DATA STRUCTURES & ALGORITHMS

Lecture 7: GRAPHS – part 1

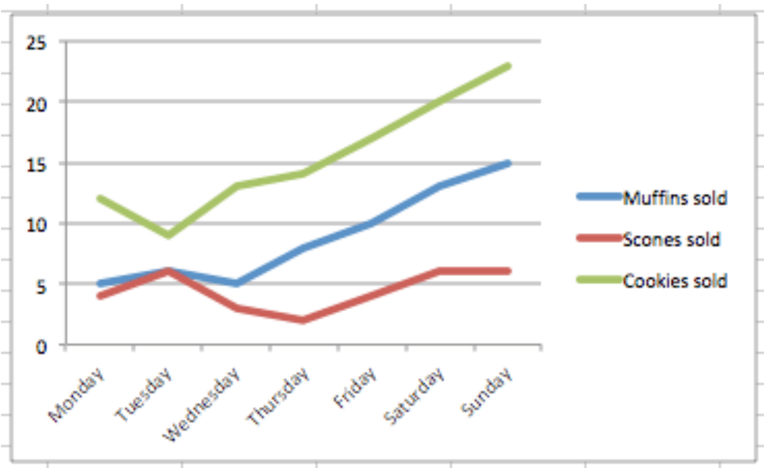
Lecturer: Dr. Nguyen Hai Minh

OUTLINE

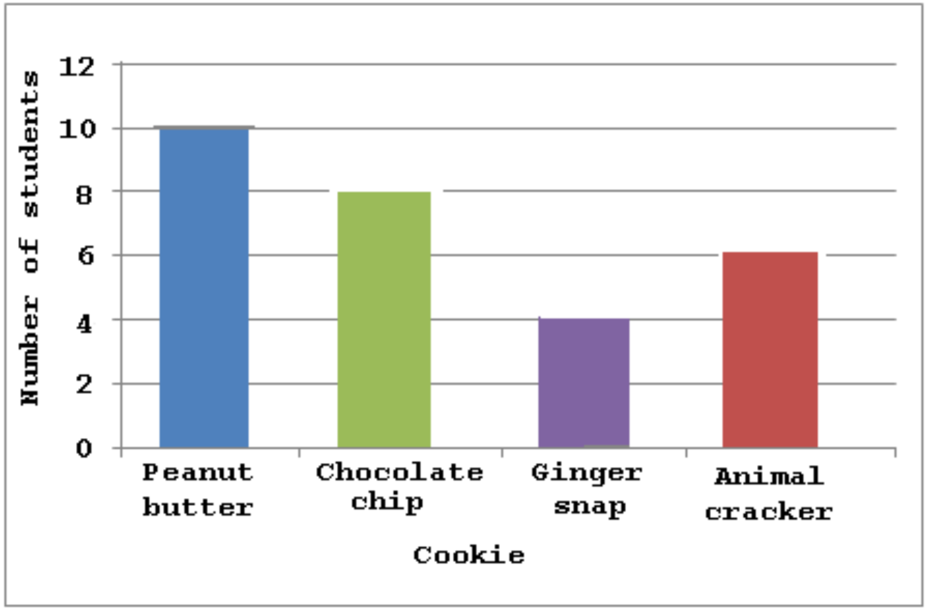
- Introduction
- Connectivity
- Graphs as ADTs
- Implementing Graphs
- Graph Traversals
- Application of Graphs
  - Topological Sorting
  - Minimum Spanning Tree

Introduction

- Common graphs that you are familiar with:

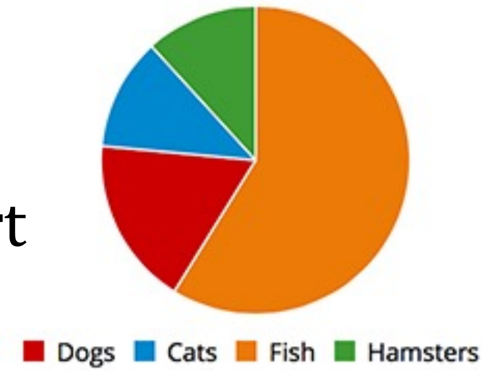


Line graph



Bar graph

Pie chart



Introduction

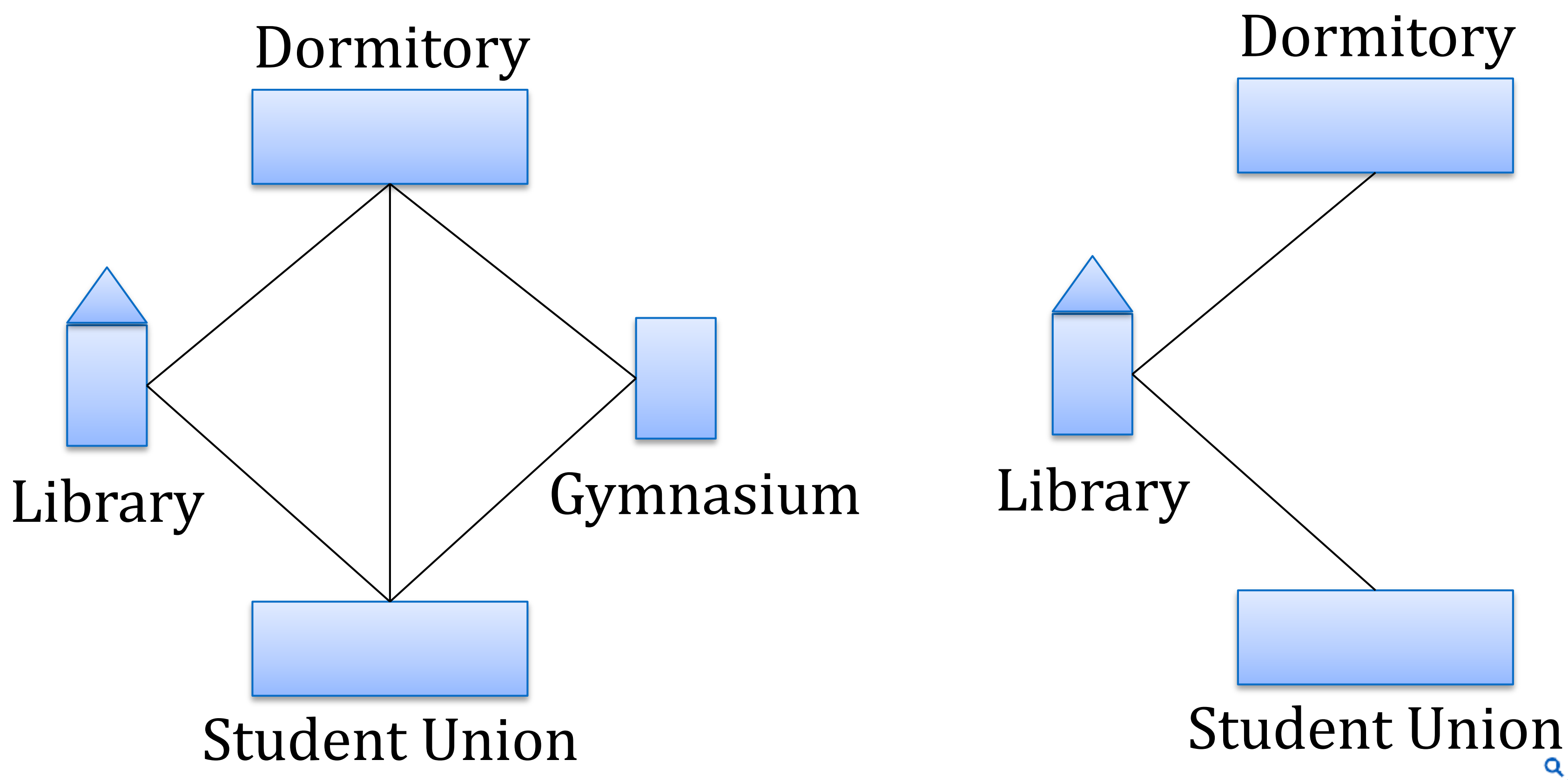
- Graphs are used to:
  - Provide a way to illustrate data
  - Represent the relationships among data items
- Graphs in computer science: consist of 2 **sets**:

$$\boldsymbol{G} = (V, E)$$

  - *V*: vertices (or nodes)
  - *E*: edges (connect the vectices)

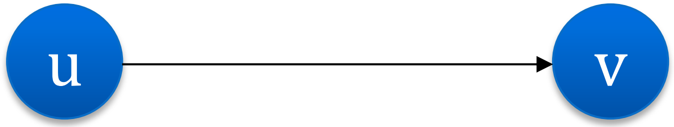
Graph – Example

- Vertices: buildings
- Edges: sidewalks between building

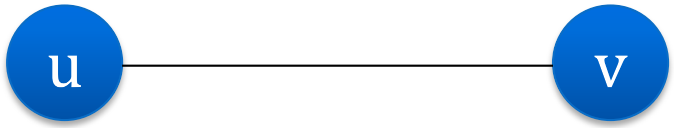


Definitions – Edge Type

- Directed:
  - Ordered pair of vertices



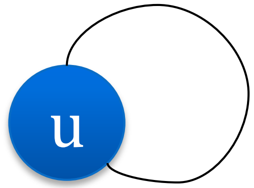
  - Represented as (u, v) directed from vertex u to v
- Undirected:
  - Unordered pair of vertices.



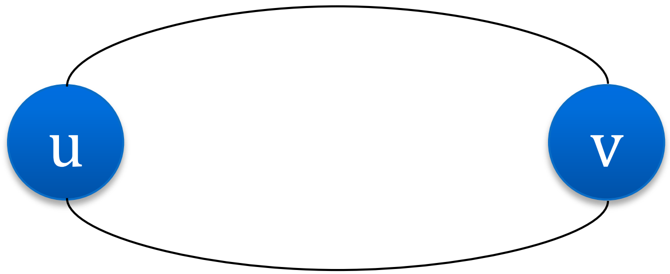
  - Represented as $\{u, v\}$

Definitions – Edge Type

- Loop/Self Edge:
  - Edge whoses endpoints are equal.
  - Represented as $\{u, u\} = \{u\}$



- Multiple Edges
  - 2 or more edges joining the same pair of vertices



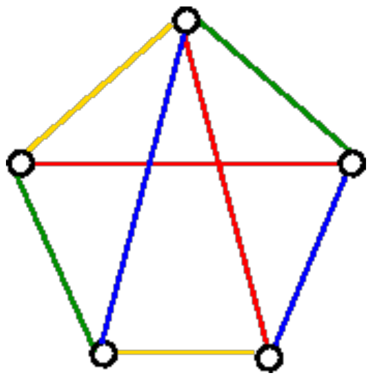
Definitions – Graph Types

- 4 graph types

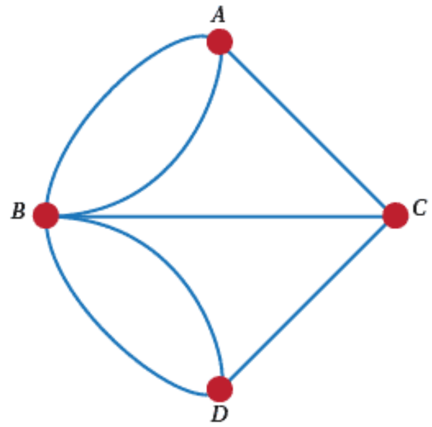
Type	Edges	Multiple edges allowed	Loops allowed?
Simple Graph /Undirected Graph	U	No	No
Multigraph	U	Yes	No/Yes
Directed Graph	D	No	No/Yes
Directed Multigraph	D	Yes	Yes

*U : undirected, D: directed

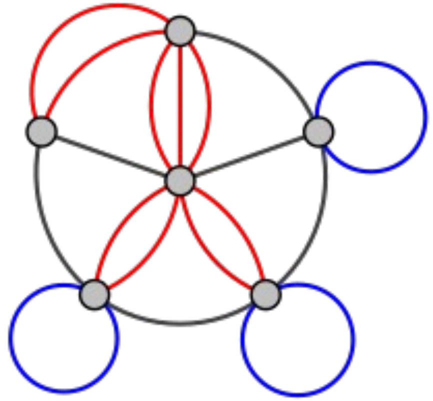
Definitions – Graph Types



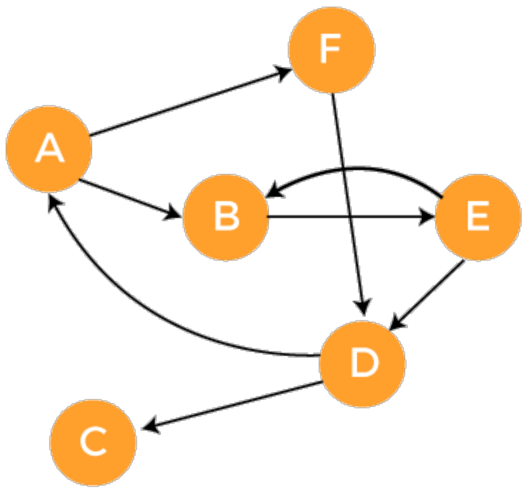
(a) Simple Graph



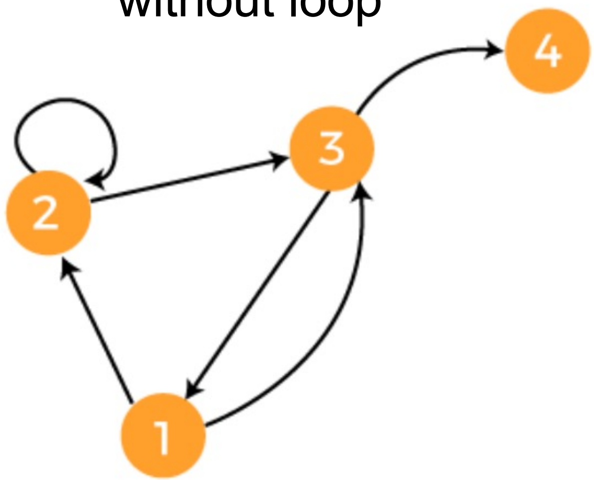
(b) Multigraph without loop



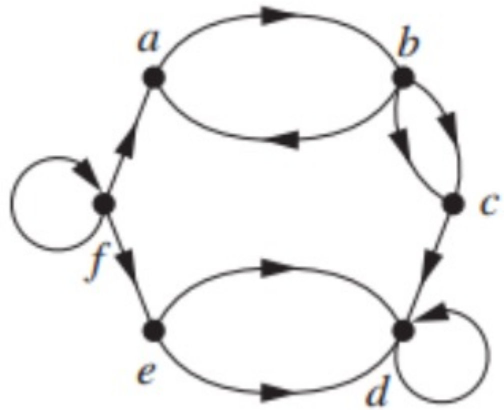
(c) Multigraph with loop



(d) Directed Graph Without loop



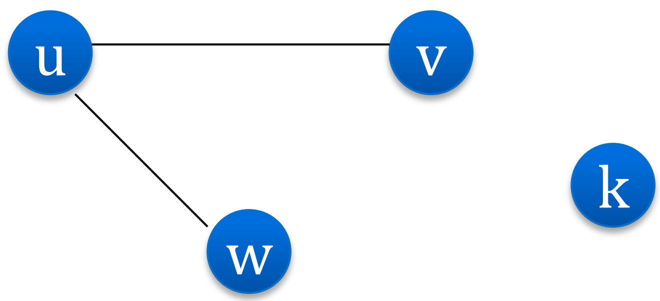
(e) Directed Graph with loop



(f) Directed Multigraph

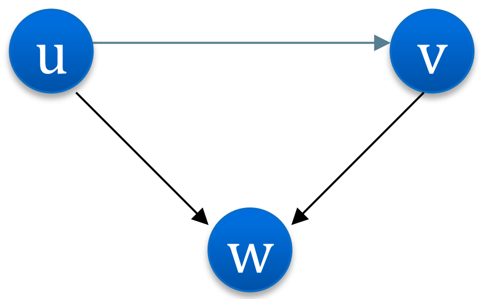
Terminology – Undirected Graphs

- Adjacent vertices: 2 vertices u, v are adjacent if they are joined by an edge $e=\{u, v\}$
- Degree of vertex: $\deg(v)$ = number of edges incident to the vertex. A loop contributes twice to the edge
  - Pendant Vertex: $\deg(v) = 1$
  - Isolated Vertex: $\deg(v) = 0$
- Example: $V = \{u, v, w, k\}$, $E=\{ \{u, w\}, \{u, v\} \}$
  - $\deg(u) = 2$
  - $\deg(v) = \deg(w) = 1$
  - $\deg(k) = 0$



Terminology – Directed Graphs

- **Adjacent vertices**: for the edge (u, v): u is **adjacent to** v or v is **adjacent from** u
  - u: Initial vertex
  - v: Terminal vertex
- **Degree of vertex**:
  - In-degree: $deg^{-}(u)$ = #edges for which u is terminal vertex
  - Out-degree: $deg^{+}(u)$ = #edges for which u is initial vertex
- Example: $V = \{u, v, w\}$, $E = \{ (u, w), (v, w), (u, v) \}$
  - $deg^{-}(u) = 0$, $deg^{+}(u) = 2$
  - $deg^{-}(v) = 1$, $deg^{+}(v) = 1$
  - $deg^{-}(w) = 2$, $deg^{+}(w) = 0$



Theorems

- Theorem 1 – ***The Handshaking theorem*** in undirected graph

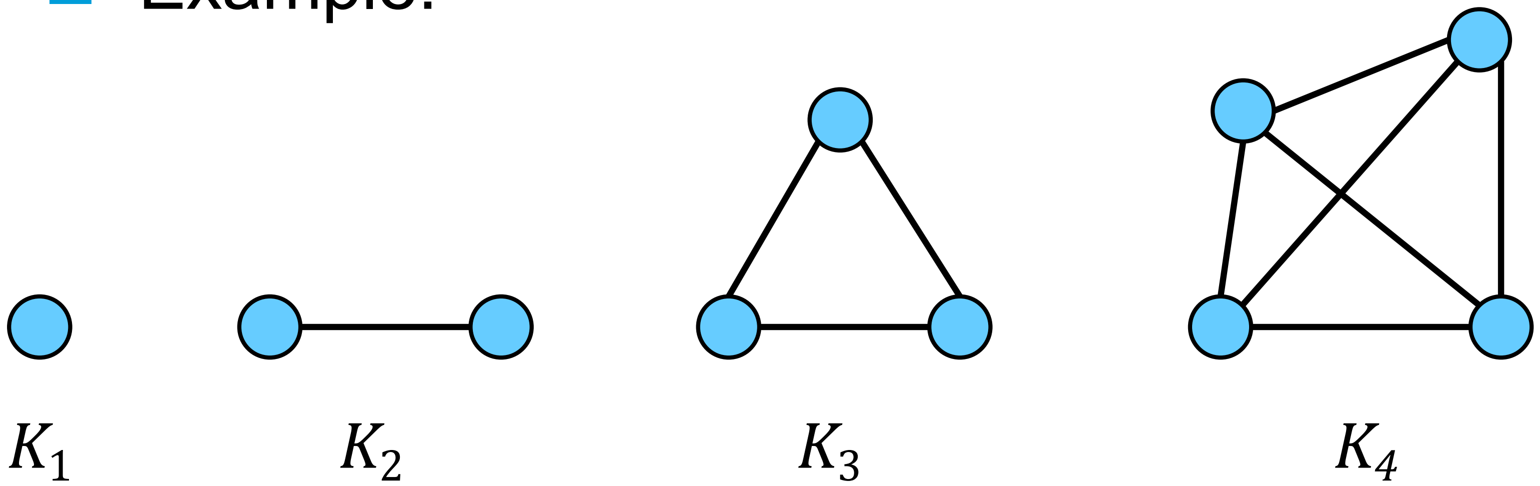
$$\sum \deg v = 2|E|$$

- Theorem 2: An undirected graph has even number of vertices with odd degree
- Theorem 3: for directed graph

$$\sum \deg^{+} u = \sum \deg^{-} u \; = |E|$$

Simple Graphs – Types

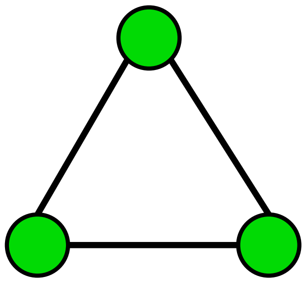
- Complete graph: K_n
  - Simple graph that contains exactly <u>one edge</u> between each pair of distinct vertices
  - Example:



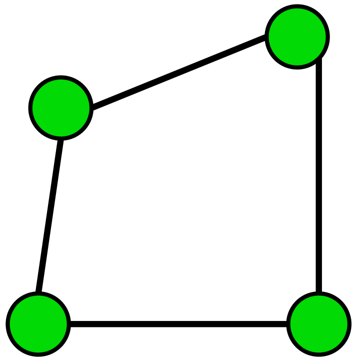
→ How many edges does K_n have?

Simple Graphs – Types

- Cycle: C_n, $n > 2$
  - Simple graph that consists of n vertices $v_1, v_2, \ldots, v_n$ and edges $\{v_1, v_2\}, \{v_2, v_3\}, \ldots, \{v_{n-1}, v_n\}, \{v_n, v_1\}$
  - Example



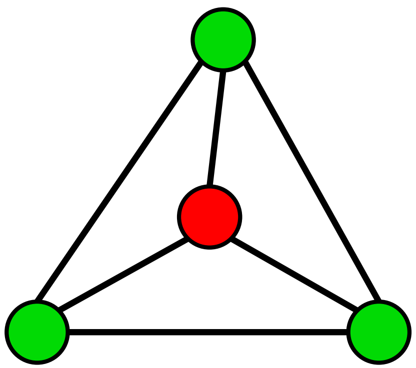
C_3



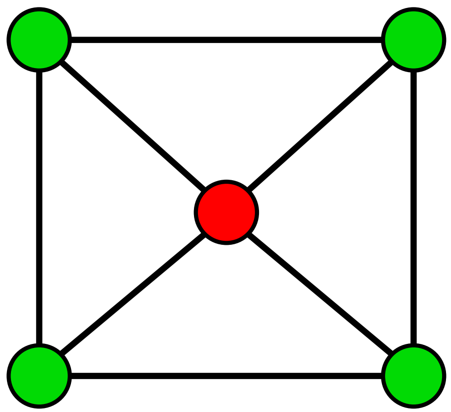
C_4

Simple Graphs – Types

- Wheel: W_n
  - Obtained by adding additional vertex to C_n and connecting all vertices to this new vertex by new edges.
  - Example



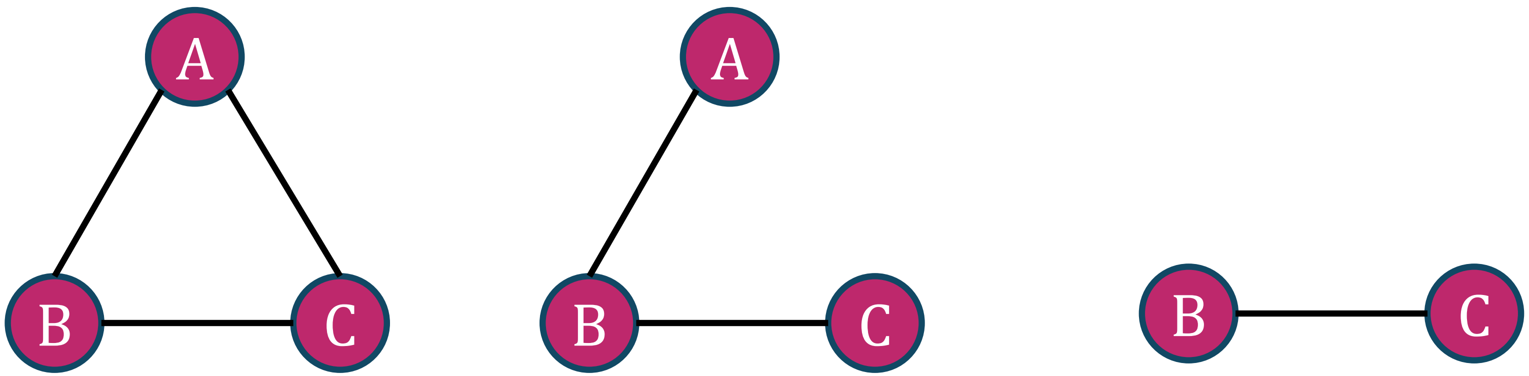
W_3



W_4

Subgraphs

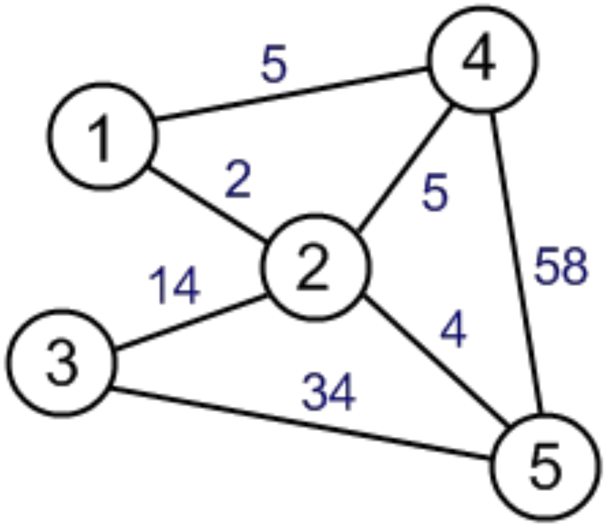
- A subgraph of a graph $G = (V, E)$ is a graph $H = (V', E')$ where V' is a subset of V and E' is a subset of E
- Example:



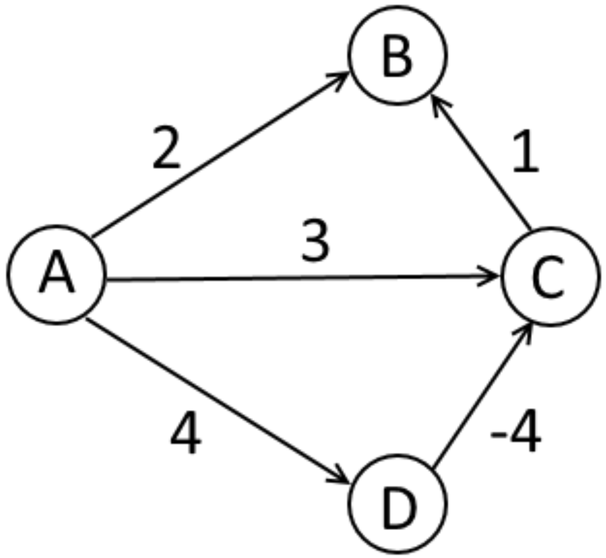
G H_1 H_2

Weighted Graph

- The edges of a weighted graph have numeric labels.
- Example:



(a) Undirected Weighted Graph



(b) Directed Weighted Graph

GRAPH CONNECTIVITY

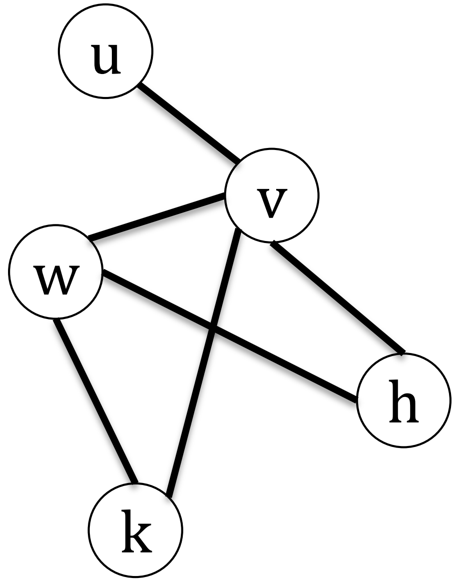
Connectivity

- Basic Idea: Is the Graph Reachable among vertices by traversing the edges?

- Example:
  - Can Japan be reached from Vietnam?

Connectivity – Path

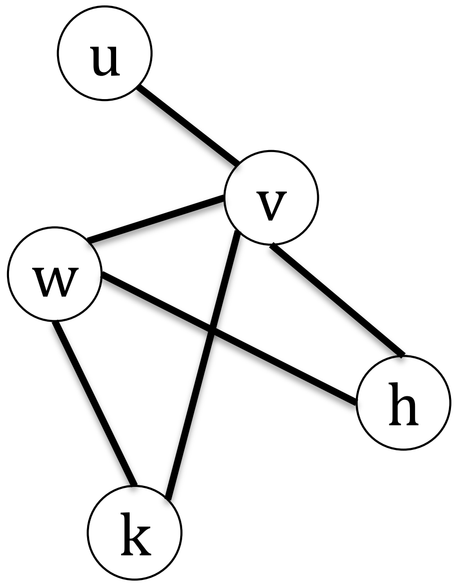
- Path: sequence of edges that begins at one vertex and ends at another vertex.
  - Example: G = (V, E)
  - Path P = { {u, v}, {v, w}, {w, h} }
- Cycle/Circuit: start vertex = end vertex
  - Cycle C = { {v, w}, {w, h}, {h, v} }



- Simple path: a path that does not pass through the same vertex more than once.

Connectivity – Connectedness

- Undirected Graph:
  - An undirected graph is **connected** if there exists a simple path between every pair of vertices
- Example: G = (V, E) is connect since for V = {u, v, w, k, h}, there exists a path between each pair of vertices



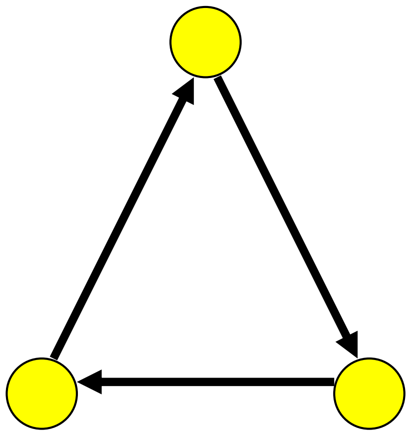
Connectivity – Connectedness

- Directed Graph:
  - A directed graph is **strongly connected** if there is a path from *a* to *b* and from *b* to *a* whenever *a* and *b* are vertices in the graph
  - A directed graph is **weakly connected** if there is a (undirected) path between every two vertices in the underlying undirected path

→ *A strongly connected graph can be weakly connected but the vice-versa is not true (why?)*

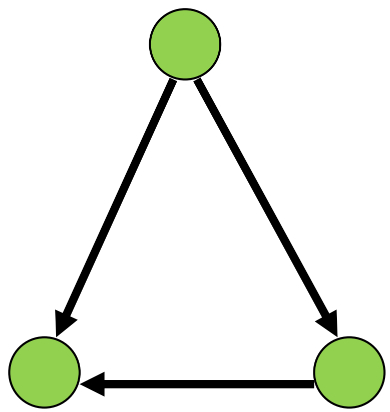
Connectivity – Connectedness

☐ Directed Graph: Example



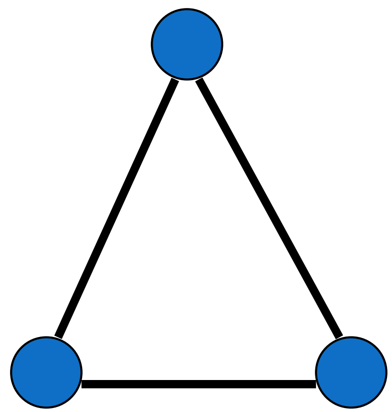
G_1

Strongly connected



G_2

Weakly connected



G_3

Undirected graph representation of G_1 or G_2

GRAPHS AS ADTS

Graphs as ADTs

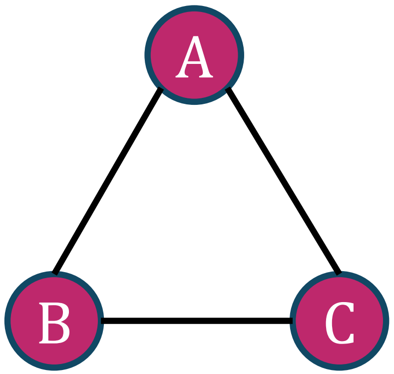
- Vertices contain values
- Edges represent relationship between vertices
- Operations:
  1. Test whether a graph is empty
  2. Get the number of vertices/edges in a graph
  3. See whether an edge exists between 2 given vertices
  4. Insert a vertex/an edge in a graph
  5. Remove a vertex/edge in a graph
  6. Search the graph for the vertex that contains a given value

Implementing Graphs – Adjacency Matrix

- Adjacency matrix of a graph with *N* vertices: an *NxN* array $A = [a_{ij}]$ such that

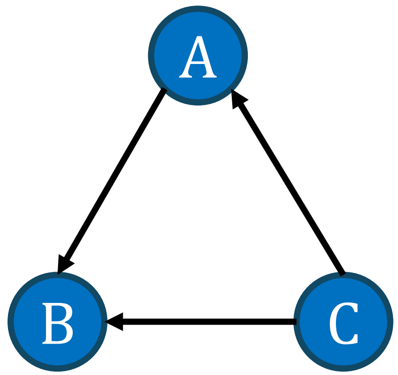
$$a_{ij} = \begin{cases} 1 & \text{if there is an edge between vertex i and j} \\ 0 & \text{otherwise} \end{cases}$$

- Example: Adjacency matrix of undirected & directed graphs



G_1

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



G_2

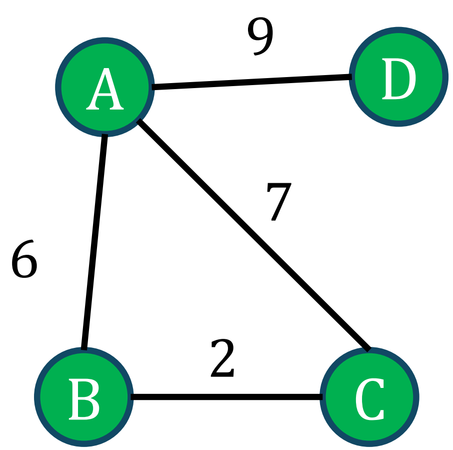
	A	B	C
A	0	1	0
B	0	0	0
C	1	1	0

Implementing Graphs – Adjacency Matrix

- When the graph is weighted,

$$a_{ij} = \begin{cases} weight & \text{of the edge between vertex i and j} \\ \infty & \text{otherwise} \end{cases}$$

- Example:

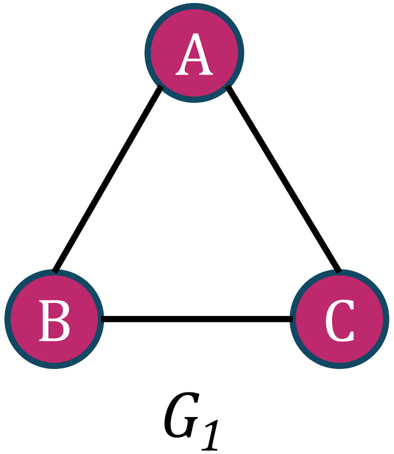


G_3

	A	B	C	D
A	∞	6	7	9
B	6	∞	2	∞
C	7	2	∞	∞
D	9	∞	∞	∞

Implementing Graphs – Adjacency List

- Each node (vertex) has a list of which nodes it is adjacent.
- Example:



Node	Adjacency List
A	B, C
B	C, A
C	B, A

Adjacency Matrix or Adjacency List, which one is better?

Implementating Graphs – Analysis

- Which implementation is better?

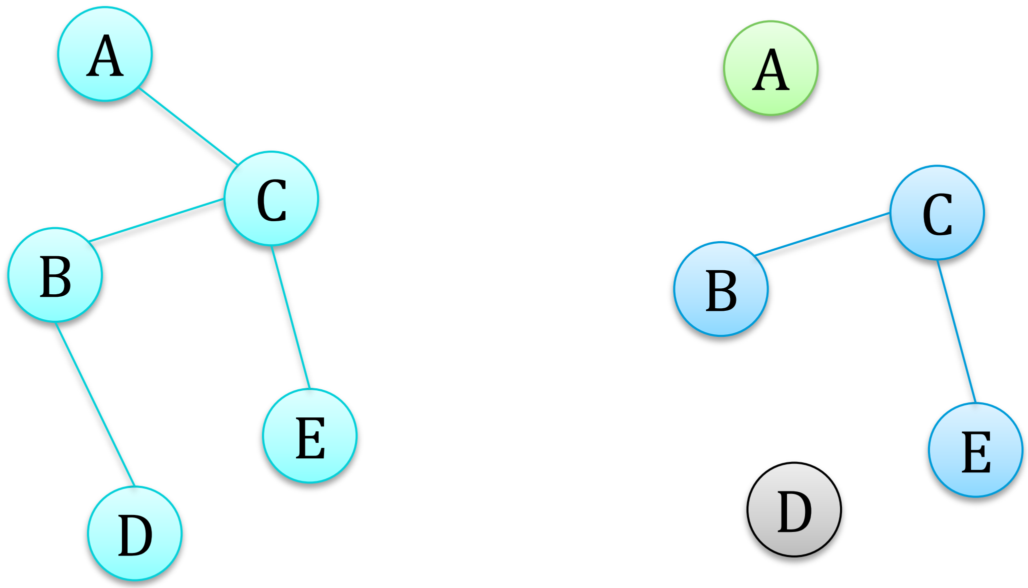
→ depends on how your particular application uses the graph.

1. Determine whether there is an edge from vertex i to vertex j
2. Find all vertices adjacent to a given vertex

- Space requirement:
  - Adjacency Matrix: n^2 entries
  - Adjacency List:
    - n head pointers
    - # nodes = # edges (or twice # edges in a directed graph)

Graph Traversals

- Visit all the vertices that it can reach.
  - Does not need to visit all the vertices? (Why?)
  - Visit only the subset of the graph's vertices: connected component



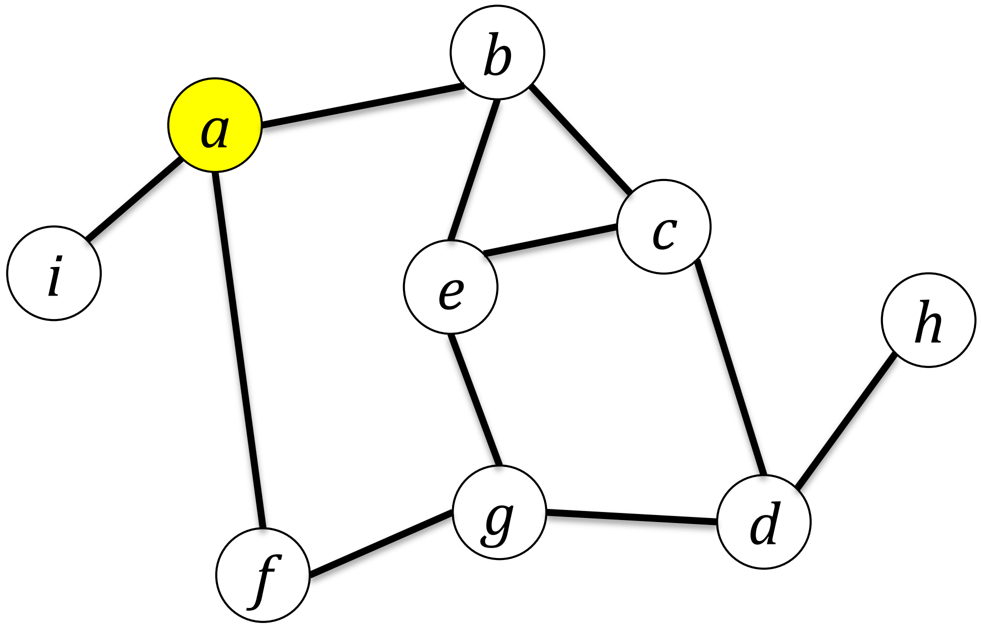
(a) Connected graph (b) Disconnected graph

Depth-first search

- From a given vertex v, the DFS strategy proceeds along a path from v as deeply into the graph as possible before backing up.
- Example:
  - DFS traversal visits

    all the vertices in order:

    a, b, c, d, g, e, f, h, i



Depth-first search implement

- Recursive version:

  DFS(*v*) //Traverses a graph beginning at vertex *v*

1. Mark *v* as visited
2. for(each unvisited vertex *u* adjacent to *v*)
3. DFS(*u*)

- Iterative version:

  ...

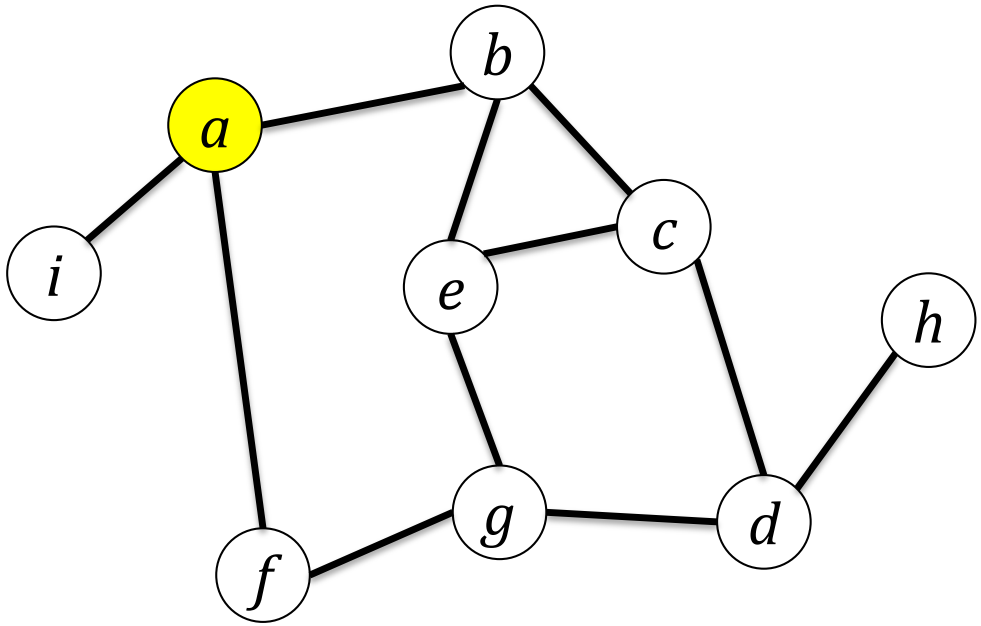
DFS embarks the most recently visited vertex → LIFO → Stack

Breath-first search

- After visiting a given vertex v, BFS visits every vertex adjacent to v that it can before visiting any other vertex
- Example:
  - BFS traversal visits

  all the vertices in order:

  a, b, f, i, c, e, g, d, h



Breadth-first search implement

- Iterative version:

```
BFS(v) //Traverses a graph beginning at vertex v
Q = a new empty queue
Q.Enqueue(v)
Mark v as visited
while(!Q.IsEmpty())
    w = Q.Dequeue()
    for(each unvisited vertex u adjacent to w)
        Mark u as visited
        Q.Enqueue(u)
```

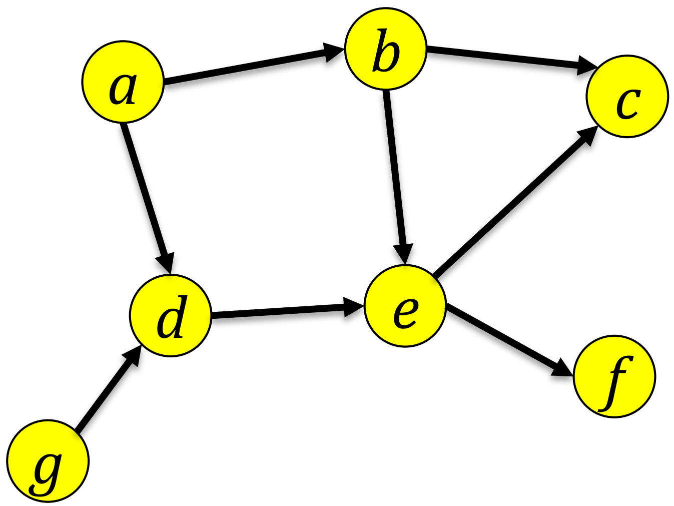
Applications of Graphs

- Topological Sorting
- Spanning Trees
- Minimum Spanning Trees
- Shortest Paths
- Circuits
- Some Difficult Problems

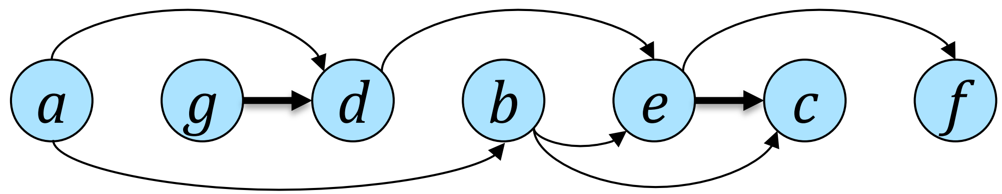
TOPOLOGICAL SORTING

Topological Sorting

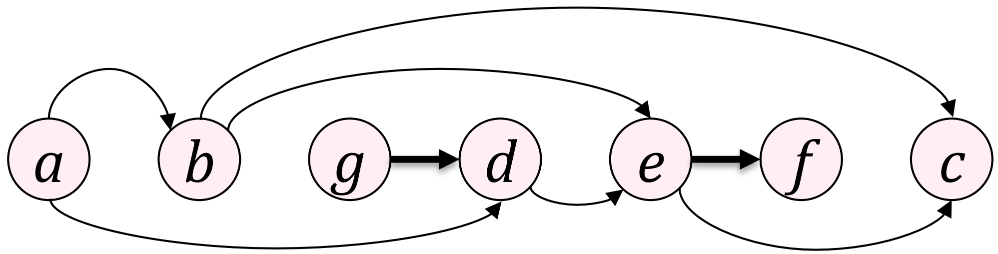
- A directed graph without cycles has a linear order called a topological order: a list of vertices where vertex *x* precedes vertex *y*



Directed graph G



G arranged according to the topological orders



Topological Sorting Algorithm

TOPOLOGICAL-SORT(G, L, n) //Graph G, list L and number of vertices in G

1. n = number of vertices in G
2. for step = 1…n
3. Select a vertex v that has no successors
4. Remove v and all edges to v from G
5. Add v to L

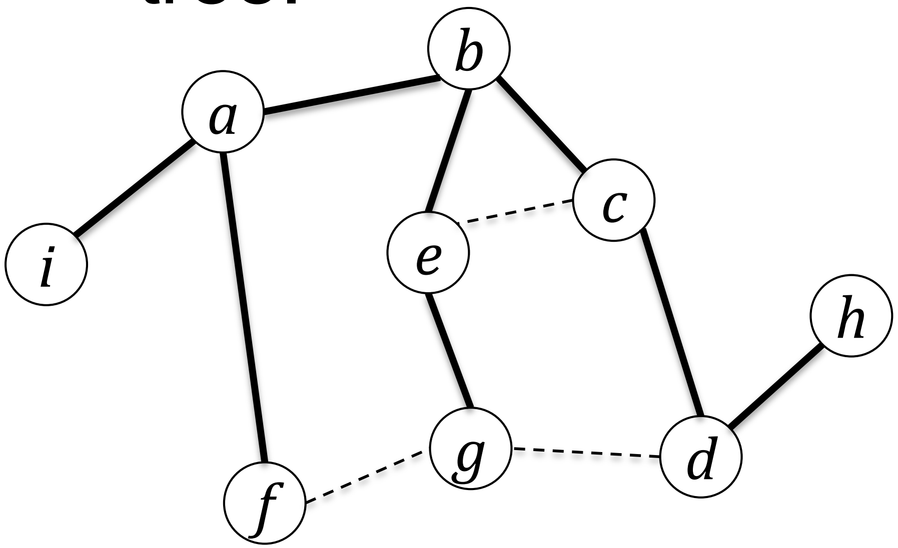
Topological Sorting

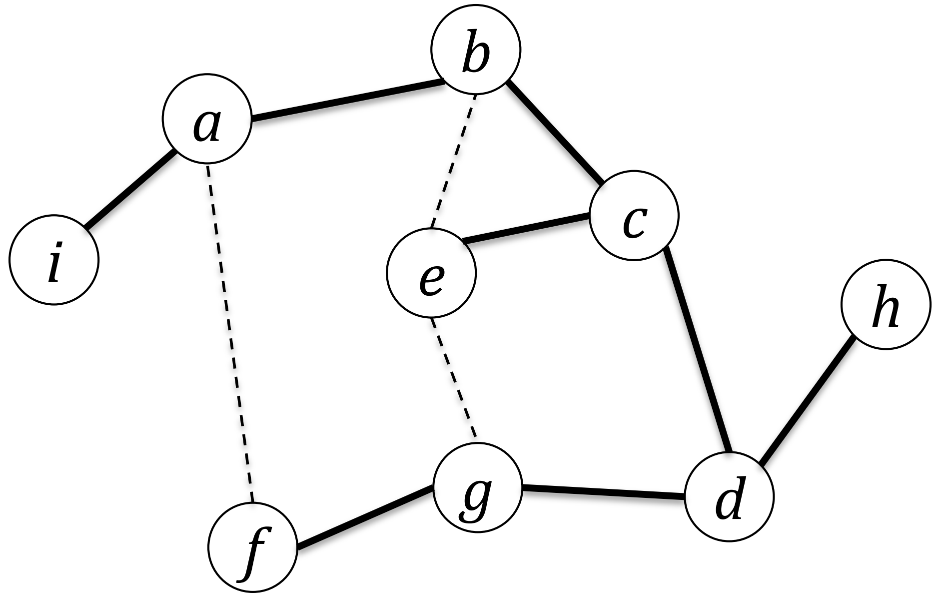
- Application:
  - Represent the prerequisite structure for academic courses
  - Schedule a sequence of jobs or tasks based on their dependencies
  - Compute shortest paths quickly

MINIMUM SPANNING TREE

Spanning Trees

- A spanning tree of a connected undirected graph G is a subgraph of G that contains all of G's vertices and enough of its edges to form a tree.





- There maybe several spanning trees for G

Spanning Trees

- Idea: Remove edges until there are no cycles
- Determine whether a graph contains a cycle:
  - A connected undirected graph that has n vertices must have at least $n - 1$ edges.
  - A connected undirected graph that has n vertices and exactly $n - 1$ edges cannot contain a cycle.
  - A connected undirected graph that has n vertices and more than $n - 1$ edges must contain at least one cycle.

→ *Counting G's vertices and edges*

DFS Spanning Tree

DFSTree(*v*) //Traverses a graph beginning at vertex *v*

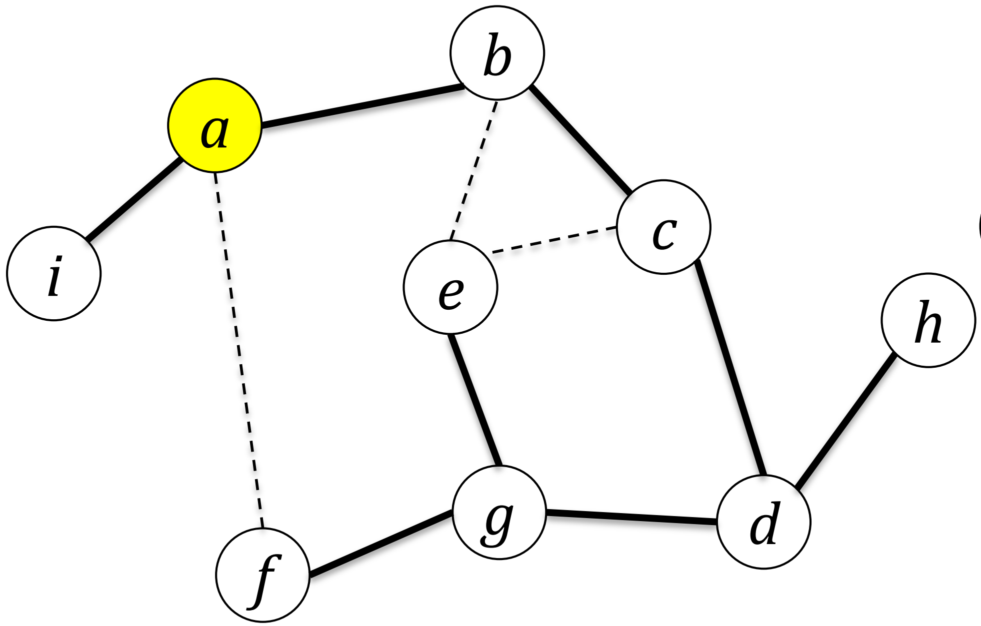
1. Mark *v* as visited
2. for(each unvisited vertex *u* adjacent to *v*)
3.   Mark the edge from *u* to *v*
4.   DFSTree(*u*)

BFS Spanning Tree

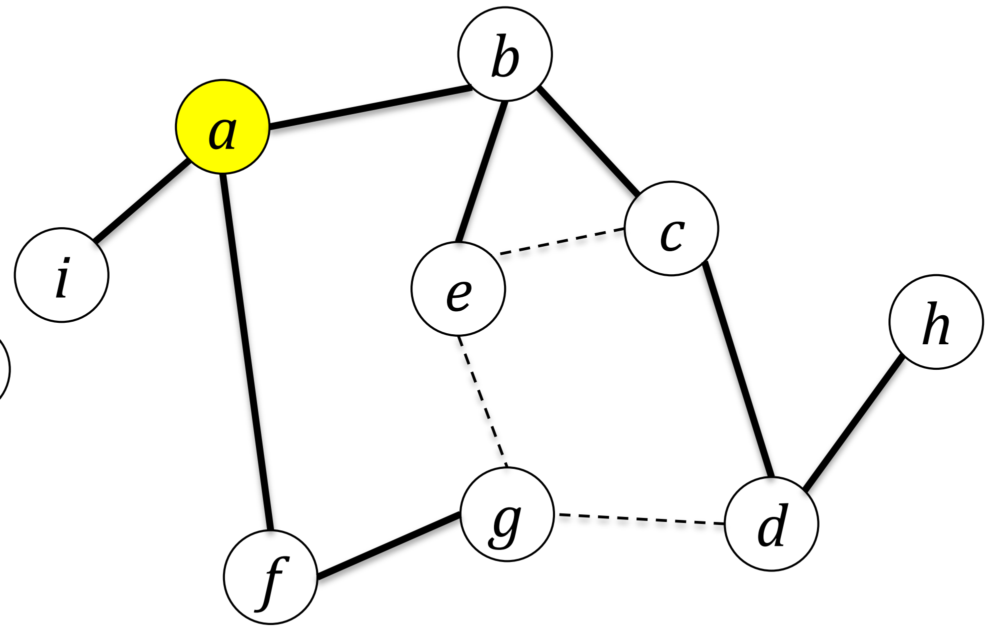
BFSTree(*v*) //Traverses a graph beginning at vertex *v*

1. Q = a new empty queue
2. Q.Enqueue(*v*)
3. Mark *v* as visited
4. while(!Q.IsEmpty())
5.   *w* = Q.Dequeue()
6.   for(each unvisited vertex *u* adjacent to *w*)
7.   Mark *u* as visited
8.   Mark edge between *w* and *u*
9.   Q.Enqueue(*u*)

Spanning Tree

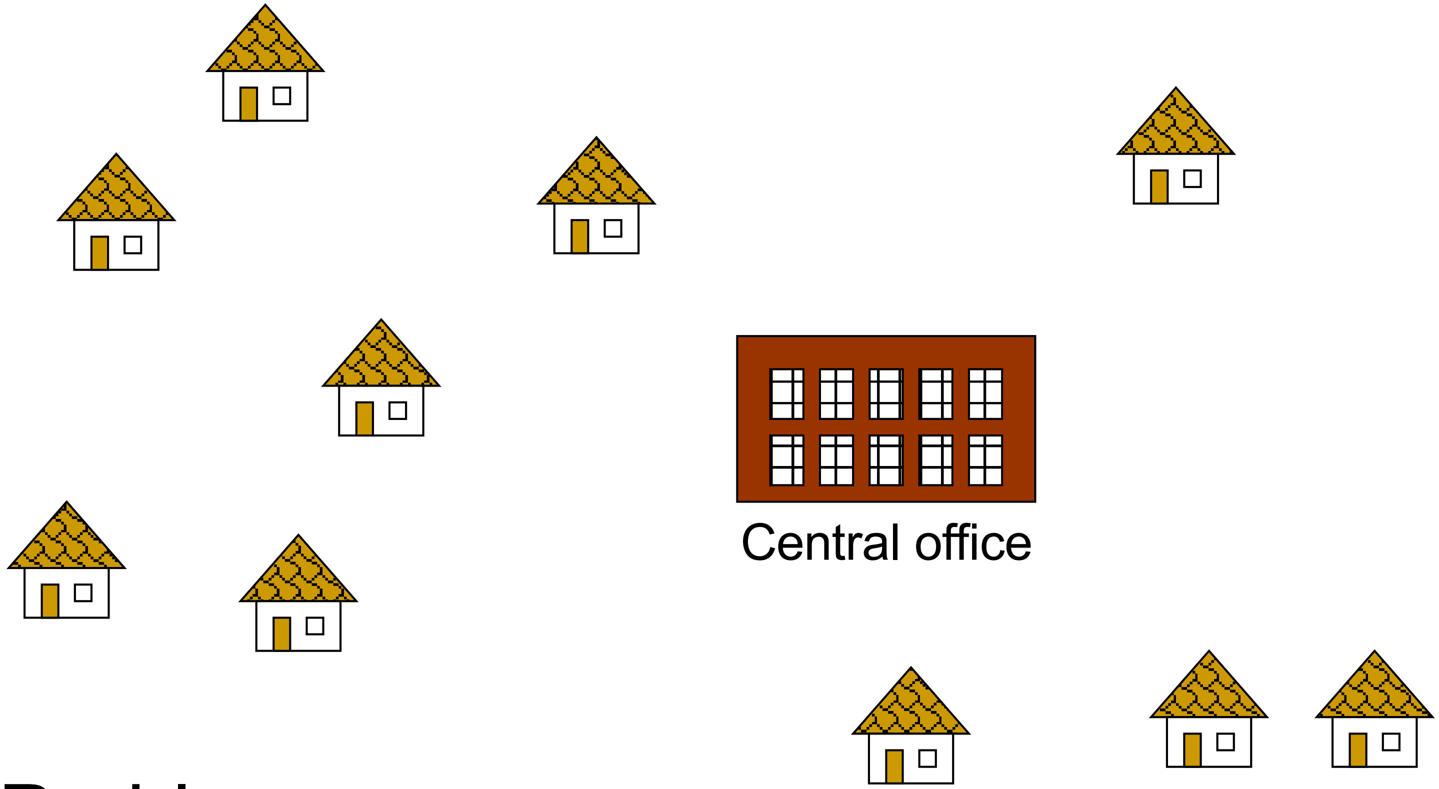


(a) DFS Spanning Tree



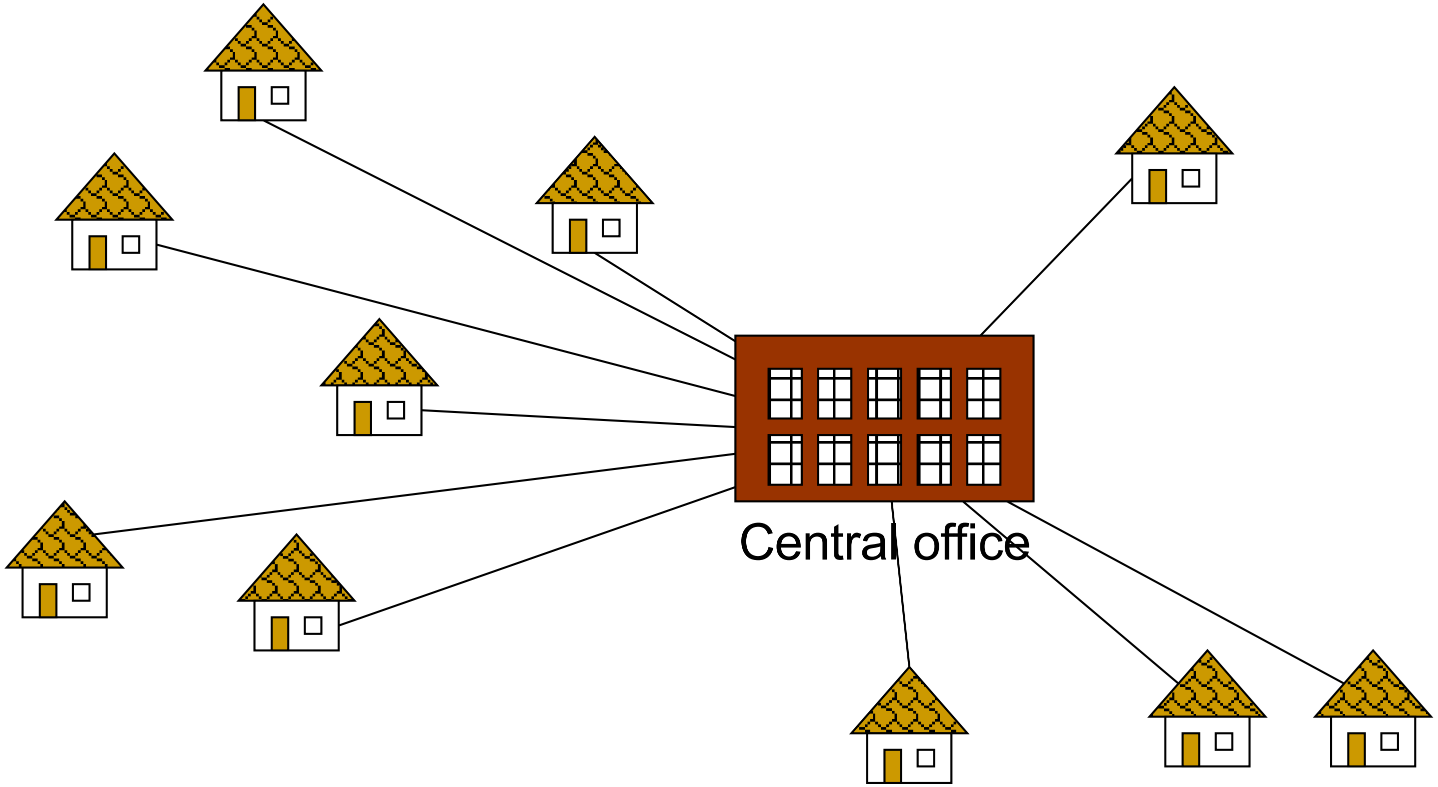
(b) BFS Spanning Tree

Minimum Spanning Trees (MST)



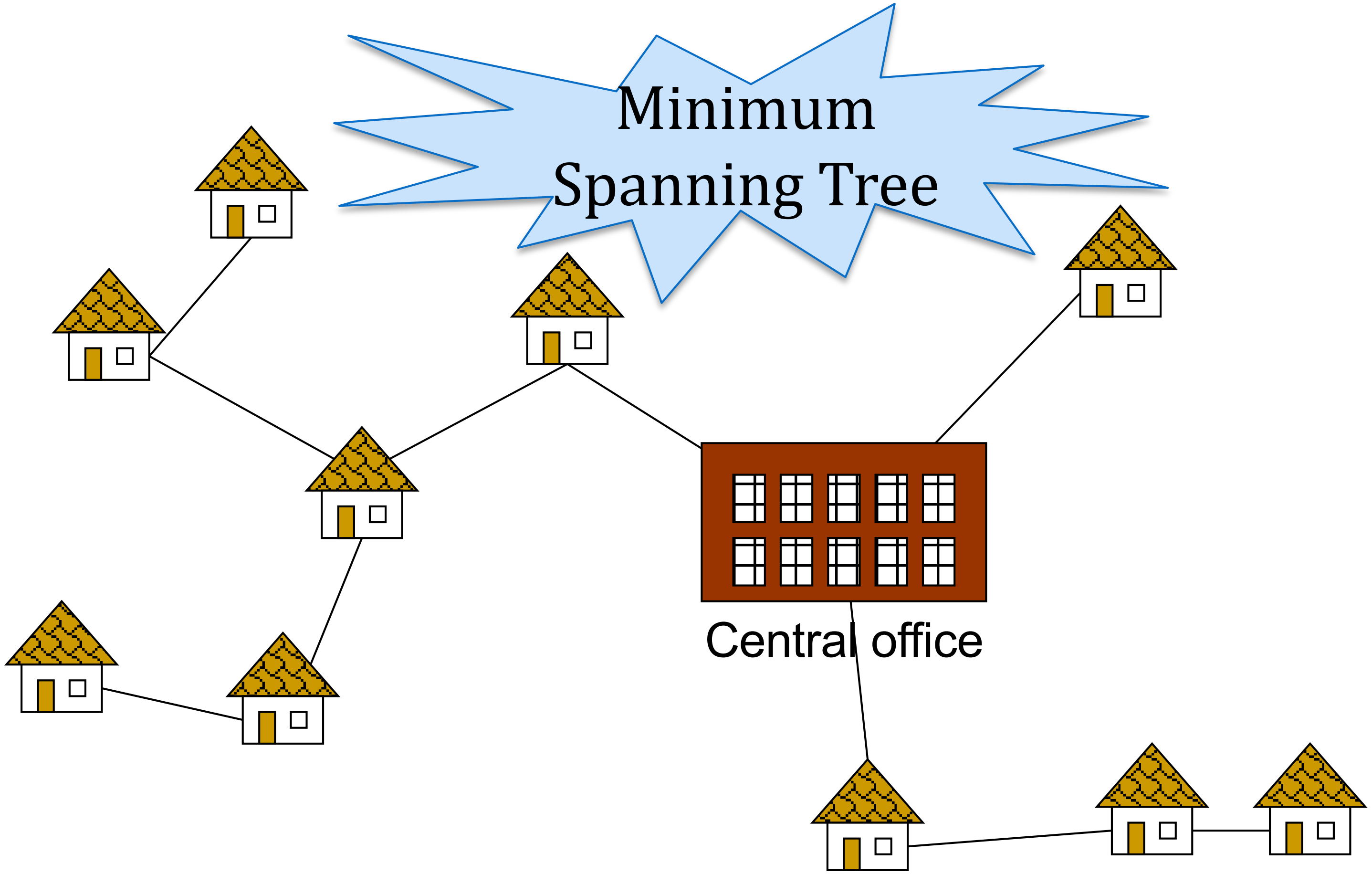
- Problem:
  - Design a telephone wire system so that all customers can call the central office.

Wiring: Naïve Approach



Expensive!

Wiring: Better Approach



Minimize the total length of wire connecting the customers

MST – Prim's algorithm

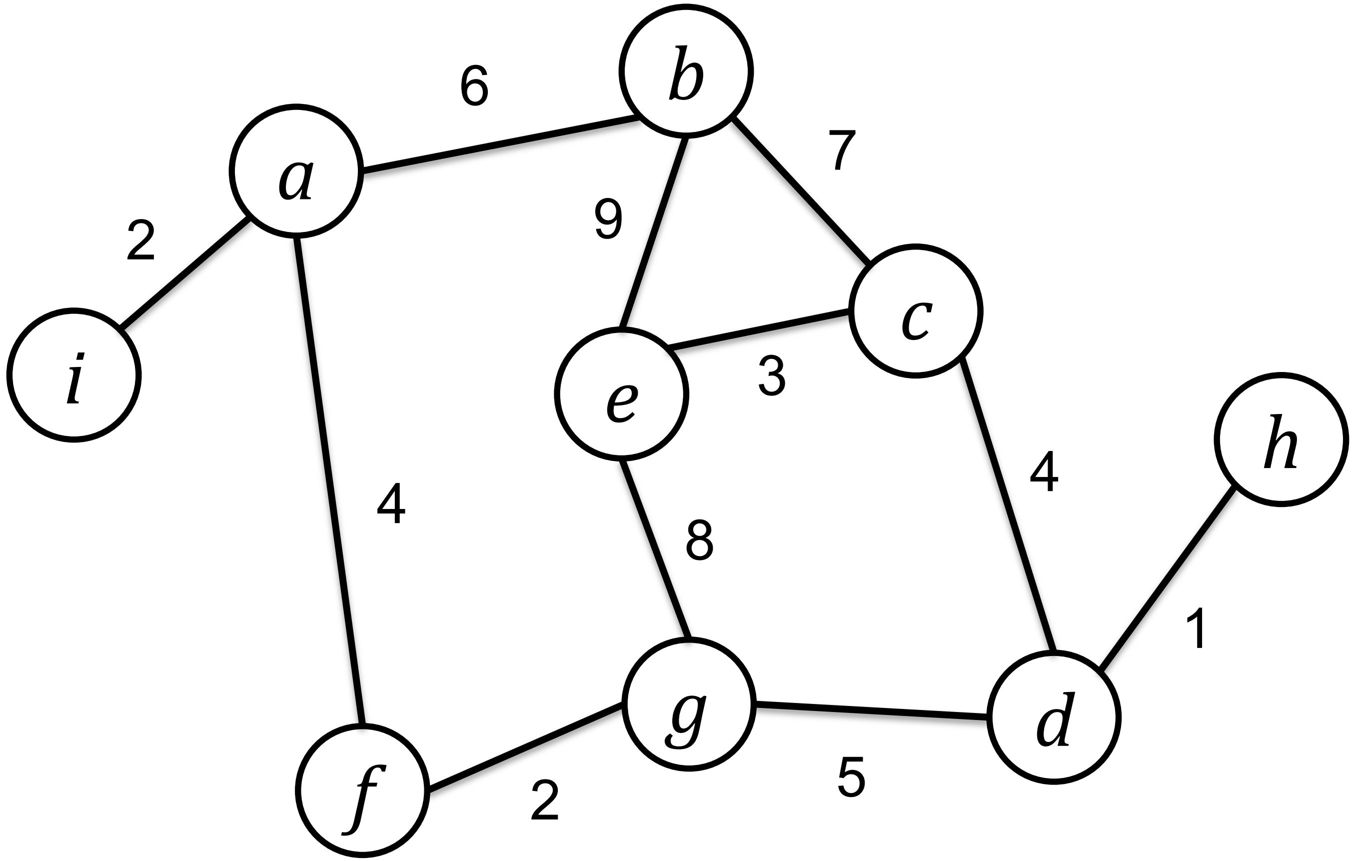
- Idea: find a minimum spanning tree *T*
  - At each stage, select a least-cost edge *e* from among those that begin with a vertex *u* in the tree and end with a vertex *v* not in the tree.
  - Add *v* and *e* into *T.*

MST – Prim’s algorithm

```
MST-PRIM(G, w, r)
for each vertex u ∈ G.V
    u.key = ∞
    u.π = NIL
r.key = 0
Q = Ø
for each vertex u ∈ G.V
    INSERT(Q, u)
while Q ≠ Ø
    u = EXTRACT-MIN(Q)   // add u to the tree
    for each vertex v in G.Adj[u]   // update keys of u’s non-tree neighbors
        if v ∈ Q and w(u, v) < v.key
            v.π = u
            v.key = w(u, v)
            DECREASE-KEY(Q, v, w(u, v))
```

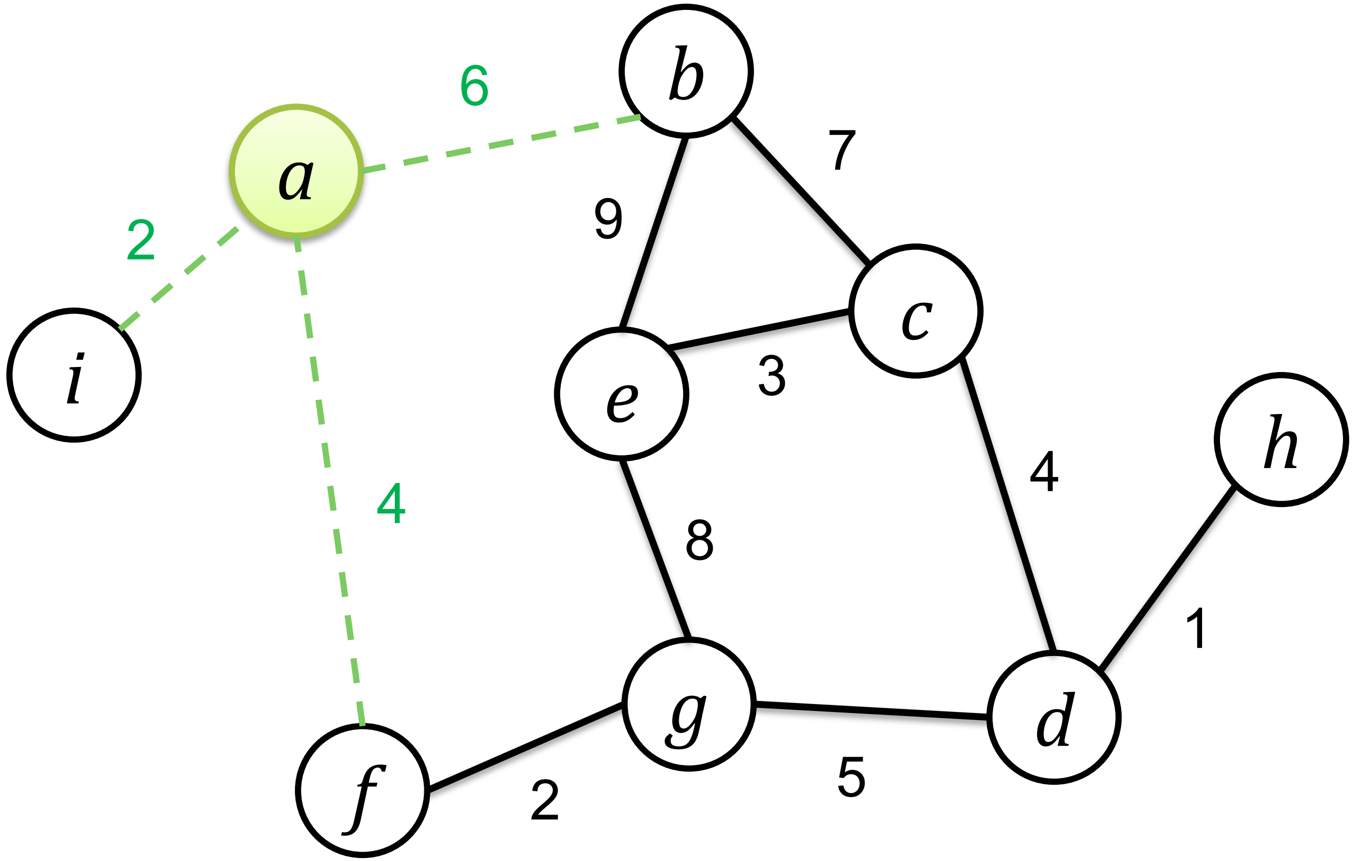
PRIM ALGORITHM – EXAMPLE

- A weighted, connected, undirected graph



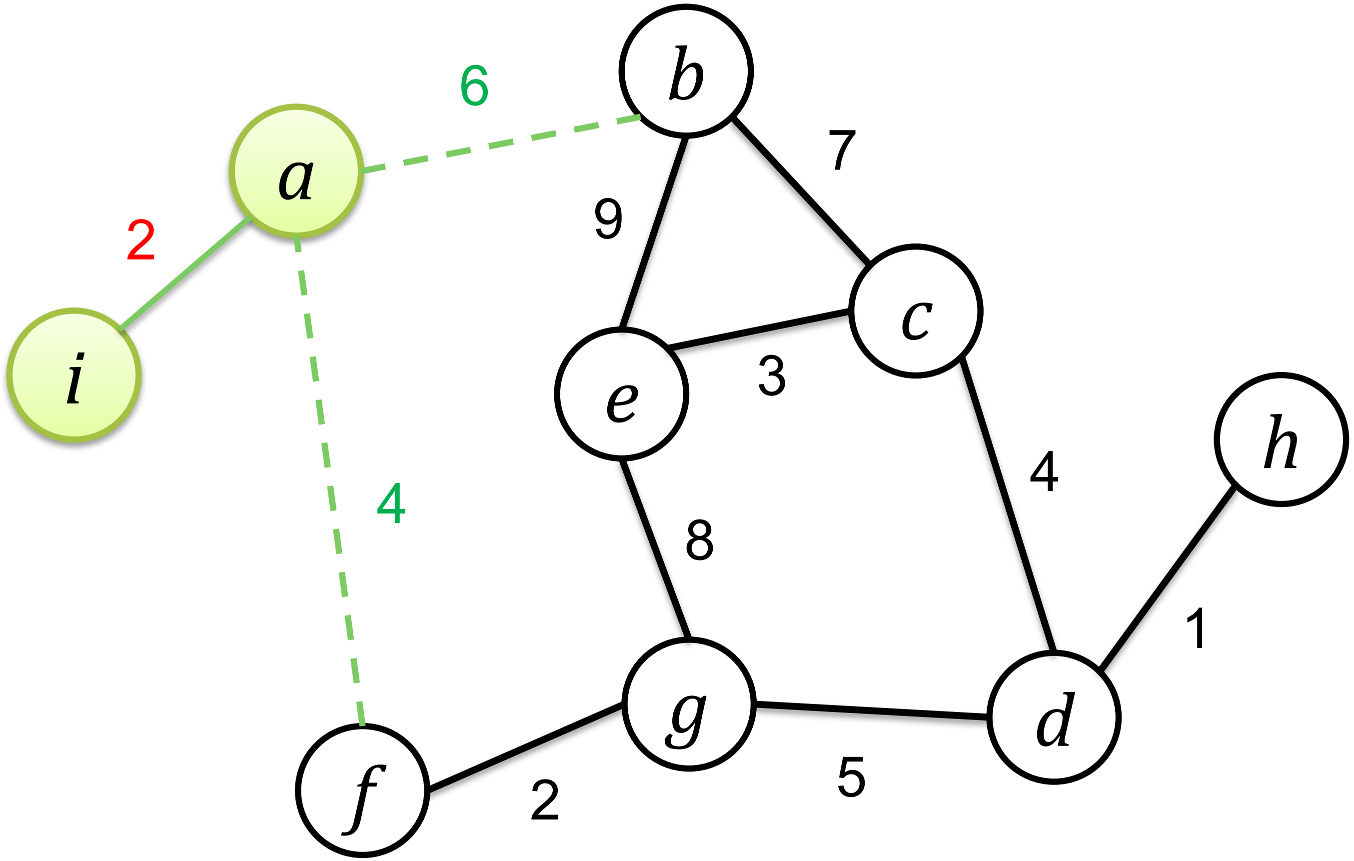
PRIM ALGORITHM – EXAMPLE

- Mark a, consider edges from a



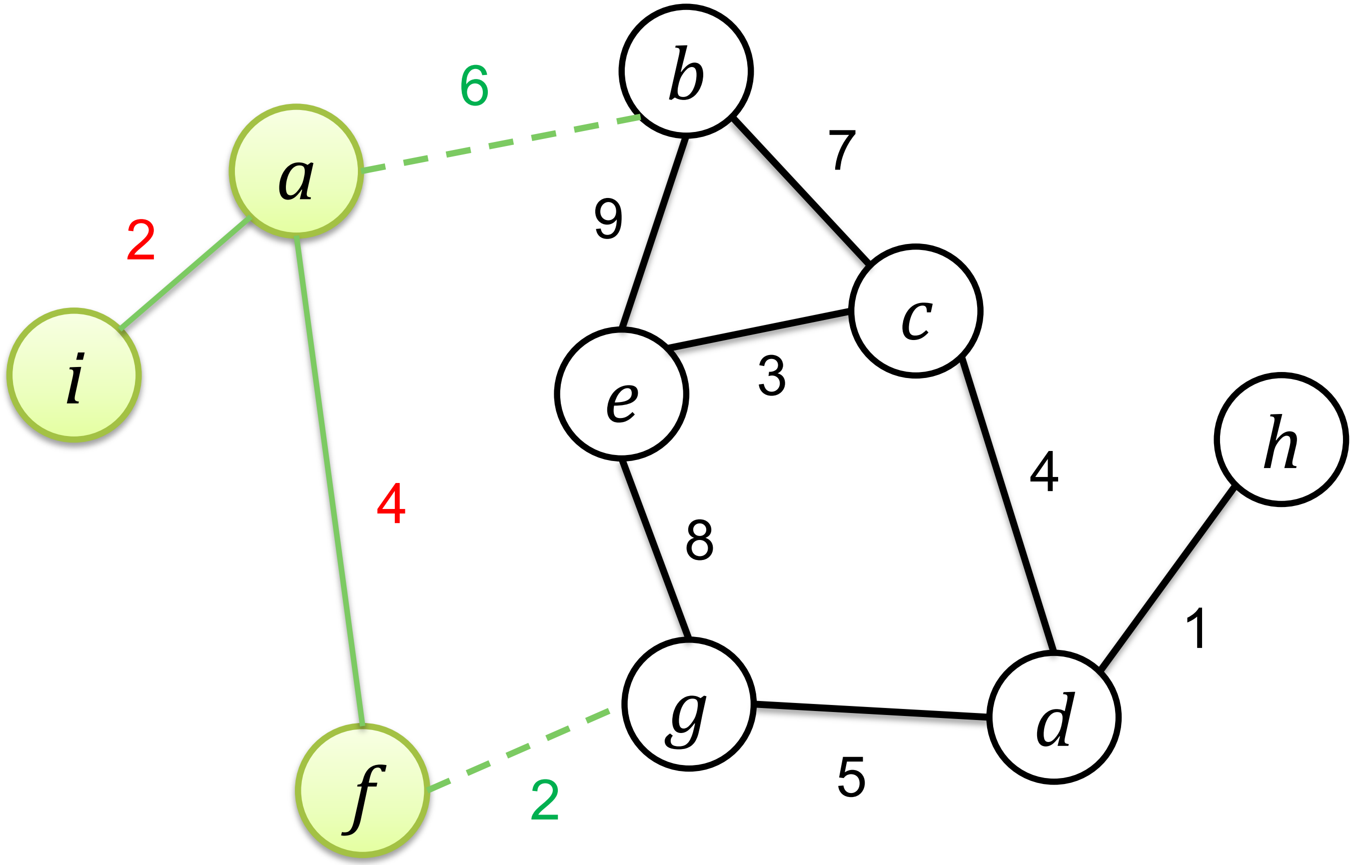
PRIM ALGORITHM – EXAMPLE

- Mark i, include edge (a, i)



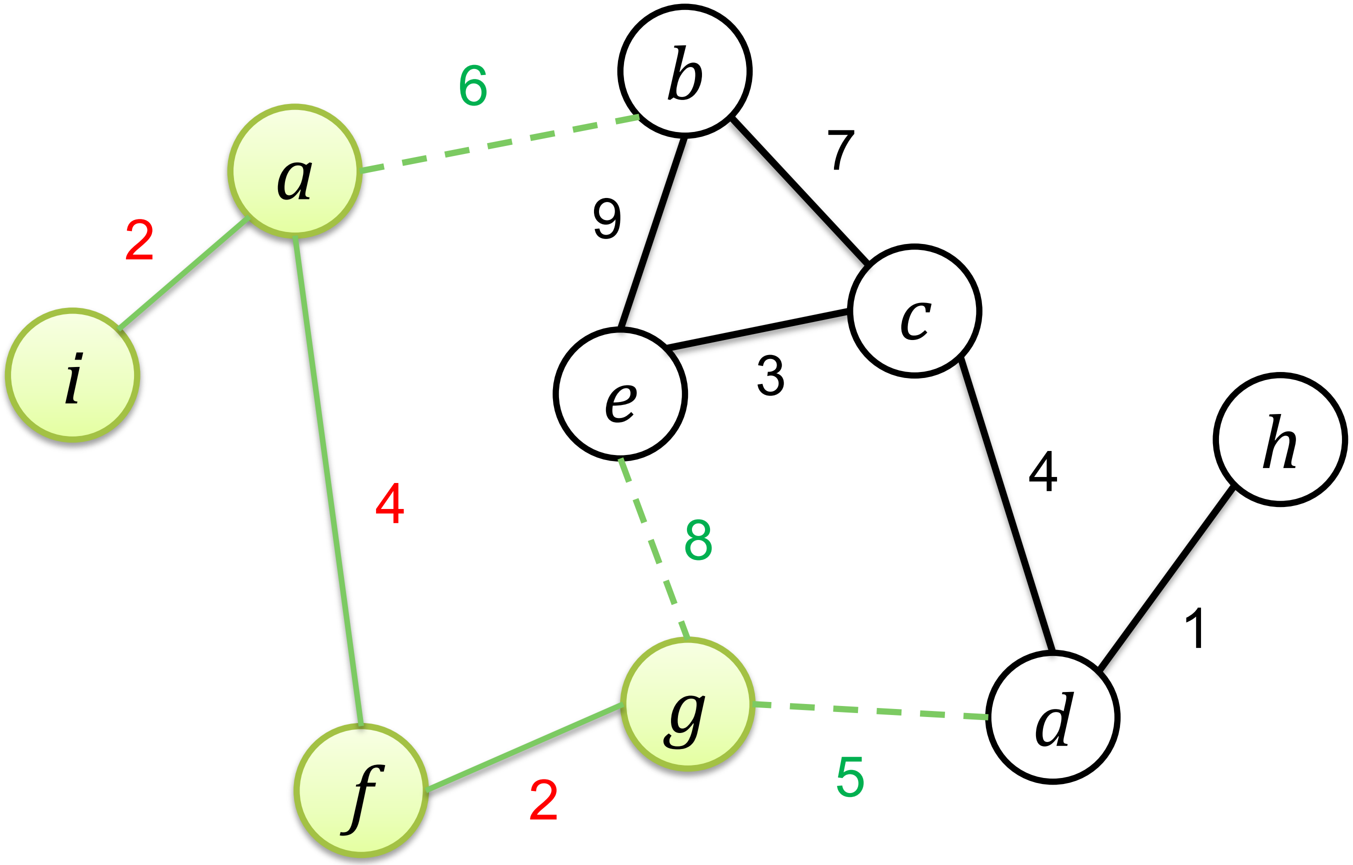
PRIM ALGORITHM – EXAMPLE

- Mark f, include edge (a, f)



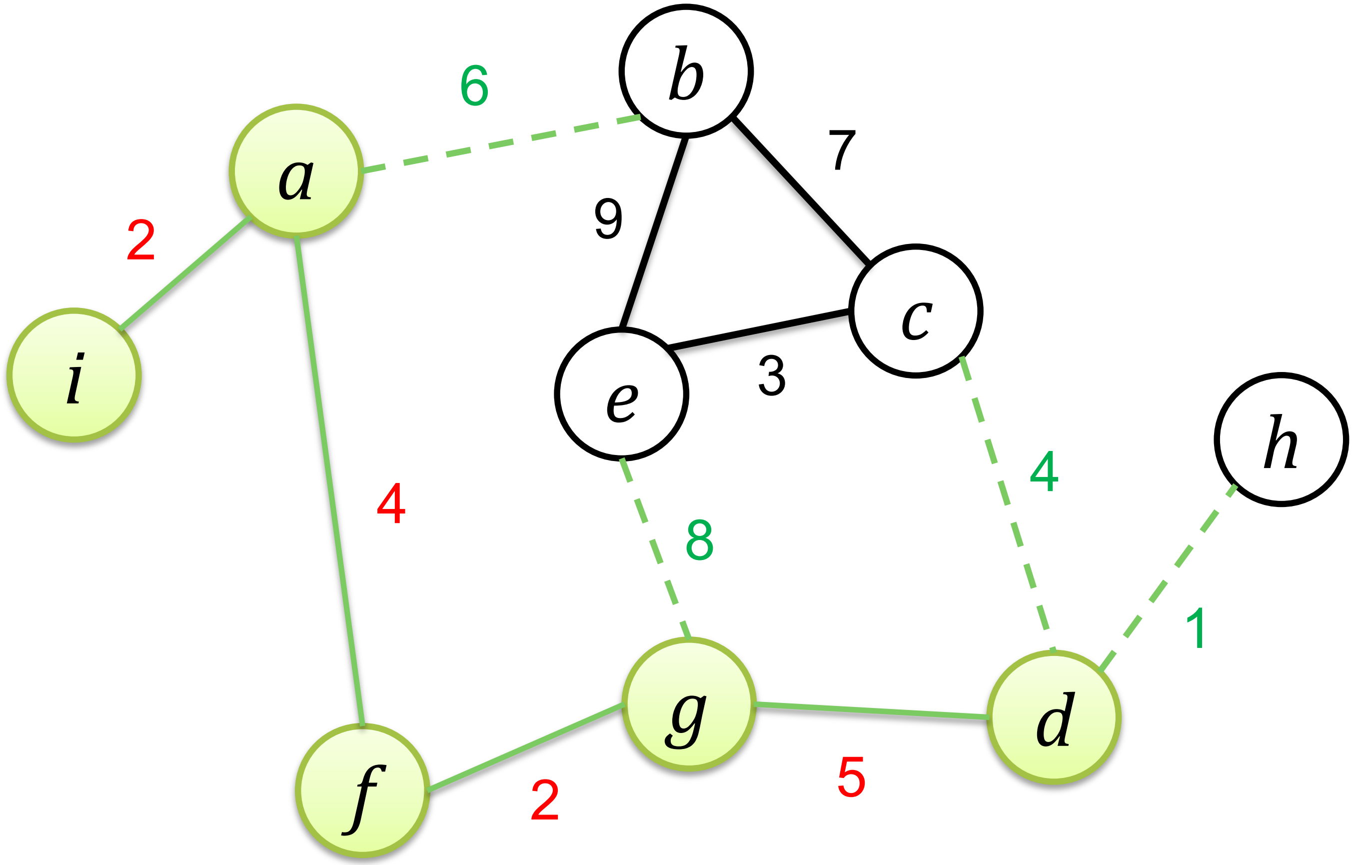
PRIM ALGORITHM – EXAMPLE

- Mark g, include edge (f, g)



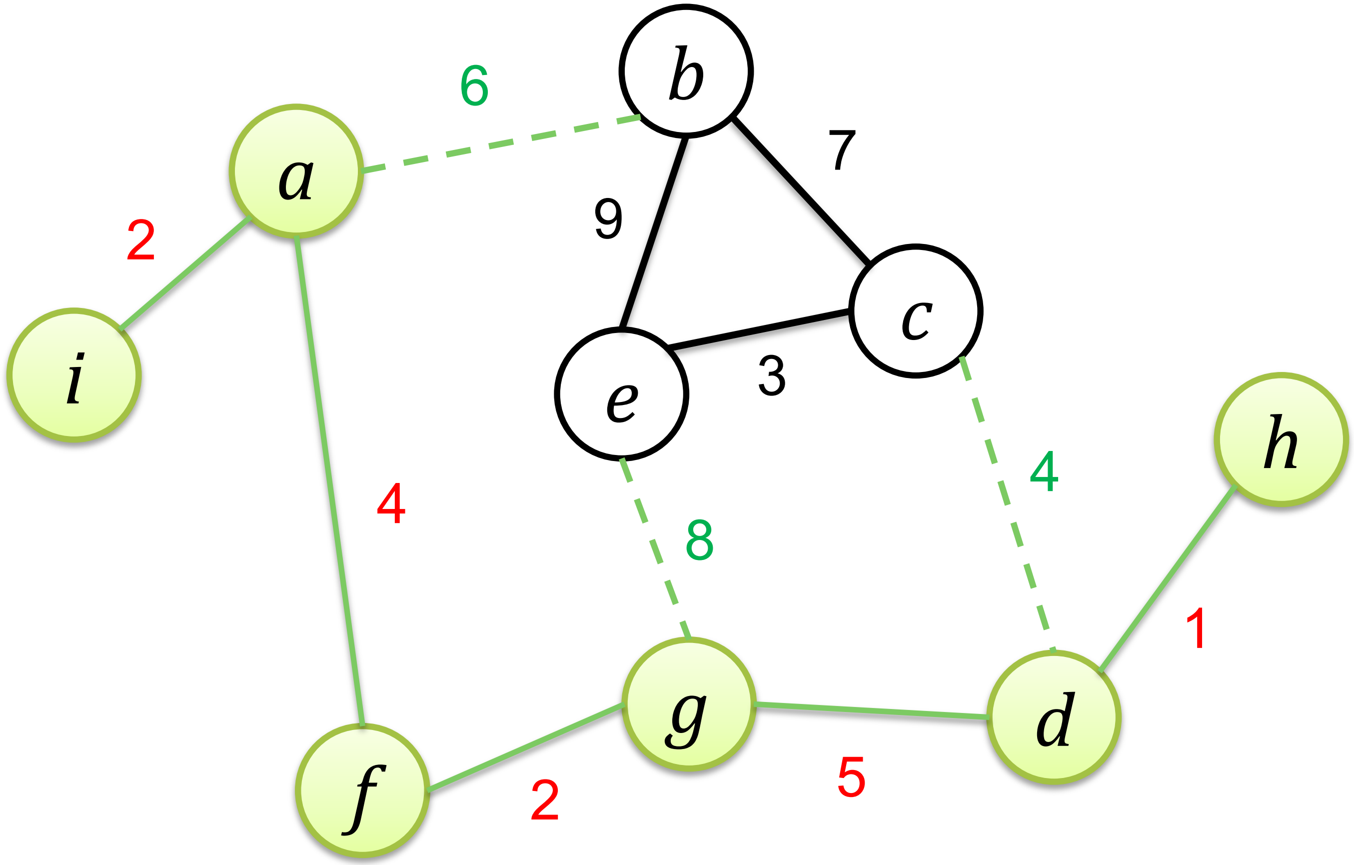
PRIM ALGORITHM – EXAMPLE

- Mark d, include edge (g, d)



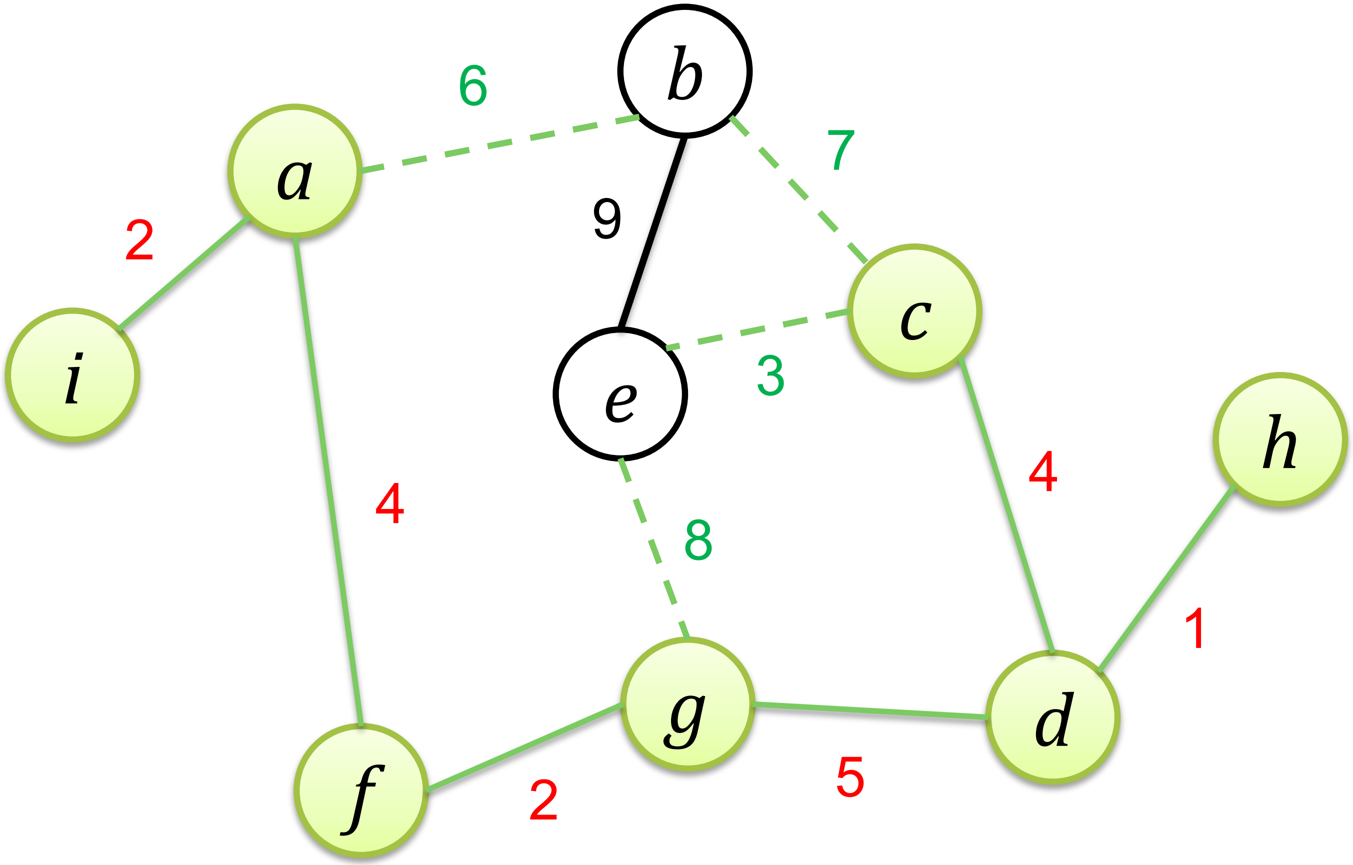
PRIM ALGORITHM – EXAMPLE

- Mark h, include edge (d, h)



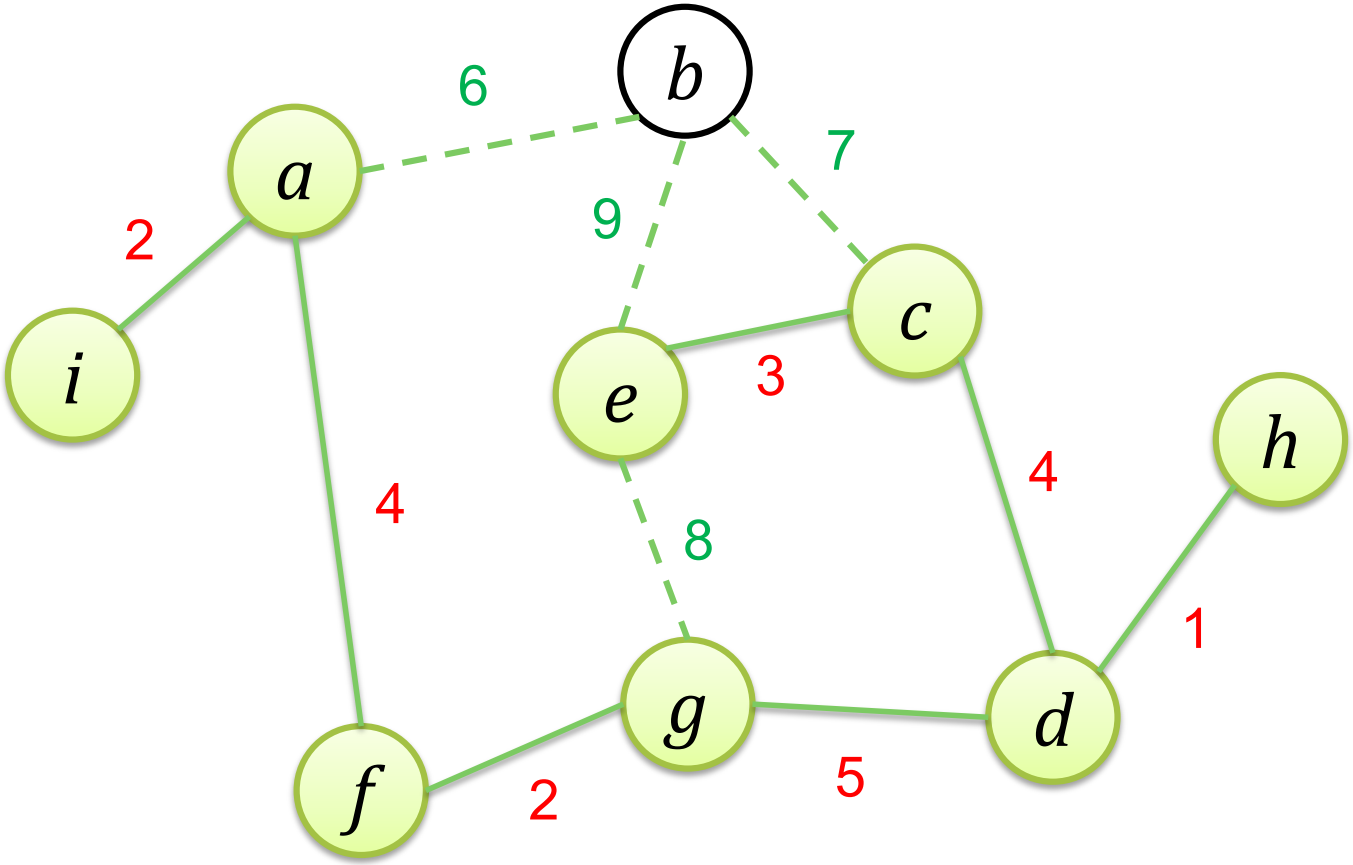
PRIM ALGORITHM – EXAMPLE

- Mark c, include edge (d, c)



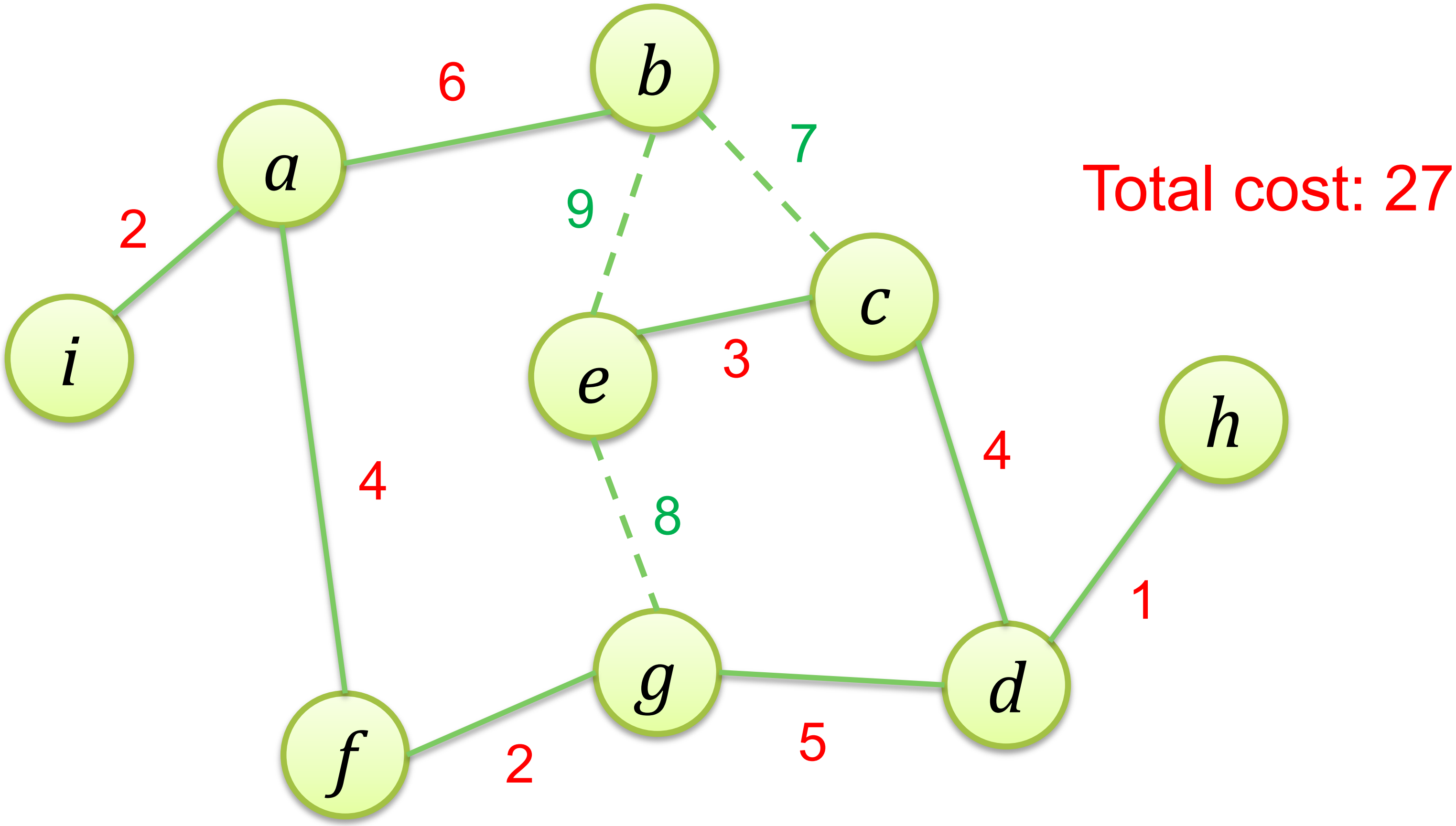
PRIM ALGORITHM – EXAMPLE

- Mark e, include edge (c, e)



PRIM ALGORITHM – EXAMPLE

- Mark b, include edge (a, b)



Prim's Algorithm Analysis

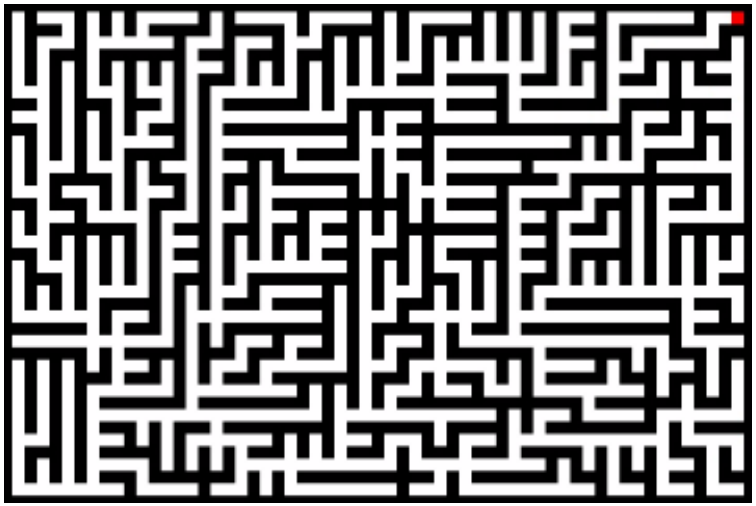
- The running time of Prim's algorithm depends on the implementation of the min-priority queue:
  - We can use binary heap to implement
  - Line 7: build min-heap: $O(\log_2 |V|)$
  - Line 9: Extract-Min: $O(|V| \log_2 |V|)$
  - For loop from line 10-14: $O(|E|)$
  - Line 14: Decrease-Key: $O(|E| \log_2 |V|)$
- Total: $O\big((|V| + |E|) \log_2 |V|\big) = O(|E| \log |V|)$ complexity

Prim's Algorithm Analysis

- You can further improve the asymptotic running time of Prim's algorithm by implementing the min-priority queue with a Fibonacci heap.
  - If a Fibonacci heap holds $|V|$ elements:
  - EXTRACT-MIN operation takes $O(\log_2 |V|)$ time.
  - INSERT and DECREASE-KEY operation takes only $O(1)$
- Therefore, by using a Fibonacci heap to implement the min-priority queue Q, the running time of Prim's algorithm improves to $O(|E| + |V| \log|V|)$

MST Applications

- Planning how to lay network cable to connect several locations to the internet
- Planning how to efficiently bounce data from router to router to reach its internet destination
- Creating a 2D maze (to print on cereal boxes, etc.)



Q&A

What's next?

- After today:
  - Reading Textbook 2, Chapter 20 (page 630~)
  - Do homework 7
- Next class:
  - Individual Assignment 5
  - Lecture 7 part 2: Finding Shortest Path & Other Graphs Problems