

CPSC 221: Data Structures

Graph Theory

Alan J. Hu

(Many slides gratefully stolen from Steve Wolfman)

Learning Goals

After this unit, you should be able to:

- Describe the properties and possible applications of various kinds of graphs (e.g., simple, complete), and the relationships among vertices, edges, and degrees.
- Prove basic theorems about simple graphs (e.g. handshaking theorem).
- Convert between adjacency matrices/lists and their corresponding graphs.
- Determine whether two graphs are isomorphic.
- Determine whether a given graph is a subgraph of another.
- Perform breadth-first and depth-first searches in graphs.
- Explain why graph traversals are more complicated than tree traversals.

Graphs and Graph Theory

- (These are not the sorts of graphs you've seen in algebra class!)
- Graphs are a really powerful formalism to model the *relationships* between *things*.
- There are many variations of graphs. We'll study several common varieties.

Graph Examples

- Graphs are a really powerful formalism to model the *relationships* between *things*.
 - Things can be whatever you want. Relationships can be any relation that makes sense on those things.
- Examples:
 - Cities, highways; intersections, streets; airports, flights; etc.
 - webpages, links; Facebook ids, friends; computers, network connections; etc.
 - Tasks, dependencies; variables, data flow; statements, control flow;

Drawing Graphs

- We draw graphs with circles or dots for the things (called vertices, or nodes) and lines or arrows for the relationships (called edges, or arcs).
 - Vertices usually labeled.
 - Edges can be labeled, too.
- Examples:
 - Cities, highways; intersections, streets; airports, flights; etc.
 - webpages, links; Facebook ids, friends; computers, network connections; etc.
 - Tasks, dependencies; variables, data flow; statements, control flow; etc.

Graph ADT

Graphs are a formalism useful for representing relationships between things

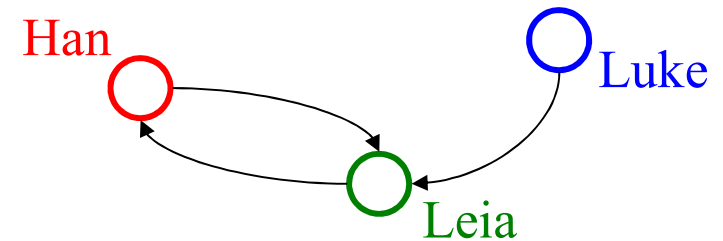
– a graph G is represented as

$$G = (V, E)$$

- V is a set of vertices: $\{v_1, v_2, \dots, v_n\}$
- E is a set of edges: $\{e_1, e_2, \dots, e_m\}$ where each e_i connects two vertices (v_{i1}, v_{i2})

– operations might include:

- creation (with a certain number of vertices)
- inserting/removing edges
- iterating over vertices adjacent to a specific vertex
- asking whether an edge exists connecting two vertices



$V = \{\text{Han}, \text{Leia}, \text{Luke}\}$

$E = \{(\text{Luke}, \text{Leia}),$
 $(\text{Han}, \text{Leia}),$
 $(\text{Leia}, \text{Han})\}$

Today's Outline

- Topological Sort: Getting to Know Graphs with a Sort
- Graph ADT and Graph Representations
- Graph Terminology (a lot of it!)
- More Graph Algorithms
 - Shortest Path (Dijkstra's Algorithm)
 - Minimum Spanning Tree (Kruskal's Algorithm)

Total Order

1

2

3

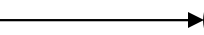
4

5

6

7

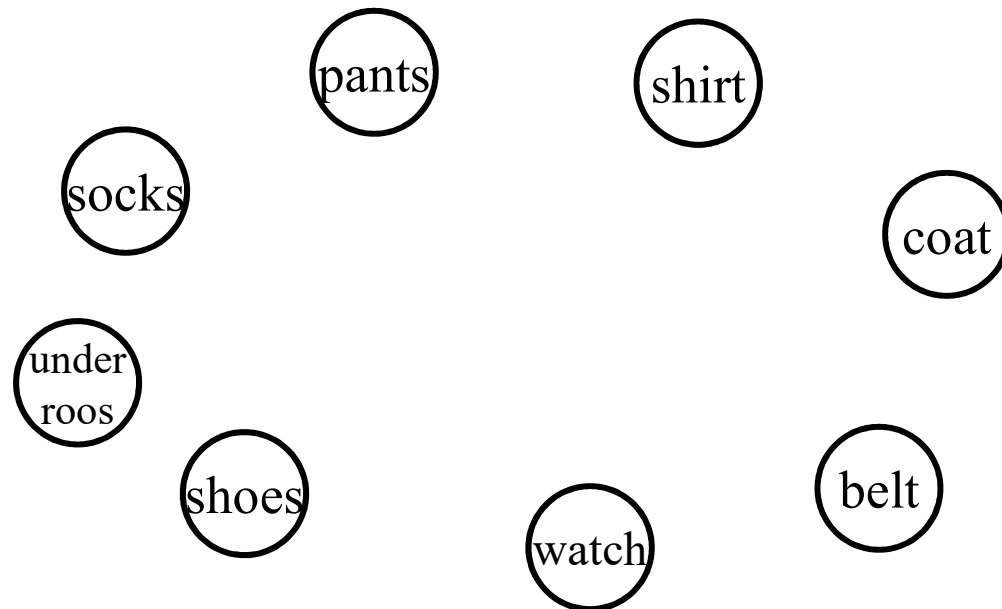
A



B

means A must go before B

Partial Order: Getting Dressed



Topological Sort

Given a graph, $G = (V, E)$, output all the vertices in V such that no vertex is output before any other vertex with an edge to it.

Topo-Sort Take One

Label each vertex's *in-degree* (# of inbound edges)

While there are vertices remaining

 Pick a vertex with in-degree of zero and output it

 Reduce the in-degree of all vertices adjacent to it

 Remove it from the list of vertices

runtime:

Topo-Sort Take Two

Label each vertex's in-degree

Initialize a queue to contain all in-degree zero vertices

While there are vertices remaining in the queue

- Pick a vertex v with in-degree of zero and output it

- Reduce the in-degree of all vertices adjacent to v

- Put any of these with new in-degree zero on the queue

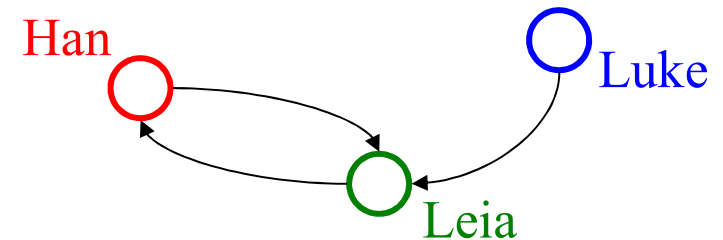
- Remove v from the queue

runtime:

Today's Outline

- Topological Sort: Getting to Know Graphs with a Sort
- Graph ADT and Graph Representations
- Graph Terminology (a lot of it!)
- More Graph Algorithms
 - Shortest Path (Dijkstra's Algorithm)
 - Minimum Spanning Tree (Kruskal's Algorithm)

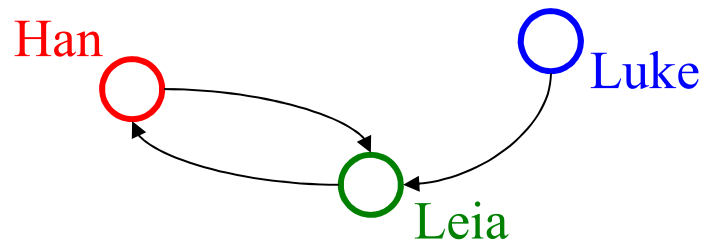
Graph Representations



- 2-D matrix of vertices (marking edges in the cells)
“adjacency matrix”
- List of vertices each with a list of adjacent vertices
“adjacency list”

Adjacency Matrix

A $|V| \times |V|$ array in which an element (u, v) is true if and only if there is an edge from u to v

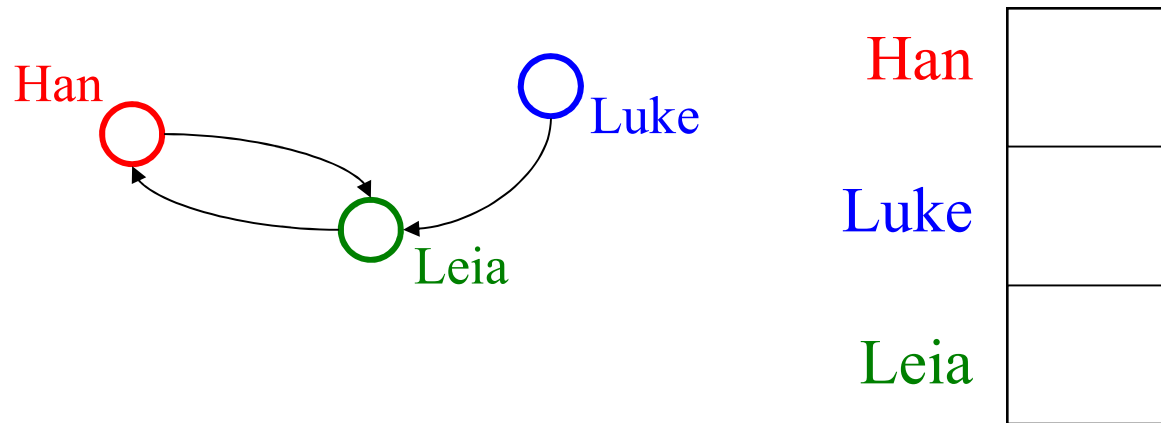


	Han	Luke	Leia
Han			
Luke			
Leia			

runtime for various operations? space requirements:

Adjacency List

A $|V|$ -ary list (array) in which each entry stores a list (linked list) of all adjacent vertices



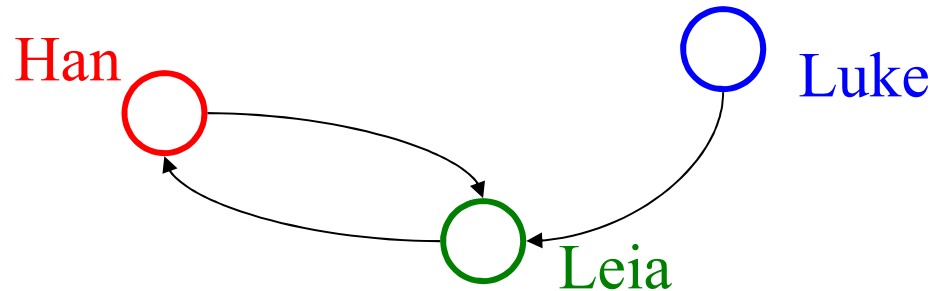
runtime for various operations? space requirements:

Today's Outline

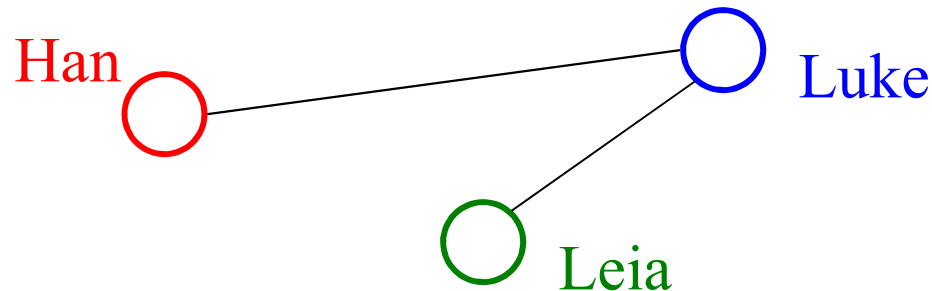
- Topological Sort: Getting to Know Graphs with a Sort
- Graph ADT and Graph Representations
- Graph Terminology (a lot of it!)
- More Graph Algorithms
 - Shortest Path (Dijkstra's Algorithm)
 - Minimum Spanning Tree (Kruskal's Algorithm)

Directed vs. Undirected Graphs

- In *directed* graphs, edges have a specific direction:



- In *undirected* graphs, they don't (edges are two-way):



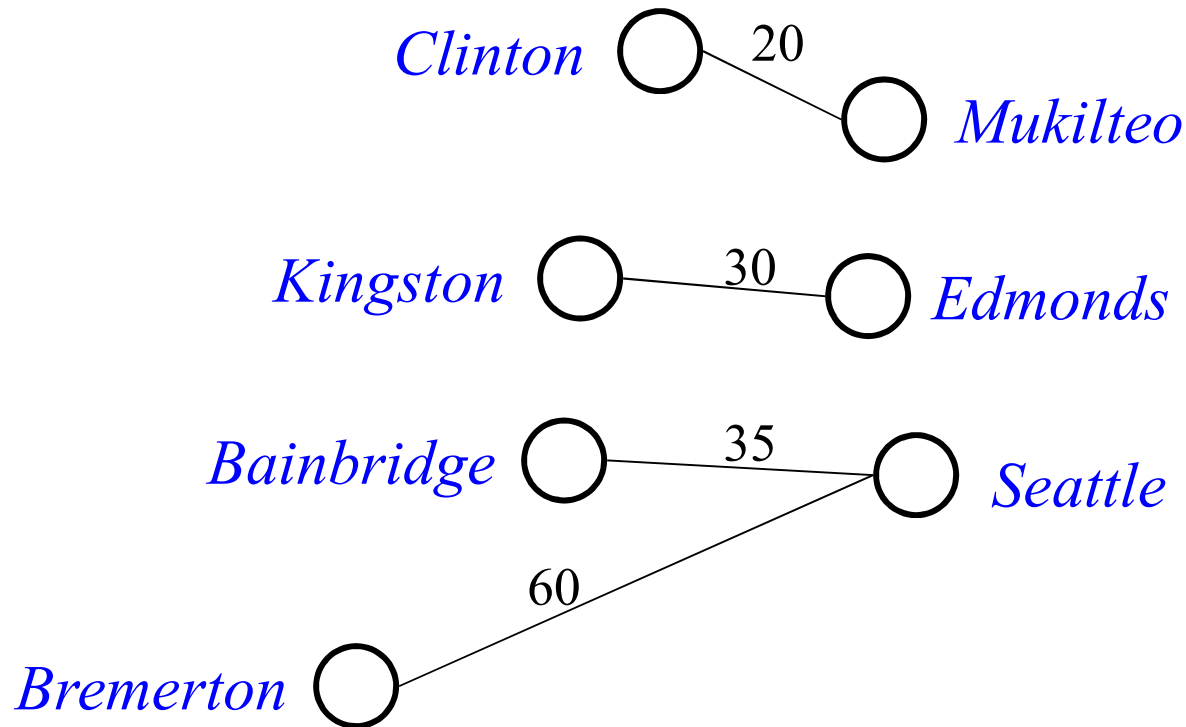
- Vertices \mathbf{u} and \mathbf{v} are *adjacent* if $(\mathbf{u}, \mathbf{v}) \in \mathbf{E}$

Directed vs. Undirected Graphs

- Adjacency lists and matrices both work fine to represent *directed* graphs.
- To represent *undirected* graphs, either ensure that both orderings of every edge are included in the representation or ensure that the order doesn't matter (e.g., always use a “canonical” order), which works poorly in adjacency lists.

Weighted Graphs

Each edge has an associated weight or cost.



How can we store weights in an adjacency matrix?

In an adjacency list?

Graph Density

A *sparse* graph has $O(|V|)$ edges

A *dense* graph has $\Theta(|V|^2)$ edges

Anything in between is either on the sparse side or on the dense side,
depending critically on context!

Graph Density

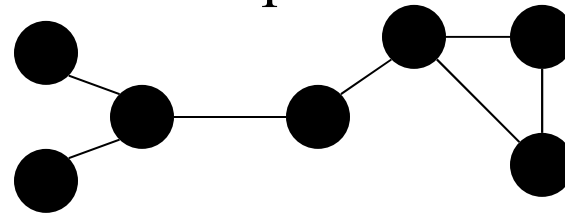
A *sparse* graph has $O(|V|)$ edges

Why is the adjacency list likely to be a better representation than the adjacency matrix for sparse graphs?

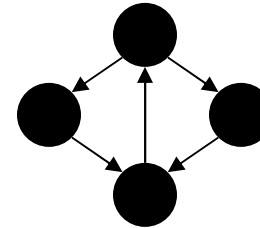
- a. Sparse graphs have too few edges to fit in an adjacency matrix.
- b. Much of the matrix will be “wasted” on 0s.
- c. The adjacency list will guarantee better performance on a sparse graph.
- d. None of these.

Connectivity

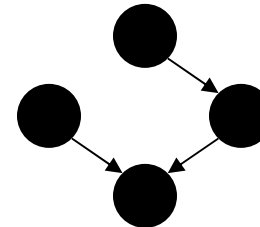
Undirected graphs are *connected* if there is a path between any two vertices



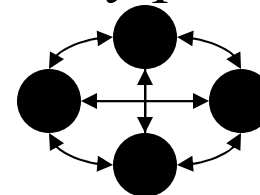
Directed graphs are *strongly connected* if there is a path from any one vertex to any other



Di-graphs are *weakly connected* if there is a path between any two vertices, *ignoring direction*



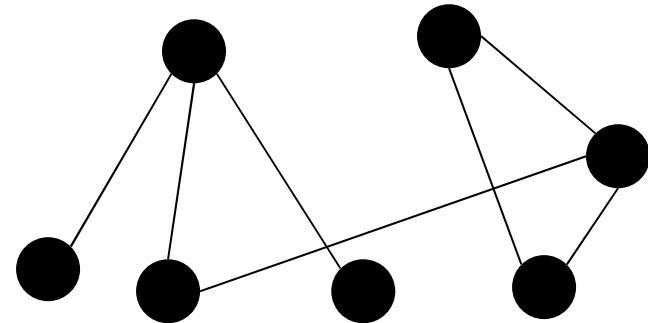
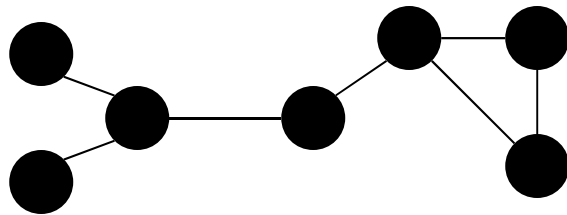
A *complete* graph has an edge between every pair of vertices



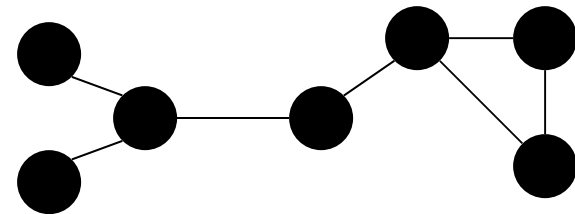
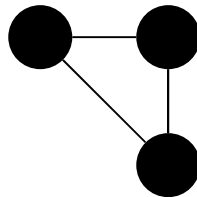
Isomorphism and Subgraphs

We often care only about the structure of a graph, not the names of its vertices. Then, we can ask:

“Are two graphs *isomorphic*?” Do the graphs have identical structure? Can you “line up” their vertices so that their edges match?



“Is one graph a *subgraph* of the other?” Is one graph isomorphic to a part of the other graph (a subset of its vertices and a subset of the edges connecting those vertices)?



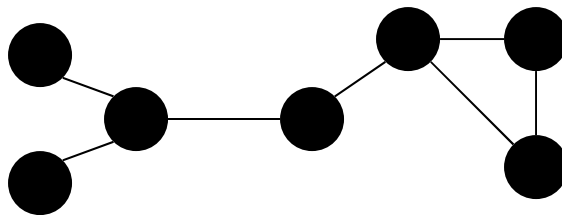
Degree

- The *degree* of a vertex v in V is denoted $\deg(v)$ and represents the number of edges incident on v . (An edge from v to itself contributes 2 towards the degree.)
- Handshaking Theorem: If $G=(V,E)$ is an undirected graph, then:
$$\sum_{v \in V} \deg(v) = 2|E|$$
- **Corollary:** An undirected graph has an even number of vertices of odd degree.

Degree/Handshake Example

- The *degree* of a vertex v in V is the number of edges incident on v .

Let's label the degree of every node and calculate the sum...



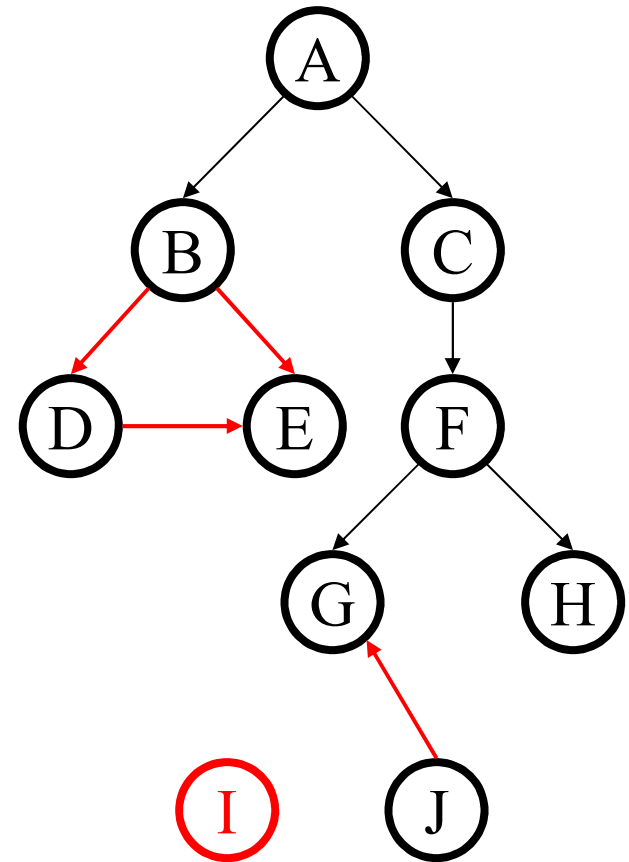
Degree for Directed Graphs

- The *in-degree* of a vertex v in V is denoted $\deg^-(v)$ is the number of edges coming *in* to v .
- The *out-degree* of a vertex v in V is denoted $\deg^+(v)$ is the number of edges coming *out* of v .
- We let $\deg(v) = \deg^+(v) + \deg^-(v)$
- Then:

$$\sum_{v \in V} \deg^-(v) = \sum_{v \in V} \deg^+(v) = \frac{1}{2} \sum_{v \in V} \deg(v) = |E|$$

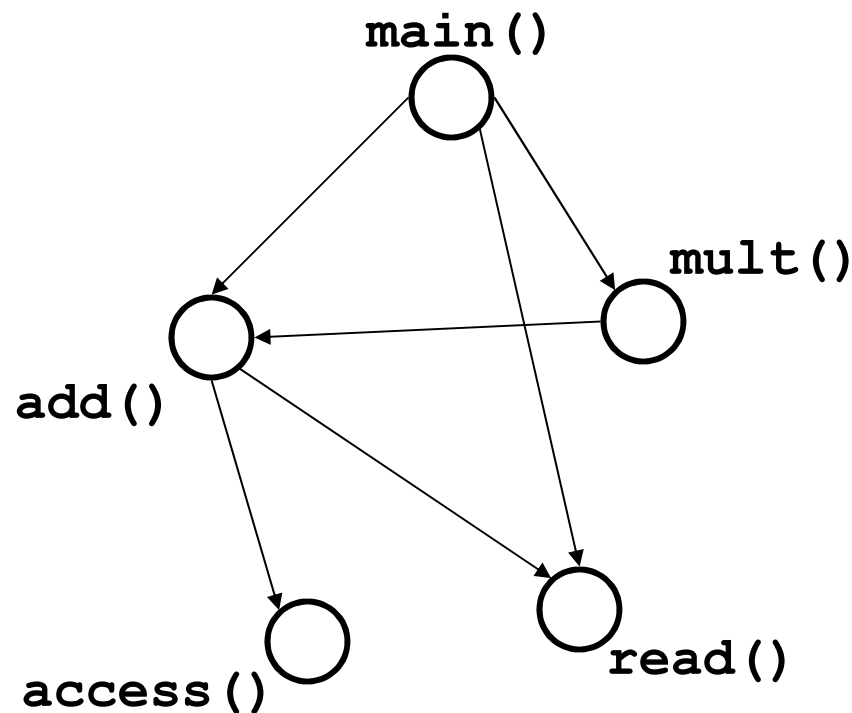
Trees as Graphs

- Every tree is a graph with some restrictions:
 - the tree is *directed*
 - there are *no cycles* (directed or undirected)
 - there is a *directed path from the root to every node*



Directed Acyclic Graphs (DAGs)

DAGs are directed graphs with no (directed) cycles.



Trees \subset DAGs \subset Graphs

We can only topo-sort DAGs!

Today's Outline

- Topological Sort: Getting to Know Graphs with a Sort
- Graph ADT and Graph Representations
- Graph Terminology (a lot of it!)
- More Graph Algorithms
 - Shortest Path (Dijkstra's Algorithm)
 - Minimum Spanning Tree (Kruskal's Algorithm)

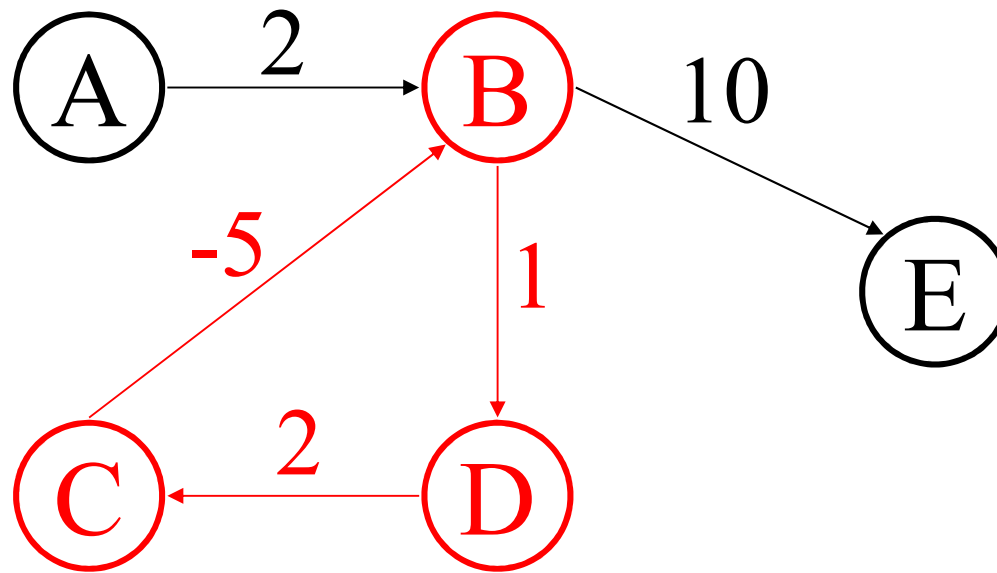
Single Source, Shortest Path

Given a graph $G = (V, E)$ and a vertex $s \in V$,
find the shortest path from s to every vertex in V

Many variations:

- weighted vs. unweighted
- cyclic vs. acyclic
- positive weights only vs. negative weights allowed
- multiple weight types to optimize

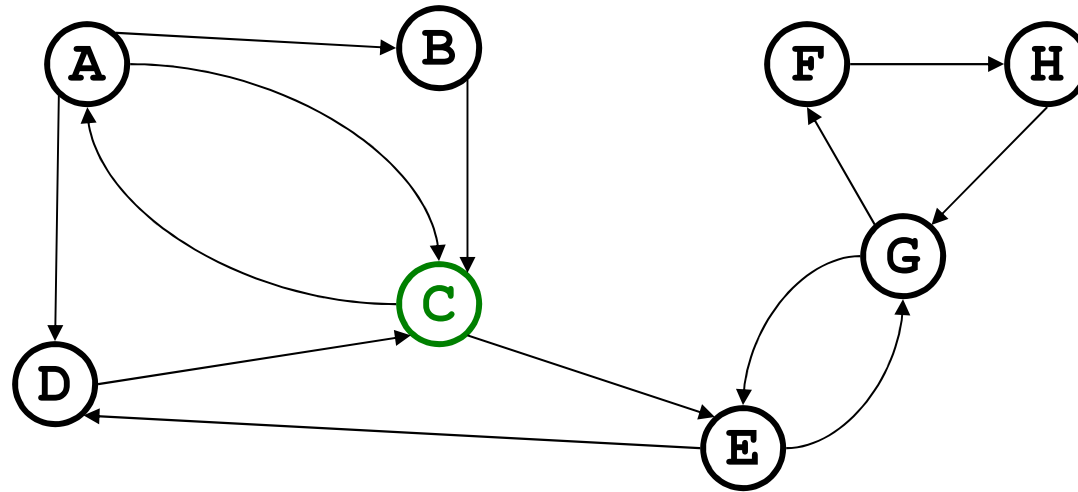
The Trouble with Negative Weighted Cycles



What's the shortest path from A to E?
(or to B, C, or D, for that matter)

Unweighted Shortest Path Problem

Assume source vertex is **C**...



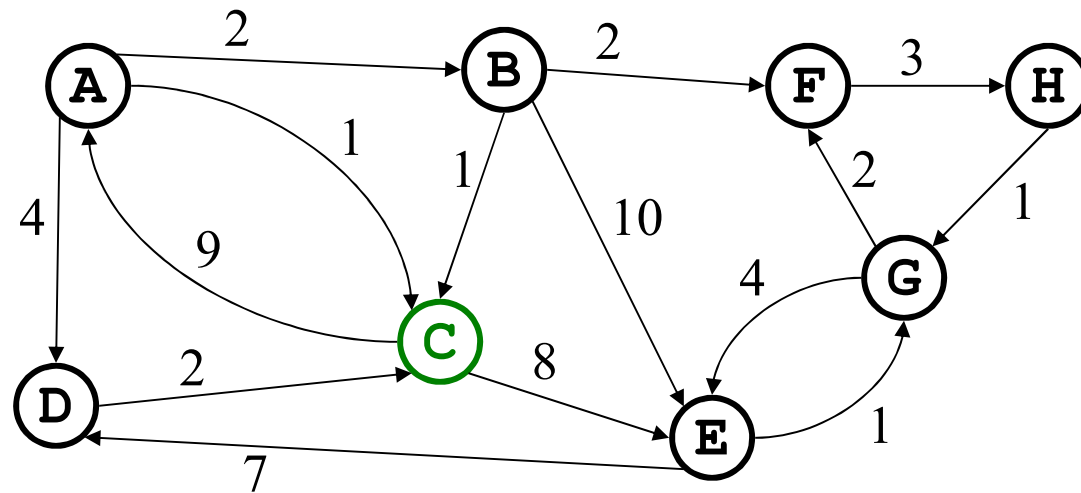
Distance to:

A B C D E F G H

Dijkstra's Algorithm for Single Source Shortest Path

- Classic algorithm for solving shortest path in weighted graphs without negative weights
- A *greedy* algorithm (irrevocably makes decisions without considering future consequences)
- Intuition:
 - shortest path from source vertex to itself is 0
 - cost of going to adjacent nodes is at most edge weights
 - cheapest of these must be shortest path to that node
 - update paths for new node and continue picking cheapest path

Intuition in Action



Dijkstra's Pseudocode

(actually, our pseudocode for Dijkstra's algorithm)

Initialize the cost of each node to ∞

Initialize the cost of the source to 0

While there are unknown nodes left in the graph

 Select the unknown node with the lowest cost: n

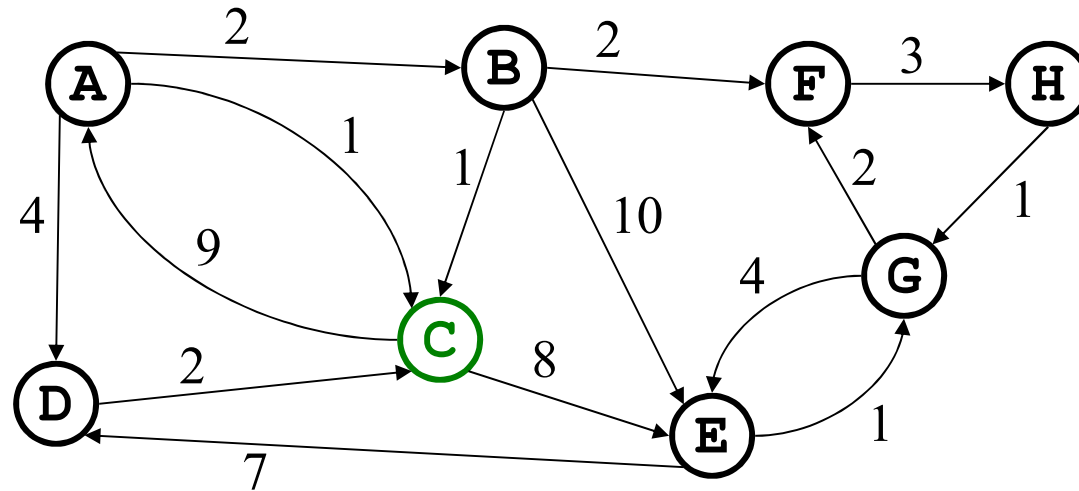
 Mark n as known

 For each node a which is adjacent to n

a 's cost = $\min(a$'s old cost, n 's cost + cost of (n, a))

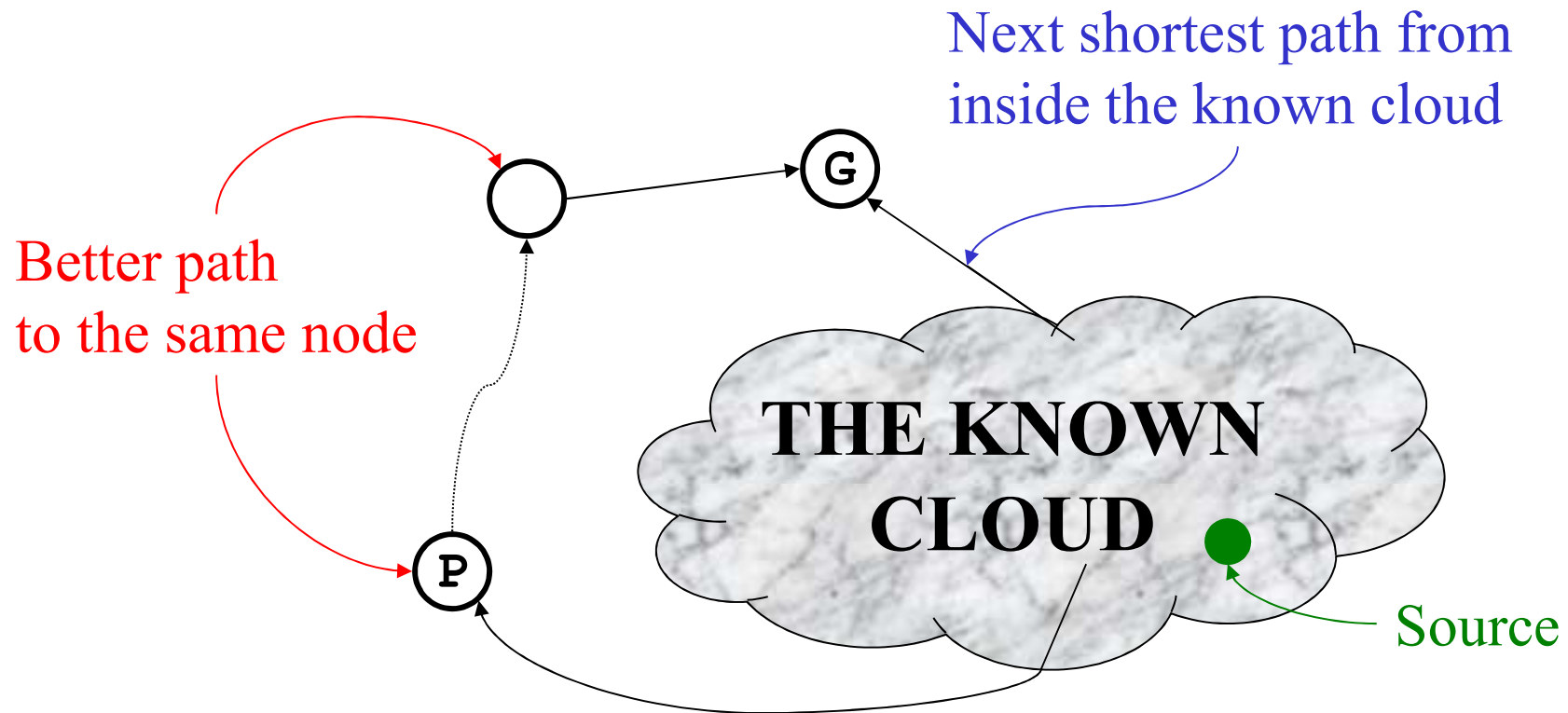
We can get the path from this
just as we did for mazes!

Dijkstra's Algorithm in Action



vertex	known	cost
A		
B		
C		
D		
E		
F		
G		
H		

The Cloud Proof



But, if the path to **G** is the next shortest path,
the path to **P** must be at least as long.

So, how can the path through **P** to **G** be shorter?

Inside the Cloud (Proof)

Everything inside the cloud has the correct shortest path

Proof is by induction on the # of nodes in the cloud:

- initial cloud is just the source with shortest path 0
- inductive step: once we prove the shortest path to G is correct, we add it to the cloud

Negative weights blow this proof away!

Inside the Cloud (Proof)

Everything inside the cloud has the correct shortest path

Proof is by induction on the # of nodes in the cloud:

- initial cloud is just the source with shortest path 0
- inductive step: once we prove the shortest path to G is correct, we add it to the cloud
- (Aside: The pseudocode was a while loop, and this is just a loop invariant proof...)

Negative weights blow this proof away!

Data Structures for Dijkstra's Algorithm

$|V|$ times:

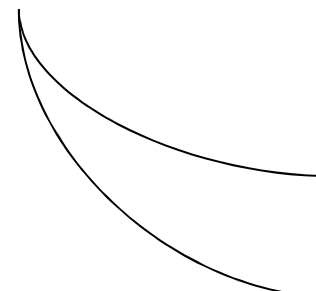
Select the unknown node with the lowest cost



findMin/deleteMin

$|E|$ times:

a 's cost = $\min(a$'s old cost, ...)



decreaseKey (i.e., change a key and
fix the heap)

find by name (dictionary lookup!)

runtime:

Today's Outline

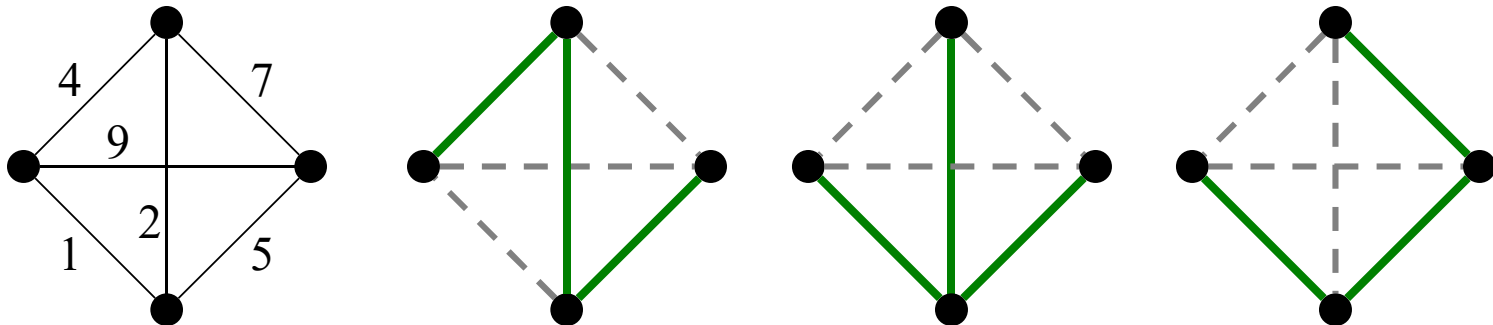
- Topological Sort: Getting to Know Graphs with a Sort
- Graph ADT and Graph Representations
- Graph Terminology (a lot of it!)
- **More Graph Algorithms**
 - Shortest Path (Dijkstra's Algorithm)
 - Minimum Spanning Tree (Kruskal's Algorithm)

Spanning Tree

Spanning tree: a subset of the edges from a connected graph that...

...touches all vertices in the graph (*spans* the graph)

...forms a tree (is connected and contains no cycles)



Minimum spanning tree: the spanning tree with the least total edge cost.

Kruskal's Algorithm for Minimum Spanning Trees

Yet another greedy algorithm:

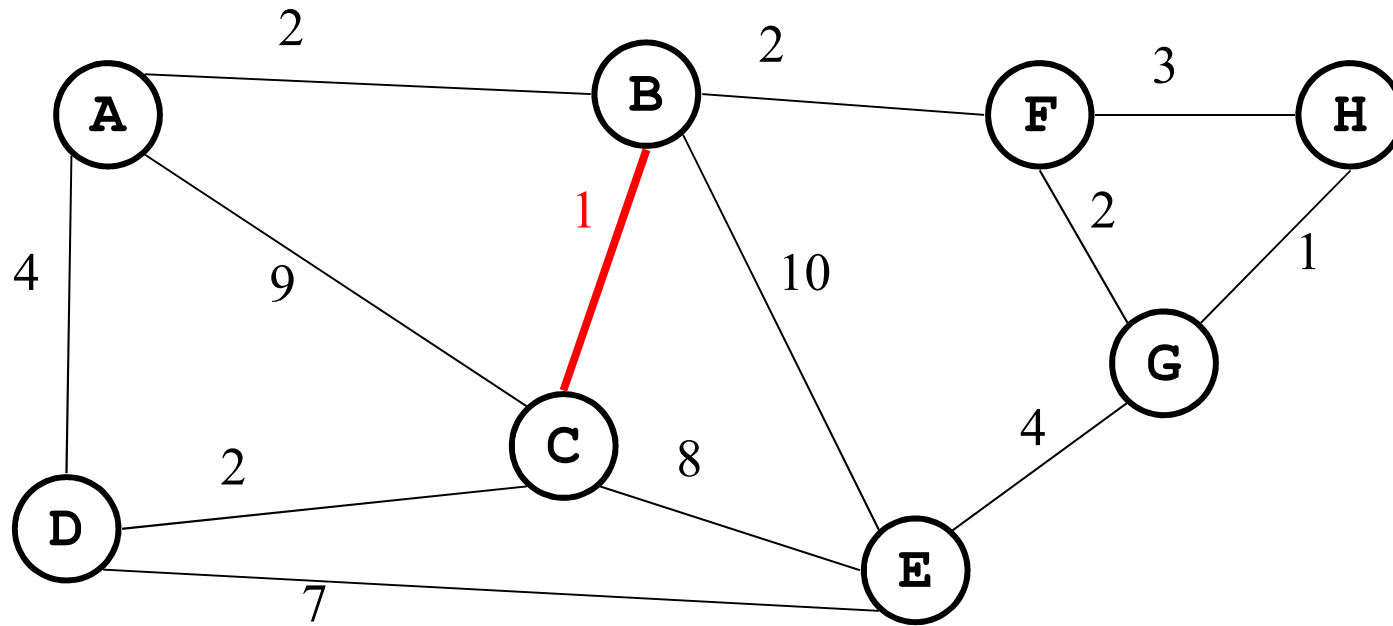
Initialize all vertices to their own sets (i.e. unconnected)

While there are still unmarked edges

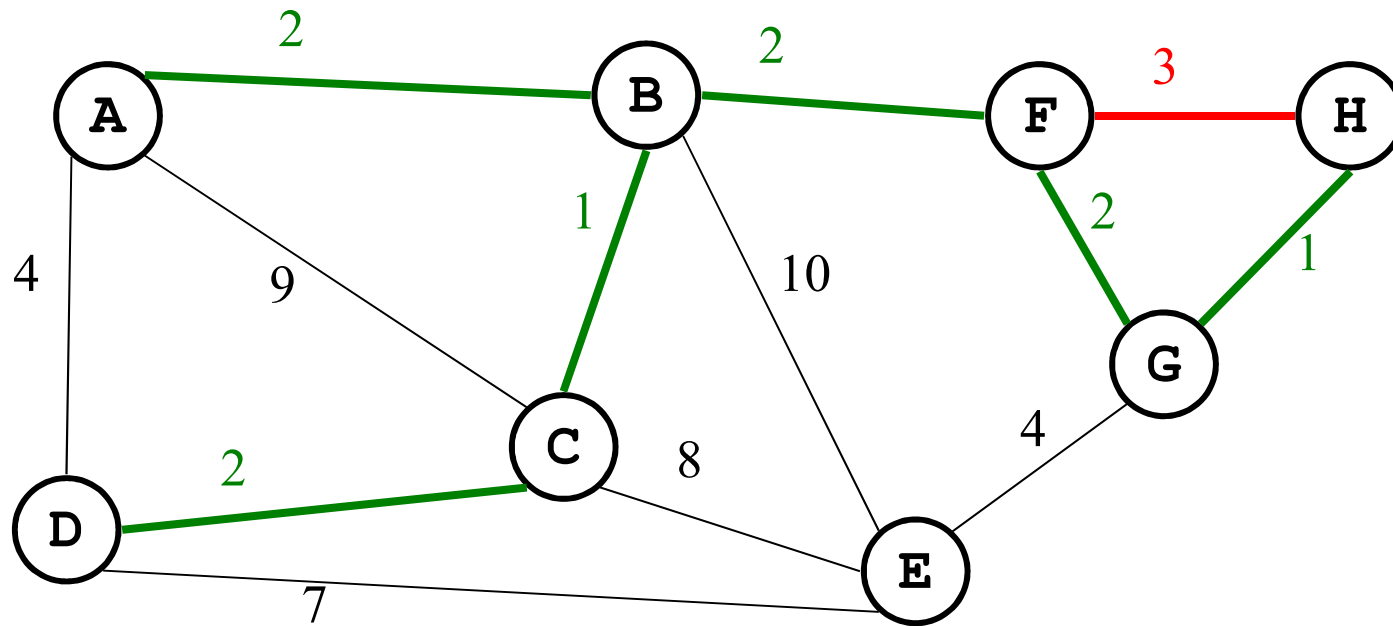
 Pick the lowest cost edge $\mathbf{e} = (\mathbf{u}, \mathbf{v})$ and mark it

 If \mathbf{u} and \mathbf{v} are in different sets, add \mathbf{e} to the minimum spanning tree and union the sets for \mathbf{u} and \mathbf{v}

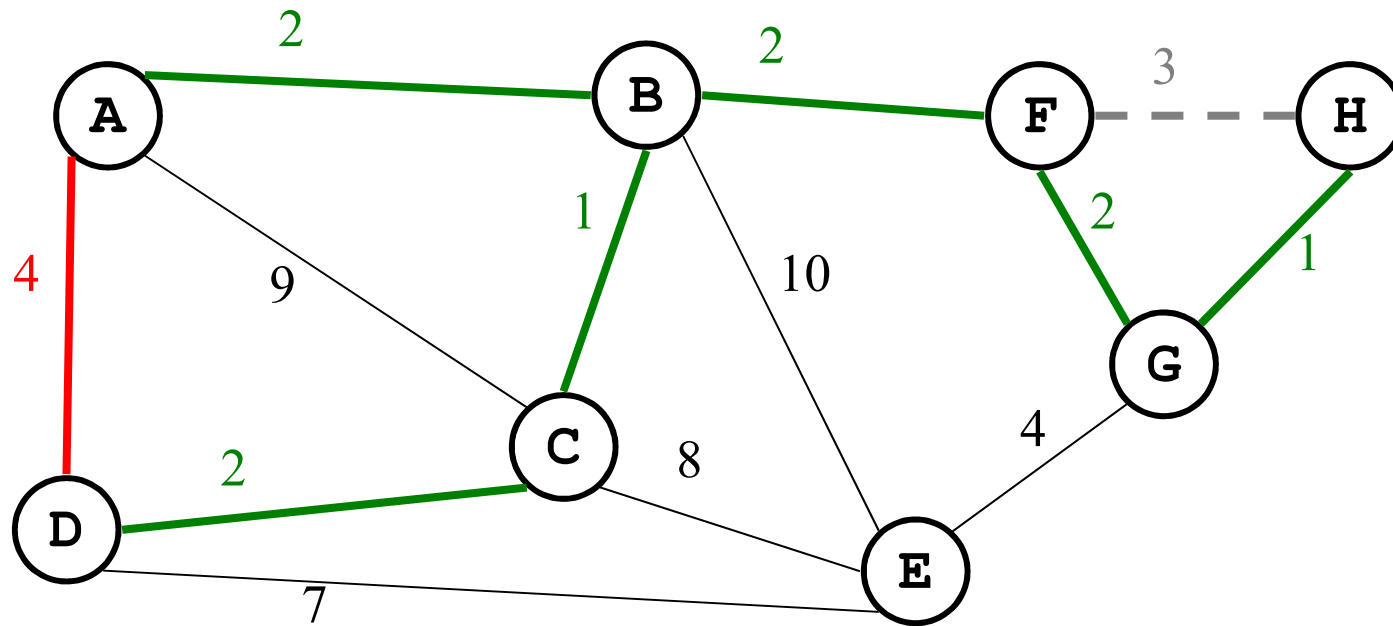
Kruskal's Algorithm in Action (1/5)



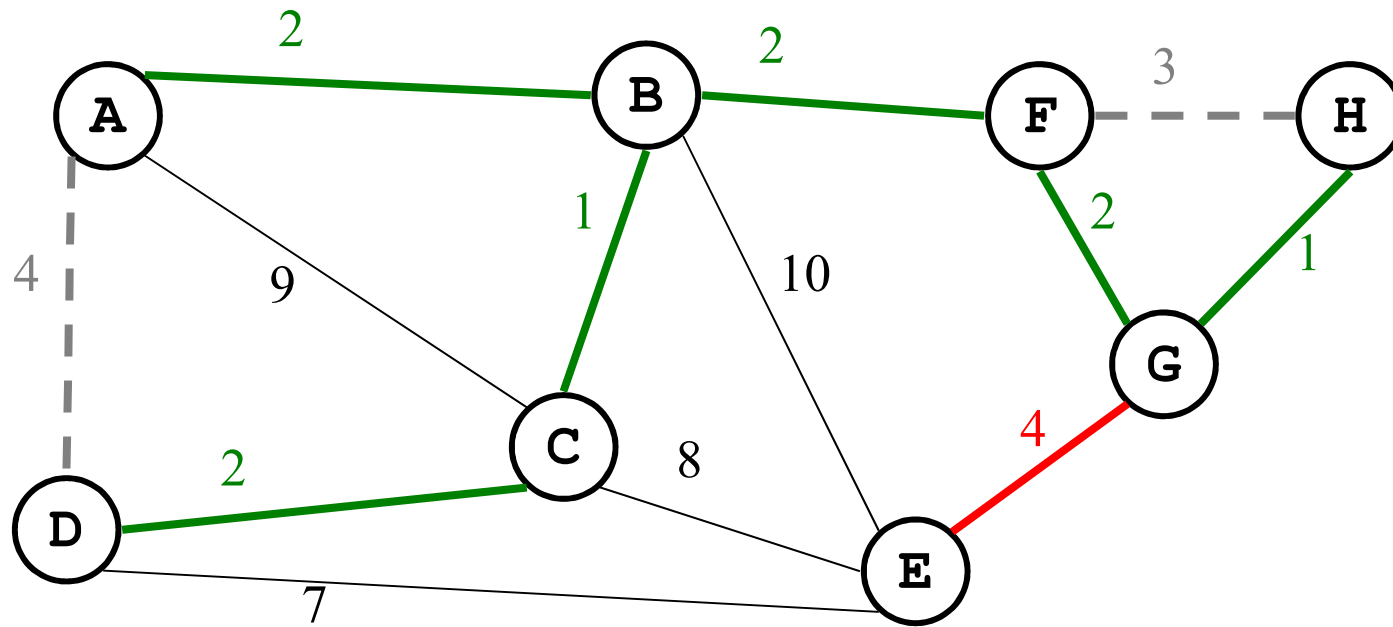
Kruskal's Algorithm in Action (2/5)



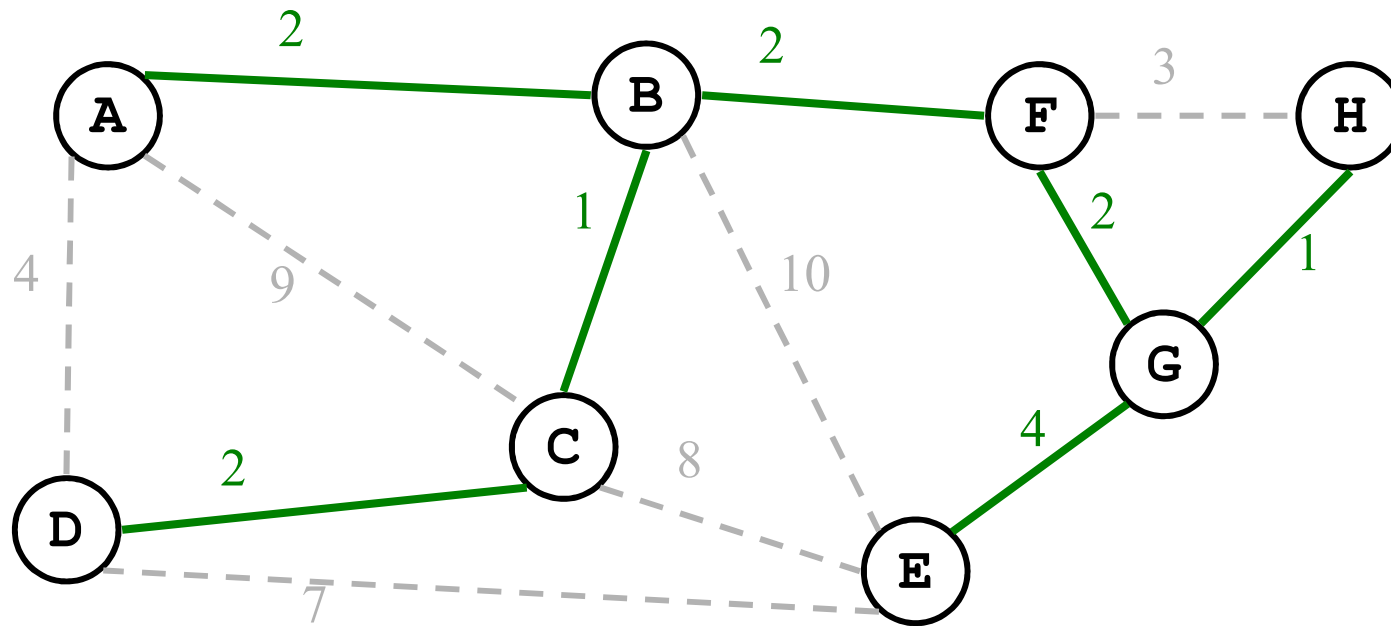
Kruskal's Algorithm in Action (3/5)



Kruskal's Algorithm in Action (4/5)



Kruskal's Algorithm Completed (5/5)



Does the algorithm work?

Warning!

- Proof in Epp (3rd p. 728) is slightly wrong.
- Wikipedia has a good proof.
 - That's basis of what I'll present.
 - It actually comes out naturally from how we've taught you to try to prove a program correct.

Kruskal's Algorithm:

Does this work?

Initialize all vertices to their own sets (i.e. unconnected)

Initialize all edges as unmarked.

While there are still unmarked edges

Pick the lowest cost unmarked edge $\mathbf{e} = (\mathbf{u}, \mathbf{v})$ and mark it.

If \mathbf{u} and \mathbf{v} are in different sets, add \mathbf{e} to the minimum spanning tree and union the sets for \mathbf{u} and \mathbf{v}

How have we learned to try to prove something like this?

Kruskal's Algorithm:

What's a good loop invariant???

Initialize all vertices to their own sets (i.e. unconnected)

Initialize all edges as unmarked.

While there are still unmarked edges

- Pick the lowest cost unmarked edge $\mathbf{e} = (\mathbf{u}, \mathbf{v})$ and mark it.

- If \mathbf{u} and \mathbf{v} are in different sets, add \mathbf{e} to the minimum spanning tree and union the sets for \mathbf{u} and \mathbf{v}

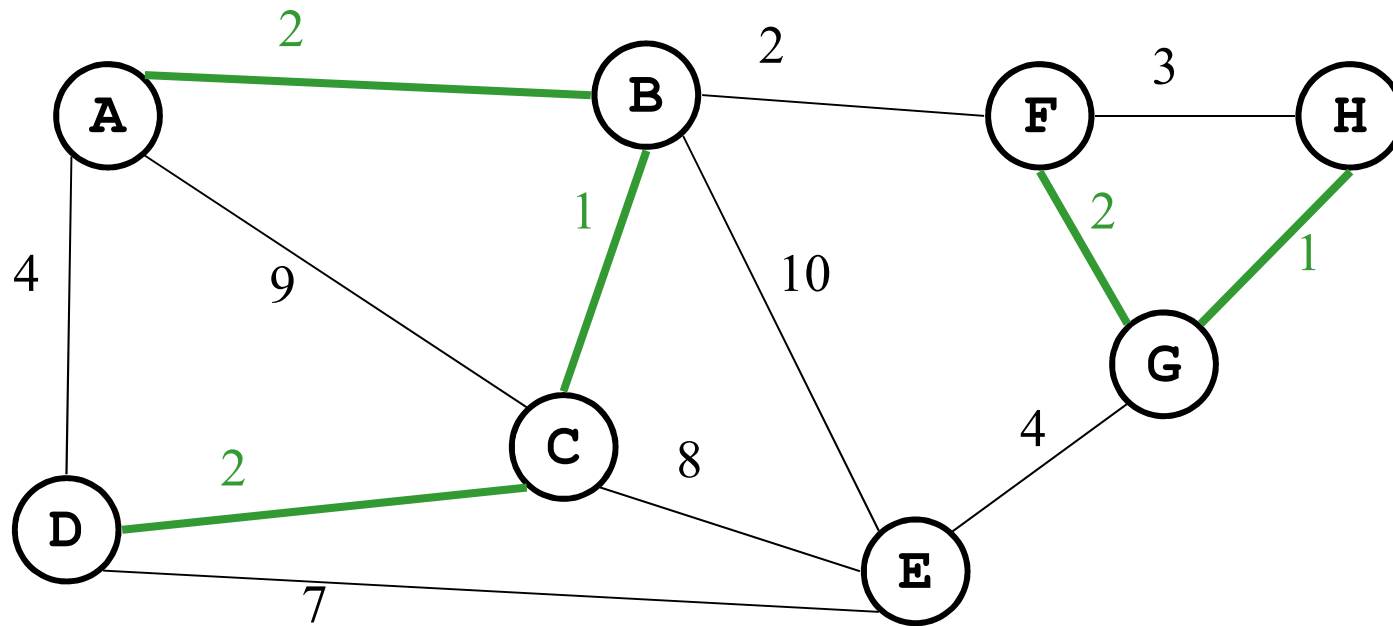
Loop Invariant for Kruskal's

- (There are lots of technical, detailed loop invariants that would be needed for a totally formal proof, e.g.:)
 - Each set is spanned by edges added to MST you are building.
 - Those edges form a tree.
 - ...
 - We will assume most of these without proof, if they are pretty obvious.

Loop Invariant for Kruskal's

- What do we know about the partial solution we're building up at each iteration?

Kruskal's Algorithm in Action (1.5/5)



Loop Invariant for Kruskal's

- What do we know about the partial solution we're building up at each iteration?
 - Since we're being greedy, we never go back and erase edges we add.
 - Therefore, for the algorithm to work, **whatever we've got so far must be part of some minimum spanning tree.**
 - That's the key to making the proof work!

Loop Invariant Proof for Kruskal's

- Candidate Loop Invariant:
 - **Whatever edges we've added at the start of each iteration are part of some minimum spanning tree.**

Loop Invariant Proof for Kruskal's

- Candidate Loop Invariant:
 - **Whatever edges we've added at the start of each iteration are part of some minimum spanning tree.**
- Base Case:
- Inductive Step:

Loop Invariant Proof for Kruskal's

- Candidate Loop Invariant:
 - **Whatever edges we've added at the start of each iteration are part of some minimum spanning tree.**
- Base Case:
 - When first arrive at the loop, the set of edges we've added is empty, so it's vacuously true. (We can't have made any mistakes yet, since we haven't picked any edges yet!)
- Inductive Step:

Loop Invariant Proof for Kruskal's

- Candidate Loop Invariant:
 - **Whatever edges we've added at the start of each iteration are part of some minimum spanning tree.**
- Base Case: Done!
- Inductive Step:
 - Assume that the loop invariant holds at start of loop body.
 - Want to prove that it holds the next time you get to start of loop body (which is also the “bottom of the loop”).

Loop Invariant Proof for Kruskal's

Inductive Step

- Candidate Loop Invariant:
 - **Whatever edges we've added at the start of each iteration are part of some minimum spanning tree.**
- Inductive Step:
 - Assume that the loop invariant holds at start of loop body.
 - Let F be the set of edges we've added so far.
 - Loop body has an if statement. Therefore, two cases!

Kruskal's Algorithm:

Initialize all vertices to their own sets (i.e. unconnected)

Initialize all edges as unmarked.

While there are still unmarked edges

Pick the lowest cost unmarked edge $\mathbf{e} = (\mathbf{u}, \mathbf{v})$ and mark it.

If \mathbf{u} and \mathbf{v} are in different sets, add \mathbf{e} to the minimum spanning tree and union the sets for \mathbf{u} and \mathbf{v}

Loop Invariant Proof for Kruskal's

Inductive Step

- Candidate Loop Invariant:
 - **Whatever edges we've added at the start of each iteration are part of some minimum spanning tree.**
- Inductive Step:
 - Assume that the loop invariant holds at start of loop body.
 - Let F be the set of edges we've added so far.
 - Loop body has an if statement. Therefore, two cases!
 - Case I: u and v are already in same set. Therefore, the edge is not needed in any spanning tree that includes the edges we have so far. Therefore, we throw out the edge, leave F unchanged, and loop invariant still holds.

Loop Invariant Proof for Kruskal's

Inductive Step

- Candidate Loop Invariant:
 - **Whatever edges we've added at the start of each iteration are part of some minimum spanning tree.**
- Inductive Step:
 - Assume that the loop invariant holds at start of loop body.
 - Let F be the set of edges we've added so far.
 - Loop body has an if statement. Therefore, two cases!
 - Case I: Done!
 - Case II: u and v are in different sets. We add the edge to F and merge the sets for u and v . This is the tricky case!

Loop Invariant Proof for Kruskal's

Inductive Step: Case II

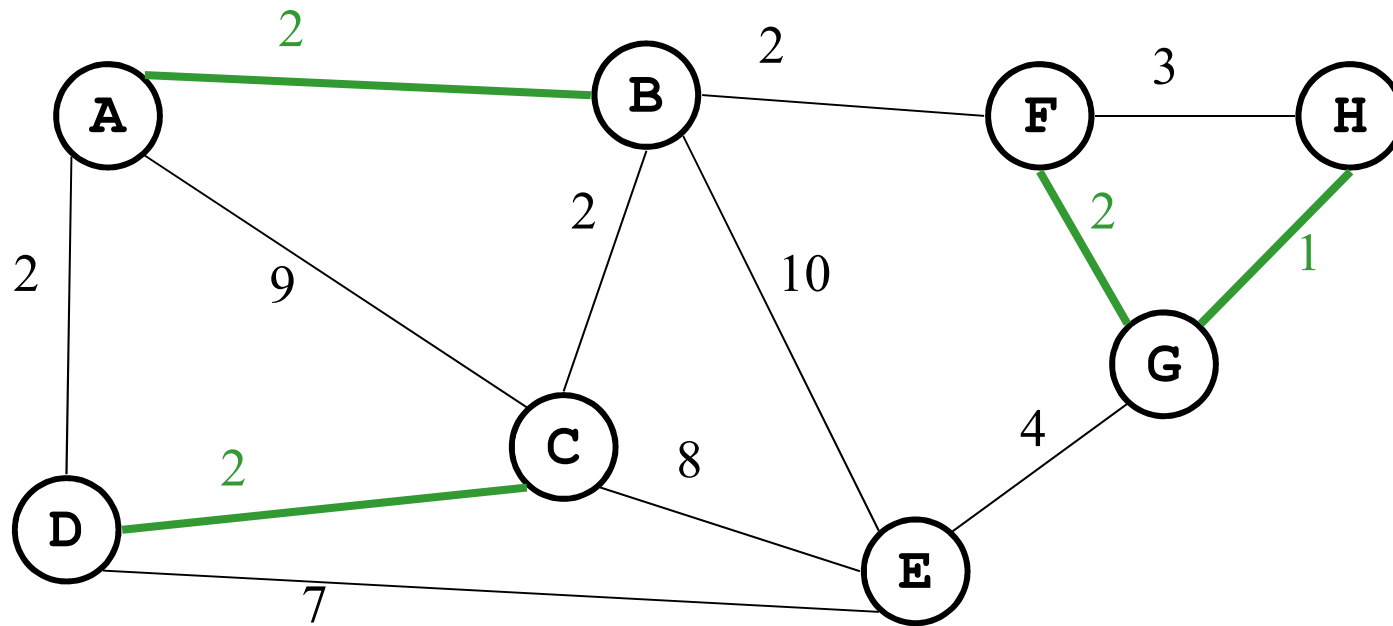
- Assume that the loop invariant holds at start of loop body.
- Let F be the set of edges we've added so far.
- Because loop invariant holds, there exists some MST T that includes all of F .
- The algorithm will pick a new edge e to add to F .
- Two Sub-Cases (of Case II)!
 - If e is in T , we add e to F and loop invariant still holds.
 - If e is not in T ,... This is tricky. We build a different MST from T that includes all of $F+e$...

Loop Invariant Proof for Kruskal's

Inductive Step: Case II-b

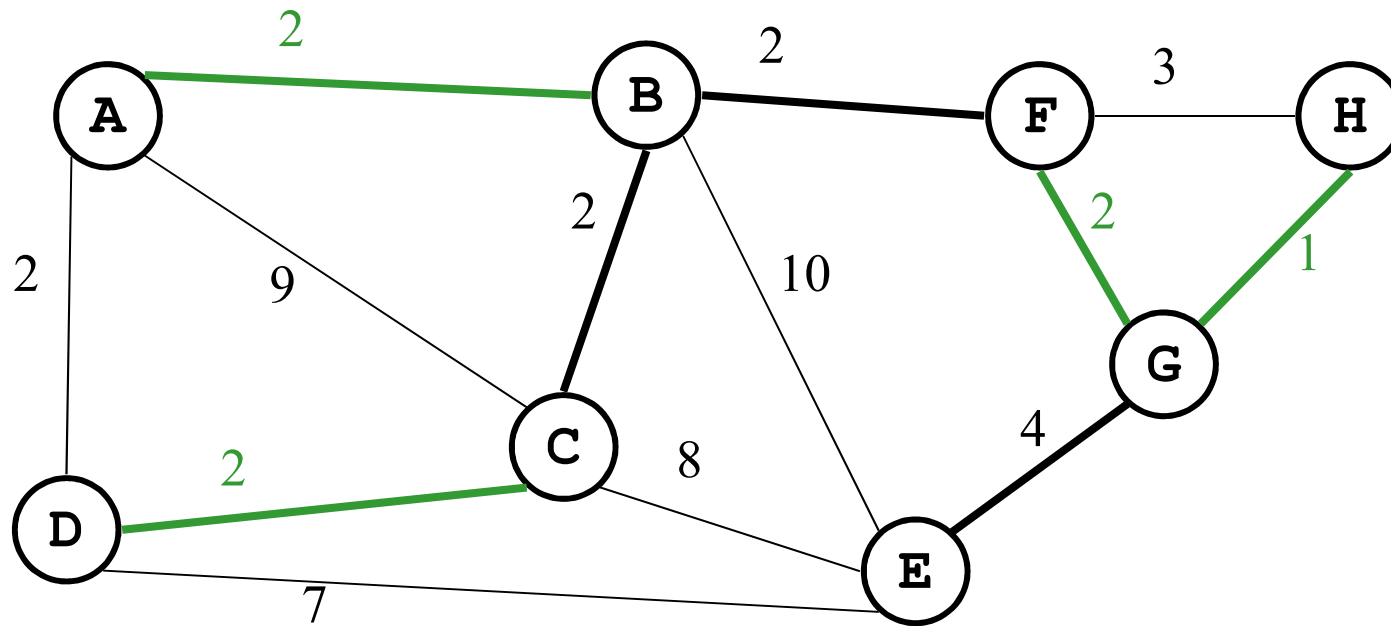
- Two Sub-Cases (of Case II)!
 - If e is in T , we add e to F and loop invariant still holds.
 - If e is not in T ,... This is tricky. We build a different MST from T that includes all of $F+e$...
 - If we add e to T , then $T+e$ must have a unique cycle C .
 - C must have a different edge f not in F . (Otherwise, adding e would have made a cycle in F .)
 - Therefore, $T+e-f$ is also a spanning tree.
 - If $w(f) < w(e)$, then Kruskal's would have picked f next, not e .
 - Therefore, $w(T+e-f) = W(T)$.
 - Therefore, $T+e-f$ is an MST that includes all of $F+e$
 - Loop invariant still holds.

Previous Example (Slightly Modified) to Show Proof Step



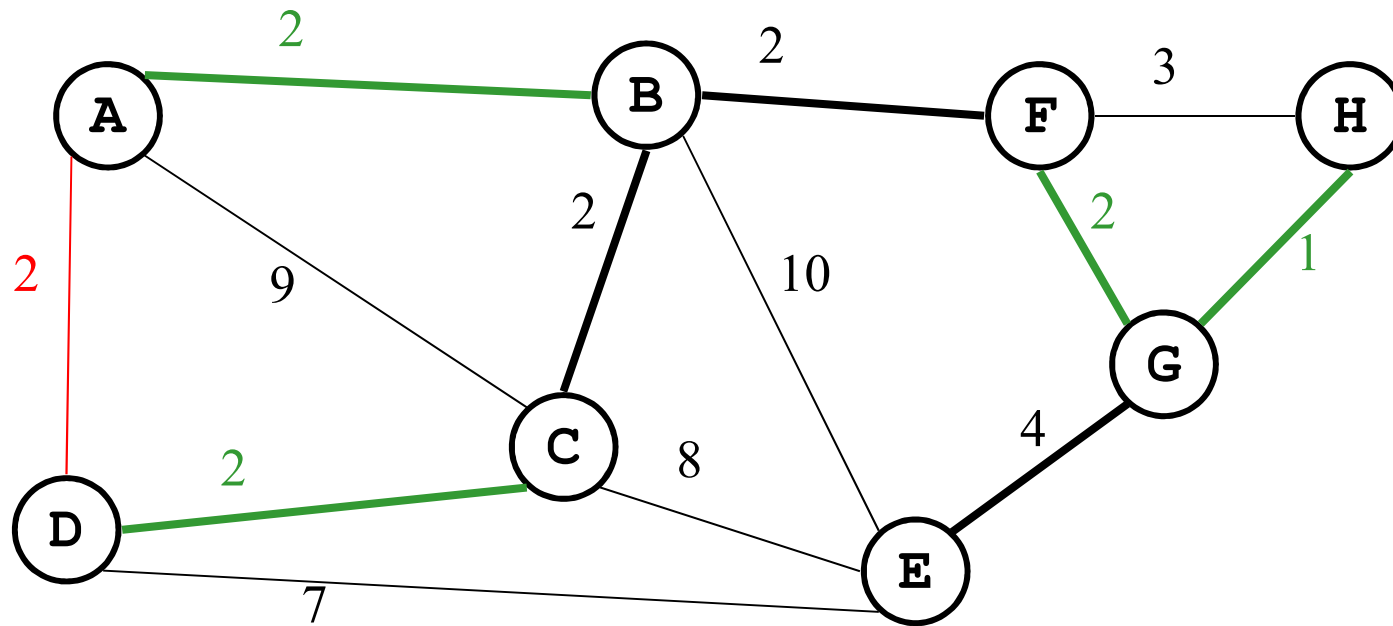
Before loop, F is the green edges.

Previous Example (Slightly Modified) to Show Proof Step



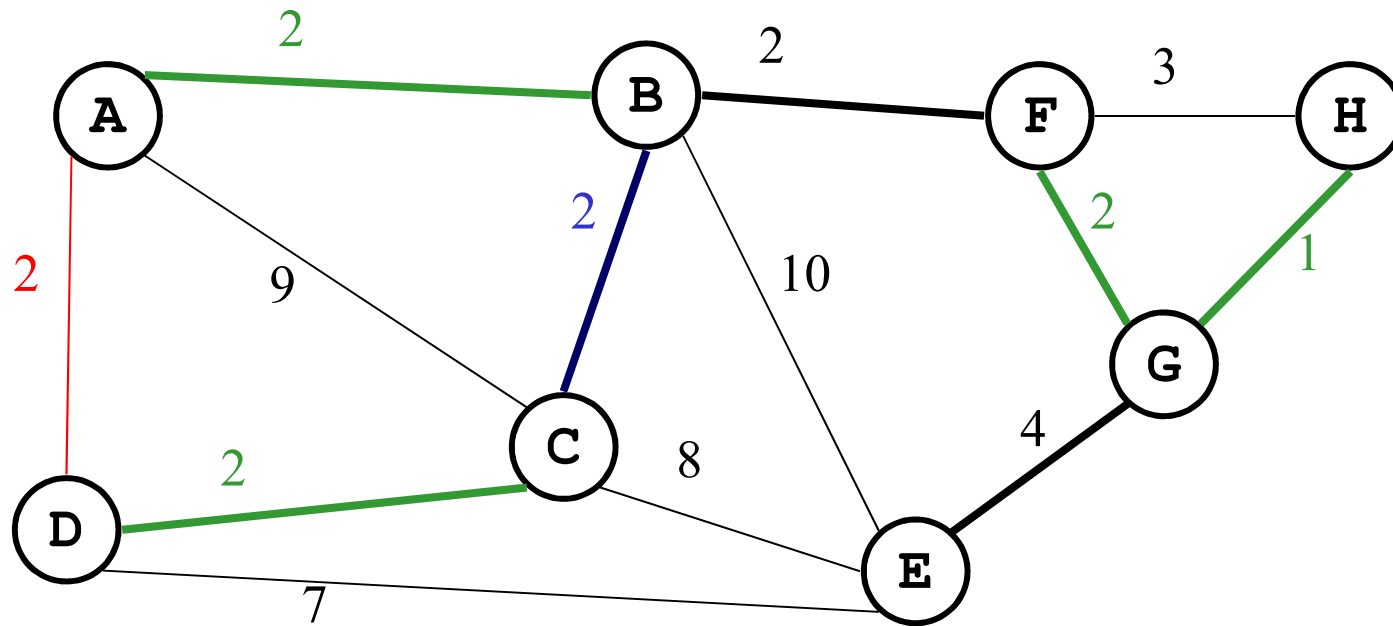
There exists an MST T that extends F
(e.g., the fat edges)

Previous Example (Slightly Modified) to Show Proof Step



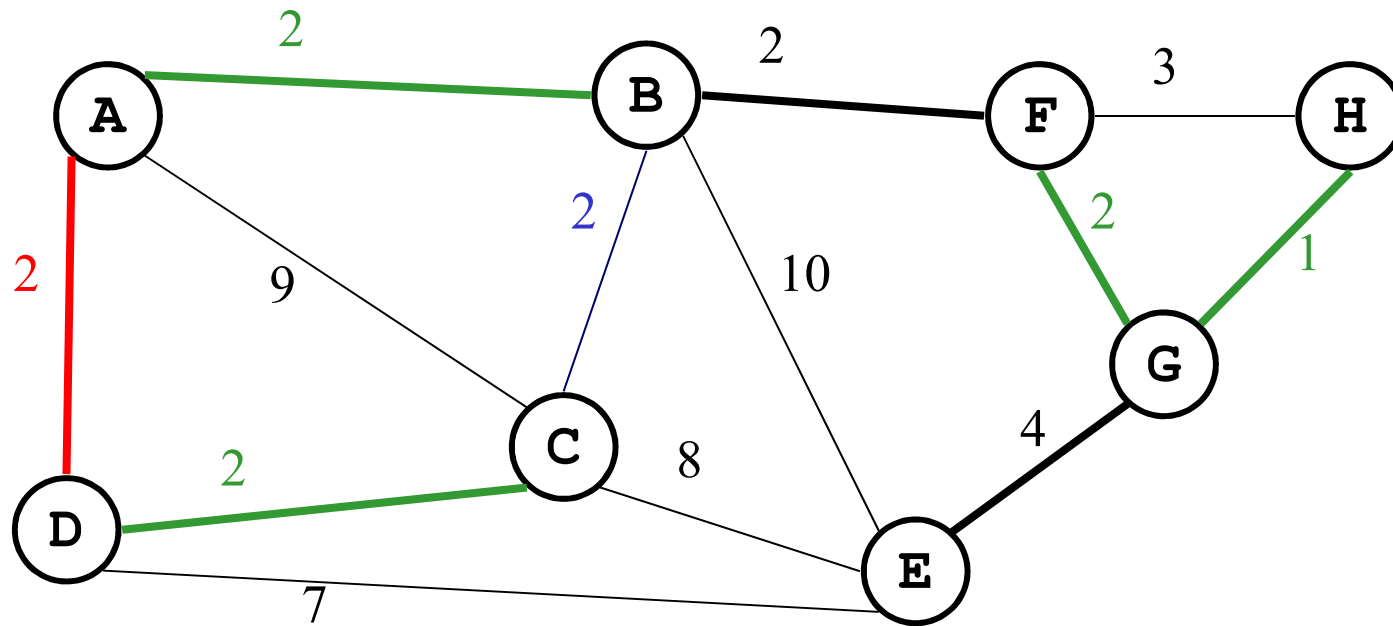
What if we pick e (red edge) that is not part of T ?
Then $T+e$ has a cycle...

Previous Example (Slightly Modified) to Show Proof Step



What if we pick e (red edge) that is not part of T ?
Then $T+e$ has a cycle, and the cycle includes an edge f not in F (blue edge).

Previous Example (Slightly Modified) to Show Proof Step



$w(e)$ must be less than or equal to $w(f)$

Therefore, $T+e-F$ is also an MST, but it includes all of $F+e$.

Data Structures for Kruskal's Algorithm

$|E|$ times:

Pick the lowest cost edge...

→ findMin/deleteMin

$|E|$ times:

If u and v are not already connected...

...connect u and v .

→ find representative

→ union

With “disjoint-set” data structure, $|E| \lg(|E|)$ runtime.

Learning Goals

After this unit, you should be able to:

- Describe the properties and possible applications of various kinds of graphs (e.g., simple, complete), and the relationships among vertices, edges, and degrees.
- Prove basic theorems about simple graphs (e.g. handshaking theorem).
- Convert between adjacency matrices/lists and their corresponding graphs.
- Determine whether two graphs are isomorphic.
- Determine whether a given graph is a subgraph of another.
- Perform breadth-first and depth-first searches in graphs.
- Explain why graph traversals are more complicated than tree traversals.

Wrong Proofs

- Skip these if you find them confusing. (Continue with efficiency.)
- It's hard to give a “counterexample”, since the algorithm is correct. I will try to show why certain steps in the proof aren't guaranteed to work as claimed.
- What goes wrong is that the proofs start from the finished result of Kruskal's, so it's hard to specify correctly which edge needs to get swapped.

Old (Wrong) Proof of Correctness

We already know this finds a spanning tree.

Proof by contradiction that Kruskal's finds the minimum:

Assume another spanning tree has *lower cost* than Kruskal's

Pick an edge $\mathbf{e}_1 = (\mathbf{u}, \mathbf{v})$ in that tree that's *not* in Kruskal's

Kruskal's tree connects \mathbf{u} 's and \mathbf{v} 's sets with another edge \mathbf{e}_2

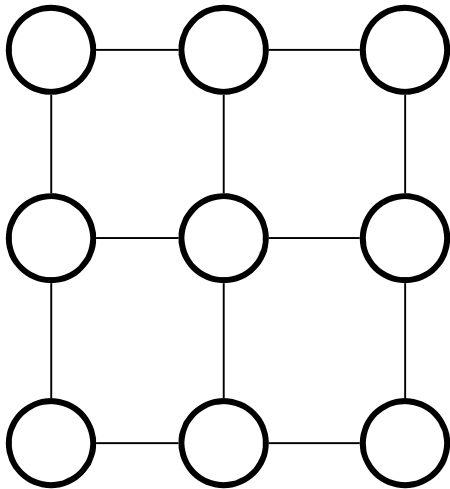
But, \mathbf{e}_2 must have at most the same cost as \mathbf{e}_1 (or Kruskal's would have found and used \mathbf{e}_1 first to connect \mathbf{u} 's and \mathbf{v} 's sets)

So, swap \mathbf{e}_2 for \mathbf{e}_1 (at worst keeping the cost the same)

Repeat until the tree is identical to Kruskal's: **contradiction!**

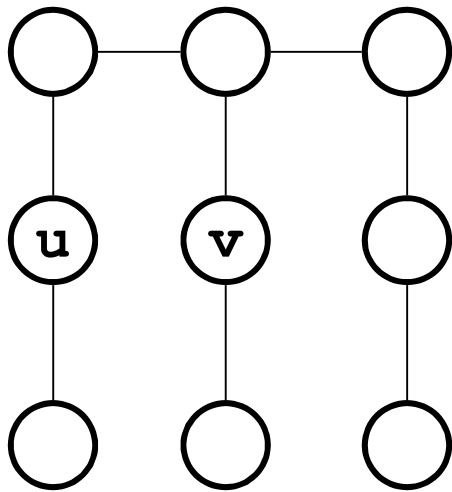
QED: Kruskal's algorithm finds a minimum spanning tree.

Counterexample Graph

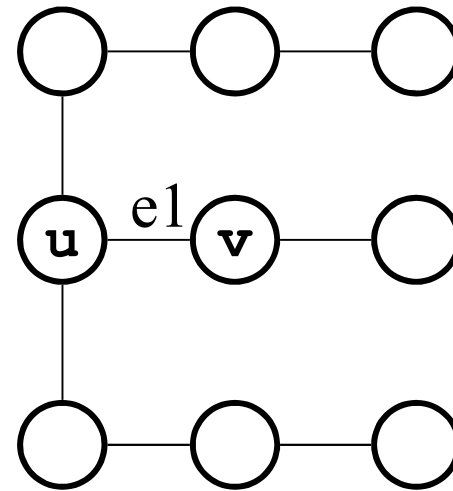


- Assume the graph is shaped like this.
- Ignore the details of edge weights. (E.g., they might all be equal or something.)

Counterexample Old Proof



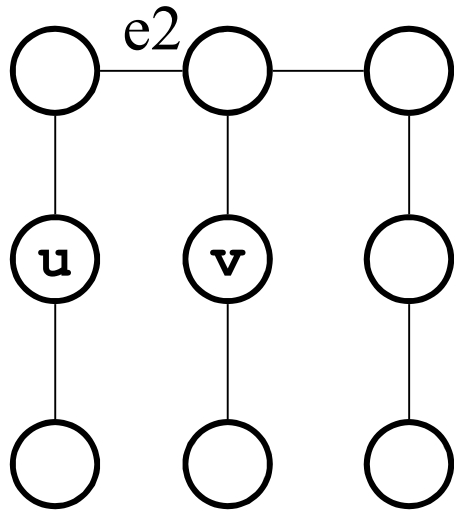
Kruskal's Result



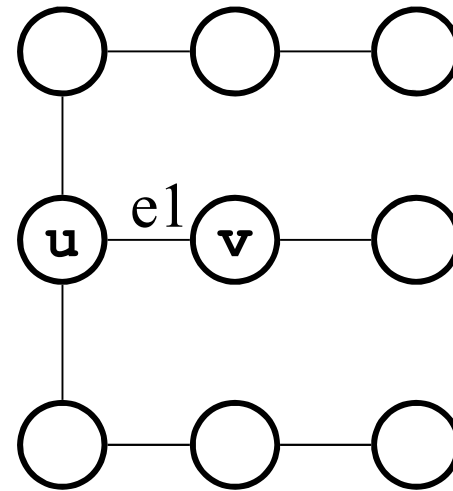
Other MST

The proof assumes some other MST and picks an edge e_1 connecting vertices u and v that's not in Kruskal's result.

Counterexample Old Proof



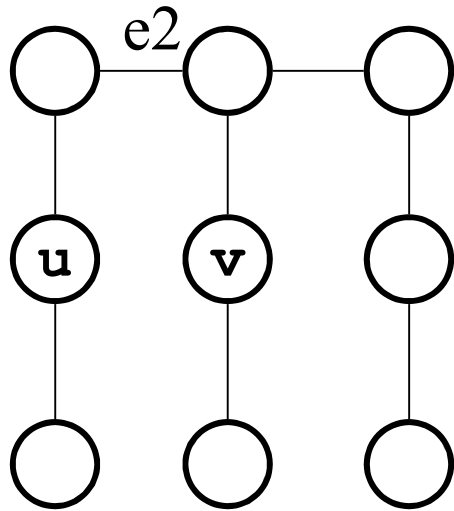
Kruskal's Result



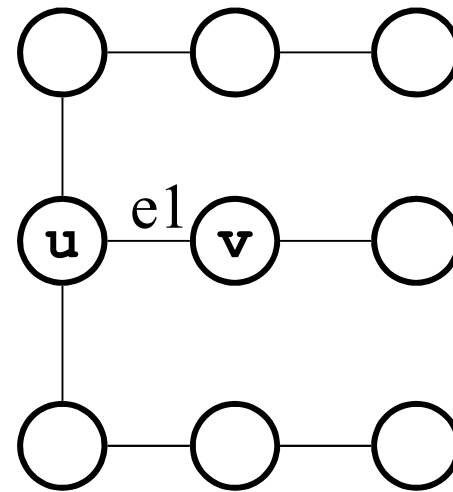
Other MST

In Kruskal's result, the sets for u and v were connected at some point by some edge $e2$. Let's suppose it was the edge shown (since we don't know when those components were connected). $w(e2) \leq w(e1)$ or else Kruskal's would have picked $e1$.

Counterexample Old Proof



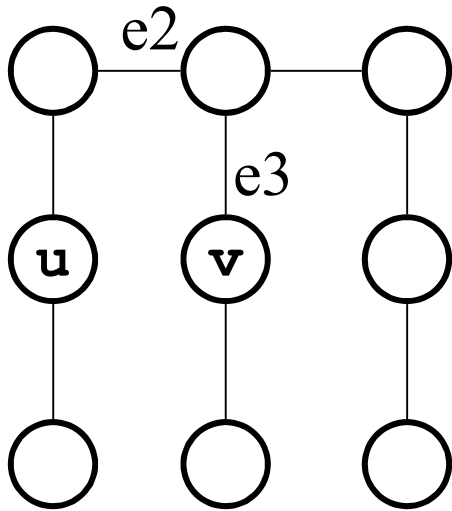
Kruskal's Result



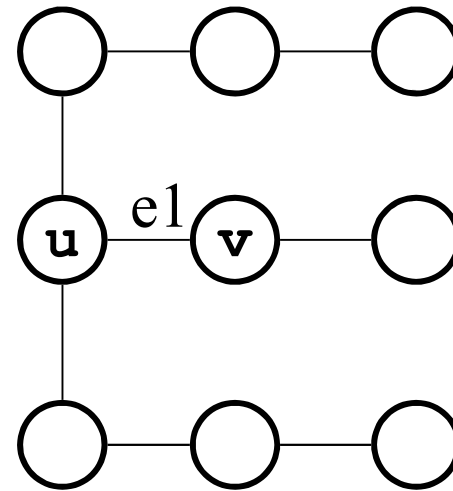
Other MST

The old wrong proof then says to swap $e2$ for $e1$ in the other MST. But we can't do it, because $e2$ is already in the other MST! So, the proof is wrong, as it is relying on an illegal step.

Fixing Old Proof



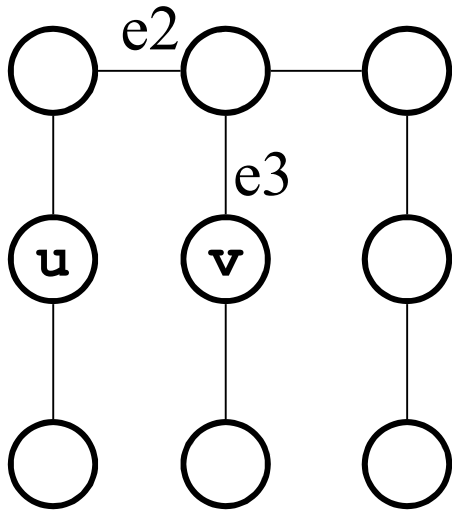
Kruskal's Result



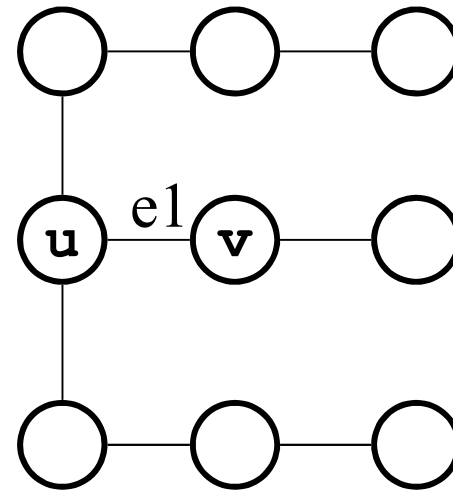
Other MST

To fix the proof, note that adding $e1$ to Kruskal's creates a cycle. Some other edge $e3$ on that cycle must be in Kruskal's but not the other MST (otherwise, other MST would have had a cycle).

Fixing Old Proof



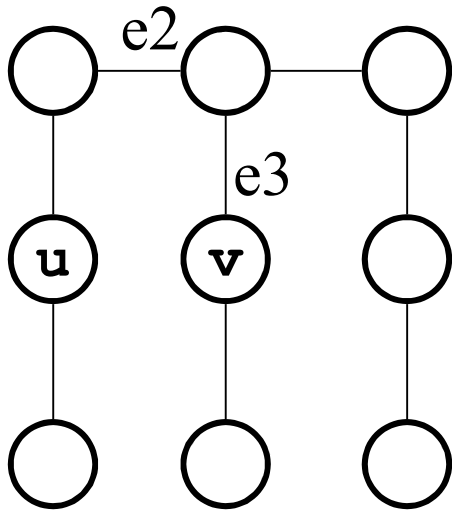
Kruskal's Result



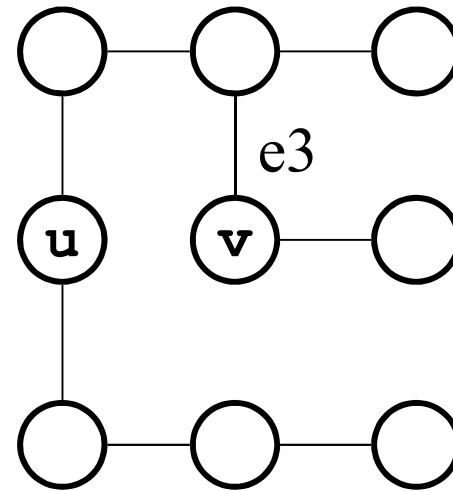
Other MST

We already know $w(e2) \leq w(e1)$, or Kruskal would have had $e1$.
Now, note that $e2$ was the edge that merged u and v 's sets.
Therefore, $w(e3) \leq w(e2)$, because Kruskal added it earlier.
So, $w(e3) \leq w(e2) \leq w(e1)$.

Fixing Old Proof



Kruskal's Result

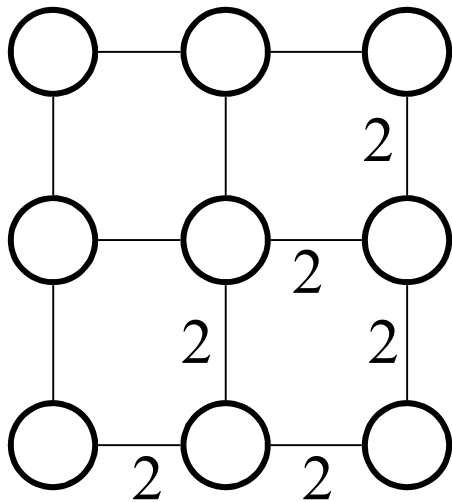


Other MST

So, $w(e3) \leq w(e2) \leq w(e1)$.

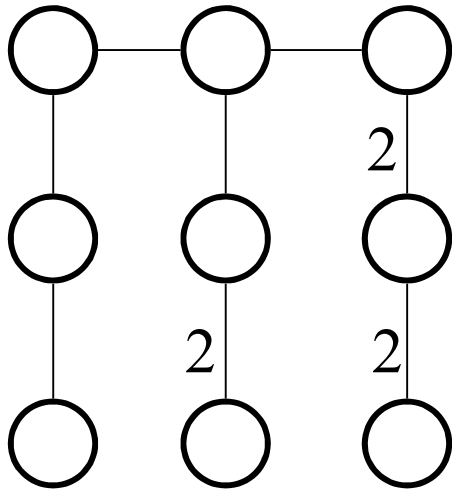
Therefore, we can swap $e3$ for $e1$ in the other MST, making it one edge closer to Kruskal's, and continue with the old proof. 😊

Counterexample for Epp's Proof

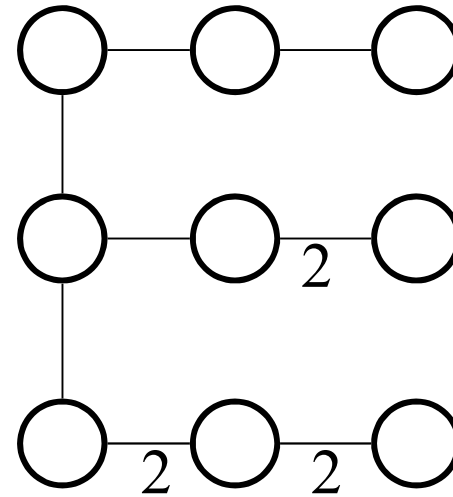


- Assume the graph is shaped like this.
- In this case, I've got an actual counterexample, with specific weights.
- Assume all edges have weight 1, except for the marked edges with weight 2.

Counterexample Epp's Proof



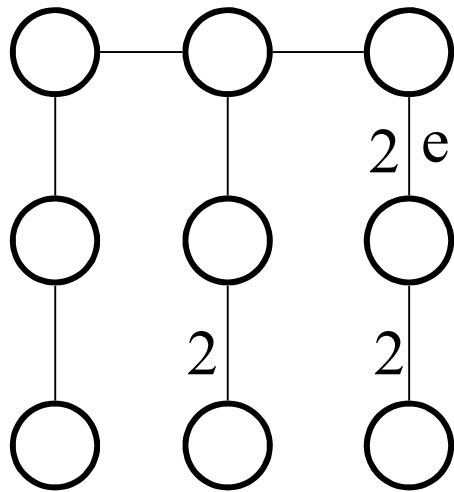
Kruskal's Result T



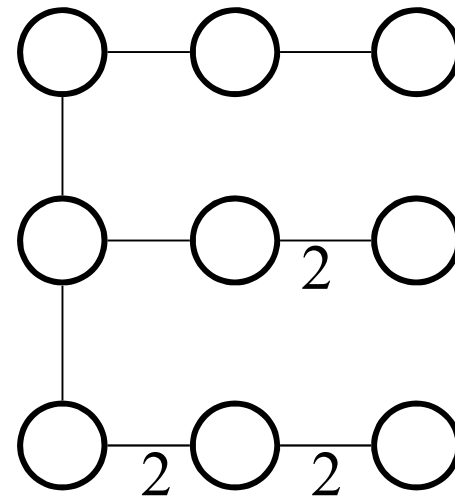
Other MST T1

Epp's proof (3rd edition, pp. 727-728) also starts with Kruskal's result (she calls it T) and some other MST, which she calls T1. She tries to show that for any other T1, you can convert it into T by a sequence of swaps that doesn't change the weight.

Counterexample Epp's Proof



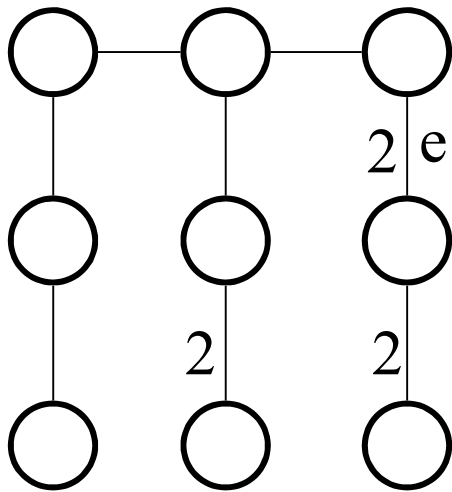
Kruskal's Result T



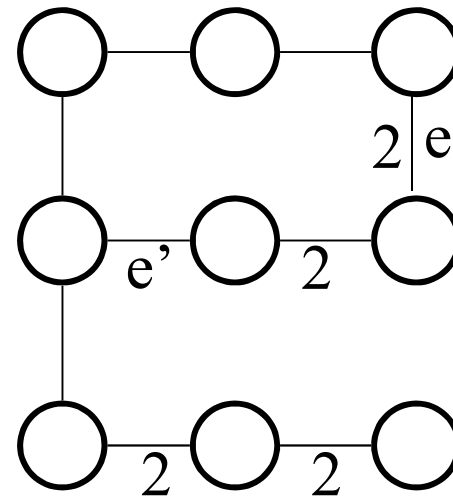
Other MST T1

If T is not equal to T1, there exists an edge e in T, but not in T1.
This could be the edge shown.

Counterexample Epp's Proof



Kruskal's Result T



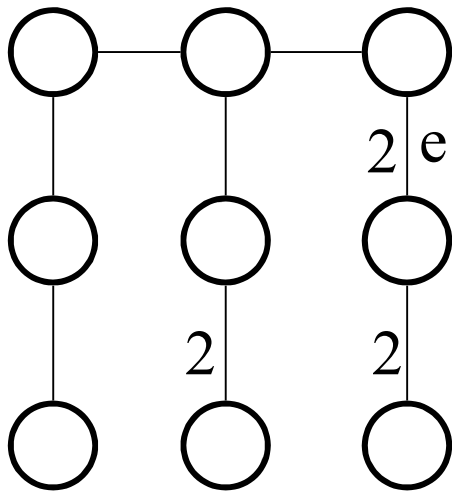
Other MST T1

Adding e to $T1$ produces a unique “circuit”. She then says,

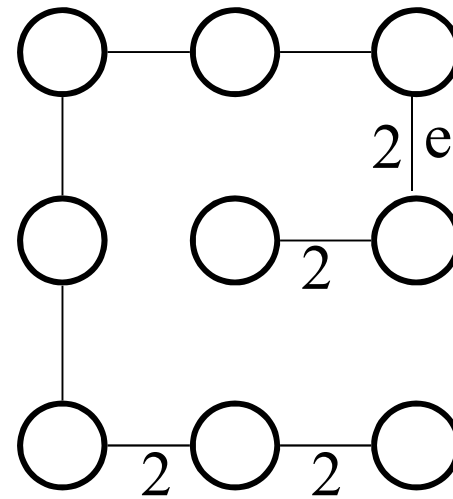
“Let e' be an edge of this circuit such that e' is not in T .”

OK, so this could be the labeled edge e' of the cycle that is not in T .

Counterexample Epp's Proof



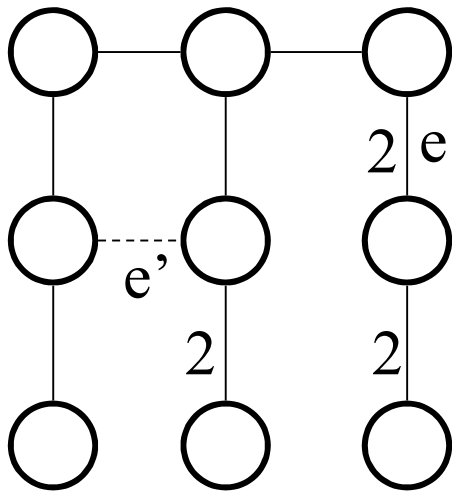
Kruskal's Result T



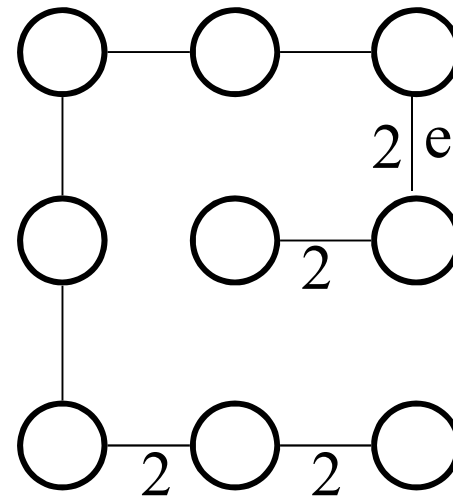
Other MST T2

Next, she creates T2 by adding e to T1 and deleting e'.
I am showing T2 above. Note, however, that we've added an edge with weight 2 and deleted an edge with weight 1! T2 has higher weight (12) than T1 did (11). The proof is wrong!

Counterexample Epp's Proof



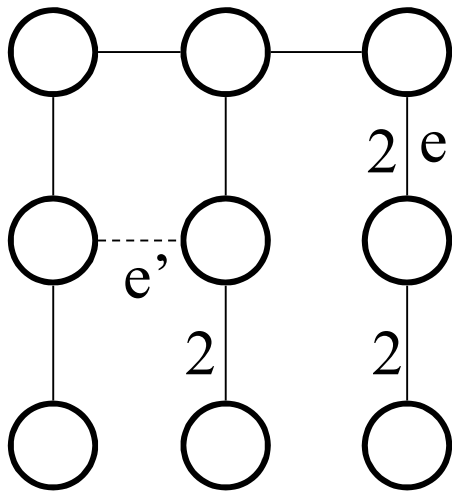
Kruskal's Result T



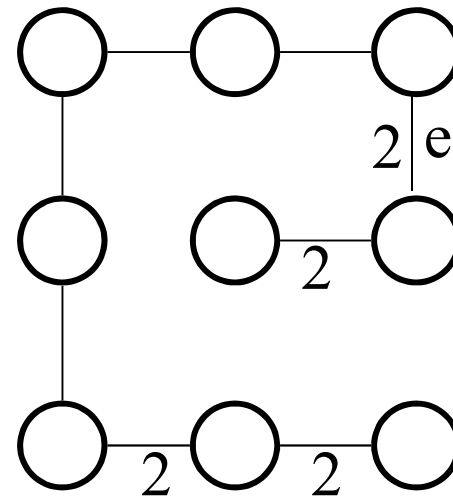
Other MST T2

It's interesting to read the wrong justification given in the proof that $w(e)=2$ has to be less than $w(e')=1$. "...at the stage in Kruskal's algorithm when e was added to T , e' was available to be added [since ... at that stage its addition could not produce a circuit...]" Oops!

Counterexample Epp's Proof



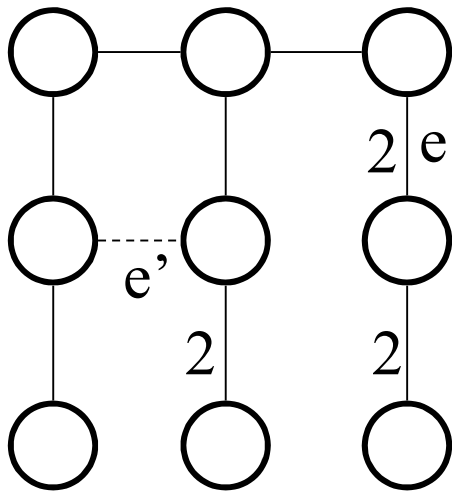
Kruskal's Result T



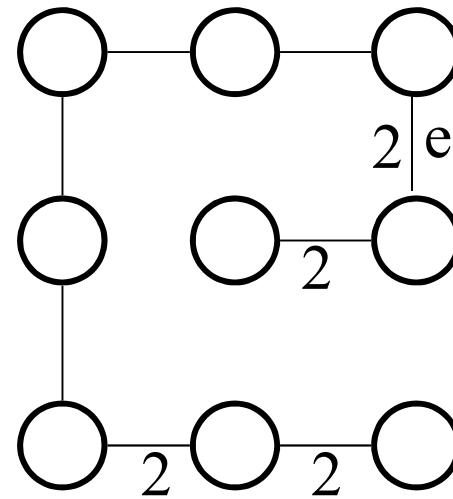
Other MST T2

I don't see an easy fix for her proof. It might be possible to show that there must be a suitable edge with sufficiently large weight. The hard part is that you have to reason back to how Kruskal's algorithm *could* have done something, after the fact!

Counterexample Epp's Proof



Kruskal's Result T



Other MST T2

See how much easier it was to do the proof with loop invariants?!
You prove what you need at exactly the point in the algorithm when you are making decisions, so you know exactly what edge e gets added and what edge f gets deleted.

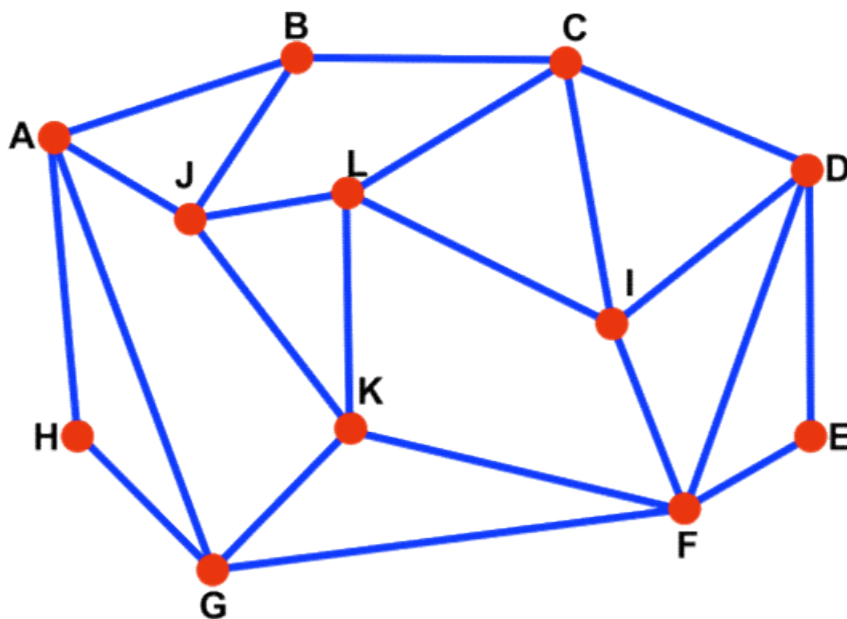
Some Extra Examples, etc.

Bigger (Undirected) Formal Graph Example

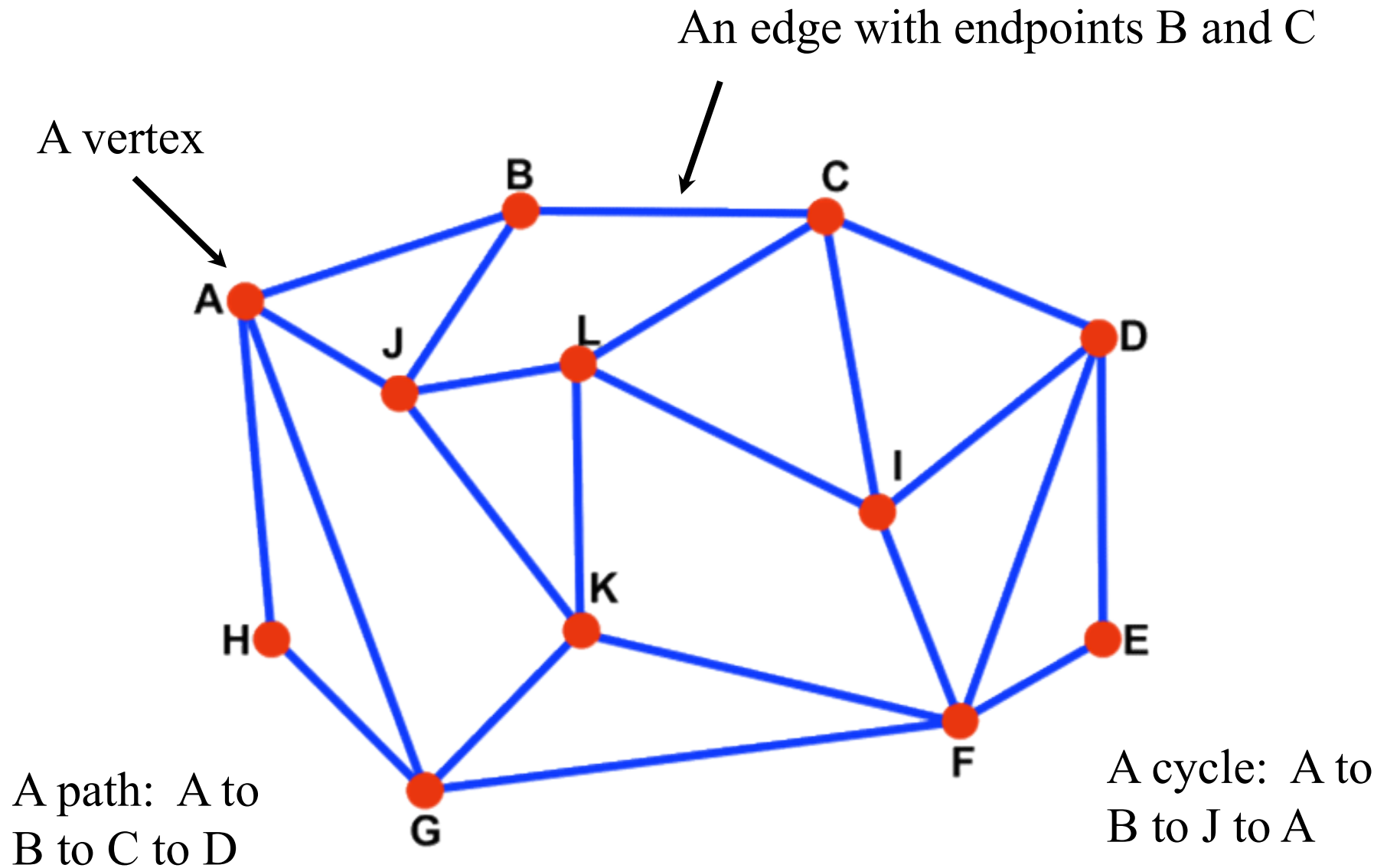
$$G = \langle V, E \rangle$$

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

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



(A *simple graph* like this one is undirected, has 0 or 1 edge between each pair of vertices, and no edge from a vertex to itself.)



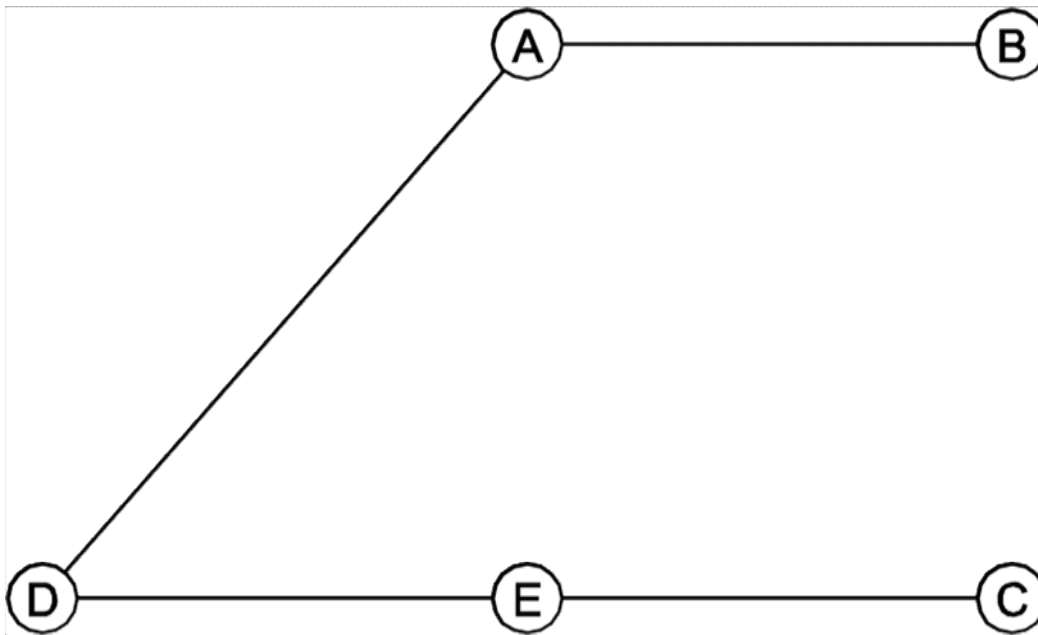
A *path* is a list of vertices $\{v_1, v_2, \dots, v_n\}$ such that $(v_i, v_{i+1}) \in E$ for all $0 \leq i < n$.

A *cycle* is a path that starts and ends at the same vertex.

Example

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

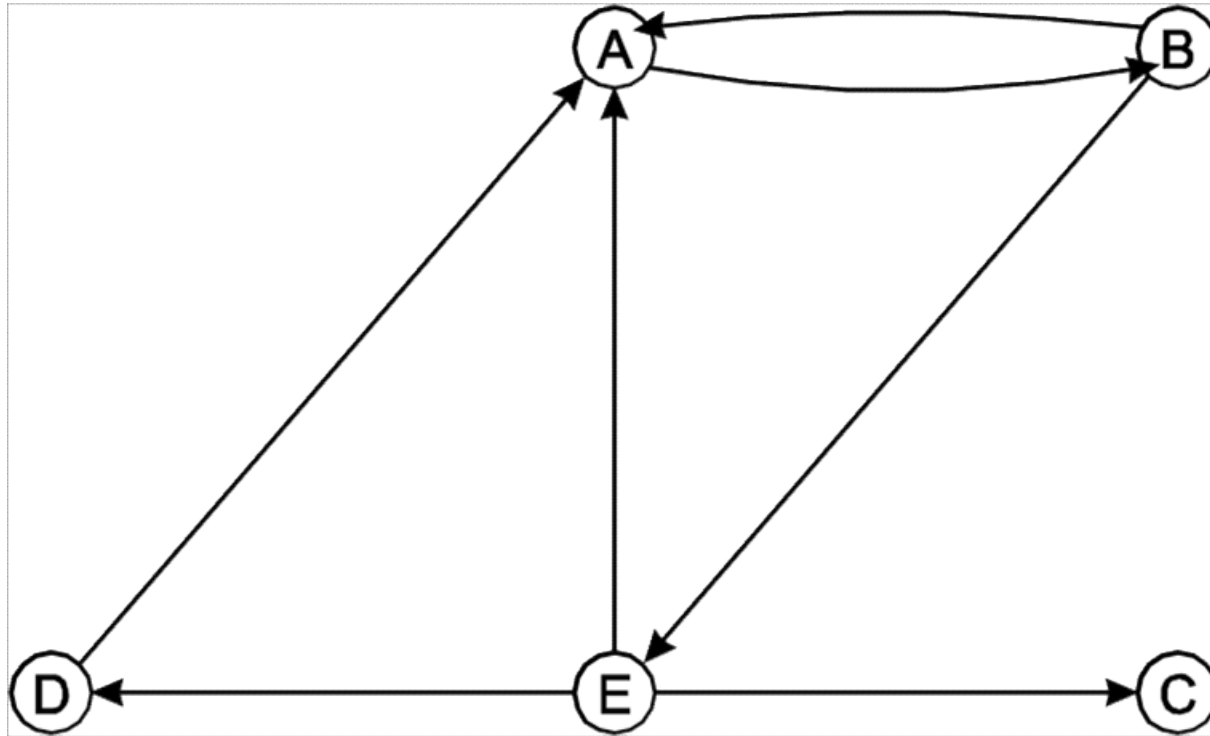
$E = \{\{A, B\}, \{A, D\}, \{C, E\}, \{D, E\}\}$



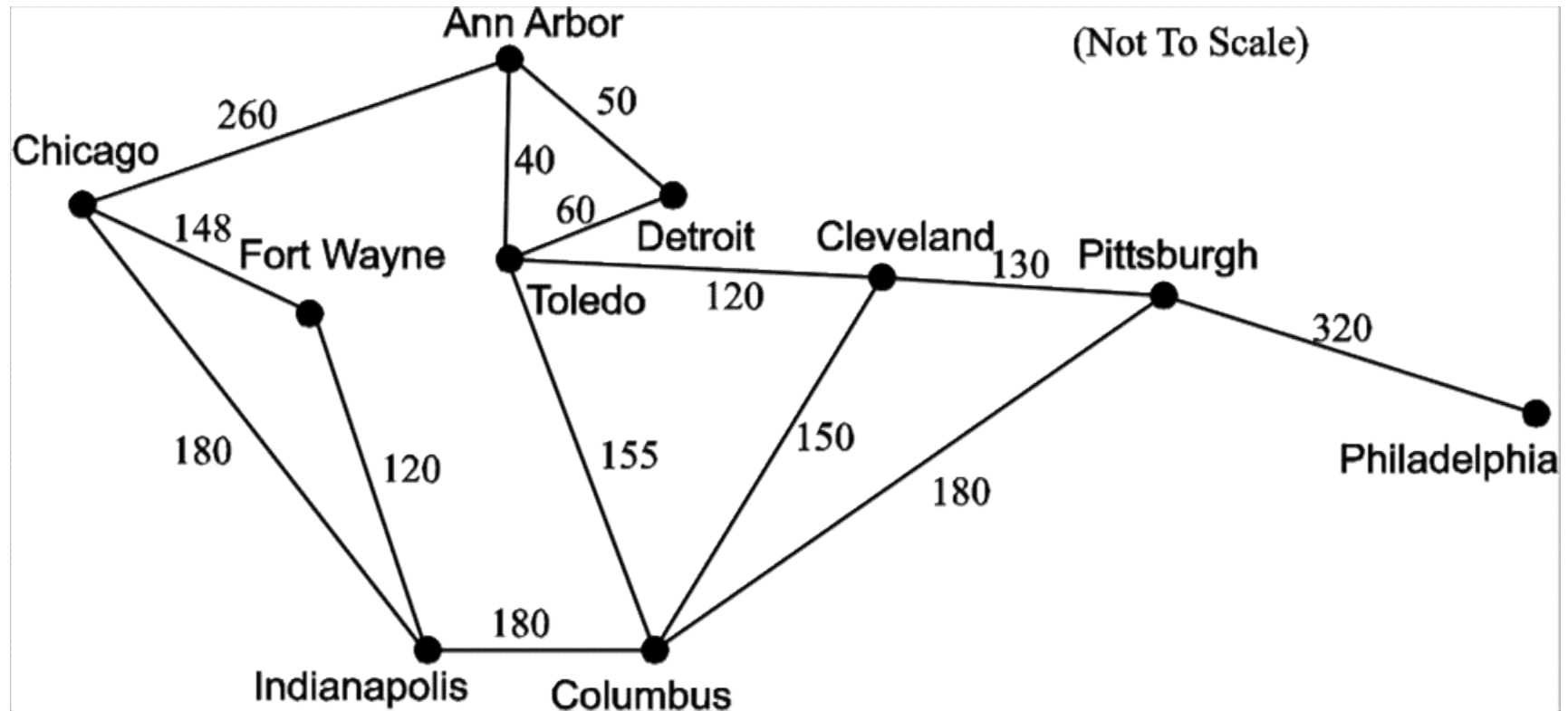
A Directed Graph

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

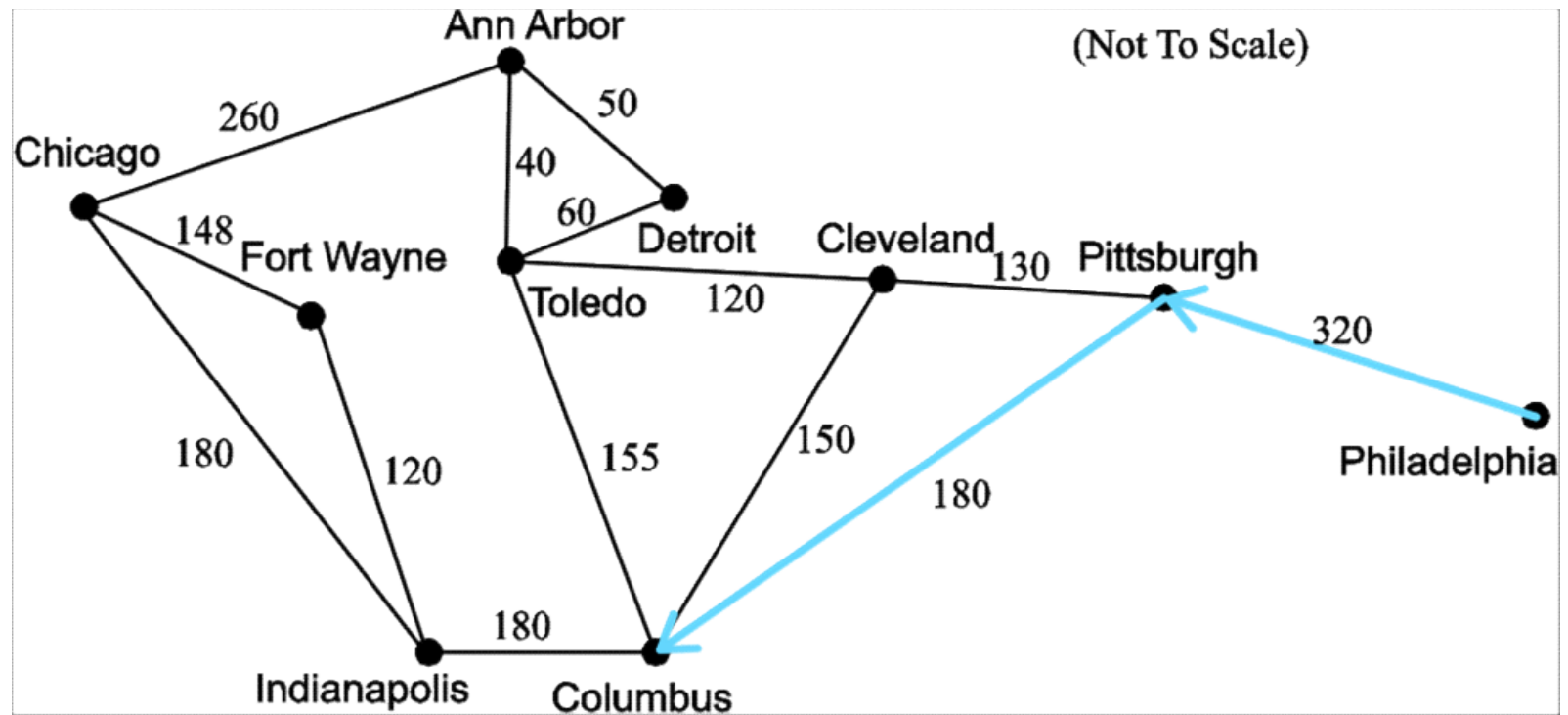
$E = \{(A, B), (B, A), (B, E), (D, A), (E, A), (E, D), (E, C)\}$



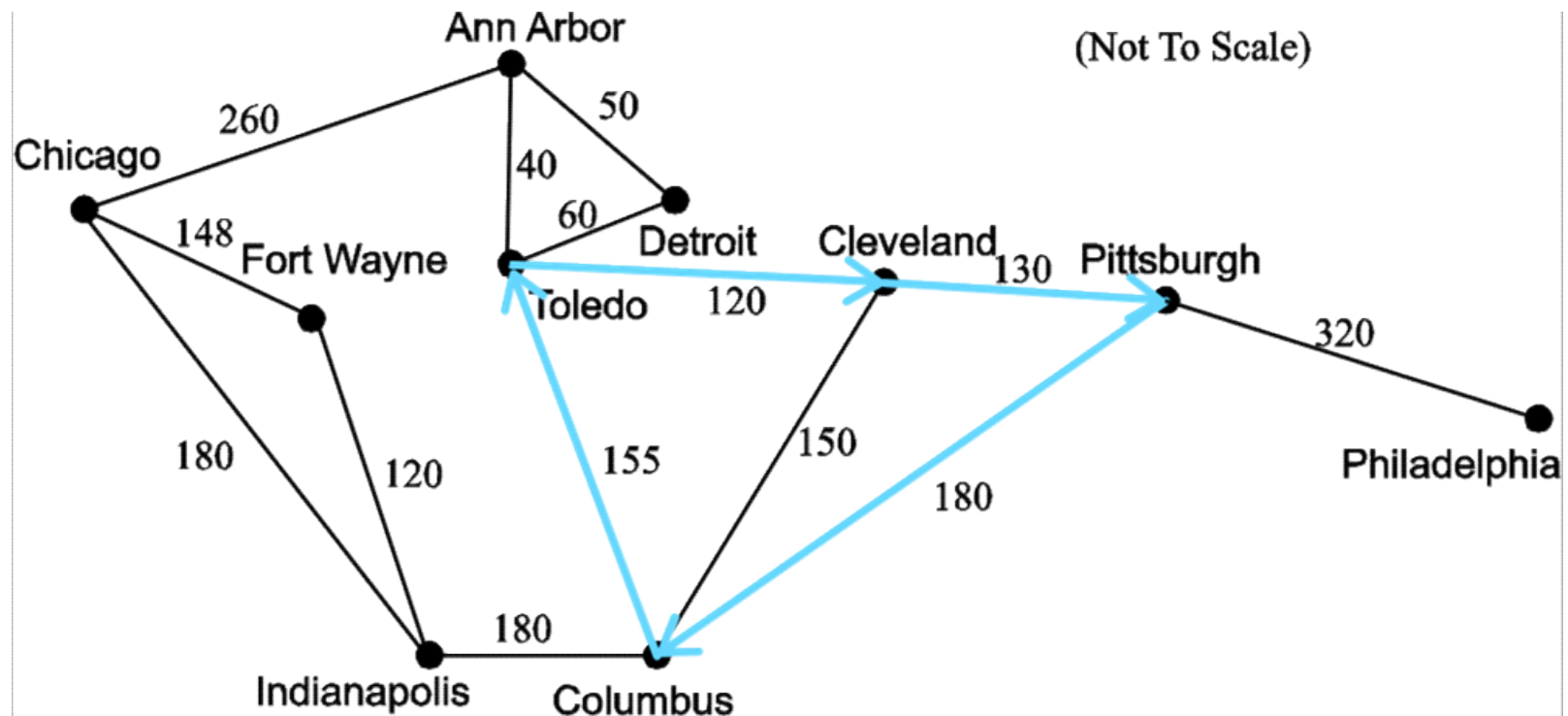
Weighted Graph



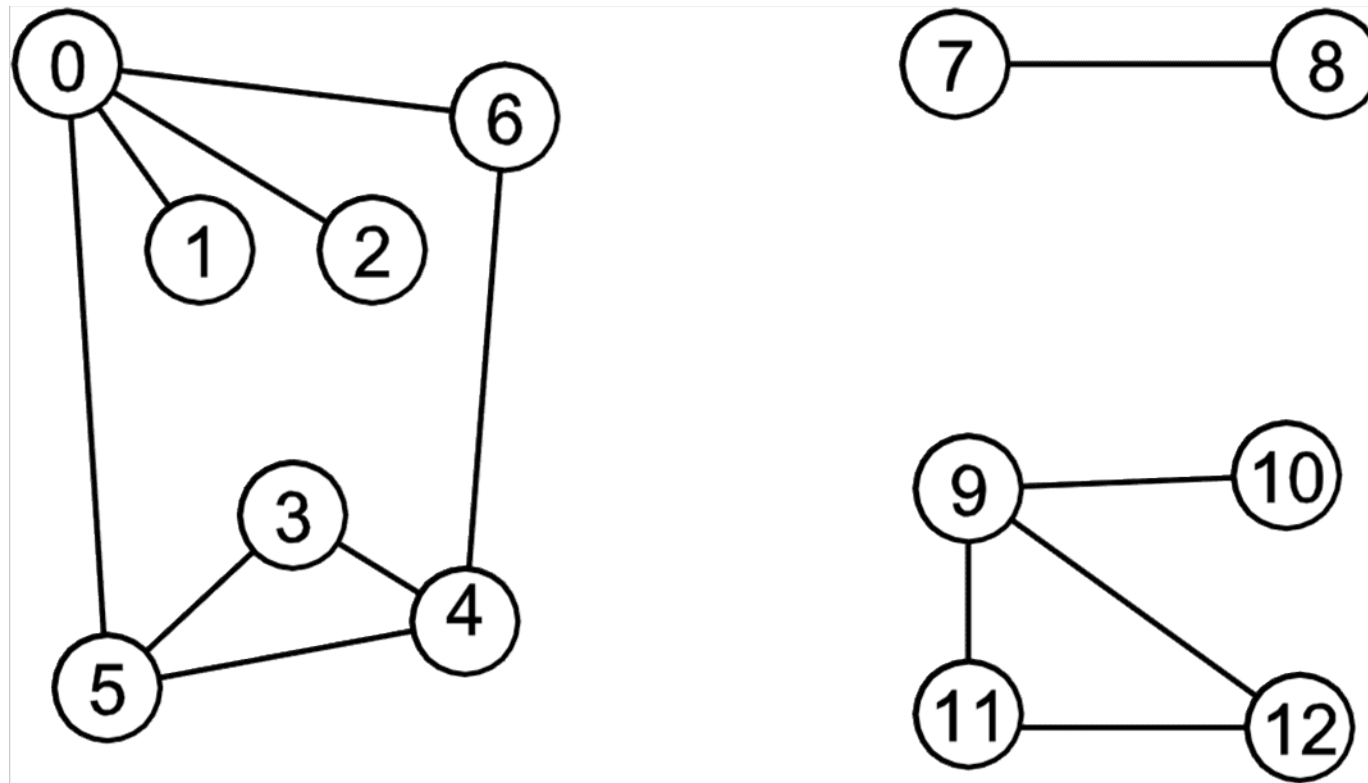
Example of a Path



Example of a Cycle

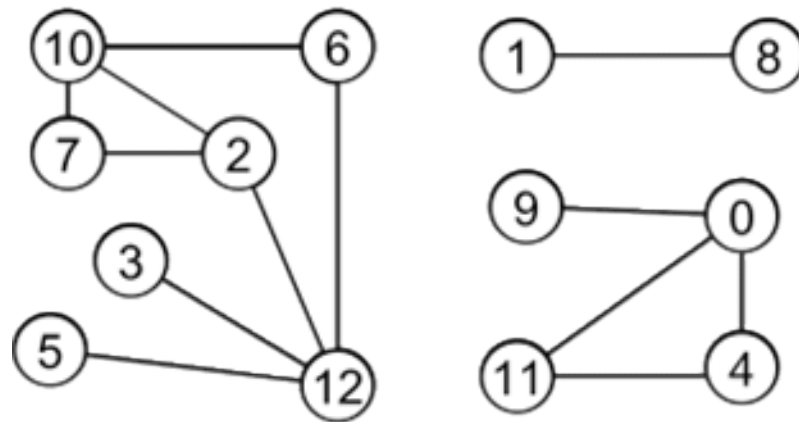


Disconnected Graph

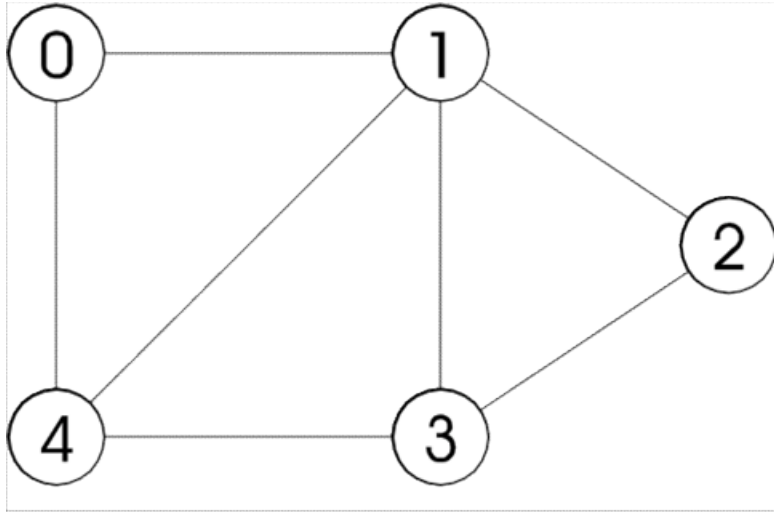


Graph Isomorphism

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

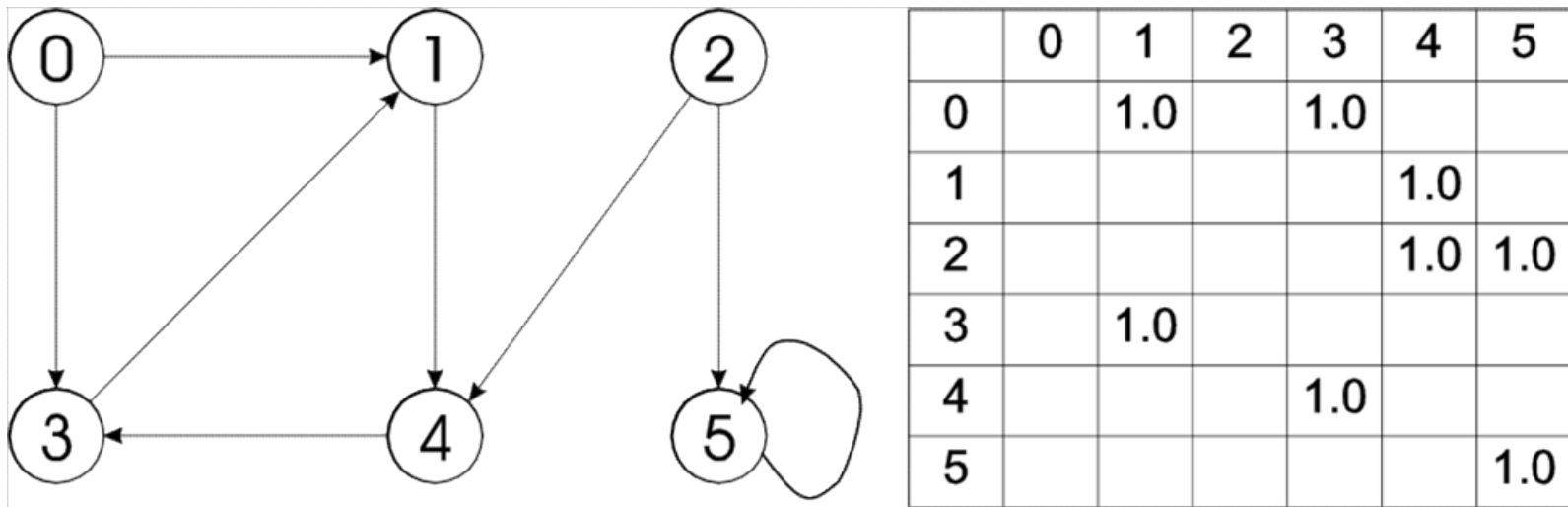


Adjacency Matrix

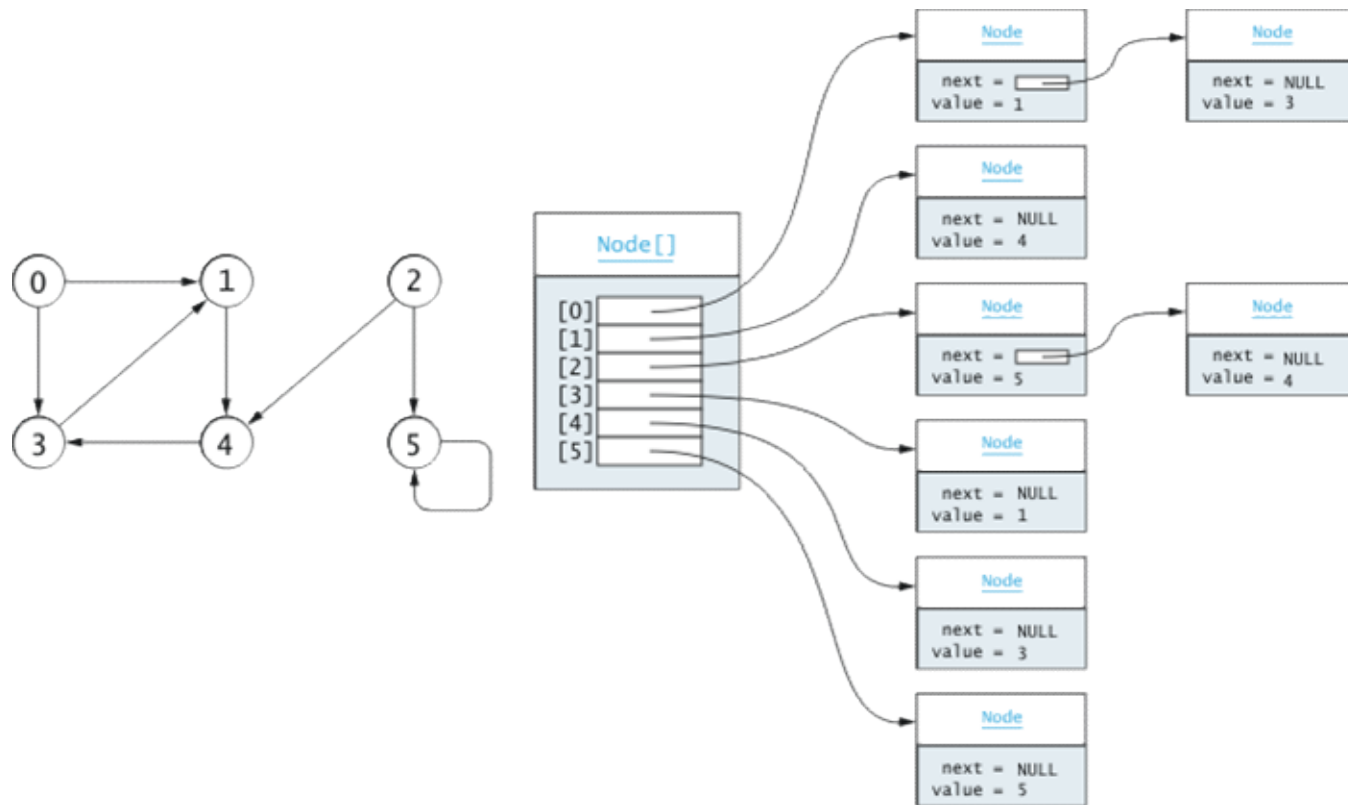


	0	1	2	3	4
0		1.0			1.0
1	1.0		1.0	1.0	1.0
2		1.0		1.0	
3		1.0	1.0		1.0
4	1.0	1.0		1.0	

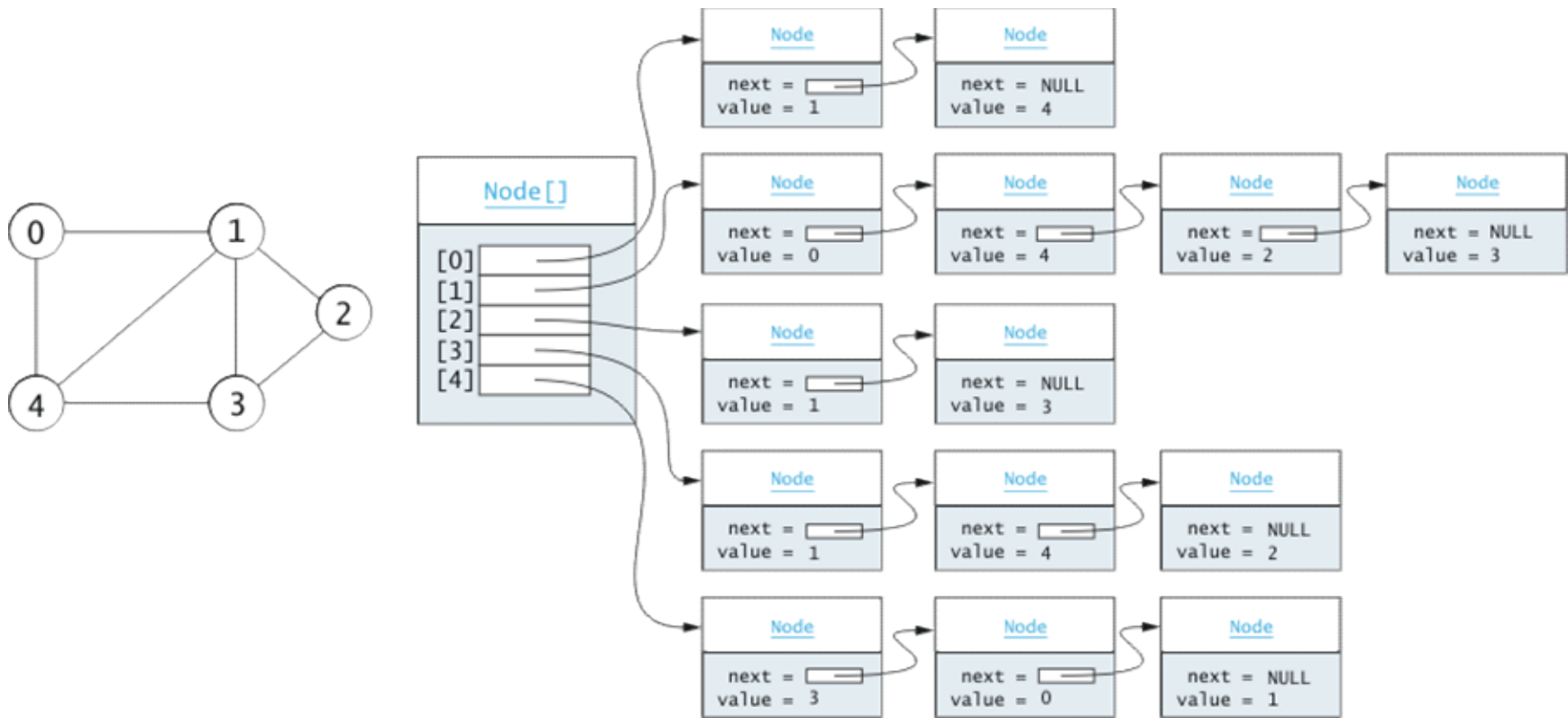
Adjacency Matrix (directed graph)



Adjacency List (Directed Graph)



Adjacency List Representation



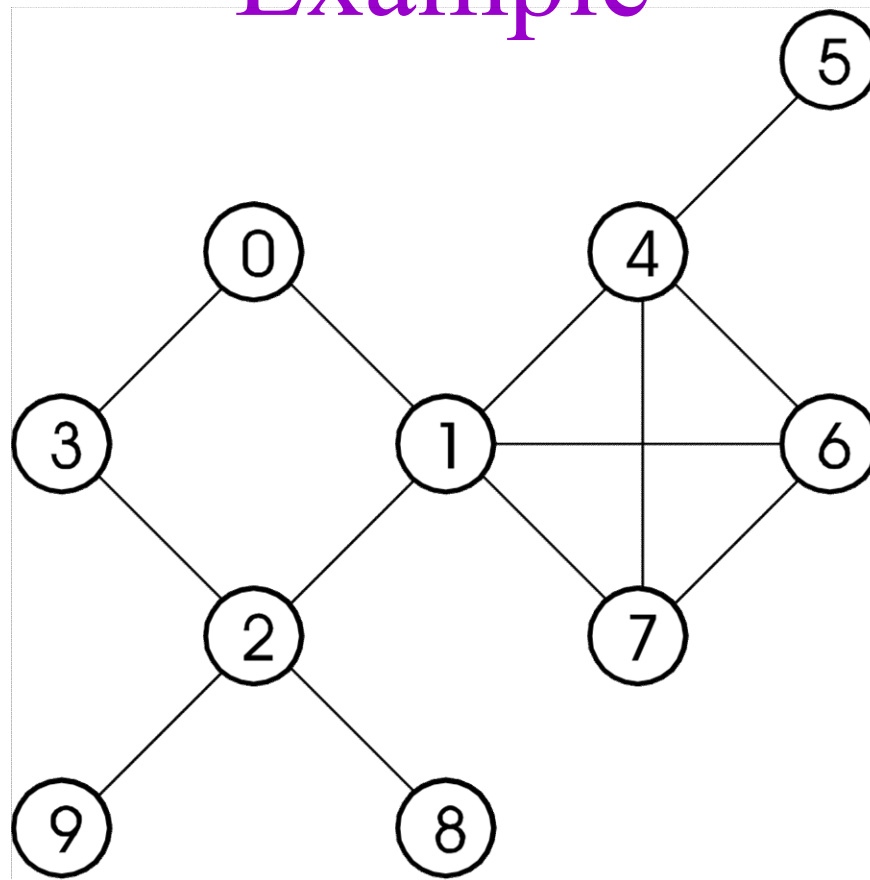
Breadth First Search

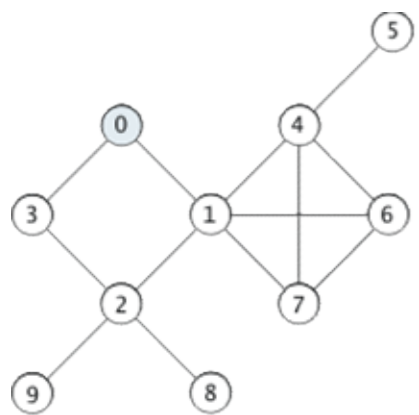
- Starting at a source vertex
- Systematically explore the edges to “discover” every vertex reachable from s .
- Produces a “breadth-first tree”
 - Root of s
 - Contains all vertices reachable from s
 - Path from s to v is the shortest path

Algorithm

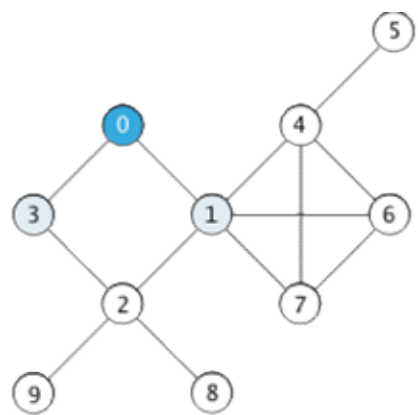
1. Take a start vertex, mark it identified (color it gray), and place it into a queue.
2. While the queue is not empty
 1. Take a vertex, u , out of the queue (Begin visiting u)
 2. For all vertices v , adjacent to u ,
 1. If v has not been identified or visited
 1. Mark it identified (color it gray)
 2. Place it into the queue
 3. Add edge u, v to the Breadth First Search Tree
 3. We are now done visiting u (color it black)

Example

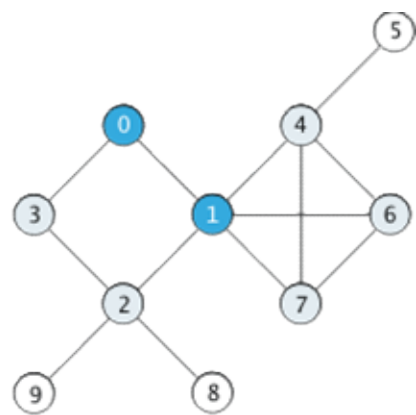




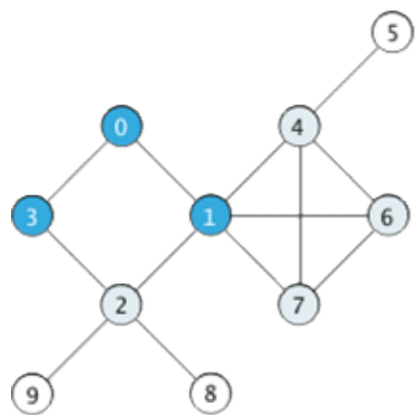
(a)



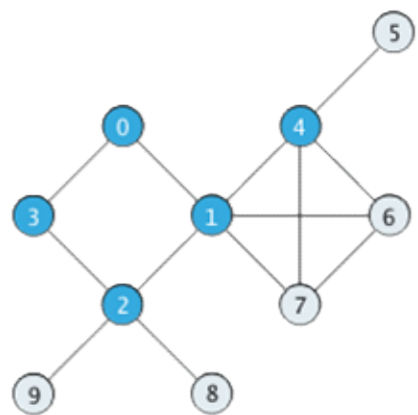
(b)



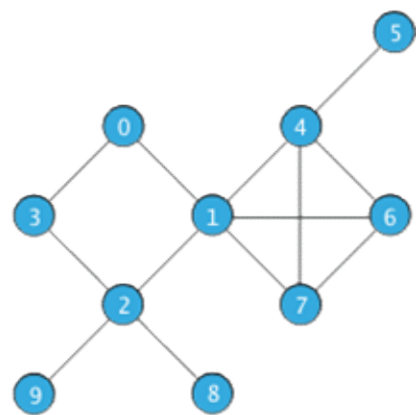
(c)



(d)



(e)

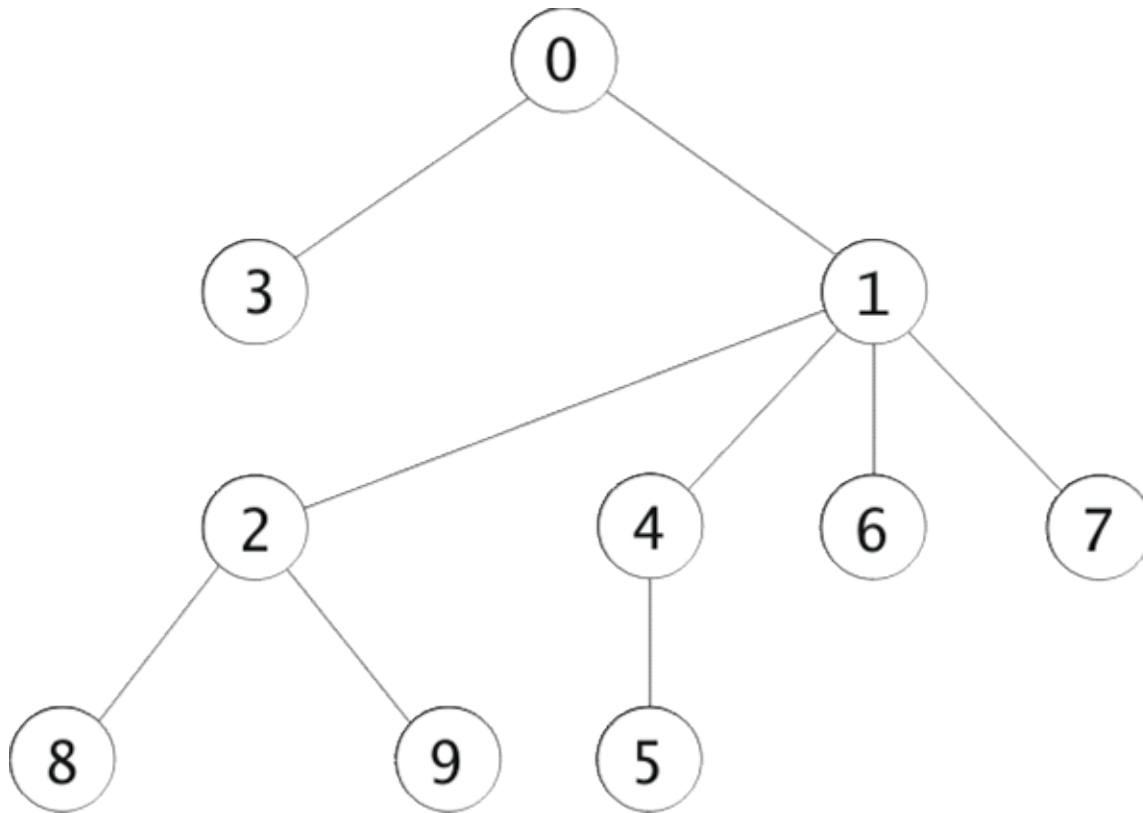


(f)

Trace of Breadth First Search

Vertex Being Visited	Queue Contents after Visit	Visit Sequence
0	1 3	0
1	3 2 4 6 7	0 1
3	2 4 6 7	0 1 3
2	4 6 7 8 9	0 1 3 2
4	6 7 8 9 5	0 1 3 2 4
6	7 8 9 5	0 1 3 2 4 6
7	8 9 5	0 1 3 2 4 6 7
8	9 5	0 1 3 2 4 6 7 8
9	5	0 1 3 2 4 6 7 8 9
5	empty	0 1 3 2 4 6 7 8 9 5

Breadth-First Search Tree



vector<int> parent	
[0]	-1
[1]	0
[2]	1
[3]	0
[4]	1
[5]	4
[6]	1
[7]	1
[8]	2
[9]	2

Application of BFS

- A Breadth First Search finds the shortest path from the start vertex to all other vertices based on number of edges.
- We can use a Breadth First Search to find the shortest path through a maze.
- This may be a more efficient solution than that found by a backtracking algorithm.

Depth First Search

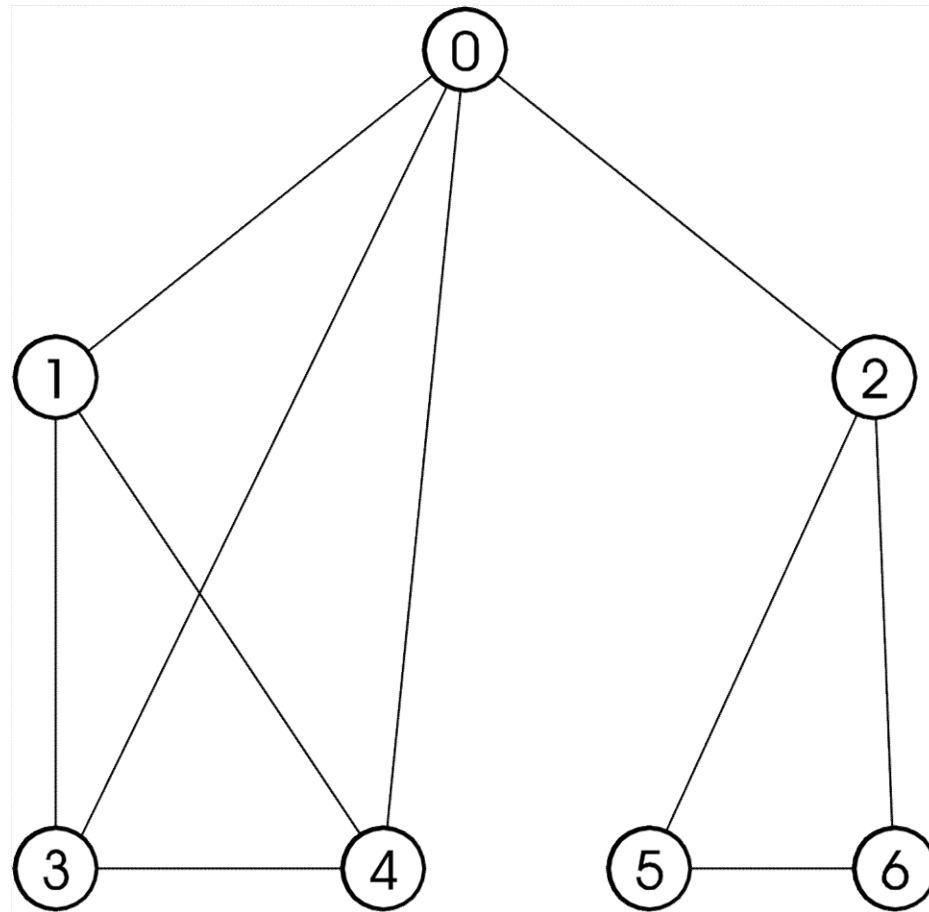
- Start at some vertex
- Follow a simple path discovering new vertices until you cannot find a new vertex.
- Back-up until you can start finding new vertices.

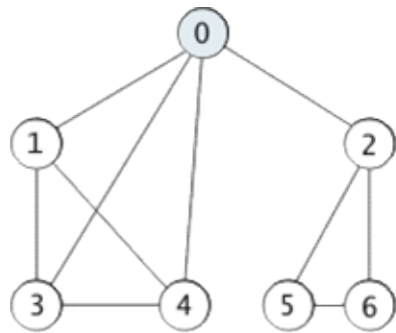
A *simple path* is a path with no cycles.

Algorithm for DFS

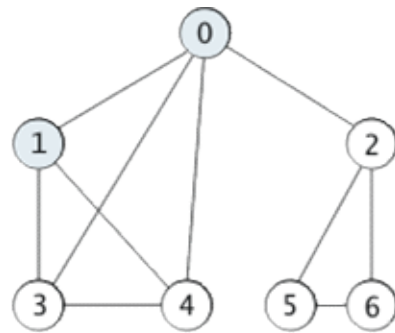
1. Start at vertex u . Mark it visited.
 2. For each vertex v adjacent to u
 1. If v has not been visited
 1. Insert u, v into DFS tree
 2. Recursively apply this algorithm starting at v
 3. Mark u finished.
- Note: for a Directed graph, a DFS may produce multiple trees – this is called a DFS forest.

DFS Example

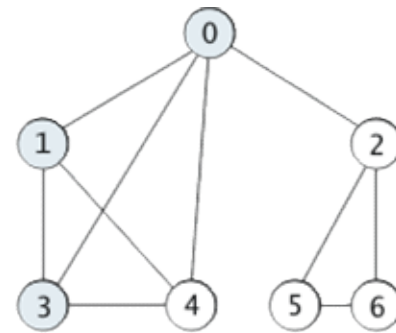




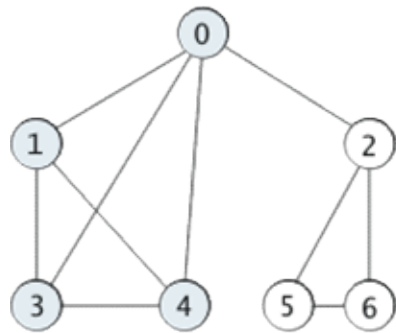
(a)



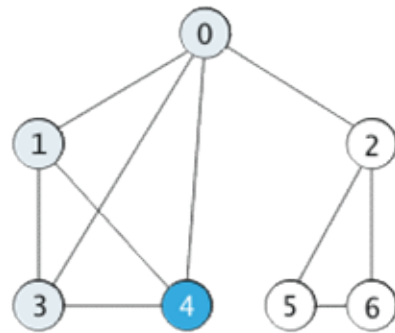
(b)



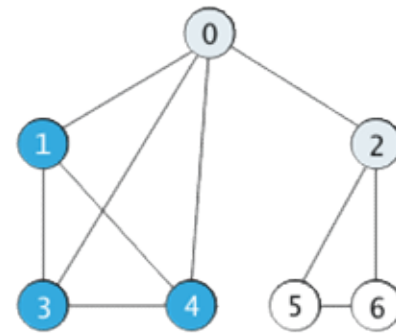
(c)



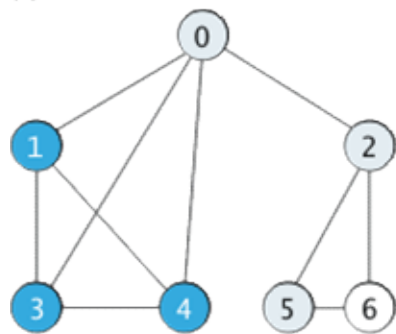
(d)



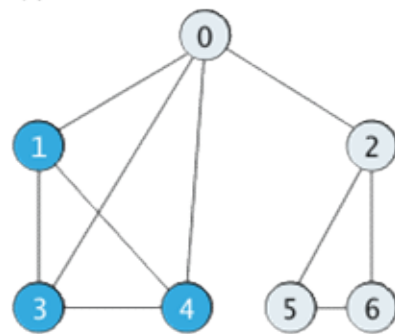
(e)



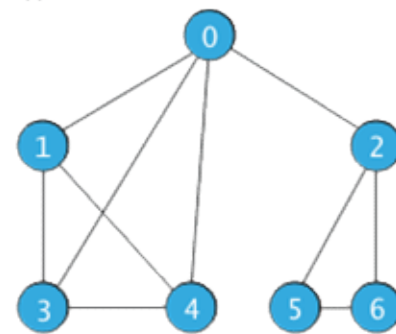
(f)



(g)



(h)

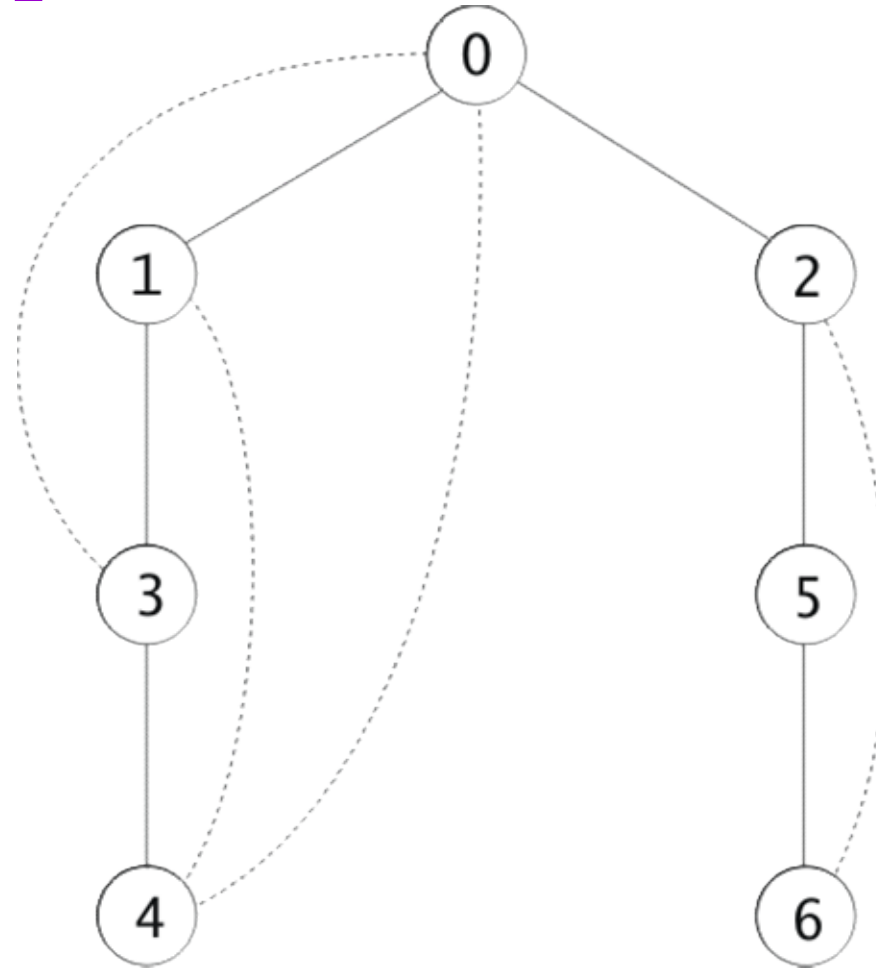


(i)

Trace of Depth-First Search

Operation	Adjacent Vertices	Discovery Order	Finish Order
Visit 0	1 2 3 4	0	
Visit 1	0 3 4	0 1	
Visit 3	0 1 4	0 1 3	
Visit 4	0 1 3	0 1 3 4	
Finish 4			4
Finish 3			4 3
Finish 1			4 3 1
Visit 2	0 5 6	0 1 3 4 2	
Visit 5	2 6	0 1 3 4 2 5	
Visit 6	2 5	0 1 3 4 2 5 6	
Finish 6			4 3 1 6
Finish 5			5 3 1 6 5
Finish 2			5 3 1 6 5 2
Finish 0			5 3 1 6 5 2 0

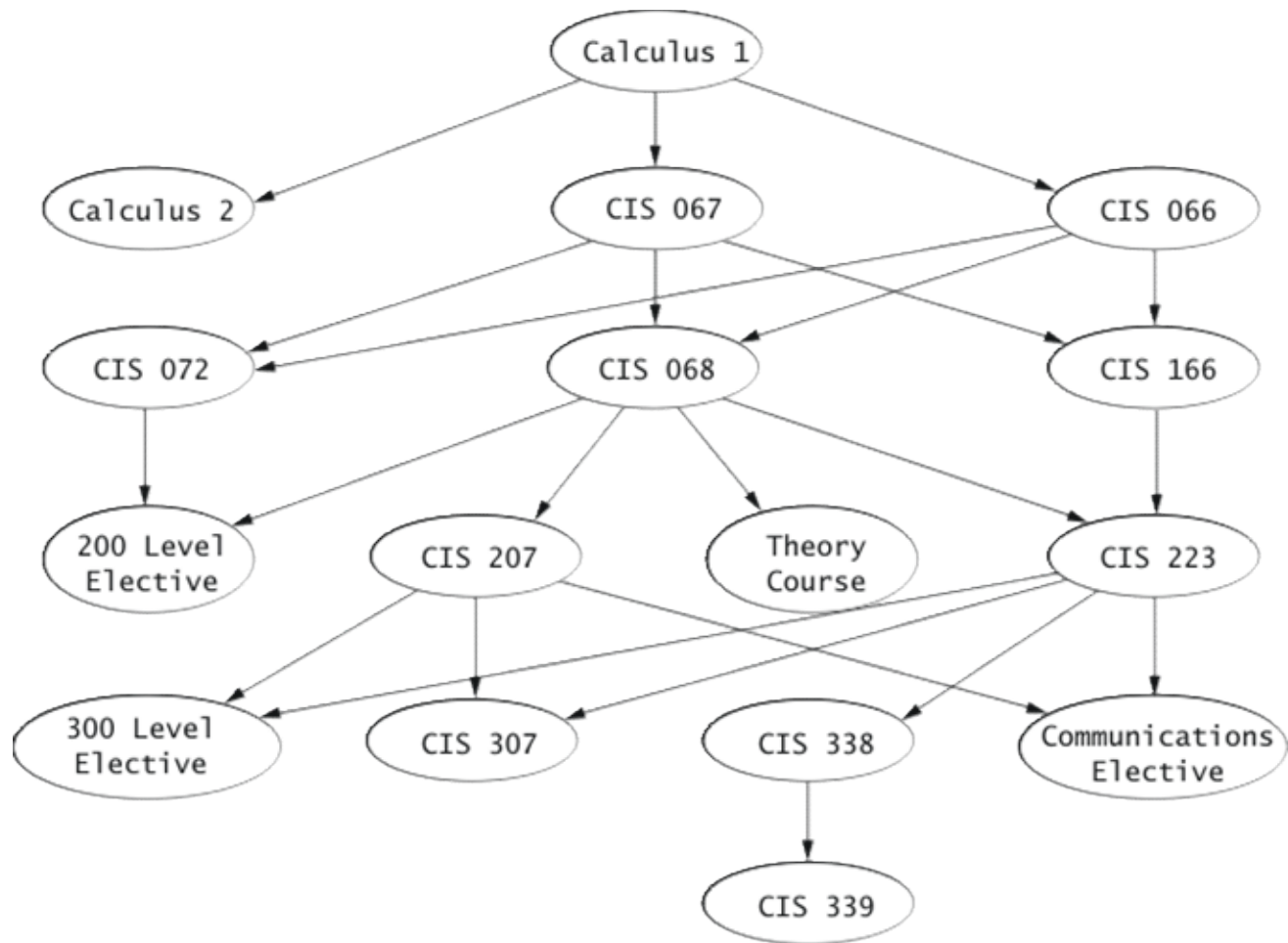
Depth-First Search Tree



Application of DFS

- A topological sort is an ordering of the vertices such that if (u, v) is an edge in the graph, then v does not appear before u in the ordering.
- A Depth First Search can be used to determine a topological sort of a Directed Acyclic Graph (DAG)
- An application of this would be to determine an ordering of courses that is consistent with the prerequisite requirements.

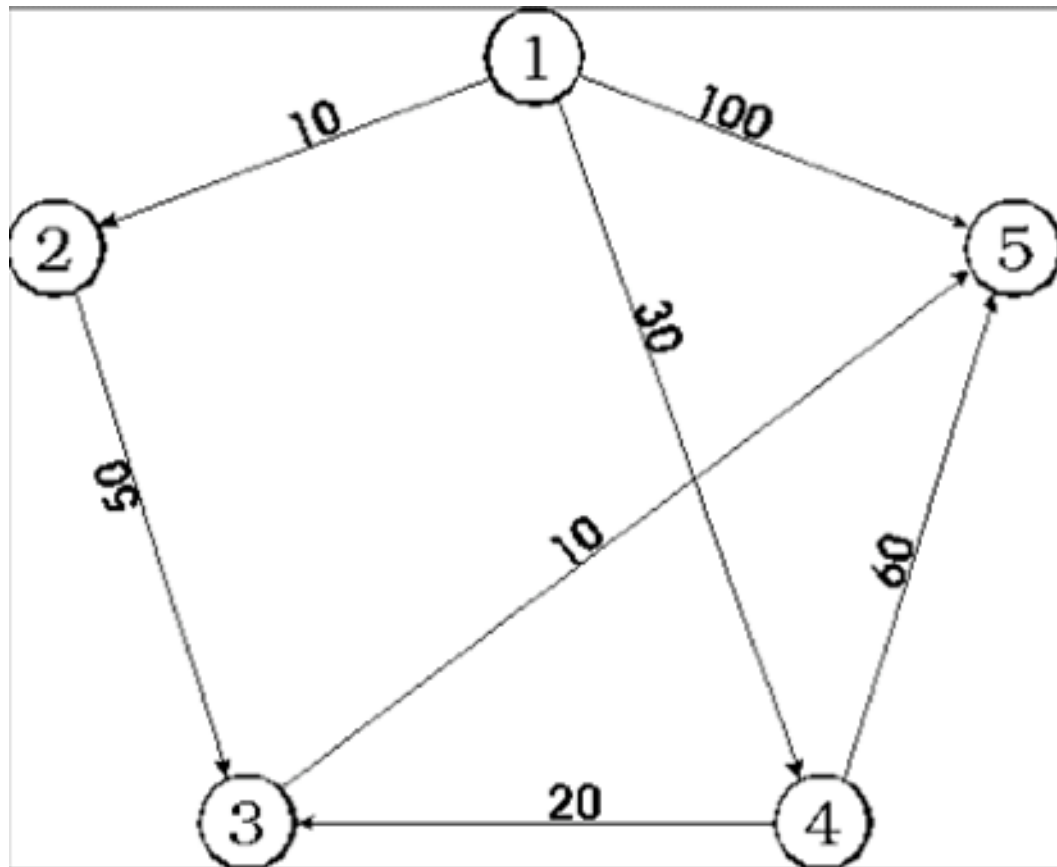
CIS Prerequisites



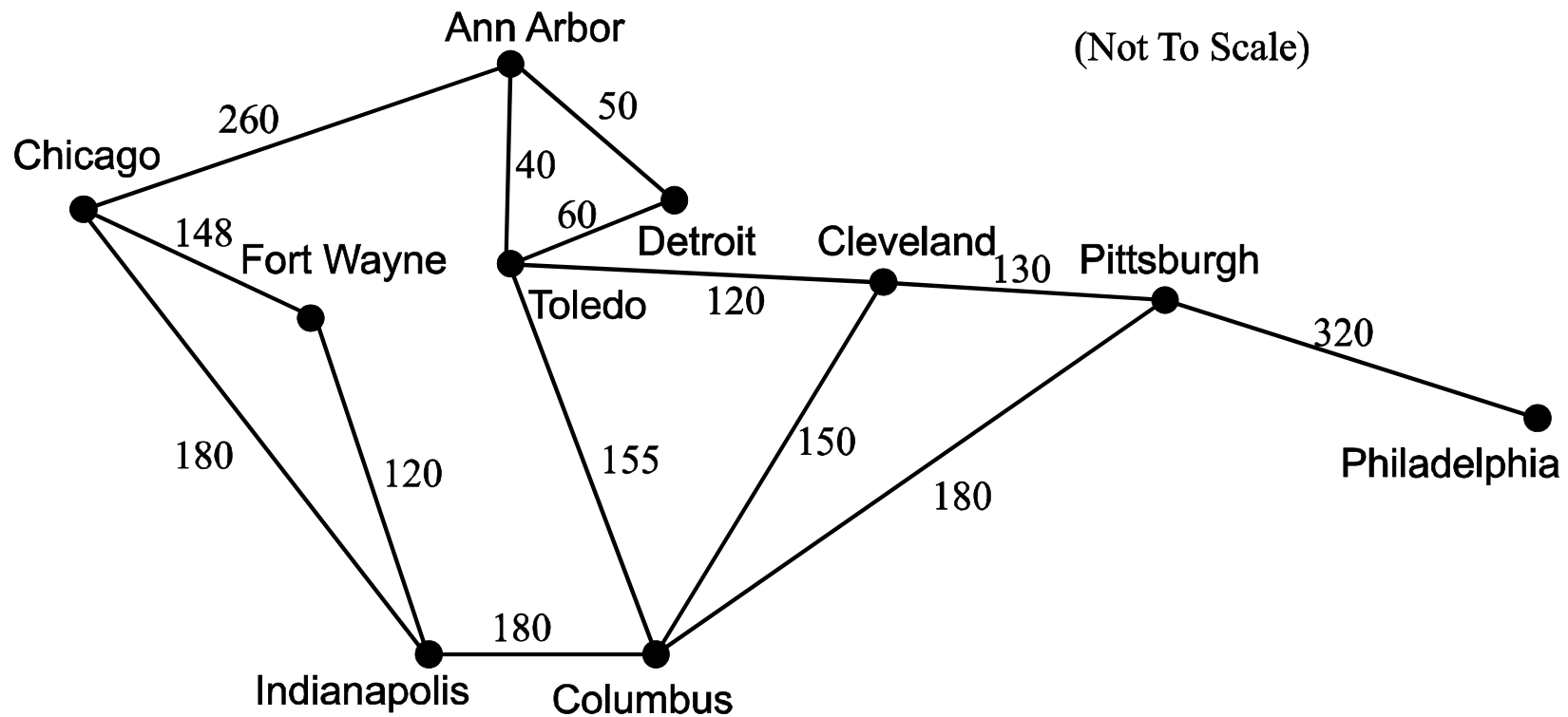
Topological Sort Algorithm

- Observation: If there is an edge from u to v , then in a DFS u will have a finish time later than v .
- Perform a DFS and output the vertices in the reverse order that they're marked as finished.

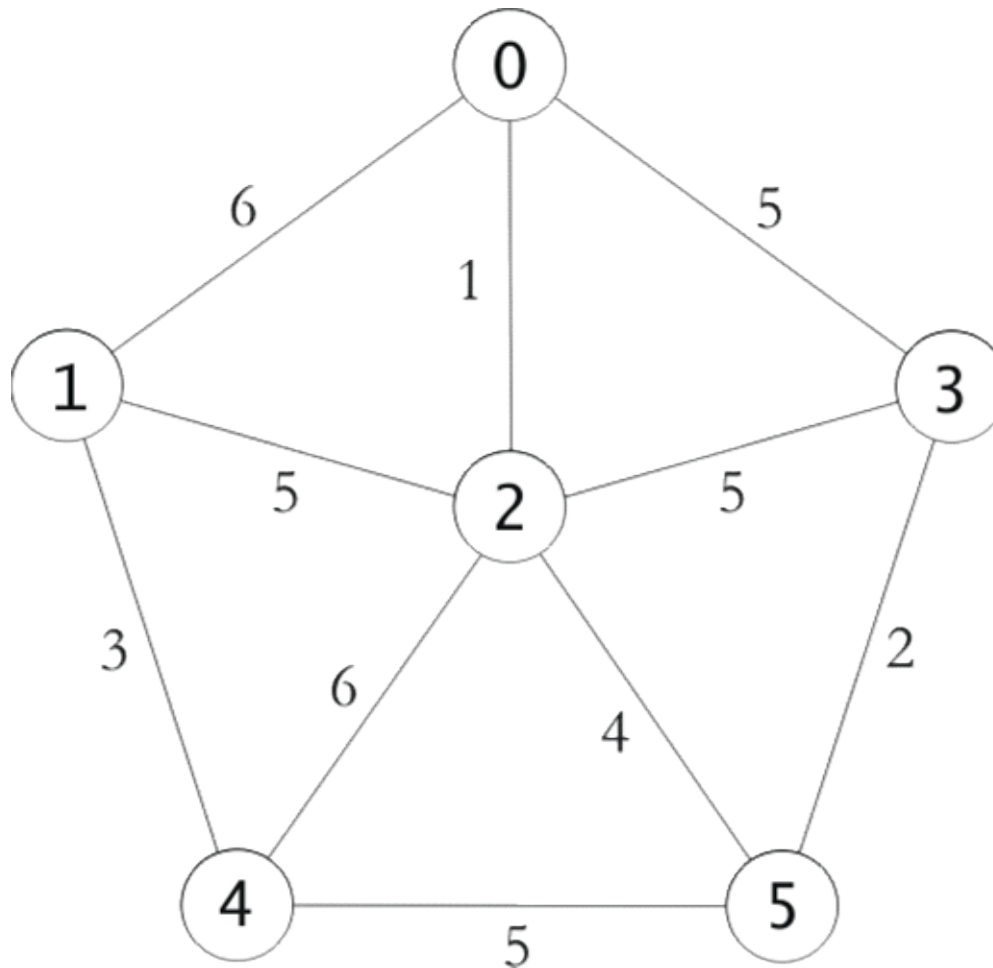
Example of a Directed Weighted Graph



Another Weighted Graph



Example



Separate Tree and Graph Examples: Cities in Germany

