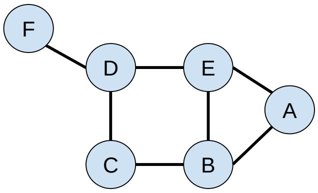
# COMP2421—DATA STRUCTURES AND ALGORITHMS

## Graphs

Dr. Radi Jarrar
Department of Computer Science
Birzeit University

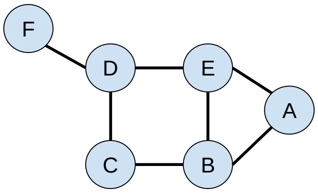
---

## Graphs



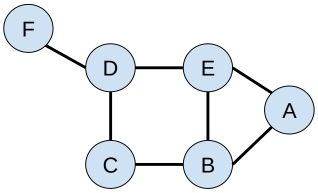
- Graphs are mathematical concepts that have many applications in computer science.
- They have many applications in real-life applications such as social networks, locations and routers in GPS, …
- A graph consists of a finite set of vertices (i.e., nodes) and a set of edges connecting these vertices.
- Two vertices are called adjacent if they are connected to each other by the same edge.

# Graphs



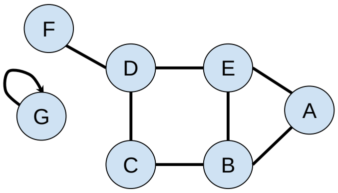
- A graph G=(V, E), is a data structure that consists of a finite set of vertices (or nodes) V, and a set of edges, E.
- Each edge is a pair (v, w) where v and w are nodes from V.

# Graphs



- If the pairs are ordered in the graph, then the graph is called **directed graph**(diagraphs).
- Vertex w is **adjacent** to v if and only if (v, w) ∈ E. In an undirected graph with edge (v, w), and hence (w, v), w is adjacent to v and v is adjacent to w.

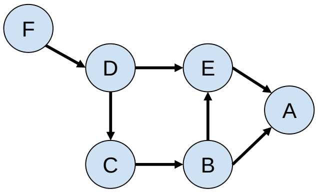
# Graphs - Definitions



- Order: is the number of vertices in a graph
- Size: is the number of edges in a graph
- Vertex degree: is the number of edges that are connected to a vertex
- Isolated vertex: is the vertex that is not connected to any other vertex in the graph
- Self-loop: an edge from a vertex to itself

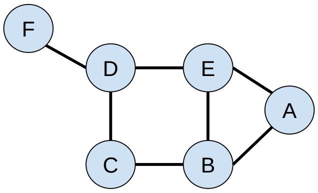
---

# Graphs - Definitions

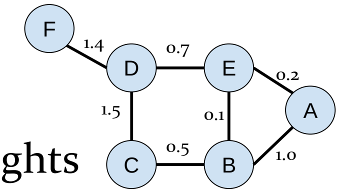


- Directed graph: is a graph where all edges have directions indicating what is the start vertex and what is the end vertex

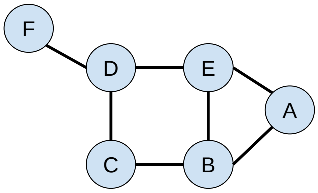
- Undirected graph: is a graph with edges that have no directions



# Graphs - Definitions

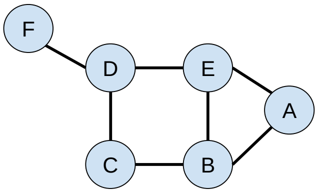


- Weighted graph: edges of a graph have weights
- Unweighted graph: edges of a graph have no weights



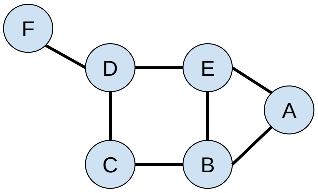
---

# Graphs - Definitions



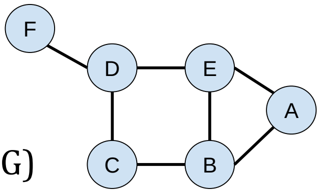
- A **path** in a graph is a sequence of vertices $w_1, w_2, w_3, ..., w_N$, such that $(wi,wi+1) \in E$ for $1 \le i < N$. The **length** of such a path is the number of edges on the path, which is equal to $N – 1$.
- A path from a vertex to itself is allowed. If it does not contain edges, then the path length is 0. If edge (v,v), then the path v (which is also referred to as a loop).
- Cycle: a path $w_1, w_2, w_3, ..., w_N$ for which N > 2, the first N - 1 vertices are all different, and $w_1 = w_N$. For example, the sequence D, E, A, B, C, D is a cycle in the graph above.

# Graphs - Definitions



- A simple path is a path such that all vertices are distinct (except that the first and last might be the same).
- The path v, u, v is cyclic. However, it is not in undirected graph because (v,u) and (u,v) is the same path.

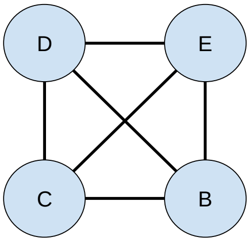
# Graphs - Definitions



- A directed graph is called acyclic if it has no cycles (DAG) - Acyclic directed graph.
- An undirected graph is called connected if there is a path from every node to every other node. A directed graph with this property is called strongly connected.

# Graphs - Definitions

- A complete graph is a graph in which there is an edge between every pair of vertices.



---

# Examples of using graphs

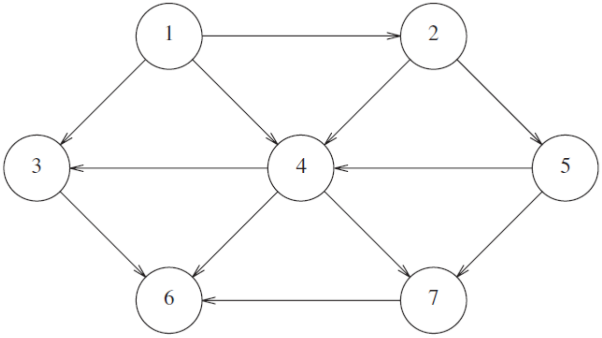
- Airport System
- Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network.
- Graphs are also used in social networks like LinkedIn, Facebook. For example, in Facebook, each person is represented with a vertex(or node). Each node is a structure and contains information like person id, name, gender, and locale.

# REPRESENTATION OF GRAPHS

---

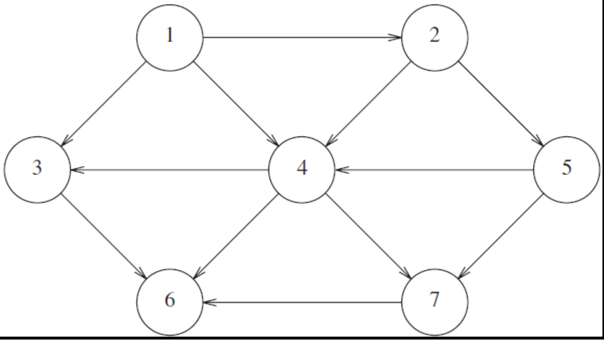
## Graph Representation

- A graph is a data structure that consists of two main components: a finite set of vertices (i.e., nodes); and a finite set of ordered pairs called edges
- Graphs are most commonly represented using
  - Adjacency matrix
  - Adjacency list



# Graph Representation

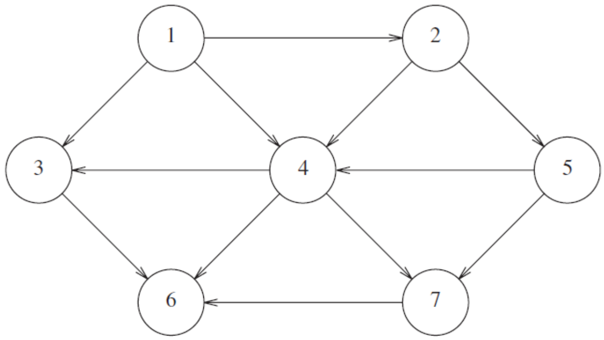
- Consider the following directed graph (the undirected graph is represented the same way)
- Suppose that we can number the vertices starting at 1. This graph has 7 vertices and 12 edges.
- One method is to represent a graph using a 2D array (adjacency matrix)



# Adjacency Matrix

- Adjacency Matrix: maintain a 2D-Boolean array of size v * v where v is the number of vertices in the graph.
- Let the adjacency matrix adj, each edge is represented with the value true: adj[v][w] = true for the edge (v, w)
- The boolean value can be replaced with a weight to represent a weighted graph
- For undirected graph, the adjacency matrix is symmetric

# Adjacency Matrix



| | **1** | **2** | **3** | **4** | **5** | **6** | **7** |
|---|---|---|---|---|---|---|---|
| **1** | | 1 | 1 | 1 | | | |
| **2** | | | | 1 | 1 | | |
| **3** | | | | | | 1 | |
| **4** | | | | | | | 1 |
| **5** | | | | | | | 1 |
| **6** | | | | | | | |
| **7** | | | | | | 1 | |

---

# Adjacency Matrix

# Adjacency Matrix

Advantages:

- Easy to implement and follow
- Removing and edge/checking if an edge exists in the graph takes O(1)

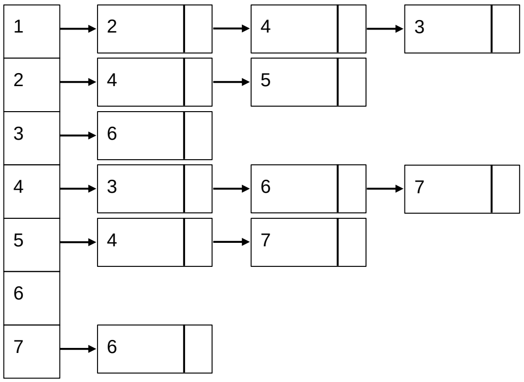
Disadvantages:

- Requires more space $O(n^2)$ if the graph has a few number of edges between vertices
- Adding a vertex will consume $O(n^2)$
- Very slow to iterate over all edges

---

# Adjacency List

- Is a better solution if the graph is sparse (not dense)
- For each vertex, we keep a list of all adjacent vertices
- The space requirement is then O(|E| + |V|), which is linear in the size of the graph



1 → 2 → 4 → 3
2 → 4 → 5
3 → 6
4 → 3 → 6 → 7
5 → 4 → 7
6
7 → 6

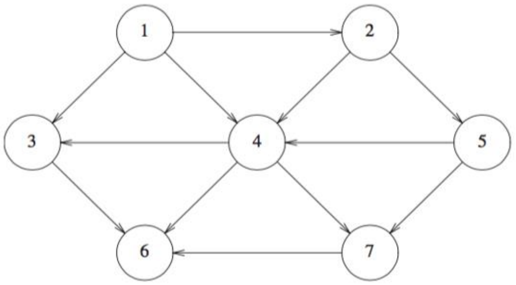


**Figure 9.1** A directed graph

# Adjacency List

- Adjacency lists are the standard way to represent graphs
- Undirected graphs can be similarly represented; each edge (u, v) appears in two lists, so the space usage essentially are doubled
- A common requirement in graph algorithms is to find all vertices adjacent to some given vertex v, and this can be done in time proportional to the number of such vertices found, by a simple scan down the appropriate adjacency list

---

# Adjacency List

Advantages:

- Fast to iterate over all edges
- Fast to add/delete a node (vertix)
- Fast to add a new edge O(1)
- Memory depends more on the number of edges (and less on the number of nodes), which saves more memory if the adjacency matrix is sparse

Disadvantages:

- Finding a specific edge between any two nodes is slightly slower than the matrix O(k); where k is the number of neighbors nodes

# SORTING GRAPHS

---

# Topological Sort

- A linear order of the vertices in a directed graph
- A topological sort is an ordering of vertices in a directed acyclic graph, such that if there is a path from $v_i$ to $v_j$, then $v_j$ appears after $v_i$ in the ordering
- An example is the a directed graph that represents the prerequisite of courses in the figure

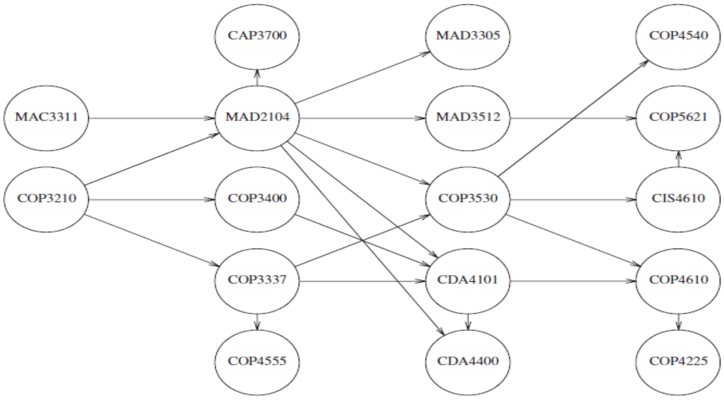
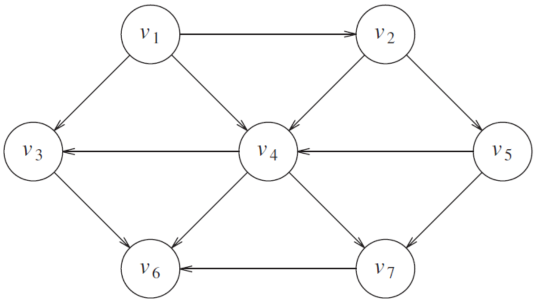


**Figure 9.3** An acyclic graph representing course prerequisite structure

# Topological Sort

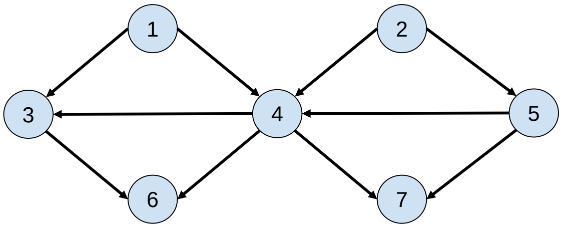
- A directed edge (v, w) indicates that course v must be completed before course w may be attempted
- A topological ordering of these courses is any course sequence that does not violate the prerequisite requirement
- Topological ordering is not possible if the graph has a cycle, since for two vertices v and w on the cycle, v precedes w and w precedes v.
- The ordering is not necessarily unique; any legal ordering will work.
- In this graph, $v_1$, $v_2$, $v_5$, $v_4$, $v_3$, $v_7$, $v_6$ and $v_1$, $v_2$, $v_5$, $v_4$, $v_7$, $v_3$, $v_6$ are both topological orderings.



# Topological Sort

- Main idea: find a vertex with nothing going into it (i.e., Starting point). Write it down. Remove it and go through the other vertices and check for anyone with nothing coming into it. Repeat.
- scan all vertices to find the starting point
- * if edge (A, B) exists, A must precede B in the final order.
- Algorithm:
- Assume indegree is sorted with each node
- Repeat until no nodes remain
  - Choose a node of zero indegree and output it
  - Remove the node and all its edges and update indegree

# Topological Sort - Example

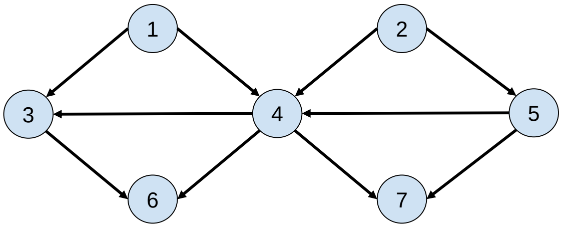


- Indegree:

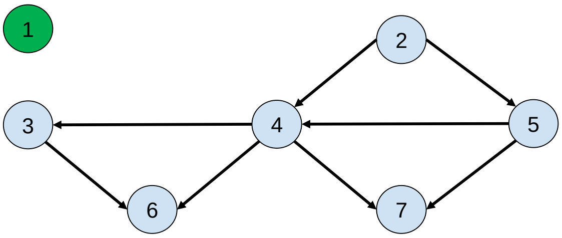
0:
1:
2:
3:

---

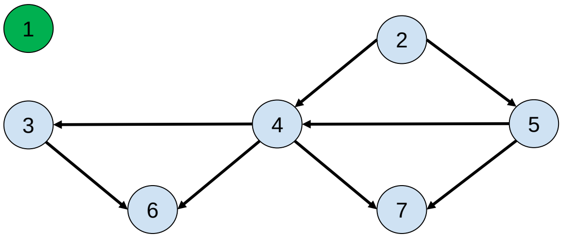
# Topological Sort - Example



- Pick 1 and then update:



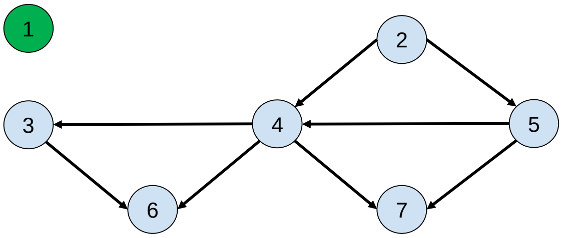
# Topological Sort - Example



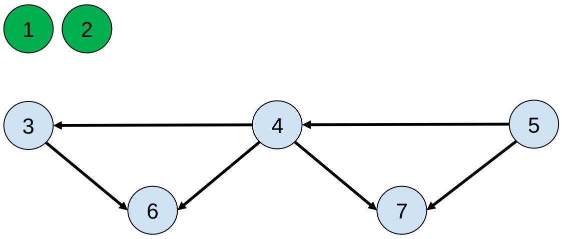
- Indegree:

0:
1:
2:
3:

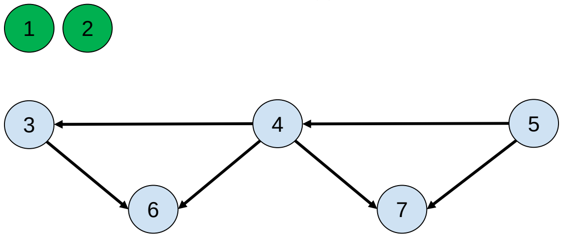
# Topological Sort - Example



- Pick 2 and then update:



# Topological Sort - Example

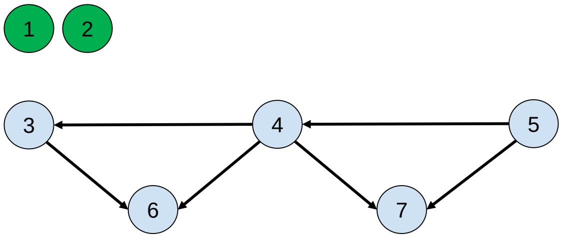


- Indegree:

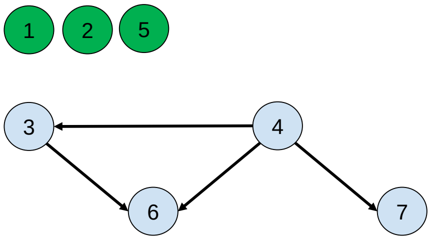
0:
1:
2:
3:

---

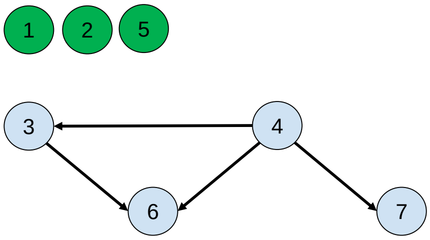
# Topological Sort - Example



- Pick 5 and then update:



# Topological Sort - Example

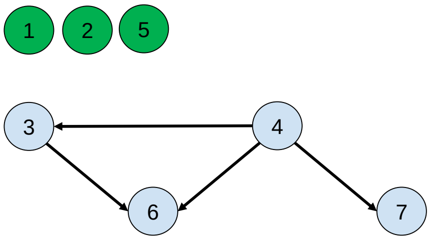


- Indegree:

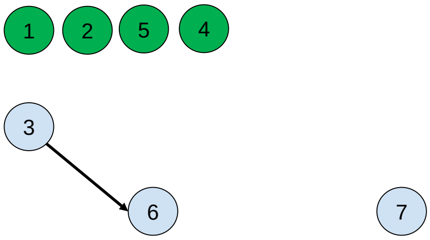
0:
1:
2:
3:

---

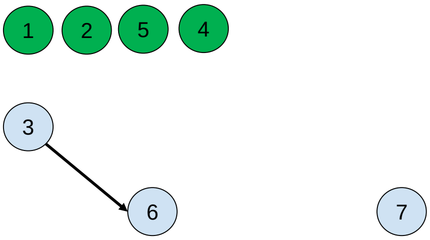
# Topological Sort - Example



- Pick 4 and then update:



# Topological Sort - Example



- Indegree:

0:

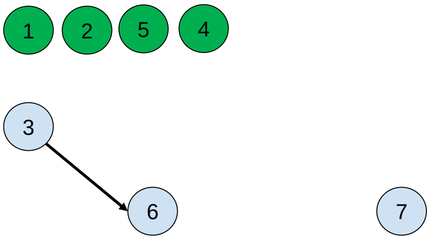
1:

2:

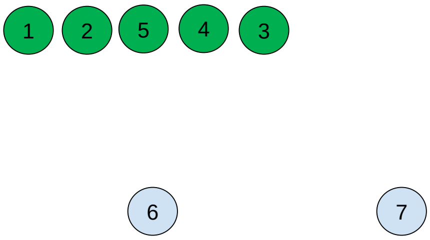
3:

---

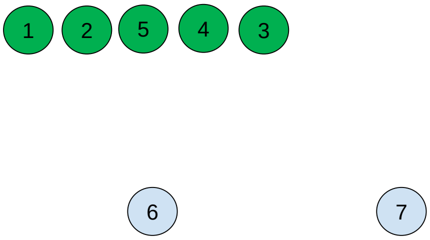
# Topological Sort - Example



- Pick 3 and then update:



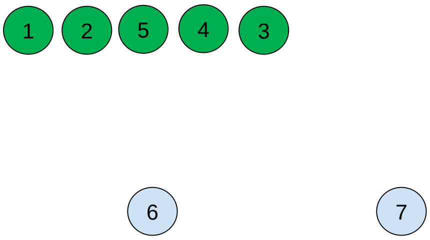
# Topological Sort - Example



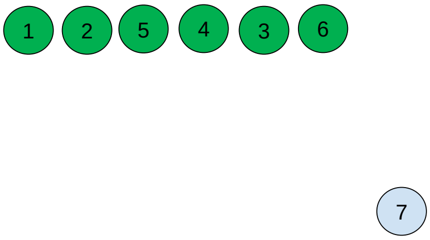
- Indegree:

0:
1:
2:
3:

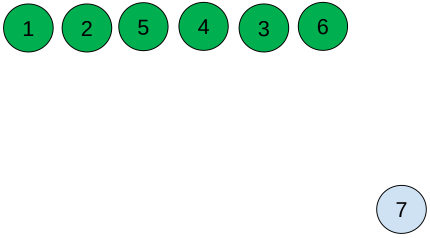
# Topological Sort - Example



- Pick 6 and then update:



# Topological Sort - Example

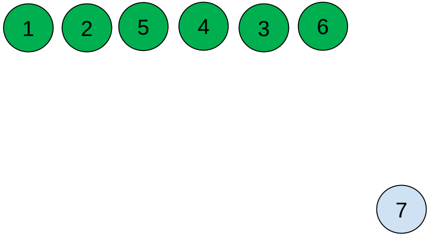


- Indegree:

0:
1:
2:
3:

---

# Topological Sort - Example



- Pick 7 and then update:

# Topological Sort

- First we find the nodes with no predecessors.
- Then, using a queue, we can keep the nodes with no predecessors and on each dequeue we can remove the edges from the node to all other nodes.

# Topological Sort

- **Pseudocode:**

```
1. Represent the graph with two lists on each vertex (incoming edges
   and outgoing edges)
2. Make an empty queue Q;
3. Make an empty topologically sorted list T;
4. Push all items with no predecessors in Q;
5. While Q is not empty
       Dequeue from Q into u;
       Push u in T;
       Remove all outgoing edges from u;
6. Return T;
```

# Topological Sort

- This approach will give us a running time complexity is O(|V| + |E|).
- The problem is that we need additional space and an operational queue.

---

# Topological Sort - Example

# Topological Sort - Example

# Topological Sort - Example

# SEARCH ALGORITHMS

---

## Shortest-Path Algorithms

- Shortest-path algorithms aim at finding the shortest path between nodes in a graph
- The input is a weighted graph: associated with each edge ($v_i$, $v_j$) is a cost $c_{i,j}$ to traverse the edge
- The cost of a path $v_1 v_2 \ldots v_N$ is $\sum_{i=1}^{N-1} c_{i,\, i+1}$
- This is referred to as the **weighted path length**
- The unweighted path length is the number of edges on the path, namely, N – 1

## Shortest-Path Algorithms

- Single-Source shortest path: find the shortest path from a source vertex s to all vertices in a graph
- Single-Destination shortest path: find a shorter path to a given destination vertex d from all vertices in a graph
- Single-Pair shortest path: find the shortest path from a source vertex u to a destination vertex v
- All-Pairs shortest path: find the shortest path from a source vertex u to a destination vertex v for all vertices u and v in the graph

---

## Single-Source Shortest-Path Algorithms

- Given as input a weighted graph, $G = (V, E)$, and a distinguished vertex, $s$, find the shortest weighted path from $s$ to every other vertex in $G$.
- For example, the shortest weighted path from $v_1$ to $v_6$ has a cost of 6 and goes from $v_1$ to $v_4$ to $v_7$ to $v_6$
- The shortest unweighted path between these vertices is 2

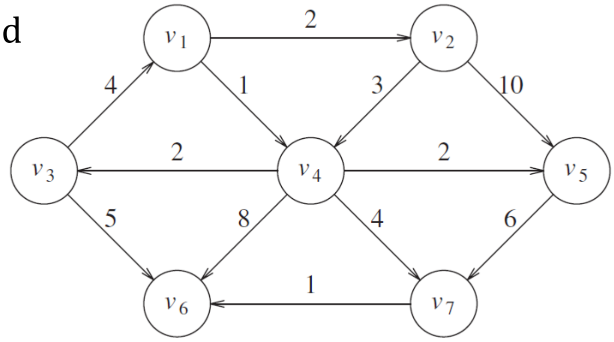


**Figure 9.8** A directed graph $G$

# Single-Source Shortest-Path Algorithms

- The shortest unweighted path between these vertices is 2
- Generally, when it is not specified whether we are referring to a weighted or an unweighted path, the path is weighted if the graph is.

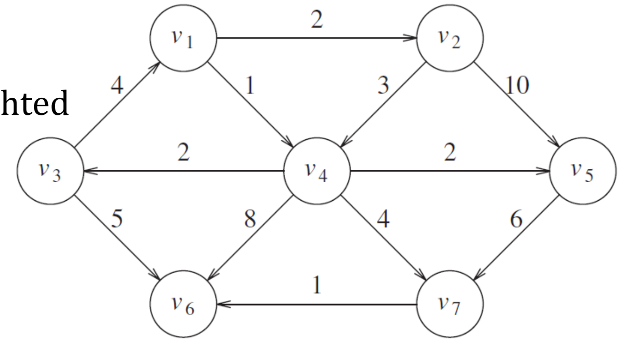


**Figure 9.8** A directed graph *G*

# Single-Source Shortest-Path Algorithms

- Having negative weights in the graph may cause some problems.
- The path from $v_5$ to $v_4$ has cost 1, but a shorter path exists by following the loop $v_5$, $v_4$, $v_2$, $v_5$, $v_4$, which has a cost of −5
- This path is still not the shortest, because we could stay in the loop arbitrarily long.
- Thus, the shortest path between these two points is **undefined**.

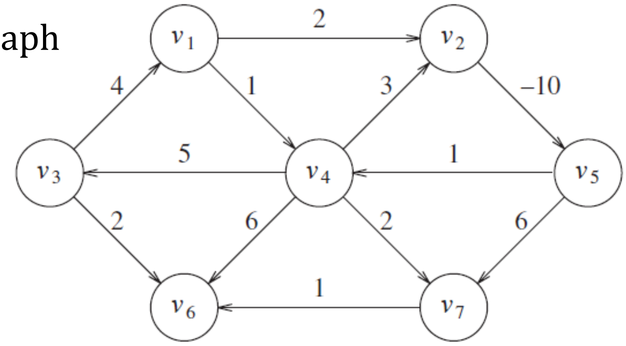


**Figure 9.9** A graph with a negative-cost cycle

# Single-Source Shortest-Path Algorithms

- Another example, the shortest path
- from $v_1$ to $v_6$ is undefined, because we can get into the same loop.
- This loop is known as a **negative-cost cycle;** when one is present in the graph, the shortest paths are not defined.

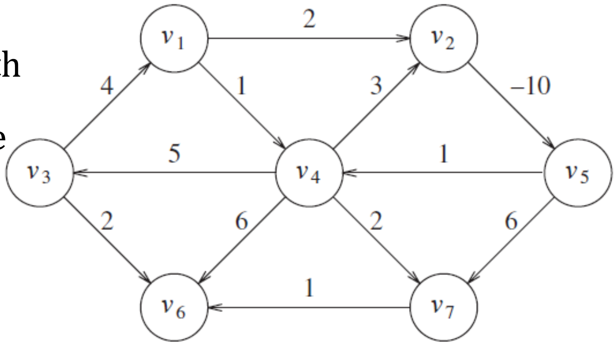


**Figure 9.9** A graph with a negative-cost cycle

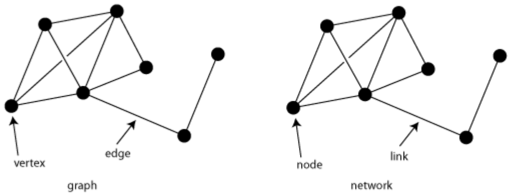
---

# Single-Source Shortest-Path Algorithms

- Negative-cost edges are not necessarily bad, as the cycles are, but their presence seems to make the problem harder.
- For convenience, in the absence of a negative-cost cycle, the shortest path from $s$ to $s$ is zero.

# Single-Source Shortest-Path Algorithms

- There are many examples where we might want to solve the shortest-path problem.
- If the vertices represent computers; the edges represent a link between computers; and the costs represent communication costs (phone bill per megabyte of data), delay costs (number of seconds required to transmit a megabyte), or a combination of these and other factors, then we can use the shortest-path algorithm to find the cheapest way to send electronic news from one computer to a set of other computers.



---

# Single-Source Shortest-Path Algorithms

- Another example is to model an airplane (or transportation routes) by graphs and use a shortest path algorithm to compute the best route between two points.
- In this and many practical applications, we might want to find the shortest path from one vertex, *s*, to only one other vertex, *t*.
- Currently there are no algorithms in which finding the path from *s* to one vertex is any faster (by more than a constant factor) than finding the path from *s* to all vertices.
- We will solve 4 variations of this problem

# Unweighted Shortest Paths

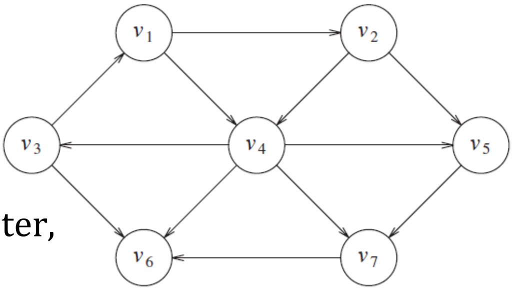


**Figure 9.10** An unweighted directed graph *G*

- Given an unweighted graph, *G*. Using some vertex, *s*, which is an input parameter, we want to find the shortest path from *s* to all other vertices.
- We are only interested in the number of edges contained on the path (because there are no weights).
- This is clearly a special case of the weighted shortest-path problem, since we could assign all edges a weight of 1.

---

# Unweighted Shortest Paths

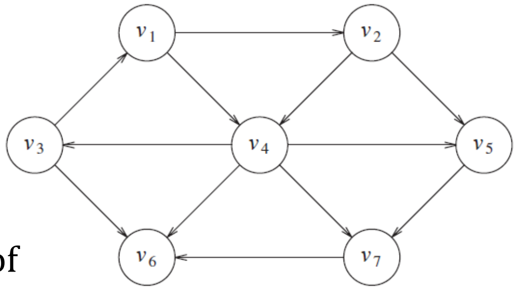


**Figure 9.10** An unweighted directed graph *G*

- Suppose we are interested in the length of the shortest path not in the actual paths themselves. Keeping track of the actual paths will turn out to be a matter of simple bookkeeping.
- Suppose we choose s to be $v_3$.
- Immediately, we can tell that the shortest path from s to $v_3$ is then a path of length 0.
- We can mark this information and then obtain the following graph

# Unweighted Shortest Paths

- Now look for vertices that are distant by 1 from s ($v_3$), which are the adjacent vertices of s.
- $v_1$ and $v_6$ are the adjacent vertices to s.

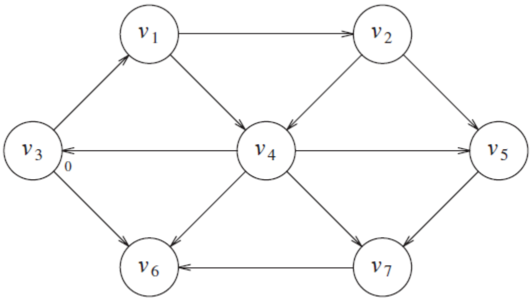


**Figure 9.11** Graph after marking the start node as reachable in zero edges

---

# Unweighted Shortest Paths

- Now find vertices whose shortest path from *s* is exactly 2, by finding all the vertices adjacent to $v_1$ and $v_6$ (the vertices at distance 1).
- $v_2$ and $v_4$ are the adjacent vertices to s.

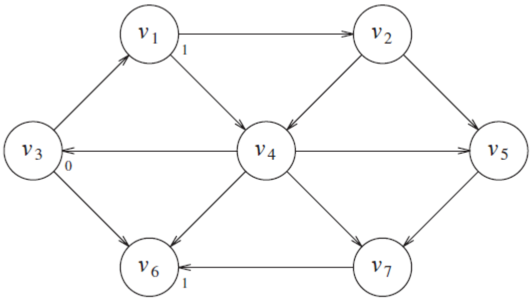


**Figure 9.12** Graph after finding all vertices whose path length from *s* is 1

# Unweighted Shortest Paths

- Finally we can find, by examining vertices adjacent to the recently evaluated $v_2$ and $v_4$, that $v_5$ and $v_7$ have a shortest path of three edges.

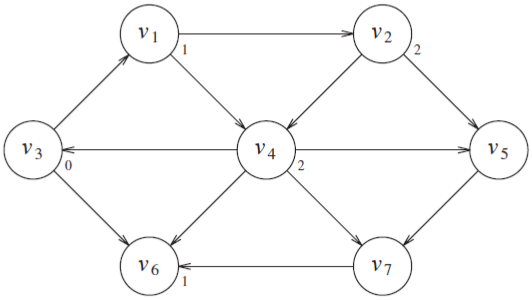


**Figure 9.13** Graph after finding all vertices whose shortest path is 2

---

# Unweighted Shortest Paths

- Now all vertices have been calculated.
- This strategy of searching a graph is known as **Breadth-First Search (BFS)**.
- It operates by processing vertices in layers: The vertices closest to the start are evaluated first, and the most distant vertices are evaluated last.
- This is much the same as a level-order traversal for trees.

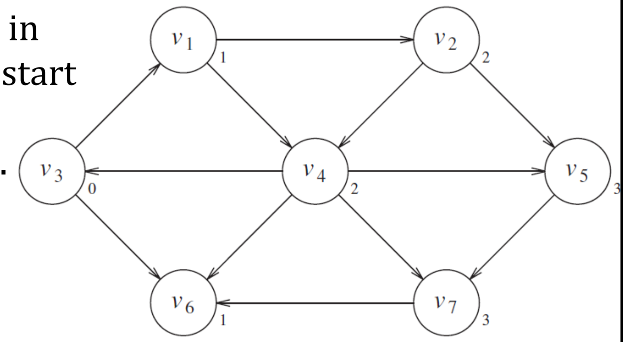


**Figure 9.14** Final shortest paths

# Unweighted Shortest Paths

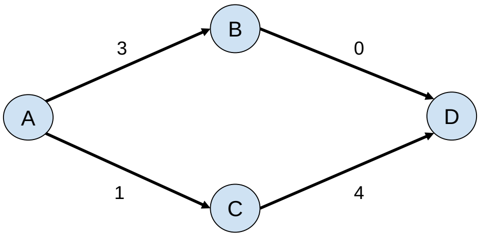
| $v$ | *known* | $d_v$ | $p_v$ |
|---|---|---|---|
| $v_1$ | F | $\infty$ | 0 |
| $v_2$ | F | $\infty$ | 0 |
| $v_3$ | F | 0 | 0 |
| $v_4$ | F | $\infty$ | 0 |
| $v_5$ | F | $\infty$ | 0 |
| $v_6$ | F | $\infty$ | 0 |
| $v_7$ | F | $\infty$ | 0 |

- The BFS can be implemented by adapting the following table
- First, for each vertex, keep its distance from $s$ in the entry $d_v$ (initially all vertices are unreachable except for $s$, whose path length is 0).
- Variable $p_v$ is the bookkeeping variable, which will allow us to print the actual paths.
- Variable known is set to true after a vertex is processed.
- Initially, all entries are not known, including the start vertex.
- When a vertex is marked known, we have a guarantee that no cheaper path will ever be found, and so processing for that vertex is essentially complete

# Dijkstra’s Algorithm

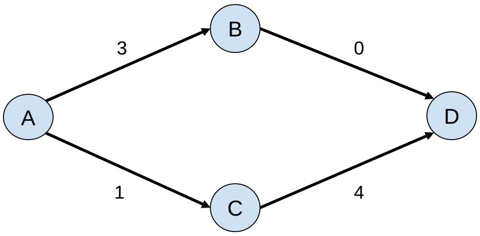
# Dijkstra’s Algorithm

- If the graph is weighted, the problem becomes harder, but we can still use the ideas from the unweighted case.
- Dijkstra’s algorithm solves the problem of finding the shortest path from a vertex (source) to another vertex (destination).
- For example, you want to get from one city to another in the fastest possible way?



# Dijkstra’s Algorithm

- BFS is to find the shortest path between two points.
- “Shortest path” means the path with the fewest segments.
- But in Dijkstra’s algorithm, a weight is assigned to each edge.
- Then Dijkstra’s algorithm finds the path with the smallest total weight.



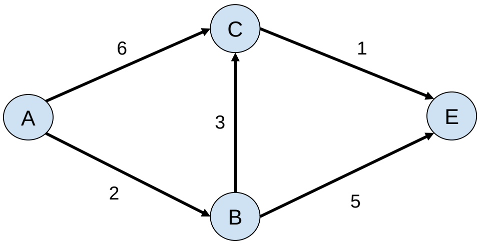
# Dijkstra’s Algorithm

- Dijkstra's algorithm computes shortest paths for positive numbers.
- However, if one allows negative numbers, the algorithm will fail.
- Alternatively, the Bellman-Ford algorithm can be used.
- Dijkstra's algorithm is considered as a prime example of a greedy-search algorithm.
- Greedy algorithms generally solve a problem in stages by doing what appears to be the best thing at each stage.

---

# Dijkstra’s Algorithm

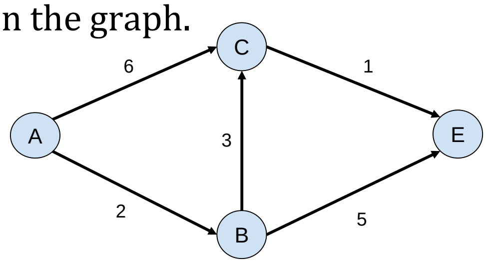
- Dijkstra’s algorithm computes the cost of the shortest path from a starting vertex to all other vertices in the graph.
- Consider the following graph: Starting point ‘A’, destination ‘E’.
- If we run this using the BFS, we will end-up with the cost of 7 (6+1)
- We aim at finding the destination is less time! (if exists)



# Dijkstra's Algorithm

- 4-basic steps for Dijkstra's algorithm:

1. Find the node with the minimal cost. This is the node you can get to in the least amount of time.
2. Update the costs of the neighbor nodes.
3. Repeat until this is done for every node in the graph.
4. Calculate the final path.



---

# Dijkstra's Algorithm

- At each stage:
  - Select an unknown vertex $v$ that has the smallest $d_v$
  - Declare that the shortest path from $s$ to $v$ is known.
  - For each vertex $w$ adjacent to $v$:
    - Set its distance $d_w$ to the $d_v$ + $\text{cost}_{v,w}$
    - Set its path $p_w$ to $v$.

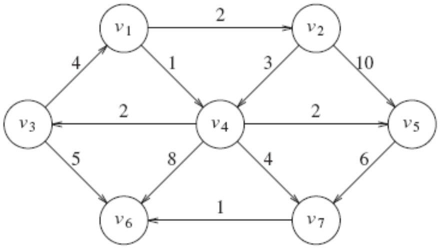
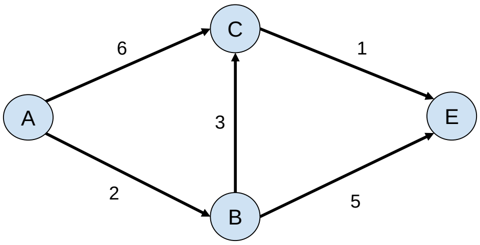


**Figure 9.20** The directed graph $G$ (again)

# Dijkstra’s Algorithm

- **Step 1:** Find the node with the minimal cost.
- We are standing at the starting node ‘A’. ‘B’ will take 6; and ‘C’ will take 2. We don’t know the rest yet.
- As we don’t know how long it will take to reach the destination, we will put it infinity.

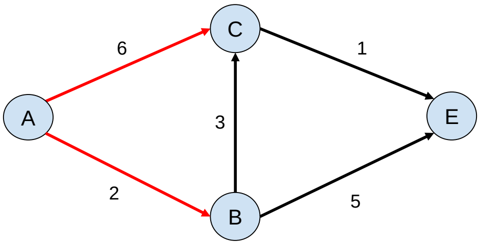


---

# Dijkstra’s Algorithm

- **Step 1:** Find the node with the minimal cost.
- We are standing at the starting node ‘A’. ‘B’ will take 6; and ‘C’ will take 2. We don’t know the rest yet.
- As we don’t know how long it will take to reach the destination, we will put it infinity.

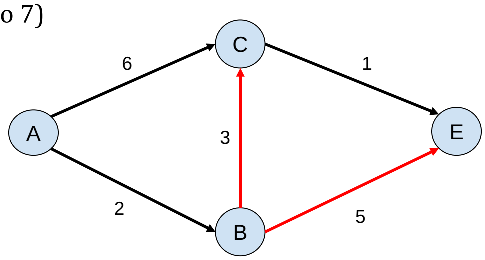
| Node | Cost to Node |
|---|---|
| B | 2 |
| C | 6 |
| E | ∞ |



# Dijkstra's Algorithm

- **Step 2:** Calculate how long it takes to get to all of node B's neighbours by following an edge from B.
- Notice that there is a shorter path to C (2 + 3)
- When there is a shorter path for a neighbor of B, update its cost. In this
- Case
  - A shorter path to C (down from 6 to 5)
  - A shorter path to the destination (down from infinity to 7)

| Node | Cost to Node |
|---|---|
| B | 2 |
| C | ~~6~~ 5 |
| E | ~~∞~~ 7 |
| | |

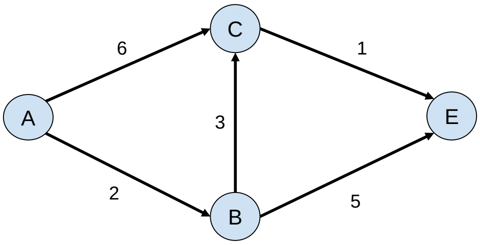


---

# Dijkstra's Algorithm

- **Step 3:** Repeat the steps:
- Step 1 again: Find the node that takes the least cost to get to. We're done with node B, so node C has the next smallest estimate.

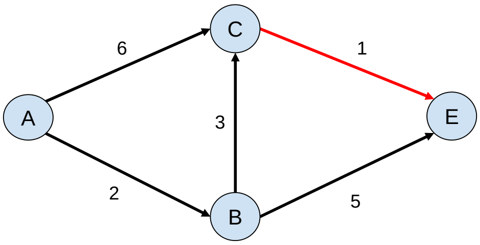
| Node | Cost to Node |
|---|---|
| B | 2 |
| C | 5 |
| E | 7 |



# Dijkstra’s Algorithm

- Step 2 again: Update the cost of C’s neighbours.
- We run Dijkstra’s algorithm for every node (you don’t need to run it
- for the finish node).
- At this point, you know
  - It takes 2 minutes to get to node B.
  - It takes 5 minutes to get to node C.
  - It takes 6 minutes to get to the destination.

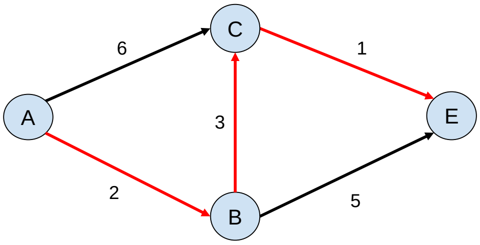
| Node | Cost to Node |
|---|---|
| B | 2 |
| C | 5 |
| E | ~~7~~ 6 |



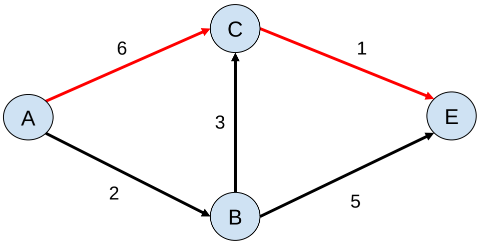
---

# Dijkstra’s Algorithm

- So the final path is

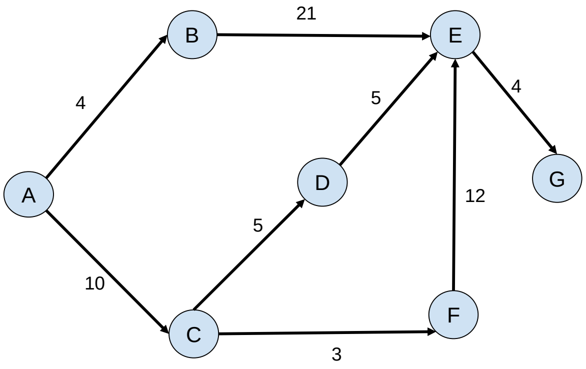


- BFS wouldn’t have found this as the shortest path, because it has three segments.
- And there’s a way to get from the start to the destination in two segments.



# Dijkstra's Algorithm - Example

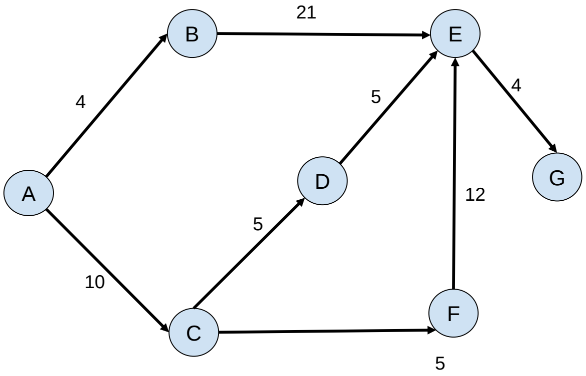
| Node | Cost to Node |
|---|---|
| A | 0 |
| B | ∞ |
| C | ∞ |
| D | ∞ |
| E | ∞ |
| F | ∞ |
| G | ∞ |



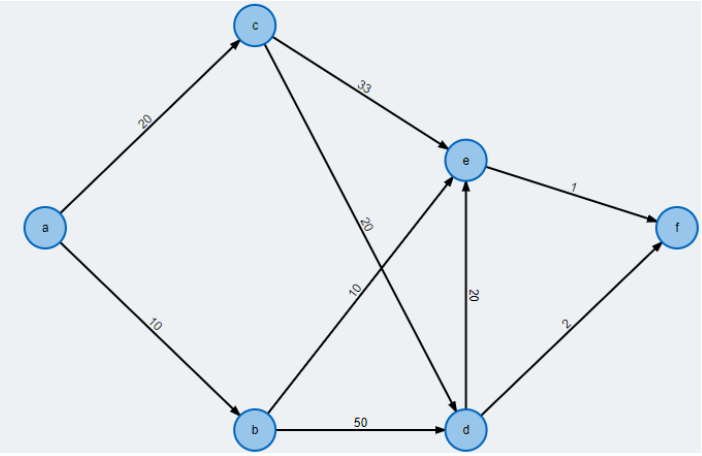
---

# Dijkstra's Algorithm - Example

| Node | Cost to Node |
|---|---|
| A | 0 |
| B | 4 |
| C | 10 |
| D | 15 |
| E | ~~25~~ 20 |
| F | 18 |
| G | ~~∞~~ ~~34~~ 24 |



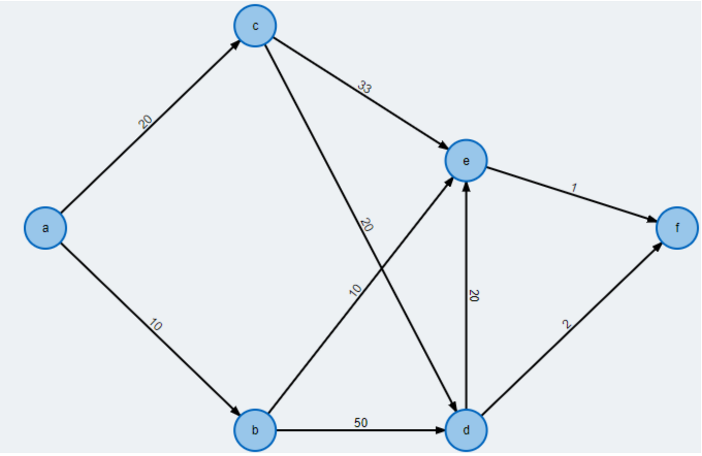
# Dijkstra's Algorithm



| Node | Initial. |
|---|---|
| A | 0 |
| B | ∞ |
| C | ∞ |
| D | ∞ |
| E | ∞ |
| F. | ∞ |

---

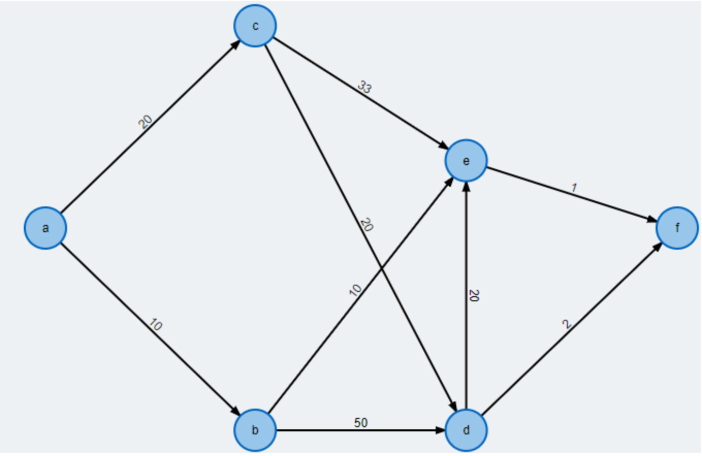
# Dijkstra's Algorithm



| Node | Initial. | Step1 |
|---|---|---|
| A | 0 | 0 |
| B | ∞ | 10 |
| C | ∞ | 20 |
| D | ∞ | ∞ |
| E | ∞ | ∞ |
| F | ∞ | ∞ |

# Dijkstra’s Algorithm

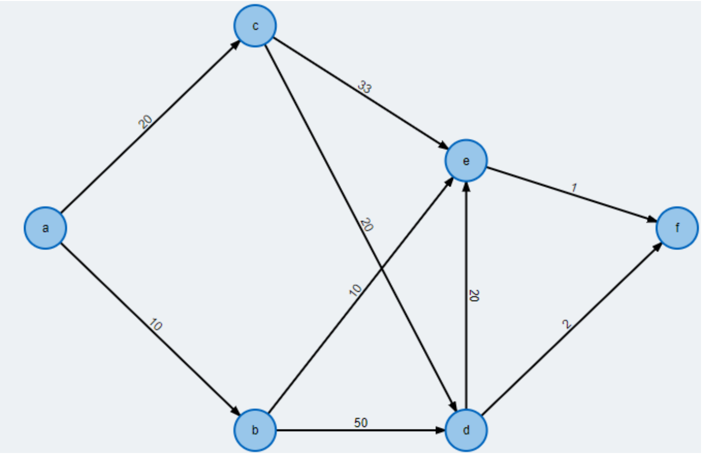
| Node | Initial. | Step1 | Step2 (C) |
|---|---|---|---|
| A | 0 | 0 | 0 |
| B | ∞ | 10 | 10 |
| C | ∞ | 20 | 20 |
| D | ∞ | ∞ | 40 |
| E | ∞ | ∞ | 53 |
| F | ∞ | ∞ | 56 |



---

# Dijkstra’s Algorithm

| Node | Initial. | Step 1 | Step2 (C) | Step3 (B) |
|---|---|---|---|---|
| A | 0 | 0 | 0 | 0 |
| B | ∞ | 10 | 10 | 10 |
| C | ∞ | 20 | 20 | 20 |
| D | ∞ | ∞ | 40 | 40 |
| E | ∞ | ∞ | 53 | 20 |
| F | ∞ | ∞ | 56 | 21 |



# Dijkstra's Algorithm

Maintain 2 sets (arrays) of vertices:

S: a set of vertices whose shortest path from vertex s has been determined

Q: a set of vertices in V-S (uses Heaps)

*keys in Q are estimates of shortest path weights.

---

# Dijkstra's Algorithm

1. Store S in a heap with distance = 0
2. While there are vertices in the queue
    1. Delete Min a vertex v from queue
    2. For all adjacent vertices w:
        1. Compute new distance
        2. Update distance table
        3. Insert/update heap

# Dijkstra's Algorithm - complexity

1. Each vertex is stored in the queue O(V)
2. Delete Min O(V log V)
3. Updating the queue (search and insert) O(log V)
    1. Performed at most for each edge O(E log V)
4. O(E log V + V log V) = O((E + V) log V)

# Graphs with Negative Edge Costs

- If the graph has negative edge costs, then Dijkstra's algorithm does not work.
- Bellman-Ford algorithm solves the single-source shortest path when there may be negative weights in the graph.
- It checks if there is a negative-weight cycle that is reachable from a source vertex
    - If exists; it indicates there is no solution exists
    - If no cycle; then the algorithm produces the shortest paths and their weights

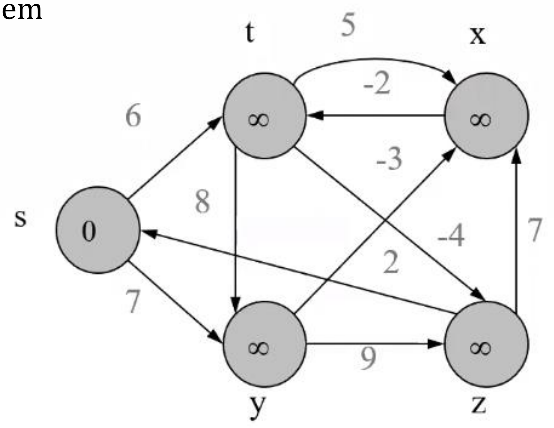
# Graphs with Negative Edge Costs

- N-1 iterations should ensure that the shortest path is reached.
- The run-time is O(V.E)

---

# Graphs with Negative Edge Costs

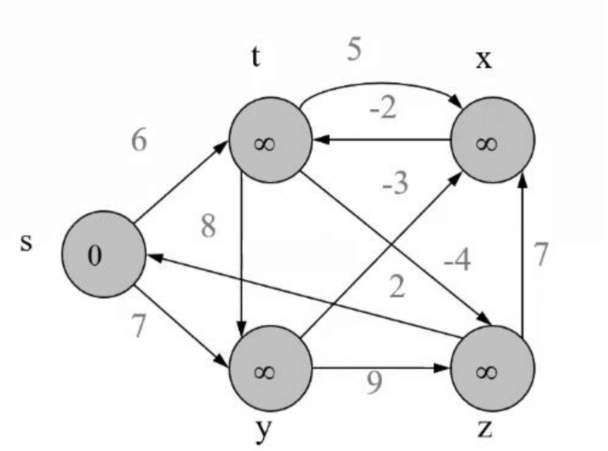
- We will visit all vertices and initialize them
- s is the source node



| Node | Initial. |
|---|---|
| s | 0 |
| t | ∞ |
| y | ∞ |
| x | ∞ |
| z | ∞ |

## Graphs with Negative Edge Costs

- The adjacent of s are y and t.

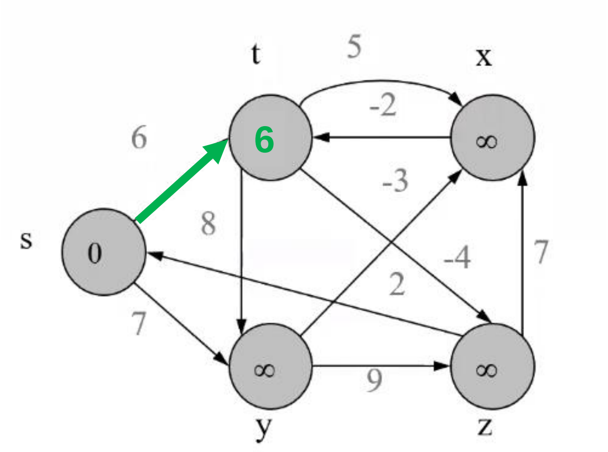


| Node | Initial. | Iter. 1 |
|---|---|---|
| s | 0 | |
| t | ∞ | |
| y | ∞ | |
| x | ∞ | |
| z | ∞ | |

---

## Graphs with Negative Edge Costs

- The adjacent of s are y and t.

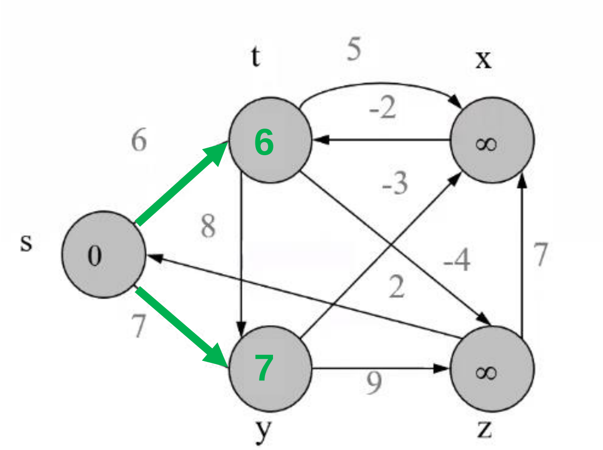


| Node | Initial. | Iter. 1 |
|---|---|---|
| s | 0 | |
| t | ∞ | 6 |
| y | ∞ | |
| x | ∞ | |
| z | ∞ | |

## Graphs with Negative Edge Costs

- The adjacent of s are y and t.

| Node | Initial. | Iter. 1 |
|---|---|---|
| s | 0 | |
| t | ∞ | 6 |
| y | ∞ | 7 |
| x | ∞ | |
| z | ∞ | |

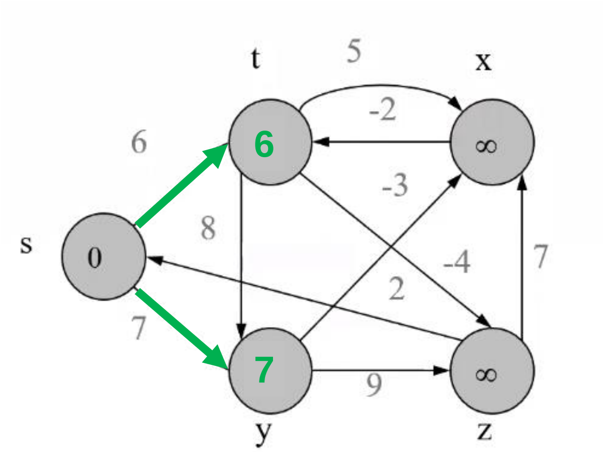


---

## Graphs with Negative Edge Costs

- Now we can reach x & z.
- We will check for all edges.
- Check for X

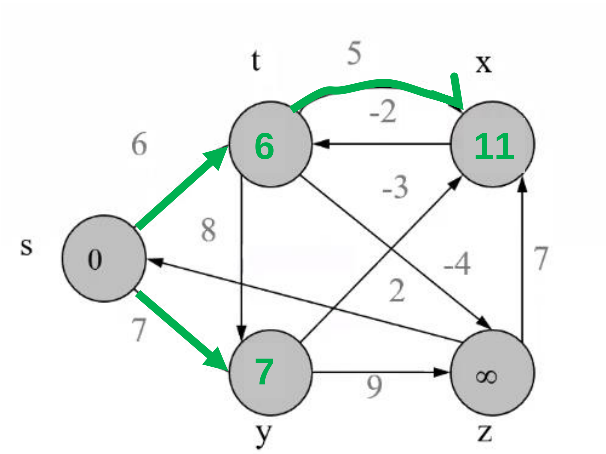
| Node | Initial. | Iter. 1 | Iter. 2 |
|---|---|---|---|
| s | 0 | 0 | |
| t | ∞ | 6 | |
| y | ∞ | 7 | |
| x | ∞ | ∞ | |
| z | ∞ | ∞ | |



# Graphs with Negative Edge Costs

- Now we can reach x & z.
- We will check for all edges.
- Check for X

| Node | Initial. | Iter. 1 | Iter. 2 |
|---|---|---|---|
| s | 0 | 0 | |
| t | ∞ | 6 | 6 |
| y | ∞ | 7 | 7 |
| x | ∞ | ∞ | 11 |
| z | ∞ | ∞ | |

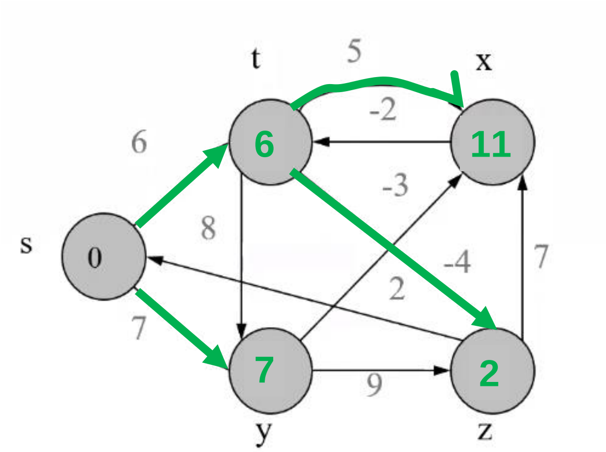


---

# Graphs with Negative Edge Costs

- Now we can reach x & z.
- We will check for all edges.
- Check for X

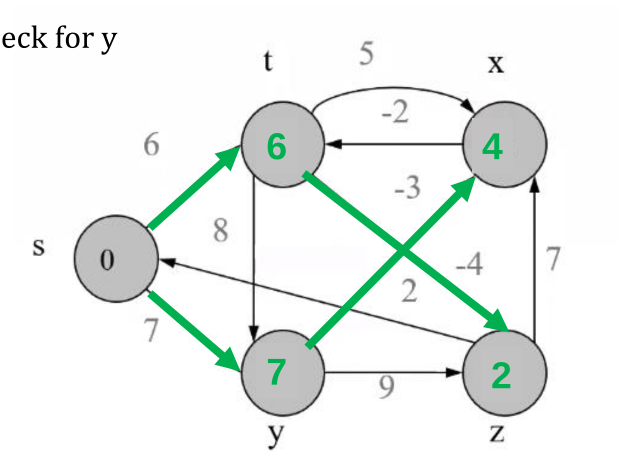
| Node | Initial. | Iter. 1 | Iter. 2 |
|---|---|---|---|
| s | 0 | 0 | 0 |
| t | ∞ | 6 | 6 |
| y | ∞ | 7 | 7 |
| x | ∞ | ∞ | 11 |
| z | ∞ | ∞ | 2 |



# Graphs with Negative Edge Costs

- Now we are done with t, we have to check for y
- y to z = 16. y to x = 4.

| Node | Initial. | Iter. 1 | Iter. 2 |
|---|---|---|---|
| s | 0 | 0 | 0 |
| t | ∞ | 6 | 6 |
| y | ∞ | 7 | 7 |
| x | ∞ | ∞ | ~~11~~ 4 |
| z | ∞ | ∞ | 2 |

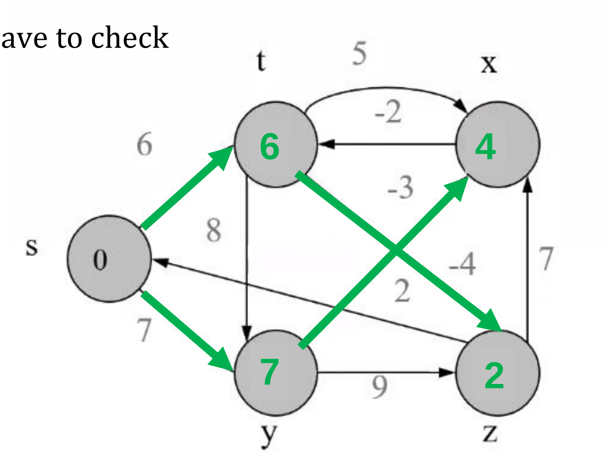


---

# Graphs with Negative Edge Costs

- After the new update on the edge, we have to check for all edges if there is a shorter path.
- We can find x->t

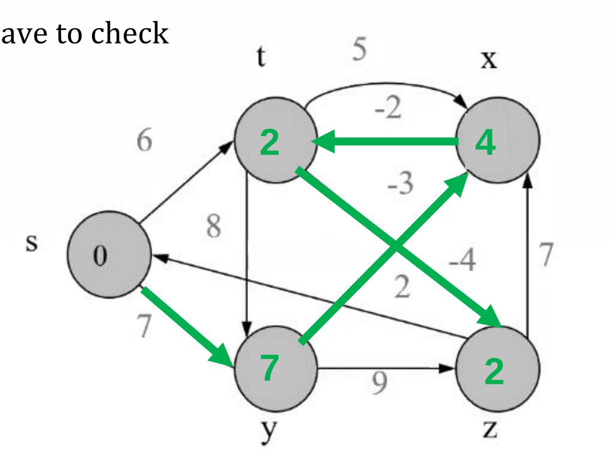
| Node | Initial. | Iter. 1 | Iter. 2 | Iter. 2 |
|---|---|---|---|---|
| s | 0 | 0 | 0 | |
| t | ∞ | 6 | 6 | |
| y | ∞ | 7 | 7 | |
| x | ∞ | ∞ | 4 | |
| z | ∞ | ∞ | 2 | |



# Graphs with Negative Edge Costs

- After the new update on the edge, we have to check for all edges if there is a shorter path.
- We can find x->t

| Node | Initial. | Iter. 1 | Iter. 2 | Iter. 3 |
|---|---|---|---|---|
| s | 0 | 0 | 0 | 0 |
| t | ∞ | 6 | 6 | ~~6~~ 2 |
| y | ∞ | 7 | 7 | 7 |
| x | ∞ | ∞ | 4 | 4 |
| z | ∞ | ∞ | 2 | 2 |

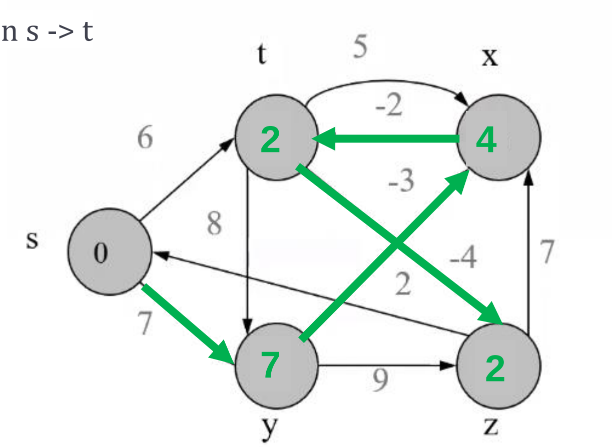


---

# Graphs with Negative Edge Costs

from s -> y -> x -> t gives a shorter cost than s -> t

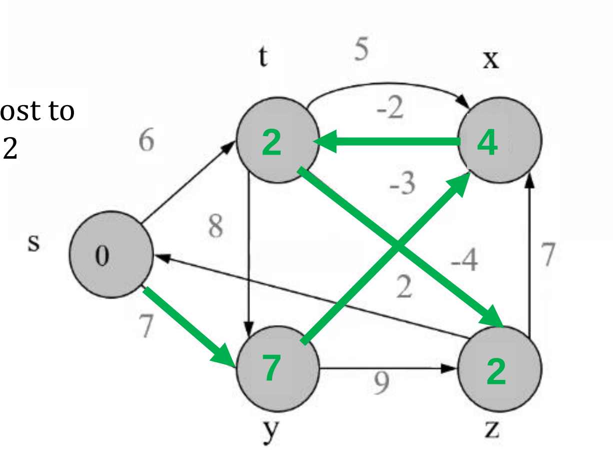
| Node | Initial. | Iter. 1 | Iter. 2 | Iter. 3 |
|---|---|---|---|---|
| s | 0 | 0 | 0 | 0 |
| t | ∞ | 6 | 6 | 2 |
| y | ∞ | 7 | 7 | 7 |
| x | ∞ | ∞ | 4 | 4 |
| z | ∞ | ∞ | 2 | 2 |



# Graphs with Negative Edge Costs

Iteration 4
We check for all vertices
We can notice a change in t -> z (the new cost to reach t is 2, and from t -> z = -4) = 2+ -4 = -2

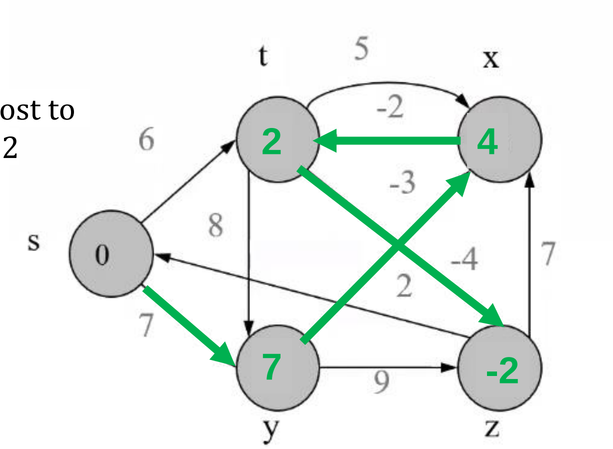
| Node | Initial. | Iter. 1 | Iter. 2 | Iter. 3 | Iter. 4 |
|---|---|---|---|---|---|
| s | 0 | 0 | 0 | 0 | |
| t | ∞ | 6 | 6 | 2 | |
| y | ∞ | 7 | 7 | 7 | |
| x | ∞ | ∞ | 4 | 4 | |
| z | ∞ | ∞ | 2 | 2 | |



# Graphs with Negative Edge Costs

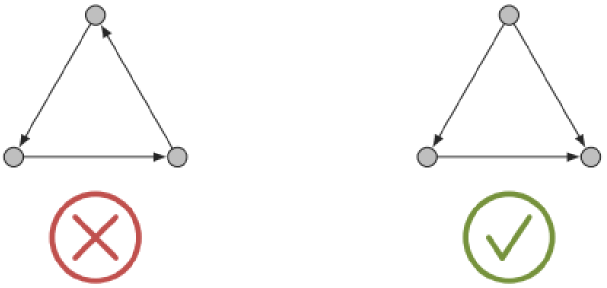
Iteration 4
We check for all vertices
We can notice a change in t -> z (the new cost to reach t is 2, and from t -> z = -4) = 2+ -4 = -2

| Node | Initial. | Iter. 1 | Iter. 2 | Iter. 3 | Iter. 4 |
|---|---|---|---|---|---|
| s | 0 | 0 | 0 | 0 | 0 |
| t | ∞ | 6 | 6 | 2 | 2 |
| y | ∞ | 7 | 7 | 7 | 7 |
| x | ∞ | ∞ | 4 | 4 | 4 |
| z | ∞ | ∞ | 2 | 2 | -2 |



# Acyclic Graphs

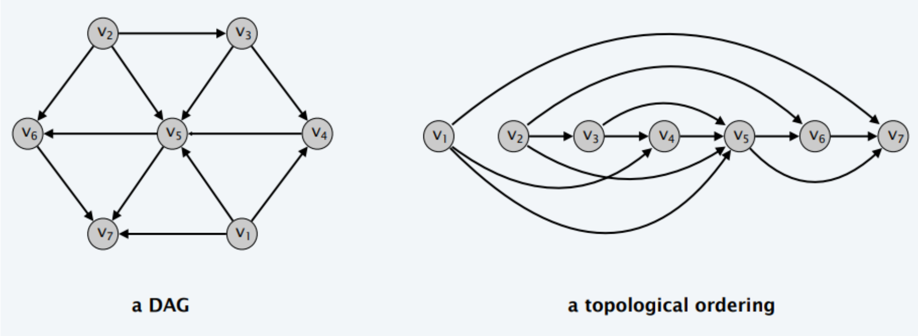
- We want to find the shortest path in acyclic graph (Directed Acyclic Graph)
- DAG contains no cycles



---

# Acyclic Graphs

- If the graph is acyclic, we can use Bellman-Ford, but it takes O(VE)
- A better solution is to use Topological sort:
  - Initialize distances to all vertices as infinite and distance to source as 0
  - Then find a topological sorting of the graph



a DAG

a topological ordering

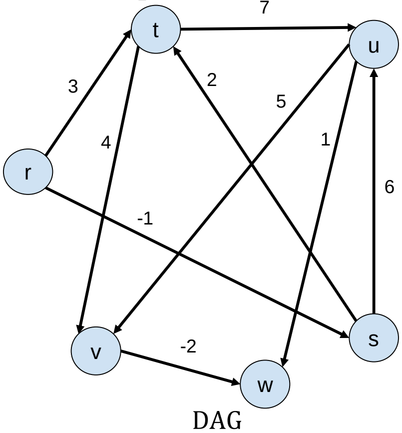
# Acyclic Graphs

- Precedence constraints: Edge $(v_i, v_j)$ means task $v_i$ must occur before $v_j$
- Examples of DAG
- Course prerequisite graph: course $v_i$ must be taken before $v_j$
- Compilation: module $v_i$ must be compiled before $v_j$
- Pipeline of computing jobs: output of job $v_i$ needed to determine input of job $v_j$

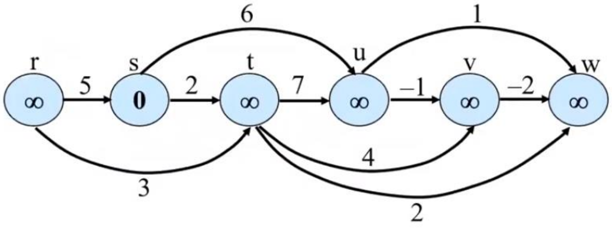
---

# Acyclic Graphs

- Topological sort represents a linear ordering of a graph
- Example



DAG



Topologically sorted

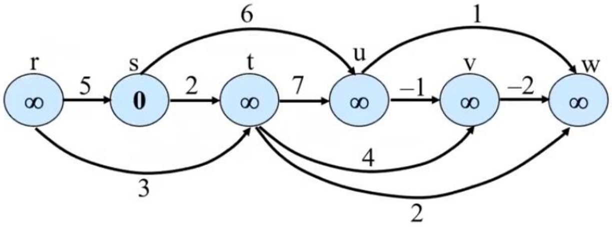
# Acyclic Graphs

- The idea: process vertices on each shortest path from left to right
- Every path in DAG is a subsequence of topologically sorted vertex order. So processing vertices in that order will do each path in forward order
- Just one pass.
- Time complexity O(V + E)

---

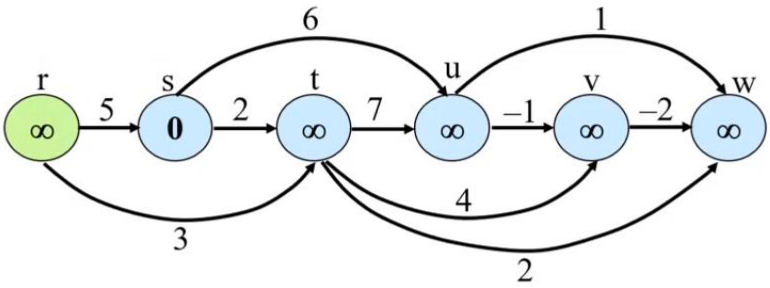
# Acyclic Graphs

- Topologically sorted graph
- Now we have vertex s as the source
- We want to find the shortest path from s to all vertices



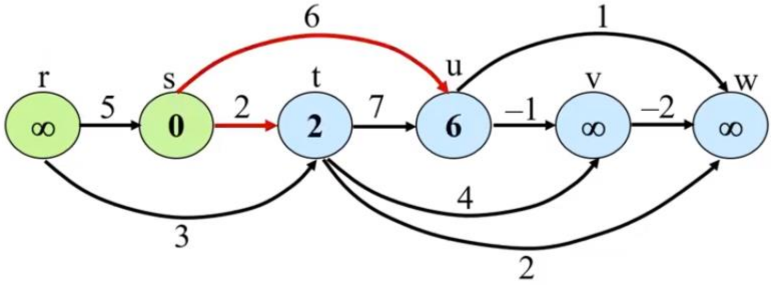
- Start with r, what is the path from s to r?
  - There is no path (infinity)

# Acyclic Graphs



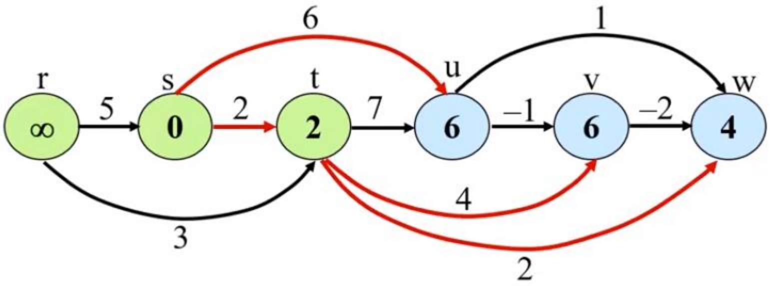
- So the first iteration, r = ∞
- Now the second pass
- Take the adjacent of s. From s to t =2, which is less than ∞, so update t and the predecessor is s
- From s to u is the same, 7 is less than ∞, so update u and the predecessor is s

# Acyclic Graphs



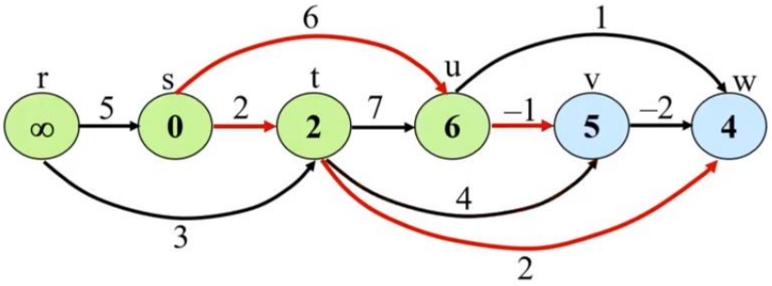
- Next iteration, check the adjacents of t
- From t to v is 2+4 = 6 which is less than ∞, so update v and the predecessor is t
- From t to w is 2+2 = 4 which is less than ∞, so update w and the predecessor is t

# Acyclic Graphs



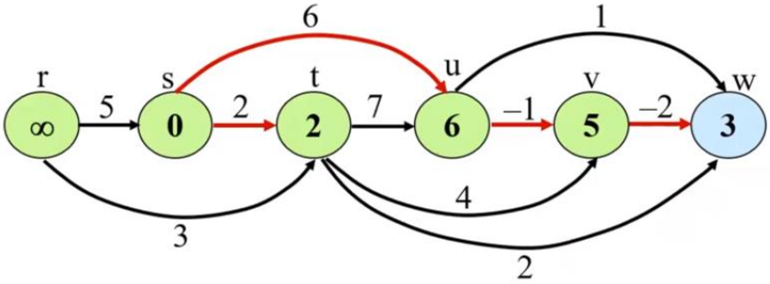
- Next iteration, check the adjacents of u
- From u to v is 6 + -1 = 5 which is less than 6, so update v and the predecessor is u
- From u to w is 6 + 1 = 7 which is more than 4, so no updates

# Acyclic Graphs



- Next iteration, check the adjacents of v
- From v to w is 5 + -2 = 3 which is less than 4, so update w and the predecessor is v instead of t

# Acyclic Graphs



- We are left with 1 iteration for w
- Notice that w has no adjacents
- Thus we reached the shortest path from the source s