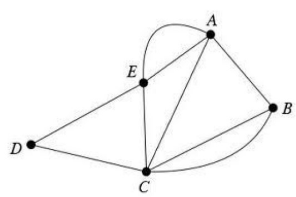
Introduction to Graphs

Outline

1. Basic graph terminology
2. Different representations
   – Adjacency matrix
   – Adjacency lists
3. Graph traversal algorithms
   – Depth First Search
   – Breadth First search
4. Examples of Graph Algorithms
   – Shortest Path Algorithm
   – Topological Sorting
   – Minimum Spanning Tree

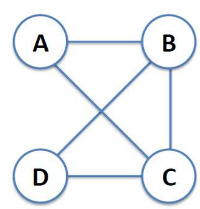
Graphs

- Similar to trees, graphs are made up of **vertices (nodes)** and **edges (links)** between those vertices.
- A **vertex** is referenced by a name/label, or index.
- An **edge** is referenced by the pair of vertices, such as (A, B), that it connects.
- Here an edge reflects the relationship between vertices, and its length does not matter.
- How is a graph different from a tree?



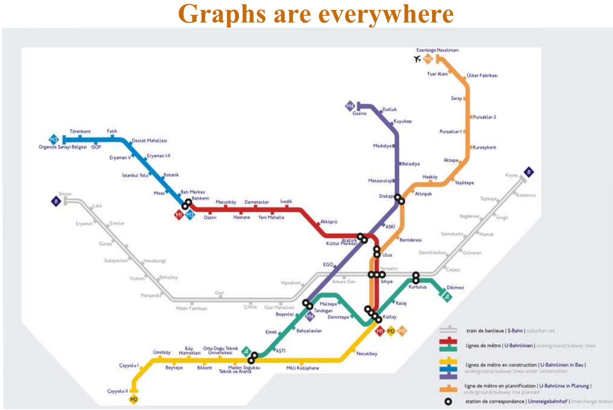
Formalism

- A **graph** G = (V, E) consists of
  – a set of ***vertices (nodes)***, V, and
  – a set of ***edges***, E, where each edge is a pair (v,w) such that v,w ∈ V




```
V = {A, B, C, D}
E = {(A,B),(A,C),(B,C),
     (B,D),(C,D)}
```


Graphs are everywhere

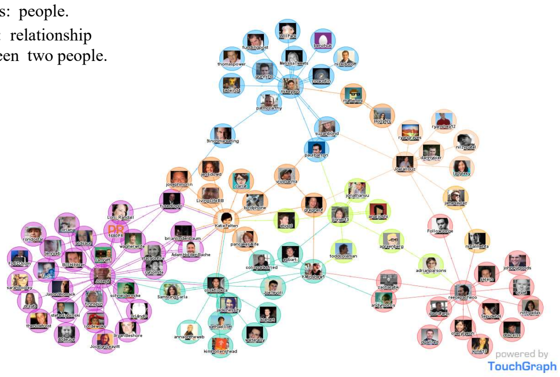


Road network

Node = intersection; edge = one-way street.

Social Network

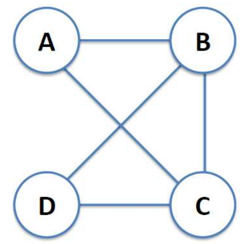
- Nodes: people.
- Edge: relationship between two people.



Undirected Graphs

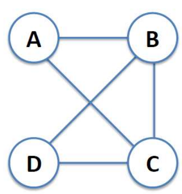
Definition:
An **undirected** graph is a graph where the pairings representing each edge are unordered

- That is, a graph in which the edges have no direction
- (A, B) and (B, A) refer to the same edge

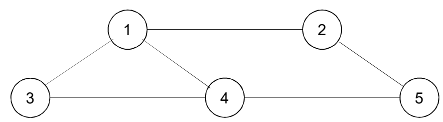


Graph Terminology

- **Adjacency**: two vertices are adjacent if they are directly connected by an edge.
- **Neighbors**: the set of vertices that are adjacent to a given vertex.
- **Path**: is a sequence of nodes where each consecutive node is connected by an edge.
- Note that there may be multiple paths that connect two vertices!
- Examples: A->C, A->B->C A->B->D->C



An Example Undirected Graph



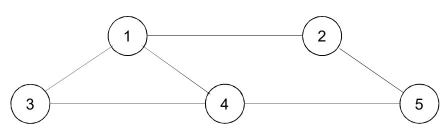
The graph G= (V,E) :
V = {1,2,3,4,5}
E = {1-2, 1-3, 1-4, 2-5, 3-4, 4-5}
|V| = 5, |E| = 6

- *Adjacent:*
  1 and 2 are adjacent -- 1 is adjacent to 2 and 2 is adjacent to 1
- *Neighbors:*
  {2, 3, 4} are neighbors of 1

Undirected Graphs Details

- A path in an undirected graph G = (V, E) is a sequence of nodes $v_1, v_2, \ldots, v_{k-1}, v_k$ with the property that each consecutive pair v_i, v_{i+1} is joined by an edge in E.
- A path is simple if all nodes are distinct.
- The length of a path is the number of edges in the path (or the number of vertices minus 1)
- Distance from u to v is the minimum number of edges in a u-v path.

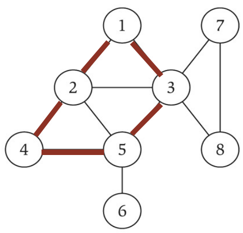
Undirected Graph



- *Path:*
  1,2,5 (a simple path), 1,3,4,1,2,5 (a path, but not a simple path)
- *Path length:*
  1,2,5,4 is a path of length 3
  1,3,4,1,2,5 is a path of length 5
- *Distance:*
  Distance from node 3 to node 2 is 2

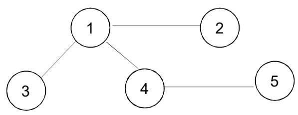
Cycles

- A cycle is a path $v_1, v_2, \ldots, v_{k-1}, v_k$ in which $v_1 = v_k$ and $k > 2$.
- A cycle is simple if the first k-1 nodes are all distinct.



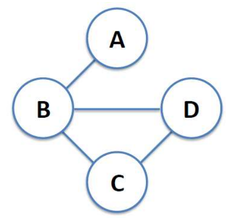
cycle C = 1-2-4-5-3-1

A graph that has no cycles is called acyclic:

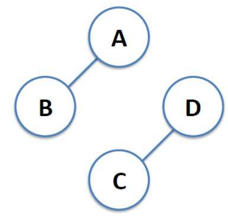


Connectivity

- **Connected graph:** is a graph that has **at least one <u>path</u> between every pair of distinct vertices**.



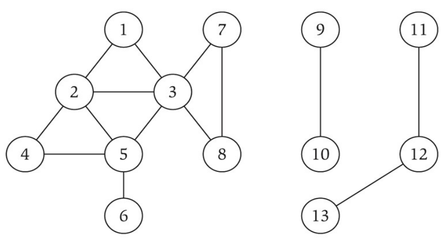
Connected



Not connected

Connectivity

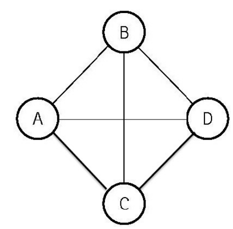
- **Connected component**: maximal subset of nodes such that a path exists between each pair in the set.



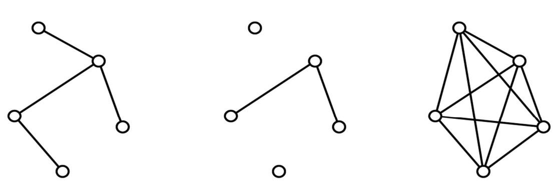
Complete Undirected Graph

- **Complete graph**: is a graph that has **an <u>edge</u> between every pair of distinct vertices.**
  - i.e. It has the maximum number of edges connecting vertices
  - A complete graph is also a connected graph. But a connected graph may not be a complete graph.

Example: a complete graph with 4 vertices has a total of 6 edges.



Undirected Graph -- Definitions

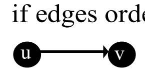


(a) **connected** (b) **disconnected** (c) **complete**

Directed Graphs

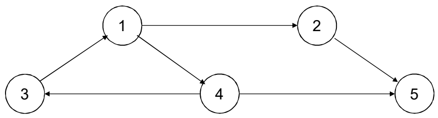
- A **directed graph,** sometimes referred to as *digraph* , is a graph where the edges are ordered pairs of vertices.

  if edges ordered pairs (u,v)



- This means that edges (u,v) and (v,u) are **different** edges.

Directed Graph – An Example



The graph G= (V,E) has 5 vertices and 6 edges:
V = {1,2,3,4,5}
E = { (1,2),(1,4),(2,5),(4,5),(3,1),(4,3) }

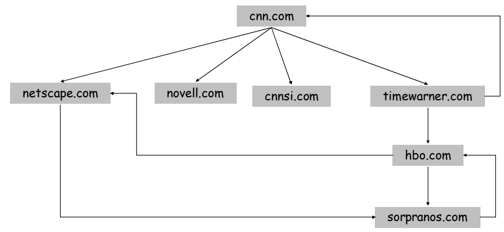
|V| = 5, |E| = 6

- *Adjacent*:
  2 is adjacent to 1, but 1 is NOT adjacent to 2

e.g. WWW is a graph – Directed or undirected?

World Wide Web

Web graph.
- Node: web page.
- Edge: hyperlink from one page to another.
- Modern web search engines exploit hyperlink structure to rank web pages by importance.



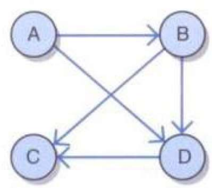
Directed Graph Definitions

- Most definitions extend naturally to directed graphs by mapping the word “edge” to “directed edge”
- When referring to a directed graph, the words “path” and “cycle” mean “directed path” and “directed cycle”

- **Directed path**: sequence $P = v_1, v_2, \ldots, v_{k-1}, v_k$ such that each consecutive pair v_i, v_{i+1} is joined by a *directed edge* in *G*.
  $v_1 \to v_k$ path.
- **Directed cycle**: directed path with $v_1 = v_k$
- Connected? Connected component? More subtle, because now there can be a path from *s* to *t* but not vice versa.

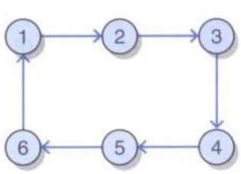
Connected Directed Graphs

- A directed graph is connected if for every pair of ordered vertices there is a directed path between them.

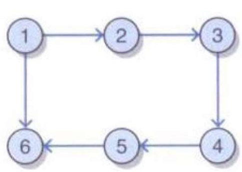


- Is there a path from A to C?
- Is there a path from C to A?
- Is this directed graph a connected graph?

Connected Directed Graphs



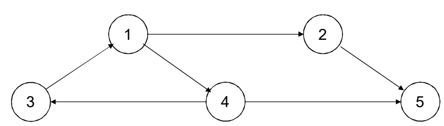
connected



unconnected

Connectivity in Directed Graphs

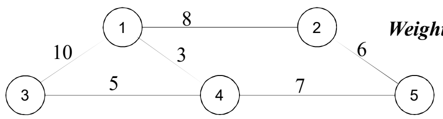
- In a *directed* graph, if there is a directed path from every vertex to every other vertex, that directed graph is called **strongly connected**.
  – If a directed graph is not strongly connected, but the underlying graph (without direction to arcs) is connected then the graph is **weakly connected**.



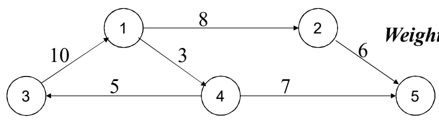
weakly connected directed graph

Weighted Graph

- When we label the edges of a graph with numeric values, the graph is called a ***weighted graph***.



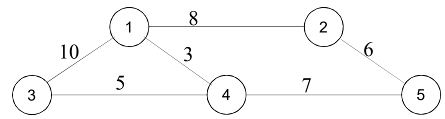
Weighted Undirected Graph



Weighted Directed Graph

Weighted Graphs

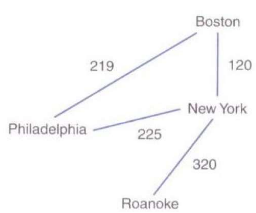
- If there is a path between two vertices, the path weight is the sum of the weights of the edges in the path.
- As you may have multiple paths, each path may have a different weight.
- In many cases, you may want the path with the minimum / maximum weight.



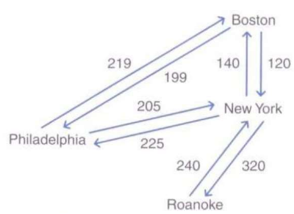
The path from node 3 to node 1 through 4: 8

Weighted Graphs

- On a directed graph, the weight may be different depending on the direction (e.g. airline price may not be symmetric).



Undirected Weighted Graph



Directed Weighted Graph

Graph Implementations

- The two most common implementations of a graph are:
  - ***Adjacency Matrix***
    - A two dimensional array
  - ***Adjacency List***
    - For each vertex we keep a list of adjacent vertices

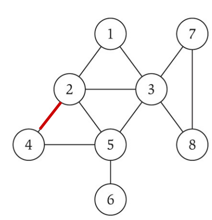
Adjacency Matrix

- An ***adjacency matrix*** for a graph with *n* vertices numbered 0,1,...,n-1 is an *n* x *n* array *matrix* such that *matrix*[*i*][*j*] is 1 (true) if there is an edge from vertex *i* to vertex *j*, and 0 (false) otherwise.
  - When the graph is *weighted*, *matrix*[*i*][*j*] is the weight on the edge from vertex *i* to vertex *j*; and if there is no edge from vertex *i* to vertex *j* *matrix*[*i*][*j*] is equal to ∞
- Adjacency matrix for an undirected graph is symmetrical.
  - i.e. *matrix*[*i*][*j*] is equal to *matrix*[*j*][*i*]
- Space requirement $O(|V|^2)$
- Acceptable if the graph is dense.

Unweighted, Undirected Graph Representation

Adjacency matrix. n-by-n matrix with $A_{uv} = 1$ if (u, v) is an edge.

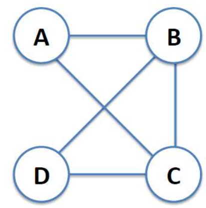
- The matrix is binary and symmetric.



	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	1	1	0	0	0
3	1	1	0	0	1	0	1	1
4	0	1	0	1	1	0	0	0
5	0	1	1	1	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0

Representing Graphs

- The adjacency matrix captures all the edge information. From it, you can calculate, for example:
  - The number of neighbors each vertex has. How?
  - Is there a path between two vertices? If so, what's the path length?
  - Is this a connected graph?

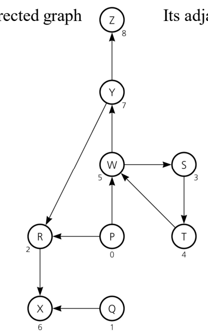


to (columns), from (rows)

	A	B	C	D
A	0	1	1	0
B	1	0	1	1
C	1	1	0	1
D	0	1	1	0

Unweighted, Directed Graph

A directed graph



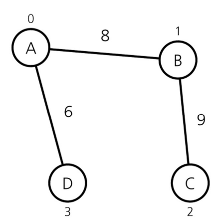
Its adjacency matrix

		0	1	2	3	4	5	6	7	8
		P	Q	R	S	T	W	X	Y	Z
0	P	0	0	1	0	0	1	0	0	0
1	Q	0	0	0	0	0	0	1	0	0
2	R	0	0	0	0	0	0	1	0	0
3	S	0	0	0	0	1	0	0	0	0
4	T	0	0	0	0	0	1	0	0	0
5	W	0	0	0	1	0	0	0	1	0
6	X	0	0	0	0	0	0	0	0	0
7	Y	0	0	1	0	0	0	0	0	1
8	Z	0	0	0	0	0	0	0	0	0

The matrix is binary and generally non-symmetric

Weighted, Undirected Graph

An Undirected Weighted Graph

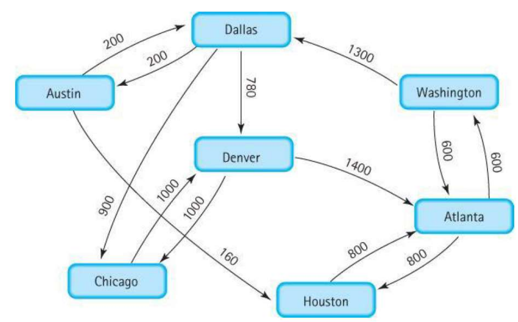


Its Adjacency Matrix

		0	1	2	3
		A	B	C	D
0	A	∞	8	∞	6
1	B	8	∞	9	∞
2	C	∞	9	∞	∞
3	D	6	∞	∞	∞

Weighted, Directed Graph

- Example of weighted, directed graph.



Adjacency Matrix

graph
.numVertices 7
.vertices

[0]	"Atlanta "
[1]	"Austin "
[2]	"Chicago "
[3]	"Dallas "
[4]	"Denver "
[5]	"Houston "
[6]	"Washington"
[7]	
[8]	
[9]	

.edges

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
[0]	0	0	0	0	0	800	600	•	•	•
[1]	0	0	0	200	0	160	0	•	•	•
[2]	0	0	0	0	1000	0	0	•	•	•
[3]	0	200	900	0	780	0	0	•	•	•
[4]	1400	0	1000	0	0	0	0	•	•	•
[5]	800	0	0	0	0	0	0	•	•	•
[6]	600	0	0	1300	0	0	0	•	•	•
[7]	•	•	•	•	•	•	•	•	•	•
[8]	•	•	•	•	•	•	•	•	•	•
[9]	•	•	•	•	•	•	•	•	•	•

Adjacency Matrix

Distance (weight) from Dallas to Denver?

graph
.numVertices 7
.vertices

[0]	"Atlanta "
[1]	"Austin "
[2]	"Chicago "
[3]	"Dallas "
[4]	"Denver "
[5]	"Houston "
[6]	"Washington"
[7]	
[8]	
[9]	

.edges

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
[0]	0	0	0	0	0	800	600	•	•	•
[1]	0	0	0	200	0	160	0	•	•	•
[2]	0	0	0	0	1000	0	0	•	•	•
[3]	0	200	900	0	780	0	0	•	•	•
[4]	1400	0	1000	0	0	0	0	•	•	•
[5]	800	0	0	0	0	0	0	•	•	•
[6]	600	0	0	1300	0	0	0	•	•	•
[7]	•	•	•	•	•	•	•	•	•	•
[8]	•	•	•	•	•	•	•	•	•	•
[9]	•	•	•	•	•	•	•	•	•	•

Adjacency Matrix

graph
.numVertices 7
.vertices

Distance (weight) from Austin to Houston?

	.vertices
[0]	"Atlanta "
[1]	"Austin "
[2]	"Chicago "
[3]	"Dallas "
[4]	"Denver "
[5]	"Houston "
[6]	"Washington"
[7]	
[8]	
[9]	

.edges

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
[0]	0	0	0	0	0	800	600	•	•	•
[1]	0	0	0	200	0	160	0	•	•	•
[2]	0	0	0	0	1000	0	0	•	•	•
[3]	0	200	900	0	780	0	0	•	•	•
[4]	1400	0	1000	0	0	0	0	•	•	•
[5]	800	0	0	0	0	0	0	•	•	•
[6]	600	0	0	1300	0	0	0	•	•	•
[7]	•	•	•	•	•	•	•	•	•	•
[8]	•	•	•	•	•	•	•	•	•	•
[9]	•	•	•	•	•	•	•	•	•	•

37

Adjacency List

- An ***adjacency list*** for a graph with *n* vertices numbered 0,1,...,n-1 consists of *n* linked lists. The i^{th} linked list has a node for vertex *j* if and only if the graph contains an edge from vertex *i* to vertex *j*.
- Adjacency list is a better solution if the graph is sparse.
- Space requirement is O(|E| + |V|), which is linear in the size of the graph.
- In an undirected graph each edge (v,w) appears in two lists.
  - Space requirement is doubled.

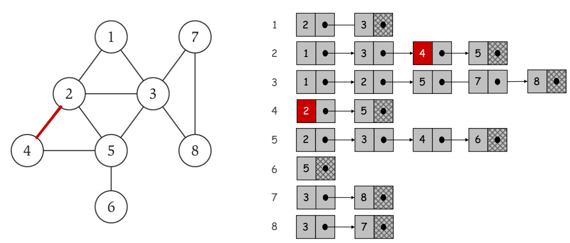
38

Graph Representation: Adjacency List

Adjacency list. Node indexed array of lists.

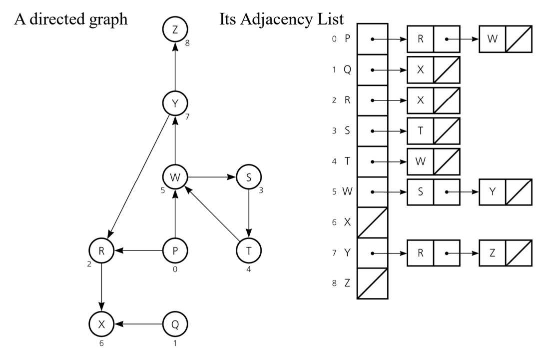
- Two representations of each edge.
- Checking if (u, v) is an edge takes O(degree(u)) time.

degree = number of neighbors of u



39

Directed Graph Adjacency List

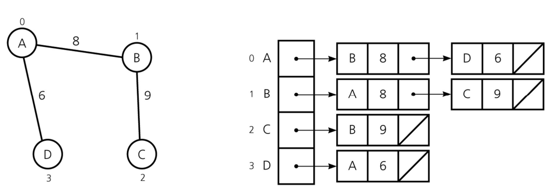


40

Weighted Graph: Adjacency List

An Undirected Weighted Graph

Its Adjacency List



41

Adjacency Matrix vs Adjacency List

- Two common graph operations:
  1. Determine whether there is an edge from vertex i to vertex j.
  - An adjacency matrix supports this operation more efficiently.

  2. Find all vertices adjacent to a given vertex i.
  - An adjacency list supports this operation more efficiently.
- An adjacency list often requires less space than an adjacency matrix.
  - Adjacency Matrix: Space requirement is $O(|V|^2)$
  - Adjacency List : Space requirement is O(|E| + |V|), which is linear in the size of the graph.
  - Adjacency matrix is better if the graph is dense (too many edges)
  - Adjacency list is better if the graph is sparse (few edges)

42

Graph Traversal

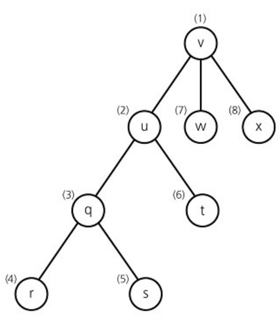
- A graph traversal algorithm systematically follows the edges of a graph to visit the vertices of the graph.
  - It can visit all vertices if and only if the graph is **connected**.
- A graph-traversal algorithm must mark each vertex during a visit and must never visit a vertex more than once.
  - Thus, if a graph contains a cycle, the graph-traversal algorithm can avoid infinite loop.
- Standard graph-traversal algorithms.
  - Depth-First Search **(DFS)**
  - Breadth-First Search **(BFS)**

Depth-First Search

"Search as deep as possible first."

- For a given vertex v, the ***depth-first search*** algorithm proceeds along a path from v as deeply into the graph as possible before backing up.
  - That is, after visiting a vertex v, the *depth-first search* algorithm visits (if possible) an unvisited adjacent vertex to vertex v.
- The depth-first search algorithm does not completely specify the order in which it should visit the vertices adjacent to v.
  - We may visit the vertices adjacent to v in sorted order.

Depth-First Search – Example



- A depth-first traversal of the graph starting from vertex v.
- Visit a vertex, then visit a vertex adjacent to that vertex.
- If there is no unvisited vertex adjacent to visited vertex, back up to the previous step.

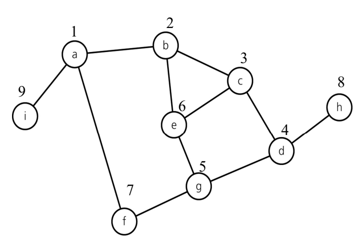
Recursive Depth-First Search Algorithm

```
dfs(v:Vertex) {

// Traverses a graph beginning at vertex v
// by using depth-first strategy
// Recursive Version

  Mark v as visited;
  for (each unvisited vertex u adjacent to v)
      dfs(u)
}
```


Iterative Depth-First Search Algorithm

```
dfs(v:Vertex) {
   // Traverses a graph beginning at vertex v
   // by using depth-first strategy: Iterative Version
   s.createStack();
   // push v into the stack and mark it
   s.push(v);
   Mark v as visited;
   while (!s.isEmpty()) {
      if (no unvisited vertices are adjacent to the vertex on
          the top of stack)
         s.pop();   // backtrack
      else {
         Select an unvisited vertex u adjacent to the vertex
            on the top of the stack;
         s.push(u);
         Mark u as visited;
      }
   }
}
```


Trace of Iterative DFS – starting from vertex a



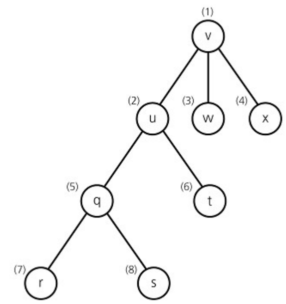
Node visited	Stack (bottom to top)
a	a
b	a b
c	a b c
d	a b c d
g	a b c d g
e	a b c d g e
(backtrack)	a b c d g
f	a b c d g f
(backtrack)	a b c d g
(backtrack)	a b c d
h	a b c d h
(backtrack)	a b c d
(backtrack)	a b c
(backtrack)	a b
(backtrack)	a
i	a i
(backtrack)	a
(backtrack)	*(empty)*

Breadth-First Search

- After visiting a given vertex v, the **breadth-first search algorithm** visits every vertex adjacent to v that it can before visiting any other vertex.
- The breadth-first search algorithm does not completely specify the order in which it should visit the vertices adjacent to v.
  – We may visit the vertices adjacent to v in sorted order.

49

Breadth-First Search – Example



• A breadth-first traversal of the graph starting from vertex v.

• Visit a vertex, then visit all vertices adjacent to that vertex.

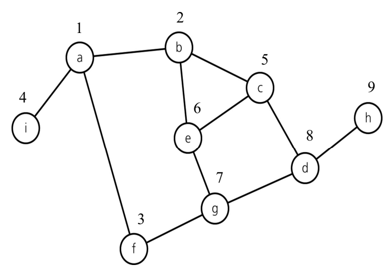
50

Iterative Breadth-First Search Algorithm

```
bfs(v:Vertex) {
   // Traverses a graph beginning at vertex v
   // by using breath-first strategy: Iterative Version
   q.createQueue();
   // add v to the queue and mark it
   q.enqueue(v);
   Mark v as visited;
   while (!q.isEmpty()) {
      q.dequeue(w);
      for (each unvisited vertex u adjacent to w) {
         Mark u as visited;
         q.enqueue(u);
      }
   }
}
```

51

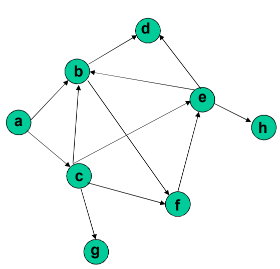
Trace of Iterative BFS – starting from vertex a



Node visited	Queue (front to back)
a	a
	(empty)
b	b
f	b f
i	b f i
	f i
c	f i c
e	f i c e
	i c e
g	i c e g
	c e g
	e g
d	e g d
	g d
	d
	(empty)
h	h
	(empty)

52

Exercise



a) Give the sequence of vertices when they are traversed starting from the vertex **a** using the **depth-first search** algorithm.

b) Give the sequence of vertices when they are traversed starting from the vertex **a** using the **breadth-first search** algorithm.

53

Graph Implementation (Adj. Lists)

```
struct AdjListNode
{
    int v;
    AdjListNode* next;
    AdjListNode(int x, AdjListNode* t)
    {
        v = x;
        next = t;
    }
};

typedef AdjListNode* link;

struct Edge
{
    int v, w;
    Edge(int v = -1, int w = -1) : v(v), w(w) { }
};
```

54

Graph Class

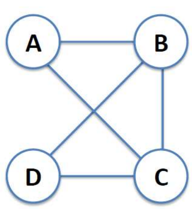
```
class Graph{
public:
    Graph(int V, bool digraph=false);
    int getV() const { return Vcnt; }
    int getE() const { return Ecnt; }
    bool directed() const { return digraph; }
    ~Graph();
    void insert(Edge e);
    void remove (Edge e);
    bool isEdge(int v, int w);
    bool isVertex(int v);
    vector<int> adjNodesOf(int v);
    int degreeOf(int v);
    void print();

private:
    int Vcnt;          // number of nodes in the graph
    int Ecnt;          // number of edges
    bool digraph;      // is it directed?
    vector<link> adj;  // An array of pointers to Node to represent adjacency list
    vector<int> distance;
    vector<bool> mark;
    vector<int> previous;
};
```


Some Graph Algorithms

- Shortest Path Algorithms
  - Unweighted shortest paths
  - Weighted shortest paths (Dijkstra's Algorithm)
- Topological sorting
- Minimum Spanning Tree
- Network Flow Problems
- …

Recap: Graphs

- A **graph** G = (V, E) consists of
  - a set of ***vertices (nodes)***, V, and
  - a set of ***edges***, E, where each edge is a pair (v,w) such that v,w ∈ V



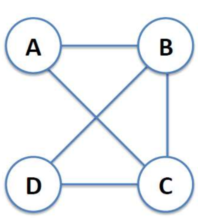

```
V = {A, B, C, D}
E = {(A,B),(A,C),(B,C),
     (B,D),(C,D)}
```


Recap: Graph Representations

- The two most common implementations of a graph are:
  - *Adjacency Matrix*
    - A two dimensional array
  - *Adjacency List*
    - For each vertex we keep a list of adjacent vertices

Recap: Adjacency Matrix

- The adjacency matrix captures all the edge information. From it, you can calculate, for example:
  - The number of neighbors each vertex has.
  - Is there a path between two vertices? If so, what's the path length?
  - Is this a connected graph?



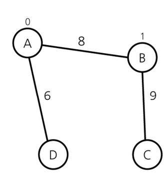
to

from

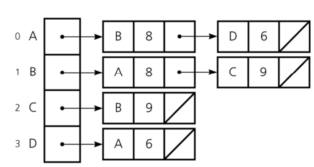
	A	*B*	*C*	*D*
A	0	1	1	0
B	1	0	1	1
C	1	1	0	1
D	0	1	1	0

Recap: Adjacency List

An Undirected Weighted Graph



Its Adjacency List



Recap: Graph Search Traversal

- Graph search / traversal is a fundamental operation:
  - Is there a path from vertex X to vertex Y? If so, what's the shortest path?
  - Is this a connected graph? If not, how many connected sub-graphs are there?
  - As you will see later, many interesting problems can be formulated as graph search problem
- For **binary trees**, we learned three types of traversals: in-order, pre-order, post-order.
- For **graphs**, generally two types: **DFS, BFS**

61

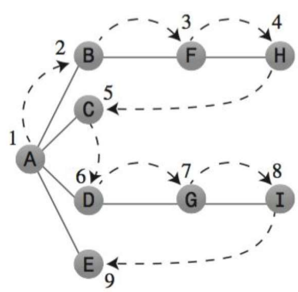
Recap: DFS and BFS

- **Depth-First Search**: start at a vertex, follows its edges to visit the deepest point, then moves up.
  - Go as far away from the starting vertex as possible (depth), before moving back.
  - Use a **Stack** to track where to go next.
- **Breadth-First Search**: traverse vertices in 'levels' — starting from a vertex, visit all its immediate neighbors, then neighbors of neighbors, and so on.
  - Stay as close to the starting vertex as possible (breadth), before moving to the next level.
  - Use a **Queue**.

62

Depth-First Search - Example

- Start from vertex A. Visit all vertices connected to A using DFS.



63

Finding a Path using DFS

```
// DFS from start vertex to end vertex
bool hasPath(int start, int end) {
   Stack s;
   // push start into the stack and mark it
   s.push(start);
   mark[start] = true;
   int b = -1;
   while (!s.isEmpty()) {
      b = get_next_univisited_neighbor(s.peek());
      if (b == end) break;
      if (b == -1) //no unvisited neighbor
         s.pop();  // backtrack
      else {
         mark[b] = true;
         s.push(b);
      }
   }
   if (b == end) return true;
   else return false;
}
```

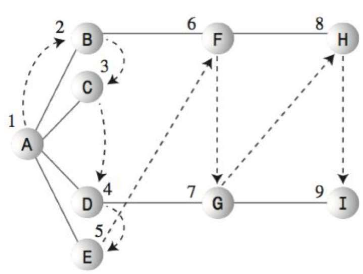
So how do we print out the path??

Vertices on the path are all stored in the stack!

64

Breadth-First Search - Example

Start from vertex A. Visit all vertices connected to A (i.e. A's neighbors) using BFS.



65

Finding a path using BFS

```
bool has_path_BFS(int start, int end){
  Queue q;
  q.enqueue(start)
  mark[start] = true;
  int b = -1;

  while (!q.isEmpty()){
      b= q.dequeue();
      if (b == end) break
      for each unvisited v adjacent to b{
         mark[v] = true;
         previous[v] = b;
         q.enqueue(v);
      }
  }
  if (b == end) return true;
  else return false;
}
```

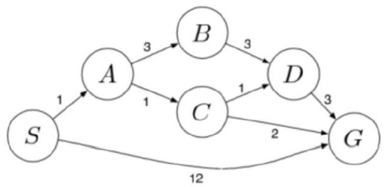
So how do we extract the path??

Vertices on the path are all stored in the array previous!

66

Search in Weighted Graphs

- In weighted graphs, we generally care about <u>the shortest path</u> from vertex X to Y in terms of the path weight (sum of weights on the path).
- Although we can still perform DFS and BFS in weighted graphs, they are often not so useful as they don't account for edge weight.
- For example, perform **BFS** on the following graph to find a path from S to G, what would you get? Is it the shortest path?



67

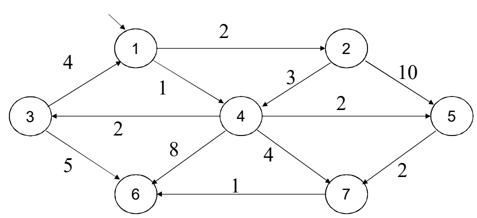
Search in Weighted Graphs

- BFS gives you <u>a path with minimum number of edges</u> (in terms of number of segments), but not necessarily the shortest path (in terms of total path weight).
- Analogy: BFS in a air flight graph can give you an itinerary with minimum number of flight segments, but not necessarily the minimum total price!
- **Shortest-Path Algorithm**: turns out we can modify BFS slightly to implement shortest-path algorithm.
- The trick here is to use a Priority Queue instead of the standard FIFO Queue.
- Shortest-path algorithm is a rich topic. You will learn more in upper-level classes.

68

Single-source shortest-path problem

- *Find the shortest path (measured by total cost) from a designated vertex S to every vertex. All edge costs are nonnegative.*



69

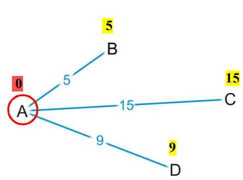
Dijkstra's algorithm

- The algorithm proceeds in stages.
- At each stage, the algorithm
  - selects a vertex v, which has the smallest distance D_v among all the *unknown* vertices, and
  - declares that the shortest path from s to v is *known*.
  - then for the adjacent nodes of v (which are denoted as w) D_w is updated with new distance information
- How do we change D_w?
  - If its current value is larger than $D_v + c_{v,w}$ we change it.

70

Strategy

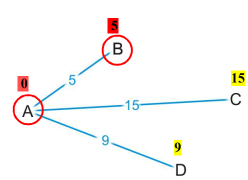
Suppose you are at vertex A
- Find the shortest distance to adjacent nodes.



71

Strategy

We accept that (A, B) is the shortest path to vertex B from A
- Let's see where we can go from B

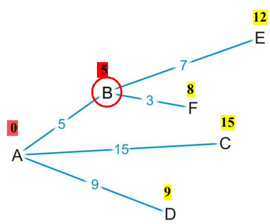


72

Strategy

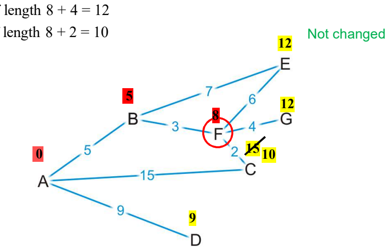
By some simple arithmetic, we can determine that

- There is a path (A, B, E) of length 5 + 7 = 12
- There is a path (A, B, F) of length 5 + 3 = 8



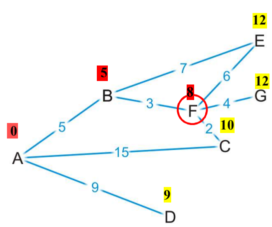
Strategy

- Now, let's visit vertex F
  - We know the shortest path is (A, B, F) and it's of length 8
- There are three edges exiting vertex F, so we have paths:
  - (A, B, F, E) of length 8 + 6 = 14
  - (A, B, F, G) of length 8 + 4 = 12
  - (A, B, F, C) of length 8 + 2 = 10



Strategy

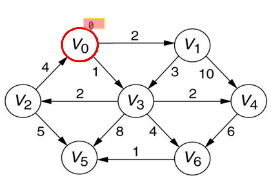
- At this point, we have the shortest distances to B and F
  - Which remaining vertex are we currently guaranteed to have the shortest distance to?



Dijkstra's Algorithm

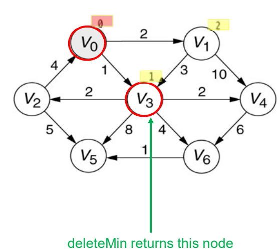
```
PriorityQueue<Vertex> pq;
for each node v {
    distance[v] = ∞; previous[v] = null; }
distance[s] = 0;
add all vertices to pq;
while(!pq.isEmpty()) {
      Vertex v = pq.deleteMin();
      for each edge(v,w)
          new_dist = distance[v] + weight(v,w);
          if (new_dist < distance[w]) {
             update distance[w] = new_dist;
             previous[w] = v;
             pq.decrease_priority(w,new_dist);
          }
       }
}
```


Example



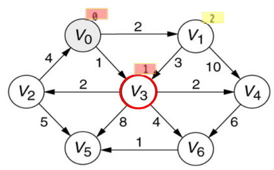
Vertex	Distance	Previous
v_0	0	Ø
v_1	∞	Ø
v_2	∞	Ø
v_3	∞	Ø
v_4	∞	Ø
v_5	∞	Ø
v_6	∞	Ø

Example



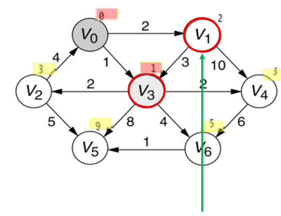
Vertex	Distance	Previous
v_0	0	Ø
v_1	2	v_0
v_2	∞	Ø
v_3	1	v_0
v_4	∞	Ø
v_5	∞	Ø
v_6	∞	Ø

Example



Vertex	Distance	Previous
v_0	0	Ø
v_1	2	v_0
v_2	∞	Ø
v_3	1	v_0
v_4	∞	Ø
v_5	∞	Ø
v_6	∞	Ø

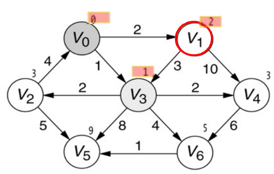
Example



deleteMin returns this node

Vertex	Distance	Previous
v_0	0	Ø
v_1	2	v_0
v_2	3	v_3
v_3	1	v_0
v_4	3	v_3
v_5	9	v_3
v_6	5	v_3

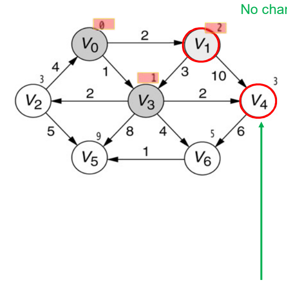
Example



Vertex	Distance	Previous
v_0	0	Ø
v_1	2	v_0
v_2	3	v_3
v_3	1	v_0
v_4	3	v_3
v_5	9	v_3
v_6	5	v_3

Example

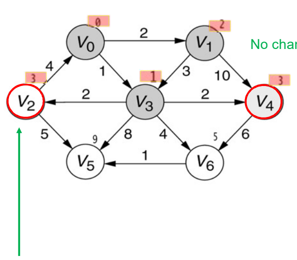
No change



deleteMin returns this node

Vertex	Distance	Previous
v_0	0	Ø
v_1	2	v_0
v_2	3	v_3
v_3	1	v_0
v_4	3	v_3
v_5	9	v_3
v_6	5	v_3

Example

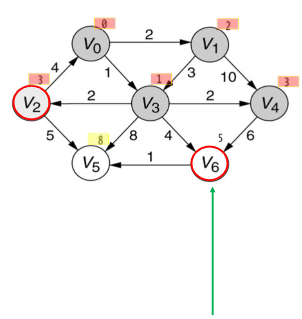


No change

deleteMin returns this node

Vertex	Distance	Previous
v_0	0	Ø
v_1	2	v_0
v_2	3	v_3
v_3	1	v_0
v_4	3	v_3
v_5	9	v_3
v_6	5	v_3

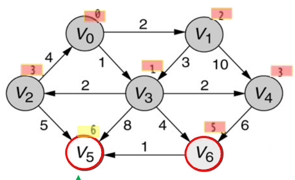
Example



deleteMin returns this node

Vertex	Distance	Previous
v_0	0	Ø
v_1	2	v_0
v_2	3	v_3
v_3	1	v_0
v_4	3	v_3
v_5	~~9~~ 8	v_2
v_6	5	v_3

Example

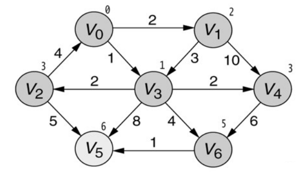


Vertex	Distance	Previous
v_0	0	Ø
v_1	2	v_0
v_2	3	v_3
v_3	1	v_0
v_4	3	v_3
v_5	~~8~~ 6	v_6
v_6	5	v_3

No change

deleteMin returns this node

Example



Vertex	Distance	Previous
v_0	0	Ø
v_1	2	v_0
v_2	3	v_3
v_3	1	v_0
v_4	3	v_3
v_5	6	v_6
v_6	5	v_3

Example

Note that this table can be used to generate the paths.

Vertex	Previous
v_0	Ø
v_1	v_0
v_2	v_3
v_3	v_0
v_4	v_3
v_5	v_6
v_6	v_3

Shortest Paths: Dijkstra's Algorithm

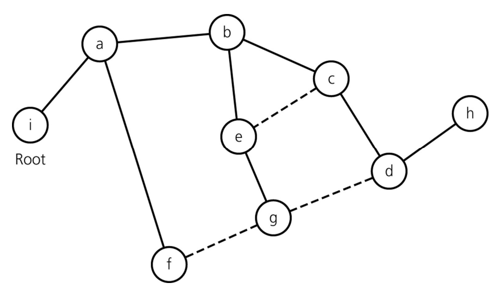
Analysis:

- With priority queue: $O(|E|\log|V|+|V|\log|V|) \rightarrow O(|E|\log|V|)$
- Without priority queue: $O(|E|+|V|^2) \rightarrow O(|V|^2)$

Spanning Trees

- A ***spanning tree*** of a connected undirected graph G is a sub-graph of G that contains all of G's vertices and enough of its edges to form a tree.
- There may be several spanning trees for a given graph.
- If we have a connected undirected graph with cycles, and we remove edges until there are no cycles, we obtain a spanning tree.

A Spanning Tree

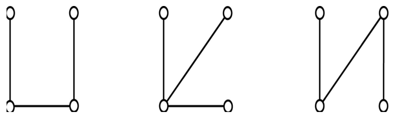


Remove dashed lines to obtain a spanning tree

Cycles?

Observations about graphs:

1. A connected undirected graph that has n vertices must have at least n-1 edges.
2. A connected undirected graph that has n vertices and exactly n-1 edges cannot contain a cycle.
3. A connected undirected graph that has n vertices and more than n-1 edges must contain a cycle.

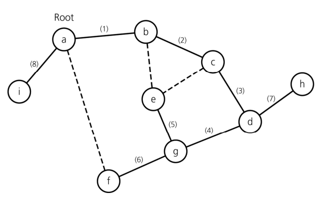


Connected graphs that each have four vertices and three edges

DFS Spanning Tree

```
dfsTree(in v:vertex) {
// Forms a spanning tree for a connected undirected graph
// beginning at vertex v by using depth-first search;
// Recursive Version
  Mark v as visited;
  for (each unvisited vertex u adjacent to v) {
     Mark the edge from u to v;
     dfsTree(u);
  }
}
```


DFS Spanning Tree – Example



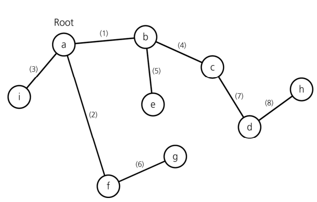
The DFS spanning tree algorithm visits vertices in this order: a, b, c, d, g, e, f, h, i. Numbers indicate the order in which the algorithm marks edges.

The DFS spanning tree rooted at vertex *a*

BFS Spanning tree

```
bfsTree(in v:vertex) {
// Forms a spanning tree for a connected undirected graph
// beginning at vertex v by using breath-first search;
// Iterative Version
   q.createQueue();
   q.enqueue(v);
   Mark v as visited;
   while (!q.isEmpty()) {
      q.dequeue(w);
      for (each unvisited vertex u adjacent to w) {
         Mark u as visited;
         Mark edge between w and u;
         q.enqueue(u);
      }
   }
}
```


BFS Spanning tree – Example



The BFS spanning tree algorithm visits vertices in this order: a, b, f, i, c, e, g, d, h. Numbers indicate the order in which the algorithm marks edges.

The BFS spanning tree rooted at vertex *a*

Minimum Spanning Tree

- For a weighted undirected graph, the *cost of a spanning tree* is the sum of the costs of the edges in the spanning tree.
- A ***minimum spanning tree*** of a connected undirected graph has a minimal edge-weight sum.
  - A minimum spanning tree of a connected undirected may not be unique.
  - Although there may be several minimum spanning trees for a particular graph, their costs are equal.

Prim’s algorithm

Prim’s Algorithm finds a minimum spanning tree that begins at any vertex.

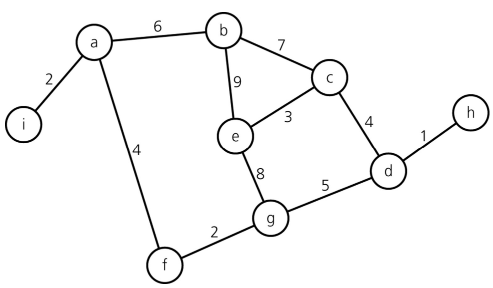
- Initially, the tree contains only the starting vertex.
- At each stage, the algorithm selects a least-cost edge from among those that begin with a vertex in the tree and end with a vertex not in the tree.
- The selected vertex and least-cost edge are added to the tree.

Prim’s Algorithm

```
primsAlgorithm(in v:Vertex) {
// Determines a minimum spanning tree for a weighted,
// connected, undirected graph whose weights are
// nonnegative, beginning with any vertex.
  Mark vertex v as visited and include it in
     the minimum spanning tree;
  while (there are unvisited vertices) {
     Find the least-cost edge (v,u) from a visited vertex v
        to some unvisited vertex u;
     Mark u as visited;
     Add the vertex u and the edge (v,u) to the minimum
        spanning tree;
  }
}
```

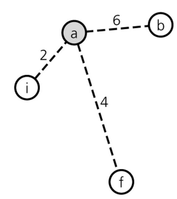

Prim’s Algorithm – Trace

A weighted, connected, undirected graph

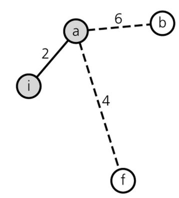


Prim’s Algorithm – Trace (cont.)

beginning at vertex *a*

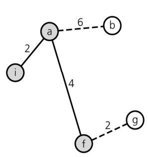


(a) Mark a, consider edges from a

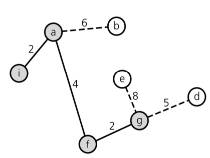


(b) Mark i, include edge (a, i)

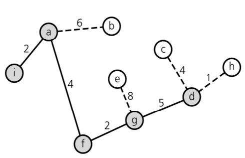
Prim’s Algorithm – Trace (cont.)



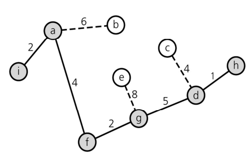
(c) Mark f, include edge (a, f)



(d) Mark g, include edge (f, g)

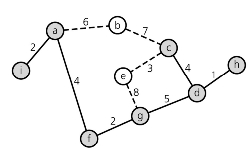


(e) Mark d, include edge (g, d)

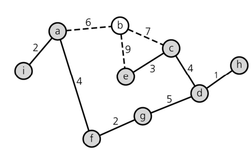


(f) Mark h, include edge (d, h)

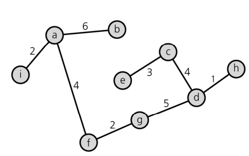
Prim’s Algorithm – Trace (cont.)



(g) Mark c, include edge (d, c)



(h) Mark e, include edge (c, e)



(i) Mark b, include edge (a, b)

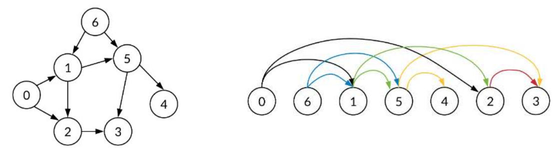
Topological Sorting

- A directed acyclic graph (DAG) has a natural order.
  - That is, vertex a precedes vertex b, which precedes c
  - For example, the prerequisite structure for the courses.
- In which order we should visit the vertices of a directed graph without cycles so that we can visit vertex v after we visit its predecessors.
  - This is a linear order, and it is known as ***topological order***.
- For a given graph, there may be more than one topological order.
- Arranging the vertices into a topological order is called ***topological sorting***.

Example

- Ordering a sequence of tasks given their dependencies on other tasks. There is an directed edge from *u* to *v* if task *u* must be completed before task *v* can start.
  - For example, when cooking, we need to turn on the oven (task *u*) before we can bake the cookies (task *v*).

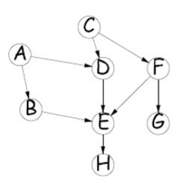
Given Directed Acyclic Graph (DAG): **Topologically sorted graph:**



- Other examples: gmake, PERT charts, etc.

Topological Sorting

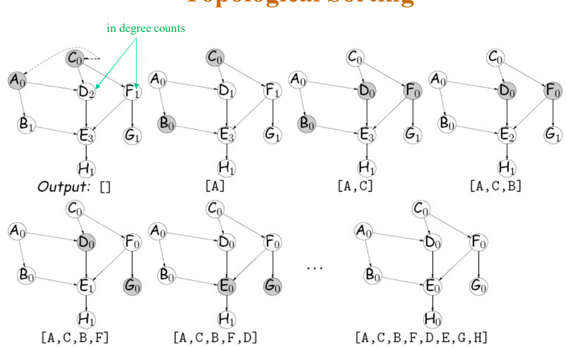
- **Problem:** Given a DAG, find a linear order of nodes consistent with the edges.
  - That is, order the nodes $v_1, v_2, \ldots, v_n$ such that v_j is never reachable from v_i if i >j.
- There can be more than one topological ordering of a graph.



[A,B,C,F,D,G,E,H], or
[A,C,B,D,F,E,G,H], or
[A,B,C,F,D,E,H,G], or
⋮


```
void Graph::topsort( )
{
    Queue<Vertex> q;
    int counter = 0;

    q.makeEmpty( );
    for each Vertex v
        if( v.indegree == 0 )
            q.enqueue( v );

    while( !q.isEmpty( ) )
    {
        Vertex v = q.dequeue( );
        v.topNum = ++counter;  // Assign next number

        for each Vertex w adjacent to v
            if( --w.indegree == 0 )
                q.enqueue( w );
    }

    if( counter != NUM_VERTICES )
        throw CycleFoundException( );
}
```


Topological Sorting



Analysis of Topological Sorting

The time to perform this algorithm is O(|E|+|V|) if adjacency lists are used:

- The body of the for loop is executed at most once per edge.
- The queue operations are done at most once per vertex.
- Initialization steps also take time proportional to the size of the graph.