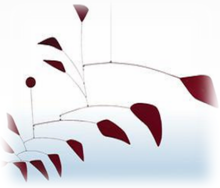
EECS 3101 - Design and Analysis of Algorithms



Shahin Kamali

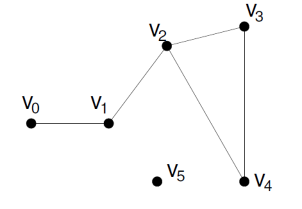
Topic 6 - Graph Algorithms

Overview

- Graph Applications & Representation
- Breadth-First Search and Depth-First Search
- Minimum Spanning Trees
- Shortest Path Algorithms

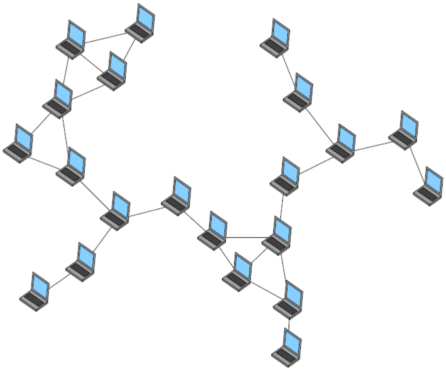
Graph Definition

- A graph $G = (V, E)$ consists of:
  - a set of **vertices**, V, representing objects in a set
  - a set of **edges**, $E \in V \times V$.
- A vertex is usually represented by a point.
- An edge (u, v) is usually represented by a line segment from u to v.



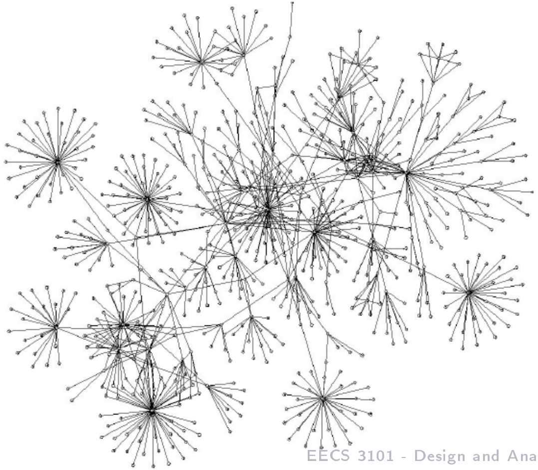
Graph Applications

- **Computer Networks:** pairs of computers (vertices) joined by a network connection (edge).



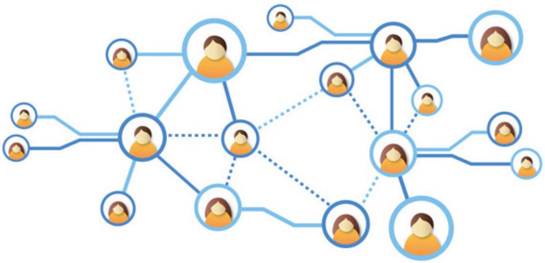
Graph Applications

- **World Wide Web:** pairs of web pages (vertices) joined by a hyperlink (edge).



Graph Applications

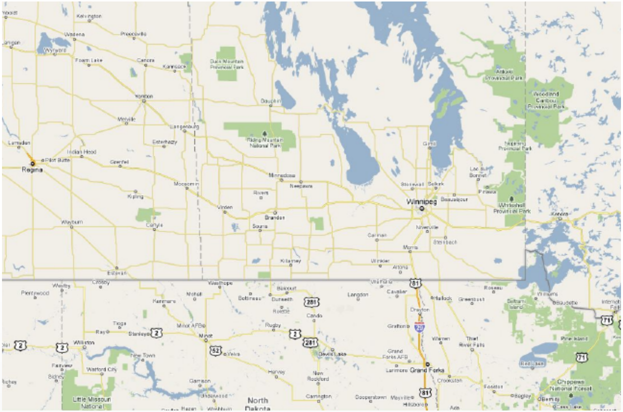
- **Social networks:** pairs of users (vertices) joined by a friendship-relation (edge).



https://www.google.com/url?sa=i&source=images&cd=&cad=rja&uact=8&ved=2ahUKEwirgef_K7hAhVin-AKHevaA1UQjRx6BAgBEAU&url=http%3A%2F%2Fblog.soton.ac.uk%2Fskillted%2F2015%2F04%2F05%2Fgraph-theory-for-skillted%2F&psig=AOvVaw3pQ6sgNWv7y1GGIrpJnG8T&ust=1554210749566524

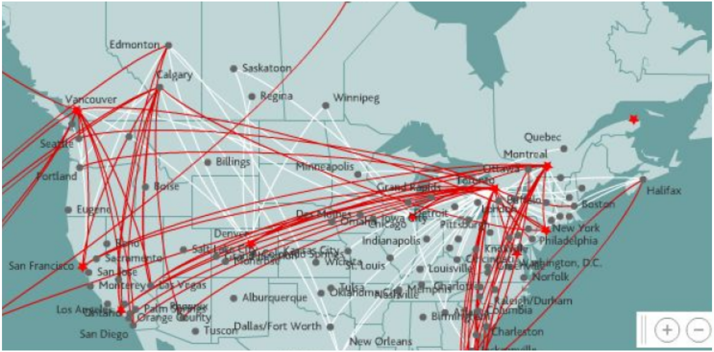
Graph Applications

- **Road networks:** pairs of locations (vertices) joined by a road (edge).



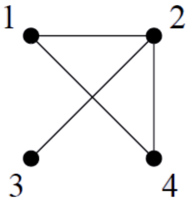
Graph Applications

- **Air map:** pairs of cities (vertices) joined by a direct flight (edge).

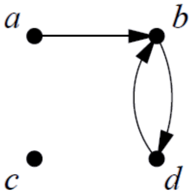


Undirected vs Directed Graphs

- In **undirected graphs**, there is no direction for edges.
- In **directed graphs**, also called **digraphs**, edges have one-way direction.
    - (u, v) and (v, u) are distinct possible edges between vertices v and u.



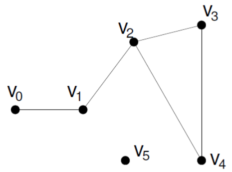
An undirected graph

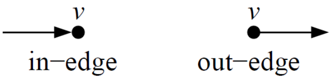


a directed graph

Terminology

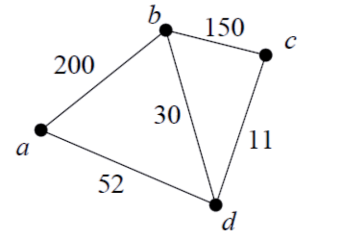
- An edge $e = (v, w)$ is **incident** on vertices v and w.
- v and w are said to be **adjacent** or **neighbouring** vertices.
- An edge coming from a vertex u into vertex v is called an **in-edge** of v.
- Conversely, an edge going from vertex v out to a vertex u is described as an **out-edge** of v.





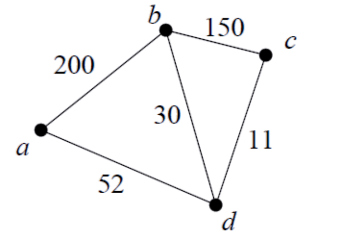
Weighted Graphs

- A numerical value may be assigned to every edge to form a **weighted graph**.



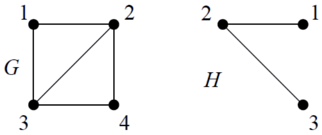
Weighted Graphs

- A numerical value may be assigned to every edge to form a **weighted graph**.
- Edge weight may represent:
  - distance
  - cost
  - speed
  - network traffic



Subgraph

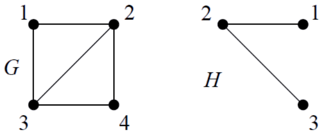
- Given graphs $G = (V, E)$ and $H = (V', E')$, H is a **subgraph** of G if and only if V' is a subset of V and E' is a subset of E.
  - If $V' = V$ then H is a **spanning subgraph** of G.



- Is G a subgraph of H?
  - We have $V = \{1, 2, 3, 4\}$, $E = \{(1, 2), (2, 4), (1, 3), (3, 4), (2, 3)\}$
  - Also $V' = \{1, 2, 3\}$, and $E' = \{(1, 2), (2, 3)\}$.

Subgraph

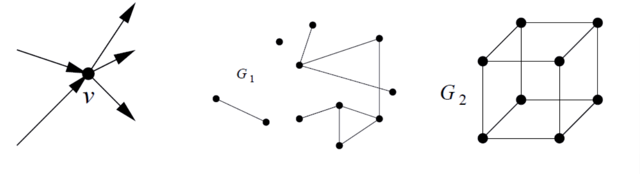
- Given graphs $G = (V, E)$ and $H = (V', E')$, H is a **subgraph** of G if and only if V' is a subset of V and E' is a subset of E.
  - If $V' = V$ then H is a **spanning subgraph** of G.



- Is G a subgraph of H?
  - We have $V = \{1, 2, 3, 4\}$, $E = \{(1, 2), (2, 4), (1, 3), (3, 4), (2, 3)\}$
  - Also $V' = \{1, 2, 3\}$, and $E' = \{(1, 2), (2, 3)\}$.
  - H is a subgraph of G but since $V \neq V'$, it is not a spanning subgraph.

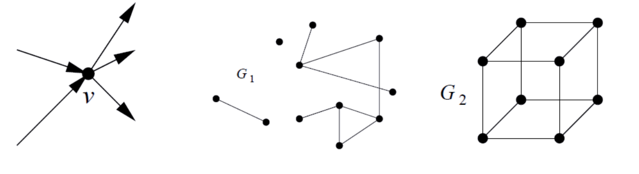
Degree

- The **degree** of a vertex v is the total number of edges incident upon v.
  - In case of a directed graph, the **in-degree** of v is the number of in-edges at v, and and the **out-degree** of v is the number of out-edges at v.



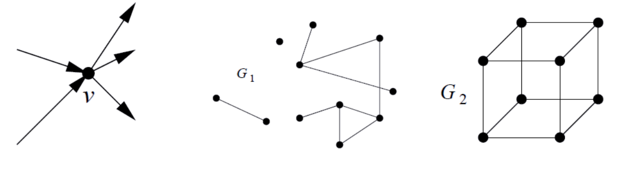
Degree

- The **degree** of a vertex v is the total number of edges incident upon v.
  - In case of a directed graph, the **in-degree** of v is the number of in-edges at v, and and the **out-degree** of v is the number of out-edges at v.
    - v has degree 5, in-degree 2, and out-degree 3.



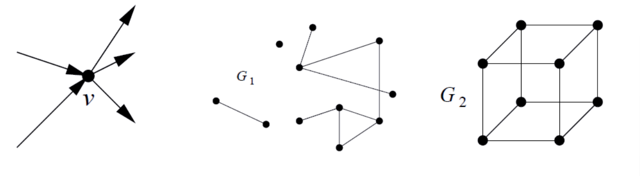
Degree

- The **degree** of a vertex v is the total number of edges incident upon v.
  - In case of a directed graph, the **in-degree** of v is the number of in-edges at v, and and the **out-degree** of v is the number of out-edges at v.
    - v has degree 5, in-degree 2, and out-degree 3.
- The **maximum degree** of a graph G, denoted $\Delta(G)$, is defined as the maximum degree amongst all vertices $v \in V$.



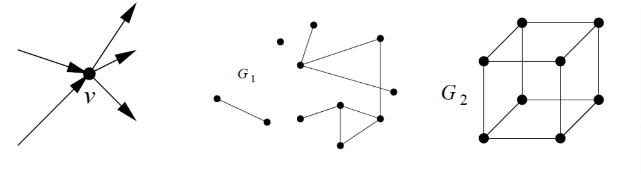
Degree

- The **degree** of a vertex v is the total number of edges incident upon v.
  - In case of a directed graph, the **in-degree** of v is the number of in-edges at v, and and the **out-degree** of v is the number of out-edges at v.
    - v has degree 5, in-degree 2, and out-degree 3.
- The **maximum degree** of a graph G, denoted $\Delta(G)$, is defined as the maximum degree amongst all vertices $v \in V$.
  - G_1 has maximum degree 3.



Degree

- The **degree** of a vertex v is the total number of edges incident upon v.
  - In case of a directed graph, the **in-degree** of v is the number of in-edges at v, and and the **out-degree** of v is the number of out-edges at v.
    - v has degree 5, in-degree 2, and out-degree 3.
- The **maximum degree** of a graph G, denoted $\Delta(G)$, is defined as the maximum degree amongst all vertices $v \in V$.
  - G_1 has maximum degree 3.
- All vertices in a **regular graph** have the same degree (e.g., G_2 is regular).



Size of a Graph

- If a graph has n vertices, what is the maximum number of edges it can have?
  - This depends on whether self-loops (edges between a vertex and itself) are permitted and wehther are directed.

Size of a Graph

- If a graph has n vertices, what is the maximum number of edges it can have?
  - This depends on whether self-loops (edges between a vertex and itself) are permitted and wehther are directed.
  - If there is no self-loop and edges are not directed, there will be $\binom{n}{2} = n(n-1)/2$ possible edges.

Size of a Graph

- If a graph has n vertices, what is the maximum number of edges it can have?
  - This depends on whether self-loops (edges between a vertex and itself) are permitted and wehther are directed.
  - If there is no self-loop and edges are not directed, there will be $\binom{n}{2} = n(n-1)/2$ possible edges.

Size of a Graph

- If a graph has n vertices, what is the maximum number of edges it can have?
  - This depends on whether self-loops (edges between a vertex and itself) are permitted and wehther are directed.
  - If there is no self-loop and edges are not directed, there will be $\binom{n}{2} = n(n-1)/2$ possible edges.

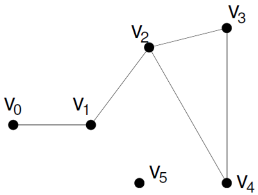
Size of a Graph

- If a graph has n vertices, what is the maximum number of edges it can have?
    - This depends on whether self-loops (edges between a vertex and itself) are permitted and wehther are directed.
    - If there is no self-loop and edges are not directed, there will be $\binom{n}{2} = n(n-1)/2$ possible edges.

	undirected	directed
self-loops not permitted	n(n-1)/2	n(n-1)
self-loops permitted	n(n+1)/2	n^2

Data Structures for Graphs

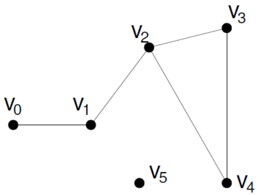
- How can we store the following graph in a data structure?



- The two common data structures for storing a graph are:

Data Structures for Graphs

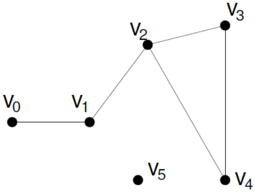
- How can we store the following graph in a data structure?



- The two common data structures for storing a graph are:
  - adjacency matrix
  - adjacency list

Adjacency Matrix

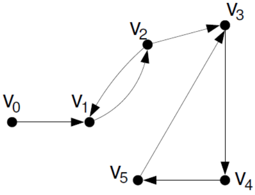
- Let $G = (V, E)$ be a graph where $V = \{v_0, v_1, \ldots, v_{n-1}\}$.
- The adjacency matrix of G is an $n \times n$ matrix A such that
  - $A[i, j] = 1$ if $(v_i, v_j) \in E$.
  - $A[i, j] = 0$ if $(v_i, v_j) \notin E$.



$$A = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

Adjacency Matrix of Digraphs

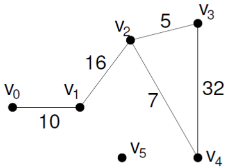
- The adjacency matrix of an undirected graph is **symmetric**.
- The adjacency matrix of a directed graph may not be asymmetric.



$$A = \begin{array}{c c} & \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} & \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \end{array}$$

Adjacency Matrix of Weighted Graphs

- We represent a weighted graph by storing the weight of edge (v_i, v_j) at $A[i, j]$.
  - We assume all weights are non-zero



$$A = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 10 & 0 & 0 & 0 & 0 \\ 10 & 0 & 16 & 0 & 0 & 0 \\ 0 & 16 & 0 & 5 & 7 & 0 \\ 0 & 0 & 5 & 0 & 32 & 0 \\ 0 & 0 & 7 & 32 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}.$$

Adjacency Matrix Summary

- Let n denote the number of vertices and m be the number of edges.
- **Storing** the matrix takes

Adjacency Matrix Summary

- Let n denote the number of vertices and m be the number of edges.
- **Storing** the matrix takes $O(n^2)$.
- We can check whether an edge (**edge-search**) between v_i and v_j exists in

Adjacency Matrix Summary

- Let n denote the number of vertices and m be the number of edges.
- **Storing** the matrix takes $O(n^2)$.
- We can check whether an edge (**edge-search**) between v_i and v_j exists in $O(1)$ time (just check the index $a[i][j]$).
  - Similar time for adding an edge (just set the value of $a[i][j]$ to 1 or another number to indicate weight).

Adjacency Matrix Summary

- Let n denote the number of vertices and m be the number of edges.
- **Storing** the matrix takes $O(n^2)$.
- We can check whether an edge (**edge-search**) between v_i and v_j exists in $O(1)$ time (just check the index $a[i][j]$).
  - Similar time for adding an edge (just set the value of $a[i][j]$ to 1 or another number to indicate weight).
- We can compute the **indegree** of a vertex v_i in time

Adjacency Matrix Summary

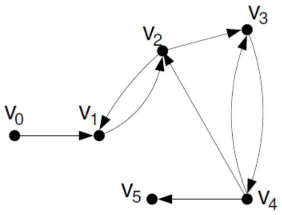
- Let n denote the number of vertices and m be the number of edges.
- **Storing** the matrix takes $O(n^2)$.
- We can check whether an edge (**edge-search**) between v_i and v_j exists in $O(1)$ time (just check the index $a[i][j]$).
  - Similar time for adding an edge (just set the value of $a[i][j]$ to 1 or another number to indicate weight).
- We can compute the **indegree** of a vertex v_i in time $O(n)$ (just scan the i'th column and count non-zero elements).
- We can compute the **outdegree** of a vertex v_i in time

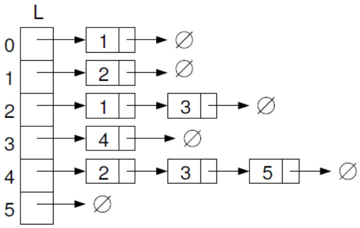
Adjacency Matrix Summary

- Let n denote the number of vertices and m be the number of edges.
- **Storing** the matrix takes $O(n^2)$.
- We can check whether an edge (**edge-search**) between v_i and v_j exists in $O(1)$ time (just check the index $a[i][j]$).
  - Similar time for adding an edge (just set the value of $a[i][j]$ to 1 or another number to indicate weight).
- We can compute the **indegree** of a vertex v_i in time $O(n)$ (just scan the i'th column and count non-zero elements).
- We can compute the **outdegree** of a vertex v_i in time $O(n)$ (just scan the i'th row and count non-zero elements).

Adjacency List

- An adjacency matrix requires $O(n^2)$space, where $n = |V|$.
  - For a sparse matrix (when m is small relative to n), a data structure that uses less space may be useful.
  - **Adjacency List:** use a linked list for each vertex.





Adjacency List Summary

- An adjacency list requires a **space** of

Adjacency List Summary

- An adjacency list requires a **space** of $O(m+n)$ space, where $n = |V|$ and $m = |E|$.
  - There is one node for each vertex (in the array) and one node for each directed edge (two nodes for undirected edges).
- Checking for an edge (v_i, v_j) takes

Adjacency List Summary

- An adjacency list requires a **space** of $O(m+n)$ space, where $n = |V|$ and $m = |E|$.
  - There is one node for each vertex (in the array) and one node for each directed edge (two nodes for undirected edges).
- Checking for an edge (v_i, v_j) takes $O(\Delta(G))$; recall that $\Delta(G)$ is the max degree and is at most $n-1$.
  - we just need to scan the list associated with one of the vertices.

Adjacency List Summary

- An adjacency list requires a **space** of $O(m+n)$ space, where $n = |V|$ and $m = |E|$.
  - There is one node for each vertex (in the array) and one node for each directed edge (two nodes for undirected edges).
- Checking for an edge (v_i, v_j) takes $O(\Delta(G))$; recall that $\Delta(G)$ is the max degree and is at most $n-1$.
  - we just need to scan the list associated with one of the vertices.
  - adding an edge takes the same time of $O(\Delta(G))$: method `addEdge(u,v)` should check whether edge (u, v) is already in the linked-list $A[u]$ to avoid inserting an edge multiple times.

Adjacency List Summary

- An adjacency list requires a **space** of $O(m+n)$ space, where $n = |V|$ and $m = |E|$.
  - There is one node for each vertex (in the array) and one node for each directed edge (two nodes for undirected edges).
- Checking for an edge (v_i, v_j) takes $O(\Delta(G))$; recall that $\Delta(G)$ is the max degree and is at most $n-1$.
  - we just need to scan the list associated with one of the vertices.
  - adding an edge takes the same time of $O(\Delta(G))$: method `addEdge(u,v)` should check whether edge (u, v) is already in the linked-list $A[u]$ to avoid inserting an edge multiple times.

Adjacency List Summary

- An adjacency list requires a **space** of $O(m+n)$ space, where $n = |V|$ and $m = |E|$.
  - There is one node for each vertex (in the array) and one node for each directed edge (two nodes for undirected edges).
- Checking for an edge (v_i, v_j) takes $O(\Delta(G))$; recall that $\Delta(G)$ is the max degree and is at most $n-1$.
  - we just need to scan the list associated with one of the vertices.
  - adding an edge takes the same time of $O(\Delta(G))$: method `addEdge(u,v)` should check whether edge (u, v) is already in the linked-list $A[u]$ to avoid inserting an edge multiple times.
- Degree queries:
  - Computing the out-degree of v_i takes

Adjacency List Summary

- An adjacency list requires a **space** of $O(m+n)$ space, where $n = |V|$ and $m = |E|$.
  - There is one node for each vertex (in the array) and one node for each directed edge (two nodes for undirected edges).
- Checking for an edge (v_i, v_j) takes $O(\Delta(G))$; recall that $\Delta(G)$ is the max degree and is at most $n-1$.
  - we just need to scan the list associated with one of the vertices.
  - adding an edge takes the same time of $O(\Delta(G))$: method `addEdge(u,v)` should check whether edge (u,v) is already in the linked-list $A[u]$ to avoid inserting an edge multiple times.
- Degree queries:
  - Computing the out-degree of v_i takes $O(\Delta(G))$; just scan the list of v_i and report its length.

Adjacency List Summary

- An adjacency list requires a **space** of $O(m+n)$ space, where $n = |V|$ and $m = |E|$.
  - There is one node for each vertex (in the array) and one node for each directed edge (two nodes for undirected edges).
- Checking for an edge (v_i, v_j) takes $O(\Delta(G))$; recall that $\Delta(G)$ is the max degree and is at most $n-1$.
  - we just need to scan the list associated with one of the vertices.
  - adding an edge takes the same time of $O(\Delta(G))$: method `addEdge(u,v)` should check whether edge (u, v) is already in the linked-list $A[u]$ to avoid inserting an edge multiple times.
- Degree queries:
  - Computing the out-degree of v_i takes $O(\Delta(G))$; just scan the list of v_i and report its length.
  - Computing the in-degree of v_i takes

Adjacency List Summary

- An adjacency list requires a **space** of $O(m+n)$ space, where $n = |V|$ and $m = |E|$.
  - There is one node for each vertex (in the array) and one node for each directed edge (two nodes for undirected edges).
- Checking for an edge (v_i, v_j) takes $O(\Delta(G))$; recall that $\Delta(G)$ is the max degree and is at most $n-1$.
  - we just need to scan the list associated with one of the vertices.
  - adding an edge takes the same time of $O(\Delta(G))$: method `addEdge(u,v)` should check whether edge (u, v) is already in the linked-list $A[u]$ to avoid inserting an edge multiple times.
- Degree queries:
  - Computing the out-degree of v_i takes $O(\Delta(G))$; just scan the list of v_i and report its length.
  - Computing the in-degree of v_i takes $O(m+n)$; we need to go through all nodes.

Adjacency Matrix vs. Adjacency List

- Recall that n denotes the number of vertices and m denotes the number of edges.
- In general, we use adjacency matrices for dense graphs (with many edges) and adjacency lists for sparse graphs (with relatively a few number of edges).

	adjacency matrix	adjacency list
space	$\Theta(n^2)$	$\Theta(n+m)$
edge search	$\Theta(1)$	$\Theta(\Delta(G))$
compute out-degree of v	$\Theta(n)$	$\Theta(\Delta(G))$
compute in-degree of v	$\Theta(n)$	$\Theta(n+m)$

Adjacency Matrix or Adjacency List?

How do you decide which representation to use?
You have to look at what your application needs:

- If you need to be able to quickly tell if there is an edge between vertices i and j, then use an adjacency matrix.
- If you need to perform a matrix multiplication, then use an adjacency matrix.
- If you need fast access to all edges out of a vertex, use the adjacency list.
- If space is an issue (a huge number of vertices), then an adjacency list is a good idea.

Traversing in a Graph

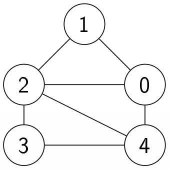
- **Graph traversal:** The most common graph operation is to visit all the vertices in a systematic way, using the edges of the graph.
- A graph traversal starts at a vertex v and visits all the vertices u such that a path exists from v to u.
- **Two types of traversals:**
  - Depth-first traversal (or depth-first search)
  - Breadth-first traversal (or breadth-first search)

Traversing in a Graph

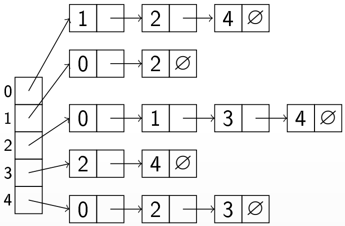
- During any traversal, we will want to **find all vertices that are adjacent to the current vertex**.
- Therefore, we should use an **adjacency list** to store the graph.
- You can do a traversal if you are using an adjacency matrix; it will simply be less efficient (slower).

Traversing in a Graph

Since we are storing the graph as an adjacency list: we will assume that, for any vertex i, the vertices adjacent to vertex i are kept in an **ordered** linked list.



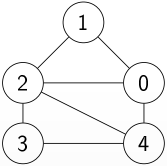
Adjacency List:



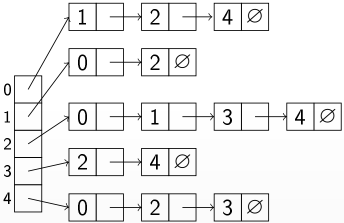
Traversals and Graph Representation

Consequence of using ordered linked lists in the adjacency list: We always examine the adjacent vertices in sorted order.

- For example, if we are at vertex 2 in this graph, we will examine adjacent vertices in the following order: 0, 1, 3, then 4.

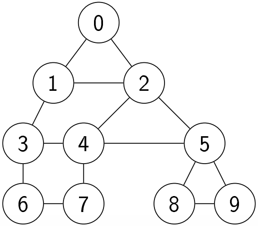


Adjacency List:



Depth-first Traversal

A depth first traversal searches **all the way down a path before backing up** to explore alternatives — it is **a recursive, stack-based traversal.**

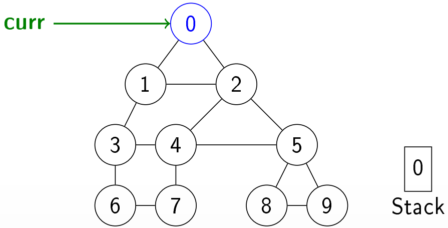


We will traverse the above graph starting at 0, with all vertices currently unvisited.

Depth-first Traversal: Example

To depth-first traverse at vertex 0:

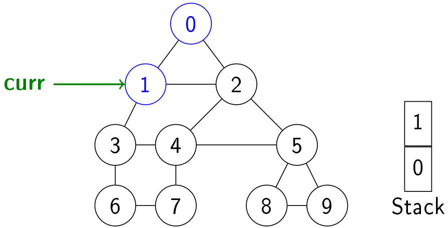
- Mark the current vertex (0) as visited, and then
- Recursively depth-first traverse each of the adjacent unvisited vertices.



Remember that we examine the adjacent vertices in sorted order: we first look at vertex 1, then at vertex 2. with 1.

Depth-first Traversal: Example

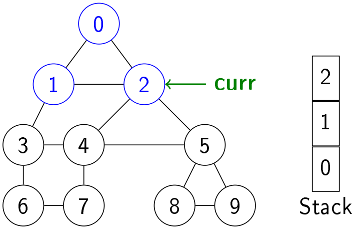
Now mark the current vertex (1) as visited, and then recursively depth-first traverse each of the adjacent unvisited vertices.



Adjacent vertex 2 is unvisited, so we next recursively depth-first traverse vertex 2.

Depth-first Traversal: Example

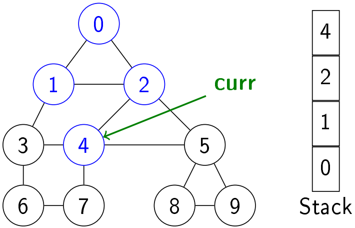
Now mark the current vertex (2) as visited, and then recursively depth-first traverse each of the adjacent unvisited vertices.



Adjacent vertex 4 is unvisited, so we next recursively depth-first traverse vertex 4.

Depth-first Traversal: Example

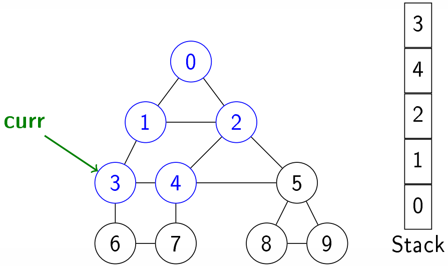
Now mark the current vertex (4) as visited, and then recursively depth-first traverse each of the adjacent unvisited vertices.



Adjacent vertex 3 is unvisited, so we next recursively depth-first traverse vertex 3.

Depth-first Traversal: Example

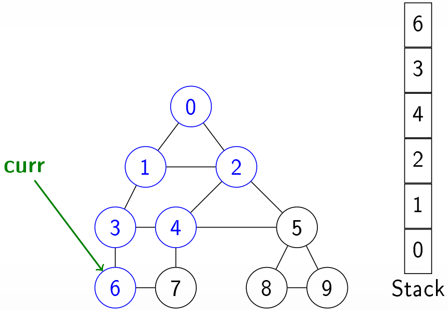
Now mark the current vertex (3) as visited, and then recursively depth-first traverse each of the adjacent unvisited vertices.



Adjacent vertex 6 is unvisited, so we next recursively depth-first traverse vertex 6.

Depth-first Traversal: Example

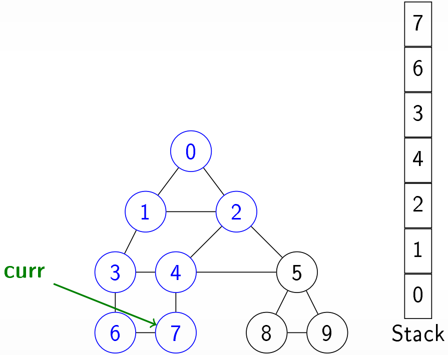
Now mark the current vertex (6) as visited, and then recursively depth-first traverse each of the adjacent unvisited vertices.



Adjacent vertex 7 is unvisited, so we next recursively depth-first traverse vertex 7.

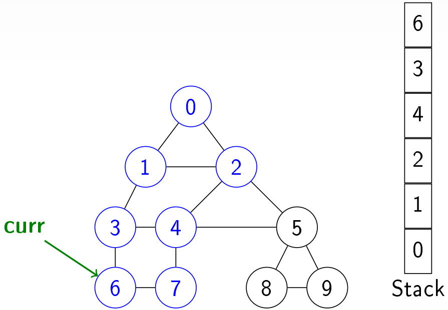
Depth-first Traversal: Example

Mark 7 as visited, and recursively depth-first traverse the adjacent unvisited vertices.



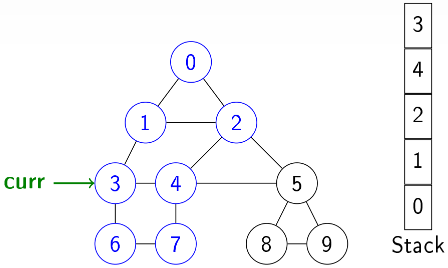
No adjacent vertices are unvisited, so pop the stack to return to a previous vertex and look for unvisited adjacent vertices there.

Depth-first Traversal: Example



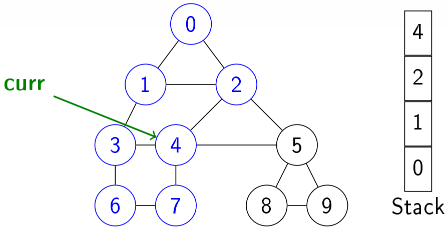
No vertices adjacent to 6 are unvisited, so pop the stack again.

Depth-first Traversal: Example



No vertices adjacent to 3 are unvisited, so pop the stack again.

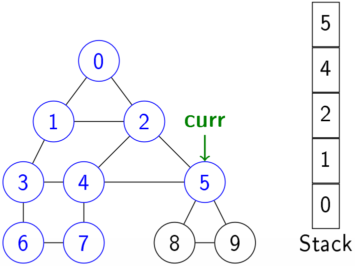
Depth-first Traversal: Example



Vertex 5 is adjacent to 4 and is unvisited, so depth-first traverse 5.

Depth-first Traversal: Example

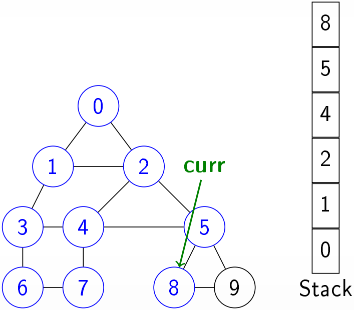
Mark 5 as visited, and recursively depth-first search adjacent unvisited vertices.



Vertex 8 is adjacent to 5 and is unvisited, so depth-first traverse 8.

Depth-first Traversal: Example

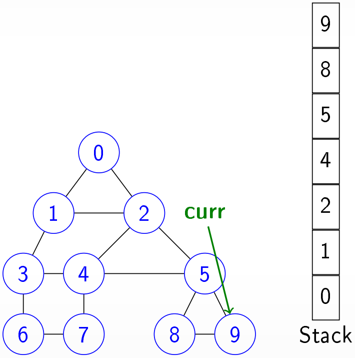
Mark 8 as visited, and recursively depth-first search adjacent unvisited vertices.



Vertex 9 is adjacent to 8 and is unvisited, so depth-first traverse 9.

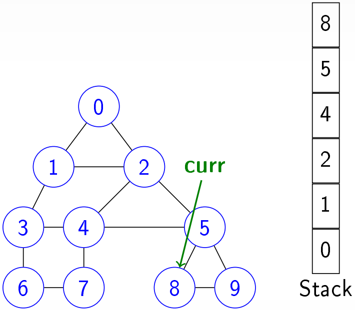
Depth-first Traversal: Example

Mark 9 as visited, and recursively depth-first search adjacent unvisited vertices.



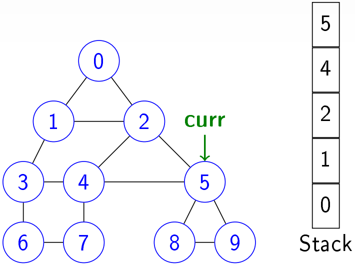
No adjacent vertex is unvisited, so pop the stack.

Depth-first Traversal: Example



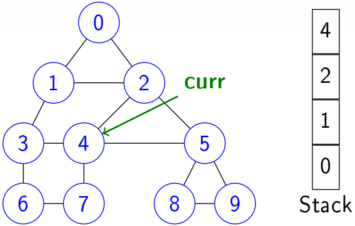
No adjacent vertex is unvisited, so pop the stack.

Depth-first Traversal: Example



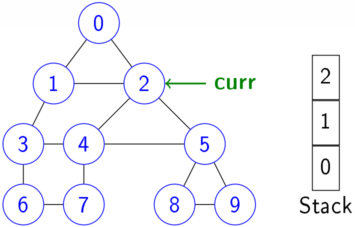
No adjacent vertex is unvisited, so pop the stack.

Depth-first Traversal: Example



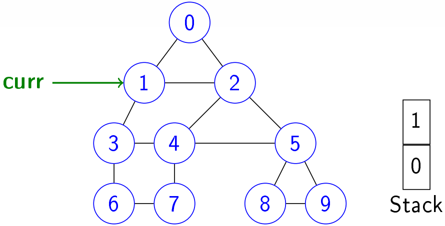
No adjacent vertex is unvisited, so pop the stack.

Depth-first Traversal: Example



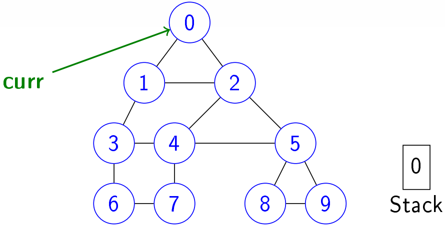
No adjacent vertex is unvisited, so pop the stack.

Depth-first Traversal: Example



No adjacent vertex is unvisited, so pop the stack.

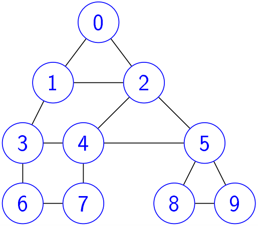
Depth-first Traversal: Example



No adjacent vertex is unvisited, so pop the stack.

Depth-first Traversal: Example

Now the stack is empty, so we have traversed the whole graph.



Stack

Depth-first Traversal and Paths

- The stack holds the path we took from the starting vertex to the current vertex.
- To find a path from some vertex u to some other vertex v:
  You could perform a depth-first traversal starting at u and simply output the stack (from bottom to top) when you find v.
- The path you find will not necessarily be the shortest path from u to v (you will, however, find a path if one exists).

Printing in a Depth-First Traversal

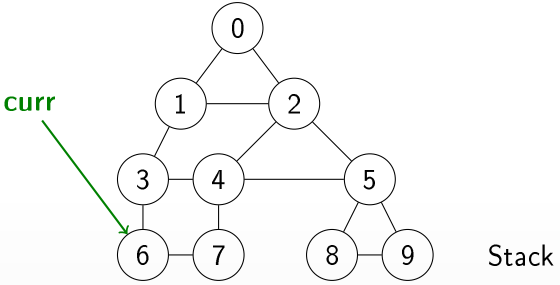
We "visit" a vertex when we mark it as visited.

- In our example, we did nothing when we visited a vertex.
- If we print out the contents of the vertex when we visit it, then the output would be

$$0, 1, 2, 4, 3, 6, 7, 5, 8, 9$$

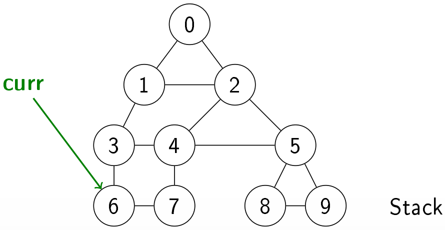
Depth-first Traversal: Example

What would the output be if we performed a depth-first traversal starting at vertex 6? (Assume that we always examine adjacent vertices in sorted order because we are using an adjacency list.)



Depth-first Traversal: Example

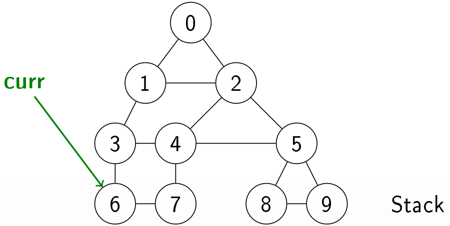
What would the output be if we performed a depth-first traversal starting at vertex 6? (Assume that we always examine adjacent vertices in sorted order.)



Answer: 6, 3, 1, 0, 2, 4, 5, 8, 9, 7

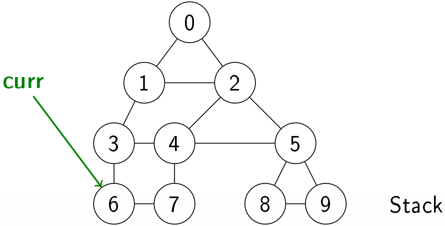
Depth-first Traversal: Example

What path would we find to vertex 7 if we performed a depth-first traversal starting at vertex 6? (Assume that we always examine adjacent vertices in sorted order.)



Depth-first Traversal: Example

What path would we find to vertex 7 if we performed a depth-first traversal starting at vertex 6? (Assume that we always examine adjacent vertices in sorted order.)



Answer: 6, 3, 1, 0, 2, 4, 7

Depth-first Traversal Pseudocode

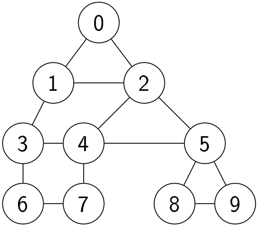
```
depthFirstTraveral (vertex curr)
    mark vertex curr as visited
    visit curr (e.g., print)
    for each vertex v adjacent to curr
        if v is unvisited
            depthFirstTraveral(v)
```

Depth-First Traversal Applications

- You are doing a depth-first traversal if you traverse a maze.
  - each intersection is a vertex, and you go to a neighbor that is not visited before.
- Detecting whether a graph is a tree!
  - A graph is a tree if there is no cylce in the graph!

Breadth-first Traversal

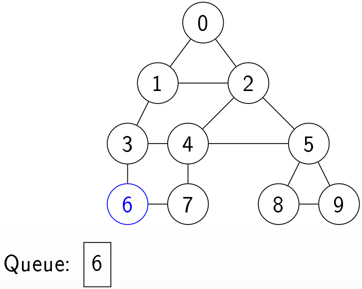
A breadth-first traversal **visits all nearby vertices** first before moving farther away. It is **a queue-based, iterative traversal.**



We will do a breadth-first traversal of the above graph starting at vertex 6.

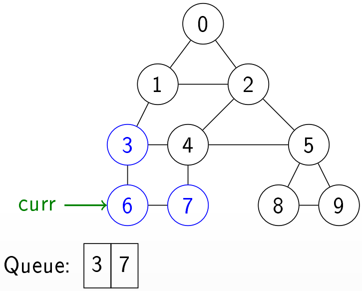
Breadth-first Traversal Example

To begin the breadth-first traversal, visit the starting vertex and put it on the queue.



Breadth-first Traversal Example

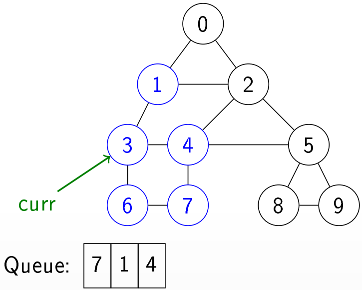
- Remove a vertex `curr` from the queue.
- Visit all the unvisited vertices adjacent to `curr`, putting each one on the queue.



3 and 7 were the unvisited vertices we found adjacent to 6.

Breadth-first Traversal Example

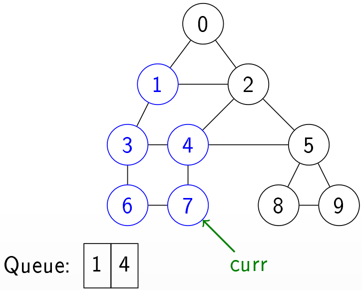
- Remove a vertex `curr` from the queue.
- Visit all the unvisited vertices adjacent to `curr`, putting each one on the queue.



1 and 4 were the unvisited vertices we found adjacent to 3.

Breadth-first Traversal Example

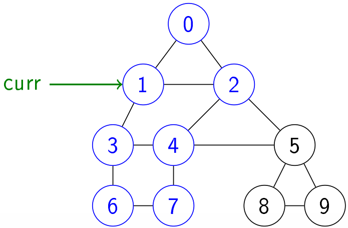
- Remove a vertex `curr` from the queue.
- Visit all the unvisited vertices adjacent to `curr`, putting each one on the queue.



There were no unvisited vertices found adjacent to 7.

Breadth-first Traversal Example

- Remove a vertex `curr` from the queue.
- Visit all the unvisited vertices adjacent to `curr`, putting each one on the queue.



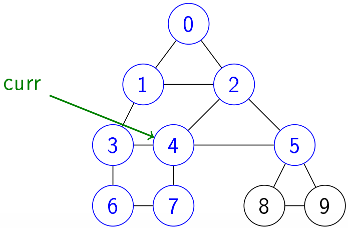
Queue:

4	0	2

0 and 2 were unvisited vertices adjacent to 1.

Breadth-first Traversal Example

- Remove a vertex `curr` from the queue.
- Visit all the unvisited vertices adjacent to `curr`, putting each one on the queue.



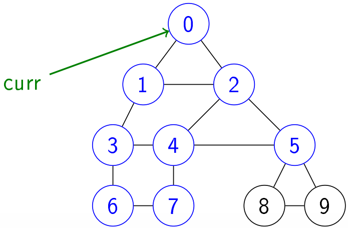
Queue:

0	2	5

5 was an unvisited vertex adjacent to 4.

Breadth-first Traversal Example

- Remove a vertex `curr` from the queue.
- Visit all the unvisited vertices adjacent to `curr`, putting each one on the queue.



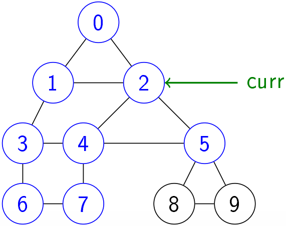
Queue:

2	5

No unvisited vertices were found adjacent to 0.

Breadth-first Traversal Example

- Remove a vertex `curr` from the queue.
- Visit all the unvisited vertices adjacent to `curr`, putting each one on the queue.

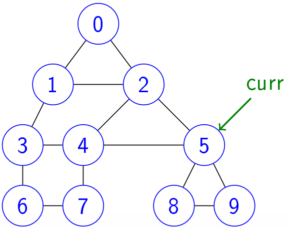


Queue: | 5 |

No unvisited vertices were found adjacent to 2.

Breadth-first Traversal Example

- Remove a vertex curr from the queue.
- Visit all the unvisited vertices adjacent to curr, putting each one on the queue.



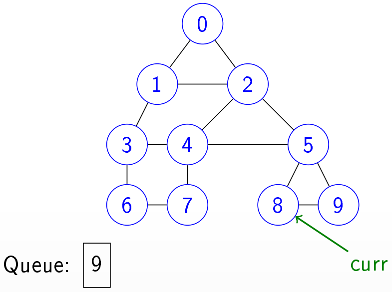
Queue:

8	9

8 and 9 were unvisited vertices adjacent to 5.

Breadth-first Traversal Example

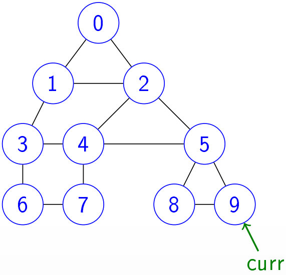
- Remove a vertex `curr` from the queue.
- Visit all the unvisited vertices adjacent to `curr`, putting each one on the queue.



No unvisited vertices were found adjacent to 8.

Breadth-first Traversal Example

- Remove a vertex `curr` from the queue.
- Visit all the unvisited vertices adjacent to `curr`, putting each one on the queue.



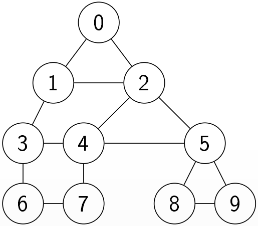
Queue:

No unvisited vertices were found adjacent to 9. Since the queue is now empty, the traversal is finished.

Printing in a Breadth-first Traversal

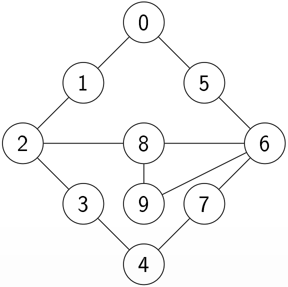
- In our example, we did nothing at each vertex when we visited it.
- If we print out the contents of the vertex when we visit it, then the output would be

$$6, 3, 7, 1, 4, 0, 2, 5, 8, 9$$



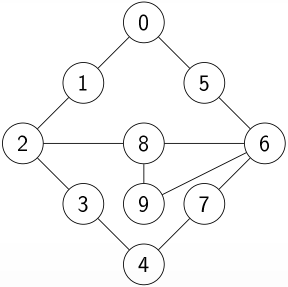
Breadth-first Traversal Example

What would the output be if we performed a breadth-first traversal of the following graph, beginning at vertex 0:



Breadth-first Traversal Example

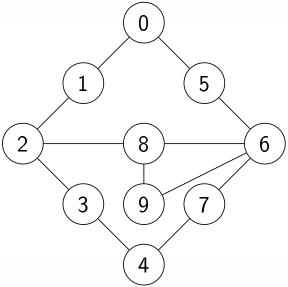
What would the output be if we performed a breadth-first traversal of the following graph, beginning at vertex 0:



Answer: 0, 1, 5, 2, 6, 3, 8, 7, 9, 4

Breadth-first Traversal Example

What would the output be if we performed a **depth**-first traversal of the following graph, beginning at vertex 0:



Breadth-first answer: 0, 1, 5, 2, 6, 3, 8, 7, 9, 4
Depth-first answer: 0, 1, 2, 3, 4, 7, 6, 5, 8, 9

Breadth-first Traversal Pseudocode

```
breadthFirstTraversal (vertex start)
    Q ← an empty queue of vertices
    visit start (e.g., print) and mark it as visited
    Q.enqueue(start)
    while Q is not empty
        curr ← Q.dequeue()
        for each unvisited vertex v adjacent to curr
            visit v and mark it as visited;
            Q.enqueue(v);
```

Breadth-first Traversal and Paths

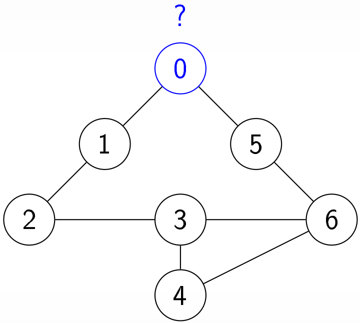
The vertices we passed through to get to a vertex v are no longer on the queue when v is visited and placed on the queue.

To reconstruct the path to v:

- When you mark a vertex w as visited, also record at w what vertex you came from to get to w (i.e., which vertex is w adjacent to when you visit w).
- Therefore, each vertex needs to have not only a "visited" bit, but also a "previous vertex" pointer.
- When the traversal is finished, retrieve the path from the starting vertex to vertex v: We get the path backwards by starting at v and following previous pointers back to the starting vertex.

Breadth-First Traversal Paths

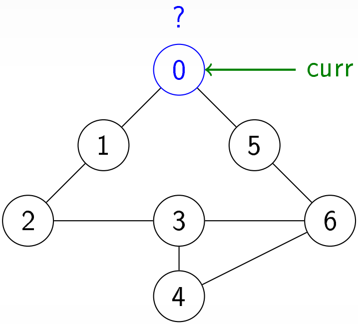
For example, beginning at vertex 0:



Queue: $\boxed{0}$

Breadth-First Traversal Paths

Vertex 0 is first out of the queue:

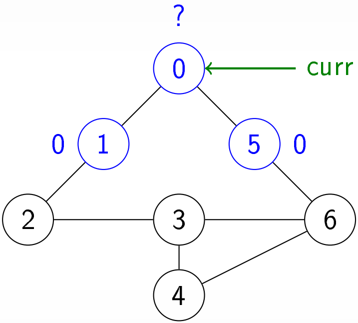


Queue:

We will visit 1 and 5 next, marking “0” as their previous vertex.

Breadth-First Traversal Paths

We're currently at vertex 0:

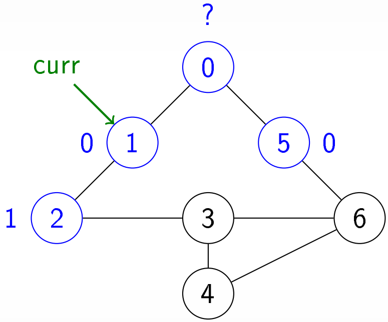


Queue:

1	5

Breadth-First Traversal Paths

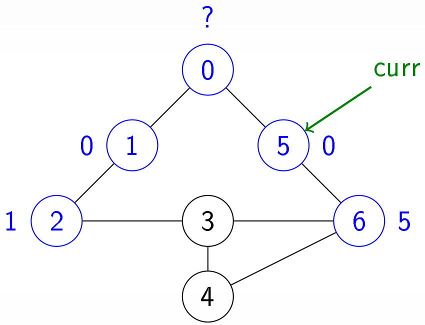
At vertex 1, we visit vertex 2:



Queue: | 5 | 2 |

Breadth-First Traversal Paths

At vertex 5, we visit vertex 6:

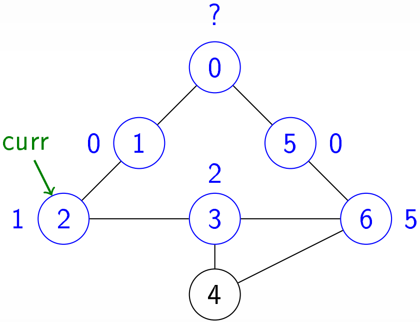


Queue:

2	6

Breadth-First Traversal Paths

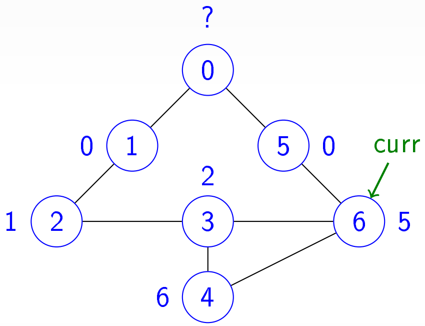
At vertex 2, we visit vertex 3:



Queue: | 6 | 3 |

Breadth-First Traversal Paths

At vertex 6, we visit vertex 4:

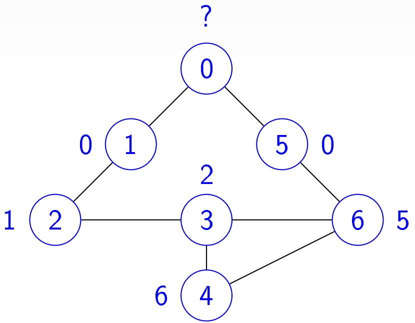


Queue: | 3 | 4 |

When we remove 3 and 4 from the queue, there is nothing left to visit, so we will skip those steps.

Breadth-First Traversal Paths

Suppose we want to find the path taken to 4 from 0:

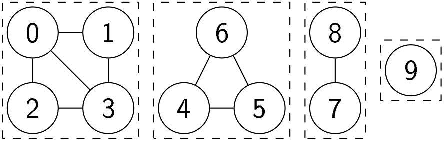


We follow "previous vertex" pointers starting from vertex 4, which gives us the path from 0 to 4 backwards:

$$4 \leftarrow 6 \leftarrow 5 \leftarrow 0$$

Connected Component

A disconnected graph can be divided into **connected components** two vertices i and j are in the same connected component if there is a path from i to j.



Example: The above disconnected graph has 4 connected components (inside dashed rectangles).

Traversals and Disconnected Graphs

- A traversal starts at a vertex v and visits all the vertices that can be reached by paths from v.
- If the graph is disconnected, then a traversal will visit all the vertices in the same component as v.
- To visit the whole graph:

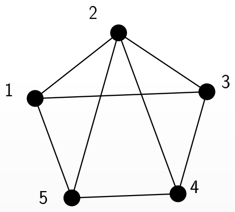
```
1.    loop
2.        find a vertex v that has not been visited yet
3.        perform a traversal from v
4.    until all vertices have been visited
```

Summary: Traversals and Paths

- **Depth-first search finds a path** from the start vertex to another vertex, not necessarily the shortest path (the path with the fewest edges).
- **Breadth-first search finds the shortest path.**

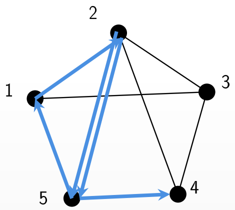
Walks, Paths, Circuits, and Cycles

- A **walk** from vertex v to vertex w is a finite sequence of adjacent vertices of G.
  - 2,5,1,2,5,4 is a walk.



Walks, Paths, Circuits, and Cycles

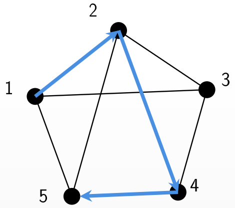
- A **walk** from vertex v to vertex w is a finite sequence of adjacent vertices of G.
  - 2,5,1,2,5,4 is a walk.



Walks, Paths, Circuits, and Cycles

- A **walk** from vertex v to vertex w is a finite sequence of adjacent vertices of G.
- A **path** from v to w is a walk from v to w that does not contain any repeated edges.

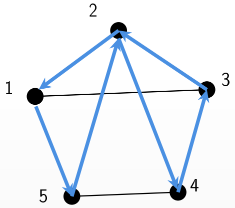
- 2,5,1,2,5,4 is a walk.
- 1,2,4,5 is a path (and also a walk).



Walks, Paths, Circuits, and Cycles

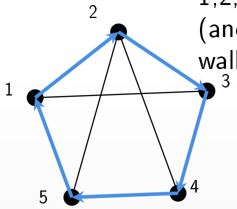
- A **walk** from vertex v to vertex w is a finite sequence of adjacent vertices of G.
- A **path** from v to w is a walk from v to w that does not contain any repeated edges.
- A **circuit** is a walk that begins and ends on the same vertex.

- 2,5,1,2,5,4 is a walk.
- 1,2,4,5 is a path (and also a walk).
- 1,5,2,4,3,2,1 is a circuit (also a path and a walk).



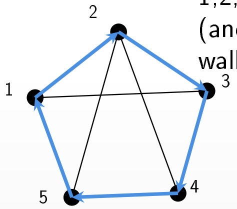
Walks, Paths, Circuits, and Cycles

- A **walk** from vertex v to vertex w is a finite sequence of adjacent vertices of G.
  - 2,5,1,2,5,4 is a walk.
- A **path** from v to w is a walk from v to w that does not contain any repeated edges.
  - 1,2,4,5 is a path (and also a walk).
- A **circuit** is a walk that begins and ends on the same vertex.
  - 1,5,2,4,3,2,1 is a circuit (also a path and a walk).
- A **cycle** is a circuit that does not contain any repeated vertices.
  - 1,2,3,4,5,1 is a cycle (and also a circuit, a path, and a walk).



Walks, Paths, Circuits, and Cycles

- A **walk** from vertex v to vertex w is a finite sequence of adjacent vertices of G.
- A **path** from v to w is a walk from v to w that does not contain any repeated edges.
- A **circuit** is a walk that begins and ends on the same vertex.
- A **cycle** is a circuit that does not contain any repeated vertices.
- A **k-cycle** is a cycle of length k.



- 2,5,1,2,5,4 is a walk.
- 1,2,4,5 is a path (and also a walk).
- 1,5,2,4,3,2,1 is a circuit (also a path and a walk).
- 1,2,3,4,5,1 is a cycle (and also a circuit, a path, and a walk).

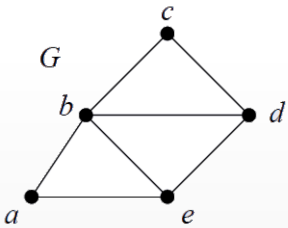
More Terminology

- The **length** of a walk, path, circuit, or cycle is the number of edges in the sequence.
- The **distance** between vertices v and w is the length of the shortest path from v to w.
- The **diameter** of graph G is the maximum distance between any two vertices v, w in G.

More Terminology

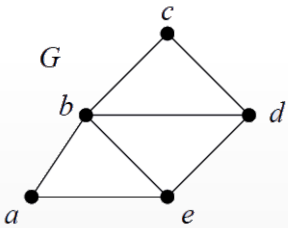
- The **length** of a walk, path, circuit, or cycle is the number of edges in the sequence.
- The **distance** between vertices v and w is the length of the shortest path from v to w.
- The **diameter** of graph G is the maximum distance between any two vertices v, w in G.

  - b and c have distance



More Terminology

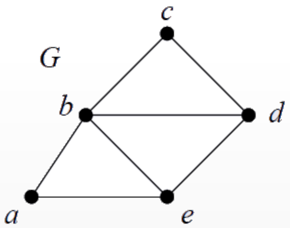
- The **length** of a walk, path, circuit, or cycle is the number of edges in the sequence.
- The **distance** between vertices v and w is the length of the shortest path from v to w.
- The **diameter** of graph G is the maximum distance between any two vertices v, w in G.



- b and c have distance 1

More Terminology

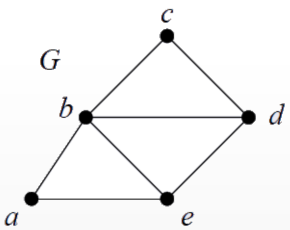
- The **length** of a walk, path, circuit, or cycle is the number of edges in the sequence.
- The **distance** between vertices v and w is the length of the shortest path from v to w.
- The **diameter** of graph G is the maximum distance between any two vertices v, w in G.



- b and c have distance 1
- a and d have distance 2

More Terminology

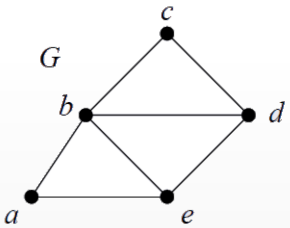
- The **length** of a walk, path, circuit, or cycle is the number of edges in the sequence.
- The **distance** between vertices v and w is the length of the shortest path from v to w.
- The **diameter** of graph G is the maximum distance between any two vertices v, w in G.



- b and c have distance 1
- a and d have distance 2

More Terminology

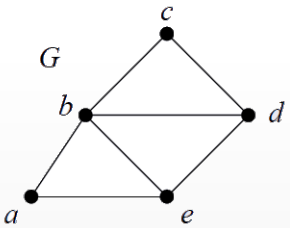
- The **length** of a walk, path, circuit, or cycle is the number of edges in the sequence.
- The **distance** between vertices v and w is the length of the shortest path from v to w.
- The **diameter** of graph G is the maximum distance between any two vertices v, w in G.



- b and c have distance 1
- a and d have distance 2
- G has diameter 2

More Terminology

- The **length** of a walk, path, circuit, or cycle is the number of edges in the sequence.
- The **distance** between vertices v and w is the length of the shortest path from v to w.
- The **diameter** of graph G is the maximum distance between any two vertices v, w in G.
  - **The small-world phenomenon:** we are all linked by short chains of acquaintances → social networks like Facebook have small diameter.



- b and c have distance 1
- a and d have distance 2
- G has diameter 2

Connected Graphs

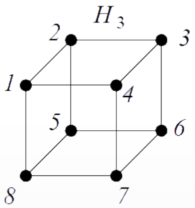
- Two vertices v and w are **connected** iff there is a path from v to w.
- Graph G is connected iff any two vertices, v, w in G are connected.

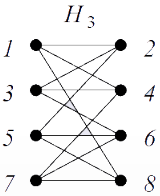
Connected Graphs

- Two vertices v and w are **connected** iff there is a path from v to w.
- Graph G is connected iff any two vertices, v, w in G are connected.
- Here G_1 is connected and G_2 is not connected.

Bipartite Graphs

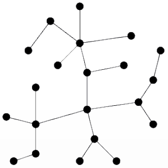
- A graph $G = (V, E)$ is **bipartite** if there exists a partition of its vertices, $V = V_1 \cup V_2$, such that:
  - $V_1 \cap V_2 = \emptyset$, and
  - every edge $(v_1, v_2) \in E$ has one endpoint in each partition: $v_1 \in V_1$ and $v_2 \in V_2$ or $v_1 \in V_2$ and $v_2 \in V_1$.



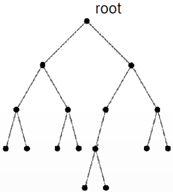


Trees

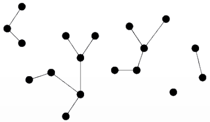
- An undirected graph T is a **tree** if T is connected and T does not contain any cycles.
    - In a **rooted tree**, one vertex is distinguished from the others and called the root.
- An undirected graph F is a **forest** if F does not contain any cycles. F is a set of trees.



a tree



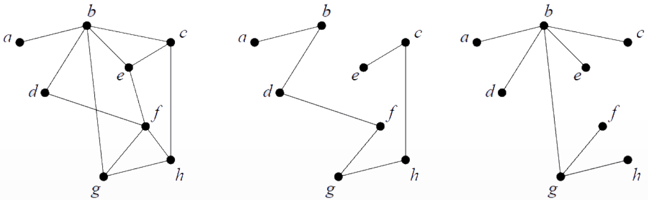
a rooted tree



a forest

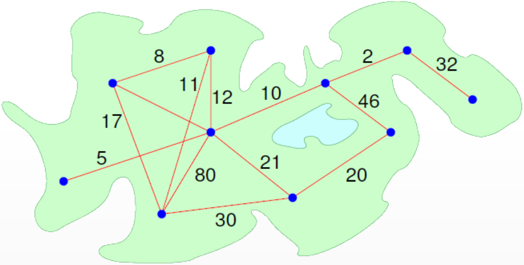
Spanning Tree

- A **spanning tree** for a graph G is a spanning subgraph of G that is a tree.
  - Every connected graph has a spanning tree.
  - Any spanning tree for a graph $G = (V, E)$ has $|V|$ vertices.
  - Any spanning tree for a graph $G = (V, E)$ has $|V| - 1$ edges.



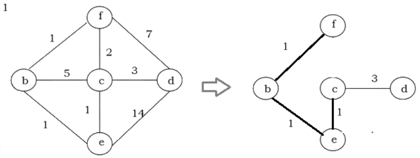
Spanning Trees Application

- Your employer has a contract to provide high-speed internet to an island.
- Each client must be connected to the network while minimizing the total cost of building the network.
- Your are provided cost estimates for various possible links in the network.



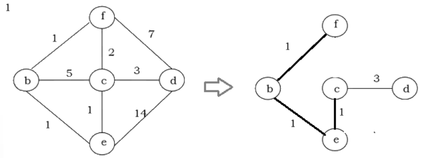
Minimum Spanning Tree

- A **minimum spanning tree** of a weighted graph G is a spanning tree of G that has the least possible total weight compared to all other spanning trees of G.



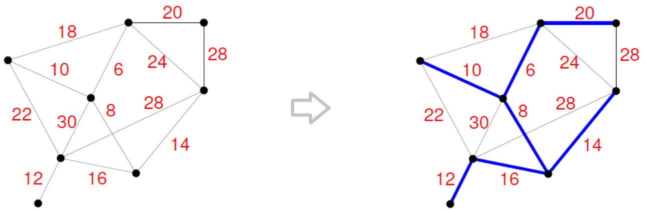
Minimum Spanning Tree

- A **minimum spanning tree** of a weighted graph G is a spanning tree of G that has the least possible total weight compared to all other spanning trees of G.
  - If two or more edges have equal weight in a graph G, then G may have more than one unique minimum spanning tree.



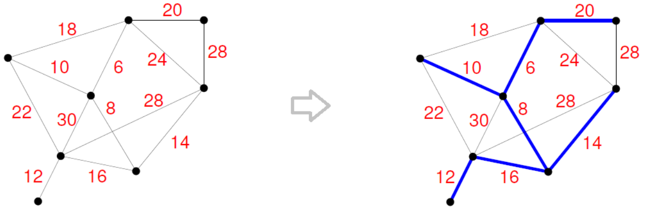
More Minimum Spanning Tree Example

- It is not always easy to derive a minimum spanning tree 'with eyes'.



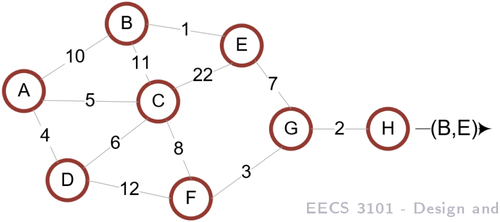
More Minimum Spanning Tree Example

- It is not always easy to derive a minimum spanning tree 'with eyes'.
- Two efficient algorithms for finding a minimum spanning tree:
  - **Kruskal's algorithm**
  - **Prim's algorithm**



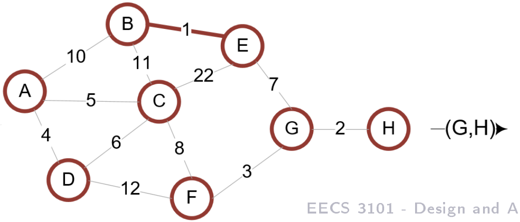
Kruskal's MST algorithm

- Initialize T to be Φ.
- Sort edges in the non-decreasing of their weights and process them one by one.
  $(B,E),(G,H),(G,F),(A,D),(A,C),(C,D),(E,G),(C,F),(A,B),(B,C),(D,F)$
- If an edge e does not form a cycle in MST, add it to MST.



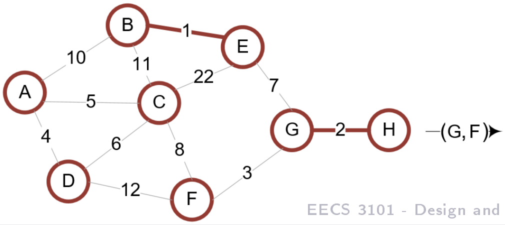
Kruskal's MST algorithm

- Initialize T to be Φ.
- Sort edges in the non-decreasing of their weights and process them one by one.
  $(B, E), (G, H), (G, F), (A, D), (A, C), (C, D), (E, G), (C, F), (A, B), (B, C), (D, F)$
- If an edge e does not form a cycle in MST, add it to MST.



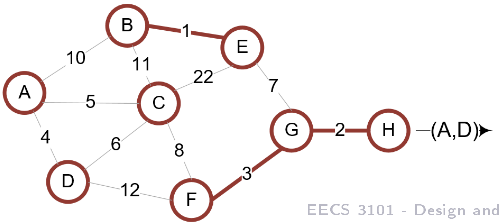
Kruskal's MST algorithm

- Initialize T to be Φ.
- Sort edges in the non-decreasing of their weights and process them one by one.
  $(B,E),(G,H),(G,F),(A,D),(A,C),(C,D),(E,G),(C,F),(A,B),(B,C),(D,F)$
- If an edge e does not form a cycle in MST, add it to MST.
  - Maintain MST's connected component as disjoint sets of vertices
  - e does not form a cycle iff its endpoints are in different components



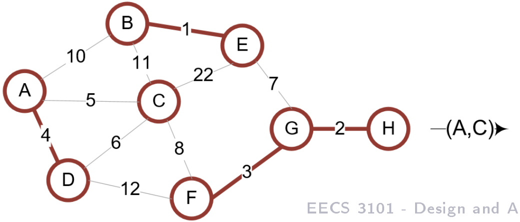
Kruskal's MST algorithm

- Initialize T to be Φ.
- Sort edges in the non-decreasing of their weights and process them one by one.
  $(B, E), (G, H), (G, F), (A, D), (A, C), (C, D), (E, G), (C, F), (A, B), (B, C), (D, F)$
- If an edge e does not form a cycle in MST, add it to MST.
  - Maintain MST's connected component as disjoint sets of vertices
  - e does not form a cycle iff its endpoints are in different components



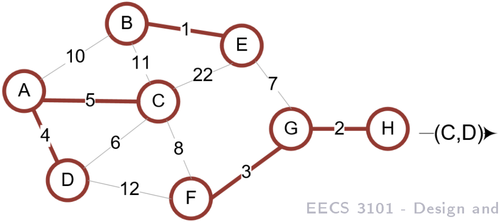
Kruskal's MST algorithm

- Initialize T to be Φ.
- Sort edges in the non-decreasing of their weights and process them one by one.
  $(B, E), (G, H), (G, F), (A, D), (A, C), (C, D), (E, G), (C, F), (A, B), (B, C), (D, F)$
- If an edge e does not form a cycle in MST, add it to MST.
  - Maintain MST's connected component as disjoint sets of vertices
  - e does not form a cycle iff its endpoints are in different components



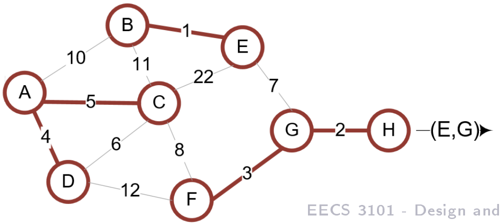
Kruskal's MST algorithm

- Initialize T to be Φ.
- Sort edges in the non-decreasing of their weights and process them one by one.
  $(B, E), (G, H), (G, F), (A, D), (A, C), (C, D), (E, G), (C, F), (A, B), (B, C), (D, F)$
- If an edge e does not form a cycle in MST, add it to MST.
  - Maintain MST's connected component as disjoint sets of vertices
  - e does not form a cycle iff its endpoints are in different components



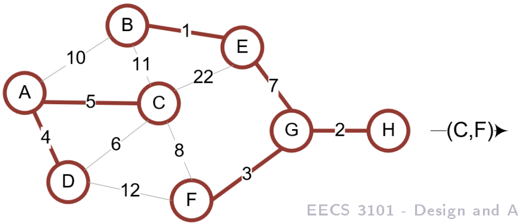
Kruskal's MST algorithm

- Initialize T to be Φ.
- Sort edges in the non-decreasing of their weights and process them one by one.
  $(B, E), (G, H), (G, F), (A, D), (A, C), (C, D), (E, G), (C, F), (A, B), (B, C), (D, F)$
- If an edge e does not form a cycle in MST, add it to MST.
  - Maintain MST's connected component as disjoint sets of vertices
  - e does not form a cycle iff its endpoints are in different components



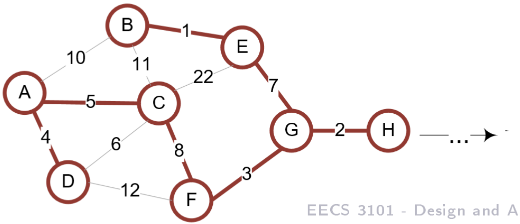
Kruskal's MST algorithm

- Initialize T to be Φ.
- Sort edges in the non-decreasing of their weights and process them one by one.
  $(B, E), (G, H), (G, F), (A, D), (A, C), (C, D), (E, G), (C, F), (A, B), (B, C), (D, F)$
- If an edge e does not form a cycle in MST, add it to MST.
  - Maintain MST's connected component as disjoint sets of vertices
  - e does not form a cycle iff its endpoints are in different components



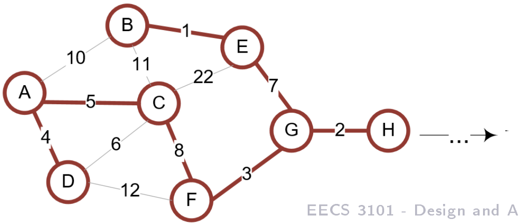
Kruskal's MST algorithm

- Initialize T to be Φ.
- Sort edges in the non-decreasing of their weights and process them one by one.
  $(B,E),(G,H),(G,F),(A,D),(A,C),(C,D),(E,G),(C,F),(A,B),(B,C),(D,F)$
- If an edge e does not form a cycle in MST, add it to MST.
  - Maintain MST's connected component as disjoint sets of vertices
  - e does not form a cycle iff its endpoints are in different components



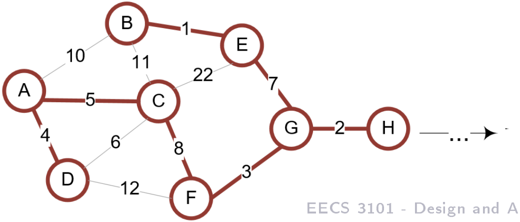
Kruskal's MST algorithm

- Initialize T to be Φ.
- Sort edges in the non-decreasing of their weights and process them one by one.
  $(B,E),(G,H),(G,F),(A,D),(A,C),(C,D),(E,G),(C,F),(A,B),(B,C),(D,F)$
- If an edge e does not form a cycle in MST, add it to MST.
  - Maintain MST's connected component as disjoint sets of vertices
  - e does not form a cycle iff its endpoints are in different components
- The time complexity of the Kruskal's algorithm is defined by the sorting of edges



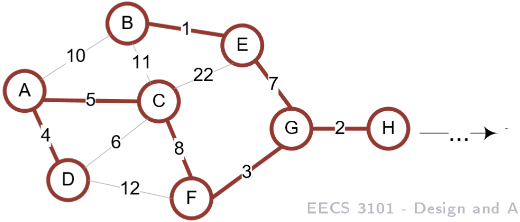
Kruskal's MST algorithm

- Initialize T to be Φ.
- Sort edges in the non-decreasing of their weights and process them one by one.
  $(B,E),(G,H),(G,F),(A,D),(A,C),(C,D),(E,G),(C,F),(A,B),(B,C),(D,F)$
- If an edge e does not form a cycle in MST, add it to MST.
  - Maintain MST's connected component as disjoint sets of vertices
  - e does not form a cycle iff its endpoints are in different components
- The time complexity of the Kruskal's algorithm is defined by the sorting of edges
  - **Kruskal's algorithm takes $O(m \log m)$ for a graph of m edges.**



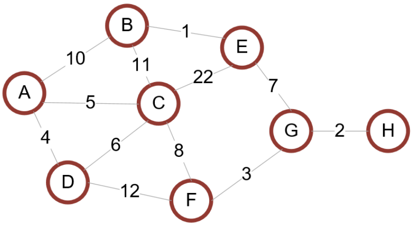
Kruskal's MST algorithm

- Initialize T to be Φ.
- Sort edges in the non-decreasing of their weights and process them one by one.
  $(B, E), (G, H), (G, F), (A, D), (A, C), (C, D), (E, G), (C, F), (A, B), (B, C), (D, F)$
- If an edge e does not form a cycle in MST, add it to MST.
  - Maintain MST's connected component as disjoint sets of vertices
  - e does not form a cycle iff its endpoints are in different components
- The time complexity of the Kruskal's algorithm is defined by the sorting of edges
  - **Kruskal's algorithm takes $O(m \log m)$ for a graph of m edges.**
    - Note that $O(m \log m) = O(m \log n)$ (why?)



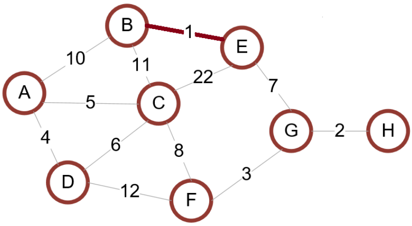
Prim's algorithm

- Initialize: let $T = \{$ an edge in the graph with minimum weight $\}$
- Repeat $n-2$ times:
  - $e =$ an edge in G of minimum weight that has one endpoint in T and one endpoint outside T
  - $T = T \cup \{e\}$



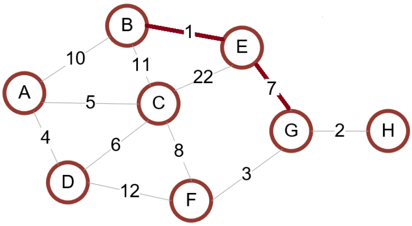
Prim's algorithm

- Initialize: let $T = \{$ an edge in the graph with minimum weight $\}$
- Repeat $n-2$ times:
  - $e =$ an edge in G of minimum weight that has one endpoint in T and one endpoint outside T
  - $T = T \cup \{e\}$



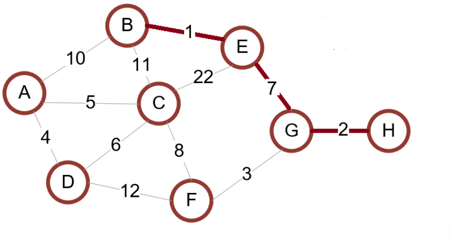
Prim's algorithm

- Initialize: let $T = \{$ an edge in the graph with minimum weight $\}$
- Repeat $n-2$ times:
  - $e =$ an edge in G of minimum weight that has one endpoint in T and one endpoint outside T
  - $T = T \cup \{e\}$

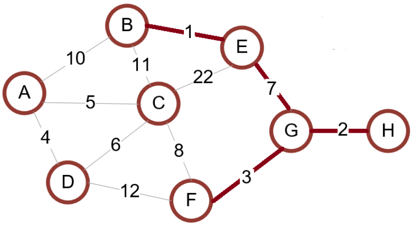


Prim's algorithm

- Initialize: let $T = \{$ an edge in the graph with minimum weight $\}$
- Repeat $n-2$ times:
  - $e =$ an edge in G of minimum weight that has one endpoint in T and one endpoint outside T
  - $T = T \cup \{e\}$

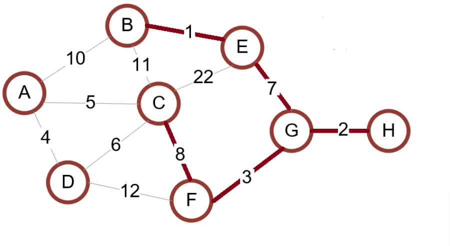


- Initialize: let $T = \{$ an edge in the graph with minimum weight $\}$
- Repeat $n - 2$ times:
  - $e =$ an edge in G of minimum weight that has one endpoint in T and one endpoint outside T
  - $T = T \cup \{e\}$



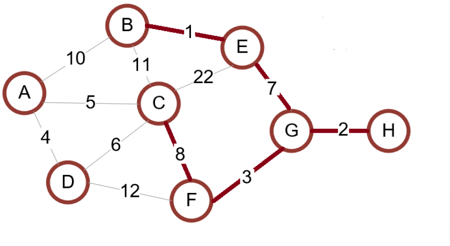
Prim's algorithm

- Initialize: let $T = \{$ an edge in the graph with minimum weight $\}$
- Repeat $n - 2$ times:
  - $e =$ an edge in G of minimum weight that has one endpoint in T and one endpoint outside T
  - $T = T \cup \{e\}$



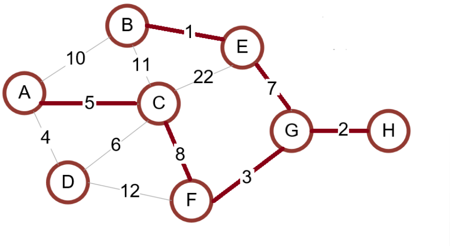
Prim's algorithm

- Initialize: let $T = \{$ an edge in the graph with minimum weight $\}$
- Repeat $n-2$ times:
  - $e =$ an edge in G of minimum weight that has one endpoint in T and one endpoint outside T
  - $T = T \cup \{e\}$



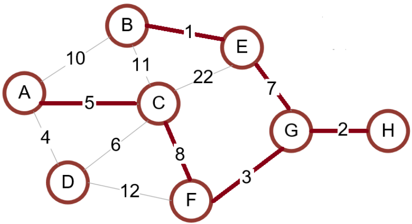
Prim's algorithm

- Initialize: let $T = \{$ an edge in the graph with minimum weight $\}$
- Repeat $n - 2$ times:
  - $e =$ an edge in G of minimum weight that has one endpoint in T and one endpoint outside T
  - $T = T \cup \{e\}$



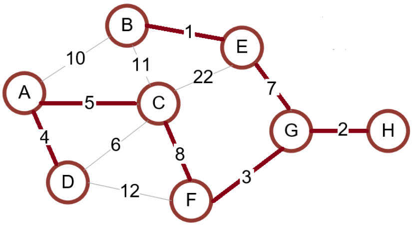
Prim's algorithm

- Initialize: let $T = \{$ an edge in the graph with minimum weight $\}$
- Repeat $n-2$ times:
  - $e =$ an edge in G of minimum weight that has one endpoint in T and one endpoint outside T
  - $T = T \cup \{e\}$



Prim's algorithm

- Initialize: let $T = \{$ an edge in the graph with minimum weight $\}$
- Repeat $n - 2$ times:
  - $e =$ an edge in G of minimum weight that has one endpoint in T and one endpoint outside T
  - $T = T \cup \{e\}$

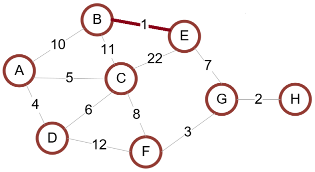


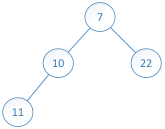
Prim's algorithm Implementation

- How to implement the Prim's algorithm?
  - Let T be the $\{e\}$ where e is the edge with min-weight
  - Insert edges incident to endpoints of e to an initially empty **min-heap** H
  - Repeatedly **extractMin** (to get the next edge e'), and insert edges incident to endpoints of e' to H.

Prim's algorithm Implementation

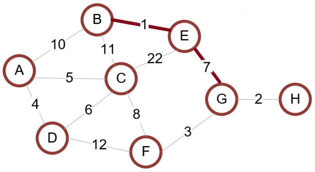
- How to implement the Prim's algorithm?
  - Let T be the $\{e\}$ where e is the edge with min-weight
  - Insert edges incident to endpoints of e to an initially empty **min-heap** H
  - Repeatedly **extractMin** (to get the next edge e'), and insert edges incident to endpoints of e' to H.

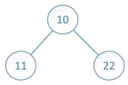




Prim's algorithm Implementation

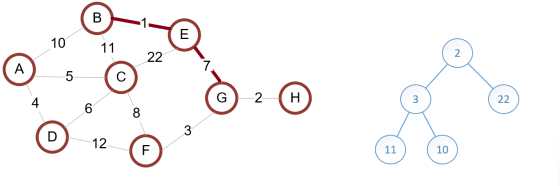
- How to implement the Prim's algorithm?
  - Let T be the $\{e\}$ where e is the edge with min-weight
  - Insert edges incident to endpoints of e to an initially empty **min-heap** H
  - Repeatedly **extractMin** (to get the next edge e'), and insert edges incident to endpoints of e' to H.





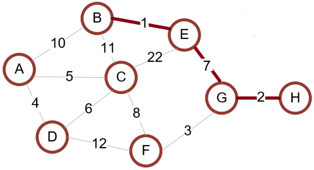
Prim's algorithm Implementation

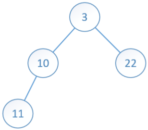
- How to implement the Prim's algorithm?
  - Let T be the $\{e\}$ where e is the edge with min-weight
  - Insert edges incident to endpoints of e to an initially empty **min-heap** H
  - Repeatedly **extractMin** (to get the next edge e'), and insert edges incident to endpoints of e' to H.



Prim's algorithm Implementation

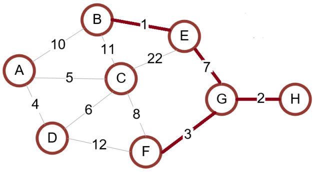
- How to implement the Prim's algorithm?
  - Let T be the $\{e\}$ where e is the edge with min-weight
  - Insert edges incident to endpoints of e to an initially empty **min-heap** H
  - Repeatedly **extractMin** (to get the next edge e'), and insert edges incident to endpoints of e' to H.

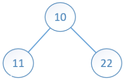




Prim's algorithm Implementation

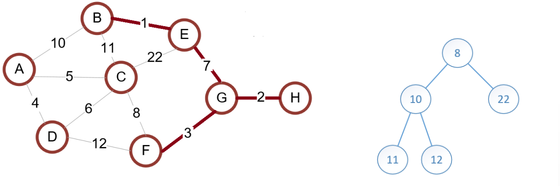
- How to implement the Prim's algorithm?
  - Let T be the $\{e\}$ where e is the edge with min-weight
  - Insert edges incident to endpoints of e to an initially empty **min-heap** H
  - Repeatedly **extractMin** (to get the next edge e'), and insert edges incident to endpoints of e' to H.





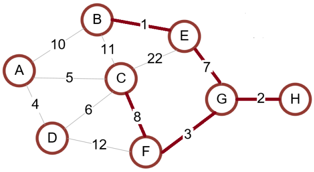
Prim's algorithm Implementation

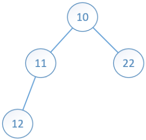
- How to implement the Prim's algorithm?
    - Let T be the $\{e\}$ where e is the edge with min-weight
    - Insert edges incident to endpoints of e to an initially empty **min-heap** H
    - Repeatedly **extractMin** (to get the next edge e'), and insert edges incident to endpoints of e' to H.



Prim's algorithm Implementation

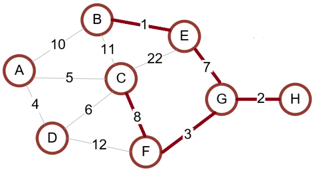
- How to implement the Prim's algorithm?
  - Let T be the $\{e\}$ where e is the edge with min-weight
  - Insert edges incident to endpoints of e to an initially empty **min-heap** H
  - Repeatedly **extractMin** (to get the next edge e'), and insert edges incident to endpoints of e' to H.

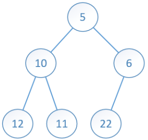




Prim's algorithm Implementation

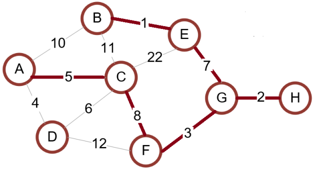
- How to implement the Prim's algorithm?
  - Let T be the $\{e\}$ where e is the edge with min-weight
  - Insert edges incident to endpoints of e to an initially empty **min-heap** H
  - Repeatedly **extractMin** (to get the next edge e'), and insert edges incident to endpoints of e' to H.

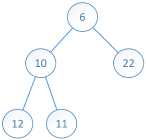




Prim's algorithm Implementation

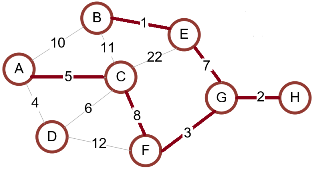
- How to implement the Prim's algorithm?
  - Let T be the $\{e\}$ where e is the edge with min-weight
  - Insert edges incident to endpoints of e to an initially empty **min-heap** H
  - Repeatedly **extractMin** (to get the next edge e'), and insert edges incident to endpoints of e' to H.

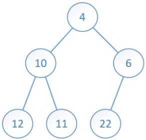




Prim's algorithm Implementation

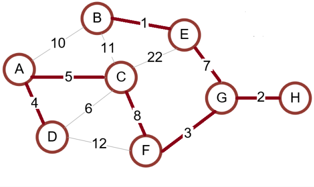
- How to implement the Prim's algorithm?
  - Let T be the $\{e\}$ where e is the edge with min-weight
  - Insert edges incident to endpoints of e to an initially empty **min-heap** H
  - Repeatedly **extractMin** (to get the next edge e'), and insert edges incident to endpoints of e' to H.

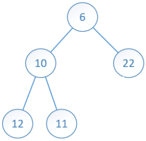




Prim's algorithm Implementation

- How to implement the Prim's algorithm?
  - Let T be the $\{e\}$ where e is the edge with min-weight
  - Insert edges incident to endpoints of e to an initially empty **min-heap** H
  - Repeatedly **extractMin** (to get the next edge e'), and insert edges incident to endpoints of e' to H.





Prim's Algorithm Running Time

- Each edge is inserted at most once and deleted at most once from the heap.
- At any given time, there are at most $m = |E| = O(n^2)$ edges in the heap
  - Insert and ExtractMax take $O(\log m) = O(\log(n^2)) = O(\log n)$ time.
- For all edges, we incur a cost of at most $O(m \log n)$.

Prim's Algorithm Running Time

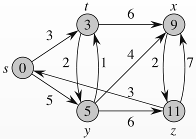
- Each edge is inserted at most once and deleted at most once from the heap.
- At any given time, there are at most $m = |E| = O(n^2)$ edges in the heap
  - Insert and ExtractMax take $O(\log m) = O(\log(n^2)) = O(\log n)$ time.
- For all edges, we incur a cost of at most $O(m \log n)$.

Theorem

Both Kruskal and Prim algorithms for finding minimum spanning tree take $O(m \log n)$ for a graph with n vertices and m edges.

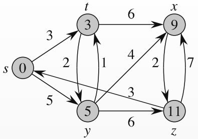
Single-source Shortest Path

- In a **shortest-paths problem**, we are given a weighted, directed graph $G = (V, E)$, with weight function w mapping edges to real-valued weights.
- The weight $w(p)$ of a path is the sum of the weights of its edges.
- In the single-source shortest path problem, we want to find a shortest path from a given source vertex $s \in V$ to each vertex $u \in V$.



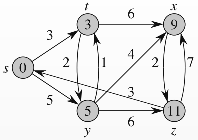
Single-source Shortest Path

- In a **shortest-paths problem**, we are given a weighted, directed graph $G = (V, E)$, with weight function w mapping edges to real-valued weights.
- The weight $w(p)$ of a path is the sum of the weights of its edges.
- In the single-source shortest path problem, we want to find a shortest path from a given source vertex $s \in V$ to each vertex $u \in V$.
  - One shortest path between s and x is s, t, x with weight 9.



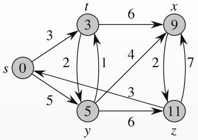
Single-source Shortest Path

- In a **shortest-paths problem**, we are given a weighted, directed graph $G = (V, E)$, with weight function w mapping edges to real-valued weights.
- The weight $w(p)$ of a path is the sum of the weights of its edges.
- In the single-source shortest path problem, we want to find a shortest path from a given source vertex $s \in V$ to each vertex $u \in V$.
  - One shortest path between s and x is s, t, x with weight 9.
  - Another shortest path between s and x is s, y, x with the same weight 9.



Single-source Shortest Path

- In a **shortest-paths problem**, we are given a weighted, directed graph $G = (V, E)$, with weight function w mapping edges to real-valued weights.
- The weight $w(p)$ of a path is the sum of the weights of its edges.
- In the single-source shortest path problem, we want to find a shortest path from a given source vertex $s \in V$ to each vertex $u \in V$.
  - One shortest path between s and x is s, t, x with weight 9.
  - Another shortest path between s and x is s, y, x with the same weight 9.
  - s, y, z, x is a path from s to x which is not a shortest path



Shortest Path Variants

- The algorithm for the single-source problem can solve many other problems:
  - **Single-destination shortest-paths**: just reverse the direction of each edge in the graph to reduce to a single-source problem.

Shortest Path Variants

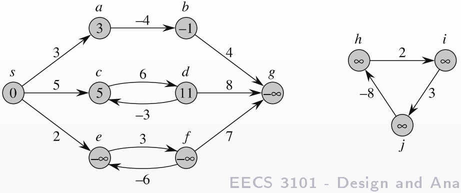
- The algorithm for the single-source problem can solve many other problems:
  - **Single-destination shortest-paths**: just reverse the direction of each edge in the graph to reduce to a single-source problem.
  - **Single-pair shortest-path**: find a shortest path from u to v for given pair of vertices (u, v).
    - Finding all shortest path from u to other vertices solves this problem too.
    - There is no faster algorithm for single-source shortest path.

Shortest Path Variants

- The algorithm for the single-source problem can solve many other problems:
  - **Single-destination shortest-paths**: just reverse the direction of each edge in the graph to reduce to a single-source problem.
  - **Single-pair shortest-path**: find a shortest path from u to v for given pair of vertices (u, v).
    - Finding all shortest path from u to other vertices solves this problem too.
    - There is no faster algorithm for single-source shortest path.
  - **All-pairs shortest-paths**: Find a shortest path from u to v for every pair of vertices u and v.
    - We can solve this problem by running a single-source algorithm once from each vertex.
    - But we usually can solve it faster as we will see later.

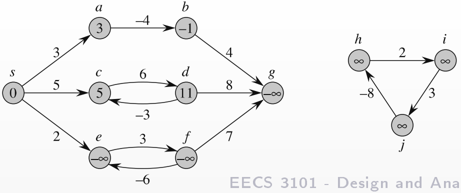
Negative Weights

- Negative weights are generally allowed (although most applications involve positive weights).



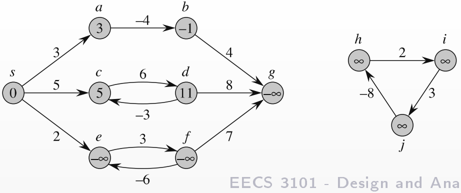
Negative Weights

- Negative weights are generally allowed (although most applications involve positive weights).
- If there is a negative cycle, the length of the shortest path for some vertices will be $-\infty$.



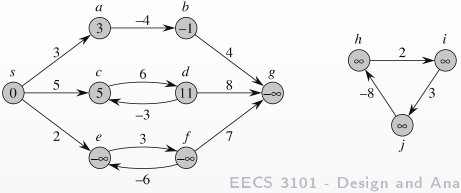
Negative Weights

- Negative weights are generally allowed (although most applications involve positive weights).
- If there is a negative cycle, the length of the shortest path for some vertices will be $-\infty$.
    - Some algorithms, e.g., Dijkstra's Algorithm, assume edge-weights are positive.



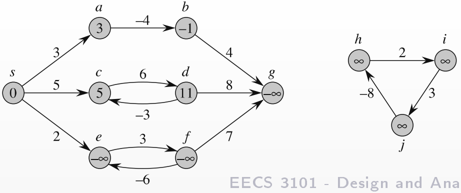
Negative Weights

- Negative weights are generally allowed (although most applications involve positive weights).
- If there is a negative cycle, the length of the shortest path for some vertices will be $-\infty$.
  - Some algorithms, e.g., Dijkstra's Algorithm, assume edge-weights are positive.
  - Some algorithms, e.g., Bellman-Ford, allow negative weights and return 'False' if a negative cycle exists.



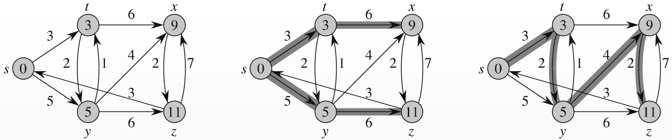
Negative Weights

- Negative weights are generally allowed (although most applications involve positive weights).
- If there is a negative cycle, the length of the shortest path for some vertices will be $-\infty$.
  - Some algorithms, e.g., Dijkstra's Algorithm, assume edge-weights are positive.
  - Some algorithms, e.g., Bellman-Ford, allow negative weights and return 'False' if a negative cycle exists.
- If the graph is disconnected, the length of the shortest path will be $+\infty$ for vertices in connected components that do not contain s.



Representing Shortest Paths

- A **Shortest Path Tree** represents the solution for single-source shortest path problem, assuming no negative cycle exists.
- A shortest-paths tree rooted at s is a directed subgraph $G' = (V', E')$, such that:
  - V' is the set of vertices reachable from s in G.
  - G' forms a rooted tree with root s.
  - for all $v \in V'$, the unique simple path from s to v in G' is a shortest path from s to v in G.
  - The parent of v in G' is called its **predecessor** and is denoted as $v.\pi$.



Initialization

- Our algorithms maintain a **shortest-path estimate** $v.d$ for each vertex v, which is an upper bound on the weight of a shortest path from source s to v.
- Estimates and parents are initialized as follows:

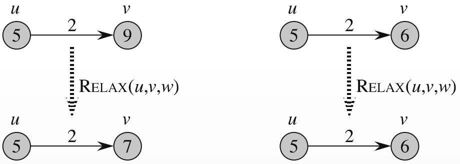
INITIALIZE-SINGLE-SOURCE(G, s)

for each vertex $v \in G.V$
    $v.d = \infty$
    $v.\pi = \text{NIL}$
$s.d = 0$

Relaxation

- We use a **Relax** procedure which takes and edge (u, v) and tests whether we can improve the shortest path to v found so far by going through u and, if so, updating $v.d$ and $v.\pi$.

```
RELAX(u, v, w)
if v.d > u.d + w(u, v)
    v.d = u.d + w(u, v)
    v.π = u
```



(a) (b)

Bellman-Ford Algorithm

- Given a weighted, directed graph G with source s and weight function w (with potentially negative weights), the **Bellman-Ford algorithm** returns a boolean value indicating whether or not there is a negative-weight cycle.
  - If there is such a cycle, the algorithm indicates that no solution exists.
  - If there is no such cycle, the algorithm produces the shortest paths and their weights.

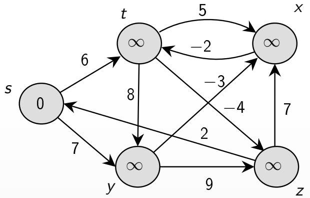
Bellman-Ford Algorithm

- The algorithm relaxes edges, progressively decreasing an estimate $v.d$ on the weight of a shortest path from the source s to each vertex v until it achieves the actual shortest-path weights.
- The algorithm returns TRUE if and only if the graph contains no negative-weight cycles that are reachable from the source.

```
BELLMAN-FORD(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
for i = 1 to |G.V| - 1
    for each edge (u, v) ∈ G.E
        RELAX(u, v, w)
for each edge (u, v) ∈ G.E
    if v.d > u.d + w(u, v)
        return FALSE
return TRUE
```

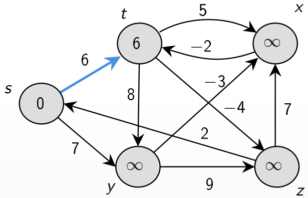
Bellman-Ford Example

- After Initialization, all estimates are ∞ except that $s.d = 0$.
- In Line 2, we iterate $|G.V| - 1 = 4$ times.
  - In each iteration, we go through all edges (in an arbitrary order) and relax them.
  - Suppose we relax edges in the order $(t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)$.



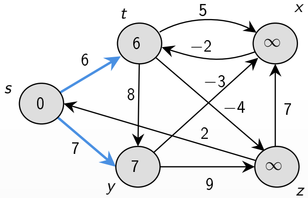
Bellman-Ford Example

- After Initialization, all estimates are ∞ except that $s.d = 0$.
- In Line 2, we iterate $|G.V| - 1 = 4$ times.
  - In each iteration, we go through all edges (in an arbitrary order) and relax them.
  - Suppose we relax edges in the order $(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$.



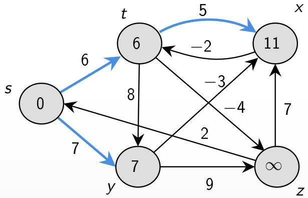
Bellman-Ford Example

- After Initialization, all estimates are ∞ except that $s.d = 0$.
- In Line 2, we iterate $|G.V| - 1 = 4$ times.
  - In each iteration, we go through all edges (in an arbitrary order) and relax them.
  - Suppose we relax edges in the order $(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$.



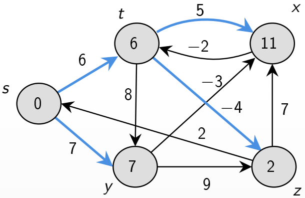
Bellman-Ford Example

- After Initialization, all estimates are ∞ except that $s.d = 0$.
- In Line 2, we iterate $|G.V| - 1 = 4$ times.
  - In each iteration, we go through all edges (in an arbitrary order) and relax them.
  - Suppose we relax edges in the order $(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$.



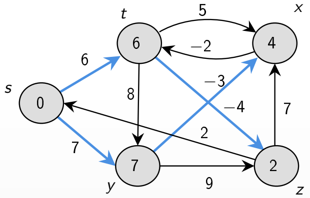
Bellman-Ford Example

- After Initialization, all estimates are ∞ except that $s.d = 0$.
- In Line 2, we iterate $|G.V| - 1 = 4$ times.
  - In each iteration, we go through all edges (in an arbitrary order) and relax them.
  - Suppose we relax edges in the order $(t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)$.



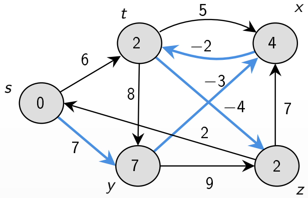
Bellman-Ford Example

- After Initialization, all estimates are ∞ except that $s.d = 0$.
- In Line 2, we iterate $|G.V| - 1 = 4$ times.
  - In each iteration, we go through all edges (in an arbitrary order) and relax them.
  - Suppose we relax edges in the order $(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$.



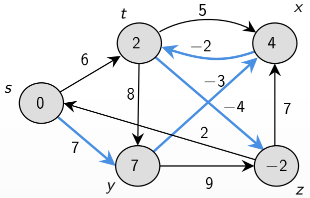
Bellman-Ford Example

- After Initialization, all estimates are ∞ except that $s.d = 0$.
- In Line 2, we iterate $|G.V| - 1 = 4$ times.
  - In each iteration, we go through all edges (in an arbitrary order) and relax them.
  - Suppose we relax edges in the order $(t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)$.



Bellman-Ford Example

- After Initialization, all estimates are ∞ except that $s.d = 0$.
- In Line 2, we iterate $|G.V| - 1 = 4$ times.
  - In each iteration, we go through all edges (in an arbitrary order) and relax them.
  - Suppose we relax edges in the order $(t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)$.



Bellman-Ford Analysis

- After iteration i, the estimate $v.d$ is the minimum distance from s to d using at most i edges (hops)
  - Since we have at most $|G.V| - 1$ edges on any shortest path, after $|G.V| - 1$ iteration, all estimates are shortest paths.

```
BELLMAN-FORD(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
for i = 1 to |G.V| - 1
    for each edge (u, v) ∈ G.E
        RELAX(u, v, w)
for each edge (u, v) ∈ G.E
    if v.d > u.d + w(u, v)
        return FALSE
return TRUE
```

Bellman-Ford Analysis

- After iteration i, the estimate $v.d$ is the minimum distance from s to d using at most i edges (hops)
  - Since we have at most $|G.V| - 1$ edges on any shortest path, after $|G.V| - 1$ iteration, all estimates are shortest paths.
- If we can still decrease the estimates after $|G.V| - 1$ iterations, there exists a negative cycle in the graph.

```
BELLMAN-FORD(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
for i = 1 to |G.V| - 1
    for each edge (u, v) ∈ G.E
        RELAX(u, v, w)
for each edge (u, v) ∈ G.E
    if v.d > u.d + w(u, v)
        return FALSE
return TRUE
```

Bellman-Ford Analysis

- After iteration i, the estimate $v.d$ is the minimum distance from s to d using at most i edges (hops)
  - Since we have at most $|G.V| - 1$ edges on any shortest path, after $|G.V| - 1$ iteration, all estimates are shortest paths.
- If we can still decrease the estimates after $|G.V| - 1$ iterations, there exists a negative cycle in the graph.
- The running time is $O(|V||E|) = O(mn)$.

```
BELLMAN-FORD(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
for i = 1 to |G.V| - 1
    for each edge (u, v) ∈ G.E
        RELAX(u, v, w)
for each edge (u, v) ∈ G.E
    if v.d > u.d + w(u, v)
        return FALSE
return TRUE
```

Dijkstra's Algorithm

- Dijkstra's algorithm solves the single-source shortest-paths problem on a weighted, directed graph G in which all **edge weights are nonnegative**.
- It is faster than Bellman-Ford but works under the above restriction (it fails when there are negative edges).

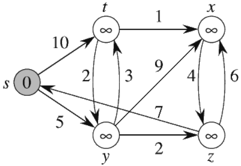
Dijkstra's Algorithm

- Dijkstra's algorithm maintains a set S of vertices whose final shortest-path weights from the source s have already been determined.
  - The algorithm repeatedly I) selects the vertex $u \in V - S$ with the minimum shortest-path estimate, II) adds u to S, and III)relaxes all edges leaving u.
  - We use a min-priority queue Q of vertices, keyed by their estimate d values.

```
DIJKSTRA(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
S = ∅
Q = G.V
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    S = S ∪ {u}
    for each vertex v ∈ G.Adj[u]
        RELAX(u, v, w)
```

Dijkstra's Exmaple

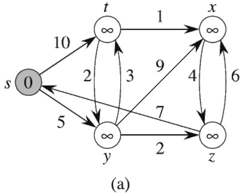
- Initially, $Q = G.V$, $S = \phi$, and $s.d = 0$ and $v.d = \infty$ for any $v \neq s$.
- Repeatedly take the vertex u with smallest estimate, add it to S, and relax edges leaving u.



(a)

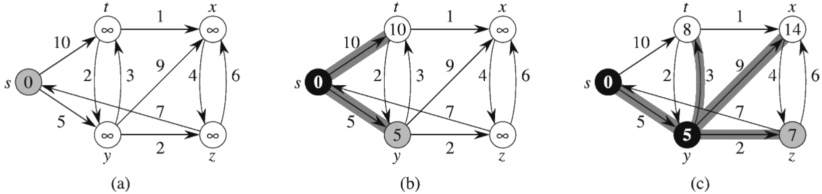
Dijkstra's Exmaple

- Initially, $Q = G.V$, $S = \phi$, and $s.d = 0$ and $v.d = \infty$ for any $v \neq s$.
- Repeatedly take the vertex u with smallest estimate, add it to S, and relax edges leaving u.



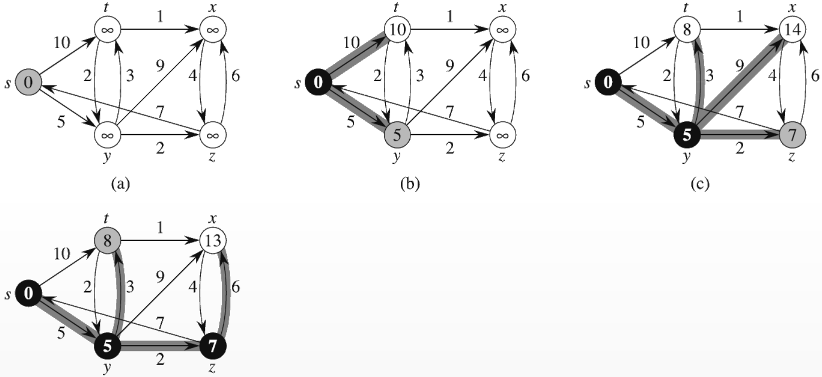
Dijkstra's Exmaple

- Initially, $Q = G.V$, $S = \phi$, and $s.d = 0$ and $v.d = \infty$ for any $v \neq s$.
- Repeatedly take the vertex u with smallest estimate, add it to S, and relax edges leaving u.



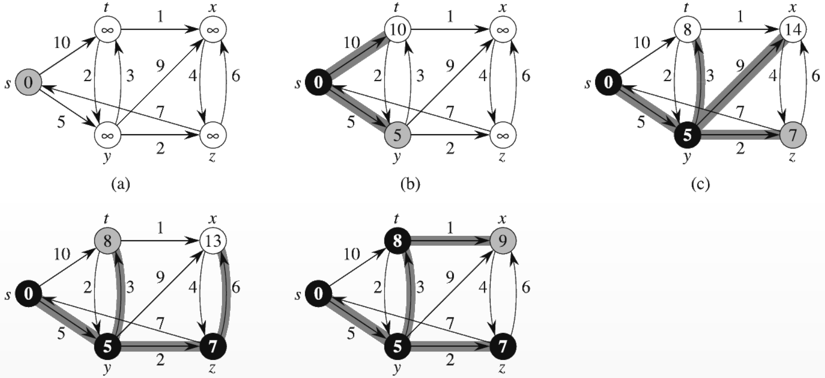
Dijkstra's Exmaple

- Initially, $Q = G.V$, $S = \phi$, and $s.d = 0$ and $v.d = \infty$ for any $v \neq s$.
- Repeatedly take the vertex u with smallest estimate, add it to S, and relax edges leaving u.



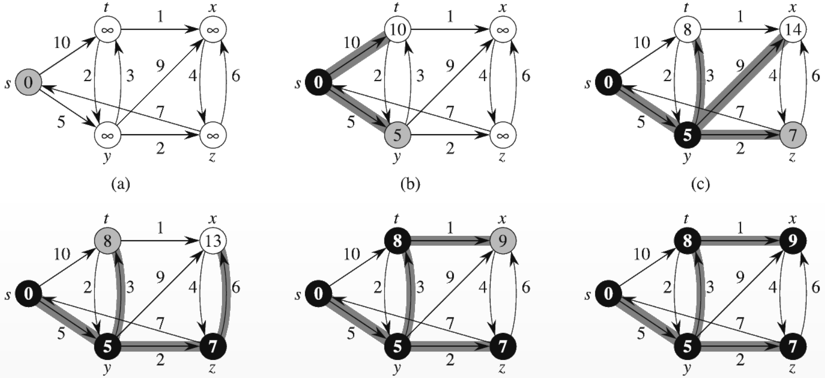
Dijkstra's Exmaple

- Initially, $Q = G.V$, $S = \phi$, and $s.d = 0$ and $v.d = \infty$ for any $v \neq s$.
- Repeatedly take the vertex u with smallest estimate, add it to S, and relax edges leaving u.



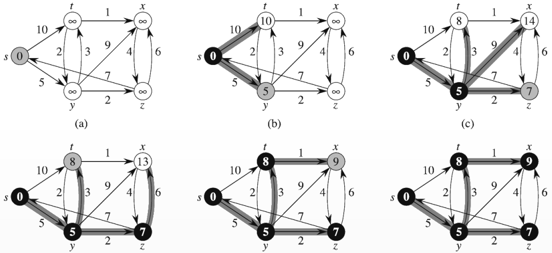
Dijkstra's Exmaple

- Initially, $Q = G.V$, $S = \phi$, and $s.d = 0$ and $v.d = \infty$ for any $v \neq s$.
- Repeatedly take the vertex u with smallest estimate, add it to S, and relax edges leaving u.



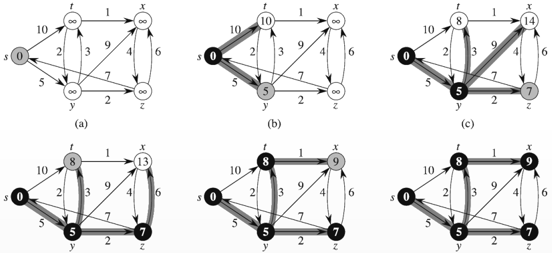
Dijkstra's Analysis

- Dijkstra's algorithm calculates the shortest path from s to every vertex.
  - Anytime we put a new vertex u in S (the vertices already added to the tree), we can say that we already know the shortest path from s to u.



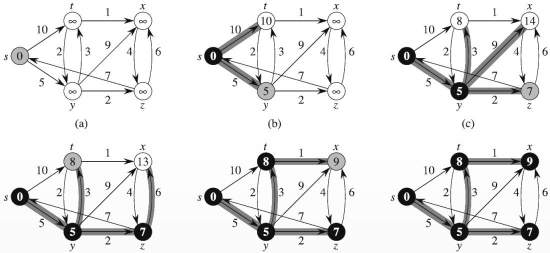
Dijkstra's Analysis

- Dijkstra's algorithm calculates the shortest path from s to every vertex.
  - Anytime we put a new vertex u in S (the vertices already added to the tree), we can say that we already know the shortest path from s to u.
  - Vertices are added to S in the sorted of their distance from s.



Dijkstra's Analysis

- Dijkstra's algorithm calculates the shortest path from s to every vertex.
  - Anytime we put a new vertex u in S (the vertices already added to the tree), we can say that we already know the shortest path from s to u.
  - Vertices are added to S in the sorted of their distance from s.
  - Notice similarities to BFS and Prim's algorithm for MTS.



Dijkstra's Algorithm

- What is the time complexity of the Dijkstra's algorithm?

```
DIJKSTRA(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
S = ∅
Q = G.V
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    S = S ∪ {u}
    for each vertex v ∈ G.Adj[u]
        RELAX(u, v, w)
```

Dijkstra's Algorithm

- What is the time complexity of the Dijkstra's algorithm?
  - Each vertex is extracted once from a priority queue of size n; summing to $\Theta(n \log n)$ for all vertices.
  - Each edge $e = (u, v)$ is visited exactly once (in Line 7, when we visit its starting point and relax e).

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5      u = EXTRACT-MIN(Q)
6      S = S ∪ {u}
7      for each vertex v ∈ G.Adj[u]
8          RELAX(u, v, w)
```

Dijkstra's Algorithm

- What is the time complexity of the Dijkstra's algorithm?
  - Each vertex is extracted once from a priority queue of size n; summing to $\Theta(n \log n)$ for all vertices.
  - Each edge $e = (u, v)$ is visited exactly once (in Line 7, when we visit its starting point and relax e).
  - After relax, we reduce the key of the endpoint v in Q; this takes $\log n$ times $\rightarrow$ we spend $O(m \log n)$ over all edges.

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5      u = EXTRACT-MIN(Q)
6      S = S ∪ {u}
7      for each vertex v ∈ G.Adj[u]
8          RELAX(u, v, w)
```

Dijkstra's Algorithm

- What is the time complexity of the Dijkstra's algorithm?
  - Each vertex is extracted once from a priority queue of size n; summing to $\Theta(n \log n)$ for all vertices.
  - Each edge $e = (u, v)$ is visited exactly once (in Line 7, when we visit its starting point and relax e).
  - After relax, we reduce the key of the endpoint v in Q; this takes $\log n$ times $\rightarrow$ we spend $O(m \log n)$ over all edges.
  - In total, the running time is $\Theta((m + n) \log n)$.

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5      u = EXTRACT-MIN(Q)
6      S = S ∪ {u}
7      for each vertex v ∈ G.Adj[u]
8          RELAX(u, v, w)
```

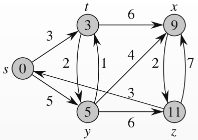
Dijkstra's Algorithm

- What is the time complexity of the Dijkstra's algorithm?
  - Each vertex is extracted once from a priority queue of size n; summing to $\Theta(n \log n)$ for all vertices.
  - Each edge $e = (u, v)$ is visited exactly once (in Line 7, when we visit its starting point and relax e).
  - After relax, we reduce the key of the endpoint v in Q; this takes $\log n$ times $\rightarrow$ we spend $O(m \log n)$ over all edges.
  - In total, the running time is $\Theta((m + n) \log n)$.
  - If we use **Fibonacci heaps** instead of binary heaps, we can improve the time complexity to $\Theta(m + n \log n)$.

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5      u = EXTRACT-MIN(Q)
6      S = S ∪ {u}
7      for each vertex v ∈ G.Adj[u]
8          RELAX(u, v, w)
```

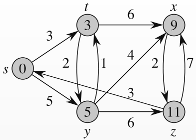
Single-source Shortest Path

- In a **shortest-paths problem**, we are given a weighted, directed graph $G = (V, E)$, with weight function w mapping edges to real-valued weights.
- The weight $w(p)$ of a path is the sum of the weights of its edges.
- In the single-source shortest path problem, we want to find a shortest path from a **given source** vertex $s \in V$ to each vertex $u \in V$.
    - The output is stored in a shortest path tree.



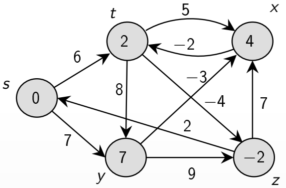
Single-source Shortest Path

- In a **shortest-paths problem**, we are given a weighted, directed graph $G = (V, E)$, with weight function w mapping edges to real-valued weights.
- The weight $w(p)$ of a path is the sum of the weights of its edges.
- In the single-source shortest path problem, we want to find a shortest path from a **given source** vertex $s \in V$ to each vertex $u \in V$.
  - The output is stored in a shortest path tree.
  - If negative weights are allowed, we use slower Bellman-Ford algorithm, which runs in $\Theta(mn)$; otherwise, we use the faster Dijkstra's algorithm, which runs in $\Theta((m+n)\log n)$.



All Pair Shortest Path problem

- Instead of the shortest path between a given source and other vertices, we are interested in the shortest distance between any pair of vertices.
  - We assume edge weights can be negative but no negative cycle.
- The output is an $n \times n$ matrix, where the (i, j) entry indicates the length of the shortest path from vertex i to vertex j.



	s	t	y	x	z
s	0	6	7	∞	∞
t	∞	0	8	5	-4
y	∞	∞	0	-3	9
x	∞	-2	∞	0	∞
z	2	∞	∞	7	0

input matrix w

	s	t	y	x	z
s	0	2	7	4	-2
t	-2	0	5	2	-4
y	-7	-5	0	-3	-9
x	-4	-2	3	0	-6
z	2	4	9	6	0

output

Preliminary Solutions

- Solution one: run Bellman-Ford algorithm $|V| = n$ times, once for each vertex as the source.

Preliminary Solutions

- Solution one: run Bellman-Ford algorithm $|V| = n$ times, once for each vertex as the source.
- The running time will be $\Theta(n^2 m)$, which is $\Theta(n^4)$ for dense graphs (when $m = \Theta(n^2)$).

Preliminary Solutions

- Solution one: run Bellman-Ford algorithm $|V| = n$ times, once for each vertex as the source.
- The running time will be $\Theta(n^2 m)$, which is $\Theta(n^4)$ for dense graphs (when $m = \Theta(n^2)$).
- Can we improve this? Yes, using Dynamic Programming.

Dynamic Programming Overview

- Recall the steps for devising a dynamic programming solution:
  - Characterize the structure of an optimal solution.
  - Recursively define the value of an optimal solution.
  - Compute the value of an optimal solution in a bottom-up fashion.

Matrix Multiplication Solution

- Let $l_{ij}^{(m)}$ be the minimum weight of any path from vertex i to vertex j that contains at most m edges.
  - For the base case, we have $l_{ij}^{(0)} = \begin{cases} 0 & \text{if } i = j \\ \infty & \text{if } i \neq j \end{cases}$

Matrix Multiplication Solution

- Let $l_{ij}^{(m)}$ be the minimum weight of any path from vertex i to vertex j that contains at most m edges.
  - For the base case, we have $l_{ij}^{(0)} = \begin{cases} 0 & \text{if } i = j \\ \infty & \text{if } i \neq j \end{cases}$
- For $m \geq 1$, we have two options:
  - Take the shortest path of length at most $m - 1$ from i to j, with weight $l_{ij}^{(m-1)}$.

Matrix Multiplication Solution

- Let $l_{ij}^{(m)}$ be the minimum weight of any path from vertex i to vertex j that contains at most m edges.
  - For the base case, we have $l_{ij}^{(0)} = \begin{cases} 0 & \text{if } i = j \\ \infty & \text{if } i \neq j \end{cases}$
- For $m \geq 1$, we have two options:
  - Take the shortest path of length at most $m - 1$ from i to j, with weight $l_{ij}^{(m-1)}$.
  - Take the shortest path of length at most $m - 1$ from i to a vertex k and then a single edge (hop) from k to j, this would have weight $l_{ik}^{(m-1)} + w_{kj}$.

Matrix Multiplication Solution

- Let $l_{ij}^{(m)}$ be the minimum weight of any path from vertex i to vertex j that contains at most m edges.
  - For the base case, we have $l_{ij}^{(0)} = \begin{cases} 0 & \text{if } i = j \\ \infty & \text{if } i \neq j \end{cases}$
- For $m \geq 1$, we have two options:
  - Take the shortest path of length at most $m-1$ from i to j, with weight $l_{ij}^{(m-1)}$.
  - Take the shortest path of length at most $m-1$ from i to a vertex k and then a single edge (hop) from k to j, this would have weight $l_{ik}^{(m-1)} + w_{kj}$.
  - The DP formula will be (the last inequality holds since $w_{jj} = 0$):

$$\begin{aligned} l_{ij}^{(m)} &= \min\left(l_{ij}^{(m-1)}, \min_{1\leq k\leq n}\left\{l_{ik}^{(m-1)} + w_{kj}\right\}\right) \\ &= \min_{1\leq k\leq n}\left\{l_{ik}^{(m-1)} + w_{kj}\right\} . \end{aligned}$$

Matrix Multiplication Solution

- The shortest distance between i and j will be stored at l_{ij}^{n-1}.
  - This is because the shortest path cannot contain more than $n-1$ edges (otherwise, there will be a loop in the path).

$$
\begin{aligned}
l_{ij}^{(m)} &= \min\left(l_{ij}^{(m-1)}, \min_{1\le k\le n}\left\{l_{ik}^{(m-1)} + w_{kj}\right\}\right) \\
&= \min_{1\le k\le n}\left\{l_{ik}^{(m-1)} + w_{kj}\right\} .
\end{aligned}
$$

Matrix Multiplication Solution

- The shortest distance between i and j will be stored at l_{ij}^{n-1}.
  - This is because the shortest path cannot contain more than $n-1$ edges (otherwise, there will be a loop in the path).
- In Step 3 of the DP solution, we compute a series of matrices $L^{(1)}, L^{(2)}, \ldots, L^{(n-1)}$, where $L^{(m)} = (l_{ij}^{(m)})$.
  - $L^{(1)} = W$ and $L^{(n)}$ contains all-pair shortest-path weights.

$$\begin{aligned} l_{ij}^{(m)} &= \min\left(l_{ij}^{(m-1)}, \min_{1 \le k \le n} \left\{l_{ik}^{(m-1)} + w_{kj}\right\}\right) \\ &= \min_{1 \le k \le n} \left\{l_{ik}^{(m-1)} + w_{kj}\right\} . \end{aligned}$$

Matrix Multiplication Solution

- The shortest distance between i and j will be stored at l_{ij}^{n-1}.
  - This is because the shortest path cannot contain more than $n-1$ edges (otherwise, there will be a loop in the path).
- In Step 3 of the DP solution, we compute a series of matrices $L^{(1)}, L^{(2)}, \ldots, L^{(n-1)}$, where $L^{(m)} = (l_{ij}^{(m)})$.
  - $L^{(1)} = W$ and $L^{(n)}$ contains all-pair shortest-path weights.
- The following procedure computes $L^{(m)}$ from $L^{(m-1)}$ and W.

$$
\begin{aligned}
l_{ij}^{(m)} &= \min\left(l_{ij}^{(m-1)}, \min_{1 \le k \le n} \left\{l_{ik}^{(m-1)} + w_{kj}\right\}\right) \\
&= \min_{1 \le k \le n} \left\{l_{ik}^{(m-1)} + w_{kj}\right\} .
\end{aligned}
$$

```
EXTEND-SHORTEST-PATHS(L, W)
n = L.rows
let L' = (l'_ij) be a new n × n matrix
for i = 1 to n
    for j = 1 to n
        l'_ij = ∞
        for k = 1 to n
            l'_ij = min(l'_ij, l_ik + w_kj)
return L'
```

Matrix Multiplication Solution

- Extend-Shortest-Paths is reminiscent of Matrix multiplication:
  - We take substitutions: $l^{(m-1)} \to a$, $w \to b$, $l^{(m)} \to c$, $\min \to +$, and $+ \to .$
  - Computing $l^{(m)}$ from l^{m-1} and W is similar to multiplying l^{m-1} and W.

```
EXTEND-SHORTEST-PATHS(L, W)
n = L.rows
let L' = (l'_ij) be a new n × n matrix
for i = 1 to n
    for j = 1 to n
        l'_ij = ∞
        for k = 1 to n
            l'_ij = min(l'_ij, l_ik + w_kj)
return L'
```

```
SQUARE-MATRIX-MULTIPLY(A, B)
n = A.rows
let C be a new n × n matrix
for i = 1 to n
    for j = 1 to n
        c_ij = 0
        for k = 1 to n
            c_ij = c_ij + a_ik · b_kj
return C
```

Matrix Multiplication Solution

- To find, $L^{(n)}$, we apply Extend-Shortest-Paths $n-1$ times.
  - This is similar to multiplying W by itself $n-1$ times (recall that $L^{(1)} = W$).

$$\begin{array}{rclcl}
L^{(1)} & = & L^{(0)} \cdot W & = & W \,, \\
L^{(2)} & = & L^{(1)} \cdot W & = & W^{2} \,, \\
L^{(3)} & = & L^{(2)} \cdot W & = & W^{3} \,, \\
 & & \vdots & & \\
L^{(n-1)} & = & L^{(n-2)} \cdot W & = & W^{n-1} \,.
\end{array}$$

SLOW-ALL-PAIRS-SHORTEST-PATHS(W)

```
n = W.rows
L^(1) = W
for m = 2 to n − 1
    let L^(m) be a new n × n matrix
    L^(m) = EXTEND-SHORTEST-PATHS(L^(m−1), W)
return L^(n−1)
```

Matrix Multiplication Solution

- The running time is similar to multiplying matrix W (an $n \times n$ matrix) by itself $n - 1$ times.
  - We can alliteratively do it in a naive way in $O(n^4)$.

Matrix Multiplication Solution

- The running time is similar to multiplying matrix W (an $n \times n$ matrix) by itself $n-1$ times.
  - We can alliteratively do it in a naive way in $O(n^4)$.
  - Alternatively, we can recursively find the outcome of the first $n/2$ multiplications (that is, $W^{n/2}$), and multiply it with itself in $O(n^3)$.

Matrix Multiplication Solution

- The running time is similar to multiplying matrix W (an $n \times n$ matrix) by itself $n-1$ times.
  - We can alliteratively do it in a naive way in $O(n^4)$.
  - Alternatively, we can recursively find the outcome of the first $n/2$ multiplications (that is, $W^{n/2}$), and multiply it with itself in $O(n^3)$.
    - This would take $\Theta(n^3 \log n)$.

Matrix Multiplication Solution

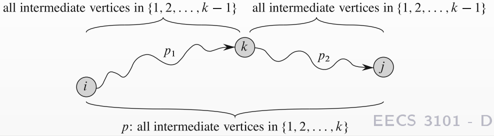
- The running time is similar to multiplying matrix W (an $n \times n$ matrix) by itself $n-1$ times.
  - We can alliteratively do it in a naive way in $O(n^4)$.
  - Alternatively, we can recursively find the outcome of the first $n/2$ multiplications (that is, $W^{n/2}$), and multiply it with itself in $O(n^3)$.
    - This would take $\Theta(n^3 \log n)$.
- The running time of $\Theta(n^3 \log n)$ is better than $\Theta(n^2|E|) = \Theta(n^4)$ of repeating Bellman-Ford algorithm. But we can still do better.

Floyd-Warshall Algorithm

- Another DP algorithm for all-pair shortest path, developed independently by Roy [1959], Floyd [1962], Warshall [1962].
- Given a path $(v_1, v_2, \ldots, v_m)$, we call vertices v_k with $k \in \{2, 3, \ldots, m-1\}$ **intermediate vertex**.

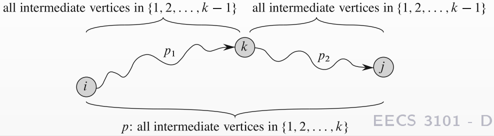
Floyd-Warshall Algorithm

- Let $d_{ij}^{(k)}$ denote the weight of the shortest path between i and j, subject to the intermediate vertices be contained in $\{1, \ldots, k\}$.



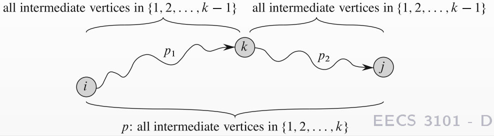
Floyd-Warshall Algorithm

- Let $d_{ij}^{(k)}$ denote the weight of the shortest path between i and j, subject to the intermediate vertices be contained in $\{1, \ldots, k\}$.
- Write a recursive formula for $d_{ij}^{(k)}$.



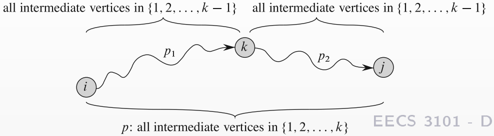
Floyd-Warshall Algorithm

- Let $d_{ij}^{(k)}$ denote the weight of the shortest path between i and j, subject to the intermediate vertices be contained in $\{1, \ldots, k\}$.
- Write a recursive formula for $d_{ij}^{(k)}$.
  - In the base case, we have $k = 0$ (no intermediate vertex), and we have $d_{ij}^{(0)} = w_{ij}$.



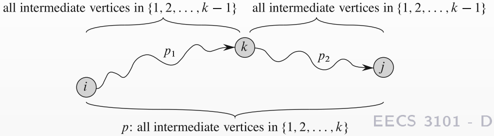
Floyd-Warshall Algorithm

- Let $d_{ij}^{(k)}$ denote the weight of the shortest path between i and j, subject to the intermediate vertices be contained in $\{1, \ldots, k\}$.
- Write a recursive formula for $d_{ij}^{(k)}$.
  - In the base case, we have $k = 0$ (no intermediate vertex), and we have $d_{ij}^{(0)} = w_{ij}$.
  - When $k > 0$, vertex k may or may not be an intermediate vertex.



Floyd-Warshall Algorithm

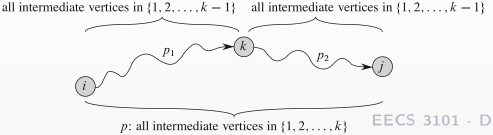
- Let $d_{ij}^{(k)}$ denote the weight of the shortest path between i and j, subject to the intermediate vertices be contained in $\{1, \dots, k\}$.
- Write a recursive formula for $d_{ij}^{(k)}$.
  - In the base case, we have $k = 0$ (no intermediate vertex), and we have $d_{ij}^{(0)} = w_{ij}$.
  - When $k > 0$, vertex k may or may not be an intermediate vertex.
    - If k is not an intermediate vertex, the weight of the shortest path will be $d_{ij}^{(k-1)}$.



Floyd-Warshall Algorithm

- Let $d_{ij}^{(k)}$ denote the weight of the shortest path between i and j, subject to the intermediate vertices be contained in $\{1, \ldots, k\}$.
- Write a recursive formula for $d_{ij}^{(k)}$.
  - In the base case, we have $k = 0$ (no intermediate vertex), and we have $d_{ij}^{(0)} = w_{ij}$.
  - When $k > 0$, vertex k may or may not be an intermediate vertex.
    - If k is not an intermediate vertex, the weight of the shortest path will be $d_{ij}^{(k-1)}$.
    - If k is an intermediate vertex, the weight of the shortest path will be $d_{ik}^{(k-1)}$ (the shortest distance from i to k) plus $d_{k,j}^{(k-1)}$ (the shortest distance from k to j). So we can write:

$$d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\}$$



Floyd-Warshall Algorithm

- The recursive DP formula for the Floyd-Warshall algorithm is thus:

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0\,, \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1\,. \end{cases}$$

Floyd-Warshall Algorithm

- The recursive DP formula for the Floyd-Warshall algorithm is thus:

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0\,, \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1\,. \end{cases}$$

- For filling the table, we only need to look at the previous value of k:

```
FLOYD-WARSHALL(W)
n = W.rows
D^(0) = W
for k = 1 to n
    let D^(k) = (d_ij^(k)) be a new n × n matrix
    for i = 1 to n
        for j = 1 to n
            d_ij^(k) = min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
return D^(n)
```

Floyd-Warshall Algorithm

- The recursive DP formula for the Floyd-Warshall algorithm is thus:

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0\,, \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1\,. \end{cases}$$

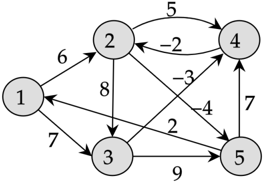
- For filling the table, we only need to look at the previous value of k:
  - The running time is clearly $\Theta(n^3)$, which is an improvement over $\Theta(n^3 \log n)$ of the Matrix-Multiplication method (and $\Theta(n^4)$ of repeating Bellman-Ford algorithm).

FLOYD-WARSHALL(W)

$n = W.rows$
$D^{(0)} = W$
for $k = 1$ **to** n
    let $D^{(k)} = \left(d_{ij}^{(k)}\right)$ be a new $n \times n$ matrix
    for $i = 1$ **to** n
        for $j = 1$ **to** n
            $d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$
return $D^{(n)}$

Floyd-Warshall Example

- Recall that $d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$

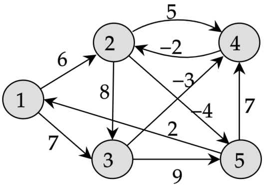


	1	2	3	4	5
1	0	6	7	∞	∞
2	∞	0	8	5	-4
3	∞	∞	0	-3	9
4	∞	-2	∞	0	∞
5	2	∞	∞	7	0

$D^0 = w$

Floyd-Warshall Example

- Recall that $d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$



	1	2	3	4	5
1	0	6	7	∞	∞
2	∞	0	8	5	-4
3	∞	∞	0	-3	9
4	∞	-2	∞	0	∞
5	2	∞	∞	7	0

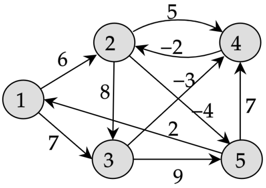
$D^0 = w$

	1	2	3	4	5
1	0	6	7	∞	∞
2	∞	0	8	5	-4
3	∞	∞	0	-3	9
4	∞	-2	∞	0	∞
5	2	8	9	7	0

D^1

Floyd-Warshall Example

- Recall that $d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$



	1	2	3	4	5
1	0	6	7	∞	∞
2	∞	0	8	5	-4
3	∞	∞	0	-3	9
4	∞	-2	∞	0	∞
5	2	∞	∞	7	0

$D^0 = w$

	1	2	3	4	5
1	0	6	7	∞	∞
2	∞	0	8	5	-4
3	∞	∞	0	-3	9
4	∞	-2	∞	0	∞
5	2	8	9	7	0

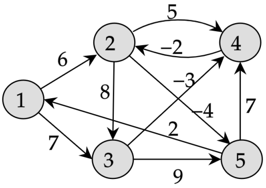
D^1

	1	2	3	4	5
1	0	6	7	11	2
2	∞	0	8	5	-4
3	∞	∞	0	-3	9
4	∞	-2	6	0	−6
5	2	8	9	7	0

D^2

Floyd-Warshall Example

- Recall that $d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$



	1	2	3	4	5
1	0	6	7	∞	∞
2	∞	0	8	5	-4
3	∞	∞	0	-3	9
4	∞	-2	∞	0	∞
5	2	∞	∞	7	0

$D^0 = w$

	1	2	3	4	5
1	0	6	7	∞	∞
2	∞	0	8	5	-4
3	∞	∞	0	-3	9
4	∞	-2	∞	0	∞
5	2	8	9	7	0

D^1

	1	2	3	4	5
1	0	6	7	11	2
2	∞	0	8	5	-4
3	∞	∞	0	-3	9
4	∞	-2	6	0	−6
5	2	8	9	7	0

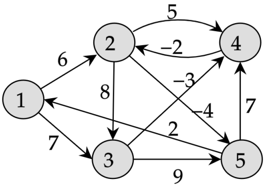
D^2

	1	2	3	4	5
1	0	6	7	4	2
2	∞	0	8	5	-4
3	∞	∞	0	-3	9
4	∞	-2	6	0	−6
5	2	8	9	6	0

D^3

Floyd-Warshall Example

- Recall that $d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$



	1	2	3	4	5
1	0	6	7	∞	∞
2	∞	0	8	5	-4
3	∞	∞	0	-3	9
4	∞	-2	∞	0	∞
5	2	∞	∞	7	0

$D^0 = w$

	1	2	3	4	5
1	0	6	7	∞	∞
2	∞	0	8	5	-4
3	∞	∞	0	-3	9
4	∞	-2	∞	0	∞
5	2	8	9	7	0

D^1

	1	2	3	4	5
1	0	6	7	11	2
2	∞	0	8	5	-4
3	∞	∞	0	-3	9
4	∞	-2	6	0	−6
5	2	8	9	7	0

D^2

	1	2	3	4	5
1	0	6	7	4	2
2	∞	0	8	5	-4
3	∞	∞	0	-3	9
4	∞	-2	6	0	−6
5	2	8	9	6	0

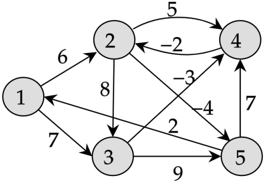
D^3

	1	2	3	4	5
1	0	2	7	4	−2
2	∞	0	8	5	-4
3	∞	−5	0	-3	-9
4	∞	-2	6	0	−6
5	2	4	9	6	0

D^4

Floyd-Warshall Example

- Recall that $d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$



	1	2	3	4	5
1	0	6	7	∞	∞
2	∞	0	8	5	-4
3	∞	∞	0	-3	9
4	∞	-2	∞	0	∞
5	2	∞	∞	7	0

$D^0 = w$

	1	2	3	4	5
1	0	6	7	∞	∞
2	∞	0	8	5	-4
3	∞	∞	0	-3	9
4	∞	-2	∞	0	∞
5	2	8	9	7	0

D^1

	1	2	3	4	5
1	0	6	7	11	2
2	∞	0	8	5	-4
3	∞	∞	0	-3	9
4	∞	-2	6	0	−6
5	2	8	9	7	0

D^2

	1	2	3	4	5
1	0	6	7	4	2
2	∞	0	8	5	-4
3	∞	∞	0	-3	9
4	∞	-2	6	0	−6
5	2	8	9	6	0

D^3

	1	2	3	4	5
1	0	2	7	4	−2
2	∞	0	8	5	-4
3	∞	−5	0	-3	-9
4	∞	-2	6	0	−6
5	2	4	9	6	0

D^4

	1	2	3	4	5
1	0	2	7	4	−2
2	-2	0	5	2	-4
3	−7	−5	0	-3	-9
4	−4	-2	3	0	−6
5	2	4	9	6	0

D^5

All Shortes Paths Summary

- The simple repetition of Bellman-Ford runs in $\Theta(n^2 m^2)$, wich is $\Theta(n^4)$ for dense graphs.
- The first DP solution, which resembles matrix multiplication, runs in $\Theta(n^3 \log n)$.
- Floyd-Warshall's algorithm is another DP solution that runs in $\Theta(n^3)$.