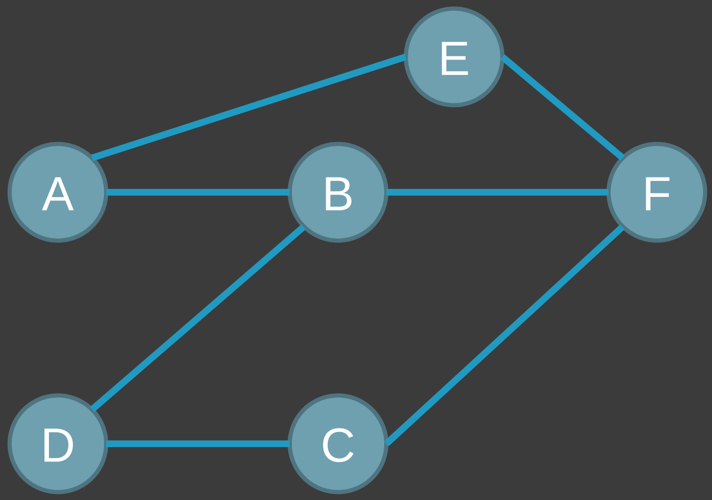
ICSI 403: DESIGN AND ANALYSIS OF ALGORITHMS

Chapter 22: Elementary Graph Algorithms – Part 1

J Marques de Carvalho

Introduction to Graphs

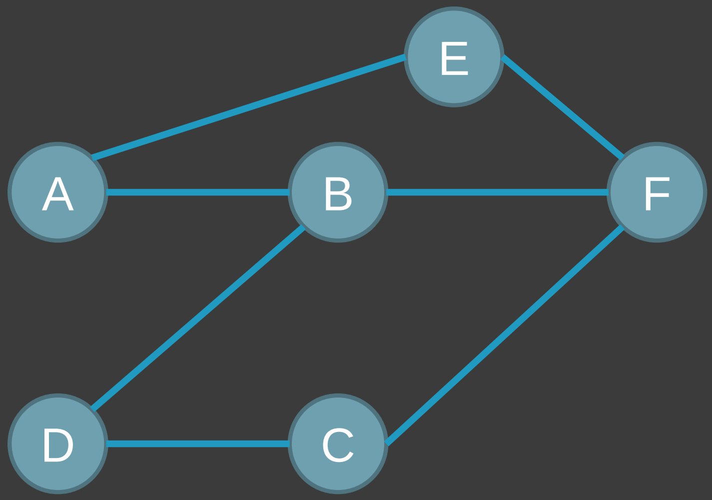
- Notation: A graph G has both _vertices_ and _edges_. The edges connect the vertices. Any vertex can have any number of edges.
- Vertices are sometimes referred to as _nodes_.



6 Vertices: A, B, C, D, E, F
7 Edges: $(A, E), (A, B), (B, D), (B, F),$
$(C, D), (C, F), (E, F)$

Introduction to Graphs

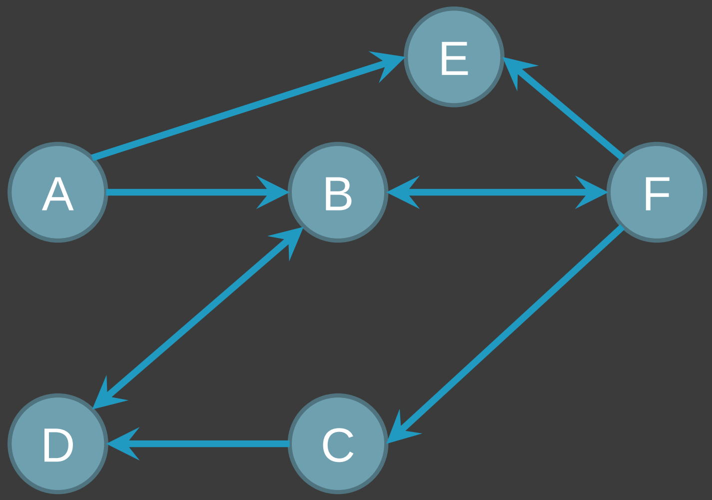
- This graph is ***non-directed***. The edge (A, E) is bidirectional – there is a path from A to E and vice versa.



6 Vertices: A, B, C, D, E, F
7 Edges: $(A, E), (A, B), (B, D), (B, F),$
$(C, D), (C, F), (E, F)$

Introduction to Graphs

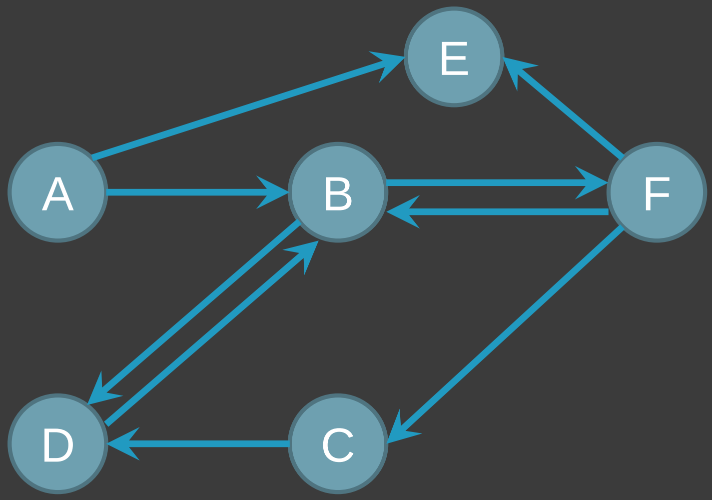
- This graph is ***directed***. Some of the edges are one-way (directional), and some are two-way (bidirectional). Arrows indicate the direction(s) of the edges. A bidirectional edge is counted as two edges in a directed graph.



6 Vertices: A, B, C, D, E, F
9 Edges: $(A, B), (A, E), (B, D), (D, B), (B, F), (F, B), (C, D), (F, C), (F, E)$

Introduction to Graphs

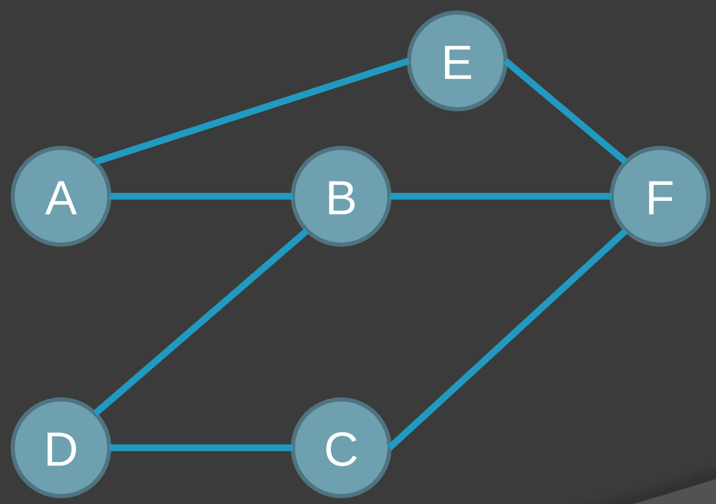
- This graph is ***directed***. Some of the edges are one-way (directional), and some are two-way (bidirectional). Arrows indicate the direction(s) of the edges. A bidirectional edge is counted as two edges in a directed graph.



6 Vertices: A, B, C, D, E, F
9 Edges: $(A, B), (A, E), (B, D), (D, B), (B, F), (F, B), (C, D), (F, C), (F, E)$

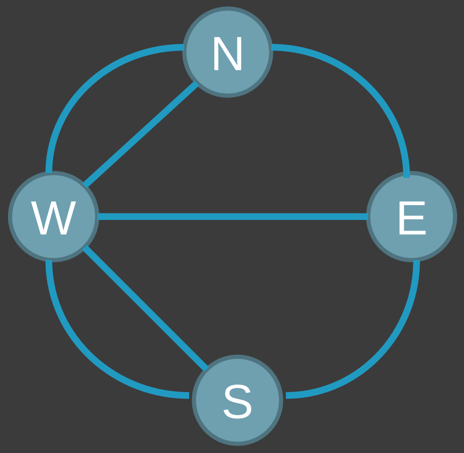
Introduction to Graphs

- The ***degree*** of a node is the number of edges connected to it.
- In this (undirected) graph, nodes A, C, D, & E have degree 2; nodes B & F have degree 3



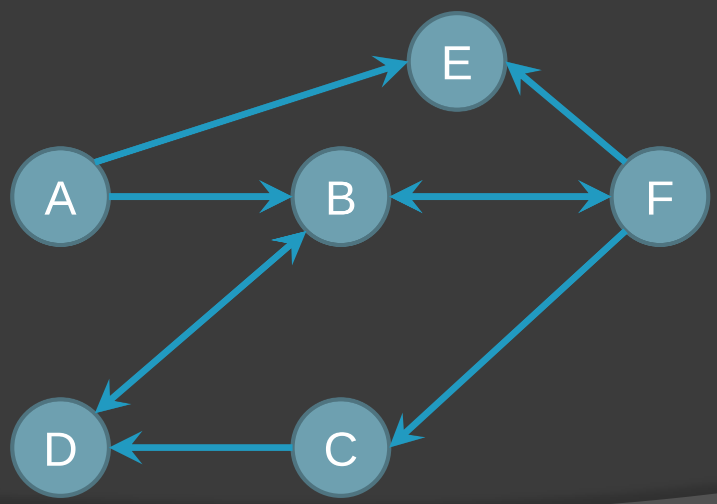
Introduction to Graphs

- Vertex W is of degree 5; nodes N, E, and S are of degree 3
- All four nodes have odd degrees



Introduction to Graphs

- In a directed graph, we also refer to the ***in-degree*** (the number of edges coming into the node) and the ***out-degree*** (outbound edges) of a node.
- The degree of a node in a directed graph is the sum of the two



Node	In-Degree	Out-Degree	Degree
A	0	2	2
B	3	2	5
C	1	1	2
D	2	1	3
E	2	0	2
F	1	2	3

Introduction to Graphs

- The number of edges is denoted by $|E|$, and the number of vertices is $|V|$
- For running time purposes, the text leaves the | | out, i.e. the search time might be $\Theta(E+V)$, which means $\Theta(|E|+|V|)$
- The *vertex set* of a graph G is $G.V$, and the *edge set* of a graph G is $G.E$

Introduction to Graphs

- Given graph $G = (V, E)$
- There are two common ways to represent a graph for algorithms:
  - *Adjacency lists* – a listing of the edges from each vertex, to each vertex
  - *Adjacency matrix* – a $|V|$ x $|V|$ matrix that has a 1 where there’s an edge between a pair of vertices, a 0 if not

Graph Representations

- The adjacency *list* representation is better for ***sparse*** graphs, where there are not a large number of edges (it's more compact)
- The adjacency *matrix* form is preferred for graphs where there are many edges (***dense*** graphs, where $|E| \to |V^2|$), because it is more compact for these types of graphs.
  - It's also convenient for quickly identifying if two vertices are connected. Some algorithms *require* their input to be (or be converted to) an adjacency matrix.

Adjacency Lists for a Graph

- An array Adj of $|V|$ lists, one per vertex.
- Vertex u's list, $Adj[u]$, has all vertices v such that there's as edge $(u, v) \in E$
  - This works for both directed and undirected graphs
- If edges have *weights*, we can put the weights in the lists.
  - Weight: $w : E \rightarrow \mathbf{R}$
- We'll use weights later on for a couple of algorithms

Adjacency Lists for a Graph

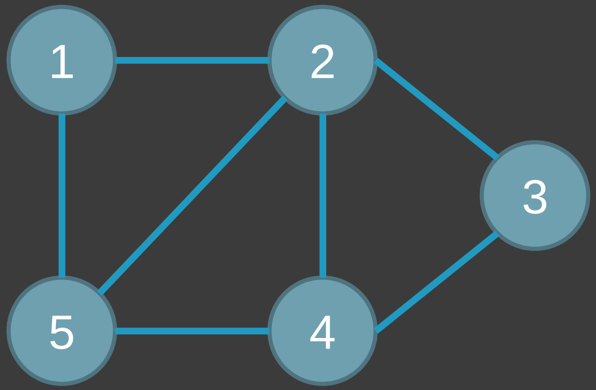
- A disadvantage with the lists is that there has to be a search algorithm through the list of a vertex to see if there is an edge to another vertex (remedies exist for this to help out, though)
- An advantage is that the lists can be modified to support other graph variations
- They can also take less space to store than a matrix

Adjacency Lists for a Graph

- For a directed graph, there is a total of $|\mathrm{E}|$ items across all of the adjacency lists
- For an undirected graph, there are $2|E|$
  - Since each edge is bidirectional, there's an edge from u to v and another from v to u.

Adjacency Lists for a Graph

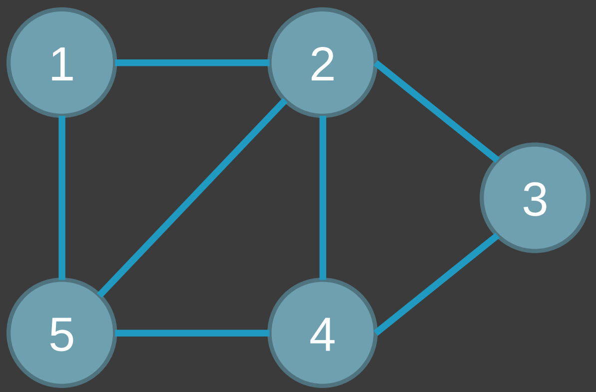
- Consider this graph:
- Is it directed or non-directed?



Its adjacency list representation consists of five lists – the lists of other nodes that can be reached from each of the nodes in the graph.

Adjacency Lists for a Graph

- Consider this graph:
- Is it directed or non-directed?

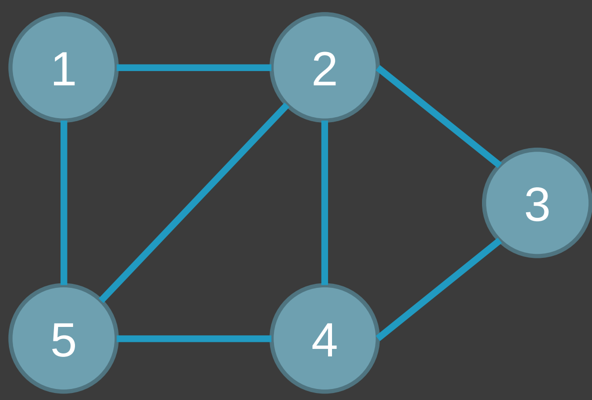


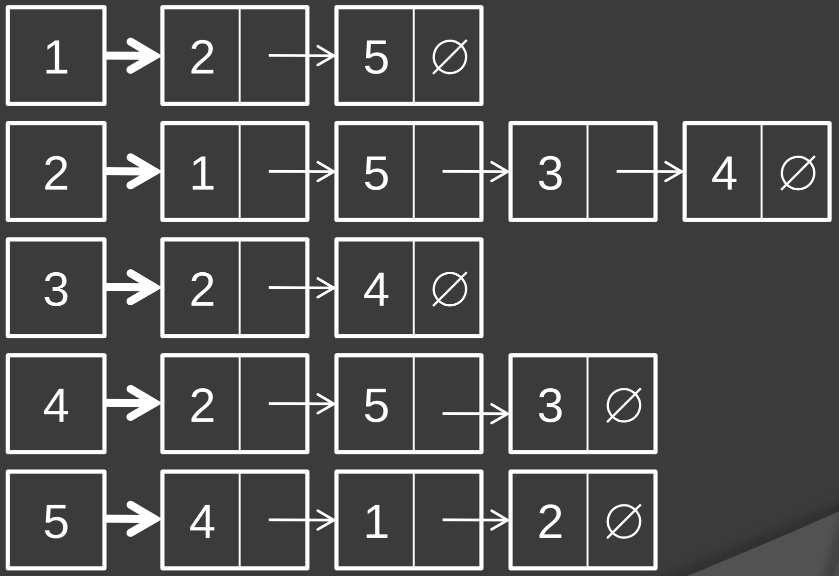
From node 1, we can reach nodes 2, 5
From node 2, we can reach nodes 1, 3, 4, 5
From node 3, we can reach nodes 2, 4
From node 4, we can reach nodes 5, 2, 3
From node 5, we can reach nodes 1, 2, 4

We can represent these lists of reachable nodes as a one-dimensional array of linked lists

Adjacency Lists for a Graph

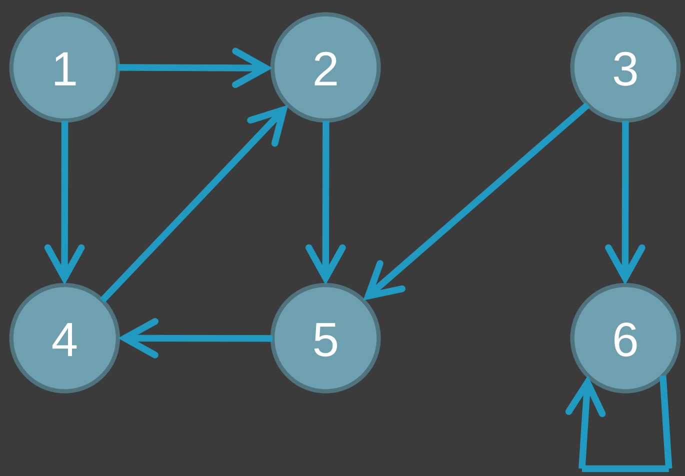
- Consider this graph:
- Is it directed or non-directed?

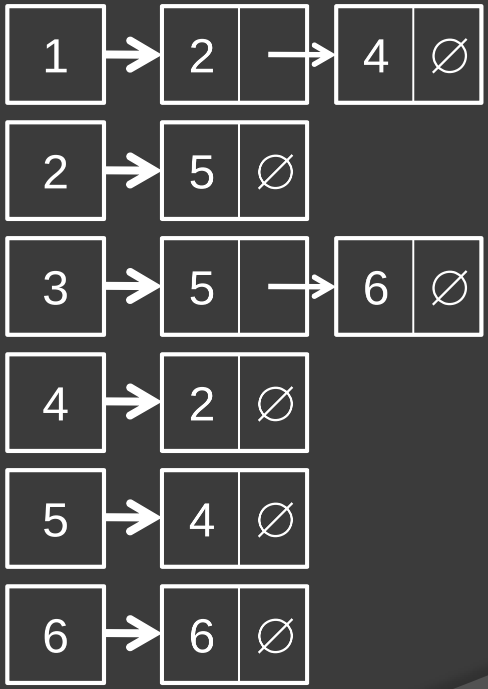




Adjacency Lists for a Graph

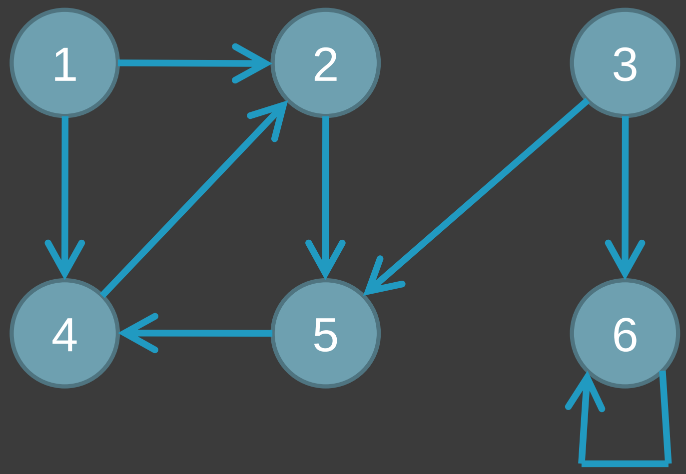
- Consider this directed graph:





Adjacency Matrix for a Graph

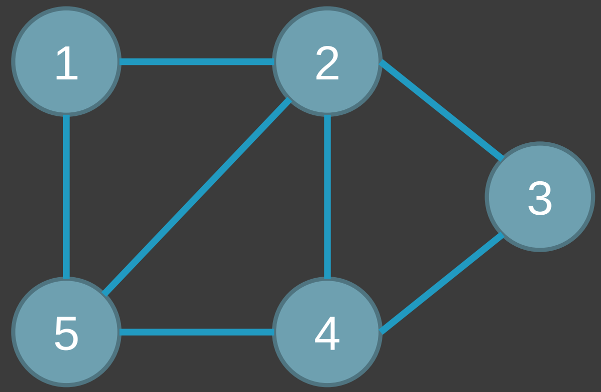
- Consider this directed graph again:



		To Vertex					
		1	**2**	**3**	**4**	**5**	**6**
From Vertex	**1**	0	1	0	1	0	0
	2	0	0	0	0	1	0
	3	0	0	0	0	1	1
	4	0	1	0	0	0	0
	5	0	0	0	1	0	0
	6	0	0	0	0	0	1

Adjacency Lists for a Graph

- Consider this non-directed graph again:

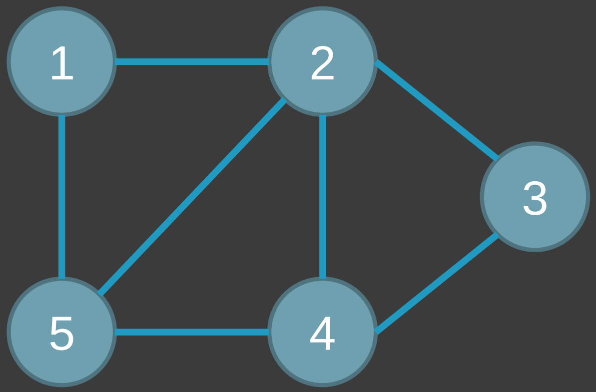


From Vertex \ To Vertex	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

- In a non-directed graph, the adjacency matrix is symmetric about its diagonal

Adjacency Lists for a Graph

- Consider this non-directed graph again:



		To Vertex				
		1	2	3	4	5
From Vertex	1	0	1	0	0	1
	2	1	0	1	1	1
	3	0	1	0	1	0
	4	0	1	1	0	1
	5	1	1	0	1	0

- The *transpose* A^{T} of a matrix is the matrix such that $a^{\mathrm{T}}_{ij} = a_{ji}$. In a non-directed graph, the adjacency matrix equals its transpose ($A = A^{\mathrm{T}}$)

Implementing Weights

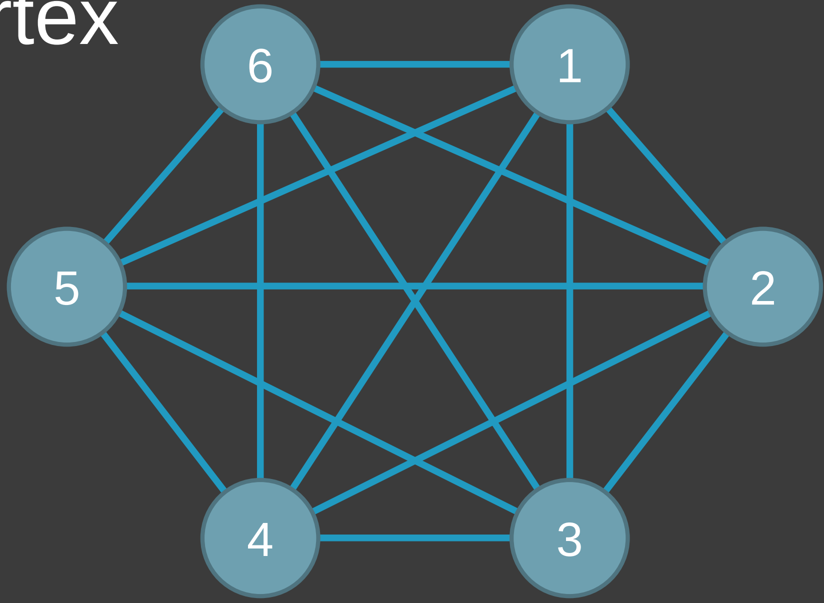
- Adjacency list:
  - Space to store: $\Theta(V+E)$
    - Linked list nodes consist of (at least) a vertex number and a pointer to the next node
  - Time to determine if edge $(u, v) \in E$: $\Theta(degree[u])$
- Adjacency matrix:
  - Space to store: $\Theta(V^2)$
  - Time to determine if edge $(u, v) \in E$: $\Theta(1)$
  - In a non-weighted graph, we can use a single bit for each cell in the matrix

Implementing Weights

- When we need to store the weight corresponding to an edge:
  - We can simply integrate the weight value into the node structure of the linked list in the list-based representation (each node stores a vertex number and a weight)
  - We can use the weight, rather than 1 in the matrix. Sometimes, we will use ∞ to signify a nonexistent edge, rather than 0 (if the weights represent "cost", then an infinite-cost edge is one we can't take).
    - Weights can even be negative in a weighted graph.

More Terms

- A ***fully-connected*** or ***complete*** graph is one in which every vertex has an edge to every other vertex



$$\frac{|V|^2 - |V|}{2}$$

- How many edges are there in *any* fully-connected graph?

More Terms

- A ***path*** of ***length*** k from one vertex u to some other vertex u' is a sequence of vertices $\langle v_0, v_1, \ldots, v_k \rangle$ such that $u = v_0$ and $u' = v_k$ and edge $(v_{i-1}, v_i) \in E$ for $i = 1, 2, \ldots, k$
- If there is a path p from node u to node u', we say that u' is ***reachable*** from u via p.
- A path is ***simple*** if the vertices $\langle v_0, v_1, \ldots, v_k \rangle$ are all distinct
- If a path $p = \langle v_0, v_1, \ldots, v_k \rangle$, and $v_0 = v_k$, then the path is called a ***cycle***.

Trees and Graphs

- A tree is a special case of a graph:
  - A tree is a directed graph
  - Every node except the root has an in-degree of 1
  - The root has an in-degree of zero
  - Every node has an out-degree (number of children) of 0, 1, or 2

Breadth-First Search (BFS)

- Given a graph $G = (V, E)$, and a starting vertex $s \in V$, discover all vertices that are reachable from s, along with the path length from s to each reachable vertex.
- BFS works by systematically expanding the "circle" around s.
- It discovers all nodes whose distance from s is k before looking for nodes at distance $k + 1$

Breadth-First Search (BFS)

- We start with a simplification of the algorithm in the book
- The output of the algorithm is $v.d$, the distance from s to all reachable vertices v.
- Also produces a “BFS Tree”, giving us the path from s to all vertices reachable from s
- For now, we omit the coloring scheme and the predecessor list ($v.\pi$)
- For now, assume undirected, unweighted graph – the weight of all edges is 1

Breadth-First Search (BFS)

- The idea behind BFS:
- Send out a “wave” from s
- The wave will first hit all vertices 1 edge away from s
- Then, it hits all vertices 2 edges away from s
- Uses a FIFO queue to maintain a list of all nodes at the wavefront

BFS Algorithm

BFS(G, s)

1 **for** each $u \in G.V - \{s\}$
2 $\quad u.d = \infty$
3 $s.d = 0$
4 $Q = \emptyset$
5 ENQUEUE(Q, s)
6 **while** $Q \neq \emptyset$
7 $\quad u =$ DEQUEUE(Q)
8 $\quad$ **for** each $v \in G.Adj[u]$
9 $\quad\quad$ **if** $v.d == \infty$
10 $\quad\quad\quad v.d = u.d + 1$
11 $\quad\quad\quad$ ENQUEUE(Q, v)

<u>Initialization</u>:

1, 2: For all nodes OTHER than s, set the shortest known distance to those nodes to ∞.

3: Set the distance to s to 0.

4, 5: Set the queue Q to contain nothing other than s

BFS Algorithm

```
BFS(G, s)
 1  for each u ∈ G.V – {s}
 2        u.d = ∞
 3  s.d = 0
 4  Q = ∅
 5  ENQUEUE(Q, s)
 6  while Q ≠ ∅
 7        u = DEQUEUE(Q)
 8        for each v ∈ G.Adj[u]
 9              if v.d == ∞
10                    v.d = u.d + 1
11                    ENQUEUE(Q, v)
```

<u>Main Processing</u>:

7: Pull the first item u from the queue

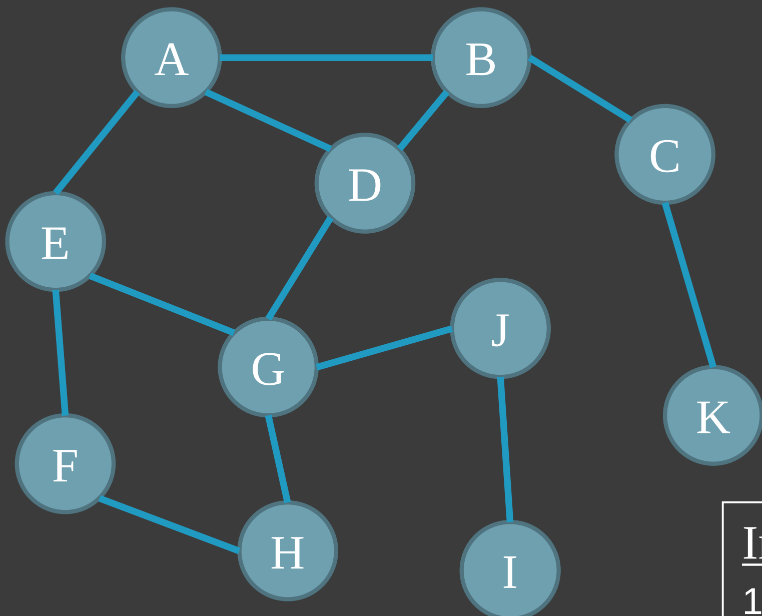
8: For each node v adjacent to u,

9: If we haven't visited node v yet,

10: Mark the distance to v as 1 more than the distance to u, and put v in the queue

6: Repeat 7-10 until the queue is empty

BFS Algorithm - Walkthrough



Let's start at vertex D.
Goal: Find all vertices reachable from D, and the distance from D to each reachable vertex

v	$v.d$
A	∞
B	∞
C	∞
D	0
E	∞
F	∞
G	∞
H	∞
I	∞
J	∞
K	∞

<u>Initialize:</u>

1, 2: For all nodes OTHER than s, set the shortest known distance to those nodes to ∞.

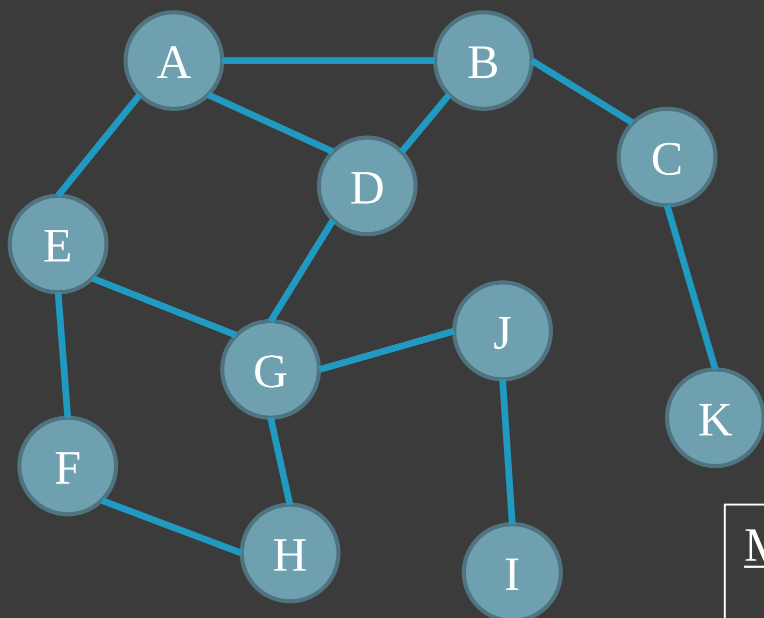
3: Set the distance to s to 0.

4, 5: Initialize the queue Q to contain nothing other than s

Queue Q

D

BFS Algorithm - Walkthrough



Let's start at vertex D.
Goal: Find all vertices reachable from D, and the distance from D to each reachable vertex

v	$v.d$
A	∞
B	∞
C	∞
D	0
E	∞
F	∞
G	∞
H	∞
I	∞
J	∞
K	∞

Main Processing Loop:

7: Pull the first item u from the queue
8: For each node v adjacent to u,
9: If we haven't visited node v yet,
10: Mark the distance to v as 1 more than the distance to u, and put v in the queue
6: Repeat until queue Q is empty

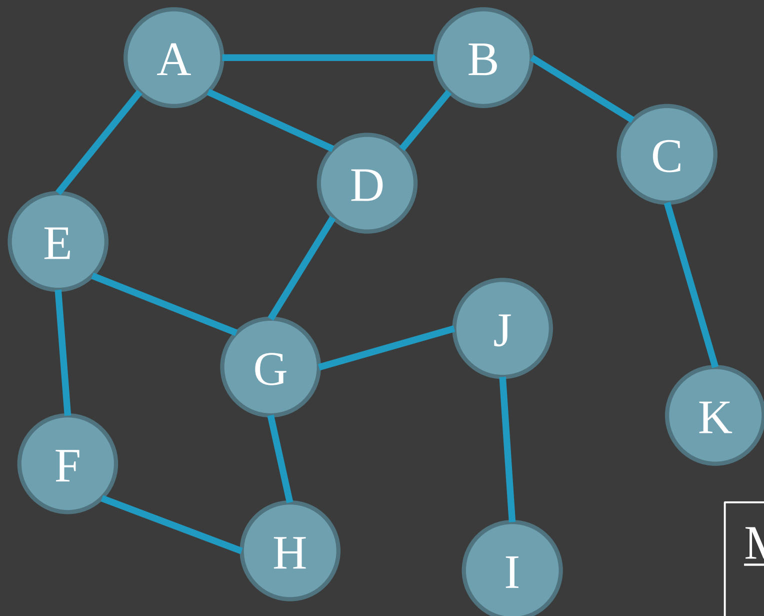
Queue Q

D

u = D

adj[D] = A, B, G

BFS Algorithm - Walkthrough



Let's start at vertex D.
Goal: Find all vertices reachable from D, and the distance from D to each reachable vertex

v	*v.d*
A	1
B	1
C	∞
D	0
E	∞
F	∞
G	1
H	∞
I	∞
J	∞
K	∞

<u>Main Processing Loop:</u>

7: Pull the first item u from the queue
8: For each node v adjacent to u,
9: If we haven't visited node v yet,
10: Mark the distance to v as 1 more than the distance to u, and put v in the queue
6: Repeat until queue Q is empty

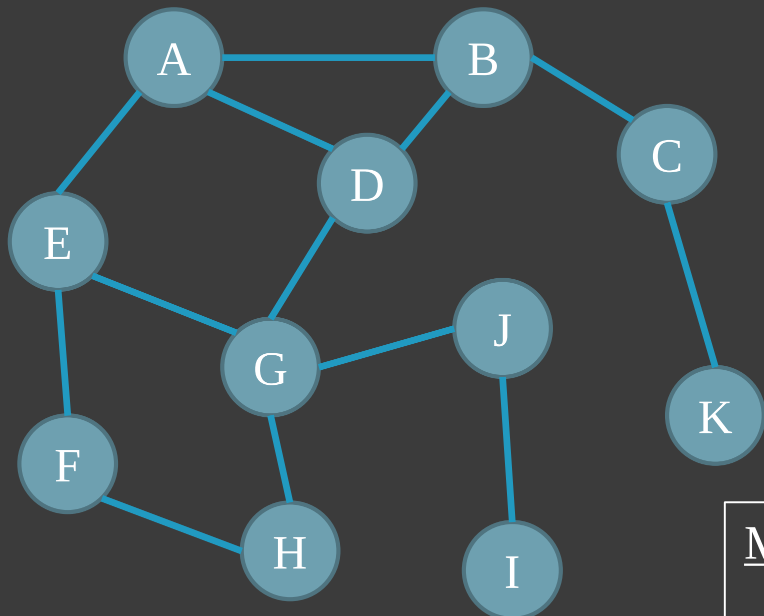
Queue Q

G, B, A

u = A

adj[A] = B, D, E

BFS Algorithm - Walkthrough



Let's start at vertex D.
Goal: Find all vertices reachable from D, and the distance from D to each reachable vertex

v	$v.d$
A	1
B	1
C	∞
D	0
E	2
F	∞
G	1
H	∞
I	∞
J	∞
K	∞

<u>Main Processing Loop:</u>

7: Pull the first item u from the queue
8: For each node v adjacent to u,
9: If we haven't visited node v yet,
10: Mark the distance to v as 1 more than the distance to u, and put v in the queue
6: Repeat until queue Q is empty

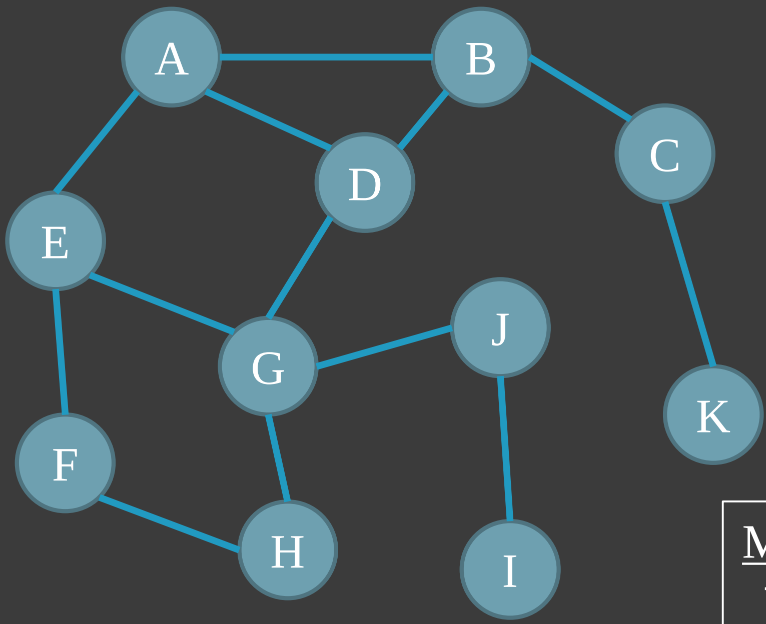
Queue Q

E, G, B

u = B

adj[B] = A, C, D

BFS Algorithm - Walkthrough



Let's start at vertex D.
Goal: Find all vertices reachable from D, and the distance from D to each reachable vertex

v	$v.d$
A	1
B	1
C	2
D	0
E	2
F	∞
G	1
H	∞
I	∞
J	∞
K	∞

Main Processing Loop:

7: Pull the first item u from the queue
8: For each node v adjacent to u,
9: If we haven't visited node v yet,
10: Mark the distance to v as 1 more than the distance to u, and put v in the queue
6: Repeat until queue Q is empty

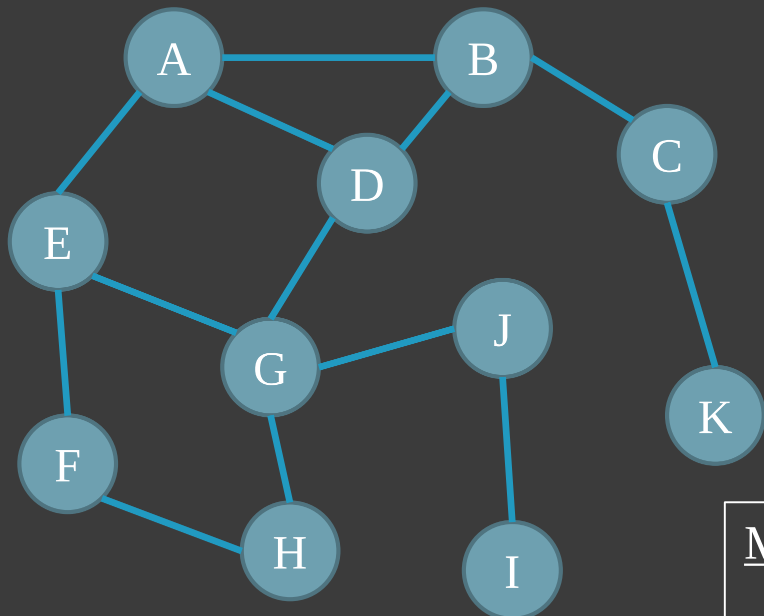
Queue Q

C, E, G

u = G

adj[G] = D, E, H, J

BFS Algorithm - Walkthrough



Let's start at vertex D.
Goal: Find all vertices reachable from D, and the distance from D to each reachable vertex

v	$v.d$
A	1
B	1
C	2
D	0
E	2
F	∞
G	1
H	2
I	∞
J	2
K	∞

<u>Main Processing Loop:</u>

7: Pull the first item u from the queue
8: For each node v adjacent to u,
9: If we haven't visited node v yet,
10: Mark the distance to v as 1 more than the distance to u, and put v in the queue
6: Repeat until queue Q is empty

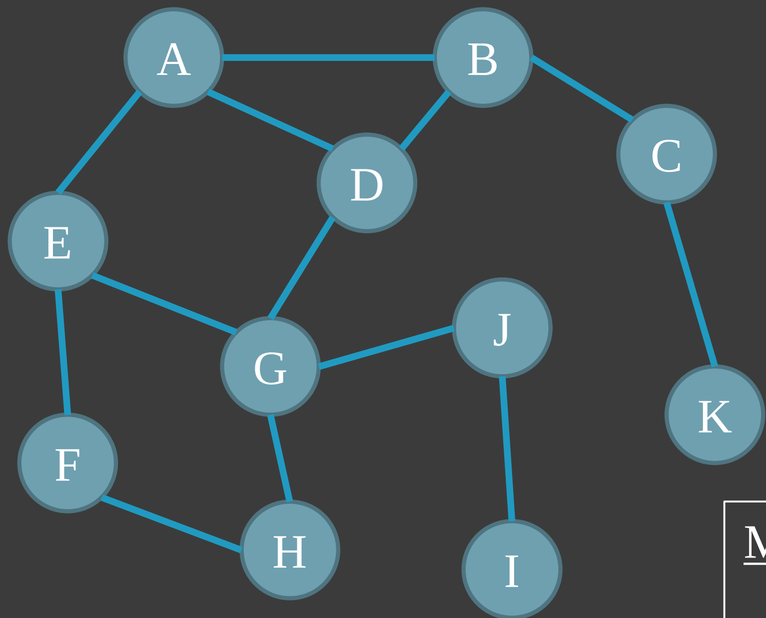
Queue Q

H, J, C, E

u = E

adj[E] = A, G, F

BFS Algorithm - Walkthrough



Let's start at vertex D.
Goal: Find all vertices reachable from D, and the distance from D to each reachable vertex

v	$v.d$
A	1
B	1
C	2
D	0
E	2
F	3
G	1
H	2
I	∞
J	2
K	∞

<u>Main Processing Loop:</u>

7: Pull the first item u from the queue
8: For each node v adjacent to u,
9: If we haven't visited node v yet,
10: Mark the distance to v as 1 more than the distance to u, and put v in the queue
6: Repeat until queue Q is empty

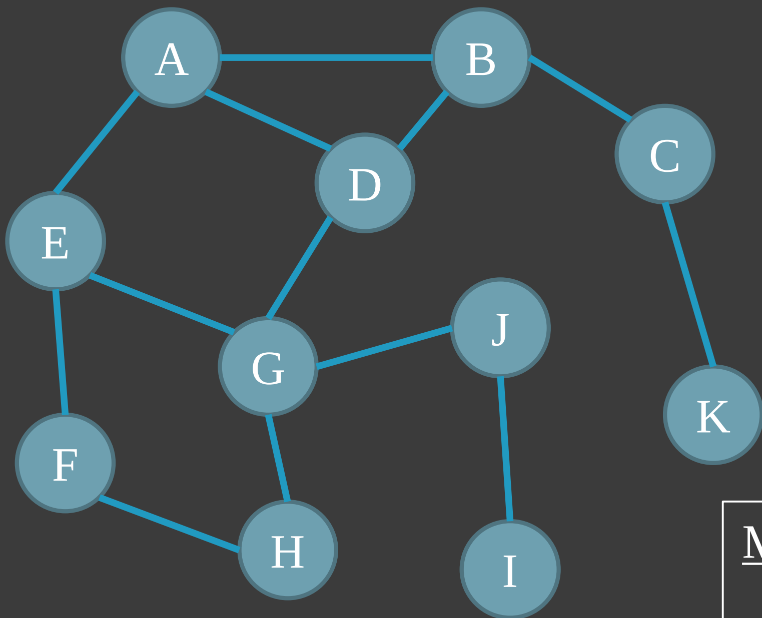
Queue Q

F, H, J, C

u = C

adj[C] = B, K

BFS Algorithm - Walkthrough



Let's start at vertex D.
Goal: Find all vertices reachable from D, and the distance from D to each reachable vertex

v	$v.d$
A	1
B	1
C	2
D	0
E	2
F	3
G	1
H	2
I	∞
J	2
K	3

Main Processing Loop:

7: Pull the first item u from the queue
8: For each node v adjacent to u,
9: If we haven't visited node v yet,
10: Mark the distance to v as 1 more than the distance to u, and put v in the queue
6: Repeat until queue Q is empty

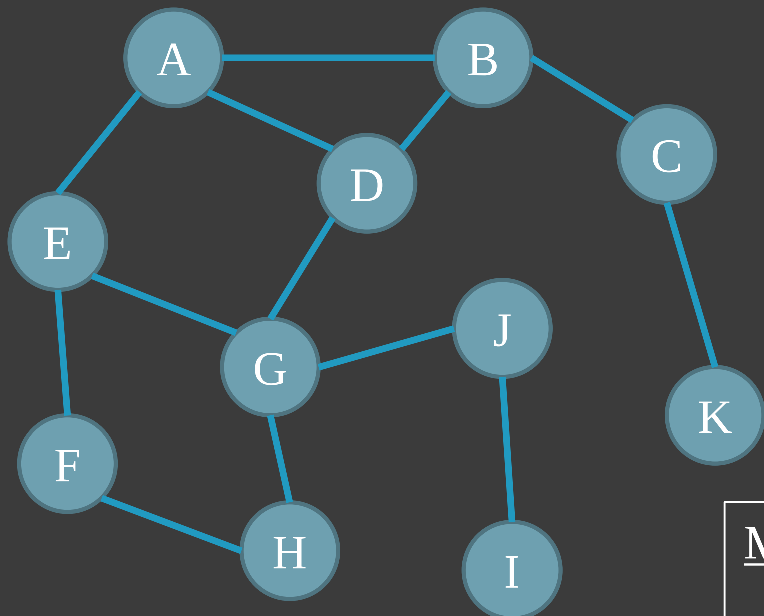
Queue Q

K, F, H, J

u = J

adj[J] = G, I

BFS Algorithm - Walkthrough



Let's start at vertex D.
Goal: Find all vertices reachable from D, and the distance from D to each reachable vertex

v	$v.d$
A	1
B	1
C	2
D	0
E	2
F	3
G	1
H	2
I	3
J	2
K	3

Main Processing Loop:

7: Pull the first item u from the queue
8: For each node v adjacent to u,
9: If we haven't visited node v yet,
10: Mark the distance to v as 1 more than the distance to u, and put v in the queue
6: Repeat until queue Q is empty

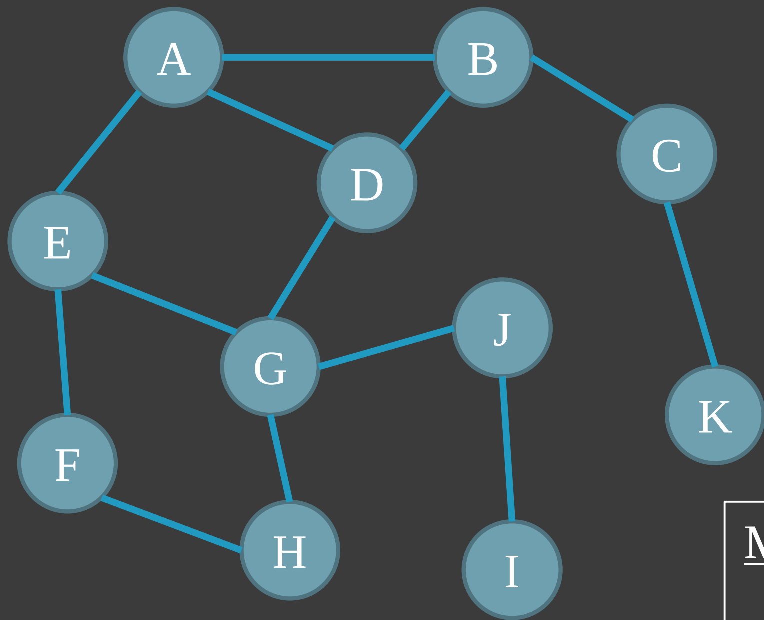
Queue Q

I, K, F, H

u = H

adj[H] = F, G

BFS Algorithm - Walkthrough



Let's start at vertex D.
Goal: Find all vertices reachable from D, and the distance from D to each reachable vertex

v	*v.d*
A	1
B	1
C	2
D	0
E	2
F	3
G	1
H	2
I	3
J	2
K	3

Main Processing Loop:

7: Pull the first item u from the queue
8: For each node v adjacent to u,
9: If we haven't visited node v yet,
10: Mark the distance to v as 1 more than the distance to u, and put v in the queue
6: Repeat until queue Q is empty

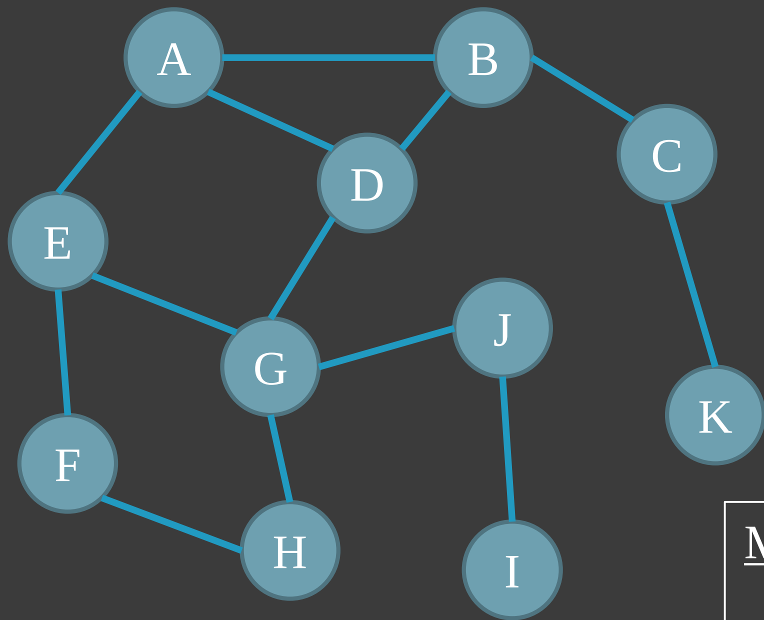
Queue Q

I, K, F

u = F

adj[F] = E, H

BFS Algorithm - Walkthrough



Let's start at vertex D.
Goal: Find all vertices reachable from D, and the distance from D to each reachable vertex

v	$v.d$
A	1
B	1
C	2
D	0
E	2
F	3
G	1
H	2
I	3
J	2
K	3

<u>Main Processing Loop:</u>

7: Pull the first item u from the queue
8: For each node v adjacent to u,
9: If we haven't visited node v yet,
10: Mark the distance to v as 1 more than the distance to u, and put v in the queue
6: Repeat until queue Q is empty

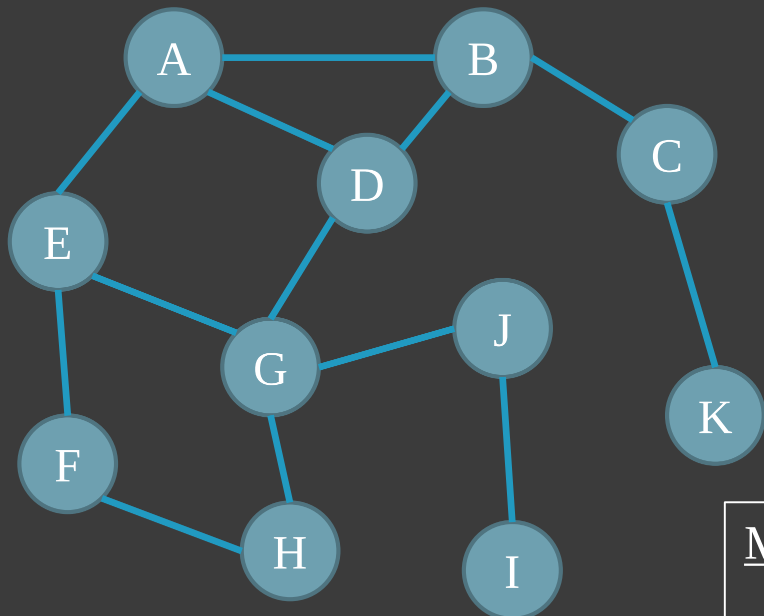
Queue Q

I, K

u = K

adj[K] = C

BFS Algorithm - Walkthrough



Let's start at vertex D.
Goal: Find all vertices reachable from D, and the distance from D to each reachable vertex

v	$v.d$
A	1
B	1
C	2
D	0
E	2
F	3
G	1
H	2
I	3
J	2
K	3

Main Processing Loop:

7: Pull the first item u from the queue
8: For each node v adjacent to u,
9: If we haven't visited node v yet,
10: Mark the distance to v as 1 more than the distance to u, and put v in the queue
6: Repeat until queue Q is empty

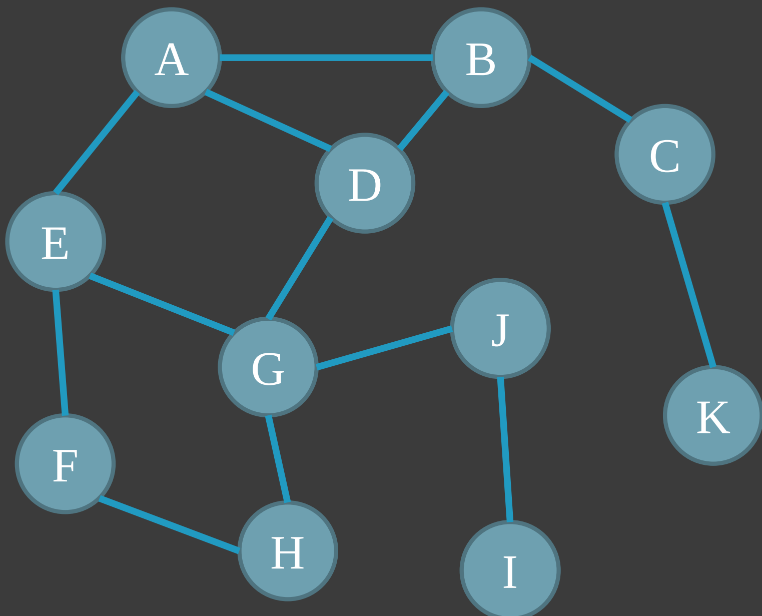
Queue Q

I

u = I

adj[I] = J

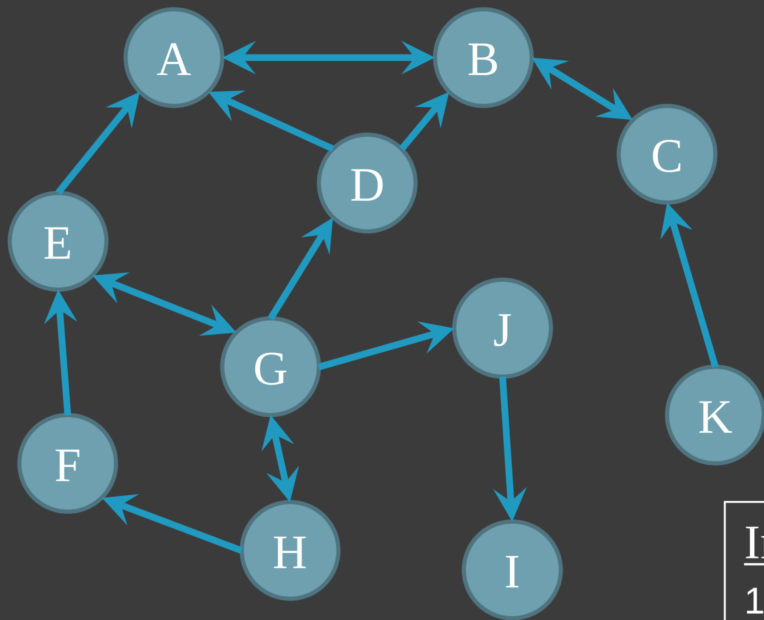
What's in the Queue?



Queue Q	Distances
D	0
GBA	111
EGB	211
CEG	221
HJCE	2222
FHJC	3222
KFHJ	3322
IKFH	3332
IKF	333
IK	33
I	3

v	*v.d*
A	1
B	1
C	2
D	0
E	2
F	3
G	1
H	2
I	3
J	2
K	3

What About Directed Graphs?



Let's start at vertex D.
Goal: Find all vertices reachable from D, and the distance from D to each reachable vertex

v	$v.d$
A	∞
B	∞
C	∞
D	0
E	∞
F	∞
G	∞
H	∞
I	∞
J	∞
K	∞

Initialize:

1, 2: For all nodes OTHER than s, set the shortest known distance to those nodes to ∞.

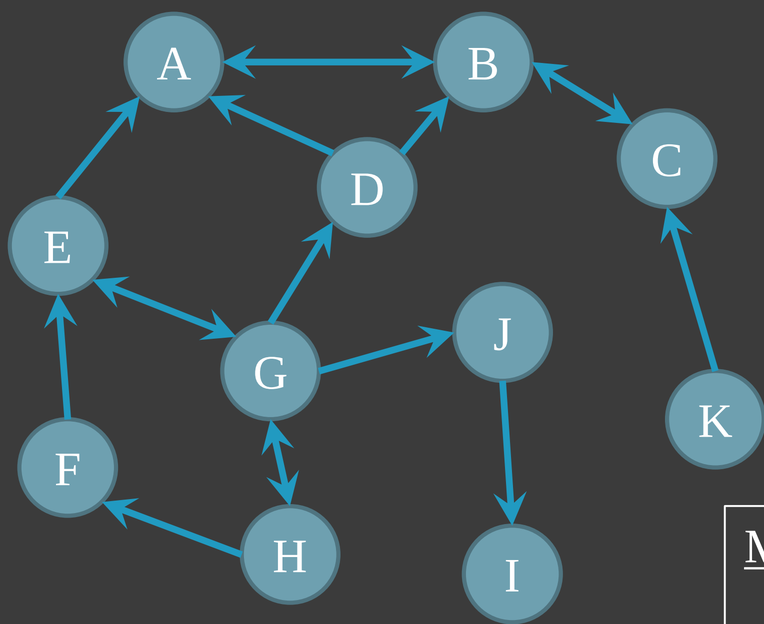
3: Set the distance to s to 0.

4, 5: Initialize the queue Q to contain nothing other than s

Queue Q

D

What About Directed Graphs?



Let's start at vertex D.
Goal: Find all vertices reachable from D, and the distance from D to each reachable vertex

v	$v.d$
A	∞
B	∞
C	∞
D	0
E	∞
F	∞
G	∞
H	∞
I	∞
J	∞
K	∞

Main Processing Loop:

7: Pull the first item u from the queue
8: For each node v adjacent to u,
9: If we haven't visited node v yet,
10: Mark the distance to v as 1 more than the distance to u, and put v in the queue
6: Repeat until queue Q is empty

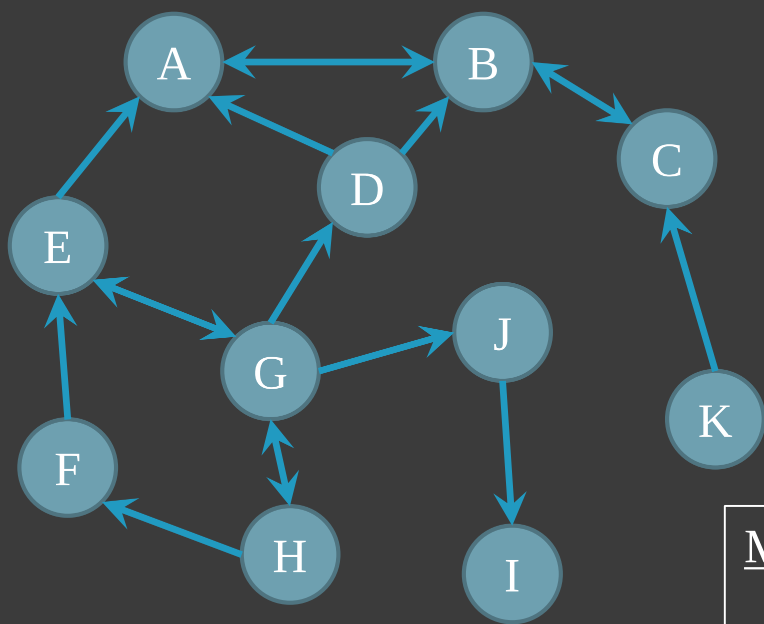
Queue Q

D

u = D

adj[D] = A, B

What About Directed Graphs?



Let's start at vertex D.
Goal: Find all vertices reachable from D, and the distance from D to each reachable vertex

v	$v.d$
A	1
B	1
C	∞
D	0
E	∞
F	∞
G	∞
H	∞
I	∞
J	∞
K	∞

<u>Main Processing Loop:</u>

7: Pull the first item u from the queue
8: For each node v adjacent to u,
9: If we haven't visited node v yet,
10: Mark the distance to v as 1 more than the distance to u, and put v in the queue
6: Repeat until queue Q is empty

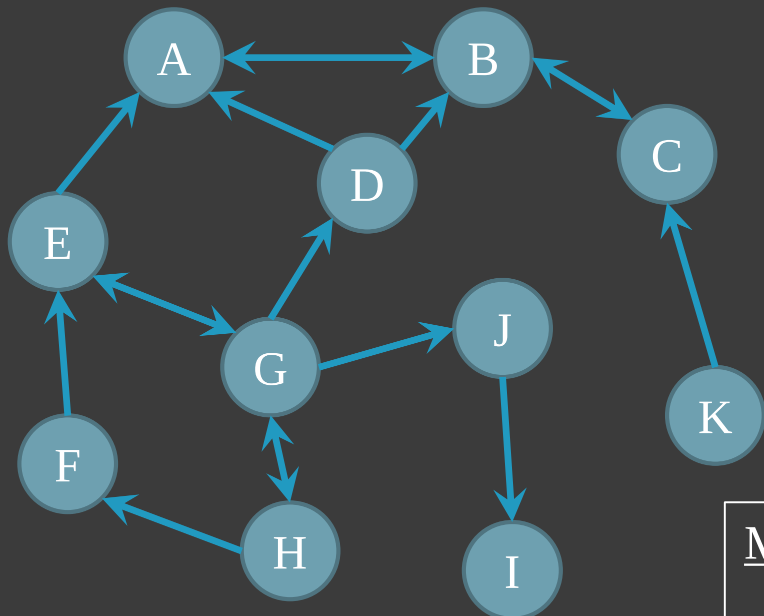
Queue Q

B, A

u = A

adj[A] = B

What About Directed Graphs?



Let's start at vertex D.
Goal: Find all vertices reachable from D, and the distance from D to each reachable vertex

v	$v.d$
A	1
B	1
C	∞
D	0
E	∞
F	∞
G	∞
H	∞
I	∞
J	∞
K	∞

Main Processing Loop:

7: Pull the first item u from the queue
8: For each node v adjacent to u,
9: If we haven't visited node v yet,
10: Mark the distance to v as 1 more than the distance to u, and put v in the queue
6: Repeat until queue Q is empty

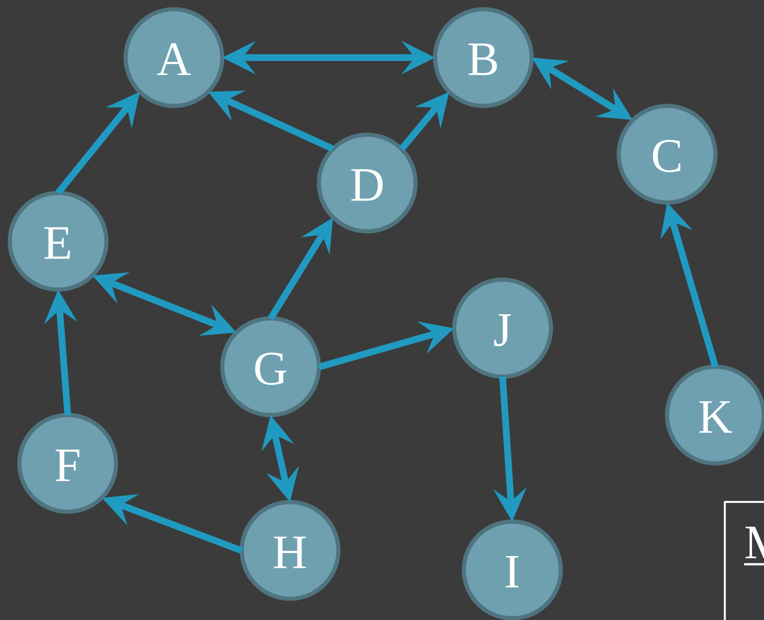
Queue Q

B

u = B

adj[B] = A, C

What About Directed Graphs?



Let's start at vertex D.
Goal: Find all vertices reachable from D, and the distance from D to each reachable vertex

v	$v.d$
A	1
B	1
C	2
D	0
E	∞
F	∞
G	∞
H	∞
I	∞
J	∞
K	∞

Main Processing Loop:

7: Pull the first item u from the queue
8: For each node v adjacent to u,
9: If we haven't visited node v yet,
10: Mark the distance to v as 1 more than the distance to u, and put v in the queue
6: Repeat until queue Q is empty

Queue Q

C

u = C

adj[C] = B

BFS Summary (1)

- Correctness:
  - Since each vertex gets a finite d value at most once, values assigned to vertices are monotonically increasing over time.
    - Actual proof of correctness is a bit trickier. See book.

BFS Summary (2)

- Run Time:
  - $O(V+E)$
    - $O(V)$ because every vertex enqueued at most once.
    - $O(E)$ because every vertex dequeued at most once and we examine edge (u, v) only when u is dequeued.
      - Therefore, every edge examined at most once if directed, at most twice if undirected.
      - Either way, that's still $O(E)$

Software Module

- A well-defined component of a software system
- A part of a system that provides a set of services to other modules
  - Services are computational elements that other modules may use

Questions

- How to define the structure of a modular system?
- What are the desirable properties of that structure?

Modules and relations

- Let S be a set of modules

$$S = \{M_1, M_2, ..., M_n\}$$

- A binary relation r on S is a subset of S x S
- If M_i and M_j are in S, $<M_i, M_j> \in r$ can be written as $M_i \; r \; M_j$

Relations

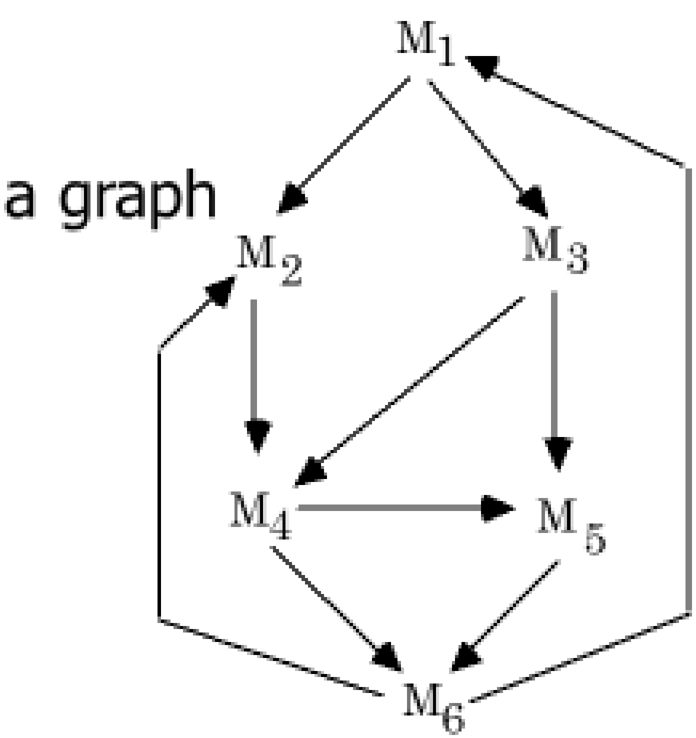
- Transitive closure r^+ of r

  $M_i\ r^+\ M_j$ <u>iff</u>

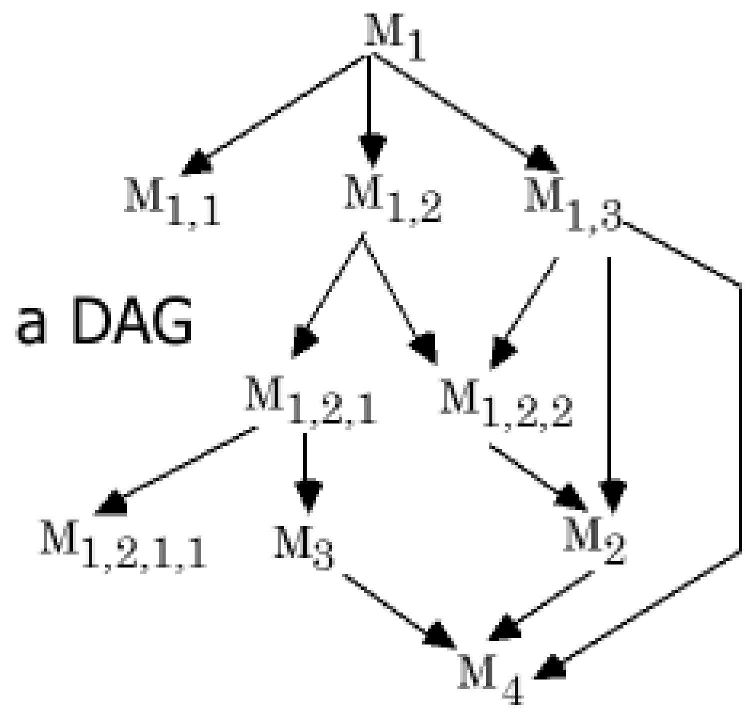
  $M_i\ r\ M_j$ or $\exists\ M_k$ in S s.t. $M_i\ r\ M_k$ and $M_k\ r^+\ M_j$
- (We assume our relations to be irreflexive)
- r is a hierarchy iff there are no two elements M_i, M_j s.t. $M_i\ r^+\ M_j \wedge M_j\ r^+\ M_i$

Relations

- Relations can be represented as graphs
- A hierarchy is a DAG (directed acyclic graph)



a)



b)

The USES relation

- A uses B
  - A requires the correct operation of B
  - A can access the services exported by B through its interface
  - it is “statically” defined
  - A depends on B to provide its services
    - example: A calls a routine exported by B
- A is a client of B; B is a server

Desirable property

- USES should be a hierarchy
- Hierarchy makes software easier to understand
  - we can proceed from leaf nodes (who do not use others) upwards
- They make software easier to build
- They make software easier to test

Hierarchy

- Organizes the modular structure through *levels of abstraction*
- Each level defines an *abstract (virtual) machine* for the next level
  - *level* can be defined precisely
    - M_i has level 0 if no M_j exists s.t. M_i r M_j
    - For each module M_i, let k be the maximum level of all nodes M_j s.t. M_i r M_j. Then M_i has level k+1

Hierarchy: USES example

- Let M_R be a module that provides input-output of record values.
- Let M_R use another module M_B that provides I/O of a single byte at a time.
- When used to output record values, the job of M_R consists of transforming the record into a sequence of bytes and isolating a single byte at a time to be output by means of $M_{B.}$
- M_B provides a service that is used by $M_{R.}$

Module Level Concepts

- Ideally we decompose up to have a minimum of interaction between modules and, conversely, a high degree of interaction within a module.
- Coupling: measure of independence
- Cohesion: logical relationship
- Cohesion and coupling help determine “quality” of the architecture.

Module Level Concepts

- The USES relation provides a way to reason about the coupling in a precise manner.
- With reference to a USES graph, we can distinguish the number of incoming edges (fan-in) and the number of outgoing edges (fan-out).

Module Level Concepts(cont)

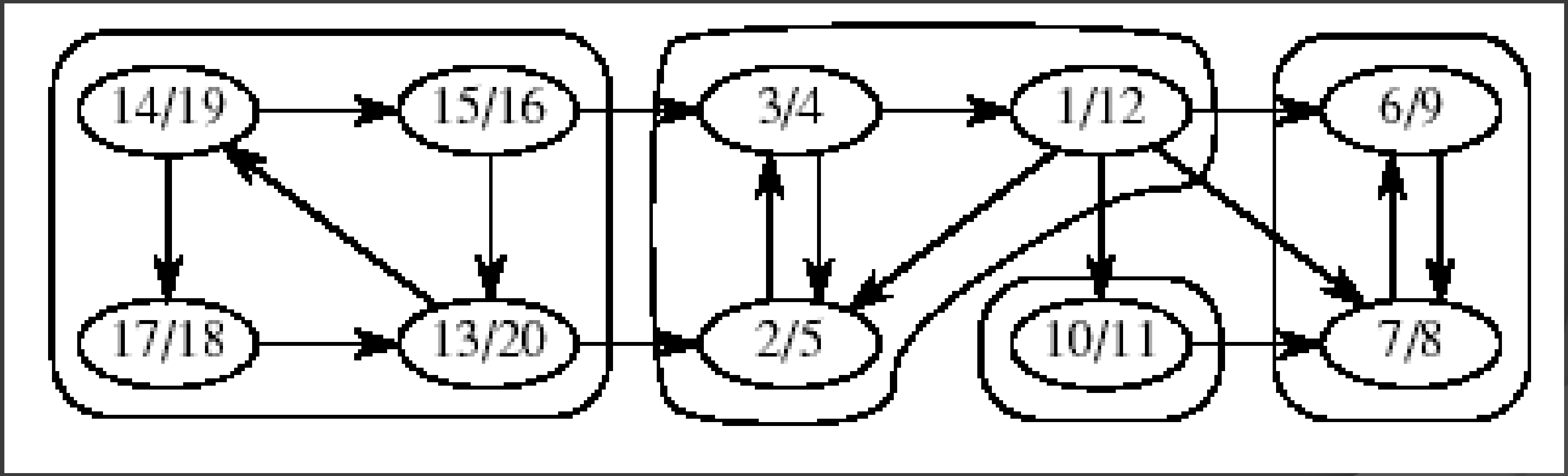
- A good design structure should keep the fan-out low and the fan-in high.

Module Level Concepts

- A high fan-in is an indication of good design because a module with high fan-in represents a meaningful i.e. general abstraction that is used heavily by other modules.
- A high fan-out is an indication that a module is doing too much which in turn may imply that a module has poor cohesion.
- The evaluation of the quality of design should not merely depend on the USES relation.

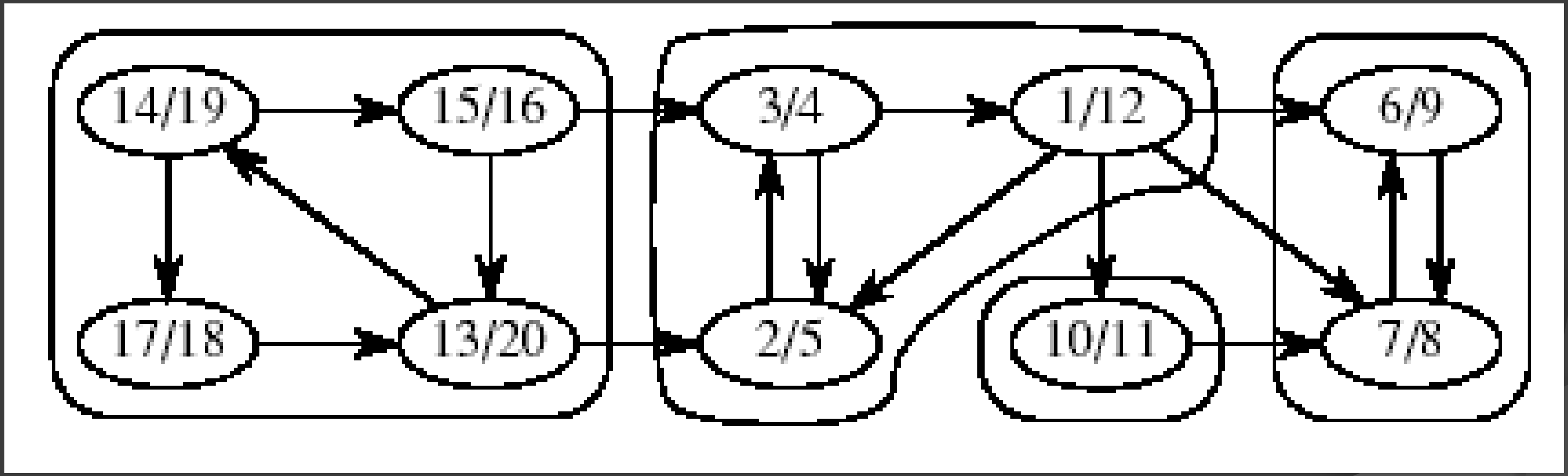
Strongly Connected Components (22.5)

- Below is an example of strongly connected components, including the $v.d$ & $v.f$ times after the depth-first search is run



Strongly Connected Components (22.5)

- Below is an example of strongly connected components, including the $v.d$ & $v.f$ times after the depth-first search is run

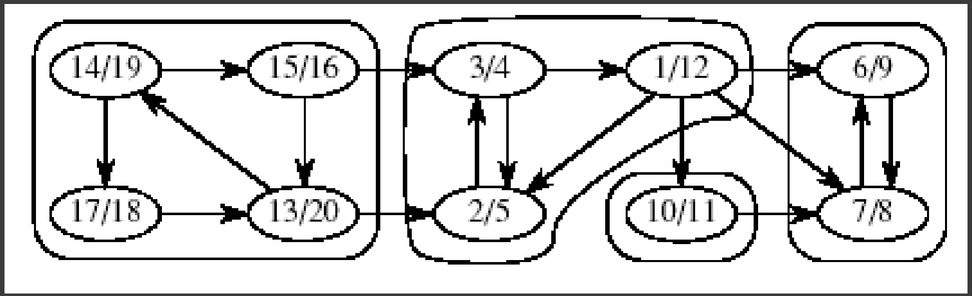


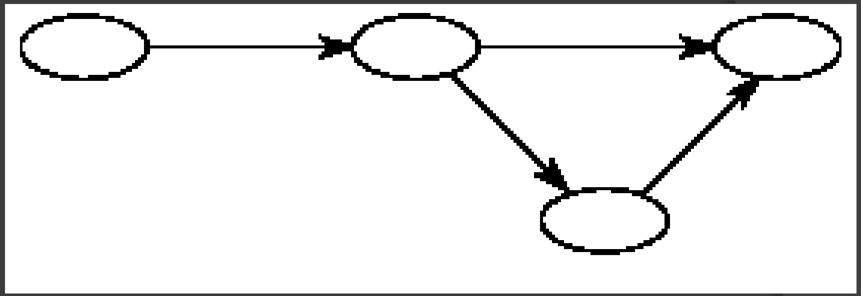
Strongly Connected Components (22.5)

- The algorithm uses G^{T}, made from $G(V, E^{\mathrm{T}})$
  - $E^{\mathrm{T}} = \{(u, v) : (v, u) \in E\}$
  - G^{T} is G with all the edges reverved
- We can create G^{T} in $\Theta(V + E)$ time if using adjacency lists
- G and G^{T} have the same strongly connected components
- u and v are reachable from each other in G iff they are reachable from each other in G^{T}

Strongly Connected Components (22.5)

- The SCC's form their own graph:





- $G^{SCC} = (V^{SCC}, E^{SCC})$
- V^{SCC} has one vertex for each SCC in G
- E^{SCC} has an edge if there's an edge between the corresponding SCCs in G