CPSC 221
Basic Algorithms and Data Structures

June 12, 2015

Administrative stuff

Programming Assignment 2 is posted and due no later than 11:59 pm on Wednesday, June 17.

Exam progress continues to be incremental.

Limitations of trees

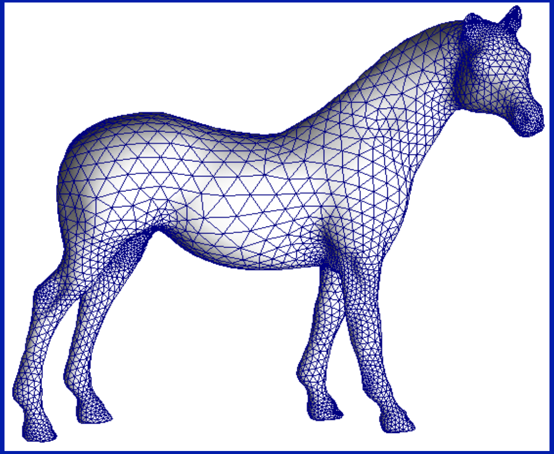
- In past weeks, we’ve seen lots of trees.
- One limitation of the tree data structure is the inability to have nodes with more than one parent
- In addition, the structure of a tree is such that it imposes an ordering on the elements within, which may not always be desirable
- A graph gets around these limitations...

Introduction to graphs

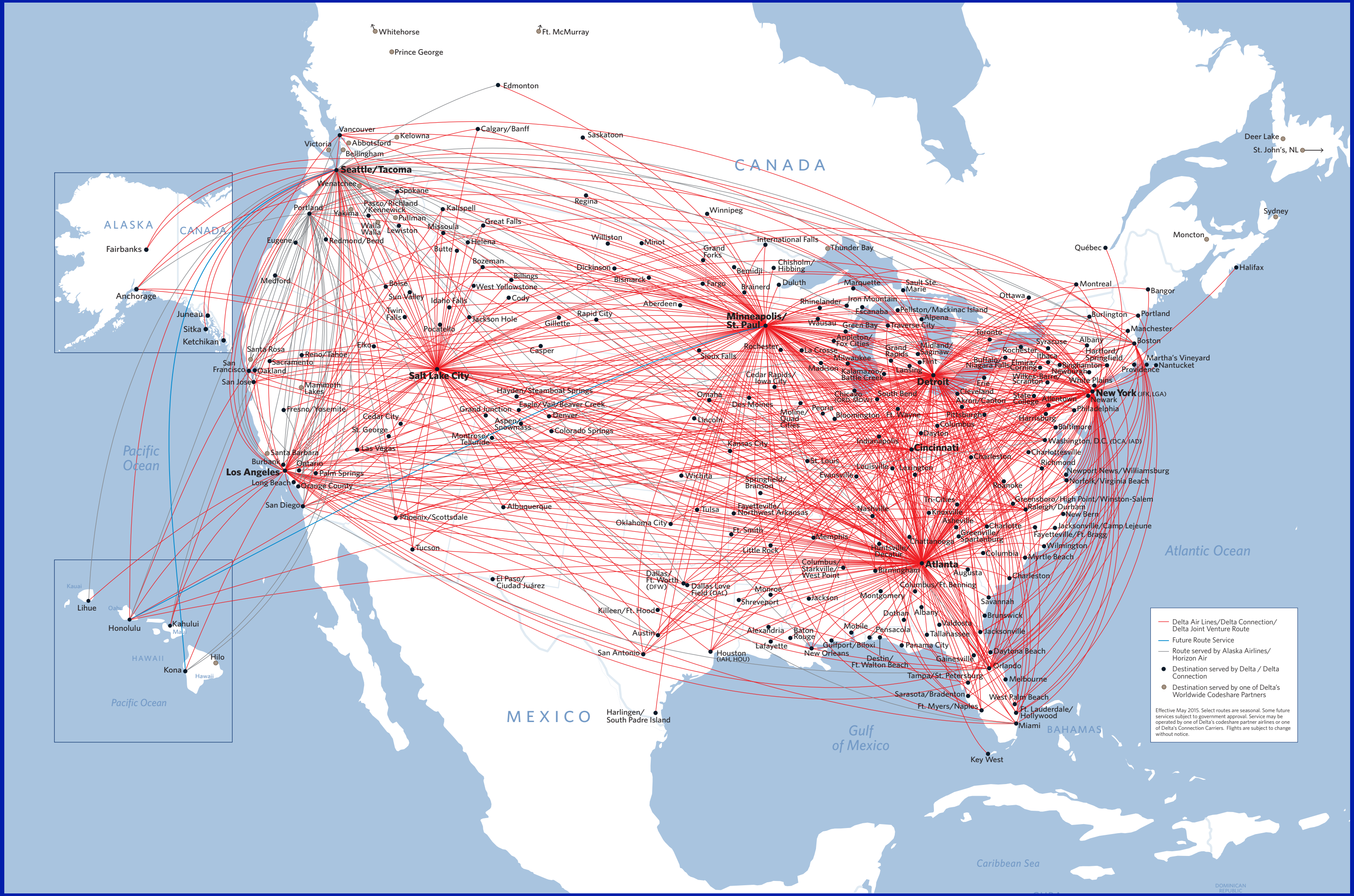
- A graph is a data structure consisting of a finite set of *vertices* (nodes in our trees) and a finite set of *edges* (the connections in our trees) that describe the relationships between the vertices.
- Both the vertex and the edge set may be empty (but note that you cannot have an edge without at least one vertex).
- A tree, therefore, is really just a specialized graph.

Examples of graphs in the wild

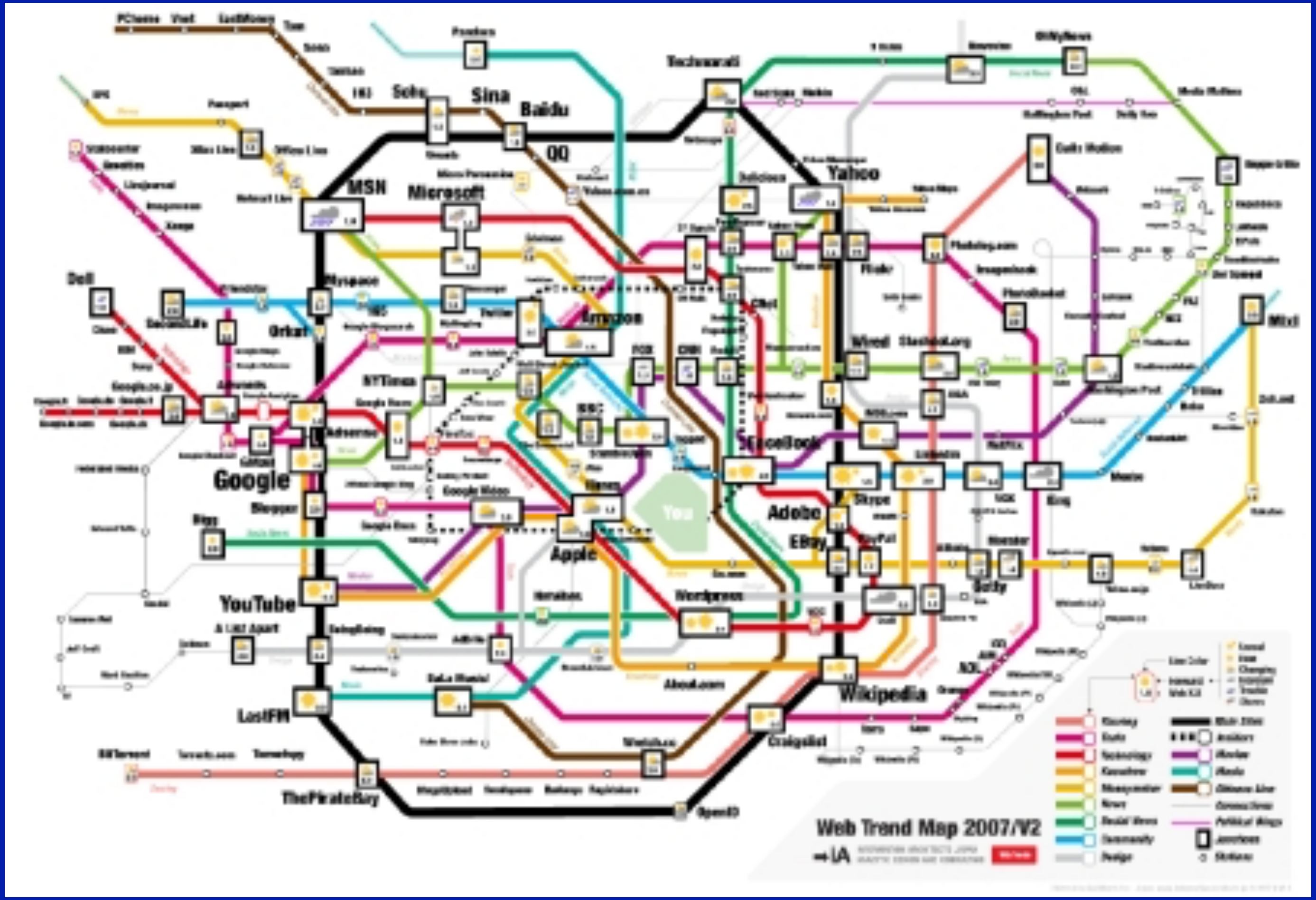
- Graphs are especially useful in analyzing networks, or anything that can be represented as a collection of connected elements.
- E.g. phone systems, Internet, maps, airline routes, course prerequisites, dependency charts, silicon-chip design, plumbing, meshes, etc.
- Google maps and MapQuest



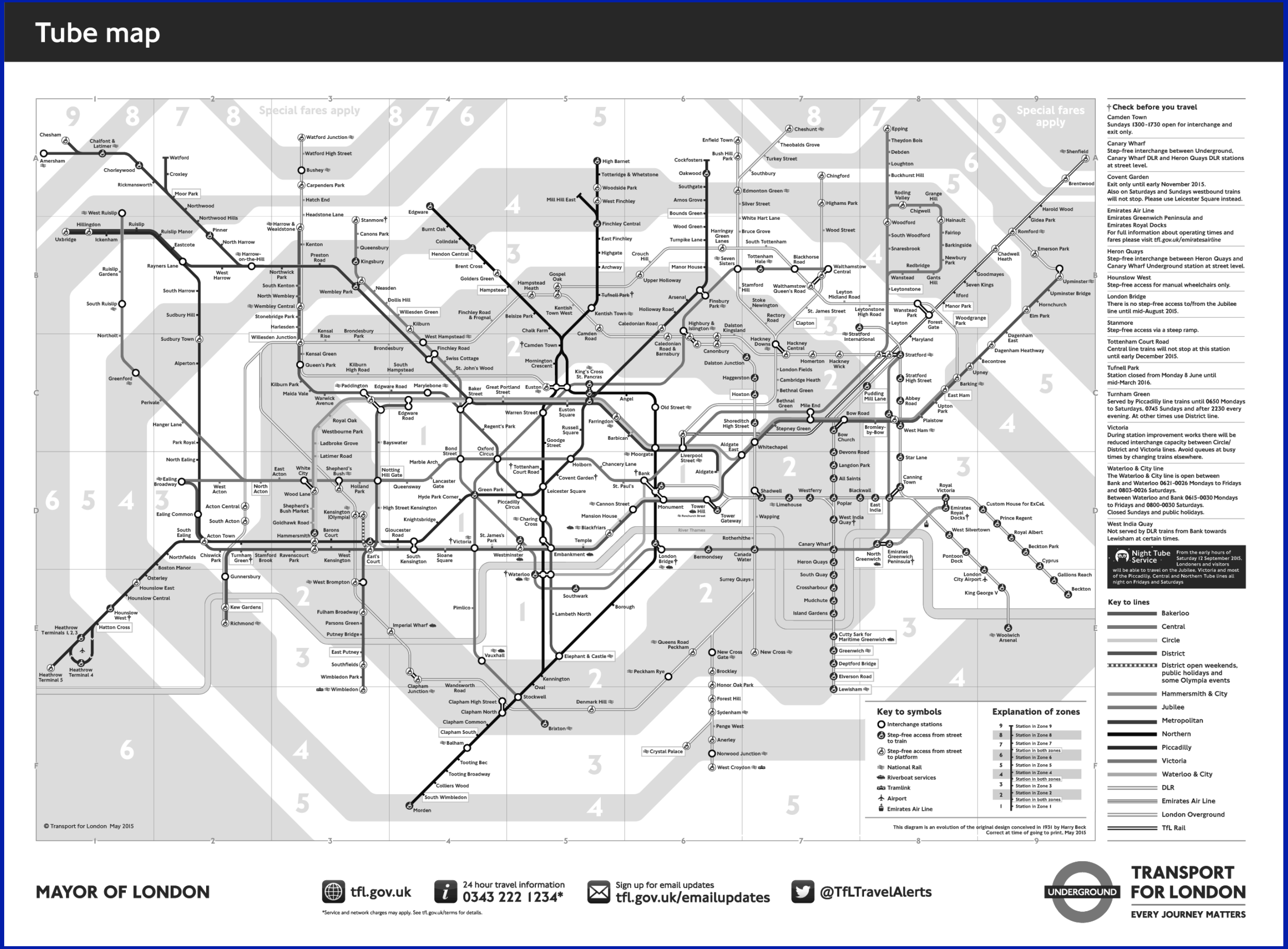
Examples of graphs in the wild



Examples of graphs in the wild



Examples of graphs in the wild



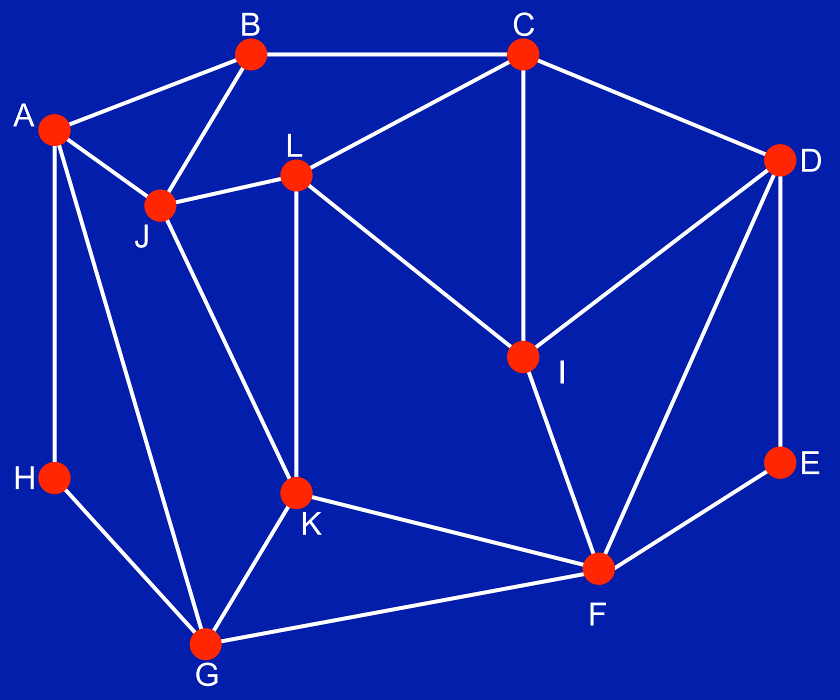
Graphs, more formally

A **graph** G=(V,E) consists of:

(a) a non-empty set of **vertices** V, and

(b) a set of **edges** E between pairs of those vertices.

An edge $e \in E \subseteq V \times V$, where $e = (u,v)$ and $u,v \in V$.



G = <**V**,**E**>

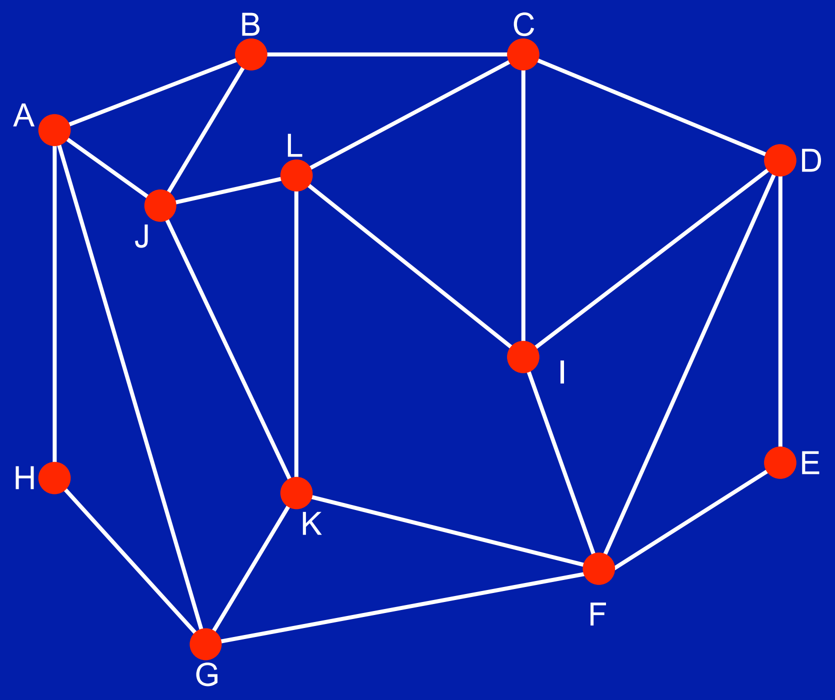
V = vertices = {A,B,C,D,E,F,G,H,I,J,K,L}

E = edges = {(A,B),(B,C),(C,D),(D,E),(E,F),(F,G),(G,H),(H,A),(A,J),(A,G),(B,J),(K,F),(C,L),(C,I),(D,I),(D,F),(F,I),(G,K),(J,L),(J,K),(K,L),(L,I)}

Graphs, more formally

Graphs may be **undirected**:

- Known as an undirected graph, or simply graph.
- $(u,v) = (v,u)$.



G = <**V**,**E**>

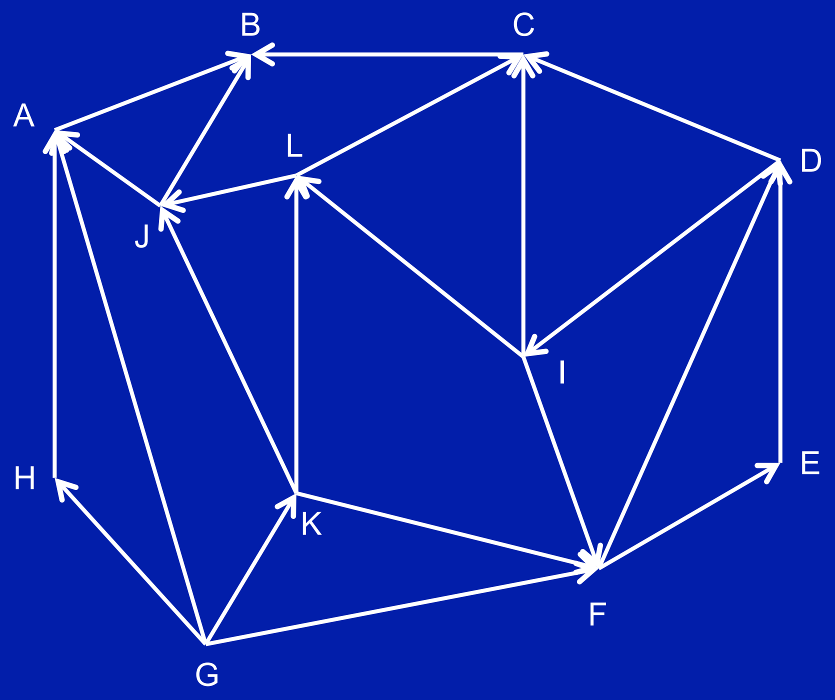
V = vertices = {A,B,C,D,E,F,G,H,I,J,K,L}

E = edges = {(A,B),(B,C),(C,D),(D,E),(E,F),(F,G),(G,H),(H,A),(A,J),(A,G),(B,J),(K,F),(C,L),(C,I),(D,I),(D,F),(F,I),(G,K),(J,L),(J,K),(K,L),(L,I)}

Graphs, more formally

Graphs may be **directed**:

- This is called a directed graph or a digraph.
- The ordered pair (u,v) implies an edge from u to v
- (red dots removed so you can see the arrows)



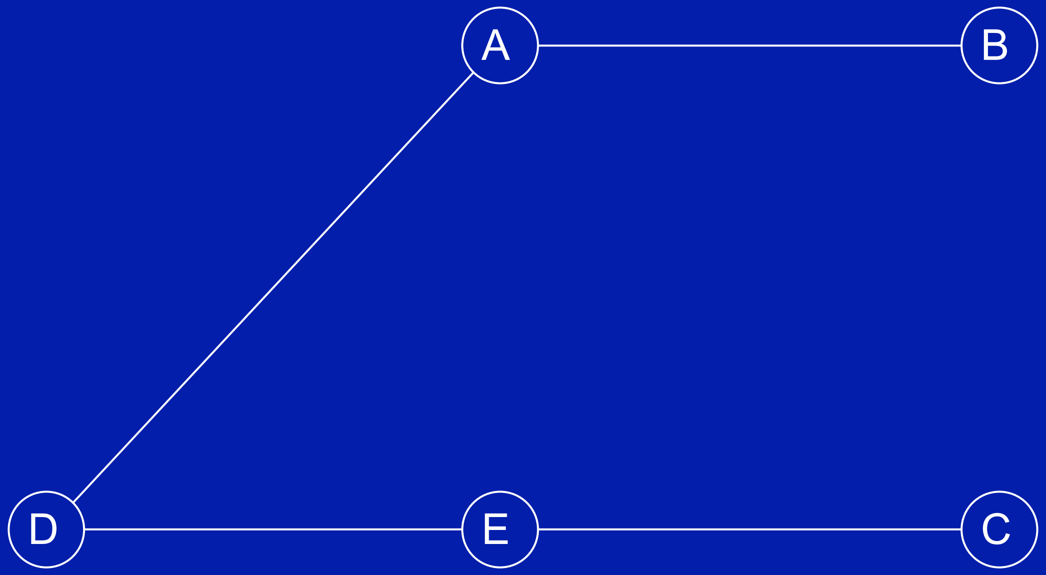
G = <**V**,**E**>

V = vertices = {A,B,C,D,E,F,G,H,I,J,K,L}

E = edges = {(A,B),(C,B),(D,C),(D,I),(E,D),(F,D),(F,E),(G,A),(G,F),(G,H),(G,K),(H,A),(I,C),(I,F),(I,L),(J,A),(J,B),(K,F),(K,J),(K,L),(L,C),(L,J)}

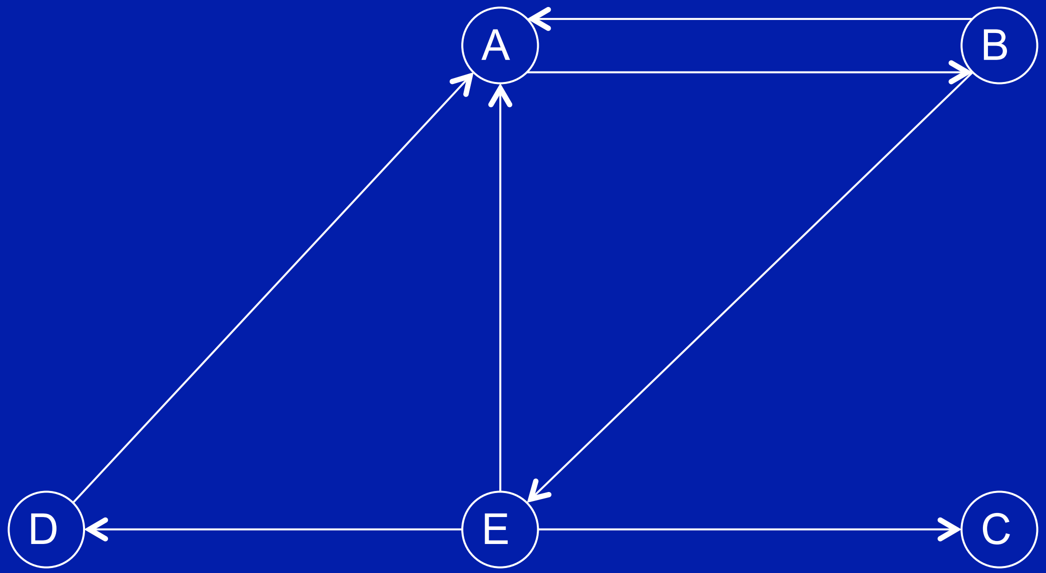
Another example

$V = \{A, B, C, D, E\}$
$E = \{\{A, B\}, \{A, D\}, \{C, E\}, \{D, E\}\}$



Another example (directed)

V = {A, B, C, D, E}
E = {{A, B}, {B, A}, {B, E}, {D, A},{E, A},{E, D},{E, C}}



Some definitions

An edge *e* between vertices *u* and *v* is said to be incident on *u* and *v*. Edges can have weights associated with them.

A vertex is adjacent to another if there is a single edge connecting them.

A path is a sequence of vertices in which each successive vertex is adjacent to its predecessor. A path can be directed or undirected.

If the first and last vertices are the same in a path, it is called a cycle.

A vertex that connects to itself is called a loop.

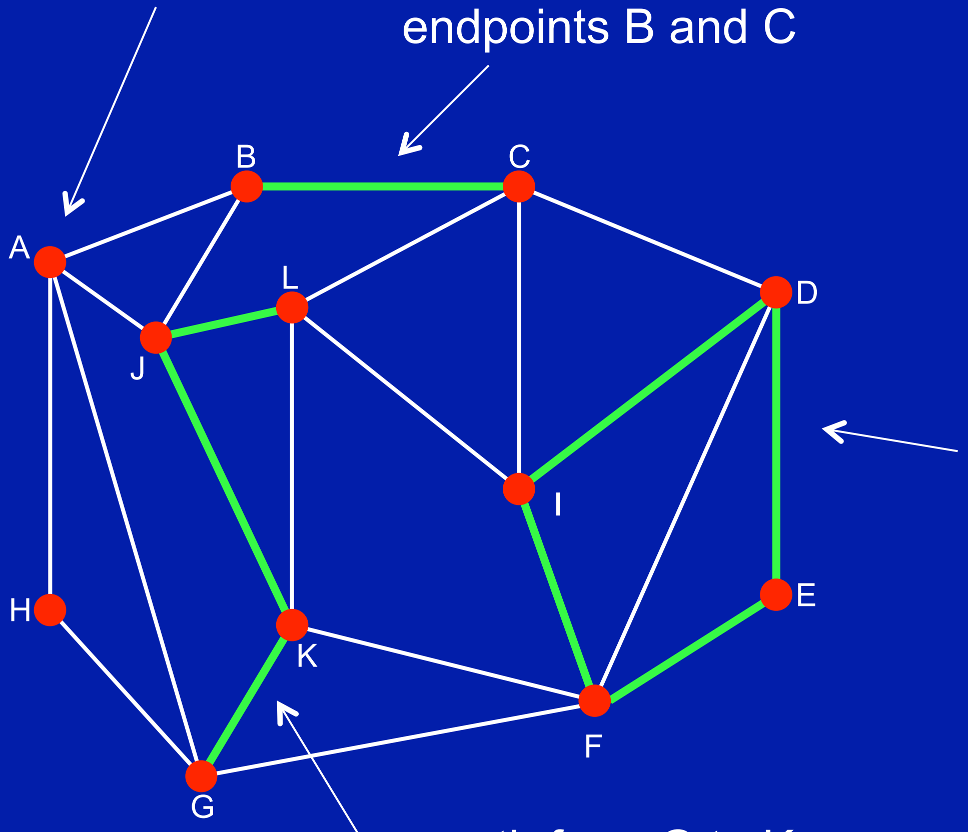
Graph anatomy

a vertex

an edge with
endpoints B and C

a cycle from D to E
to F to I to D

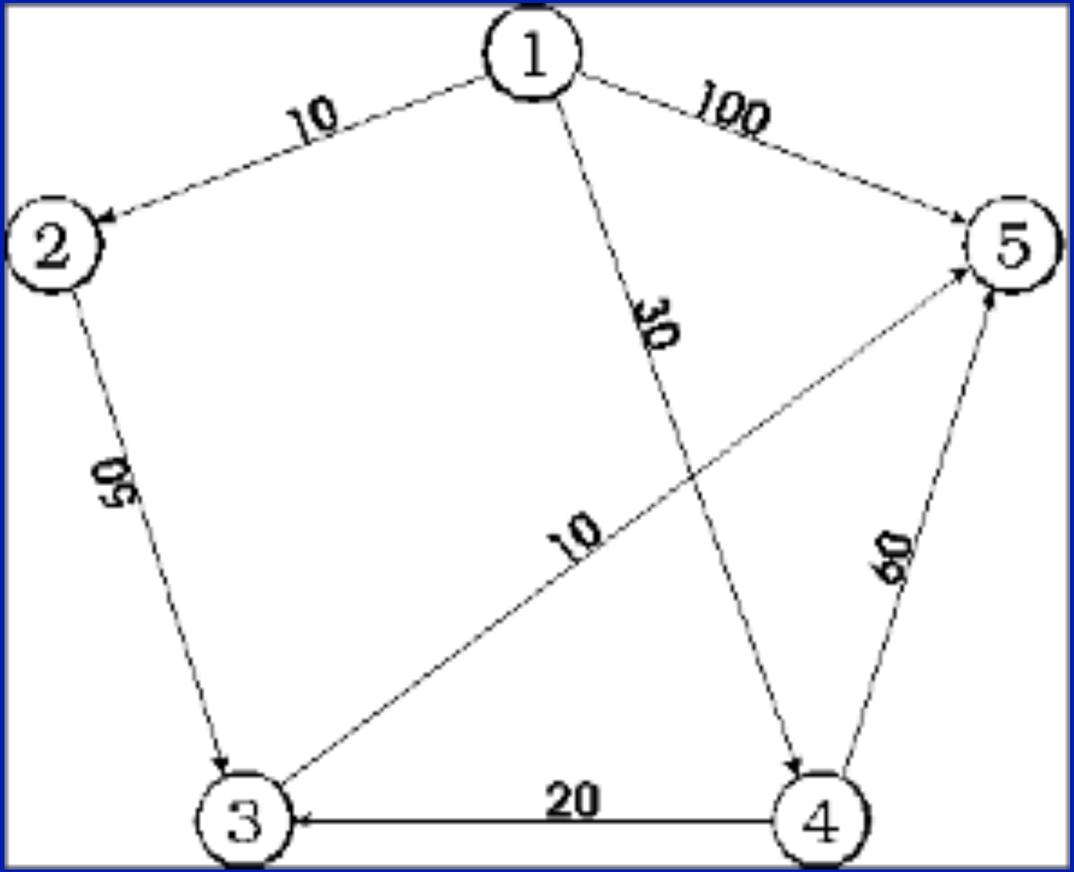
a path from G to K
to J to L



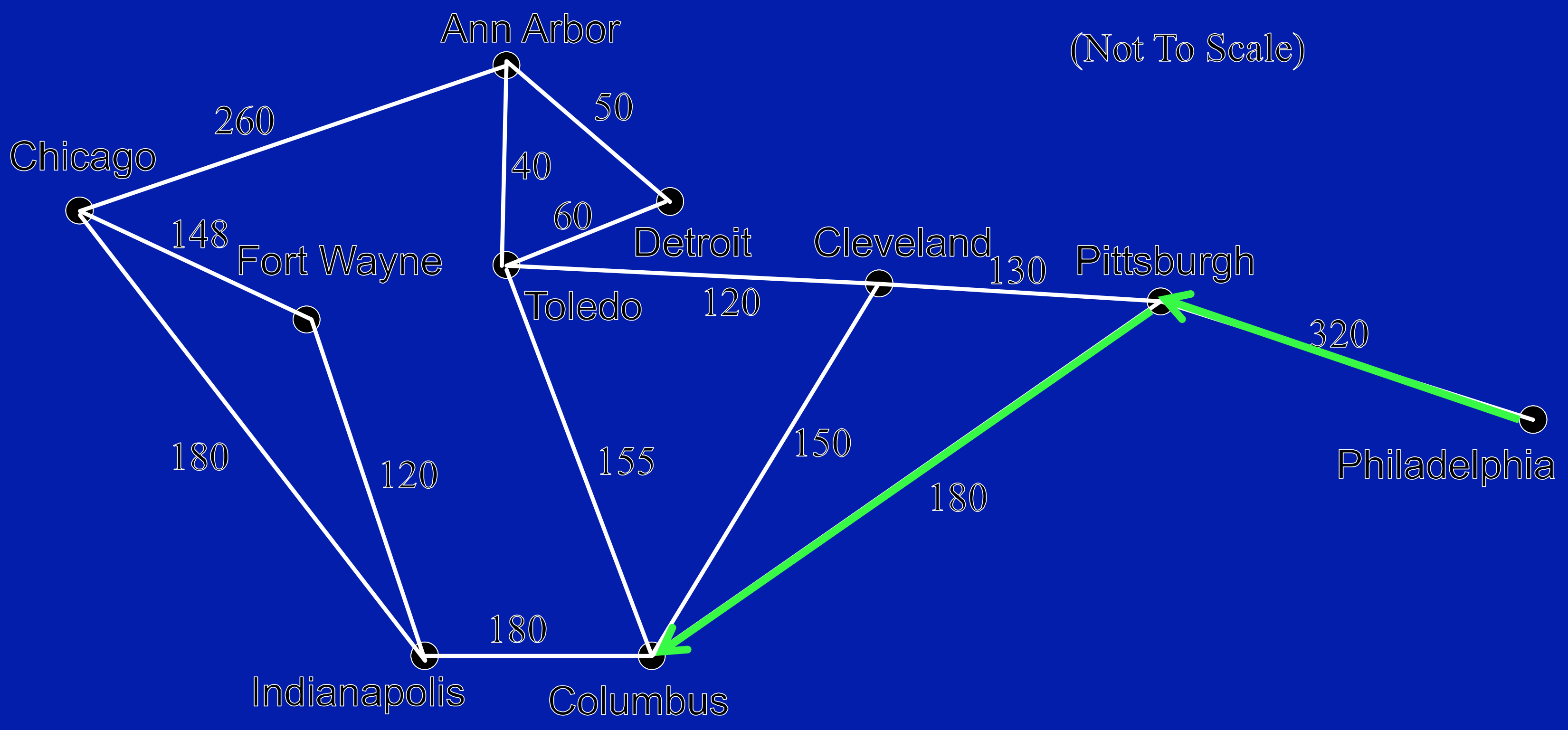
Weighted graph

- A weighted graph is a graph in which a value is associated with each of the edges.
- Weighted graphs may be either directed or undirected.

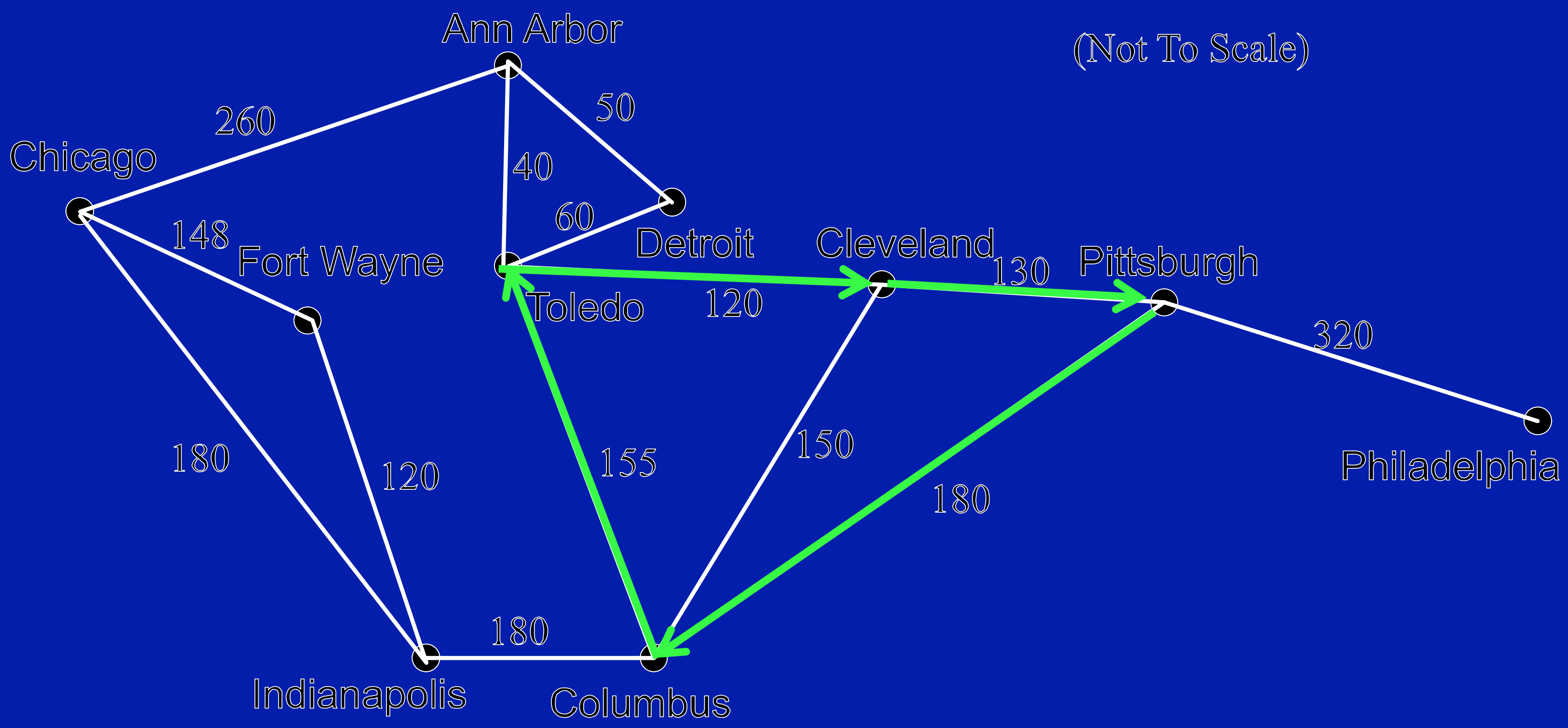
Example of directed weighted graph



Example of a path



Example of a cycle



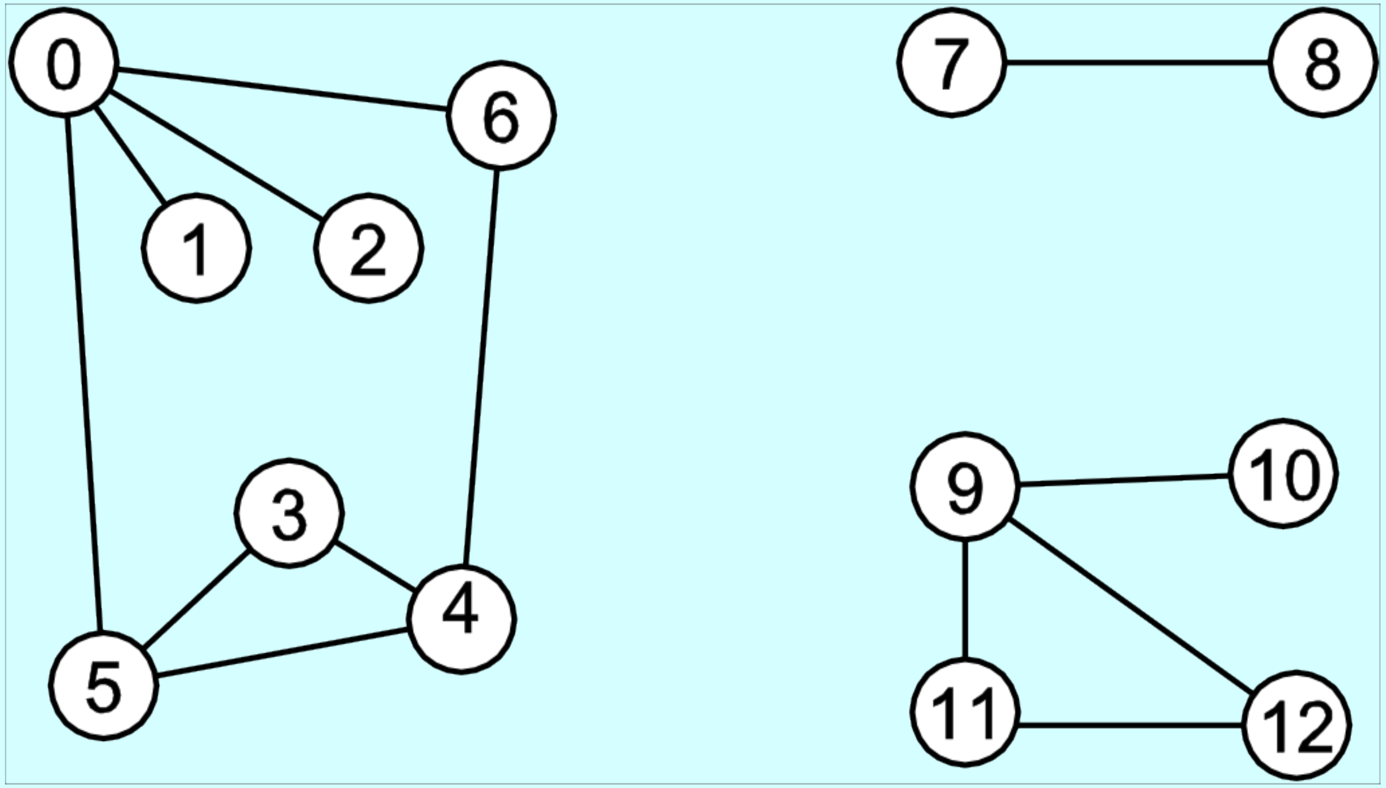
Connected and disconnected graphs

- If there exists a path from every vertex to every other vertex a graph is said to be *connected*.
- Otherwise a graph is *disconnected*, consisting of a series of components.
- The "pieces" themselves are called connected components or connected subgraphs.

In general:

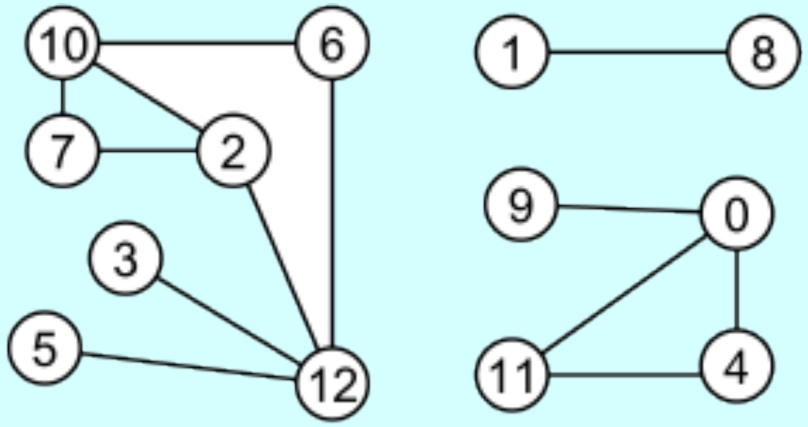
- A **subgraph** of a graph G=(V,E) is a graph H=(W,F) where $W \subseteq V$ and $F \subseteq E$. (Note about the trivial case: a subgraph can be unconnected (i.e., a single vertex).

Disconnected graph



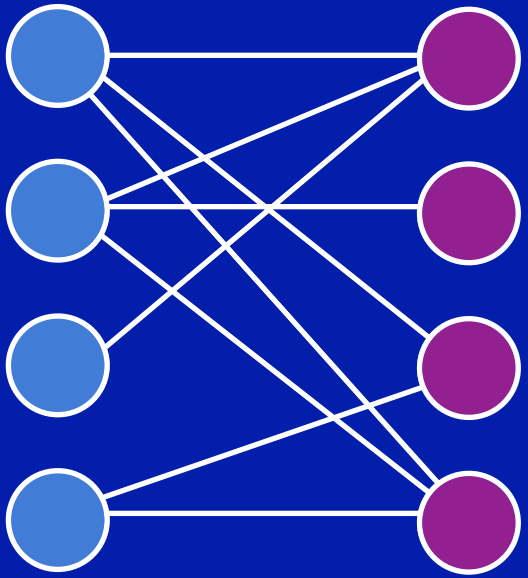
Graph isomorphism

The numbering of the vertices, and their physical arrangement are not important. The following is the same graph as the previous slide.



Bipartite graphs

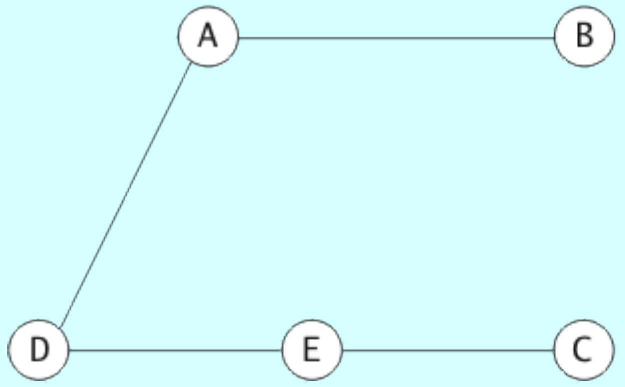
A graph whose nodes can be split into two pairwise disjoint sets is said to be bipartite.

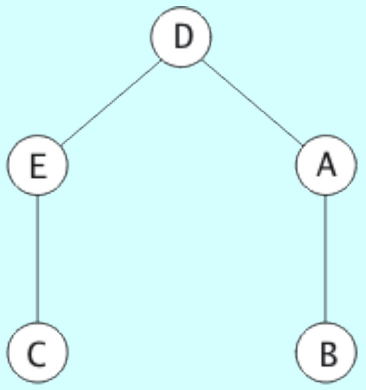


All edges connect one set to the other – no intra-set edges

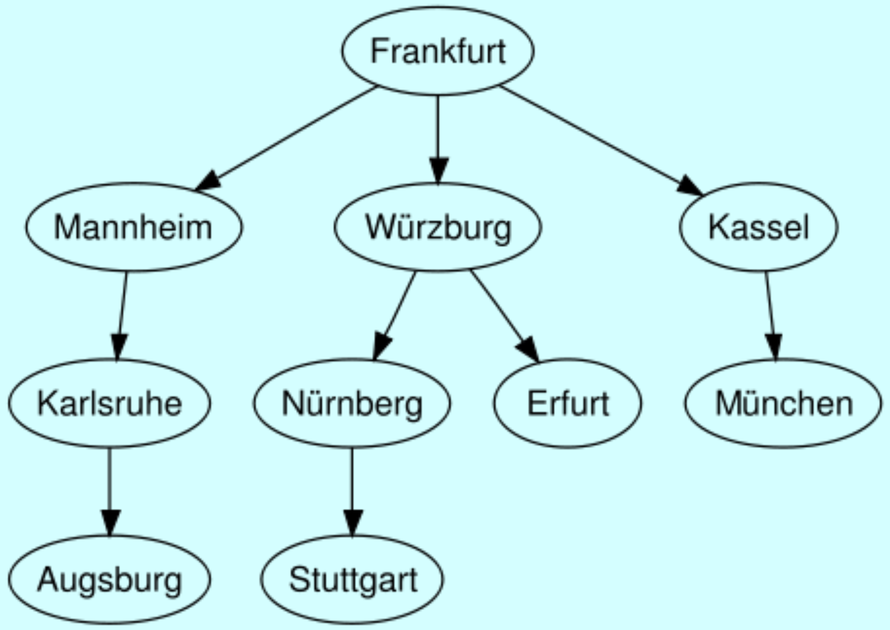
Relationship between graphs and trees

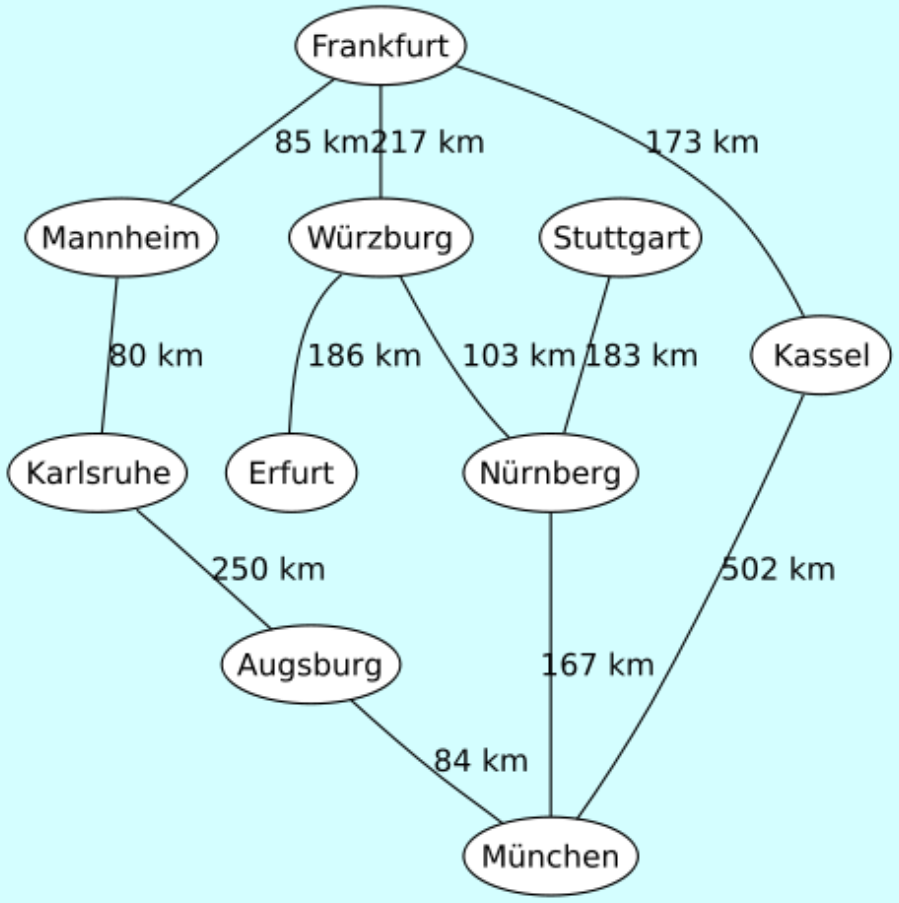
- A tree is a special case of a graph.
- Any connected graph which does not contain cycles can be considered a tree.
- Any node can be chosen to be the root.





Separate tree and graph examples





Abstract representation of a graph

- We’ve been looking at abstract representations; now we’ll see how we might implement them

- Operations on a graph (the things we might want to do):
  - Create a graph of n vertices.
  - Insert an edge
  - Remove an edge
  - Query the existence of an edge
  - Iterate over the vertices adjacent to a given vertex.

Concrete representation of a graph

Two methods of representing a graph are popular:

- Adjacency Matrix (edge based)
  - A $|V| \times |V|$ matrix with 1 (or the weight) (or true) for an edge and 0 (or infinity) (or false) for not-an-edge
- Adjacency List (vertex based)
  - A vector (array) of lists, one for each vertex. Each list has the adjacent vertices.

Concrete representation of a graph

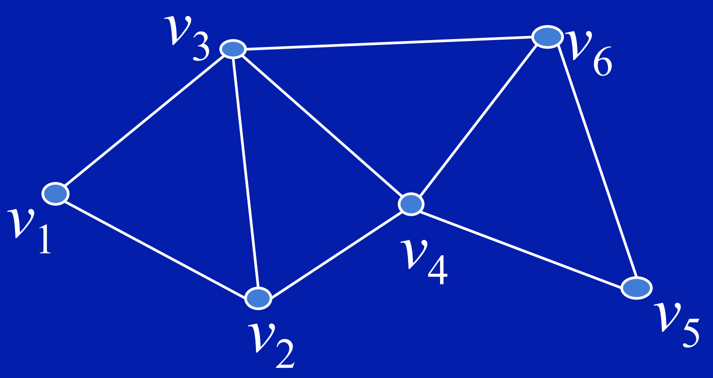
E.g. A graph with *n* vertices $v_1, v_2, \ldots v_n$.

The entry in row *i* and column *j* of *A* is the number of edges (v_i, v_j) in the graph.

The adjacency matrix of a simple graph contains only 0's and 1's.

The adjacency matrix of an undirected graph is symmetric.

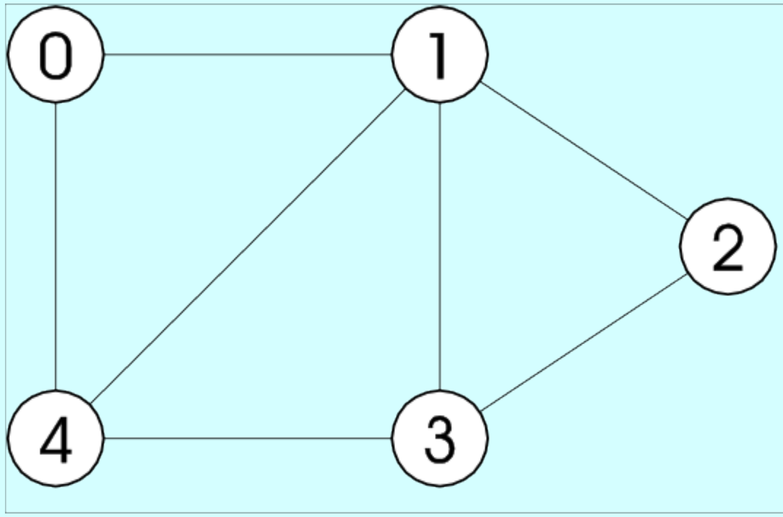
For graphs with few edges, the adjacency matrix is *sparse* (i.e., contains mostly zeros).

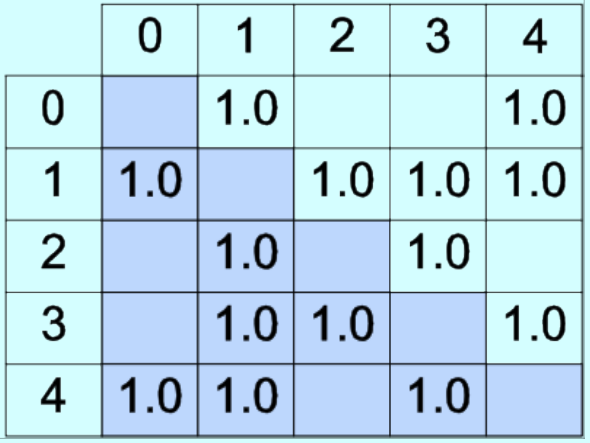


	v_1	v_2	v_3	v_4	v_5	v_6
v_1		1	1			
v_2	1		1	1		
v_3	1	1		1		1
v_4		1	1		1	1
v_5				1		1
v_6			1	1	1	

Adjacency Matrix

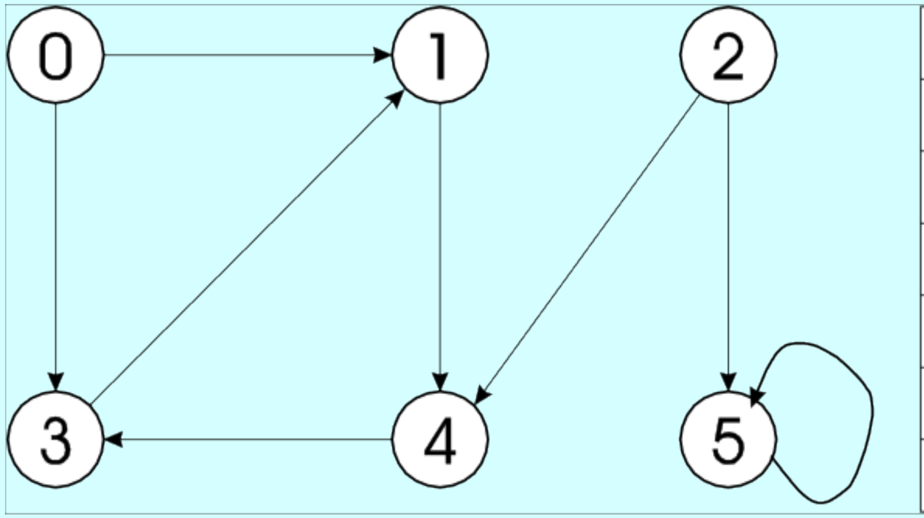
Another adjacency matrix





	0	1	2	3	4
0		1.0			1.0
1	1.0		1.0	1.0	1.0
2		1.0		1.0	
3		1.0	1.0		1.0
4	1.0	1.0		1.0	

Adjacency matrix (directed graph)



	0	1	2	3	4	5
0		1.0		1.0		
1					1.0	
2					1.0	1.0
3		1.0				
4				1.0		
5						1.0

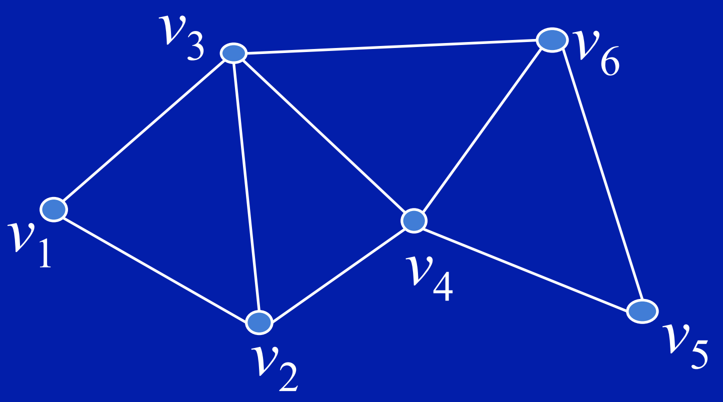
Concrete representation of a graph

Two methods of representing a graph are popular:

- Adjacency Matrix (edge based)
  - A $|V| \times |V|$ matrix with 1 (or the weight) (or true) for an edge and 0 (or infinity) (or false) for not-an-edge
- Adjacency List (vertex based)
  - A vector (array) of lists, one for each vertex. Each list has the adjacent vertices.

Adjacency list

E.g. A graph with *n* vertices $v_1, v_2, \ldots v_n$.

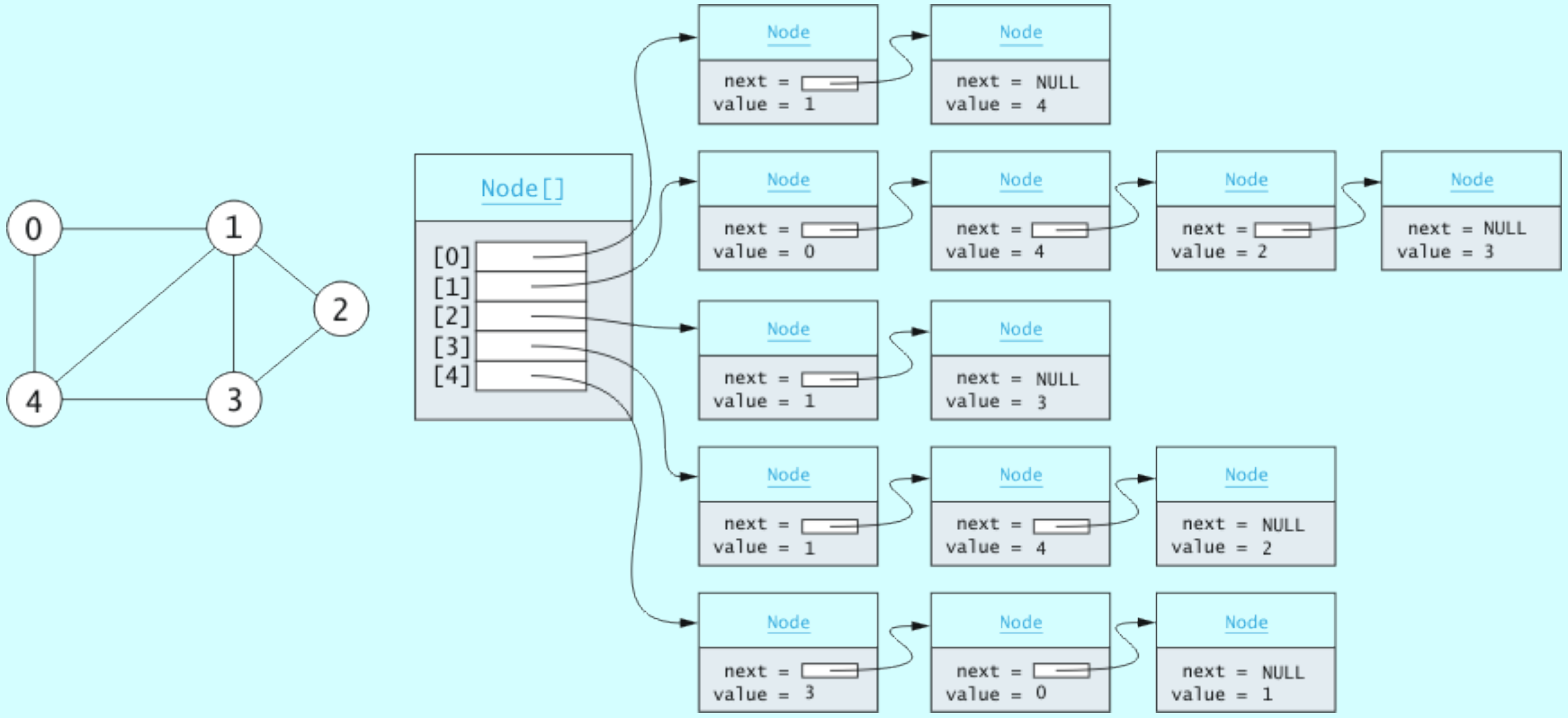


The adjacency list is never sparse (even for a graph with few edges). In other words, the space is well-used.

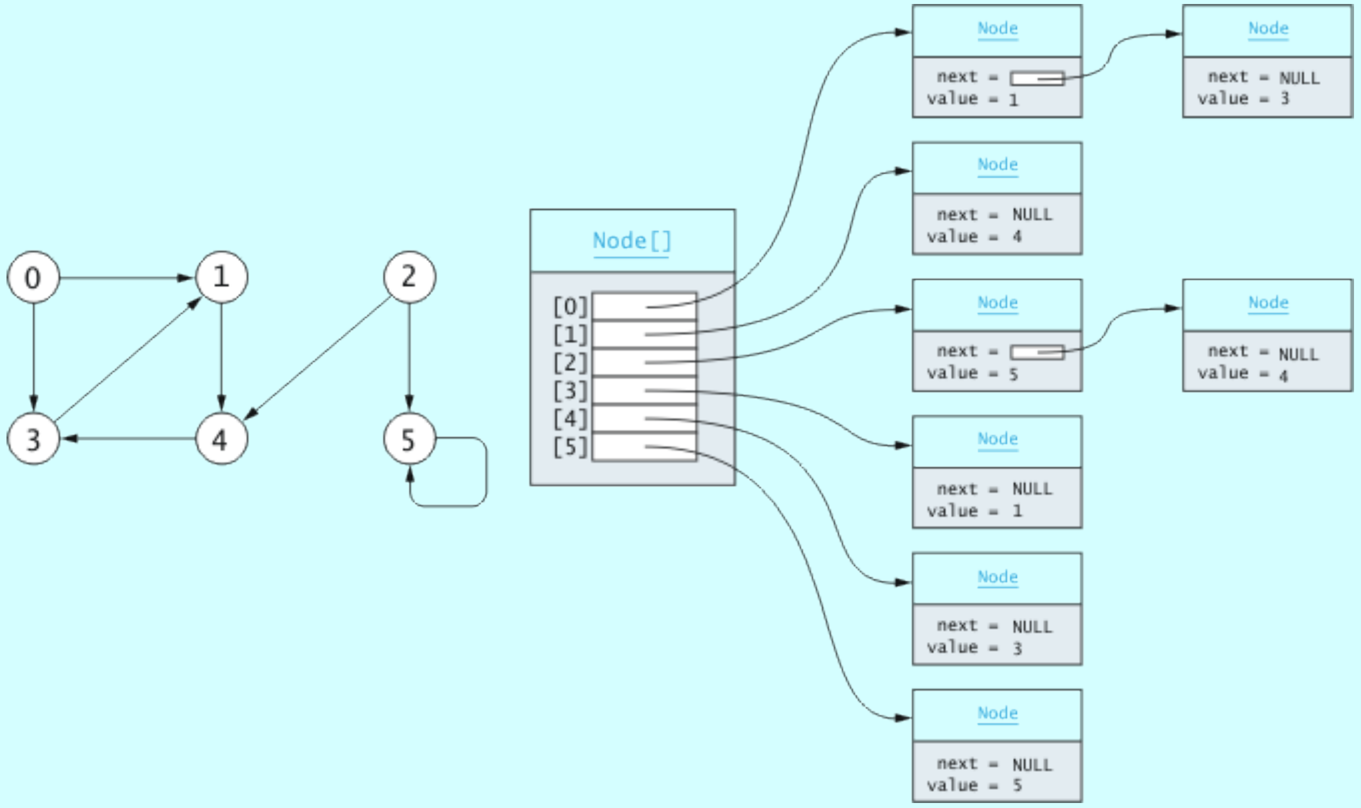
v_1	v_2, v_3
v_2	v_1, v_3, v_4
v_3	v_1, v_2, v_4, v_6
v_4	v_2, v_3, v_5, v_6
v_5	v_4, v_6
v_6	v_3, v_4, v_5

Adjacency List

Adjacency list representation



Adjacency list (directed graph)



Efficiency comparisons

Which one is better?

Efficiency comparisons

Which one is better? It depends on the algorithm and the density of the graph. Graph density is the ratio of (the number of edges) $|E|$ to (the number of vertices) $|V|^{2.}$

A dense graph is one in which $|E|$ is close to but less than $|V|^2$. A sparse graph is one in which $|E|$ is much less than $|V|^2$.

Efficiency comparisons

In general, if the graph is dense, the adjacency matrix is better, and if the graph is sparse, the adjacency list is better. Usually, our graphs are sparse, but not always.

The math is in your textbook, but intuitively, a sparse graph will lead to a sparse matrix (few 1s, lots of 0s). The algorithm has to process every location in the matrix, whether it indicates an edge there or not, and that has an undesirable impact on processing time. Those entries don’t show up in the list, however, so they will have no effect on processing time in the adjacency list implementation.

Graph traversal

Graph algorithms typically involve visiting each vertex in systematic order. The two most common traversal algorithms are breadth-first and depth-first.

Although these are traversals, the traversals are usually employed to find something in the graph (e.g., a vertex/ node, or more likely a path to that vertex/node), so they’re more commonly called breadth-first search and depth-first search.

Breadth-first

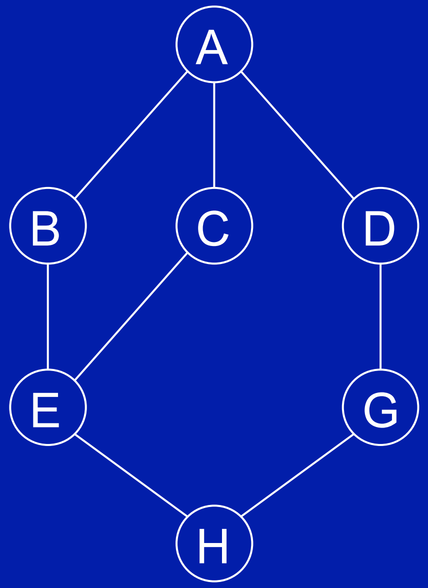
- Starting at a source vertex s
- Systematically explore the edges to "discover" every vertex reachable from s.
- Produces a "breadth-first tree"
  - Root of s
  - Contains all vertices reachable from *s*
  - Path from *s* to *v* is the shortest path

Breadth-first algorithm

```
Take a start vertex, mark it identified (colour it green), and place it
into a queue.

While the queue is not empty
    Take a vertex, u, out of the queue (Begin visiting u)
    For all vertices v, adjacent to u,
        If v has not been identified or visited
            Mark it identified (colour it green)
            Place it into the queue
            (Add edge u, v to the Breadth First Search Tree)
    We are now done visiting u (colour it orange)
```

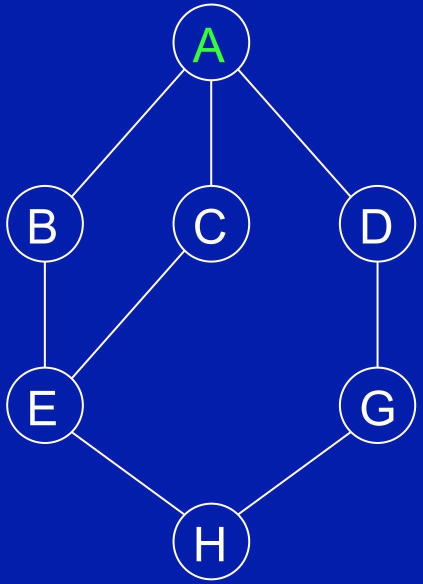
Breadth-first algorithm



Queue: [ ]

Processing:

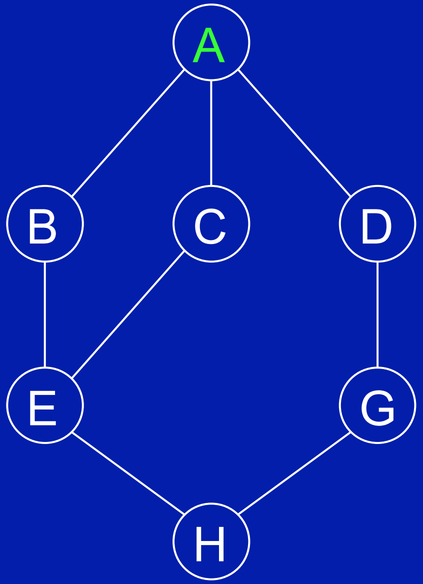
Breadth-first algorithm



Queue: [A]

Processing:

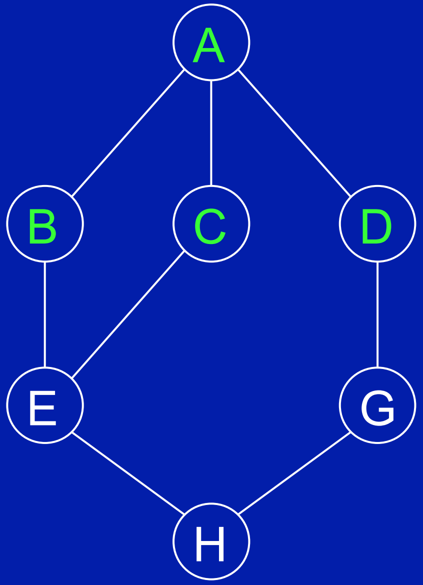
Breadth-first algorithm



Queue: [ ]

Processing: A

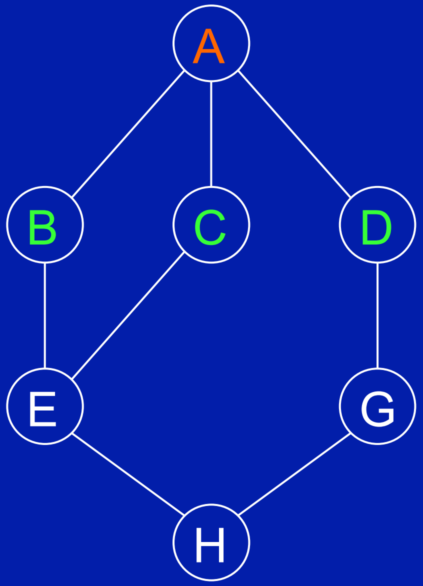
Breadth-first algorithm



Queue: [B C D]

Processing: A

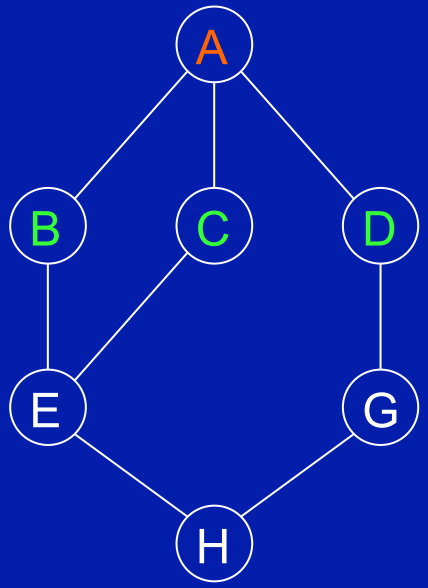
Breadth-first algorithm



Queue: [B C D]

Processing:

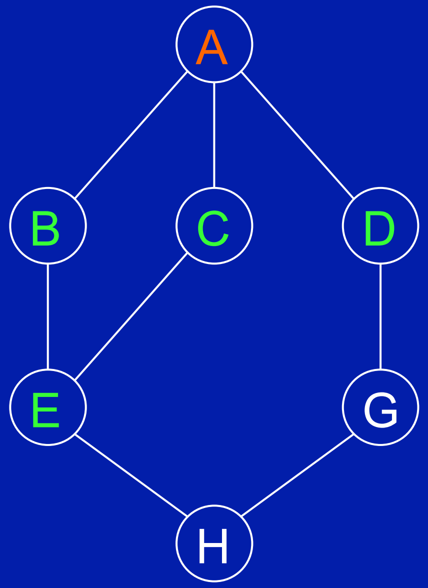
Breadth-first algorithm



Queue: [C D]

Processing: B

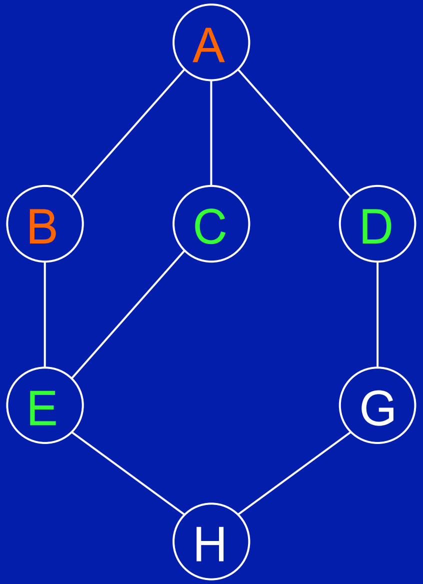
Breadth-first algorithm



Queue: [C D E]

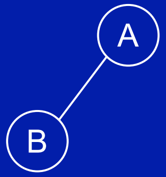
Processing: B

Breadth-first algorithm

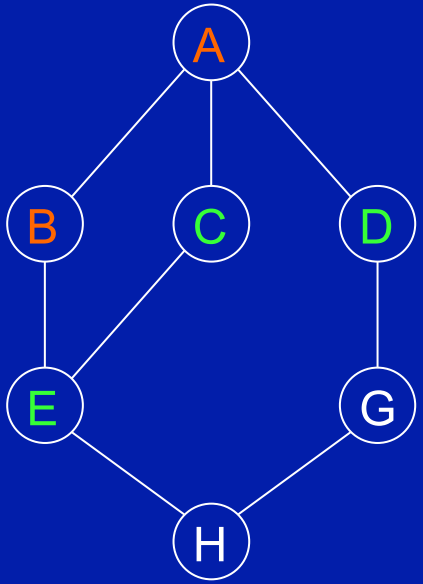


Queue: [C D E]

Processing:

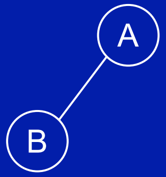


Breadth-first algorithm

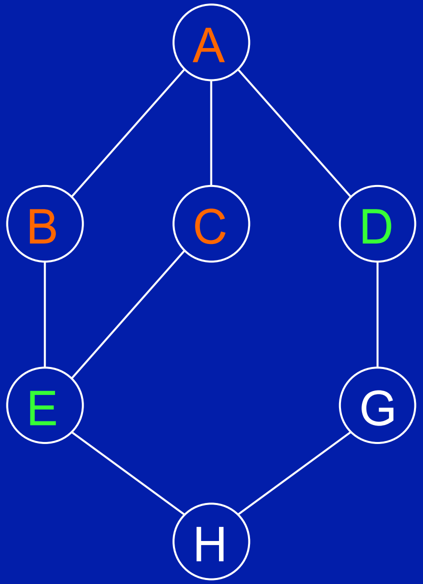


Queue: [D E]

Processing: C

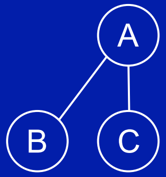


Breadth-first algorithm

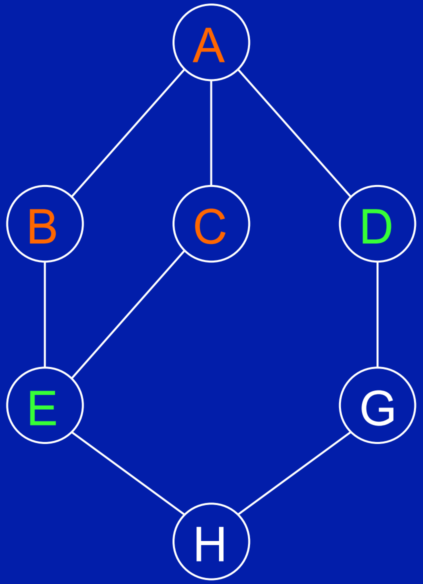


Queue: [D E]

Processing:

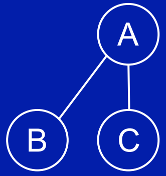


Breadth-first algorithm

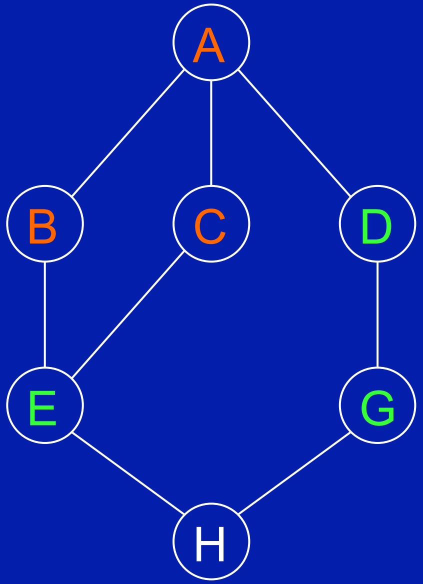


Queue: [E]

Processing: D

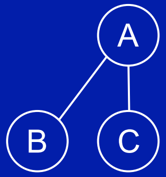


Breadth-first algorithm

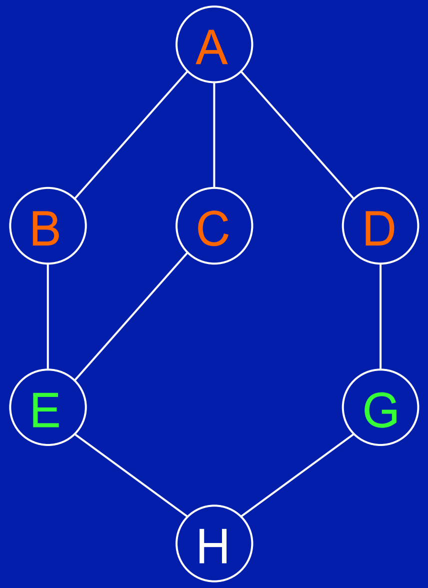


Queue: [E G]

Processing: D

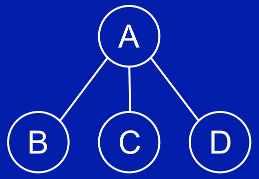


Breadth-first algorithm

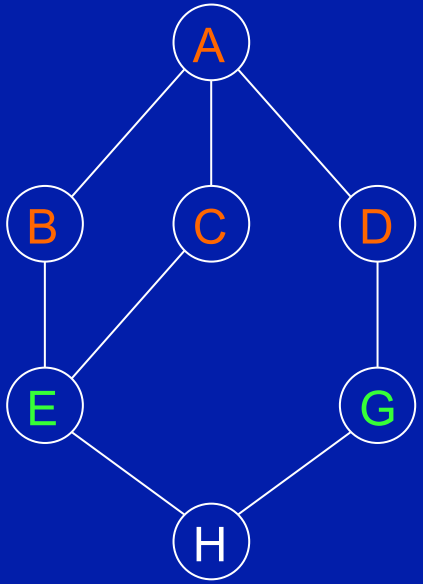


Queue: [E G]

Processing:

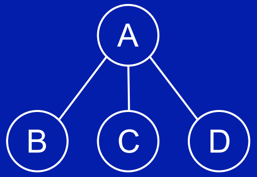


Breadth-first algorithm

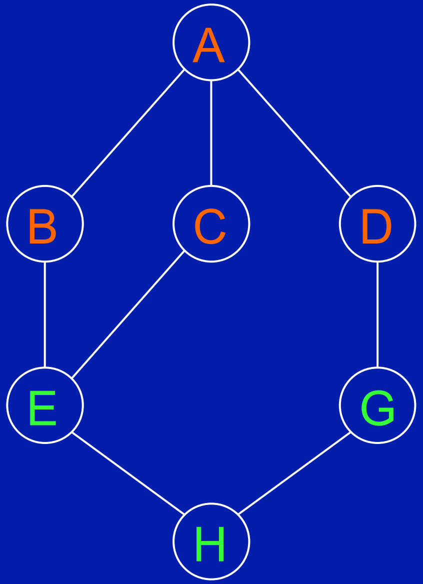


Queue: [G]

Processing: E

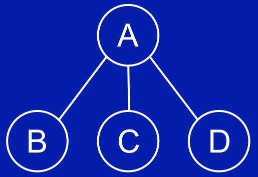


Breadth-first algorithm

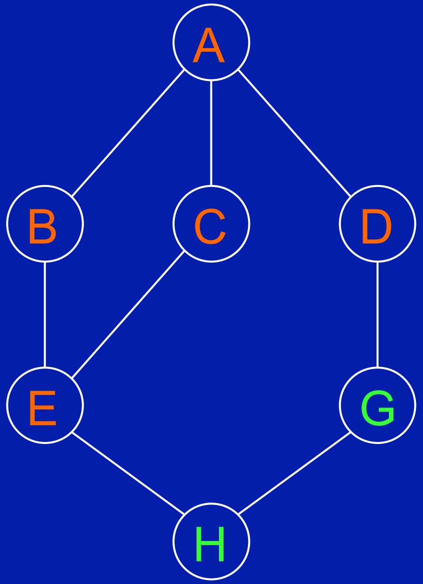


Queue: [G H]

Processing: E

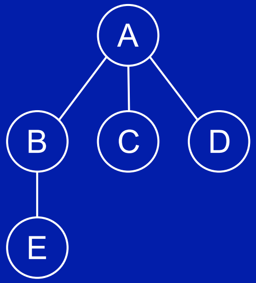


Breadth-first algorithm

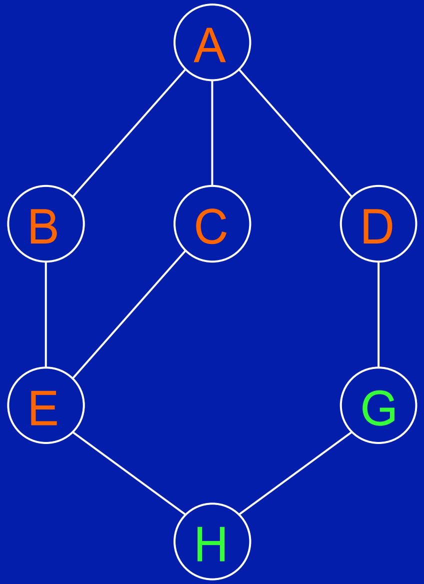


Queue: [G H]

Processing:

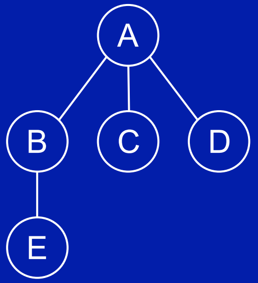


Breadth-first algorithm

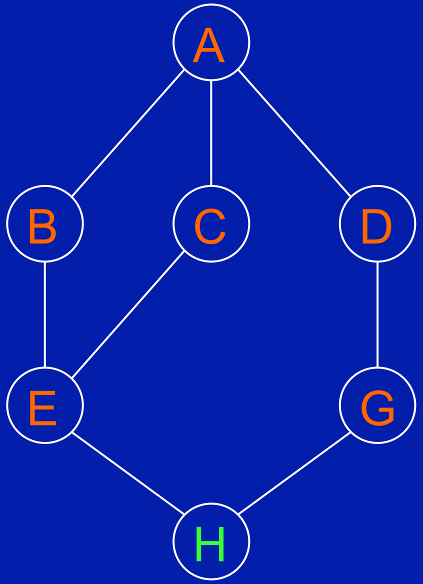


Queue: [H]

Processing: G

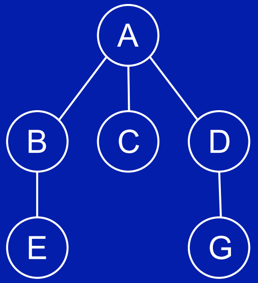


Breadth-first algorithm

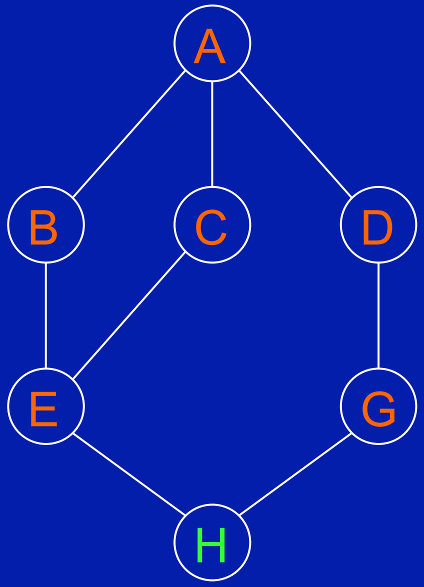


Queue: [H]

Processing:

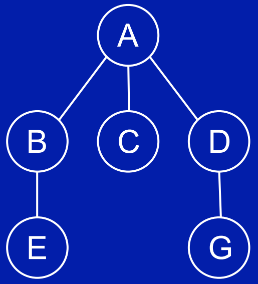


Breadth-first algorithm

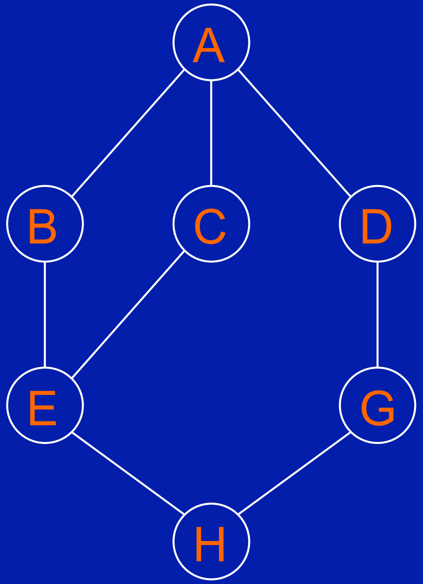


Queue: [ ]

Processing: H

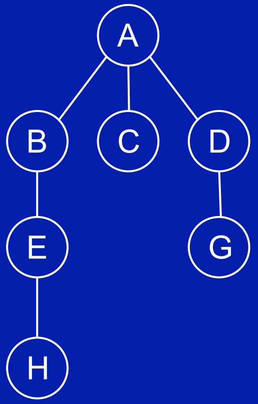


Breadth-first algorithm

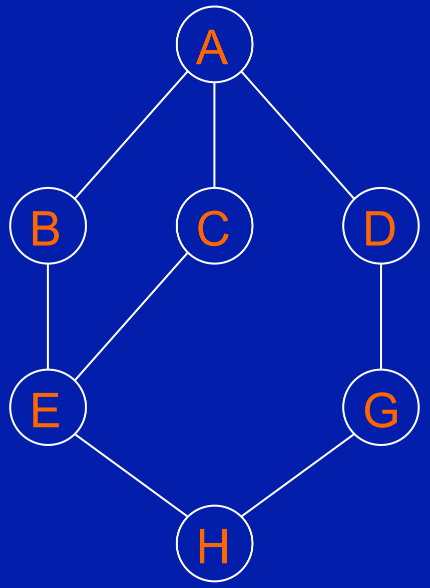


Queue: [ ]

Processing:

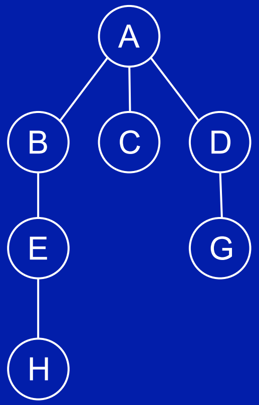


Breadth-first algorithm



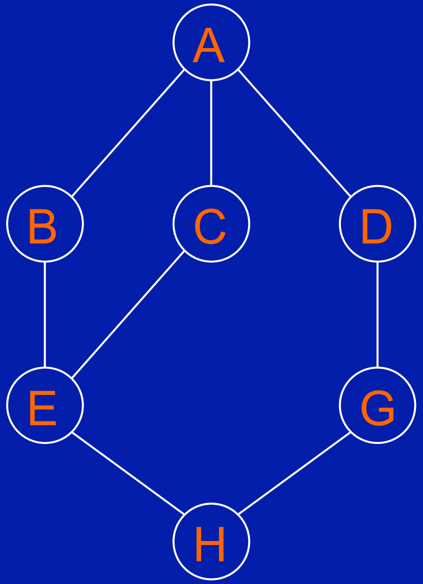
Queue: [ ]

Processing:



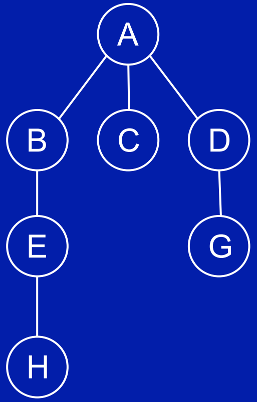
Assuming all edges have equal cost (e.g., distance), breadth-first will always find shortest path from start to goal.
Breadth-first can be difficult to program due to explicit queue management (compared to depth-first).

Breadth-first algorithm



Queue: [ ]

Processing:



Yes, this would have been a way to solve the maze problem for Programming Assignment 1. You may have figured it out on your own.

Depth-first

- Starting at a source vertex s
- Follow a simple path discovering new vertices until you cannot find a new vertex
- Back-up until you can start finding new vertices

Depth-first algorithm

```
Start at vertex $u$. Mark it visited (colour green) and put it in
discovery order list.

For each vertex $v$ adjacent to u
    If $v$ has not been visited
        Insert $u$, $v$ into DFS tree
        Recursively apply this algorithm starting at v

Mark $u$ finished (color it orange) and put it in finish order list.
```

Depth-first algorithm

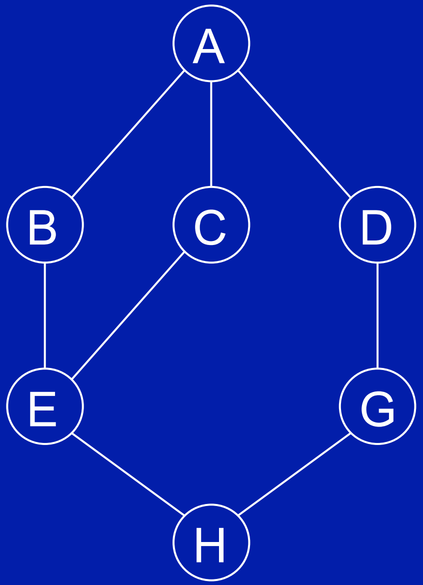
Start at vertex *u*. Mark it visited (colour green) and put it in discovery order list.

```
For each vertex v adjacent to u
    If v has not been visited
        Insert u, v into DFS tree
        Recursively apply this algorithm starting at v
```

Mark *u* finished (color it orange) and put it in finish order list.

This demo will ignore some of these details. We'll just do what breadth-first did, but we'll use a stack instead of a queue.

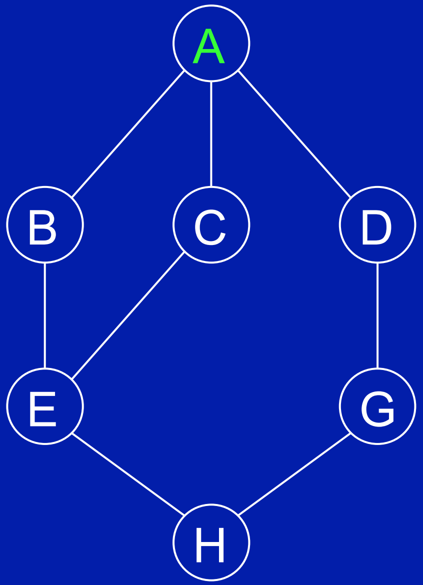
Depth-first algorithm



Stack: [ ]

Processing:

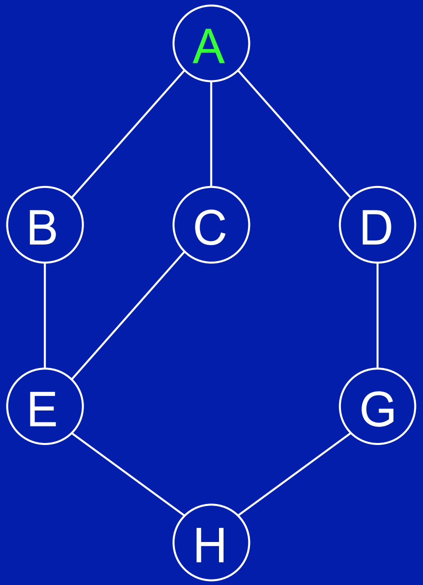
Depth-first algorithm



Stack: [A]

Processing:

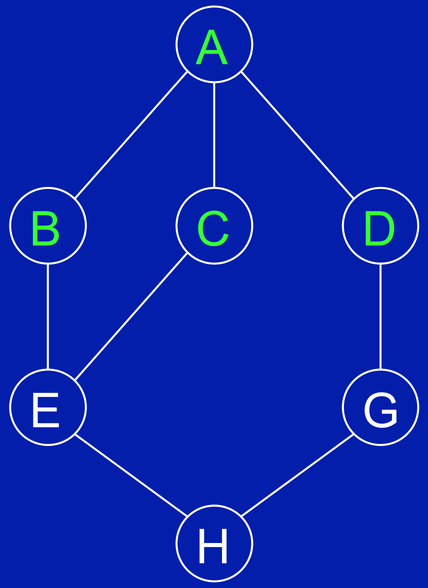
Depth-first algorithm



Stack: [ ]

Processing: A

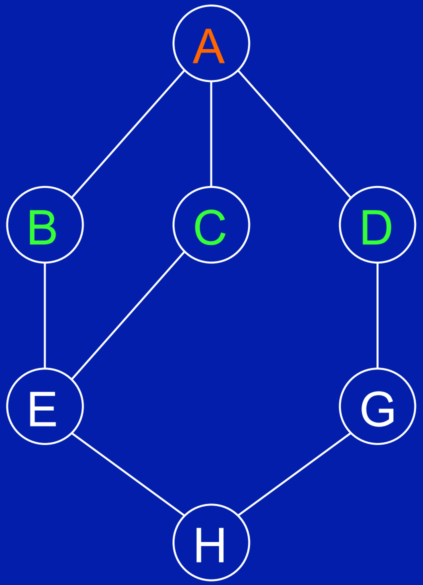
Depth-first algorithm



Stack: [B C D]

Processing: A

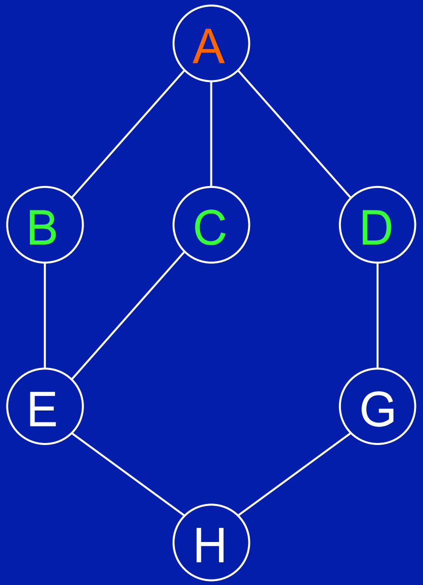
Depth-first algorithm



Stack: [B C D]

Processing:

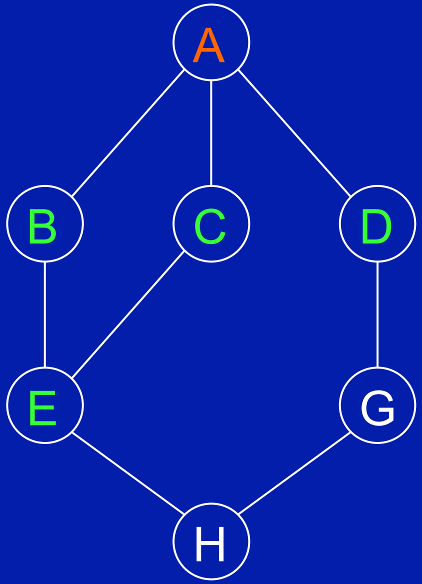
Depth-first algorithm



Stack: [C D]

Processing: B

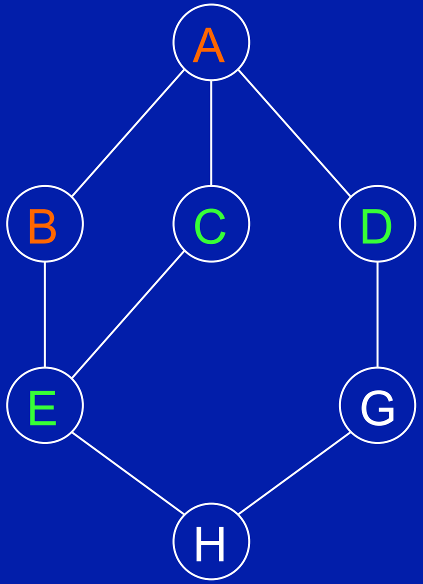
Depth-first algorithm



Stack: [E C D]

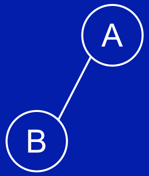
Processing: B

Depth-first algorithm

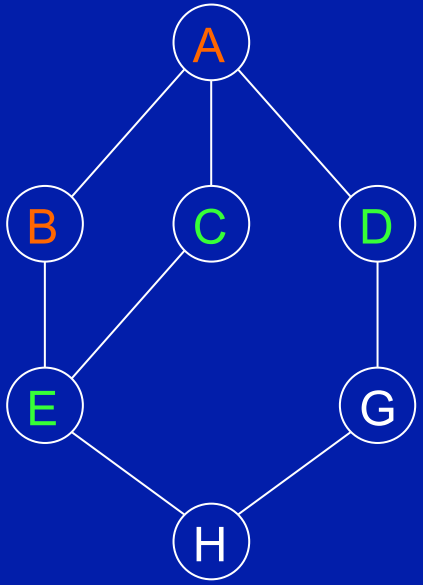


Stack: [E C D]

Processing:

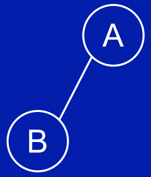


Depth-first algorithm

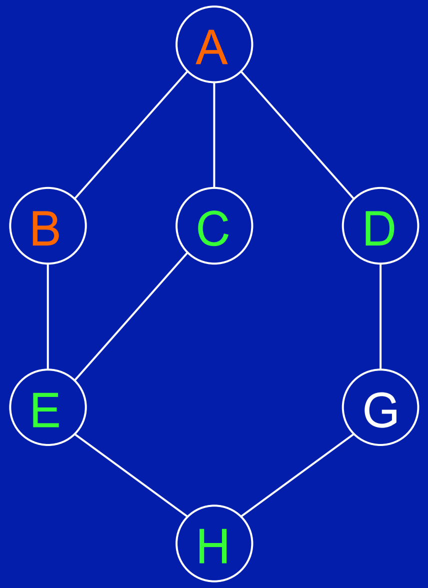


Stack: [C D]

Processing: E

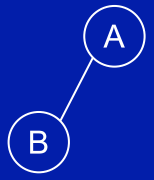


Depth-first algorithm

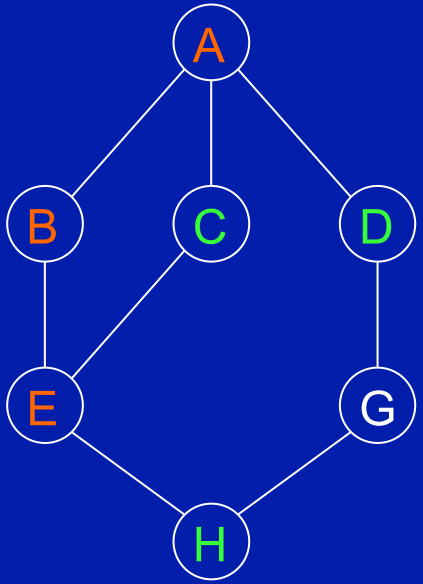


Stack: [H C D]

Processing: E

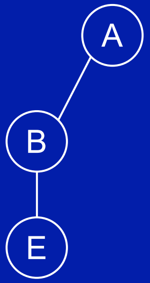


Depth-first algorithm

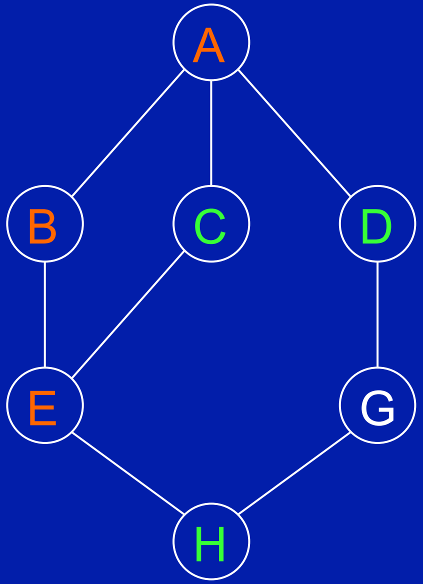


Stack: [H C D]

Processing:

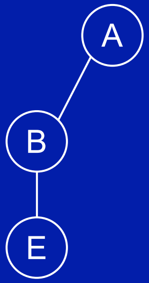


Depth-first algorithm

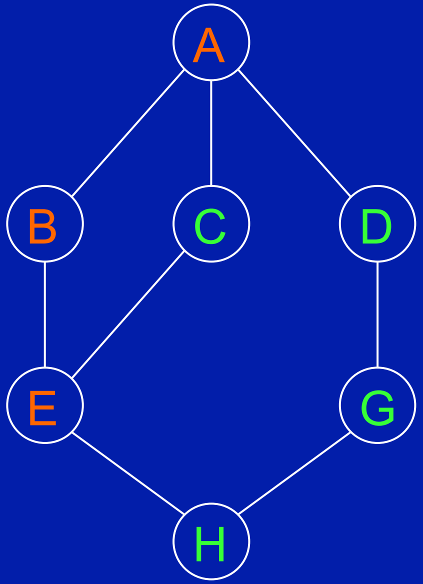


Stack: [C D]

Processing: H

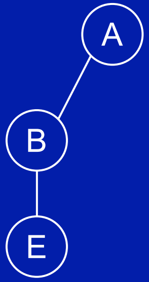


Depth-first algorithm

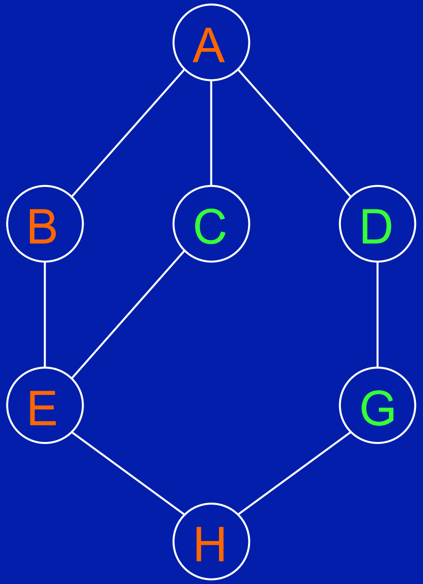


Stack: [G C D]

Processing: H

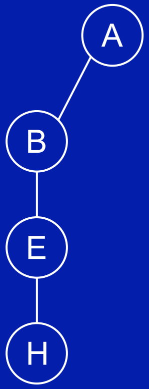


Depth-first algorithm

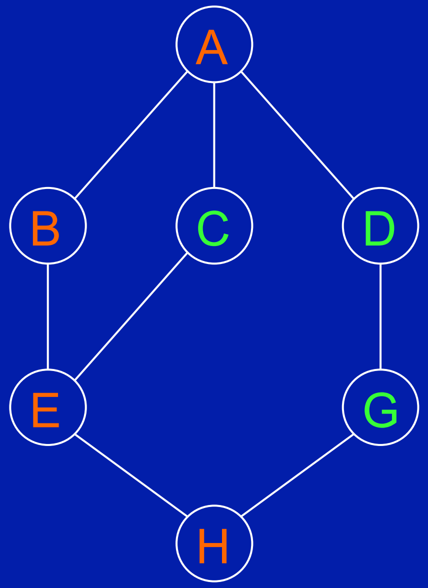


Stack: [G C D]

Processing:

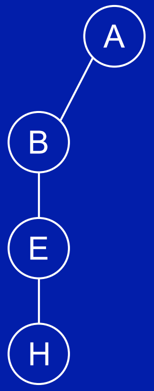


Depth-first algorithm

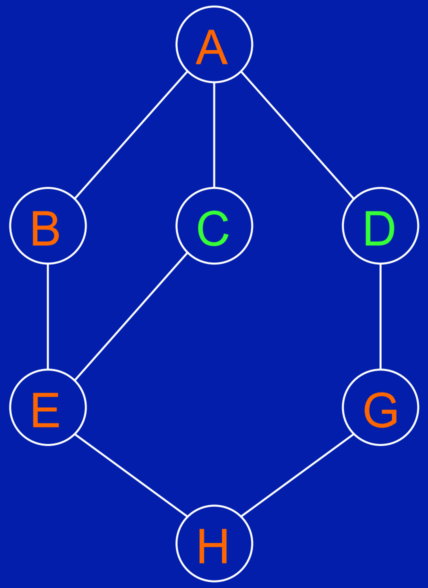


Stack: [C D]

Processing: G

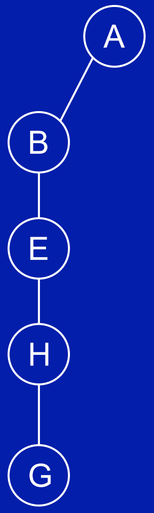


Depth-first algorithm

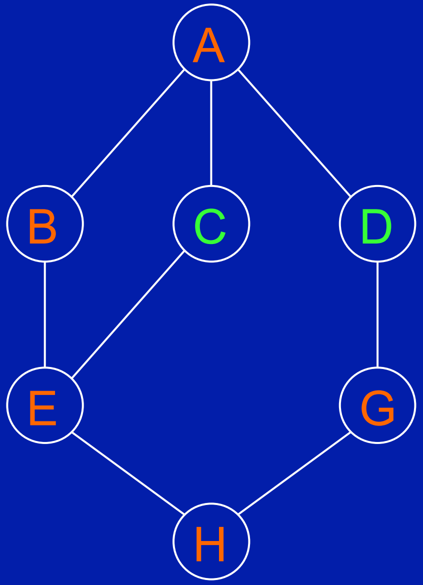


Stack: [C D]

Processing:

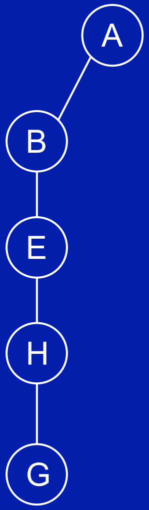


Depth-first algorithm

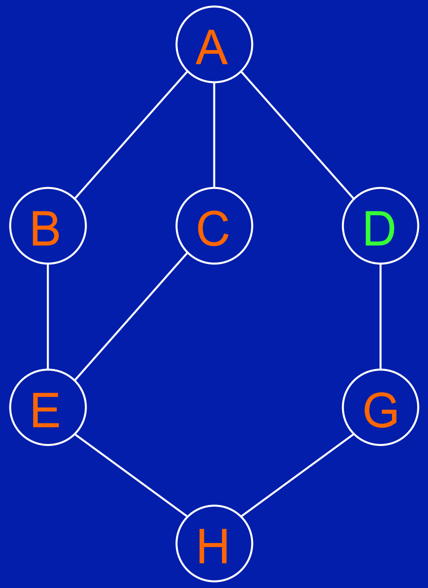


Stack: [D]

Processing: C

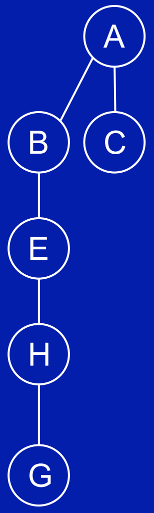


Depth-first algorithm

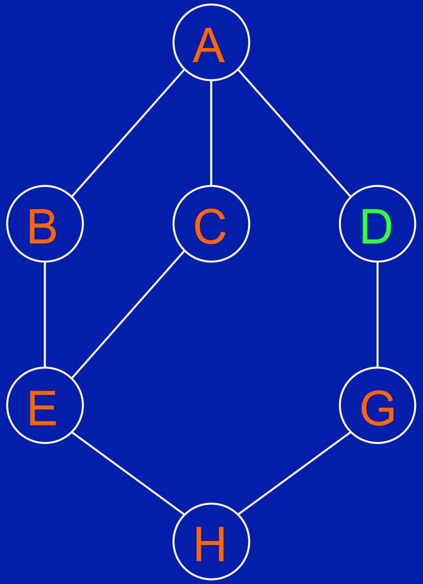


Stack: [D]

Processing:

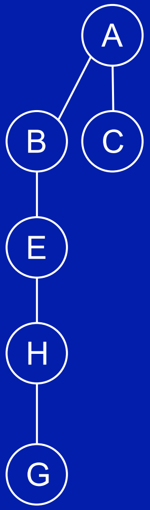


Depth-first algorithm

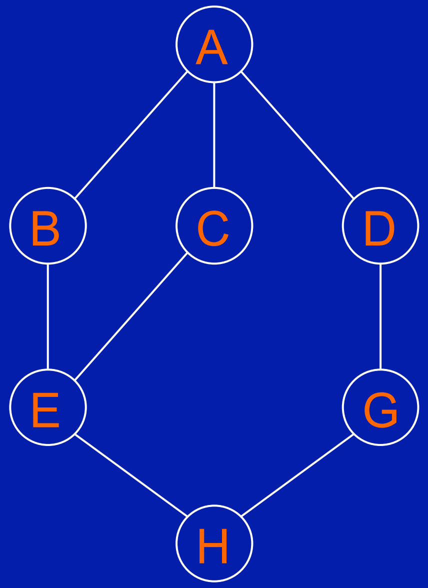


Stack: [ ]

Processing: D

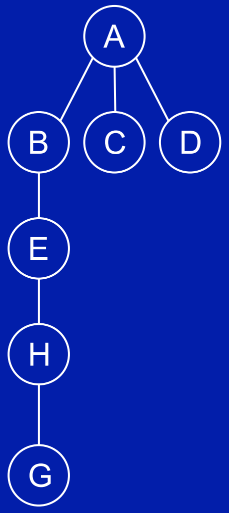


Depth-first algorithm

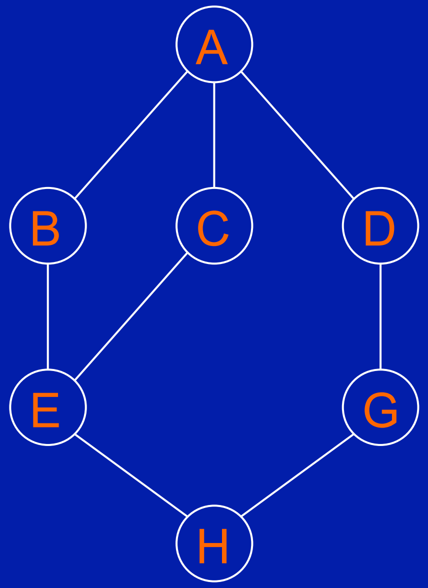


Stack: [ ]

Processing:

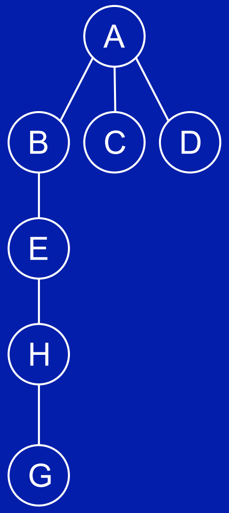


Depth-first algorithm



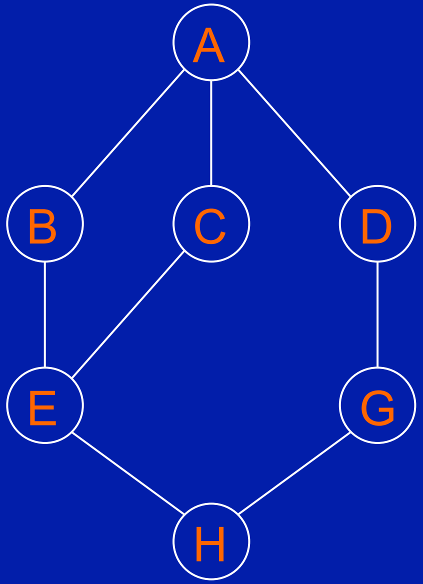
Stack: [ ]

Processing:



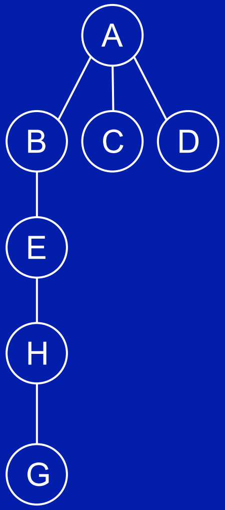
Yes, you could have approached the maze problem this way too.
Not to mention one of the midterm exam multiple choice questions.

Depth-first algorithm



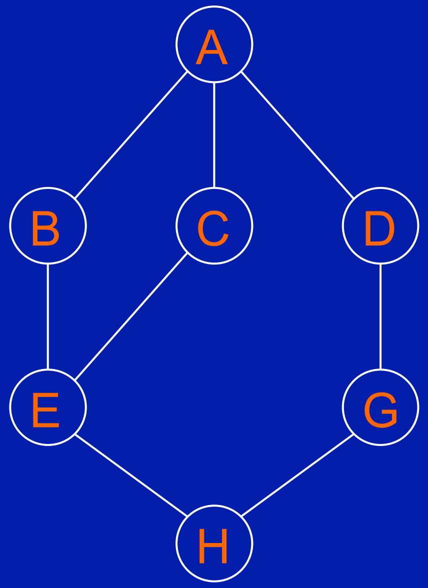
Stack: [ ]

Processing:



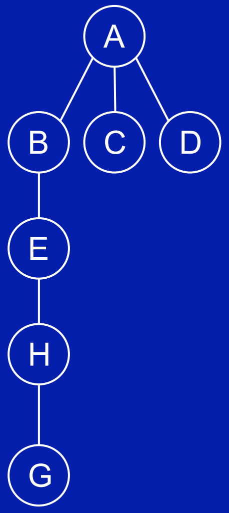
Depth-first will find what it’s looking for eventually, assuming the graph isn’t infinitely large. It tends to be easier to program than breadth-first, because you don’t need an explicit stack. Why?

Depth-first algorithm



Stack: [ ]

Processing:



Depth-first will find what it’s looking for eventually, assuming the graph isn’t infinitely large. It tends to be easier to program than breadth-first, because you don’t need an explicit stack. Why? You let the activation stack take care of those details for you.