxed{(V, E)} \wedge (n, m) \in E\right\}\ g.has\_edge(n, m) \Rightarrow True\ \{g = g_0\}$$

$$\left\{g = \boxed{(V, E)} \wedge (n, m) \notin E\right\}\ g.has\_edge(n, m) \Rightarrow False\ \{g = g_0\}$$

Digraph ADT laws

$$\left\{g = \boxed{(V, E)}\right\}\ g.new\_vertex() \Rightarrow n\ \left\{g = \boxed{((V \cup \{n\}, E))} \wedge n = \max(V) + 1\right\}$$

$$\left\{g = \boxed{(V, E)} \wedge n, m \in V\right\}\ g.add\_edge(n, m) \Rightarrow None\ \left\{g = \boxed{(V, E \cup \{(n, m)\})}\right\}$$

$$\left\{g = \boxed{(V, E)} \wedge (n, m) \in E\right\}\ g.has\_edge(n, m) \Rightarrow True\ \{g = g_0\}$$

$$\left\{g = \boxed{(V, E)} \wedge (n, m) \notin E\right\}\ g.has\_edge(n, m) \Rightarrow False\ \{g = g_0\}$$

$$\left\{g = \boxed{(V, E)}\right\}\ g.get\_vertices() \Rightarrow V\ \{g = g_0\}$$

Digraph ADT laws

$$\left\{ g = \boxed{(V, E)} \right\} \; g.new\_vertex() \Rightarrow n \; \left\{ g = \boxed{((V \cup \{n\}, E))} \wedge n = \max(V) + 1 \right\}$$

$$\left\{ g = \boxed{(V, E)} \wedge n, m \in V \right\} \; g.add\_edge(n, m) \Rightarrow None \; \left\{ g = \boxed{(V, E \cup \{(n, m)\})} \right\}$$

$$\left\{ g = \boxed{(V, E)} \wedge (n, m) \in E \right\} \; g.has\_edge(n, m) \Rightarrow True \; \{ g = g_0 \}$$

$$\left\{ g = \boxed{(V, E)} \wedge (n, m) \notin E \right\} \; g.has\_edge(n, m) \Rightarrow False \; \{ g = g_0 \}$$

$$\left\{ g = \boxed{(V, E)} \right\} \; g.get\_vertices() \Rightarrow V \; \{ g = g_0 \}$$

$$\left\{ g = \boxed{(V, E)} \right\} \; g.get\_succs(n) \Rightarrow \{ m \in V : (n, m) \in E \} \; \{ g = g_0 \}$$

$$\left\{ g = \boxed{(V, E)} \right\} \; g.get\_preds(n) \Rightarrow \{ m \in V : (m, n) \in E \} \; \{ g = g_0 \}$$

A weighted digraph ADT

Looks like $\boxed{(V, E, w)}$ (as above)

Operations:

```
let weight? = OrC(num?, inf)

interface WDIGRAPH:
    def new_vertex(self) -> nat?
    def set_edge(self, src: nat?, w: weight?,
                         dst: nat?) -> NoneC
    def get_edge(self, src: nat?, dst: nat?) -> weight?
    def get_vertices(self) -> VertexSet
    def get_succs(self, v: nat?) -> VertexSet
    def get_preds(self, v: nat?) -> VertexSet
```

Weighted digraph ADT laws (1/2)

$$\left\{g = \boxed{(V, E, w)}\right\}$$

$$g.new\_vertex() = n \Rightarrow None$$

$$\left\{g = \boxed{(V \cup \{n\}, E, w)} \wedge n = \max(V) + 1\right\}$$

$$\left\{g = \boxed{(V, E, w)} \wedge n, m \in V \wedge a < \infty\right\}$$

$$g.set\_edge(n, a, m) \Rightarrow None$$

$$\left\{g = \boxed{(V, E \cup \{(n, m)}\}, w\{(n, m) \mapsto a\})\right\}$$

$$\left\{g = \boxed{(V, E, w)} \wedge n, m \in V\right\}$$

$$g.set\_edge(n, \infty, m) \Rightarrow None$$

$$\left\{g = \boxed{(V, E \setminus \{(n, m)}\}, w \setminus \{(n, m)\})\right\}$$

Weighted digraph ADT laws (1/2)

$$\left\{g = \boxed{(V, E, w)} \wedge (n, m) \in E\right\}\ g.get\_edge(n, m) \Rightarrow w(n, m)\ \ \{g = g_0\}$$

$$\left\{g = \boxed{(V, E, w)} \wedge (n, m) \notin E\right\}\ g.get\_edge(n, m) \Rightarrow \infty\ \ \{g = g_0\}$$

$$\left\{g = \boxed{(V, E, w)}\right\}\ g.get\_vertices(g) \Rightarrow V\ \ \{g = g_0\}$$

$$\left\{g = \boxed{(V, E, w)}\right\}\ g.get\_succs(n) \Rightarrow \{m \in V : (n, m) \in E\}\ \ \{g = g_0\}$$

$$\left\{g = \boxed{(V, E, w)}\right\}\ g.get\_preds(n) \Rightarrow \{m \in V : (m, n) \in E\}\ \ \{g = g_0\}$$

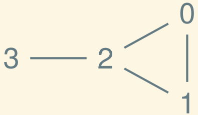
Graph representation

Two graph representations

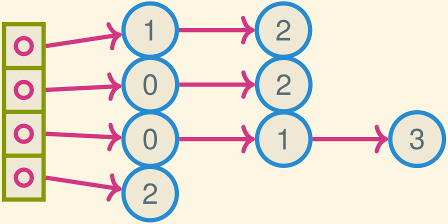
There are two common ways that graphs are represented on a computer:

- adjacency list
- adjacency matrix

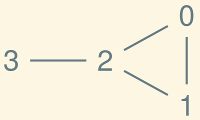
Adjacency list



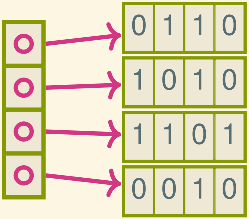
In an array, store a list of neighbors (or successors) for each vertex:



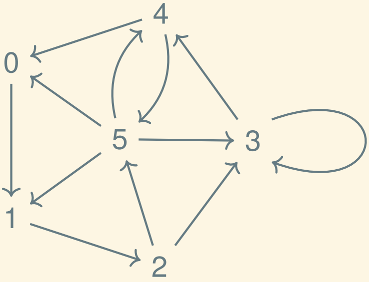
Adjacency matrix



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

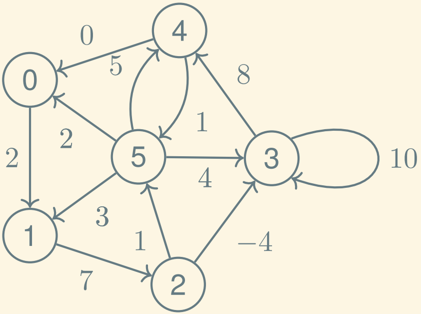


A directed adjacency matrix example



	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
add_edge/set_edge		

Time comparison

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

Time comparison

	adj. list	adj. matrix
add_edge/set_edge	$\mathcal{O}(\mathit{setInsert}(d))$	$\mathcal{O}(1)$
get_edge/has_edge?		

Time comparison

	adj. list	adj. matrix
add_edge/set_edge	$\mathcal{O}(\textit{setInsert}(d))$	$\mathcal{O}(1)$
get_edge/has_edge?	$\mathcal{O}(\textit{setLookup}(d))$	$\mathcal{O}(1)$

Time comparison

	adj. list	adj. matrix
add_edge/set_edge	$\mathcal{O}(\textit{setInsert}(d))$	$\mathcal{O}(1)$
get_edge/has_edge?	$\mathcal{O}(\textit{setLookup}(d))$	$\mathcal{O}(1)$
get_succs		

Time comparison

	adj. list	adj. matrix
add_edge/set_edge	$\mathcal{O}(\mathit{setInsert}(d))$	$\mathcal{O}(1)$
get_edge/has_edge?	$\mathcal{O}(\mathit{setLookup}(d))$	$\mathcal{O}(1)$
get_succs	$\mathcal{O}(\lvert \mathit{Result} \rvert)$	$\mathcal{O}(V)$

Time comparison

	adj. list	adj. matrix
add_edge/set_edge	$\mathcal{O}(\mathit{setInsert}(d))$	$\mathcal{O}(1)$
get_edge/has_edge?	$\mathcal{O}(\mathit{setLookup}(d))$	$\mathcal{O}(1)$
get_succs	$\mathcal{O}(\lvert \mathit{Result} \rvert)$	$\mathcal{O}(V)$
get_preds		

Time comparison

	adj. list	adj. matrix
add_edge/set_edge	$\mathcal{O}(\mathit{setInsert}(d))$	$\mathcal{O}(1)$
get_edge/has_edge?	$\mathcal{O}(\mathit{setLookup}(d))$	$\mathcal{O}(1)$
get_succs	$\mathcal{O}(\lvert \mathit{Result} \rvert)$	$\mathcal{O}(V)$
get_preds	$\mathcal{O}(V + E)$	$\mathcal{O}(V)$

Next time: graph search