CSC 172– Data Structures and Algorithms

Lecture #24

Spring 2018

Please put away all electronic devices

CONNECTEDNESS

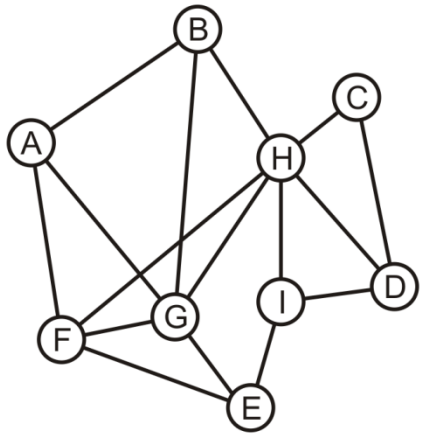
Outline

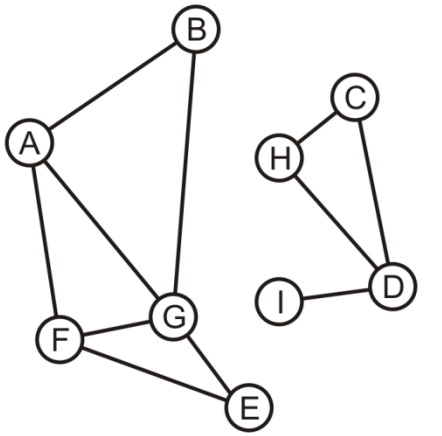
We will use graph traversals to determine:

– Whether one vertex is connected to another

Determining Connections

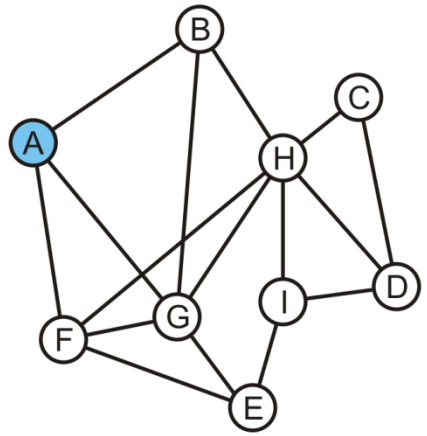
Is A connected to D?

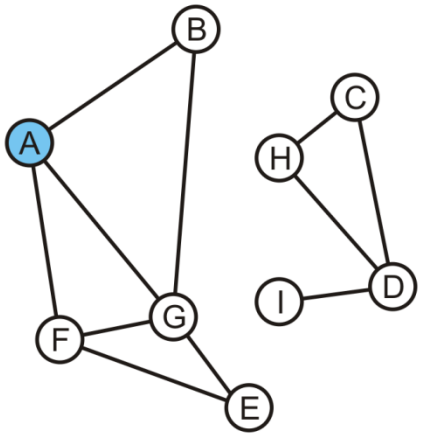




Determining Connections

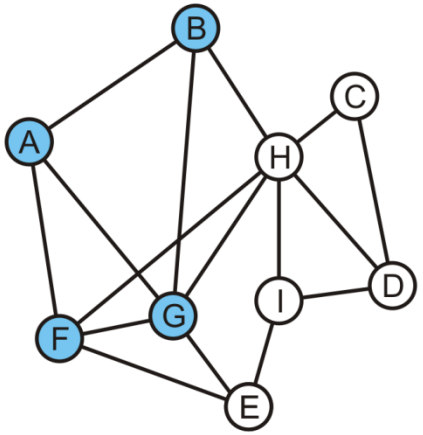
Vertex A is marked as visited and pushed onto the queue



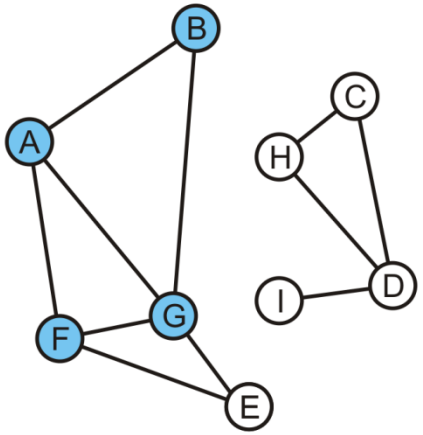


Determining Connections

Pop the head, A, and mark and push B, F and G



B	F	G			

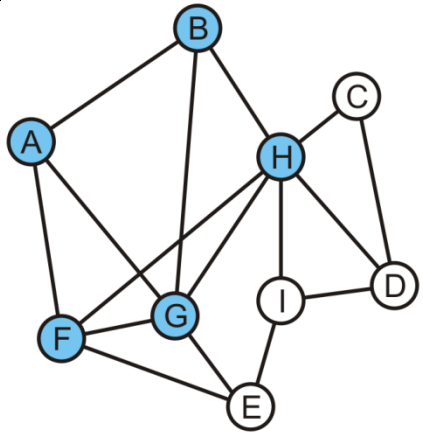


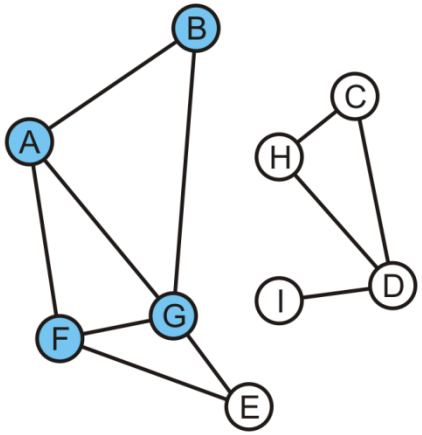
B	F	G			

Determining Connections

Pop B and mark and, in the left graph, mark and push H

– On the right graph, B has no unvisited adjacent vertices



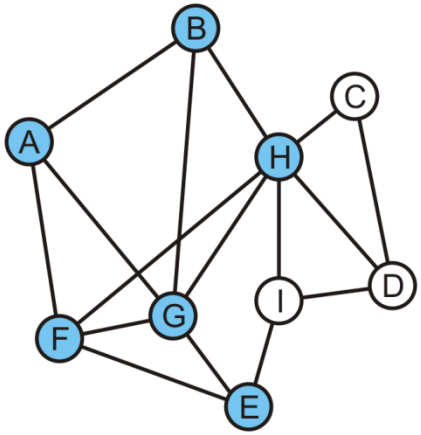


F	G	H			

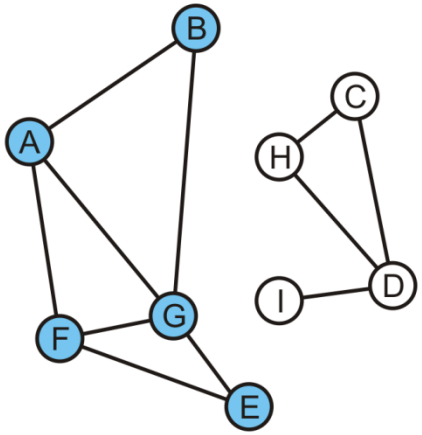
F	G				

Determining Connections

Popping F results in the pushing of E



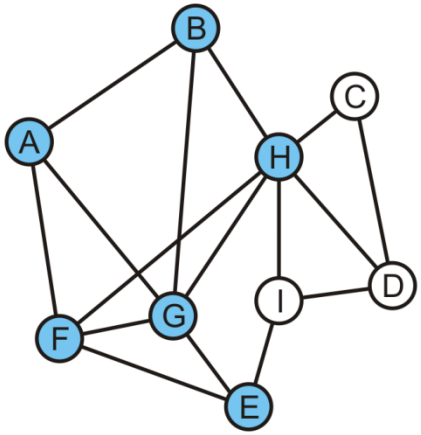
G	H	E			



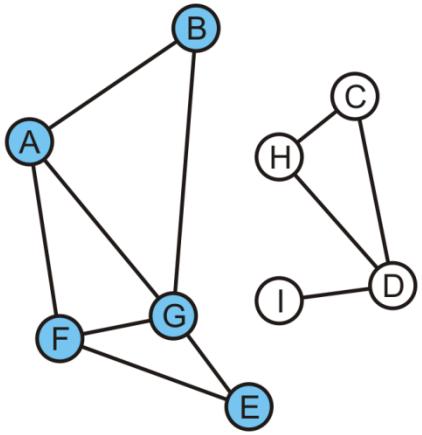
G	E				

Determining Connections

In either graph, G has no adjacent vertices that are unvisited



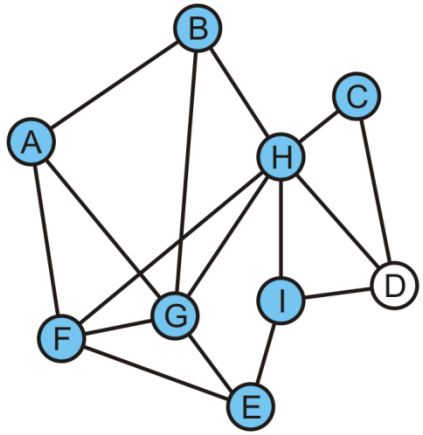
H	E				

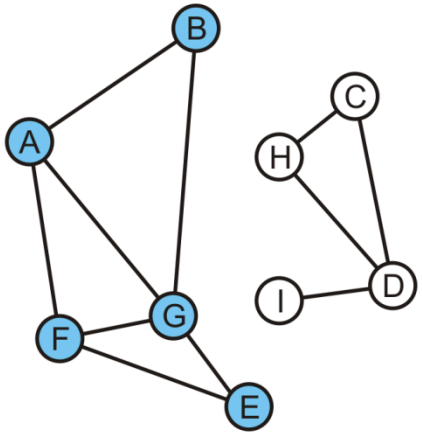


E					

Determining Connections

Popping H on the left graph results in C and I being pushed



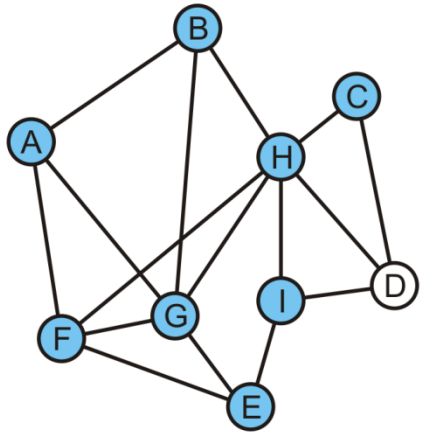


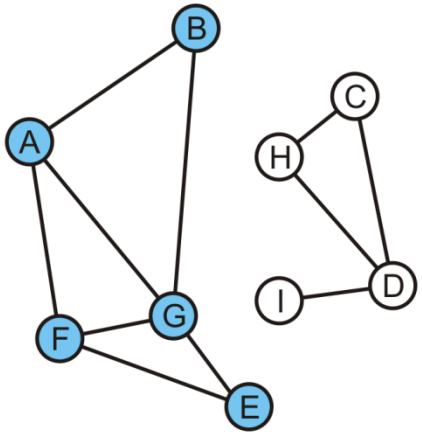
C	I	D			

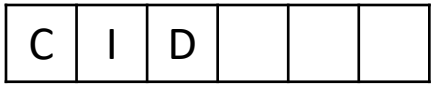
Determining Connections

The queue op the right is empty

– We determine A is not connected to D



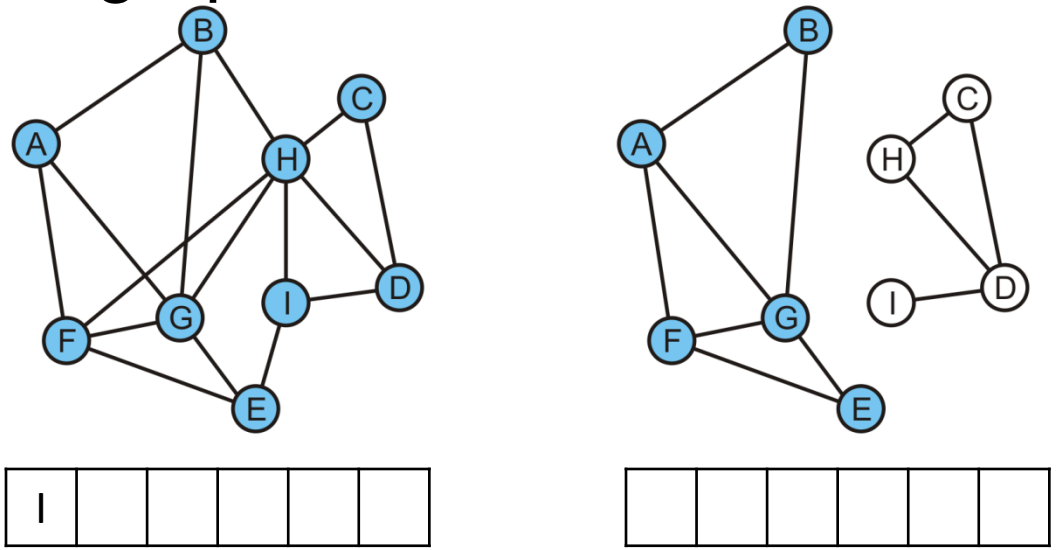




Determining Connections

On the left, we pop C and return `true` because D is adjacent to C

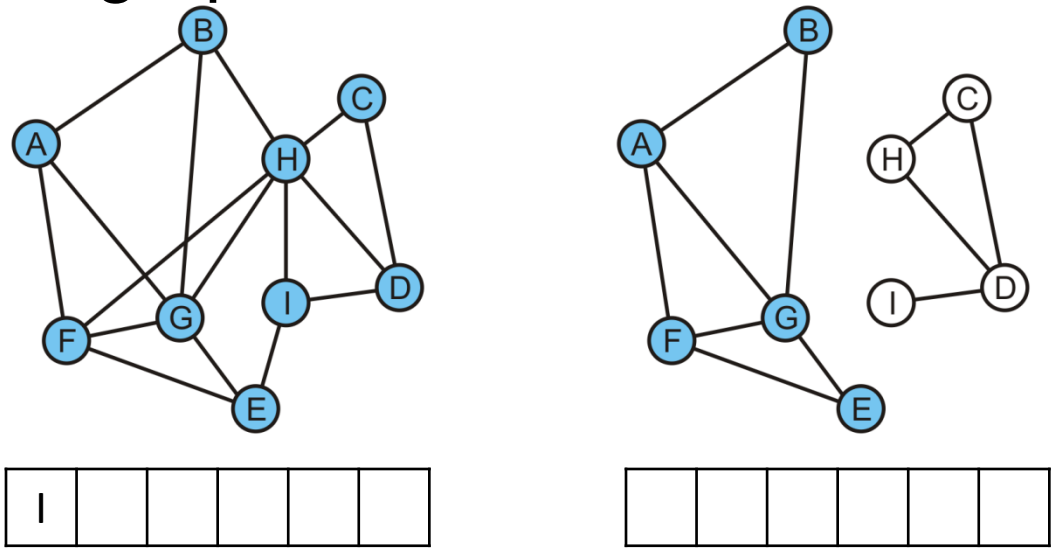
– In the left graph, A is connected to D



Determining Connections

On the left, we pop C and return `true` because D is adjacent to C

– In the left graph, A is connected to D



PATH LENGTH

Outline

This topic looks at another problem solved by breadth-first traversals

- Finding all path lengths in an unweighted graph

Determining Distances

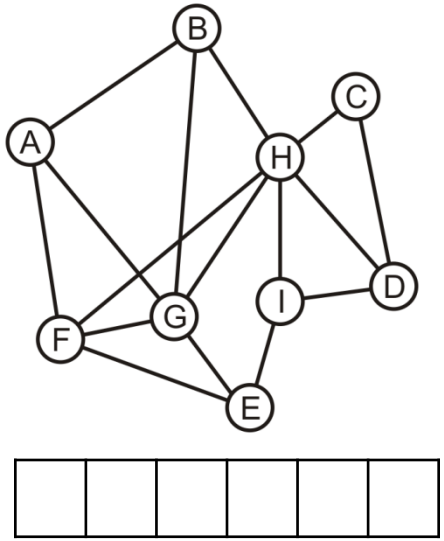
Problem: find the distance from one vertex v to all other vertices

- Use a breadth-first traversal
- Vertices are added in *layers*
- The starting vertex is defined to be in the zeroth layer, L_0
- While the k^{th} layer is not empty:
  - All unvisited vertices adjacent to verticies in L_k are added to the $(k+1)^{\text{st}}$ layer

Any unvisited vertices are said to be an infinite distance from v

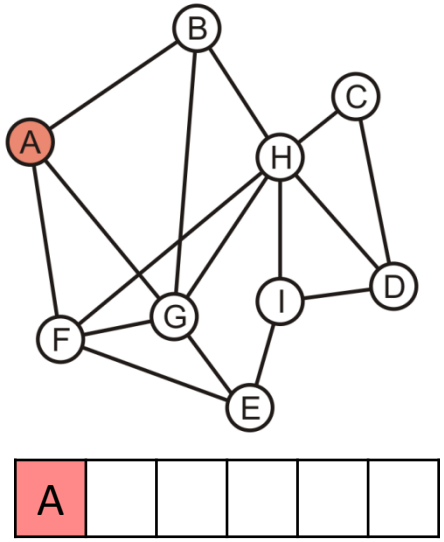
Determining Distances

Consider this graph: find the distance from A to each other vertex



Determining Distances

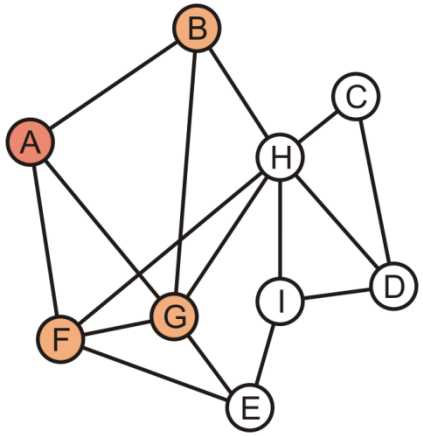
A forms the zeroeth layer, L_0



Determining Distances

The unvisited vertices B, F and G are adjacent to A

– These form the first layer, L_1

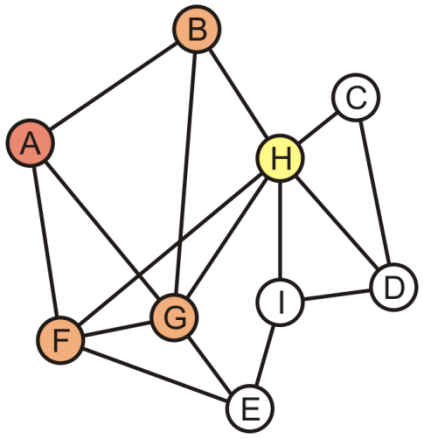


Determining Distances

We now begin popping L_1 vertices: pop B

– H is adjacent to B

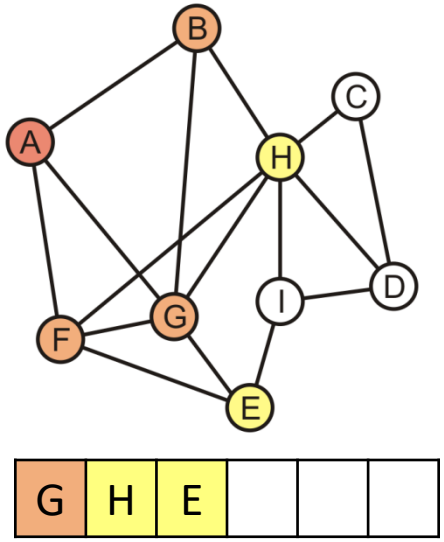
– It is tagged L_2



Determining Distances

Popping F pushes E onto the queue

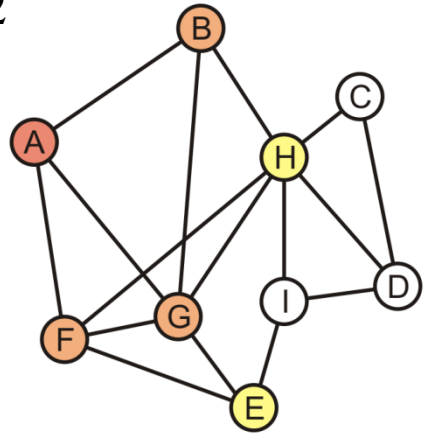
– It is also tagged L_2



Determining Distances

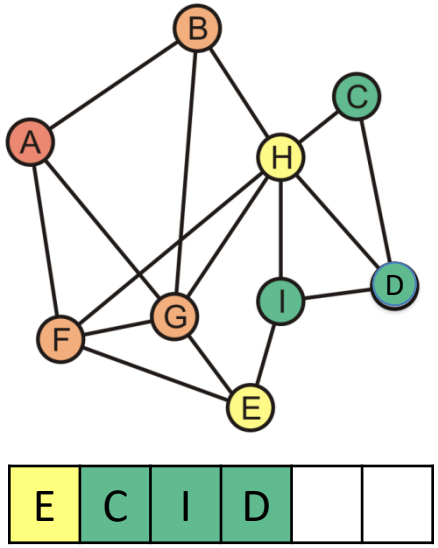
We pop G which has no other unvisited neighbours

– G is the last L_1 vertex; thus H and E form the second layer, L_2



Determining Distances

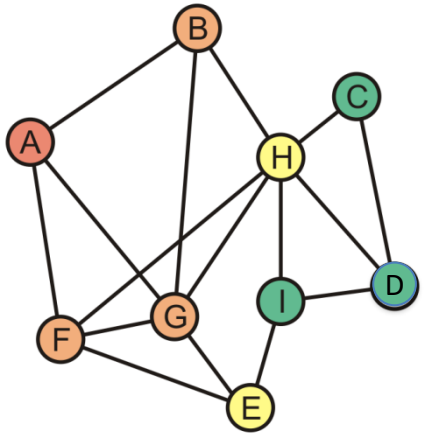
Popping H in L_2 adds C , I, and D to the third layer L_3



Determining Distances

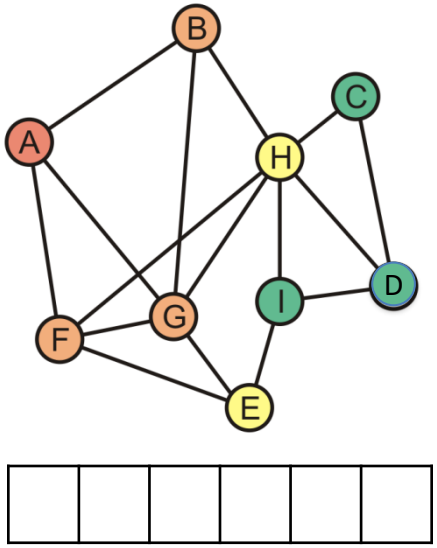
E has no more adjacent unvisited vertices

– Thus C, I, and D form the third layer, L_3



Determining Distances

Pop C,I, D. The queue is Empty. Stop.



TOPOLOGICAL SORTING

Topological Sort

In this topic, we will discuss:

- Motivations
- Review the definition of a directed acyclic graph (DAG)
- Describe a topological sort and applications

Motivation

Given a set of tasks with dependencies,

is there an order in which we can complete the tasks?

Definition of topological sorting

A topological sorting of the vertices in a DAG is an ordering

$$v_1, v_2, v_3, \ldots, v_{|V|}$$

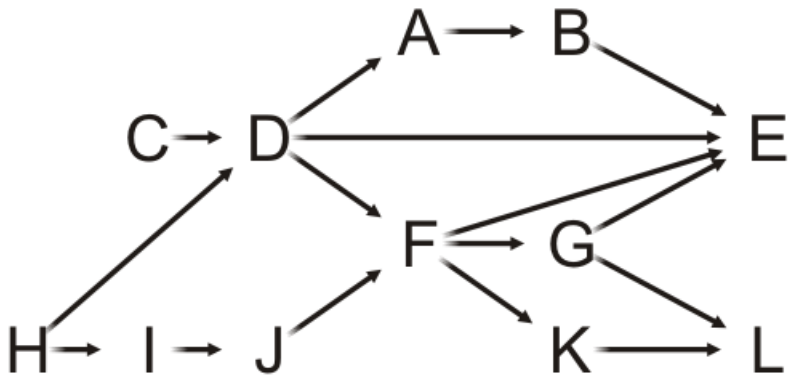
such that v_j appears before v_k if there is a path from v_j to v_k

Example: CSC 171, MTH 150, CSC 172

Definition of topological sorting

Given this DAG, a topological sort is

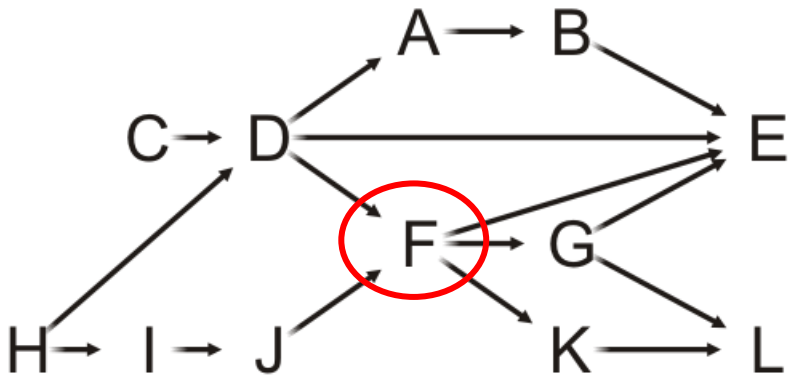
H, C, I, D, J, A, F, B, G, K, E, L



Example

For example, there are paths from H, C, I, D and J to F, so all these must come before F in a topological sort

H, **C**, **I**, **D**, **J**, A, **F**, B, G, K, E, L



Clearly, this sorting need not be unique

Applications

Consider a gentleman is getting ready for a dinner out

He must wear the following:

- jacket, shirt, briefs, socks, tie, etc.

There are certain constraints:

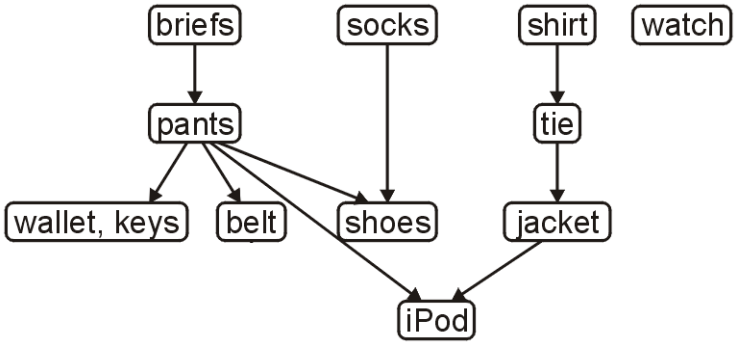
- the pants really should go on after the briefs,
- socks are put on before shoes

Otherwise

Gentleman to Superman

Applications

The following is a task graph for getting dressed:



One topological sort is:

briefs, pants, wallet, keys, belt, socks, shoes, shirt, tie, jacket, iPod, watch

A more reasonable topological sort is:

briefs, socks, pants, shirt, belt, tie, jacket, wallet, keys, iPod, watch, shoes

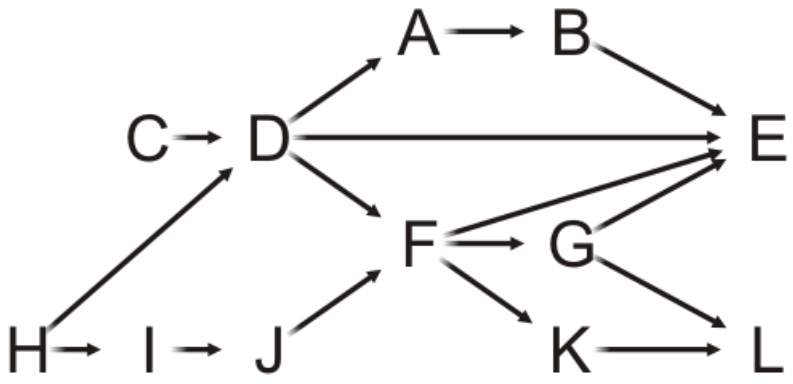
Topological Sort

A graph is a DAG if and only if it has a topological sorting

Example

On this graph, iterate the following $|V| = 12$ times

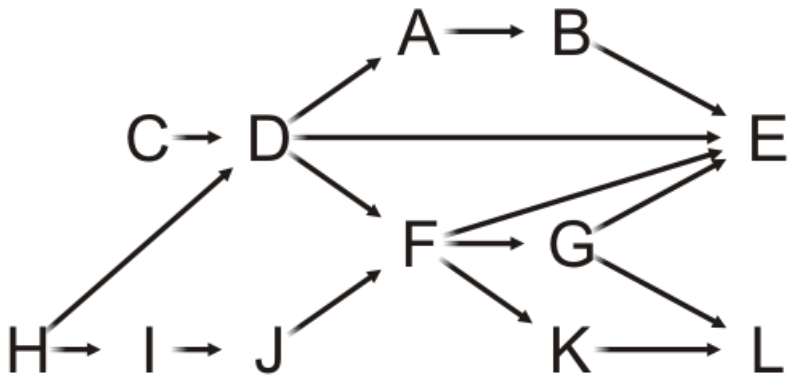
- Choose a vertex v that has in-degree zero
- Let v be the next vertex in our topological sort
- Remove v and all edges connected to it



Example

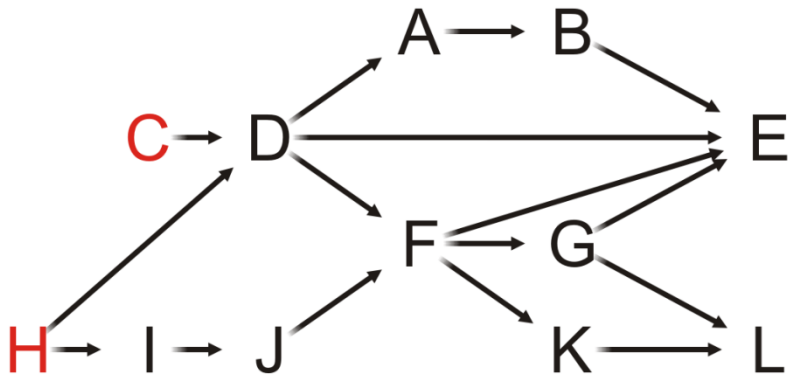
Let's step through this algorithm with this example

- Which task can we start with?



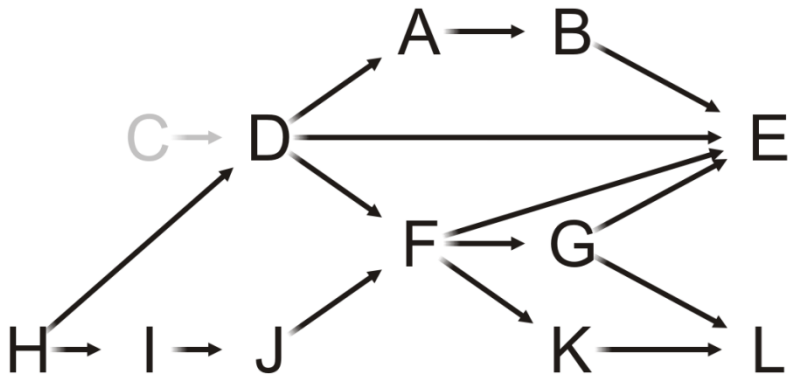
Example

Of Tasks C or H, choose Task C



Example

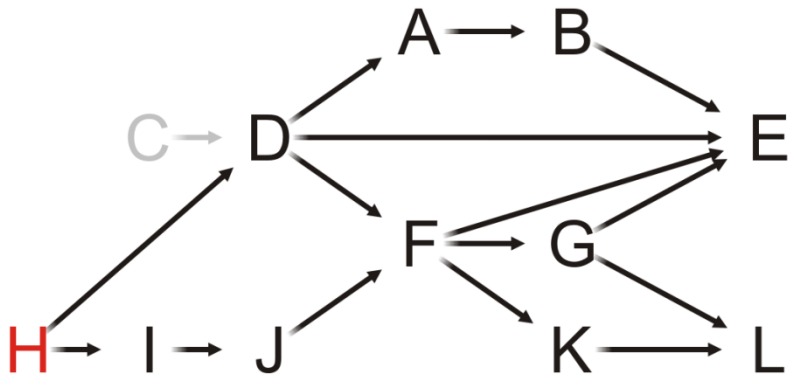
Having completed Task C, which vertices have in-degree zero?



C

Example

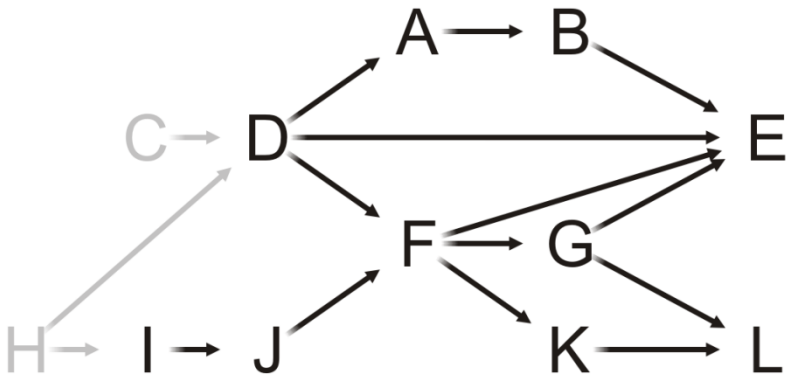
Only Task H can be completed, so we choose it



C

Example

Having removed H, what is next?

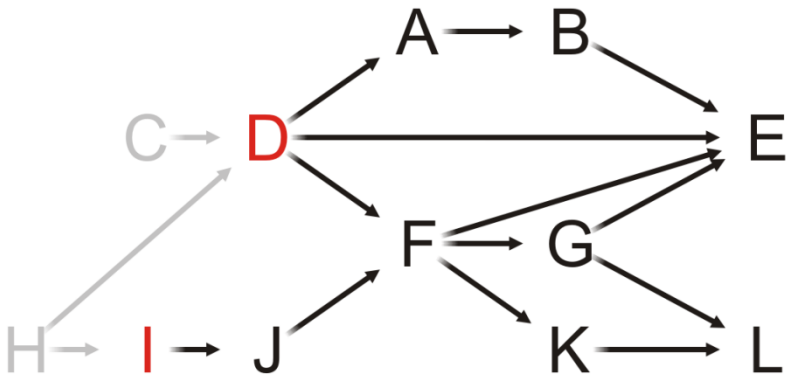


C, H

Example

Both Tasks D and I have in-degree zero

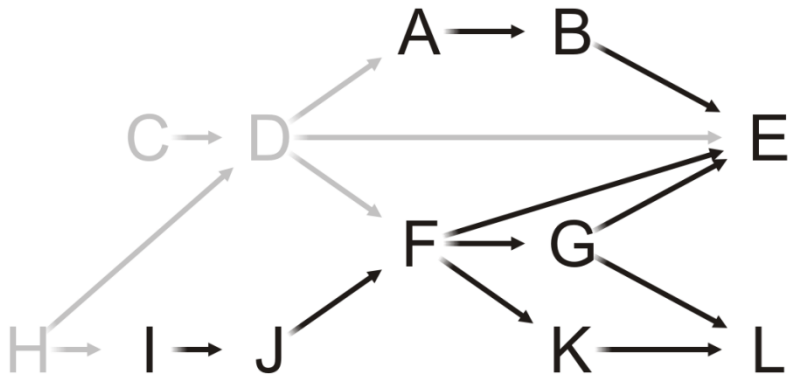
– Let us choose Task D



C, H

Example

We remove Task D, and now?

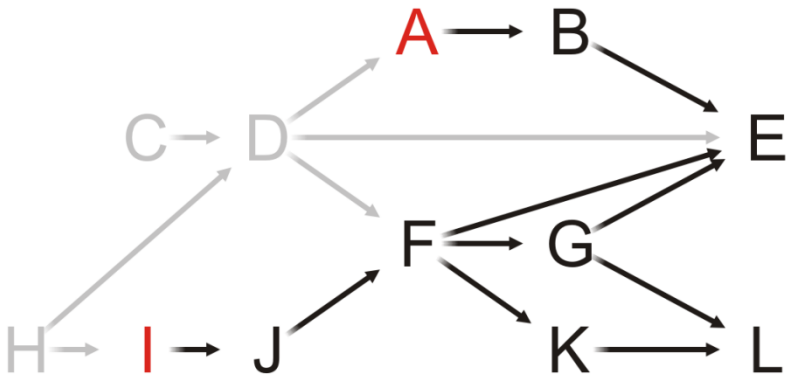


C, H, D

Example

Both Tasks A and I have in-degree zero

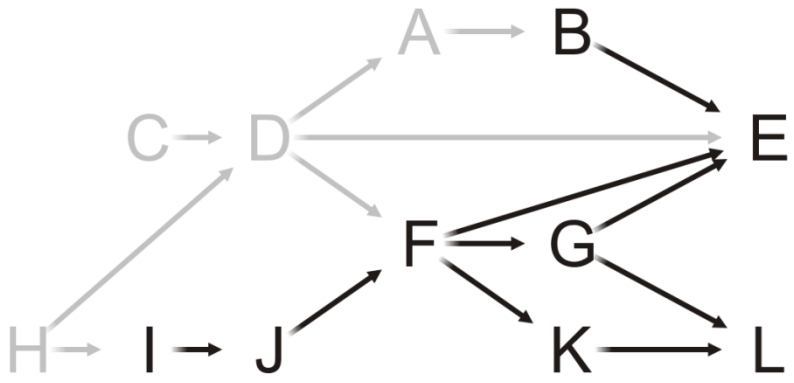
– Let's choose Task A



C, H, D

Example

Having removed A, what now?

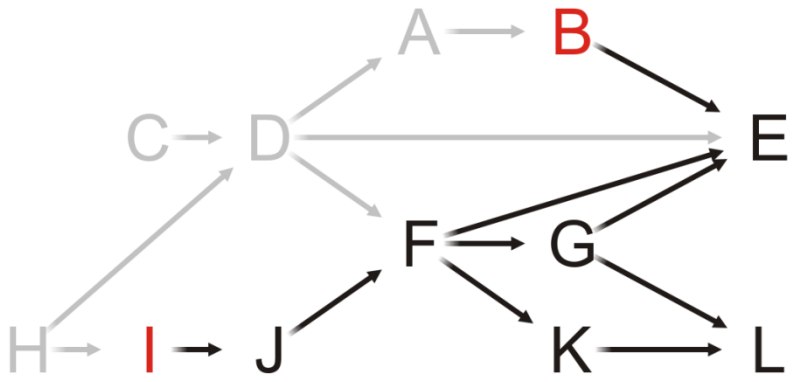


C, H, D, A

Example

Both Tasks B and I have in-degree zero

– Choose Task B

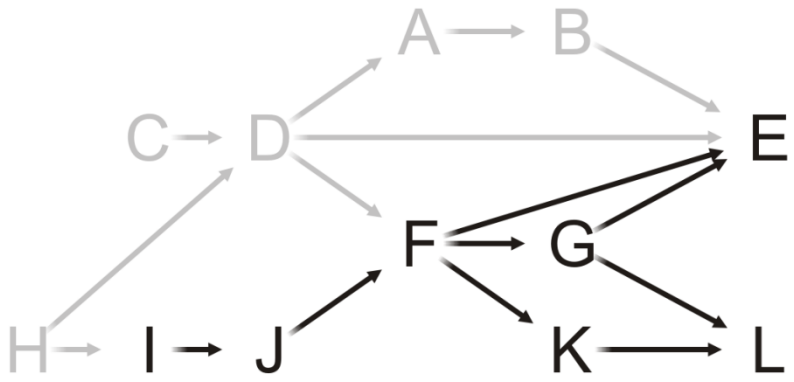


C, H, D, A

Example

Removing Task B, we note that Task E still has an in-degree of two

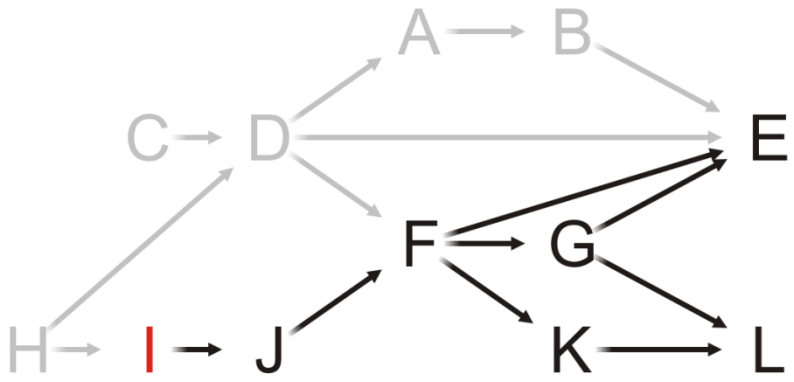
– Next?



C, H, D, A, B

Example

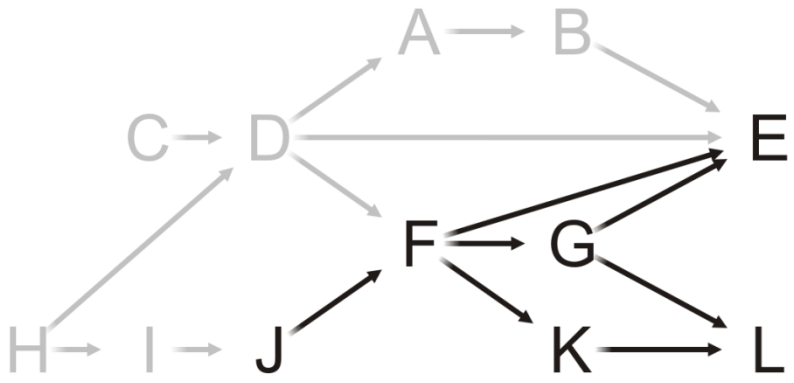
As only Task I has in-degree zero, we choose it



C, H, D, A, B

Example

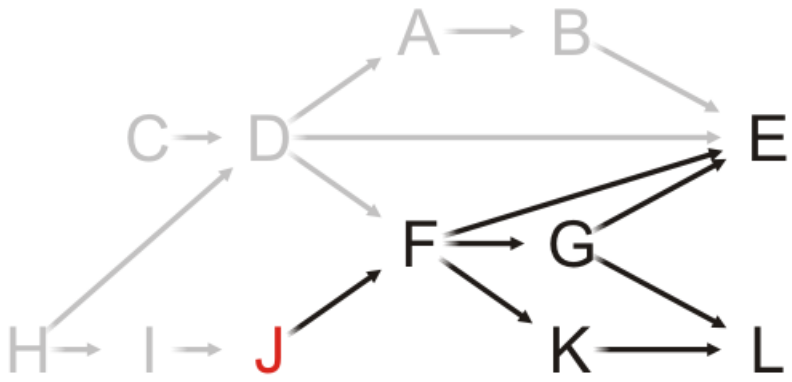
Having completed Task I, what now?



C, H, D, A, B, I

Example

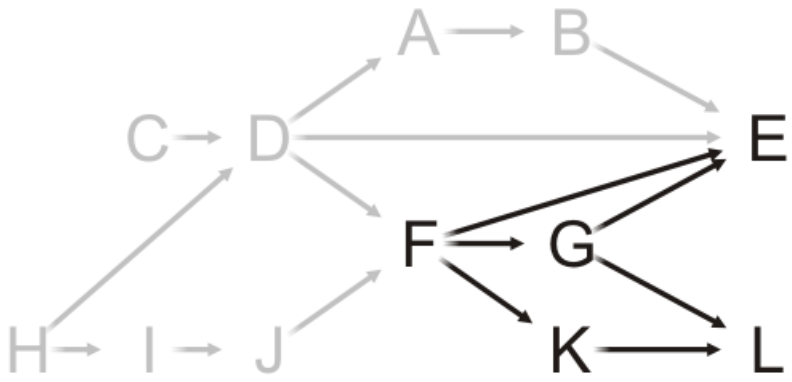
Only Task J has in-degree zero: choose it



C, H, D, A, B, I

Example

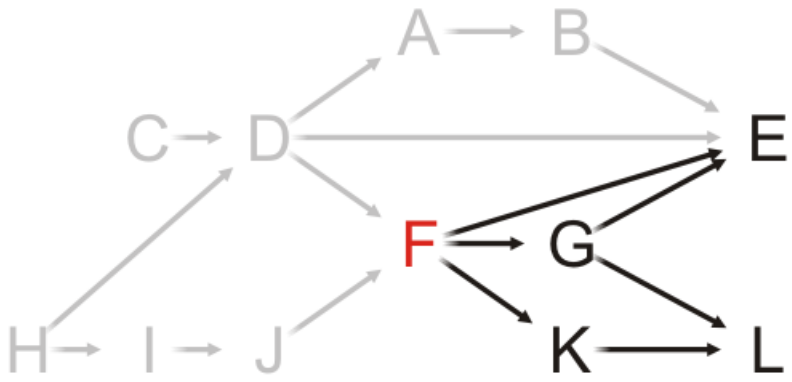
Having completed Task J, what now?



C, H, D, A, B, I, J

Example

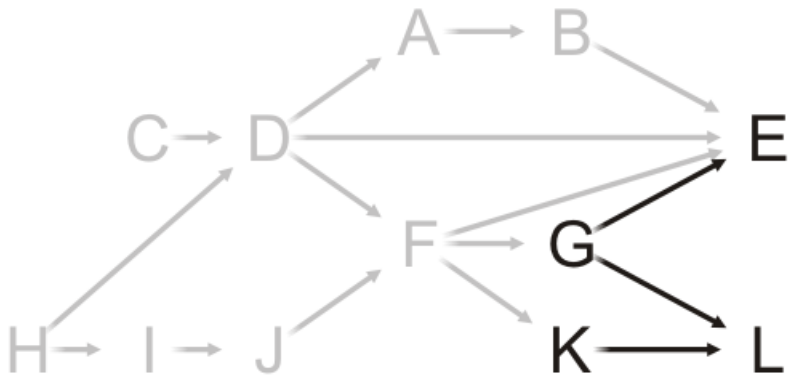
Only Task F can be completed, so choose it



C, H, D, A, B, I, J

Example

What choices do we have now?

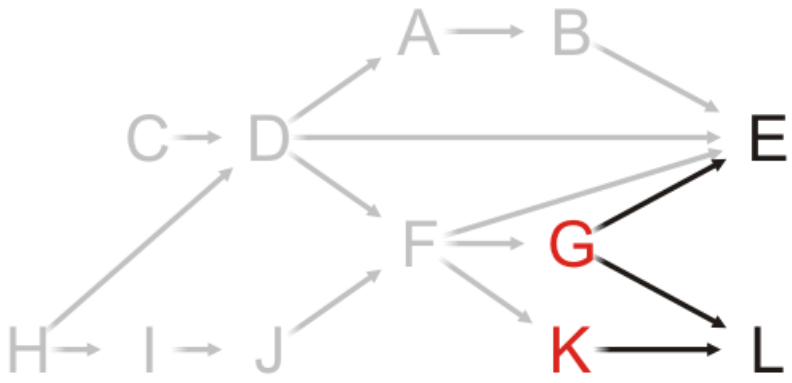


C, H, D, A, B, I, J, F

Example

We can perform Tasks G or K

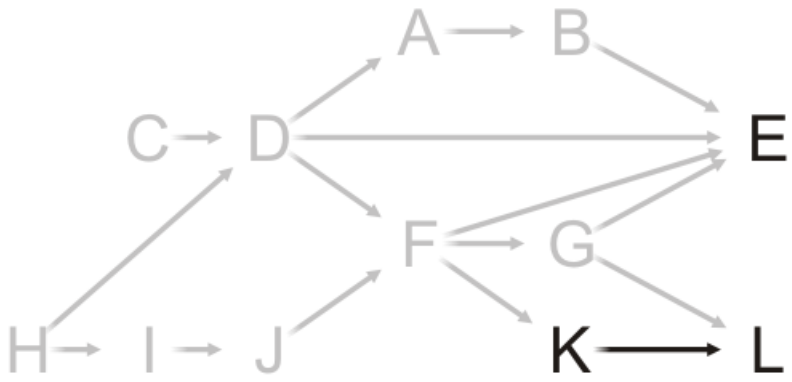
– Choose Task G



C, H, D, A, B, I, J, F

Example

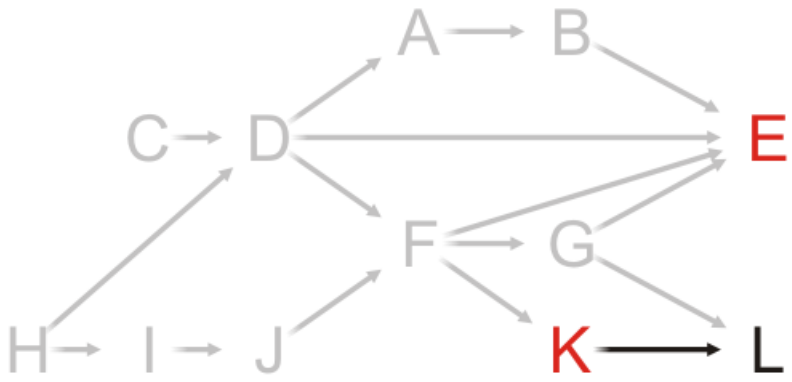
Having removed Task G from the graph, what next?



C, H, D, A, B, I, J, F, G

Example

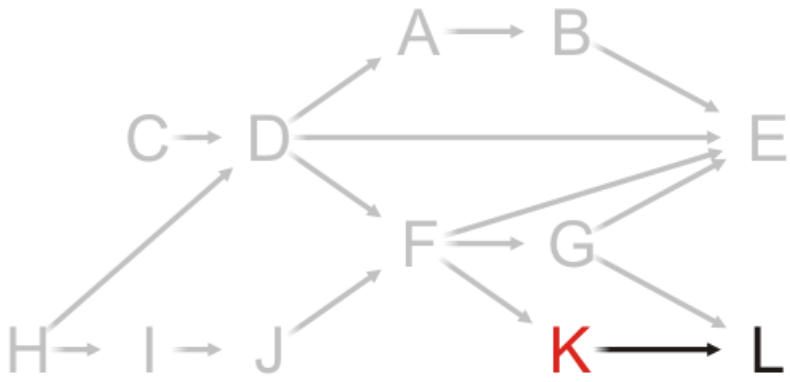
Choosing between Tasks E and K, choose Task E



C, H, D, A, B, I, J, F, G

Example

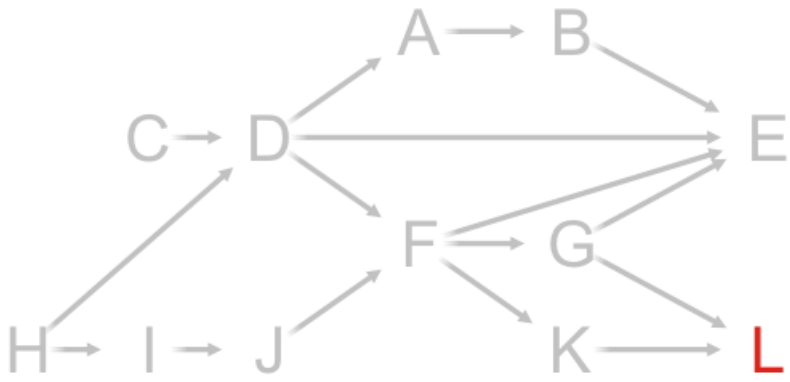
At this point, Task K is the only one that can be run



C, H, D, A, B, I, J, F, G, E

Example

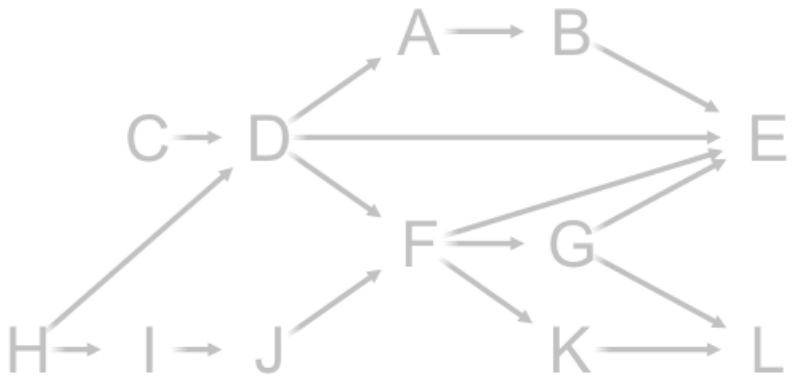
And now that both Tasks G and K are complete,
we can complete Task L



C, H, D, A, B, I, J, F, G, E, K

Example

There are no more vertices left

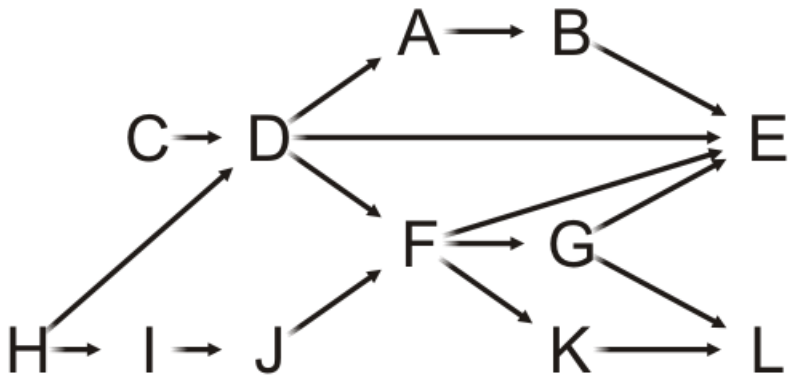


C, H, D, A, B, I, J, F, G, E, K, L

Example

Thus, one possible topological sort would be:

C, H, D, A, B, I, J, F, G, E, K, L

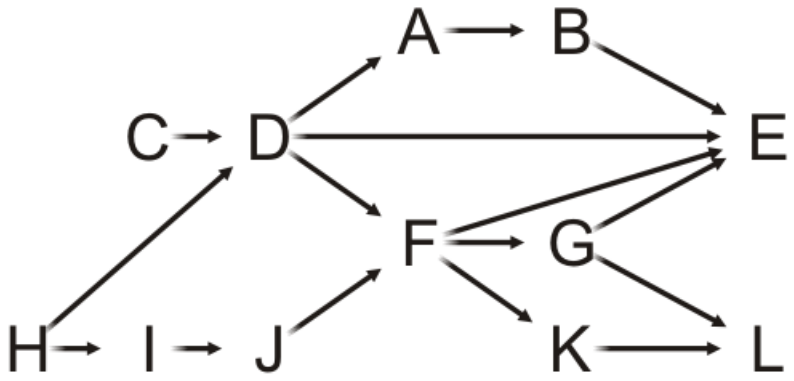


Example

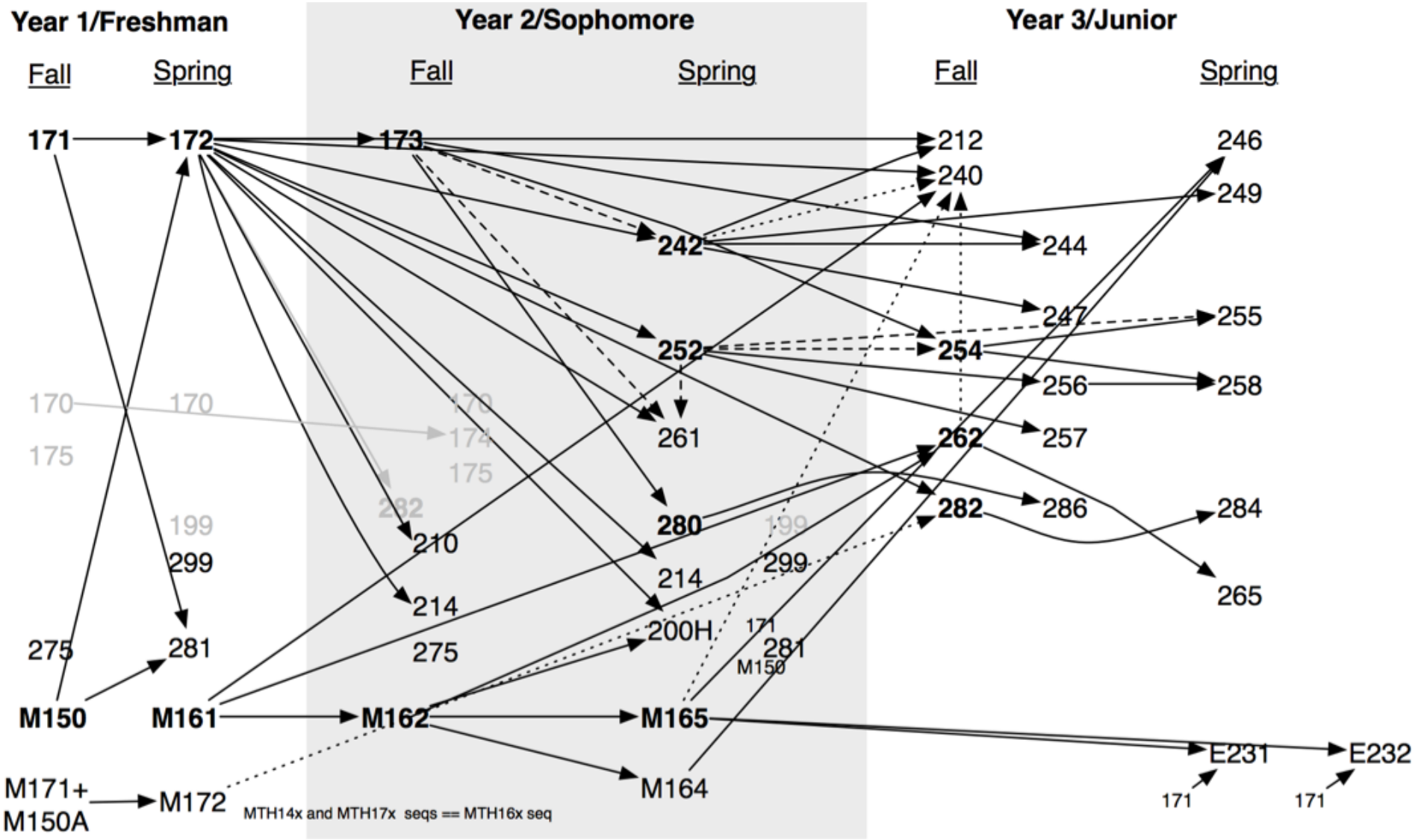
Note that topological sorts need not be unique:

C, H, D, A, B, I, J, F, G, E, K, L

H, I, J, C, D, F, G, K, L, A, B, E



The topological sort will be your roadmap to graduation



Summary

In this topic, we have discussed topological sorts

- Sorting of elements in a DAG

MINIMUM SPANNING TREE

Outline

In this topic, we will

- Define a spanning tree
- Define the weight of a spanning tree in a weighted graph
- Define a minimum spanning tree
- Consider applications
- List possible algorithms

Spanning trees

Given a connected graph with $|V| = n$ vertices, a spanning tree is defined a collection of $n - 1$ edges which connect all n vertices

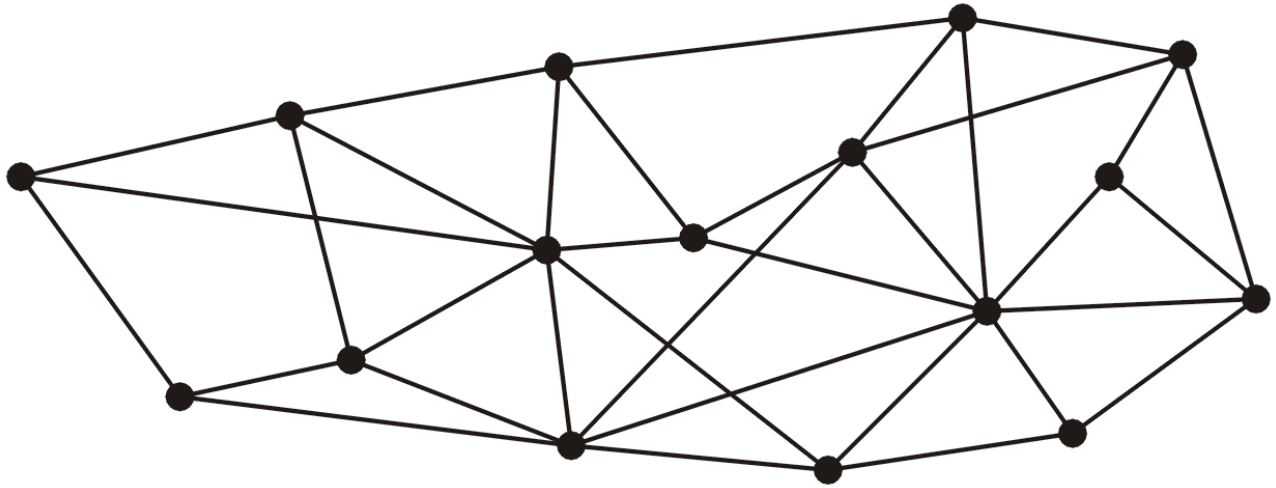
- The n vertices and $n - 1$ edges define a connected sub-graph

A spanning tree is not necessarily unique

A spanning tree is a tree which connects all the vertices

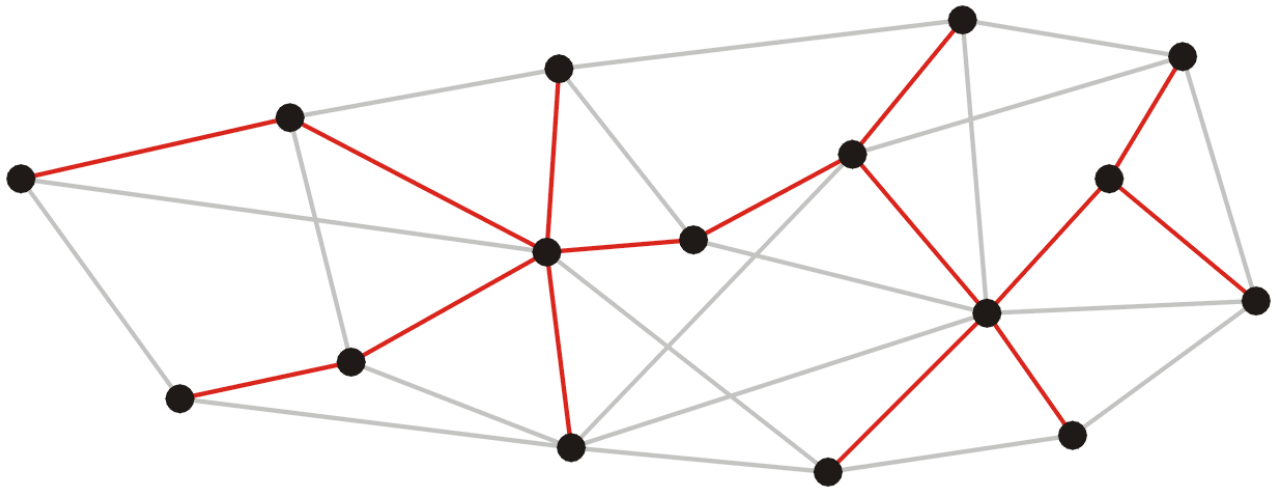
Spanning trees

This graph has 16 vertices and 35 edges



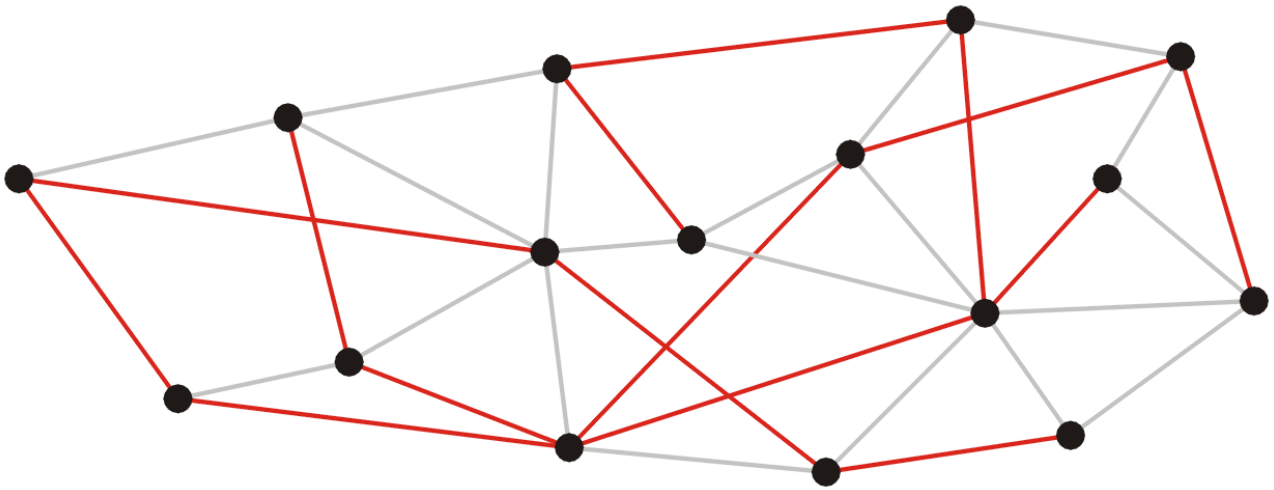
Spanning trees

These 15 edges form a minimum spanning tree



Spanning trees

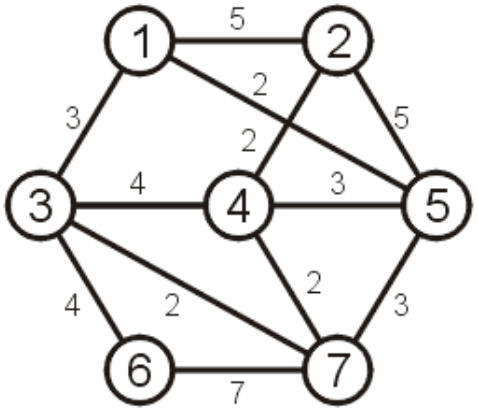
As do these 15 edges:

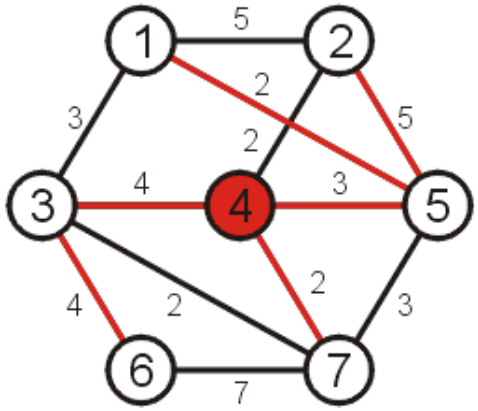


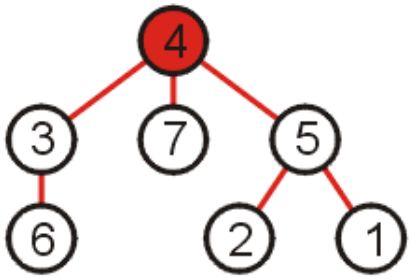
Spanning trees on weighted graphs

The weight of a spanning tree is the sum of the weights on all the edges which comprise the spanning tree

– The weight of this spanning tree is 20





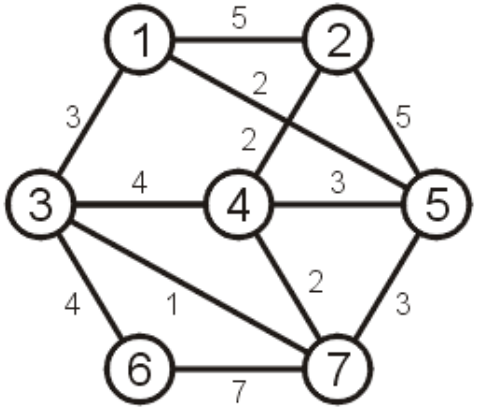


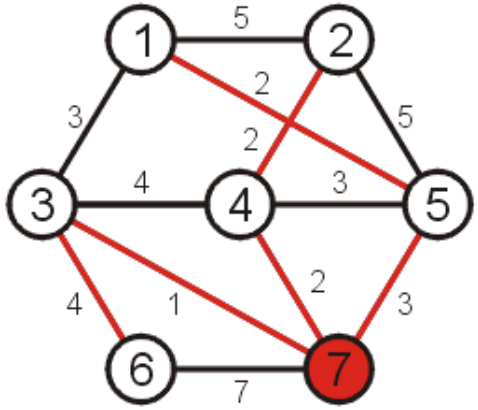
Minimum Spanning Trees

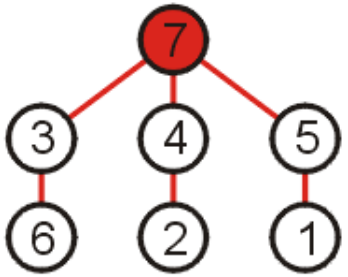
Which spanning tree which minimizes the weight?

– Such a tree is termed a *minimum spanning tree*

The weight of this spanning tree is 14

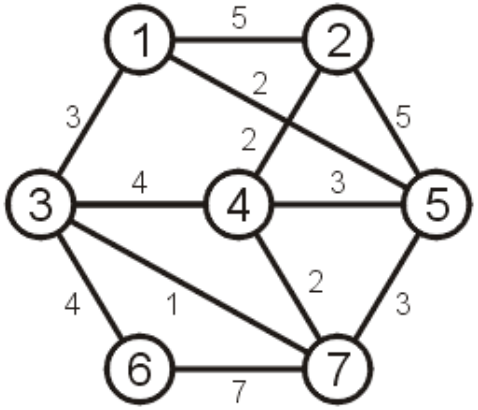


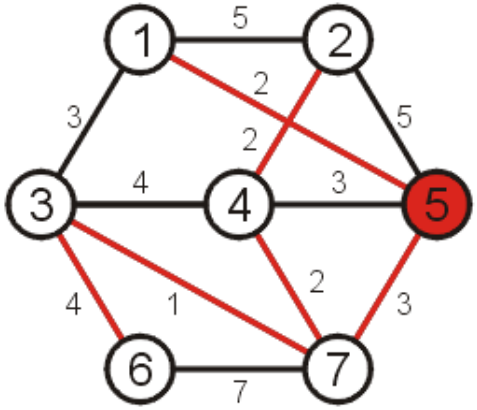


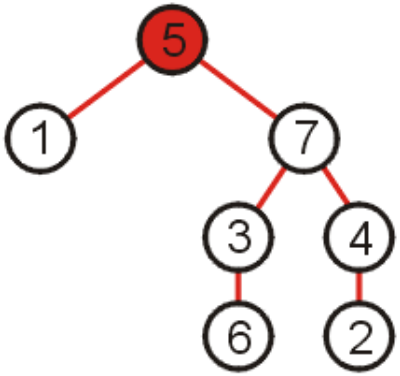


Minimum Spanning Trees

If we use a different vertex as the root, we get a different tree, however, this is simply the result of one or more rotations







Unweighted graphs

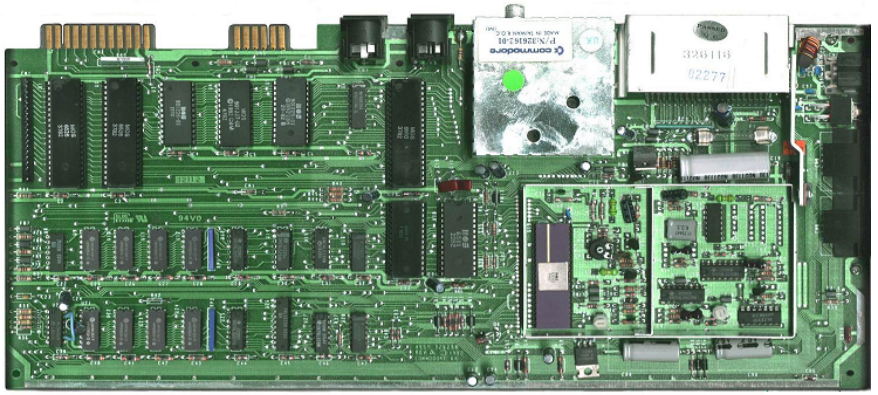
Observation

- In an unweighted graph, we nominally give each edge a weight of 1
- Consequently, all minimum spanning trees have weight $|V| - 1$

Application

Consider supplying power to

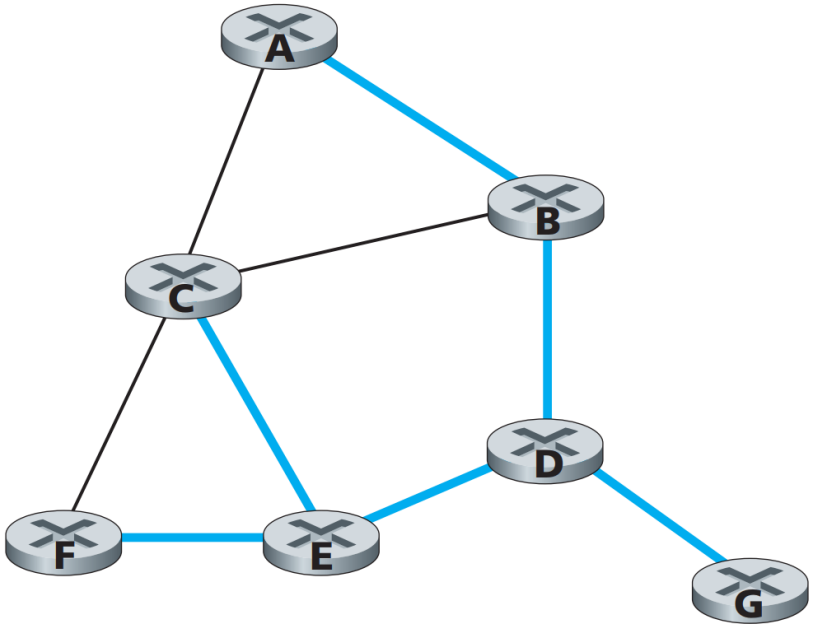
– All circuit elements on a board



Computer Networks:

– Broadcasting messages to all nodes

A minimum spanning tree will give the lowest-cost solution



Application

Consider attempting to find the best means of connecting a number of LANs

- Minimize the number of bridges
- Costs not strictly dependant on distances

Application

A minimum spanning tree will provide the optimal solution

Application

Consider an *ad hoc* wireless network

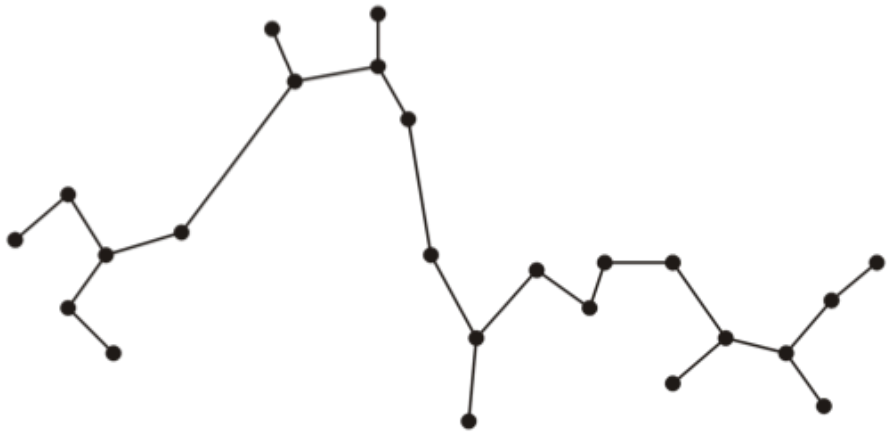
- Any two terminals can connect with any others

Problem:

- Errors in transmission increase with transmission length
- Can we find clusters of terminals which can communicate safely?

Application

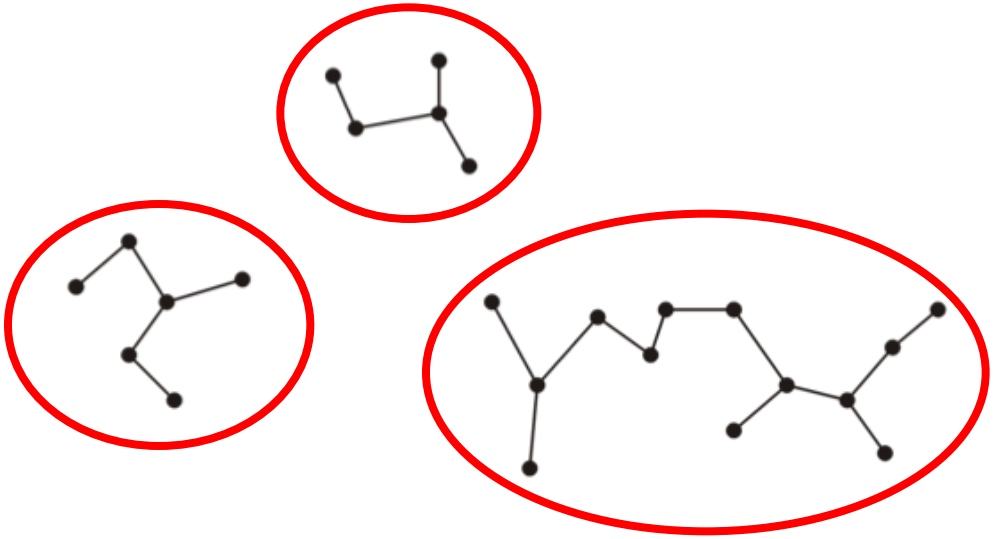
Find a minimum spanning tree



Application

Remove connections which are too long

This *clusters* terminals into smaller and more manageable sub-networks



Algorithms

Two common algorithms for finding minimum spanning trees are:

- Prim's algorithm (Next topic)
- Kruskal's algorithm (We won't cover)

Summary

This topic covered

- The definition of spanning trees, weighted graphs, and minimum spanning trees
- Applications generally involve networks (electrical or communications)

PRIM'S ALGORITHM

Outline

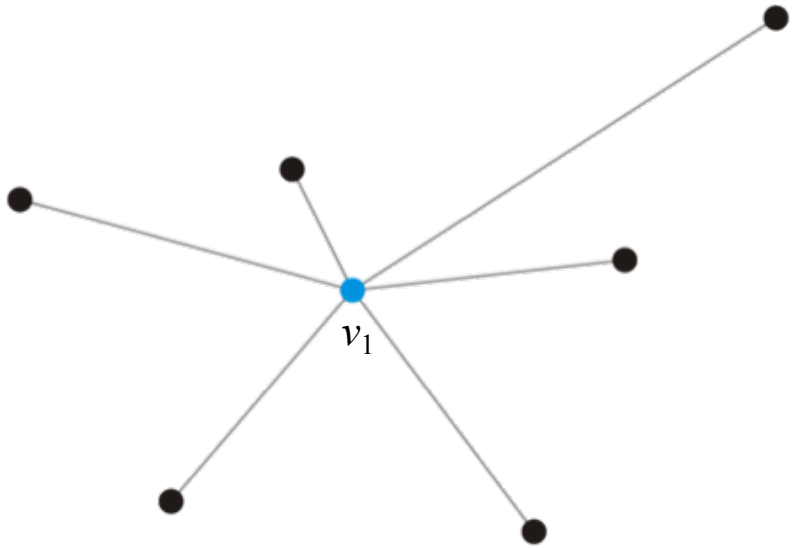
This topic covers Prim's algorithm:

- Finding a minimum spanning tree
- The idea and the algorithm
- An example

Strategy

Suppose we take a vertex

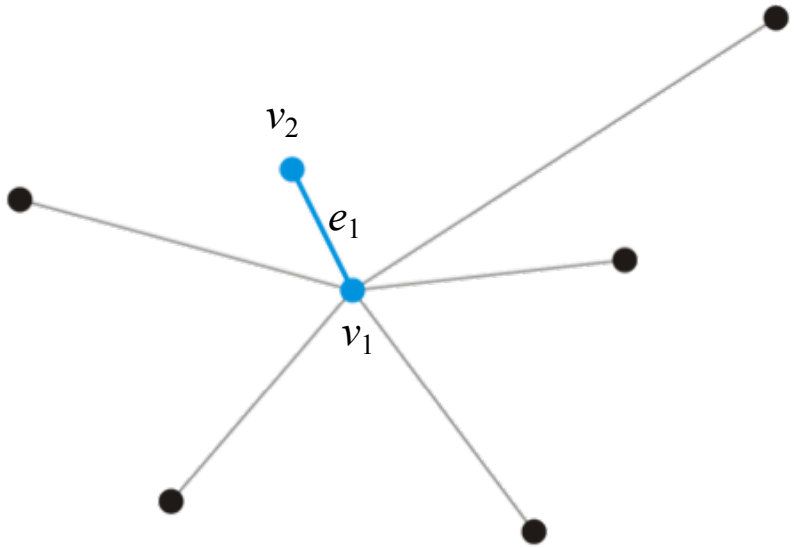
– Given a single vertex v_1, it forms a minimum spanning tree on one vertex



Strategy

Add that adjacent vertex v_2 that has a connecting edge e_1 of minimum weight

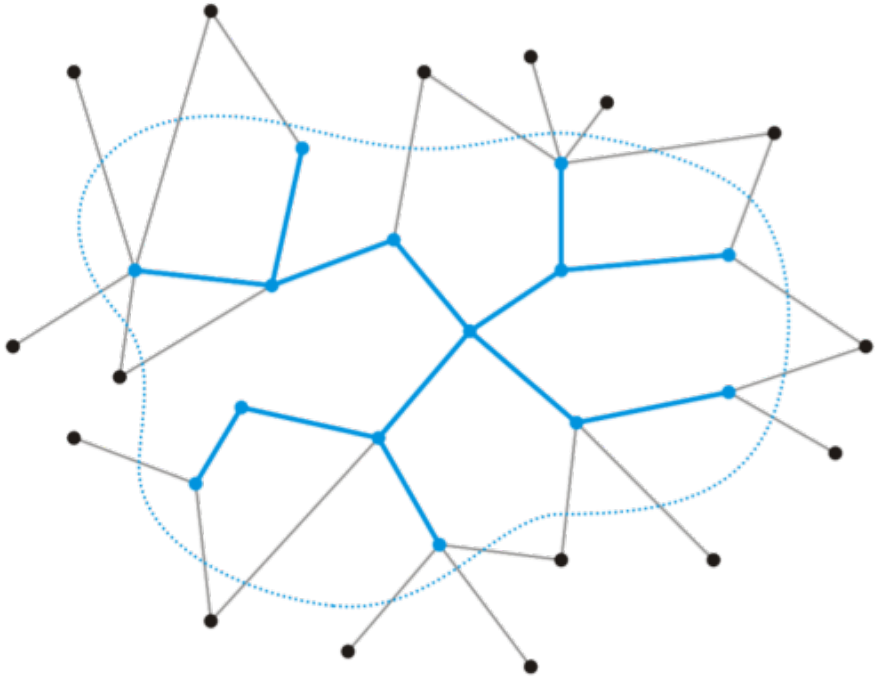
– This forms a minimum spanning tree on our two vertices and e_1 must be in any minimum spanning tree containing the vertices v_1 and v_2



Strategy

Strategy:

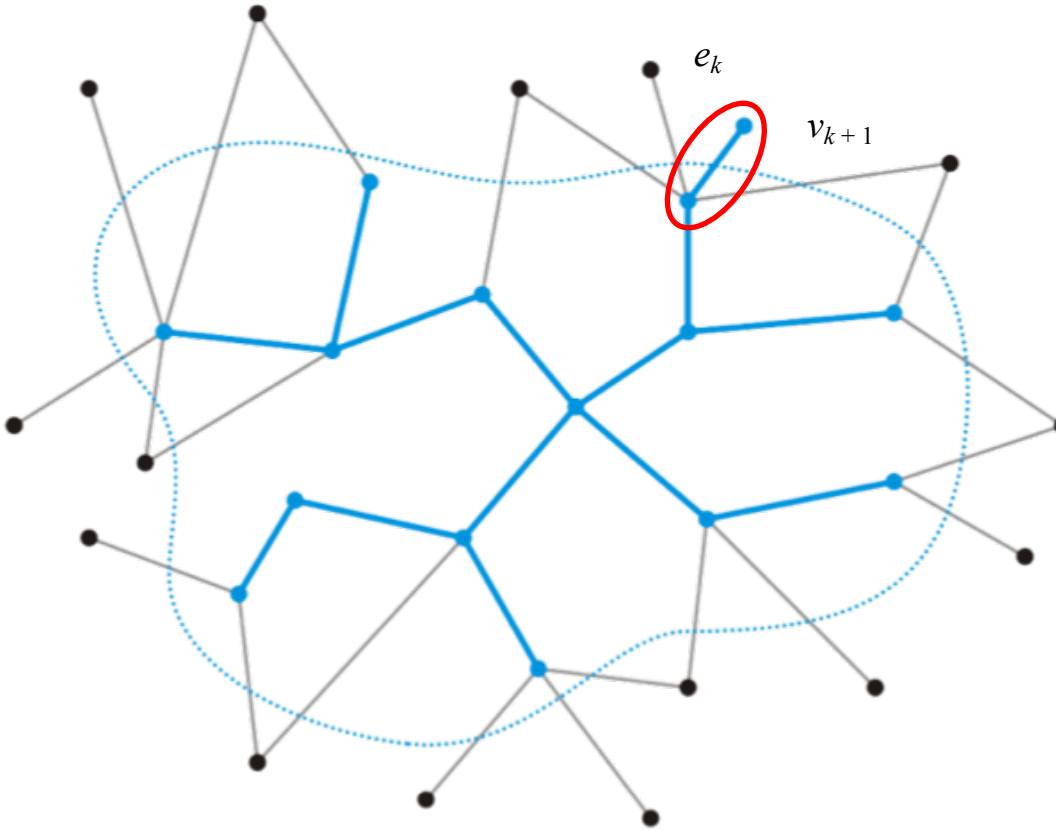
- Suppose we have a known minimum spanning tree on $k < n$ vertices
- How could we extend this minimum spanning tree?



Strategy

Add that edge e_k with least weight that connects this minimum spanning tree to a new vertex v_{k+1}

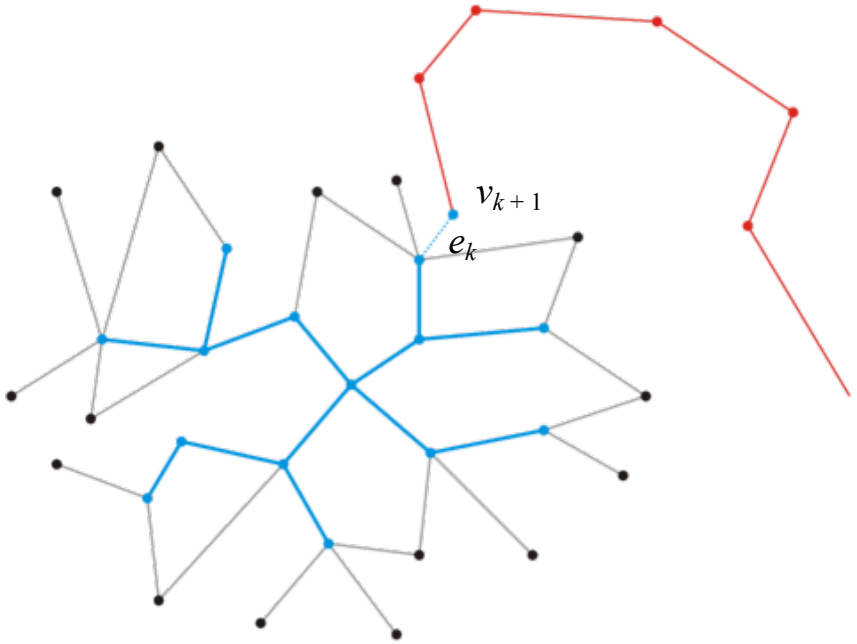
- This does create a minimum spanning tree on $k+1$ nodes—there is no other edge we could add that would connect this vertex
- Does the new edge, however, belong to the minimum spanning tree on all n vertices?



Strategy

Suppose it does not

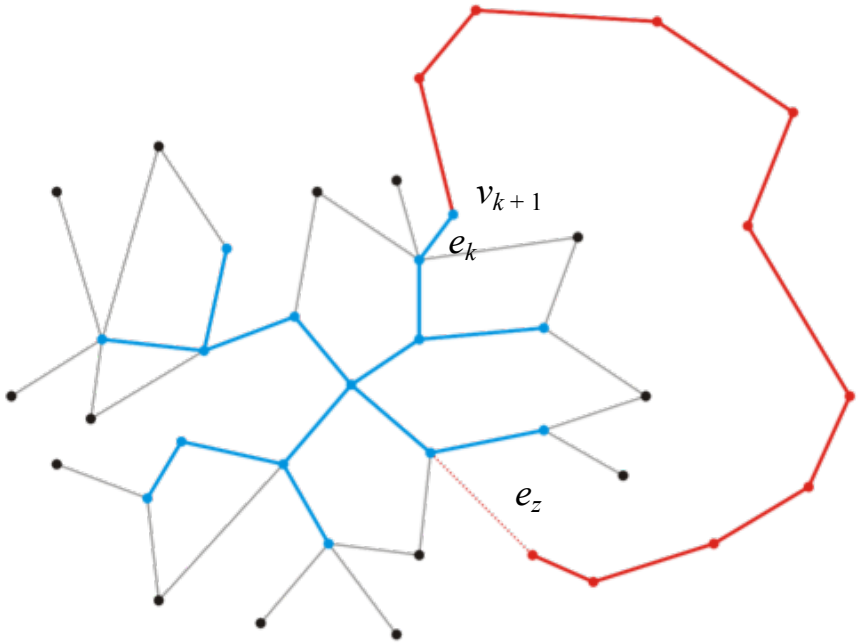
– Thus, vertex v_{k+1} is connected to the minimum spanning tree via another sequence of edges



Strategy

Where at least one edge (let's assume e_z) would be greater than e_k

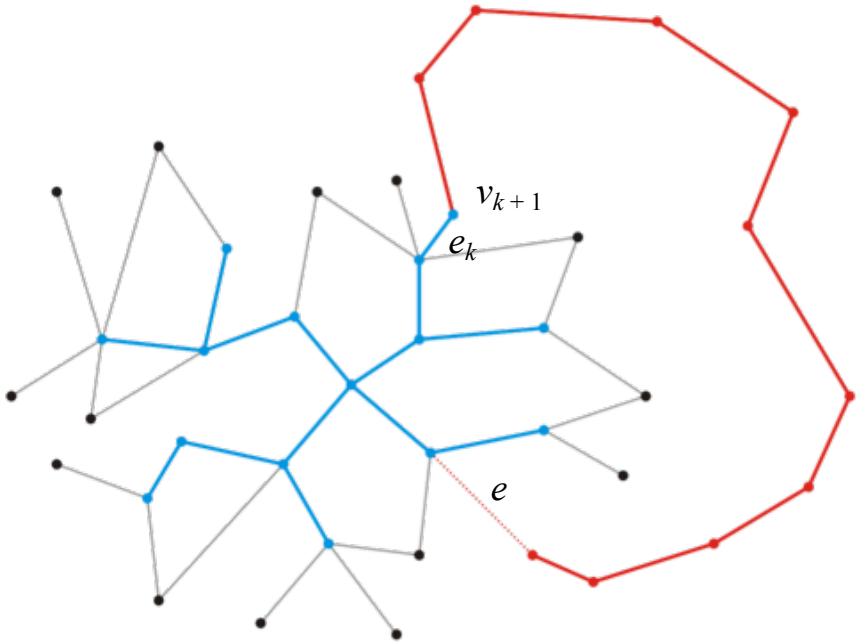
Therefore, our minimum spanning tree must contain e_k



$e_z \geq e_k$

Strategy

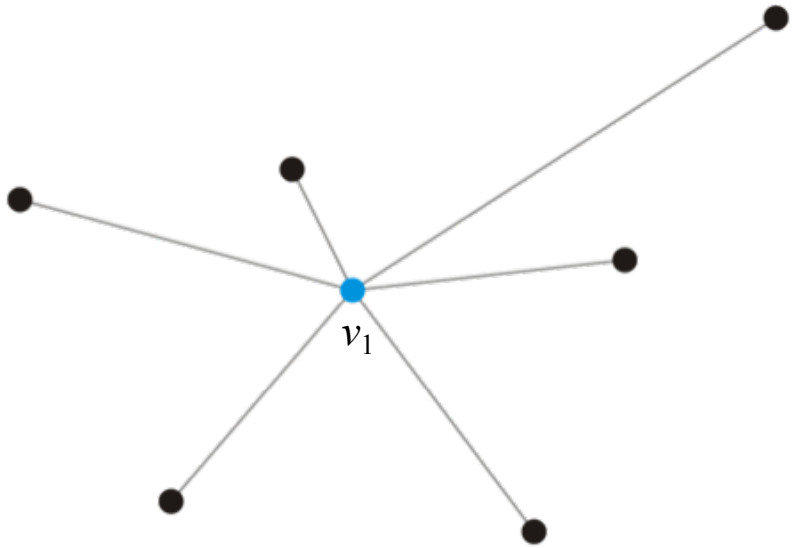
Recall that we did not prescribe the value of k, and thus, k could be any value, including $k = 1$



Strategy

Recall that we did not prescribe the value of k, and thus, k could be any value, including $k = 1$

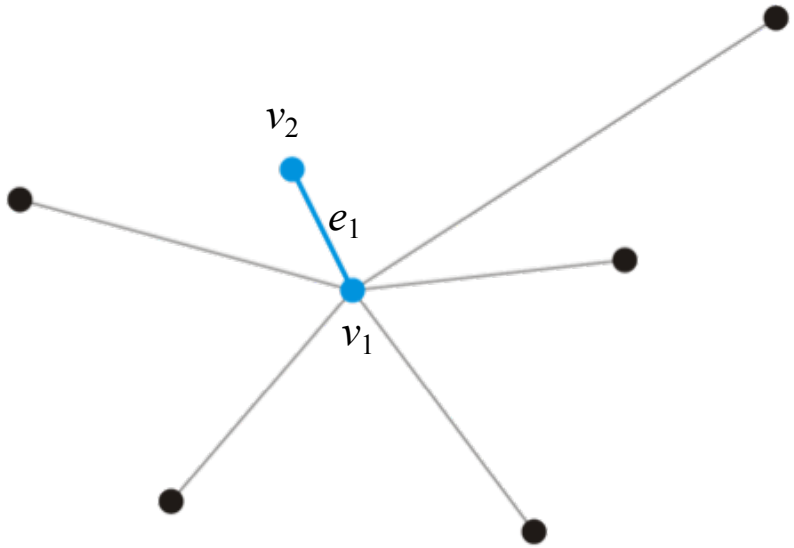
– Given a single vertex v_1, it forms a minimum spanning tree on one vertex



Strategy

Add that adjacent vertex v_2 that has a connecting edge e_1 of minimum weight

– This forms a minimum spanning tree on our two vertices and e_1 must be in any minimum spanning tree containing the vertices v_1 and v_2



Minimum Spanning Trees

Prim's algorithm for finding the minimum spanning tree states:

- Start with an arbitrary vertex to form a minimum spanning tree on one vertex
- At each step, add that vertex v not yet in the minimum spanning tree that has an edge with least weight that connects v to the existing minimum spanning sub-tree
- Continue until we have $n - 1$ edges and n vertices

Prim's Algorithm

Initialization:

- Select a root node and set its distance as 0
- Set the distance to all other vertices as ∞
- Set all vertices to being unvisited
- Set the parent of all vertices to null

Prim's Algorithm

Iterate while there exists an unvisited vertex with distance < ∞

- Select that unvisited vertex with minimum distance
- Mark that vertex as having been visited
- For each adjacent vertex, if the weight of the connecting edge is less than the current distance to that vertex:
  - Update the distance to equal the weight of the edge
  - Set the current vertex as the parent of the adjacent vertex

Prim's Algorithm

Halting Conditions:

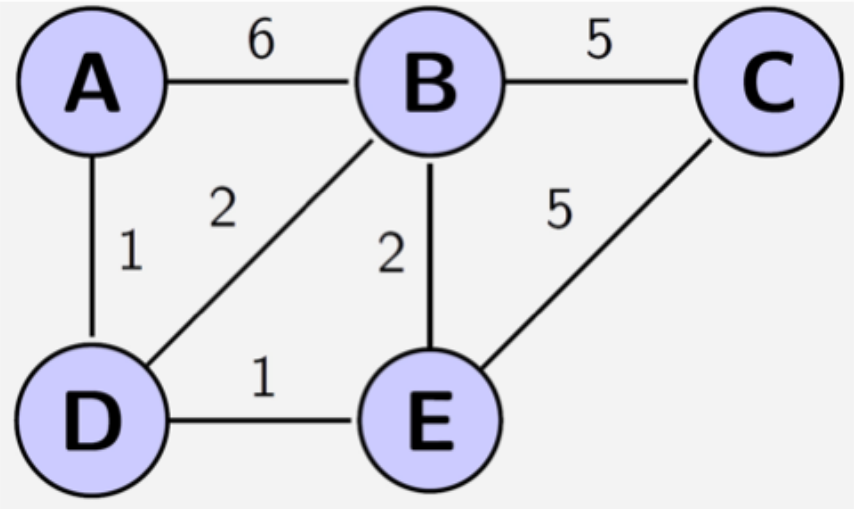
- There are no unvisited vertices
  - which have a distance < ∞ (important for unconnected graph)

If all vertices have been visited, we have a spanning tree of the entire graph

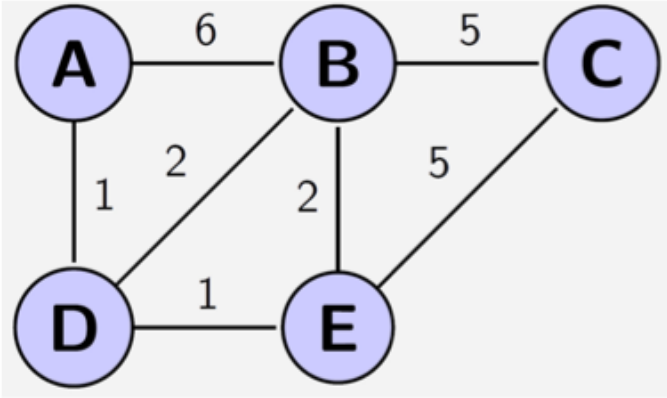
If there are vertices with distance ∞, then the graph is not connected and we only have a minimum spanning tree of the connected sub-graph containing the root

Prim's Algorithm

Let us find the minimum spanning tree for the following undirected weighted graph

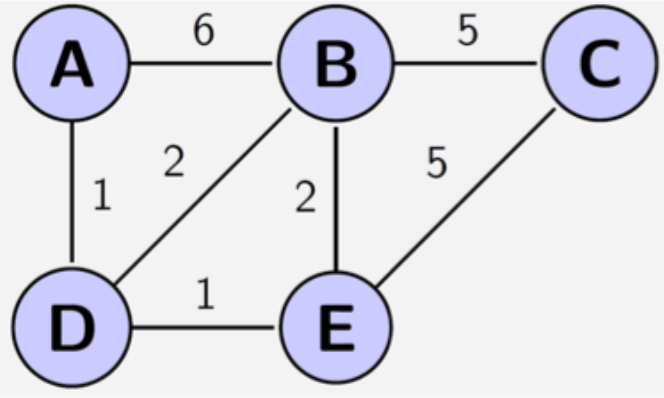


Prim's Algorithm



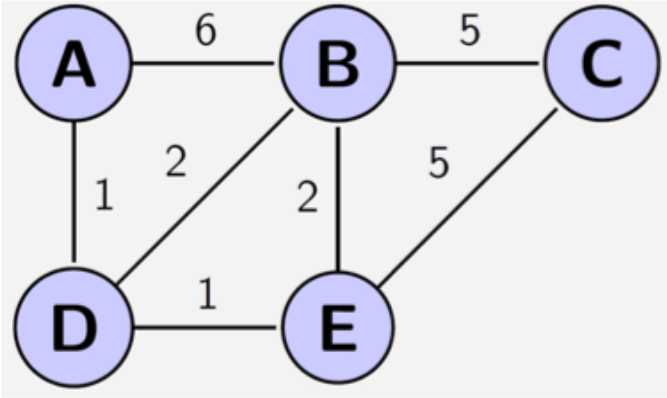
	A	B	C	D	E
A	0,A	inf	inf	inf	inf

Prim's Algorithm



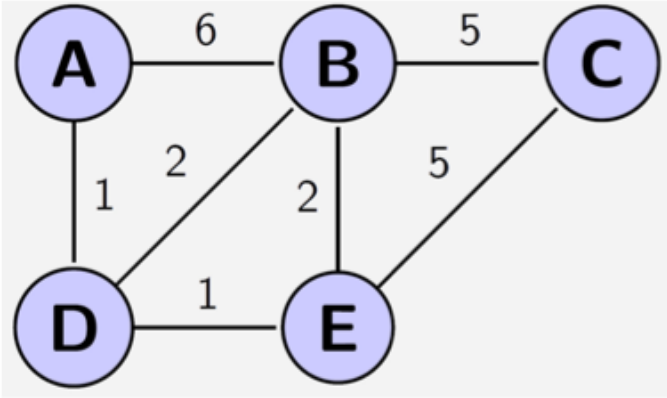
	A	B	C	D	E
A	0,A	6,A	inf	1,A	inf

Prim's Algorithm



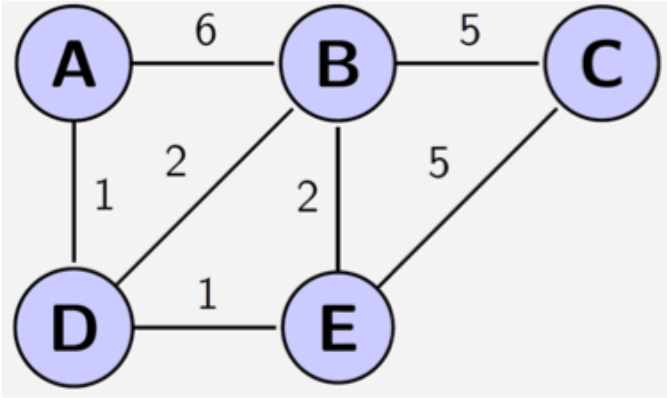
	A	B	C	D	E
A	0,A	6,A	inf	1,A	inf
D,A		2,D	inf		1,D

Prim's Algorithm



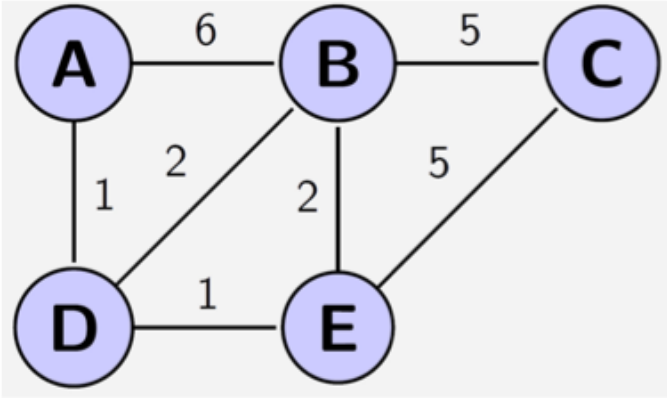
	A	B	C	D	E
A	0,A	6,A	inf	1,A	inf
D,A		2,D	inf		1,D
E,D		2,D	5,E		

Prim's Algorithm



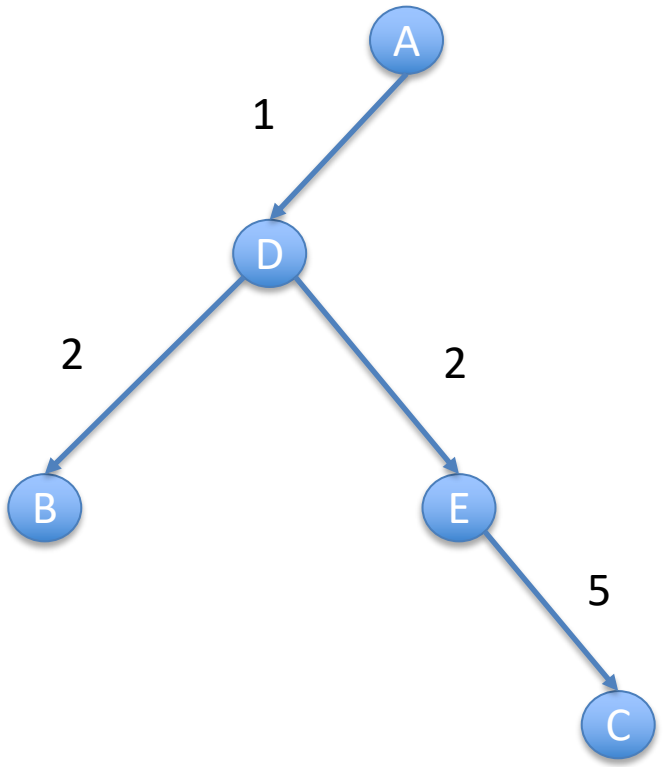
	A	B	C	D	E
A	0,A	6,A	inf	1,A	inf
D,A		2,D	inf		1,D
E,D		2,D	5,E		
B,D			5,E		

Prim's Algorithm



	A	B	C	D	E
A	0,A	6,A	inf	1,A	inf
D,A		2,D	inf		1,D
E,D		2,D	5,E		
B,D			5,E		
C,E					

Minimal Spanning Tree



Prim's Algorithm

To summarize:

- we begin with a vertex which represents the root
- starting with this trivial tree and iteration, we find the shortest edge which we can add to this already existing tree to expand it

This is a reasonably efficient algorithm: the number of visits to vertices is kept to a minimum

Summary

We have seen an algorithm for finding minimum spanning trees

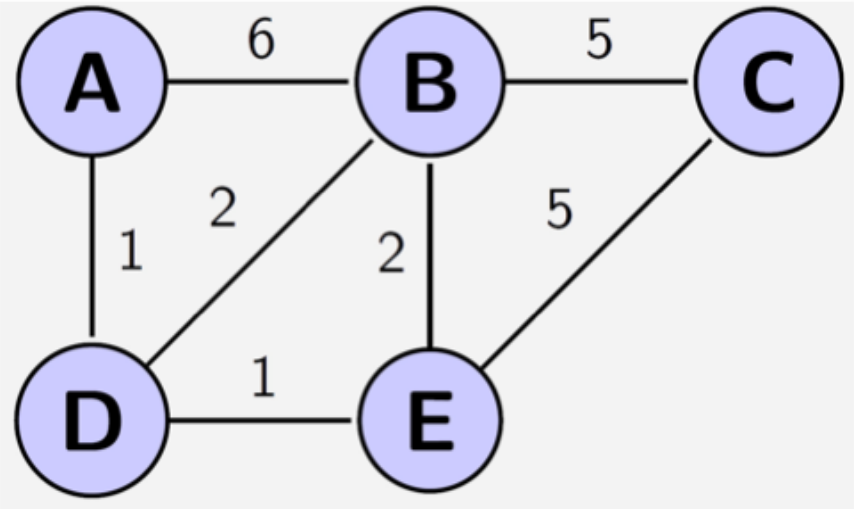
- Start with a trivial minimum spanning tree and grow it

Prim's algorithm finds an edge with least weight which grows an already existing tree

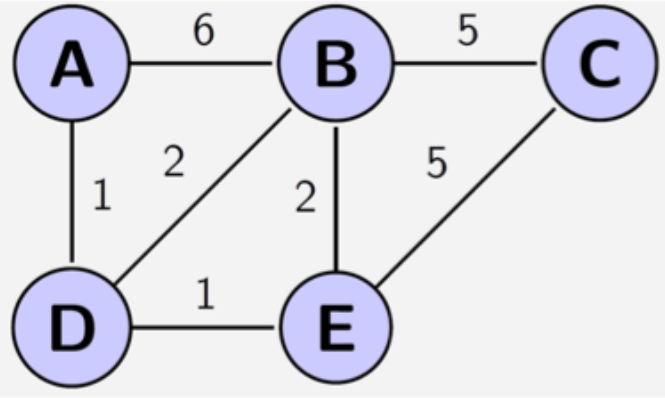
DIJKSTRA'S ALGORITHM

Dijkstra's algorithm

Let us find the shortest distance from A to every other nodes for the following undirected weighted graph

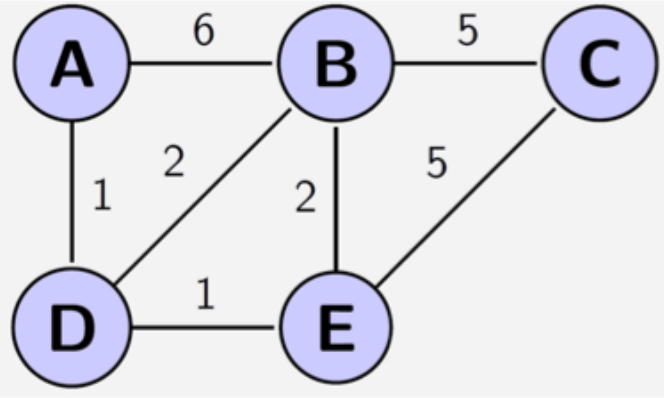


Dijkstra's algorithm



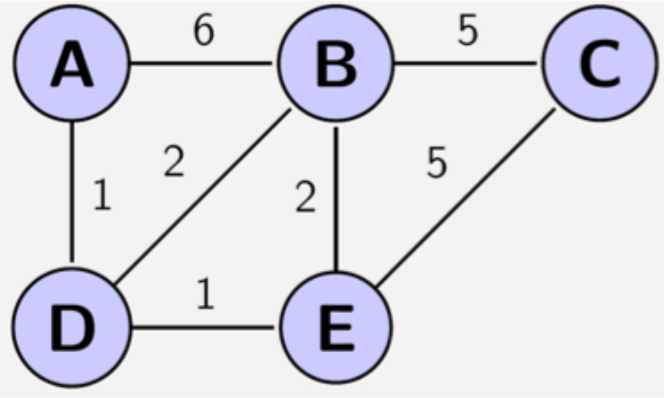
	A	B	C	D	E
A	0,A	inf	inf	inf	inf

Dijkstra's algorithm



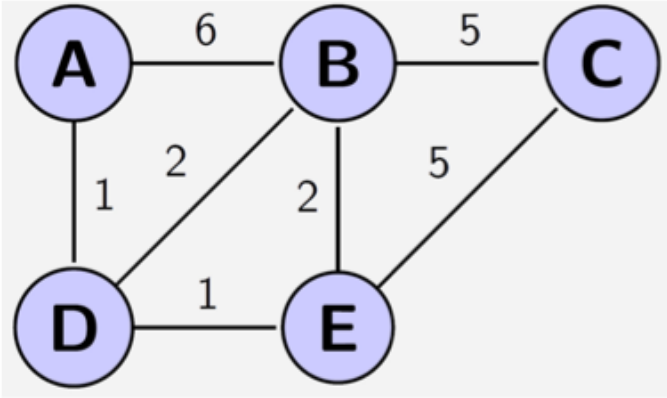
	A	B	C	D	E
A	0,A	6,A	inf	1,A	inf

Dijkstra's algorithm



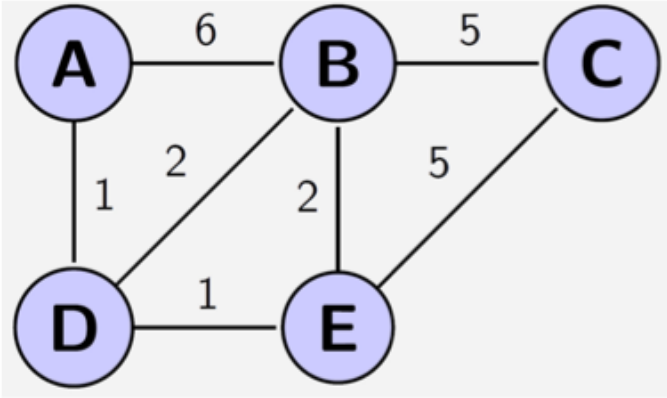
	A	B	C	D	E
A	0,A	6,A	inf	1,A	inf
D,A		3,D	inf		2,D

Dijkstra's algorithm



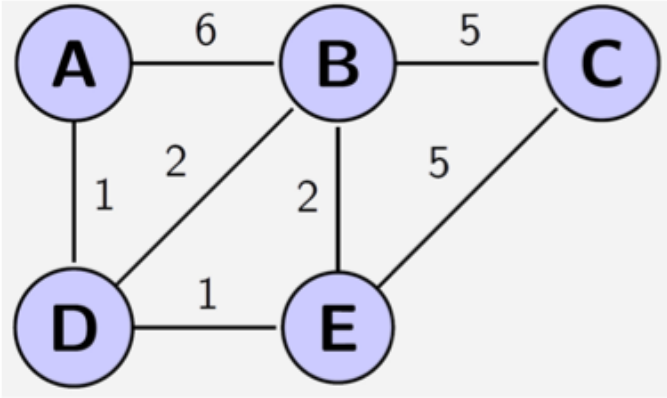
	A	B	C	D	E
A	0,A	6,A	inf	1,A	inf
D,A		3,D	inf		2,D
E,D		3,D	7,E		

Dijkstra's algorithm



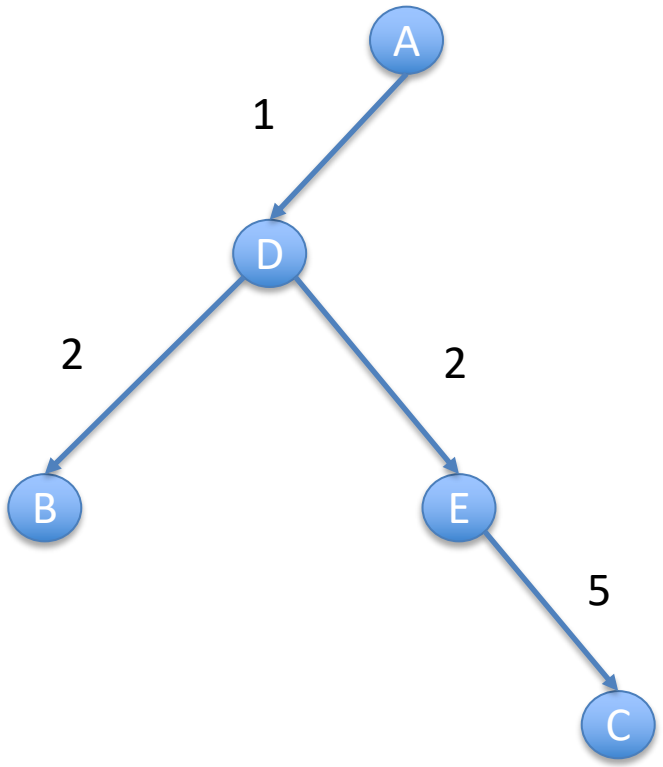
	A	B	C	D	E
A	0,A	6,A	inf	1,A	inf
D,A		3,D	inf		2,D
E,D		3,D	7,E		
B,D			7,E		

Dijkstra's algorithm



	A	B	C	D	E
A	0,A	6,A	inf	1,A	inf
D,A		3,D	inf		2,D
E,D		3,D	7,E		
B,D			7,E		
C,E					

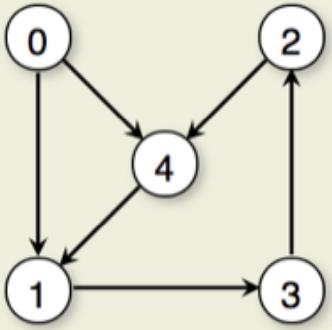
Dijkstra's algorithm

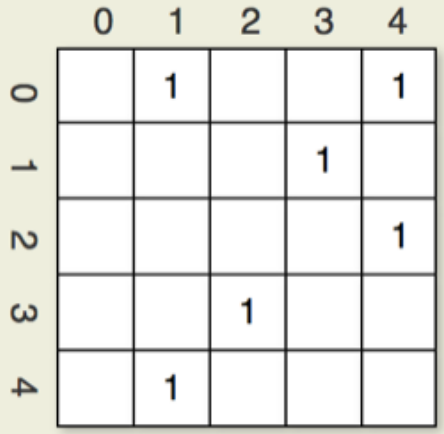


Dijkstra's vs Prim's

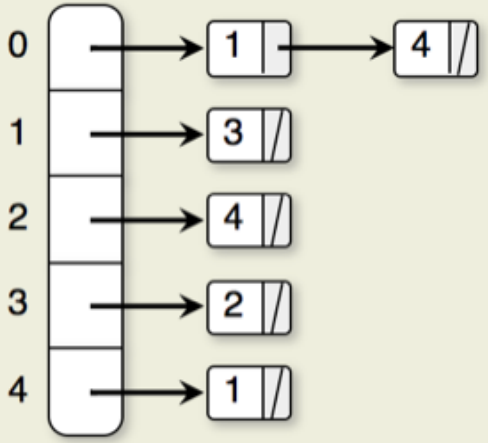
- A, B = 5
- B,C = 5
- A, C = 8

ANALYSIS OF DIJKSTRA'S ALGORITHM





Adajceny Matrix

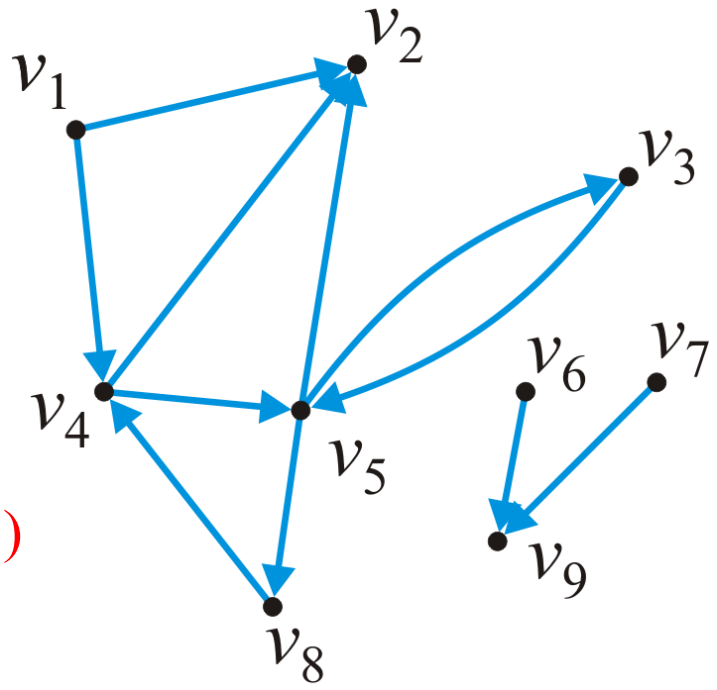


Adajceny List

Adjacency matrix

The matrix entry (j, k) is set to true if there is an edge (v_j, v_k)

	1	2	3	4	5	6	7	8	9
1		T		T					
2									
3					T				
4		T			T				
5		T	T					T	
6									T
7									T
8				T					
9									



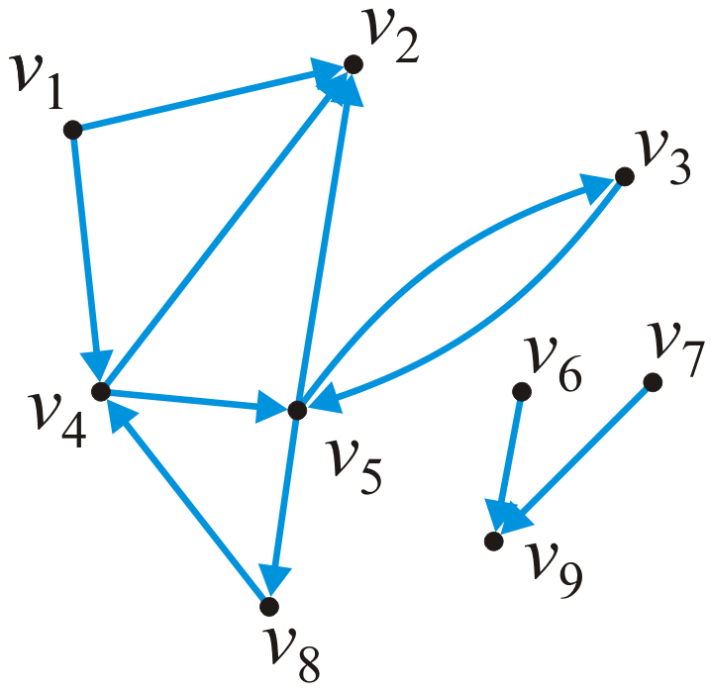
- Requires $\Theta(|V|^2)$ memory
- Determining if v_j is adjacent to v_k is $O(1)$
- Finding all neighbors of v_j is $\Theta(|V|)$

Adjacency list

Most efficient for algorithms is an adjacency list

- Each vertex is associated with a list of its neighbors

$$
\begin{array}{ll}
1 & \bullet \rightarrow 2 \rightarrow 4 \\
2 & \bullet \\
3 & \bullet \rightarrow 5 \\
4 & \bullet \rightarrow 2 \rightarrow 5 \\
5 & \bullet \rightarrow 2 \rightarrow 3 \rightarrow 8 \\
6 & \bullet \rightarrow 9 \\
7 & \bullet \rightarrow 9 \\
8 & \bullet \rightarrow 4 \\
9 & \bullet
\end{array}
$$



- Requires $\Theta(|V| + |E|)$ memory
- On average:
  - Determining if v_j is adjacent to v_k is $O\left(|E|/|V|\right)$
  - Finding all neighbors of v_j is $\Theta\left(|E|/|V|\right)$

– On average:
  - Determining if v_j is adjacent to v_k is $O\left(\left|E\right|/\left|V\right|\right)$

- checking this |V| number of times: O(|E|);

Implementation and analysis

The initialization requires $\Theta(|V|)$ memory and run time

We iterate $|V| - 1$ times, each time finding next closest vertex to the source

- Iterating through the table requires is $\Theta(|V|)$ time
- Each time we find a vertex, we must check all of its neighbors
- With an adjacency matrix, the run time is $\Theta(|V|(|V| + |V|)) = \Theta(|V|^2)$
- With an adjacency list, the run time is $\Theta(|V|^2 + |E|) = \Theta(|V|^2)$ as $|E| = \mathrm{O}(|V|^2)$

Can we do better?

- Recall, we only need the closest vertex
- How about a priority queue?

Pseudocode (Using PQ)

```
void dijkstra(s) {
  queue = new PriorityQueue<Vertex>();
  for (each vertex v) {
    v.dist = infinity;  // can use Integer.MAX_VALUE
    queue.enqueue(v);
    v.pred = null;
  }
  s.dist = 0;

  while (!queue.isEmpty()) {
    u = queue.extractMin();
    for (each vertex v adjacent to u)
      relax(u, v);
  }
}
```

```
void relax(u, v) {
  if (u.dist + w(u,v) < v.dist) {
    v.dist = u.dist + w(u,v);
    v.pred = u;
  }
}
```

Runtime: O(log |V|)

Source: http://www.cs.dartmouth.edu/~thc/cs10/lectures/0509/0509.html

Let's find the runtime

```
while (!queue.isEmpty()) {
  u = queue.extractMin();
  for (each vertex v adjacent to u)
    relax(u, v);
}
```

while (!queue.isEmpty()) { // O(|V|)
    u = queue.extractMin(); // O(log |V|)

while (!queue.isEmpty()) {
    for (each vertex v adjacent to u) {

How many times?:
We are traversing the whole adjacency list.
So, O(|E|)

relax(u, v); Runtime: O(log |V|)

So: total runtime = O(|V|log |V| + |E|log|V|) = O(|E|log|V|)

Implementation and analysis

The initialization still requires $\Theta(|V|)$ memory and run time
- The priority queue will also requires $O(|V|)$ memory
- We must use an adjacency list, not an adjacency matrix

We iterate $|V| - 1$ times, each time finding the *closest* vertex to the source
- Place the distances into a priority queue
- The size of the priority queue is $O(|V|)$
- Thus, the work required for this is $O(|V| \log(|V|))$

Is this all the work that is necessary?
- Recall that each edge visited may result in updating the priority queue
- Thus, the work required for this is $O(|E| \log(|V|))$

Thus, the total run time is $O(|V| \ln(|V|) + |E| \log(|V|)) = O(|E| \log(|V|))$

Summary

We have seen an algorithm for finding single-source shortest paths

- Start with the initial vertex
- Continue finding the next vertex that is closest

Dijkstra's algorithm always finds the next closest vertex

- It solves the problem in $O(|E| \log(|V|))$ time using Priority Queue
- The algorithm can be further improved to run in time $O((|V|\log(|V|) + |E|)$ --- (using Fibonacci Heap; We won't cover it)

Acknowledgement

- A lot of these slides are taken from ECE 250 *Algorithms and Data Structures* course (University of Waterloo) offered by Prof. Douglas W. Harder.