GRAPHS part II

Agenda

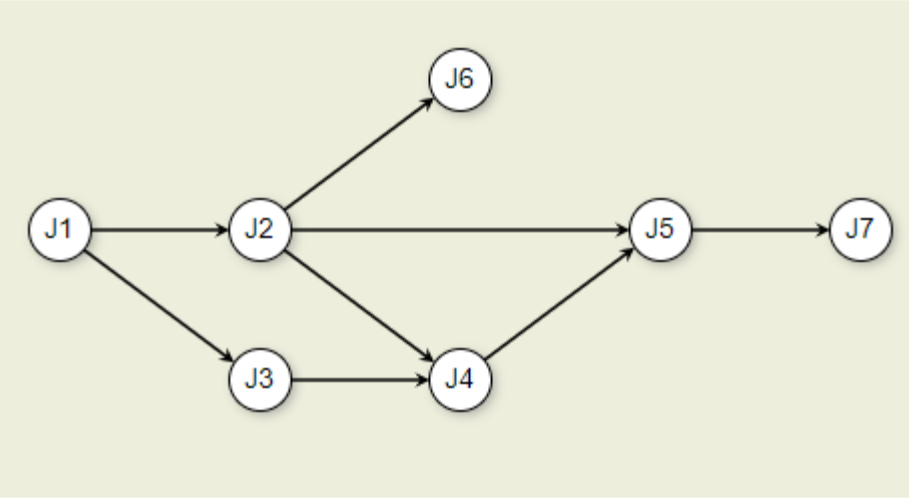
3. Topological Sort
4. Dijkstra's algorithm: finding shortest distance

3. Topological sort

Topological sort : DFS Solution

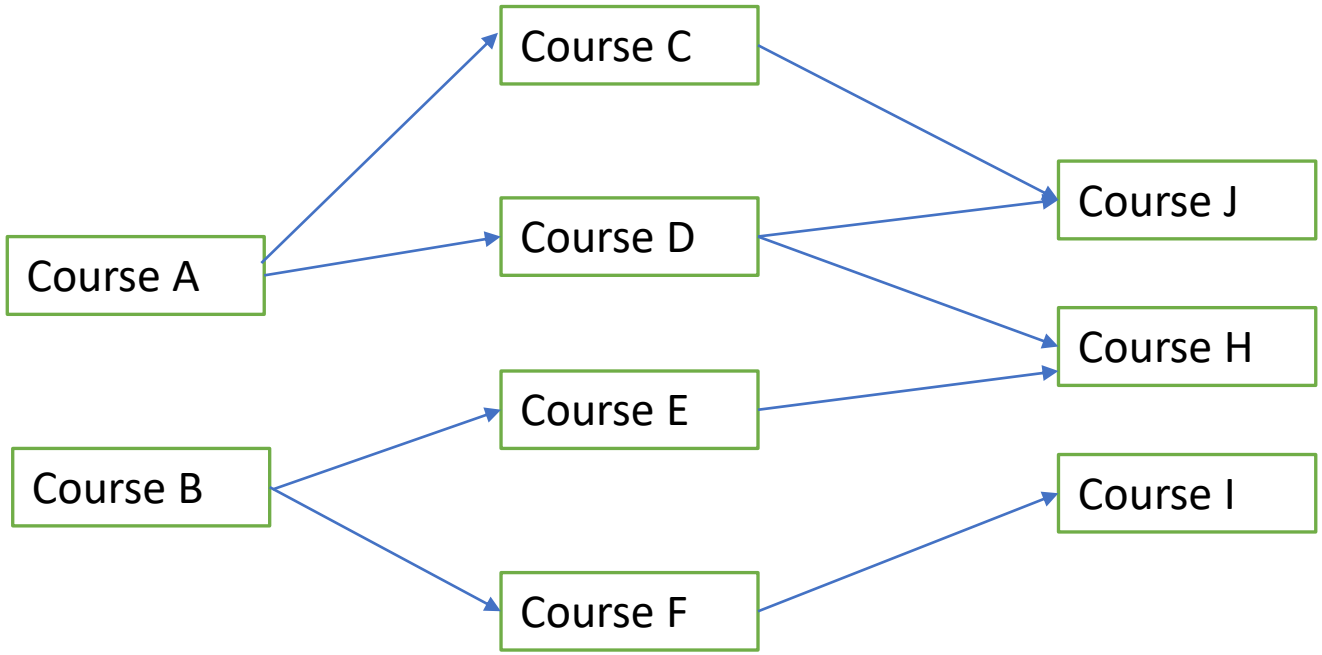
Motivation:

- Schedule a series of tasks, such as classes or construction jobs, where we cannot start one task until after its prerequisites are completed.
- Organize the tasks into a linear order that allows us to complete them one at a time without violating any prerequisites.

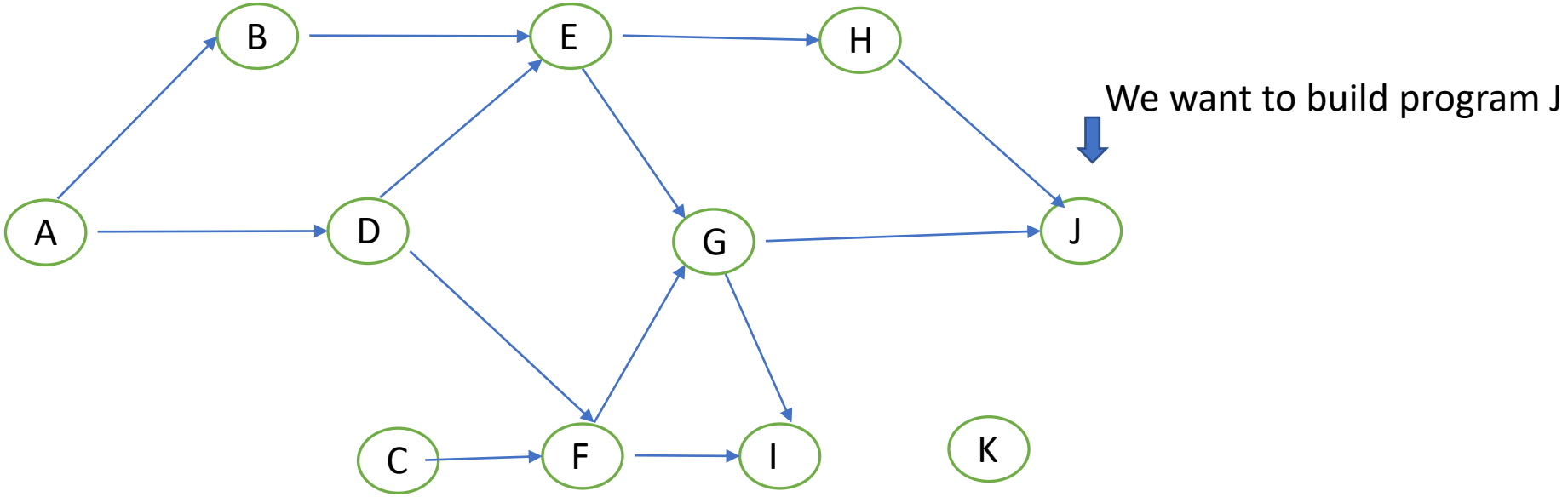


an acceptable topological sort for this example is J1, J2, J3, J4, J5, J6, J7. However, other orders are also acceptable, such as J1, J3, J2, J6, J4, J5, J7.

Example 1

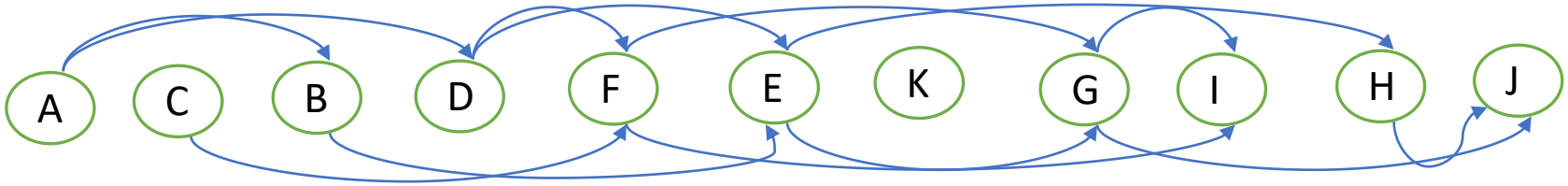

Course C
Course D
Course A
Course J
Course H
Course E
Course B
Course I
Course F

Example 2



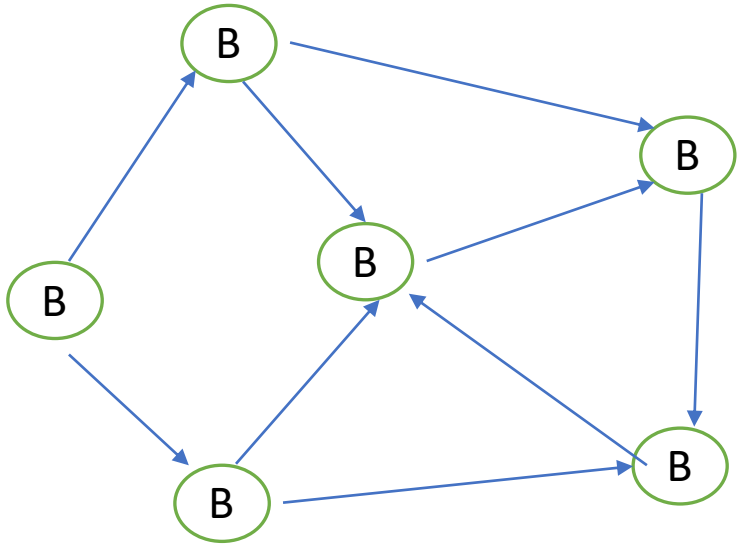
Topological Sort Characteristics

- Is an ordering of the vertices in a directed graph where for each directed edge from vertex A to vertex B, a vertex A appears before vertex B in the ordering.
- A topological sort algorithm can find a topological ordering in **O(V+E)**
- Model the problem using a DAG.
- Not unique



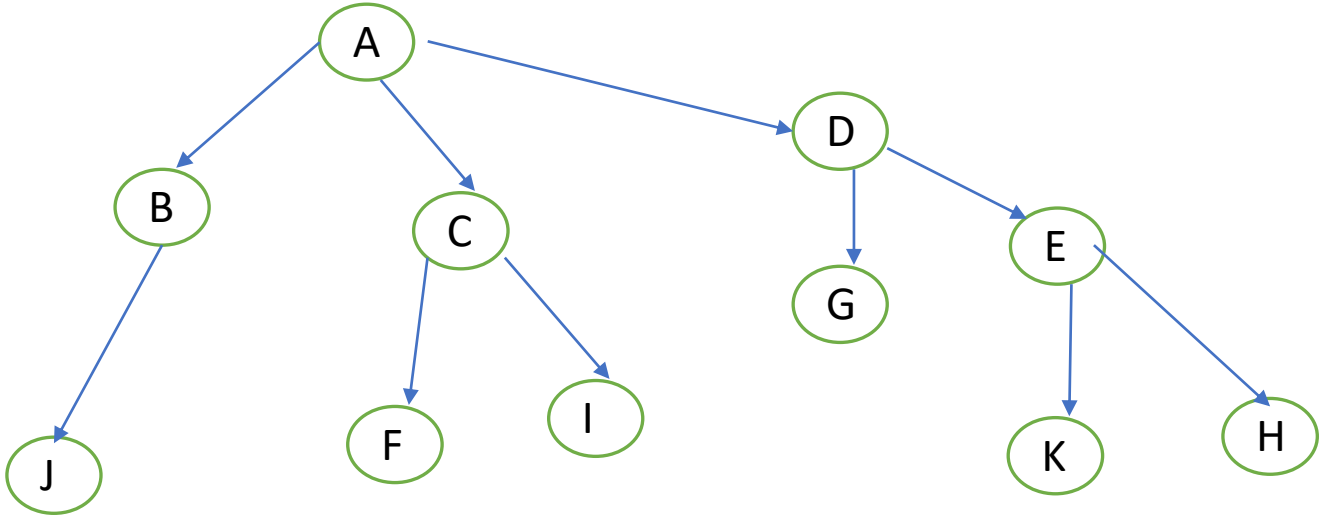
Directed Acyclic Graphs (DAG)

- Not every graph can have a topological ordering.
- A graph which contains a cycle cannot have a valid ordering



Directed Acyclic Graphs (DAG)

- By definition, all rooted trees have a topological ordering since they do not contain any cycle



Topological ordering from left to right

Topological Sort Algorithm

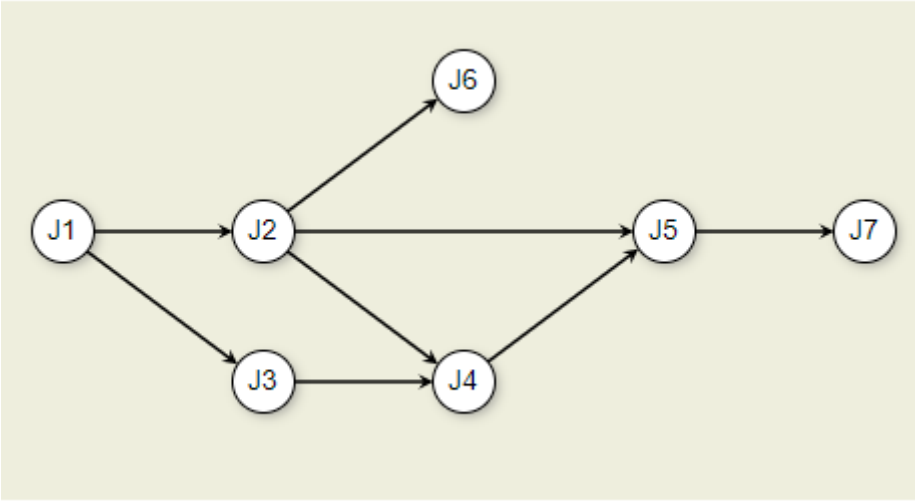
- Pick up unvisited vertex
- Beginning with the selected vertex, do a DFS exploring only unvisited node
- On the recursive callback of DFS, add the current vertex to the topological ordering in reverse order.

Topological sort (Cont'd): DFS Solution

- A topological sort may perform a DFS on the graph. When a vertex is visited, no action is taken (i.e., function `PreVisit` does nothing).
- When the recursion pops back to that vertex, function `PostVisit` prints the vertex. This yields a topological sort in reverse order.
- It does not matter where the sort starts, as long as all vertices are visited in the end. Here is implementation for the DFS-based algorithm.

Topological sort implemented as DFS:

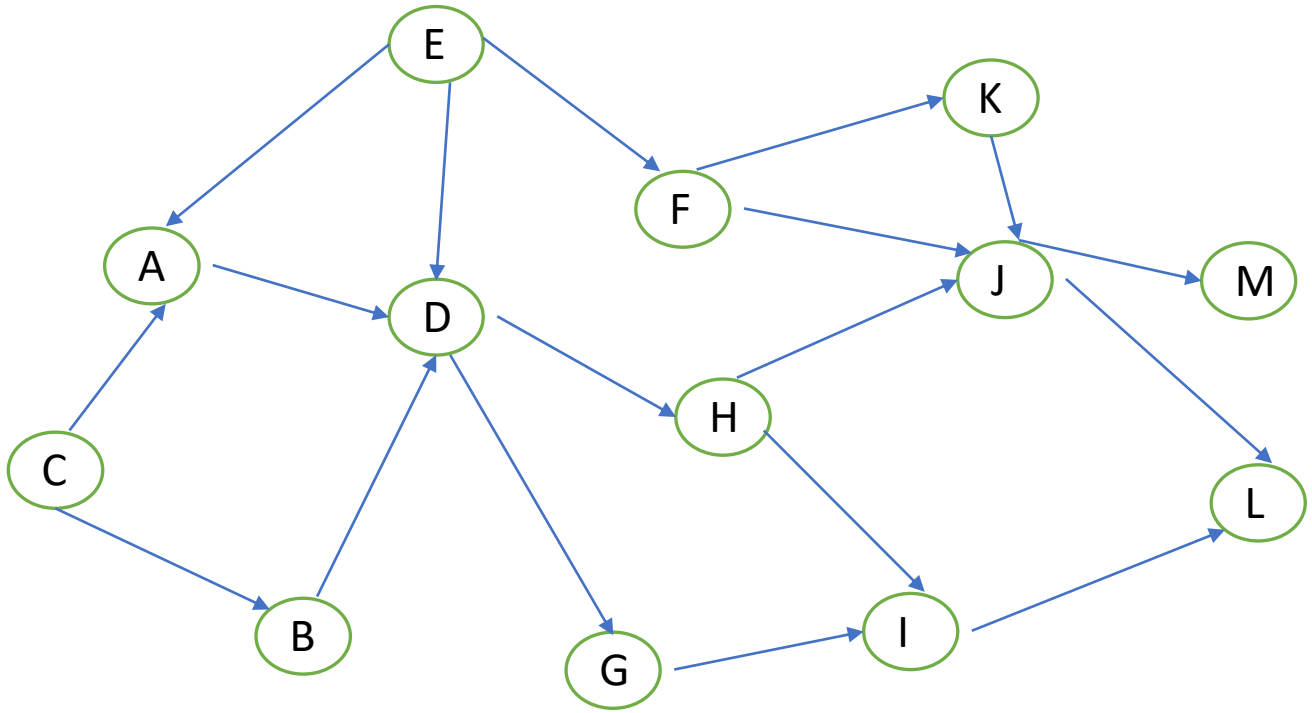
```
static void topsortDFS(Graph G) {
  int v;
  for (v=0; v<G.nodeCount(); v++)
    G.setValue(v, null); // Initialize
  for (v=0; v<G.nodeCount(); v++)
    if (G.getValue(v) != VISITED)
      tophelp(G, v);
}

static void tophelp(Graph G, int v) {
  G.setValue(v, VISITED);
  int[] nList = G.neighbors(v);
  for (int i=0; i< nList.length; i++)
    if (G.getValue(nList[i]) != VISITED)
      tophelp(G, nList[i]);
  printout(v);
}
```



Using this algorithm starting at J1 and visiting adjacent neighbors in alphabetic order, vertices of the graph in are printed out in the order J7, J5, J4, J6, J2, J3, J1. Reversing this yields the topological sort J1, J3, J2, J6, J4, J5, J7.

Example

DFS recursion
call stack:

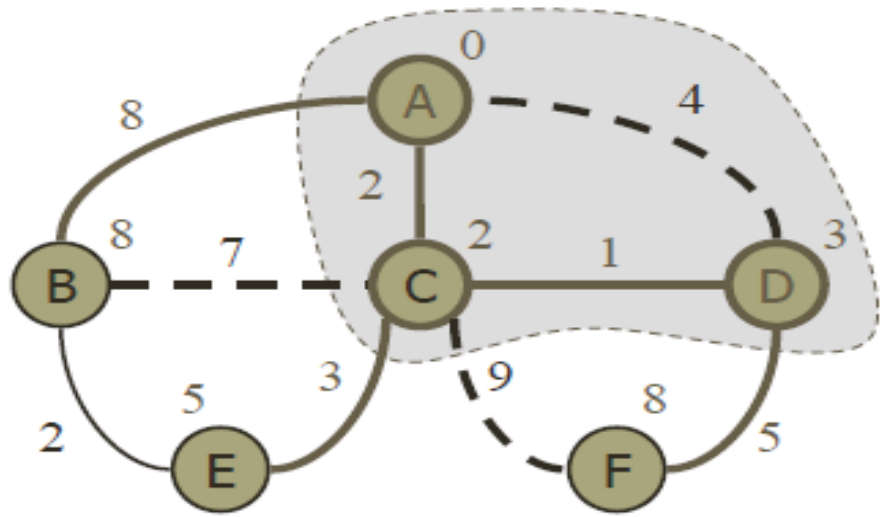


One of the Topological ordering:

4. Dijkstra's Algorithm

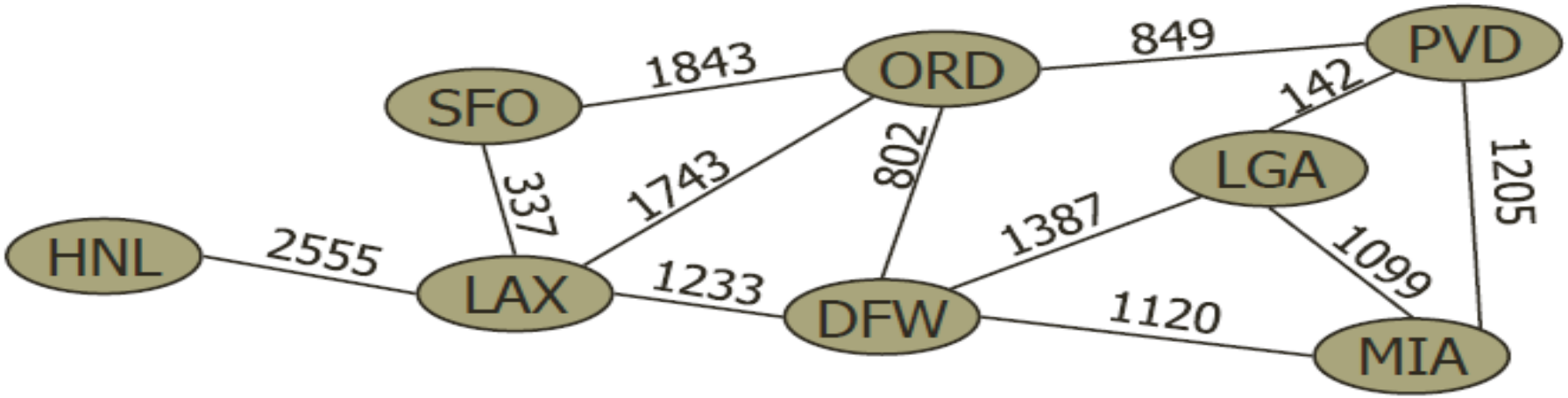
- finding shortest distance

Shortest Paths



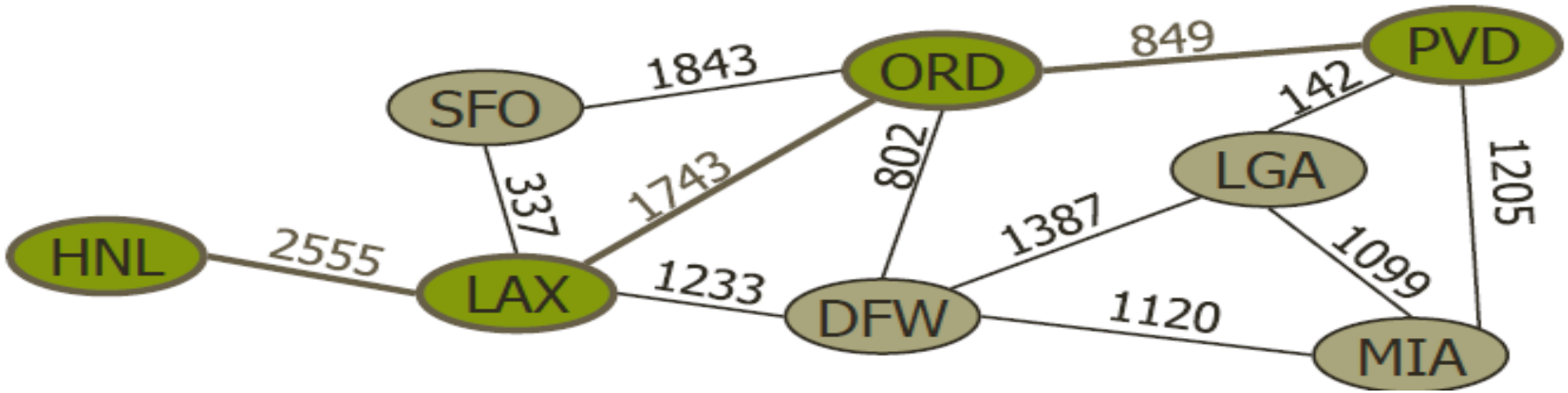
Weighted Graphs

- In a weighted graph, each edge has an associated numerical value, called the weight of the edge
- Edge weights may represent, distances, costs, etc.
- **Example:**
  - In a flight route graph, the weight of an edge represents the distance in miles between the endpoint airports



Shortest Paths

- Given a weighted graph and two vertices ***u*** and ***v***, we want to find a path of minimum total weight between ***u*** and ***v***.
  - Length of a path is the sum of the weights of its edges.
- **Example:** Shortest path between Providence and Honolulu
- **Applications:**
  - Internet packet routing
  - Flight reservations
  - Driving directions

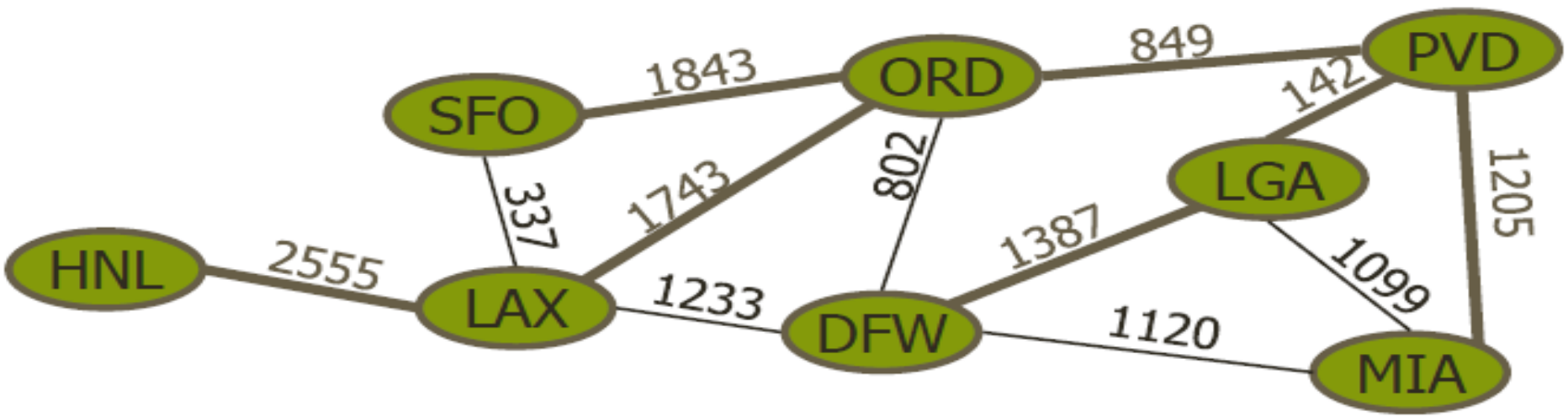


Shortest Path Properties

Property 1: A subpath of a shortest path is itself a shortest path

Property 2: There is a tree of shortest paths from a start vertex to all the other vertices

Example: Tree of shortest paths from Providence



Dijkstra's Algorithm

- The distance from a vertex ***s*** to ***v*** is the length of a shortest path from ***s*** to ***v***

- **Assumptions:**
  - connected & undirected graph
  - edge weights nonnegative

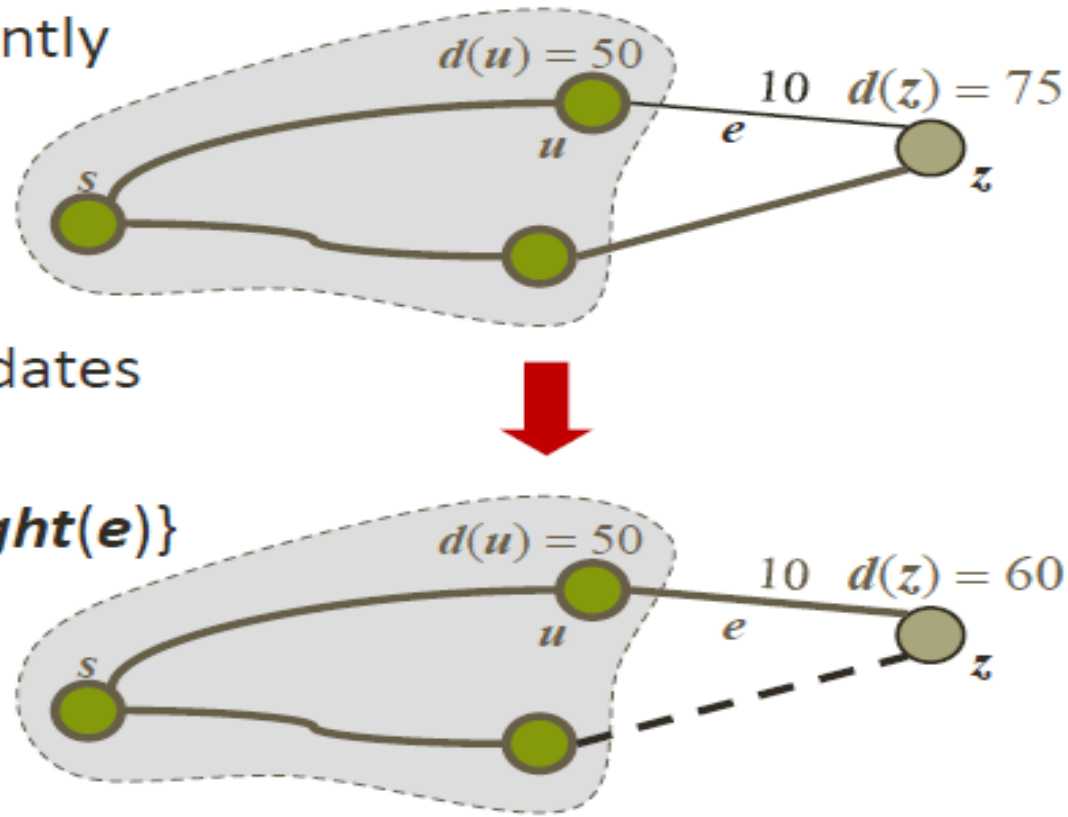
- Dijkstra's algorithm computes distances of all vertices from a given start vertex ***s***

Dijkstra's Algorithm

- We grow a "cloud" of vertices, beginning with ***s*** and eventually covering all vertices
- We store with each vertex ***v*** a label ***d(v)*** representing the distance of ***v*** from ***s*** in the subgraph consisting of the cloud and its adjacent vertices
- At each step
  - We add to the cloud the vertex ***u*** outside the cloud with the smallest distance label, ***d(u)***
  - We update the labels of the vertices adjacent to ***u***

Edge Relaxation

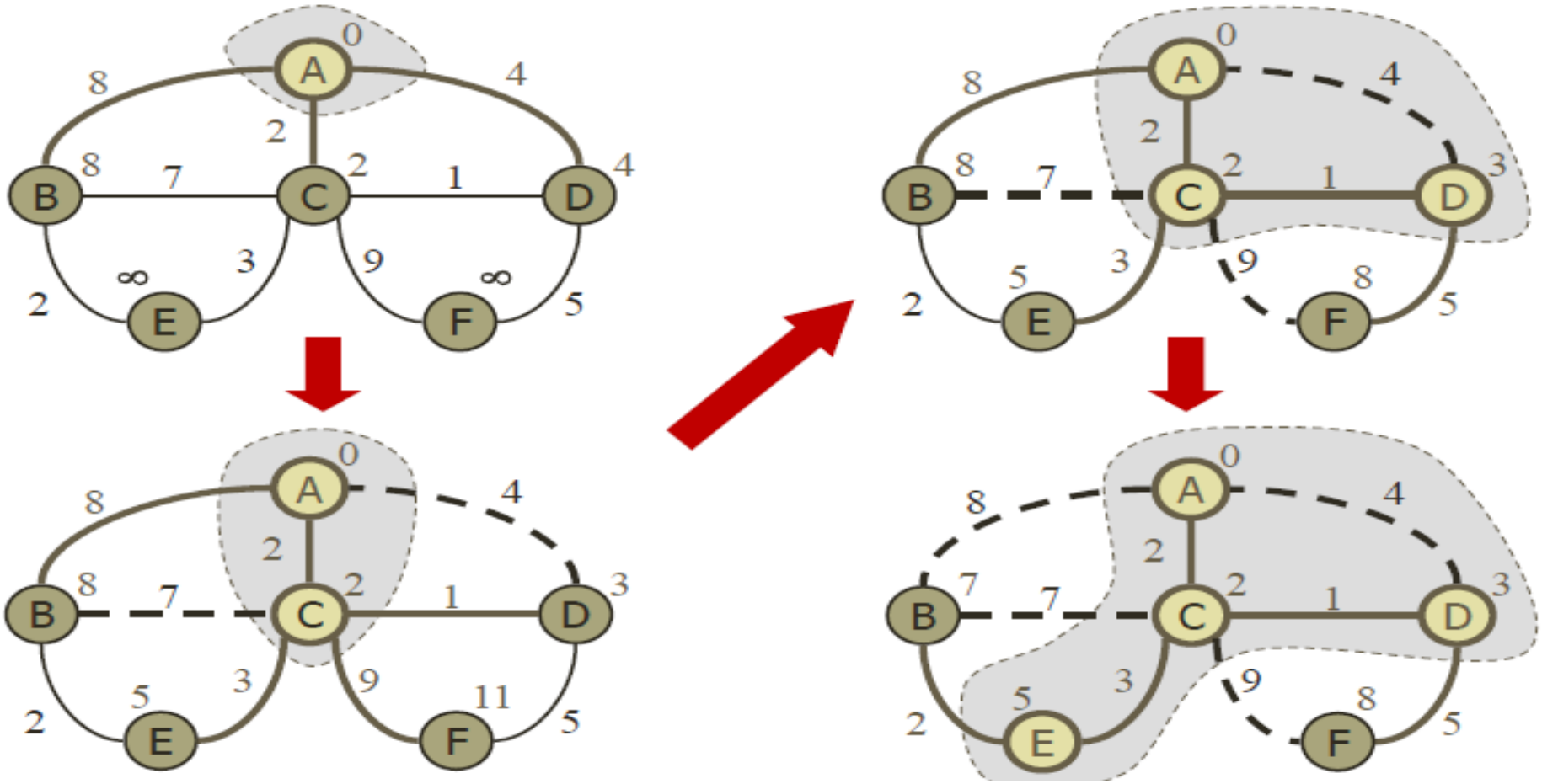
- Consider an edge ***e*** **=** (***u,z***) such that
  - ***u*** is the vertex most recently added to the cloud
  - ***z*** is not in the cloud



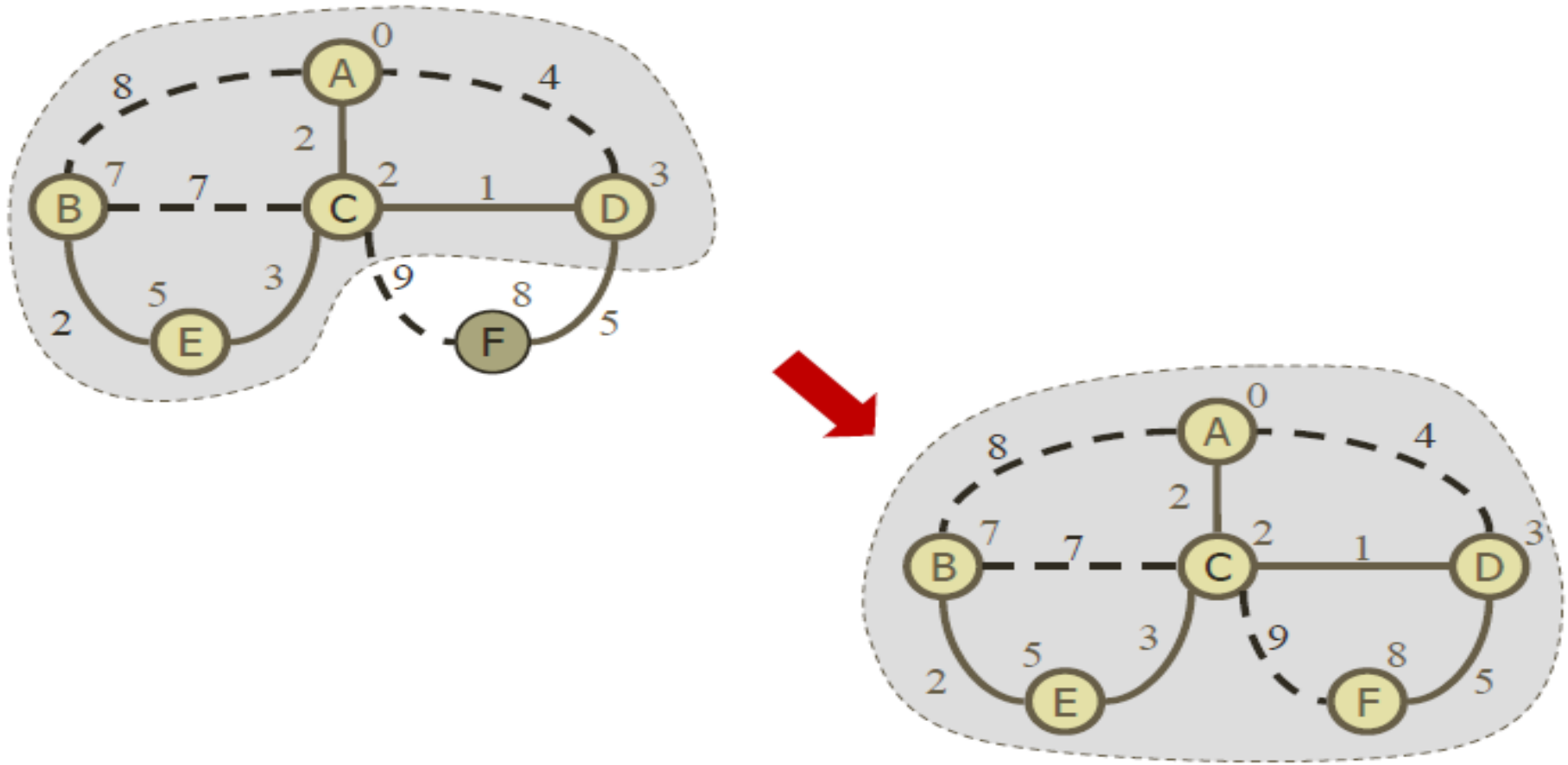
- The relaxation of edge ***e*** updates distance ***d***(***z***) as follows:

  d(***z***) ← min{***d***(***z***), ***d***(***u***) + ***weight***(***e***)}

Example



Example (cont.)



Dijkstra's Algorithm

Let the distance of start vertex from start vertex=0

Let distance of all other vertices from start = ∞ (infinity)

Repeat

- Visit the unvisited vertex with the smallest known distance from start vertex
- For the current vertex, examine its unvisited neighbors
- For the current vertex, calculate distance of each neighbor from start vertex
- If the calculated distance of a vertex is less that the known distance, updated the shortest distance
- Update the previous vertex for each of the updated distances
- Add the current vertex to the list of visited vertices

Until all vertices visited.

Dijkstra's Algorithm

Algorithm ShortestPath(G, s):

Input: A weighted graph G with nonnegative edge weights, and a distinguished vertex s of G.

Output: The length of a shortest path from s to v for each vertex v of G.

Initialize $D[s] = 0$ and $D[v] = \infty$ for each vertex $v \neq s$.

Let a priority queue Q contain all the vertices of G using the D labels as keys.

while Q is not empty **do**

{pull a new vertex u into the cloud}

u = value returned by Q.remove_min()

for each vertex v adjacent to u such that v is in Q **do**

{perform the ***relaxation*** procedure on edge (u, v)}

if $D[u] + w(u, v) < D[v]$ **then**

$D[v] = D[u] + w(u, v)$

Change to $D[v]$ the key of vertex v in Q.

return the label $D[v]$ of each vertex v

Analysis of Dijkstra's Algorithm

- Graph operations: We find all the incident edges once for each vertex
- Label operations:
  - We set/get the distance and locator labels of vertex z $O(\deg(z))$ times
  - Setting/getting a label takes $O(1)$ time
- Priority queue operations:
  - Each vertex is inserted/removed once into/from the priority queue, where each insertion or removal takes $O(\log n)$ time
  - The key of a vertex in the priority queue is modified at most $\deg(w)$ times, where each key change takes $O(\log n)$ time

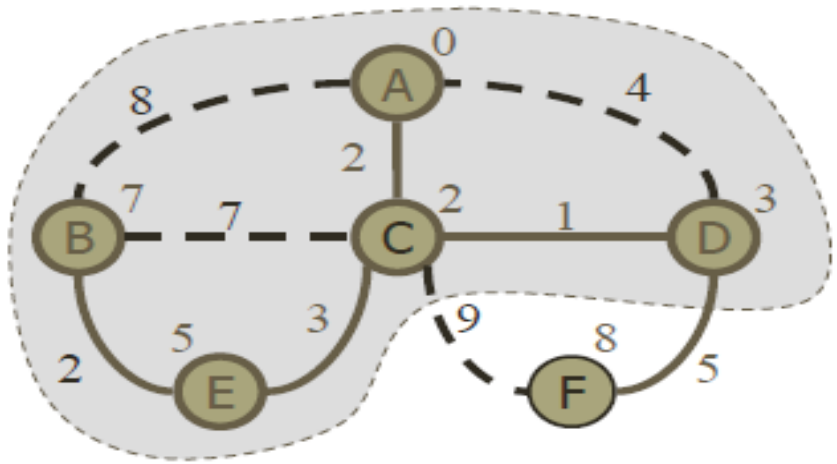
- Dijkstra's algorithm runs in $O((n + m) \log n)$ time provided the graph is represented by the adjacency list/map structure
  - Recall that $\sum_v \deg(v) = 2m$

- So, running time is $O(m \log n)$ since the graph is connected

Why Dijkstra's Algorithm Works

Dijkstra's algorithm is based on the greedy method.
It adds vertices by increasing distance.

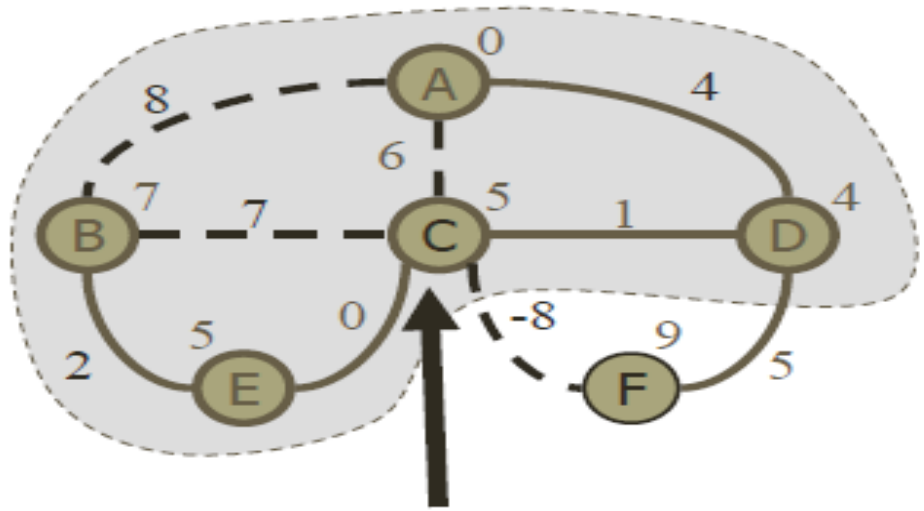
- Suppose it didn't find all shortest distances.
  Let F be the first wrong vertex the algorithm processed.
- When the previous node, D, on the true shortest path was considered, its distance was correct
- But the edge (D,F) was relaxed at that time!
- Thus, so long as $d(F) \geq d(D)$, F's distance cannot be wrong.
  That is, there is no wrong vertex



Problem: Doesn't Work for Negative-Weight Edges

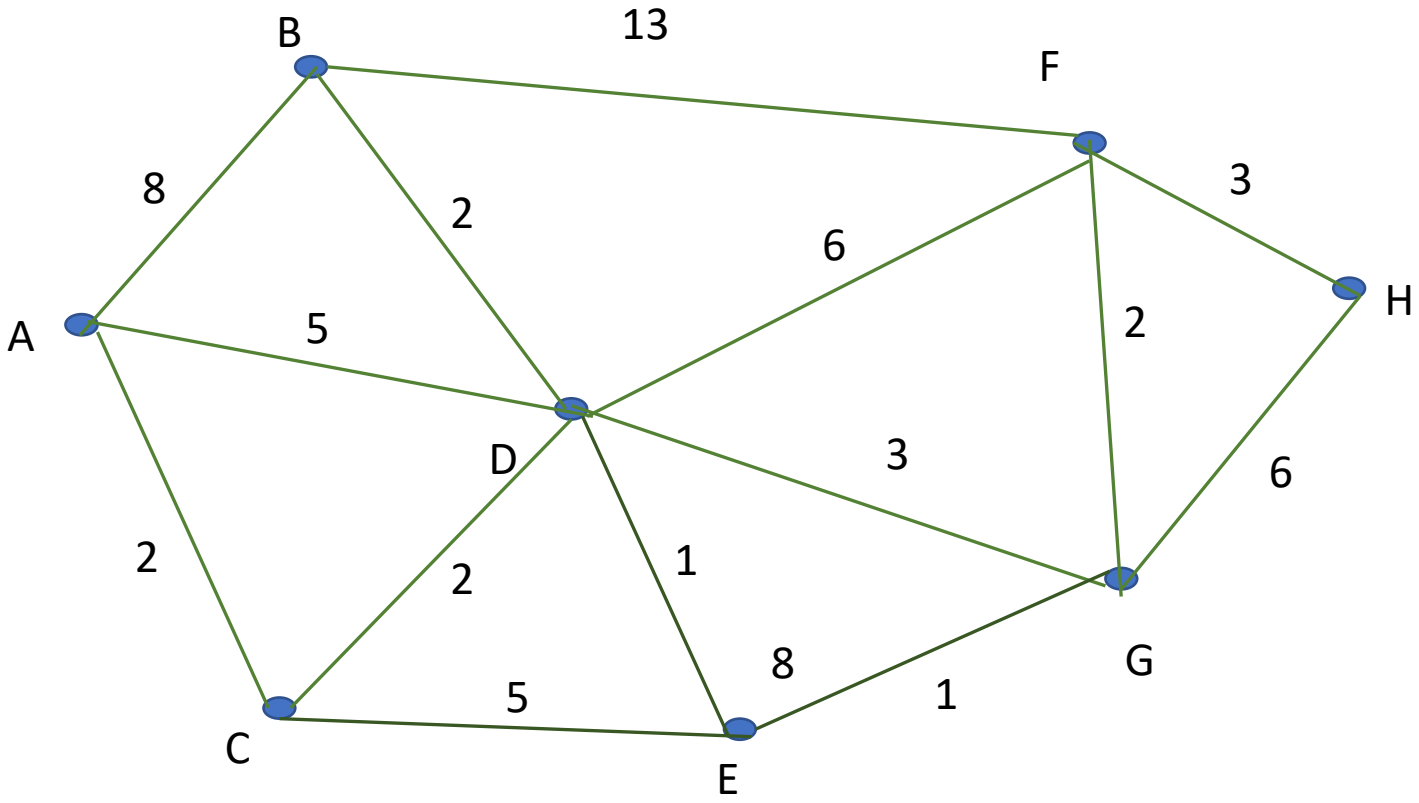
Dijkstra's algorithm is based on the greedy method. It adds vertices by increasing distance.

If a node with a negative incident edge were to be added late to the cloud, it could mess up distances for vertices already in the cloud.



C's true distance is 1, but it is already in the cloud with d(C) = 5!

Example



V	A	B	C	D	E	F	G	H