Graphs

COMP 2210 – Dr. Hendrix

AUBURN

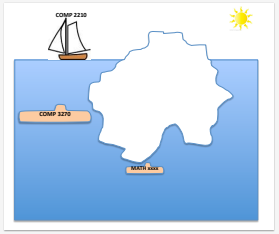
UNIVERSITY

SAMUEL GINN
COLLEGE OF ENGINEERING

Graphs

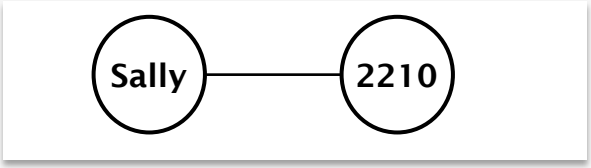
The most general data structure that we've talked about.

Another iceberg

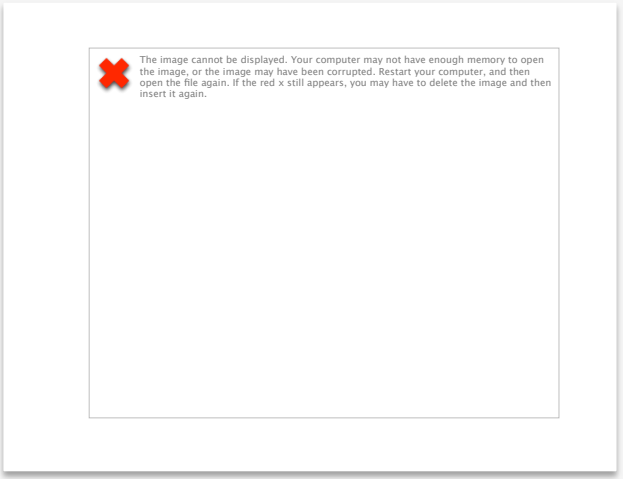


Represents pairwise relationships between objects.

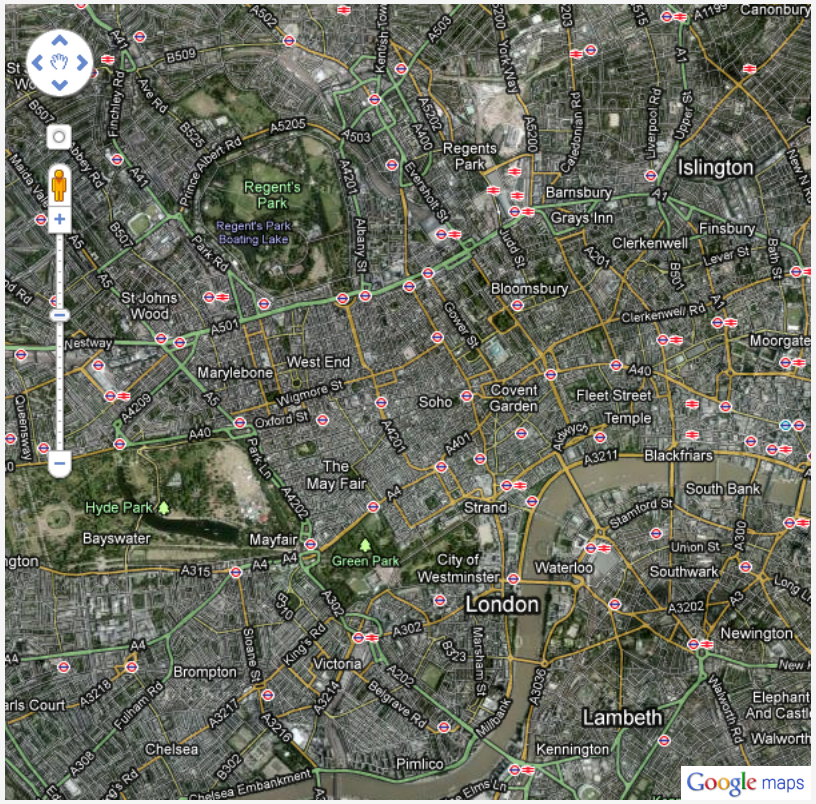
The objects are represented as vertices and the relationships are represented as edges between the vertices.



A graph is a **model** more than a collection.



Graphs: Motivating problems

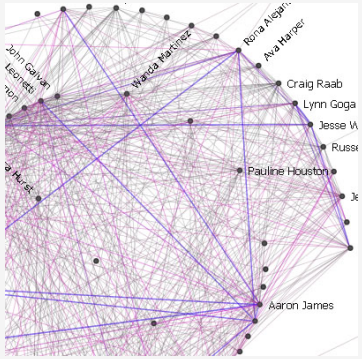


What is the shortest walking route from Westminster Abbey to Hyde Park?

What is the quickest tube route from Trafalgar Square to the Tower of London?

High definition surveillance cameras are to be added at 25 selected locations around the city, to supplement the thousands of cameras that already exist. These new HD cameras are to by physically connected to each other and to a central intelligence command center by fiber optic cables. Engineers have identified all the possible ways of laying the fiber and have cost estimates for each line. What is the cheapest way to connect all the necessary sites?

Graphs: Motivating problems



Who are the five most active friends of the various people on the terrorist watch list?

Which registered sex offenders have friends younger than 19?

Identify a group that has the strongest connections to Jane Doe.

Bachelor of Science in Computer Science (CSCI) Curriculum
2011-2012

Fall Semester			Spring Semester		
FRESHMAN YEAR					
ENGL 1100 English Composition I	3	C	ENGL 1120 English Composition II	3	C
Core History	3	C	Core History or other Social Science*	3	C
MATH 1610 Calculus I	4	C	MATH 1620 Calculus II	4	
Core Science Sequence I	4	C	Core Science Sequence II	4	C
ENGR 1110 Introduction to Engineering	2		**COMP 1210 Fundamentals of Computing I**	**3**	**M**
	16			17	
SOPHOMORE YEAR					
Core Literature	3	C	Core Literature or other Humanities*	3	C
Core Social Science*	3	C	COMP 3240 Discrete Structures	3	M
COMM 1000 Public Speaking	3		ELEC 2200 Digital Logic Circuits	3	
Science Elective	4		MATH 2660 Linear Algebra	3	
COMP 2210 Fundamentals of Computing II	**4**	**M**	**COMP 2710 Software Construction**	**3**	**M**
	17			15	
JUNIOR YEAR					
STAT 3600 Probability and Statistics	3		PHIL 1040 or PHIL 1020	3	C
COMP 3220 Principles of Prog Languages	**3**	**M**	Concentration	3	
COMP 3270 Introduction to Algorithms	**3**	**M**	Core Social Science	3	C
COMP 3350 Comp Org & Assmb Lang Prog	**3**	**M**	**COMP 3500 Intro to Operating Systems**	**3**	**M**
Core Fine Arts	3	C	**COMP 3700 Software Modeling and Design**	**3**	**M**
	15			15	
SENIOR YEAR					
Free Elective or ROTC	3		Concentration	3	
Concentration	3		**COMP 4730 Computer Ethics**	**1**	**M**
COMP 4200 Formal Languages	**3**	**M**	**COMP Elective**	**3**	**M**
COMP 4320 Computer Networks	**3**	**M**	**COMP Elective**	**3**	**M**
COMP Elective	**3**	**M**	Math Elective	3	
			UNIV 4AA0 EN1 Undergraduate Graduation	0	
	15			13	

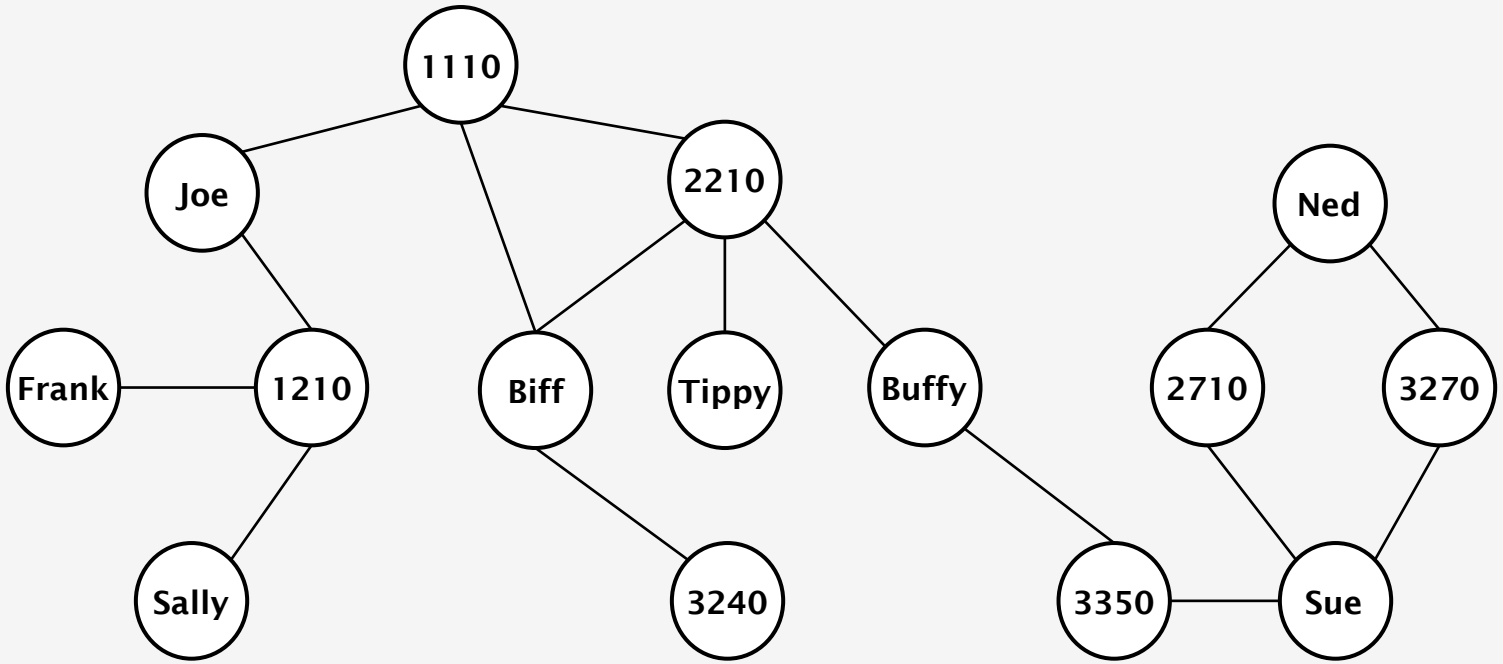
TOTAL 123 SEMESTER HOURS

What is a legal sequence of courses to take that doesn't violate any prerequisite?

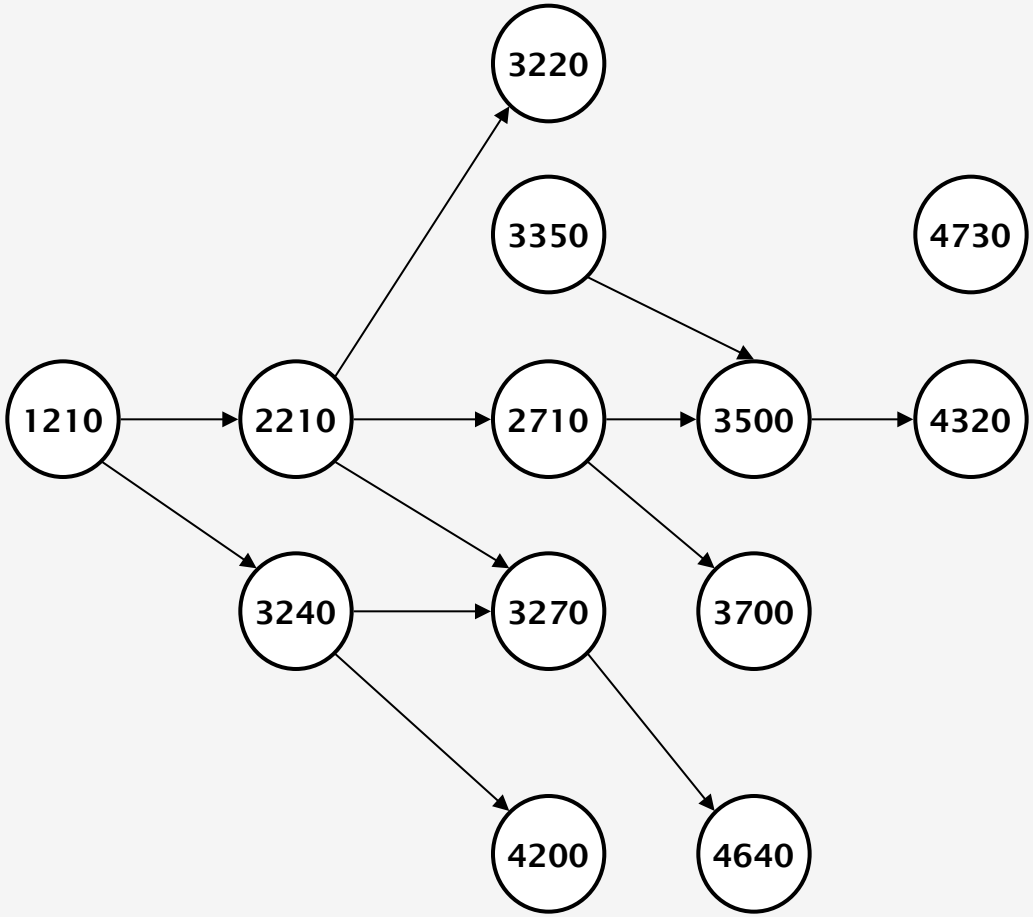
What is the fewest number of semesters required to complete the degree?

How many sets of courses are not related by prerequisites?

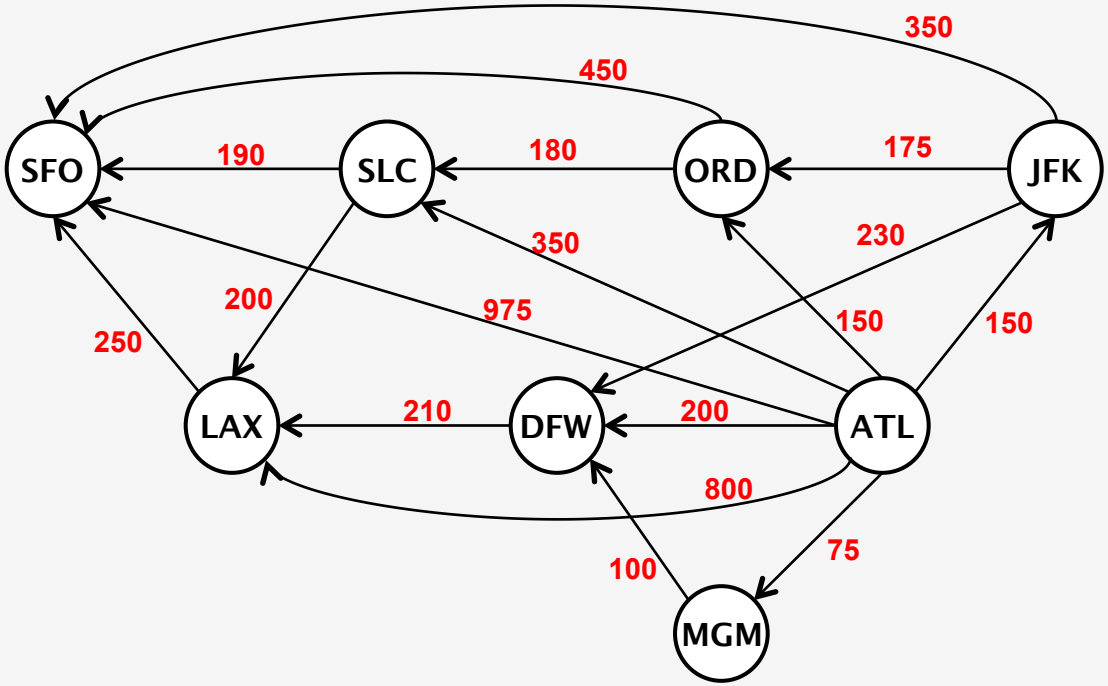
Example Graph: Students taking courses



Example Graph: Course prerequisites



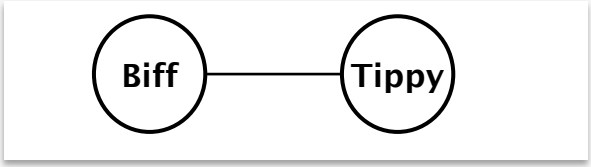
Example Graph: Direct flights from ATL to selected cities



Edge characteristics

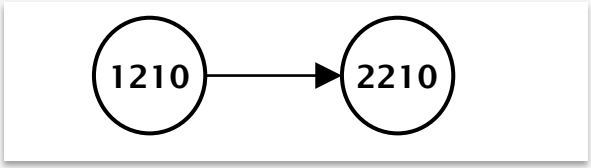
Directed v. Undirected

Undirected edges represent symmetric relationships, and are indicated by a line.



Biff and Tippy are friends.

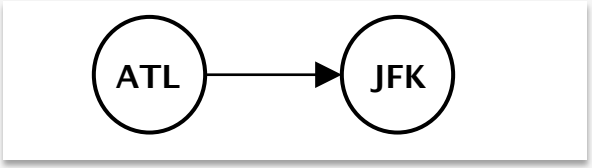
Directed edges represent asymmetric relationships, and are indicated by an arrow.

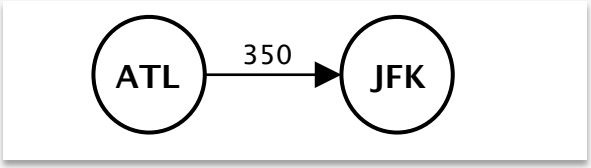


COMP 1210 is a prerequisite for COMP 2210.

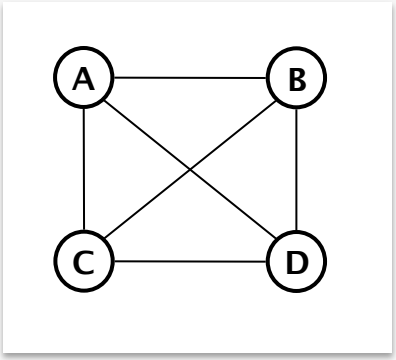
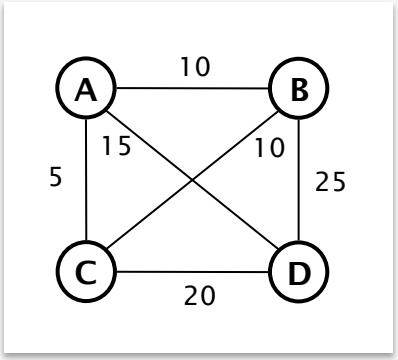
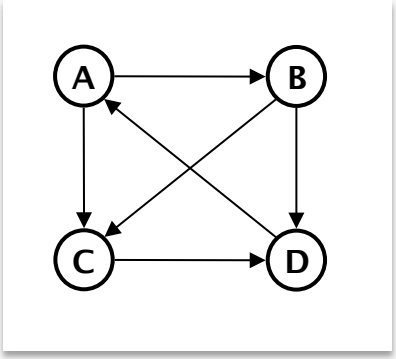
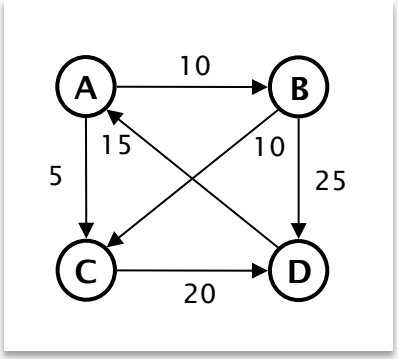
Weighted v. Unweighted

Edges can have numeric values (weights) associated with them or not. These values can represent cost, time, distance, etc. – anything that is relevant to the relationship.





Four basic graph types

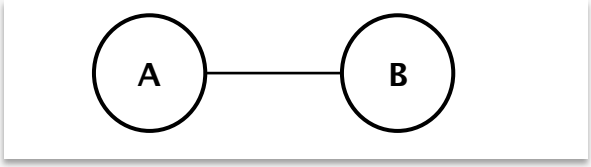
	Unweighted	Weighted
Undirected		
Directed *(digraph)*		

Adjacency

Adjacency is the basic connectedness property of vertices.

For undirected graphs

Two vertices are **adjacent** to each other iff there is an edge between them.

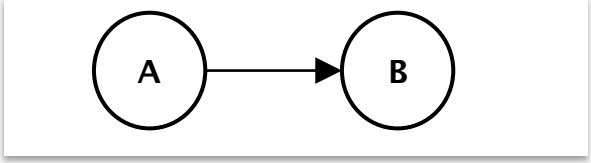


A is adjacent to B.

B is adjacent to A.

For directed graphs

Node B is **adjacent** to node A iff there is an edge from A to B.



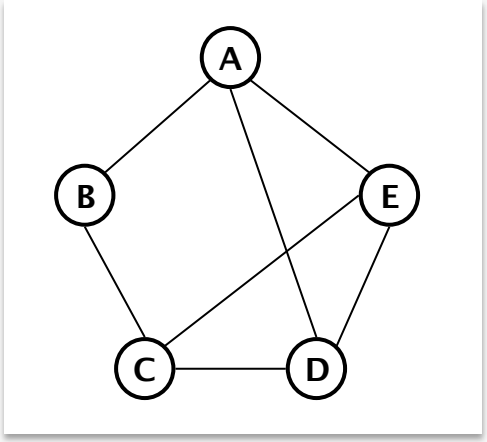
B is adjacent to A.

A is *not* adjacent to B.

Adjacency is the property that is captured in the two primary graph representations.

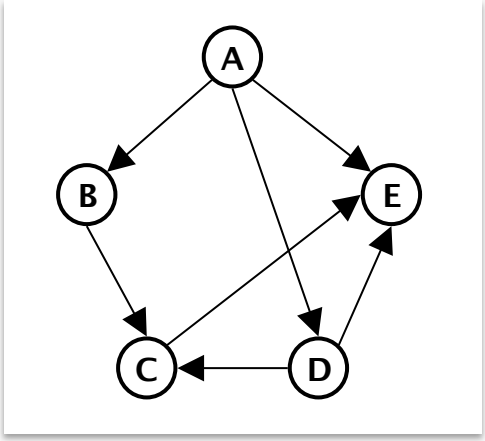
Graph representation: Adjacency Matrix

An **adjacency matrix** is a two dimensional table where both the rows and the columns represent the vertices of the graph. Cell *(i, j)* indicates if vertex *j* is adjacent to vertex *i*.



	A	B	C	D	E
A		1		1	1
B	1		1		
C		1		1	1
D	1		1		1
E	1		1	1	

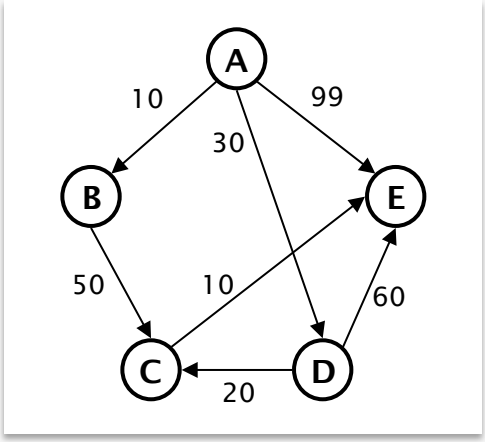
Undirected graphs will always have a symmetric matrix.



	A	B	C	D	E
A		1		1	1
B			1		
C					1
D			1		1
E					

Graph representation: Adjacency Matrix

For a weighed graph, cell (i, j) indicates if vertex j is adjacent to vertex i by storing the weight of the edge from vertex i to vertex j.



	A	B	C	D	E
A		10		30	99
B			50		
C					10
D			20		60
E					

Space complexity: For a graph with n vertices, an adjacency matrix will require O(n^2) memory. *(Upper bound; can do better)*

Good for: Edge-existence questions – O(1)

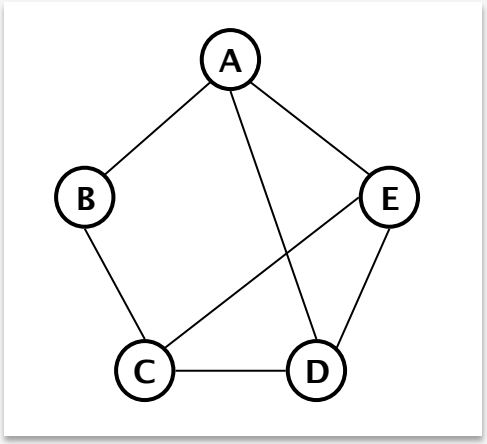
Is there an edge between A and B?
How much does it cost to go from A to B?

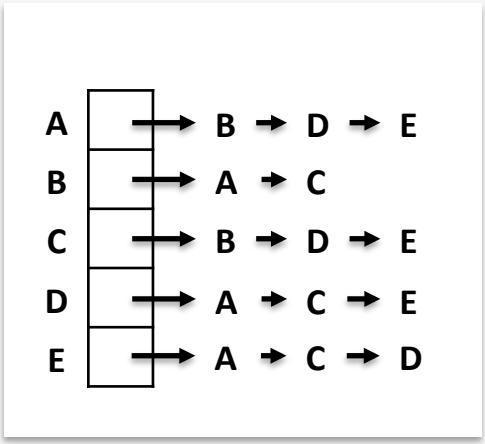
Esoteric questions that can be answered with linear algebra.

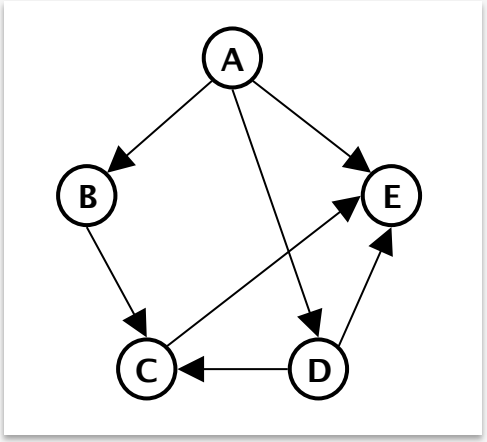
Are A and B connected by a path of length k?

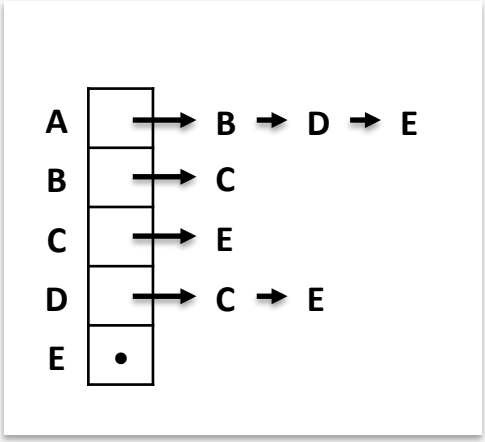
Graph representation: Adjacency List

An **adjacency list** is a one dimensional table where each entry represents the vertices of the graph. Entry k stores a linked list of all the vertices that are adjacent to vertex k.



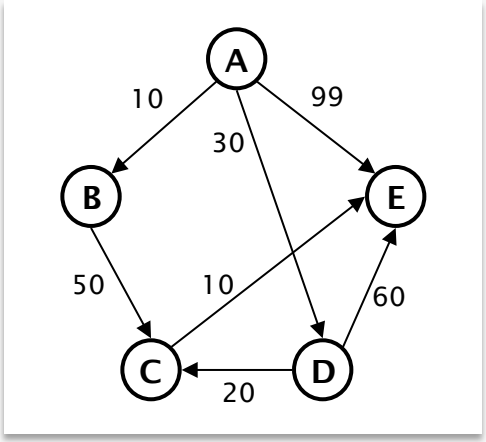


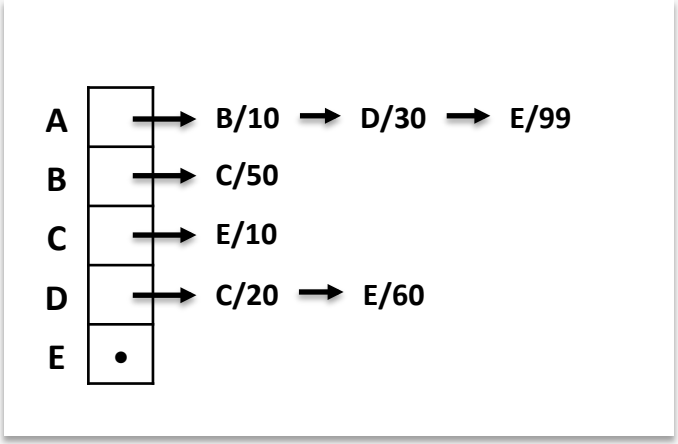




Graph representation: Adjacency List

For a weighted graph, entry k stores a linked list of all the vertices that are adjacent to vertex k. Each node in the linked list stores not only the label of an adjacent vertex m but also the weight on the edge from k to m.





Space complexity: For a graph with n vertices and e edges, an adjacency list will require O($n + e$) memory.

Good for: Sparse graphs, in general.

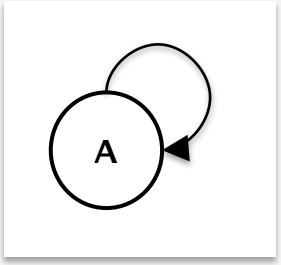
Finding the neighbors of a given node.

Better for directed graphs than undirected, especially if the undirected graph is weighted.

Glossary

Vertex *Edge* *Directed edge* *Undirected edge* *Weighted edge*

Unweighted edge *Adjacency*



Self loop – an edge that links a vertex to itself.

Simple graph – a graph with no self loops.

Path – A sequence of vertices/edges from a start vertex to an end vertex.

Simple path – A path that does not cross the same edge twice.

Cycle – A simple path that starts and ends at the same vertex.

Acyclic graph – a graph with no cycles.

Paths

A-B-C

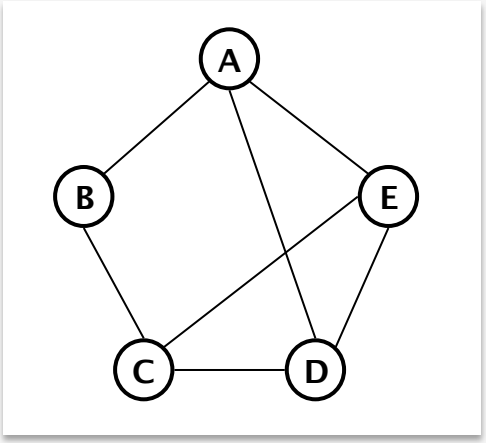
D-C-B-A-E

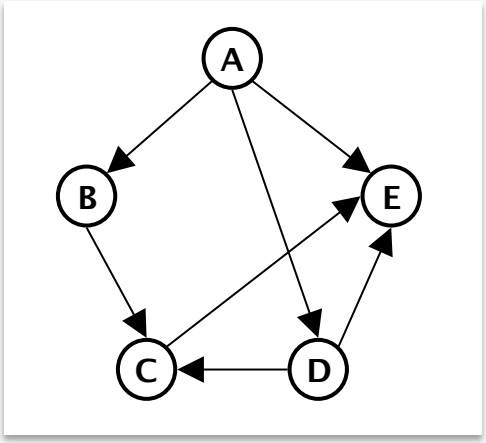
A-B-C-B

A-B-C-B-A

Cycle

A-E-D-C-B-A





Paths

A-B-C

A-B-C-E

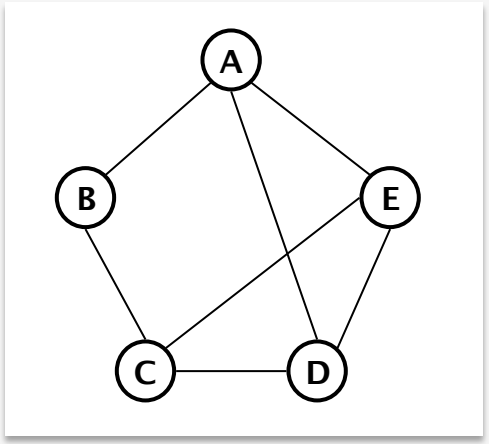
D-C-E

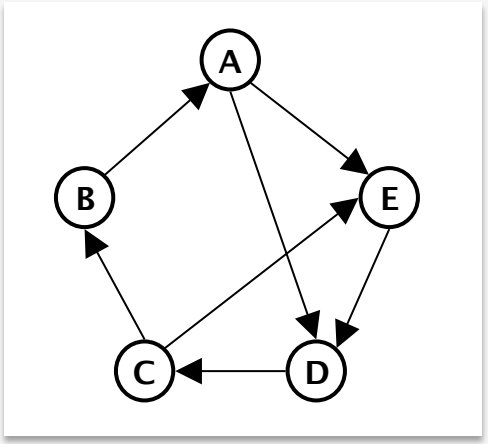
Acyclic

Glossary

Connected Graph – a graph in which there is a simple path between any two pair of vertices.

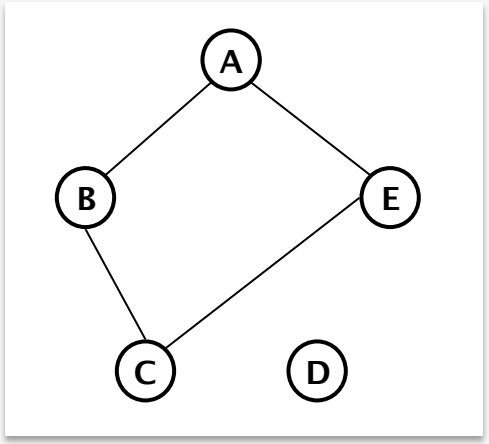
Connected:

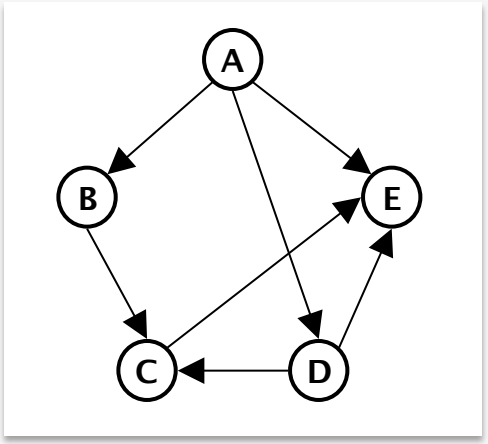




Strongly connected

Not connected:

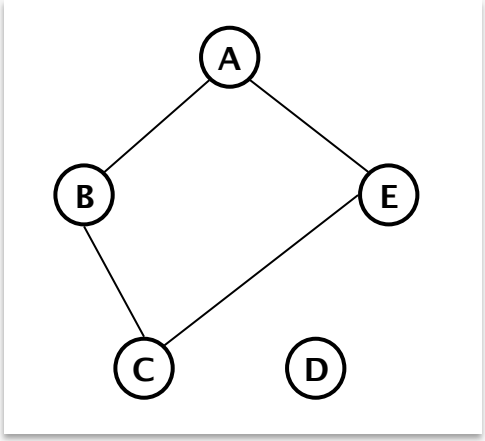




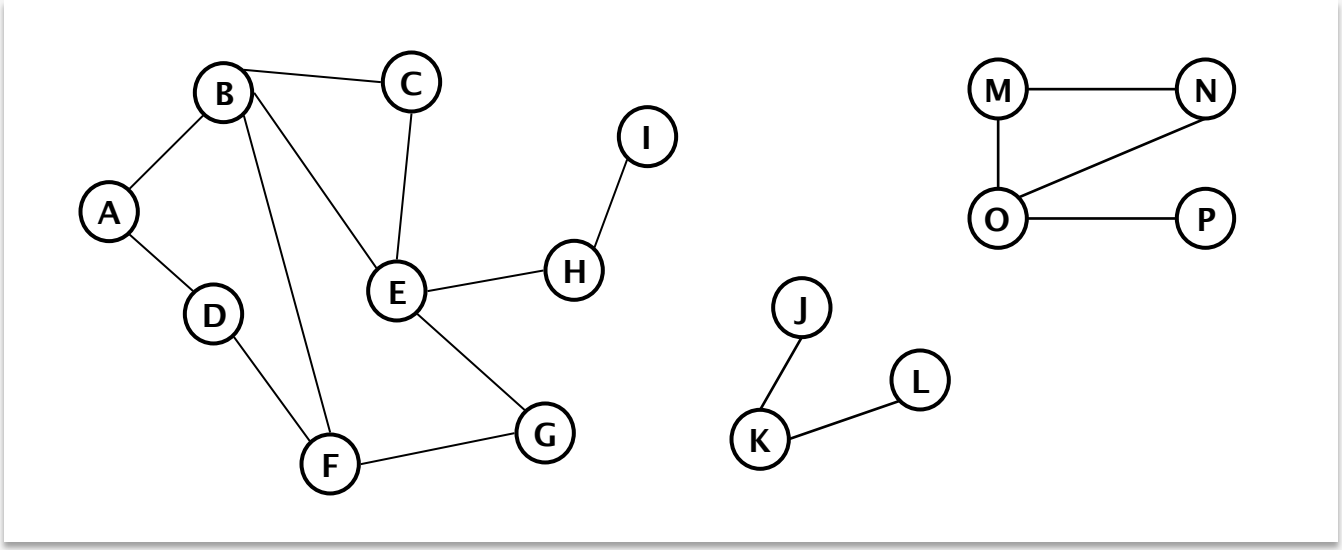
Not strongly connected

Glossary

Connected Component – maximal subgraph that is connected.



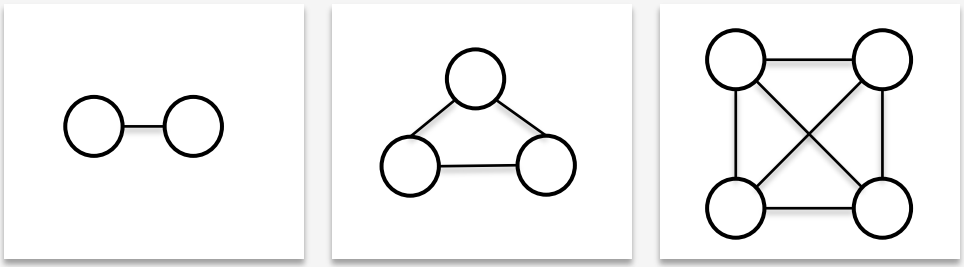
2 connected components

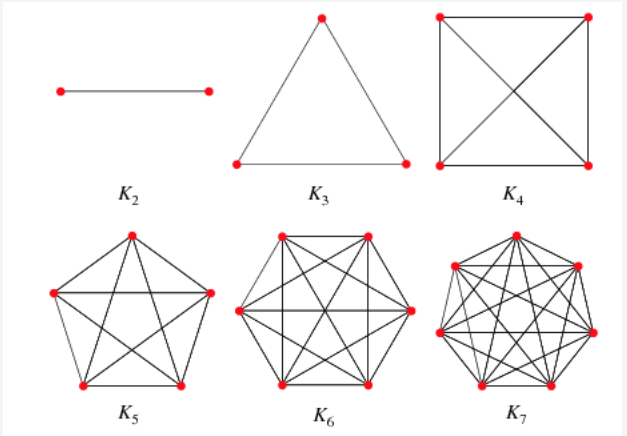


3 connected components

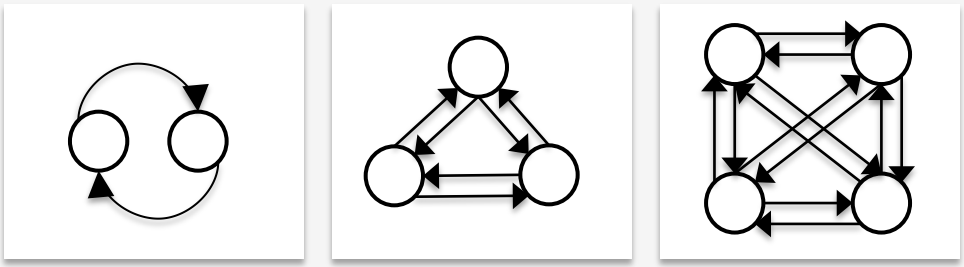
Glossary

Complete Graph – a simple graph in which every pair of vertices is adjacent to each other.





A complete undirected graph with N vertices will have N(N-1)/2 edges.

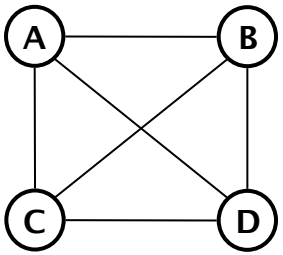


A complete directed graph with N vertices will have N(N-1) edges.

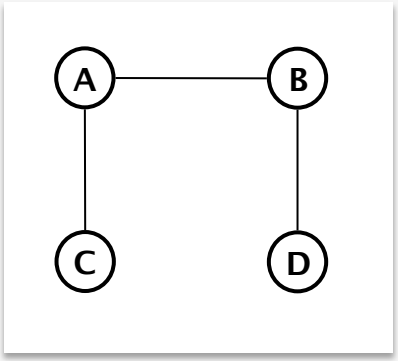
Glossary

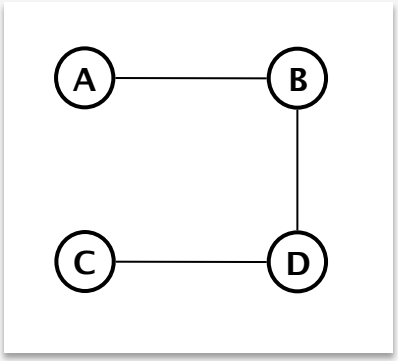
Spanning Tree – A spanning tree of a *connected, undirected* graph is a connected, acyclic subgraph that contains all the vertices of the graph.

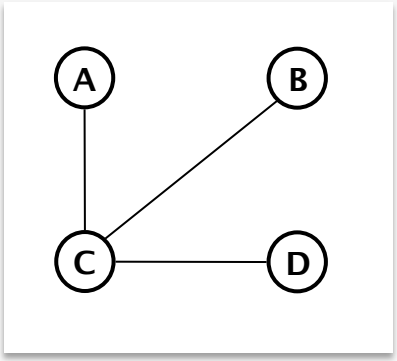
Graph:

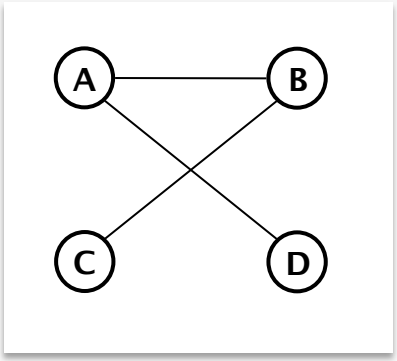


Spanning Trees:

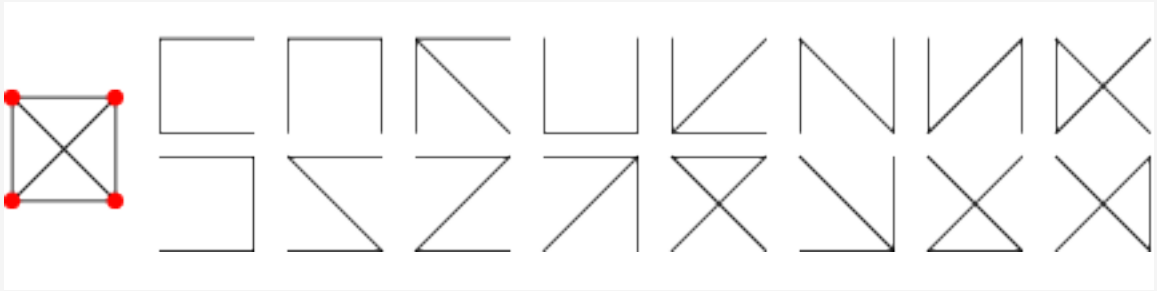








and many more …

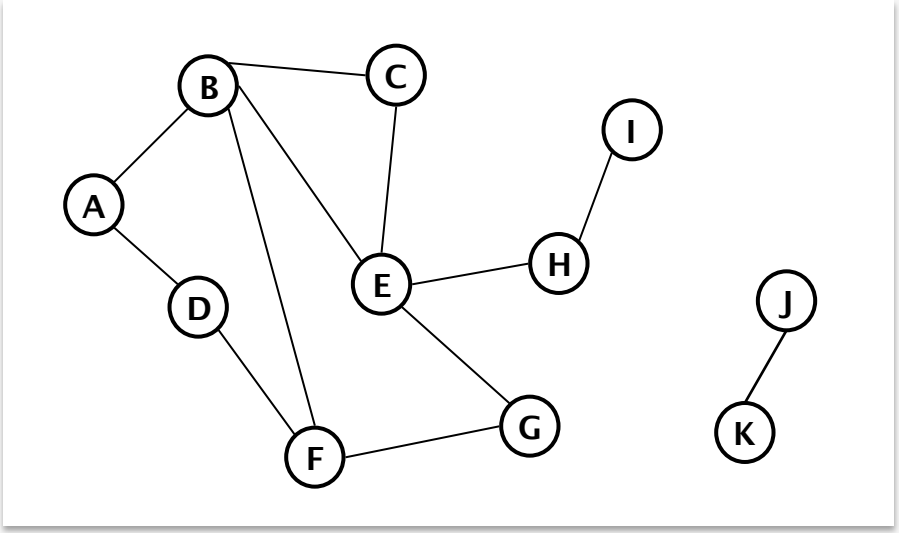


Graph traversals: Depth-First

Explore the graph by looking for new vertices far away from the start vertex, and examining nearer vertices only when dead ends are encountered.

Has a very simple recursive formulation.

```
dfs(Vertex v)
visit(v)
mark v as visited
for each w adjacent to v {
   if notVisited(w) {
      dfs(w)
   }
}
```



Will visit each vertex that is *reachable* from the start vertex.

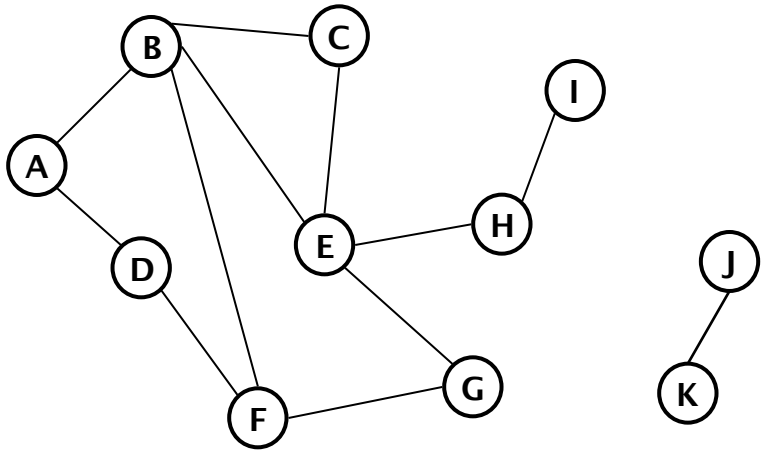
For a graph with V vertices and E edges, DFS can be implemented with O(V + E) time complexity.

Graph traversals: Breadth-First

Explore the graph by looking all the vertices closest to the start vertex, and move farther away only when everything nearby has been examined.

Typically implemented iteratively with a FIFO queue.

```
bfs(Vertex v)
visit(v)
mark v as visited
queue.add(v)
while (!queue.isEmpty()) {
   w = queue.remove()
   for each p adjacent to w {
      if notVisited(p) {
         visit(p)
         mark p as visited
         queue.add(p)
      }
    }
}
```



Will visit each vertex that is *reachable* from the start vertex.

For a graph with V vertices and E edges, BFS can be implemented with O(V + E) time complexity.

DFS, BFS are the basis of many important solutions

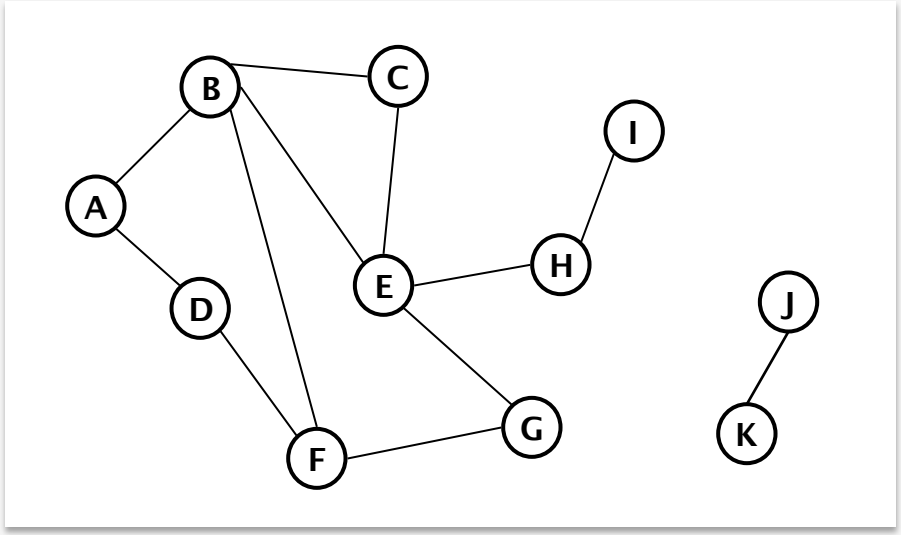
Is there a path from A to H?

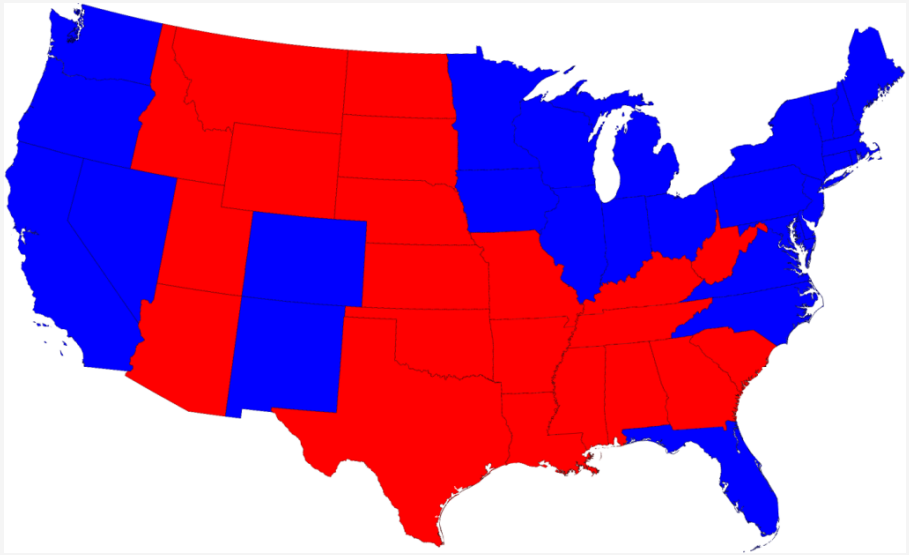
What is the shortest path from A to H?

Does the graph have any cycles?

Is the graph connected?

Identify the connected components in the graph.

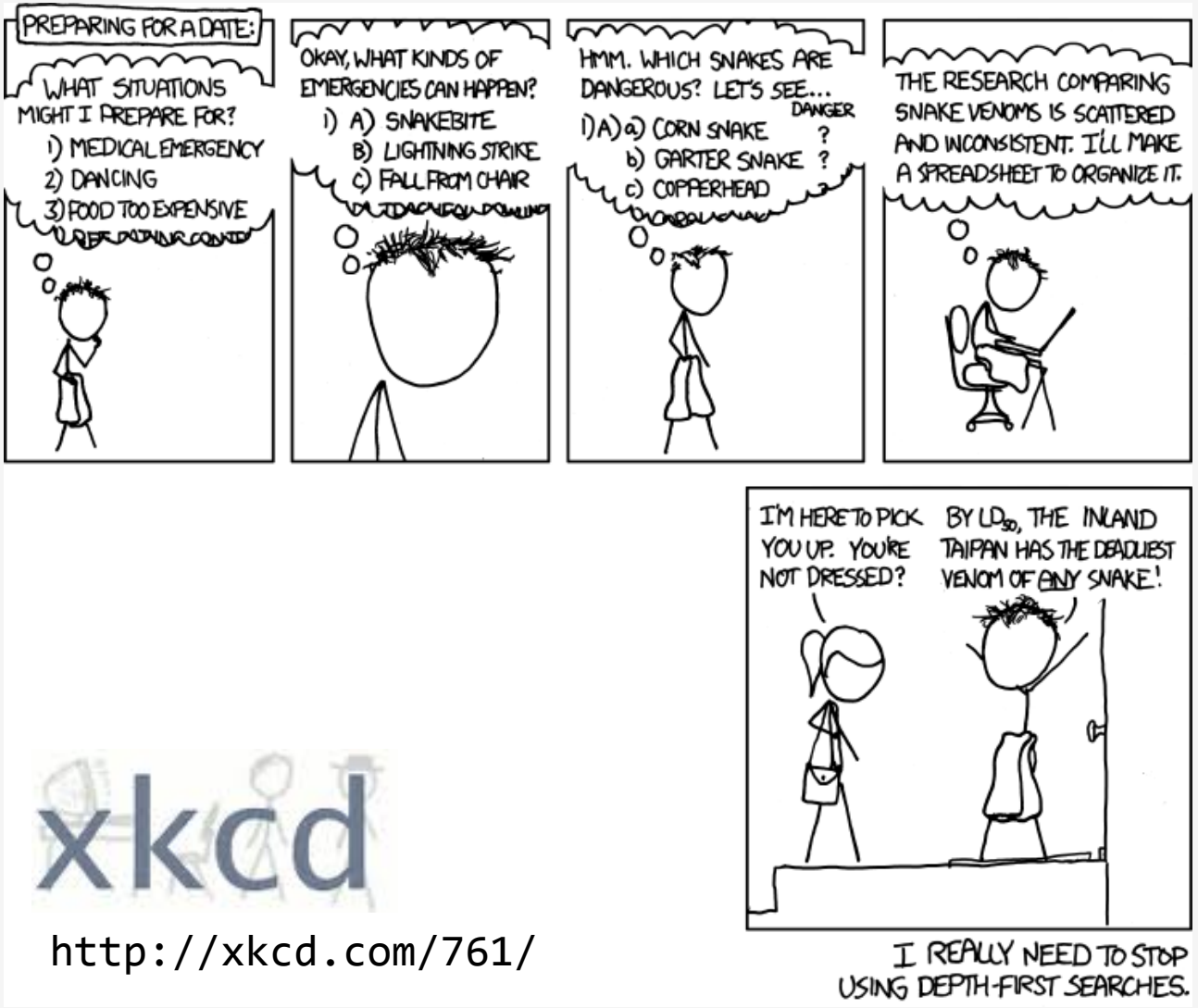




Color Florida red.

Two connected components – blue states, red states. Move Florida from the blue component to the red component.

But sometimes one is more applicable than the other…

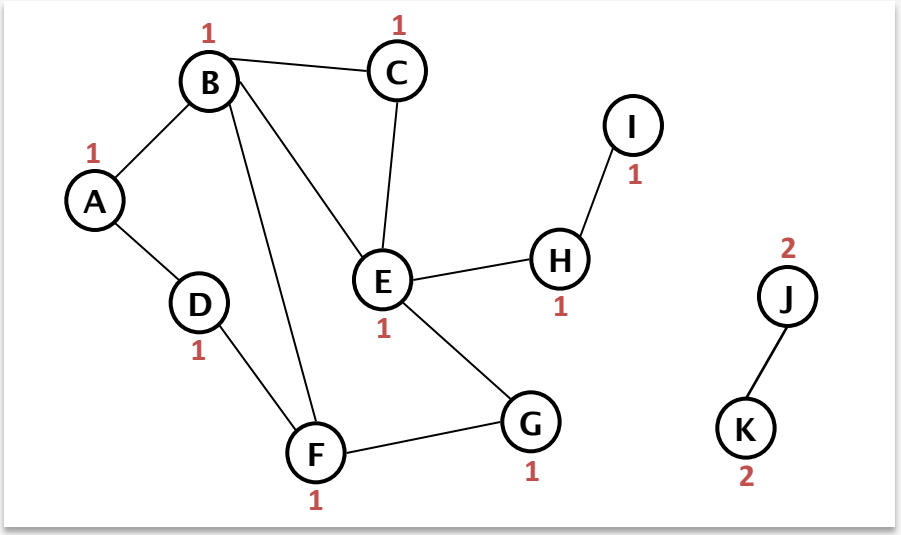


xkcd

http://xkcd.com/761/

DFS: connected components

```
dfs(Vertex v)
visit(v)
mark v as visited
for each w adjacent to v {
   if notVisited(w) {
      dfs(w)
   }
}
```



Modify DFS to label all vertices in same component as v.

```
dfsComp(Vertex v, int c)
visit(v)
mark v as visited
mark v with c
for each w adjacent to v {
   if notVisited(w) {
      dfsComp(w, c)
   }
}
```

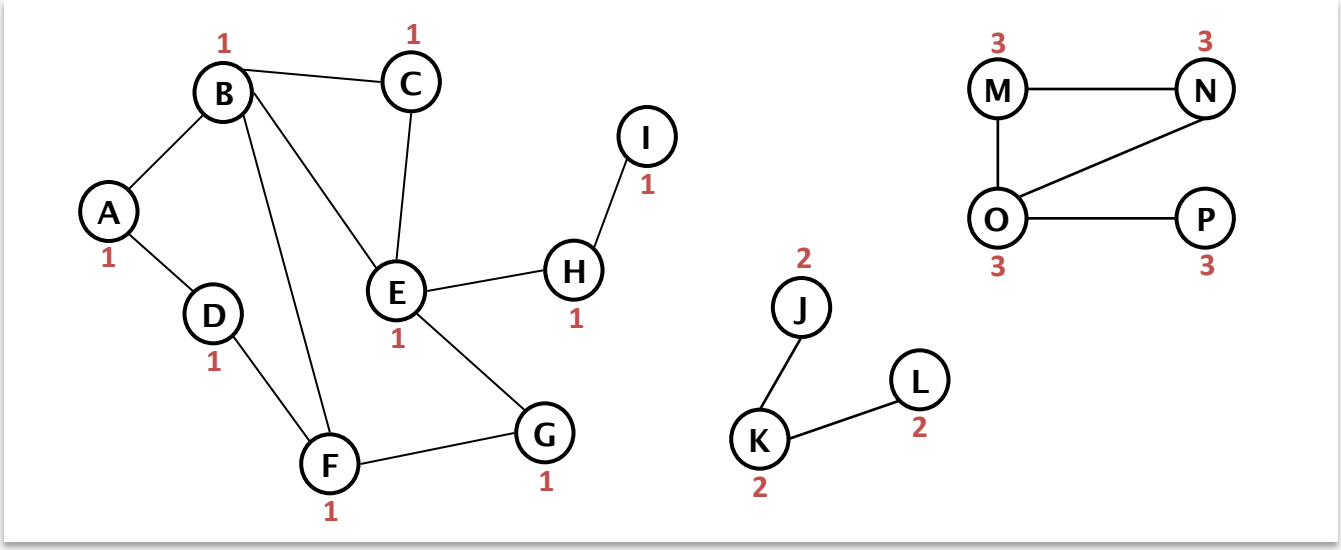
Do this for each vertex.

```
dfsCompDriver()
c = 0
for each vertex v {
   if notVisited(v) {
      c = c + 1
      dfsComp(v, c)
   }
}
```

DFS: connected components

```
dfsCompDriver()
c = 0
for each vertex v {
   if notVisited(v) {
      c = c + 1
      dfsComp(v, c)
   }
}
```

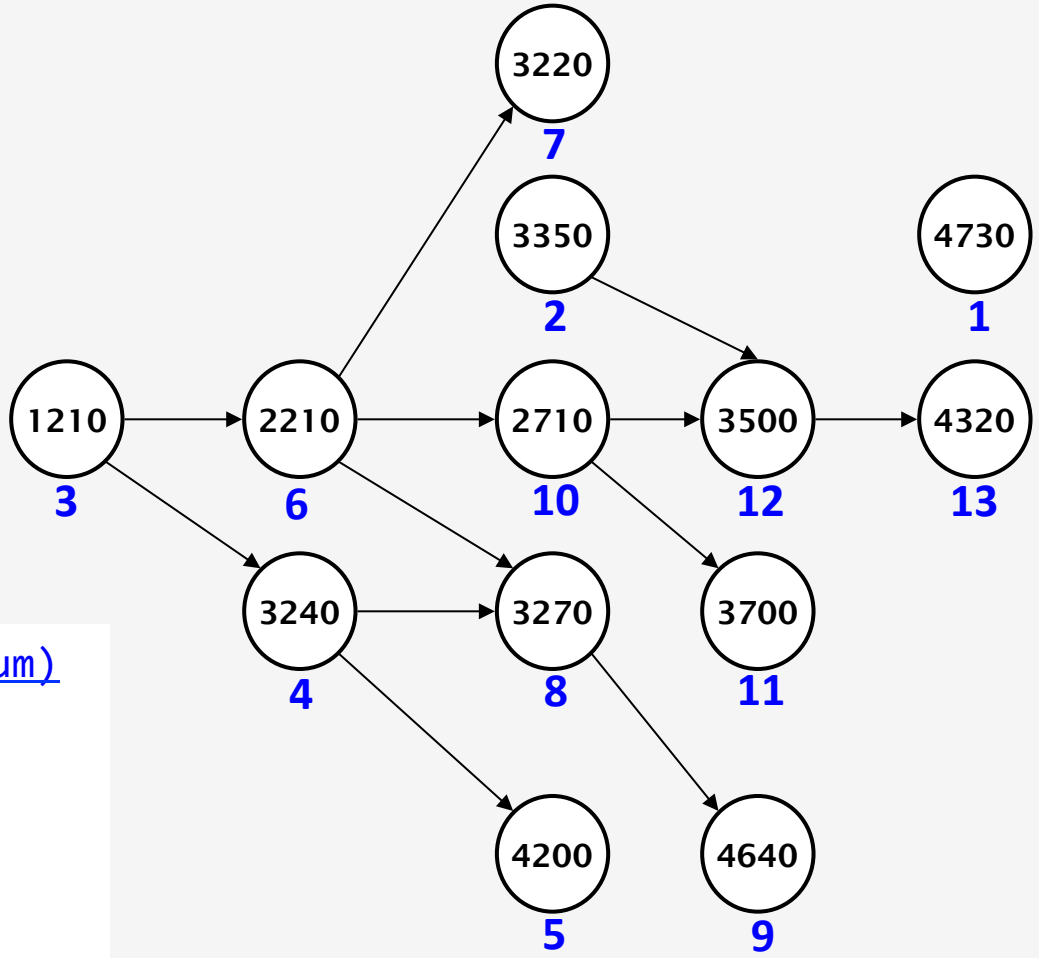
```
dfsComp(Vertex v, int c)
visit(v)
mark v as visited
mark v with c
for each w adjacent to v {
   if notVisited(w) {
      dfsComp(w, c)
   }
}
```



DFS: topological sorting

List the nodes in an order such that for every edge (u, v) u appears before v in the order.

```
dfsTSDriver()
topnum = N // #vertices
for each vertex v {
   if notVisited(v) {
      dfsTopSort(v, topnum)
   }
}
```



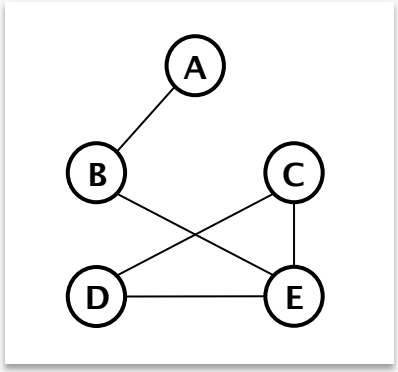

```
dfsTopSort(Vertex v, int topnum)
visit(v)
mark v as visited
for each w adjacent to v {
   if notVisited(w) {
      dfsTopSort(w, topnum)
   }
}
mark v with topnum
topnum = topnum - 1
```

DFS: Hamilton paths

A **Hamilton path** is a path in a graph that visits each vertex in the graph exactly once.

Examples:

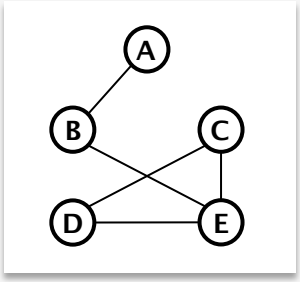
C-D-E-B-A
D-C-E-B-A
A-B-E-D-C




```
dfsHamilton(Vertex v, int length, int N)
mark v as visited
if (length == N) {
   // just found a Hamilton path
   // do whatever is appropriate
   return
}
for each w adjacent to v {
   if notVisited(w) {
      dfsHamilton(w, length + 1, N)
   }
}
mark v as unvisited
```

```
dfsHamiltonDriver()
for each vertex v {
   if notVisited(v) {
      dfsHamilton(v, 1, 5)
   }
}
```

DFS: Hamilton paths




```
dfsHamiltonDriver()
for each vertex v
   if notVisited(v)
      dfsHamilton(v, 1, 5)
```

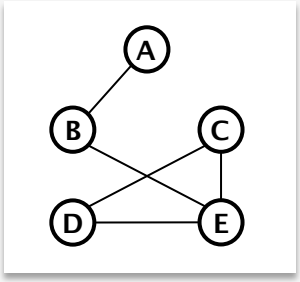
```
dfsHamilton(Vertex v, int length, int N)
mark v as visited
if (length == N)   // found a Hamilton path
   return
for each w adjacent to v
   if notVisited(w)
      dfsHamilton(w, length + 1, N)
mark v as unvisited
```

current vertex

neighbors

path to v	v	length	N	comment	A	B	C	D	E
	C	1	5	{D, E} : choose E	F	F	T	F	F
C	E	2	5	{B, D} : choose D	F	F	T	F	T
C E	D	3	5	{} : dead end	F	F	T	T	T
C E	*D*	*3*	*5*	*clean up, backtrack*	*F*	*F*	*T*	***F***	*T*
C	E	2	5	{B} : choose B	F	F	T	F	T
C E	B	3	5	{A} : choose A	F	T	T	F	T
C E B	A	4	5	{} : dead end	T	T	T	F	T
C E B	*A*	*4*	*5*	*clean up, backtrack*	***F***	*T*	*T*	*F*	*T*
C E	B	3	5	{} : dead end	F	T	T	F	T
C E	*B*	*3*	*5*	*clean up, backtrack*	*F*	***F***	*T*	*F*	*T*
C	E	2	5	{} : dead end	F	F	T	F	T
C	*E*	*2*	*5*	*clean up, backtrack*	*F*	*F*	*T*	*F*	***F***
	C	1	5	{D} : choose D	F	F	T	F	F

DFS: Hamilton paths



current vertex

neighbors

path to v	v	length	N	comment	A	B	C	D	E
	C	1	5	{D, E} : choose E	F	F	T	F	F
C	E	2	5	{B, D} : choose D	F	F	T	F	T
C E	D	3	5	{} : dead end	F	F	T	T	T
C E	*D*	*3*	*5*	*clean up, backtrack*	*F*	*F*	*T*	***F***	*T*
C	E	2	5	{B} : choose B	F	F	T	F	T
C E	B	3	5	{A} : choose A	F	T	T	F	T
C E B	A	4	5	{} : dead end	T	T	T	F	T
C E B	*A*	*4*	*5*	*clean up, backtrack*	***F***	*T*	*T*	*F*	*T*
C E	B	3	5	{} : dead end	F	T	T	F	T
C E	*B*	*3*	*5*	*clean up, backtrack*	*F*	***F***	*T*	*F*	*T*
C	E	2	5	{} : dead end	F	F	T	F	T
C	*E*	*2*	*5*	*clean up, backtrack*	*F*	*F*	*T*	*F*	***F***
	C	1	5	{D} : choose D	F	F	T	F	F
C	D	2	5	{E} : choose E	F	F	T	T	F
C D	E	3	5	{B} : choose B	F	F	T	T	T
C D E	B	4	5	{A} : choose A	F	T	T	T	T
C D E B	**A**	**5**	**5**	**Hamilton path found**	**T**	**T**	**T**	**T**	**T**

Reached each vertex with no backtracking.

References

1. http://en.wikipedia.org/wiki/Graph_(data_structure)
2. http://en.wikipedia.org/wiki/Graph_(mathematics)
3. http://mathworld.wolfram.com/Graph.html
4. http://algs4.cs.princeton.edu/40graphs/
5. http://en.wikipedia.org/wiki/Graph_theory