# COMP2421—DATA STRUCTURES AND ALGORITHMS

**Graphs**

Dr. Radi Jarrar
Department of Computer Science
Birzeit University

# 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 $v_6$

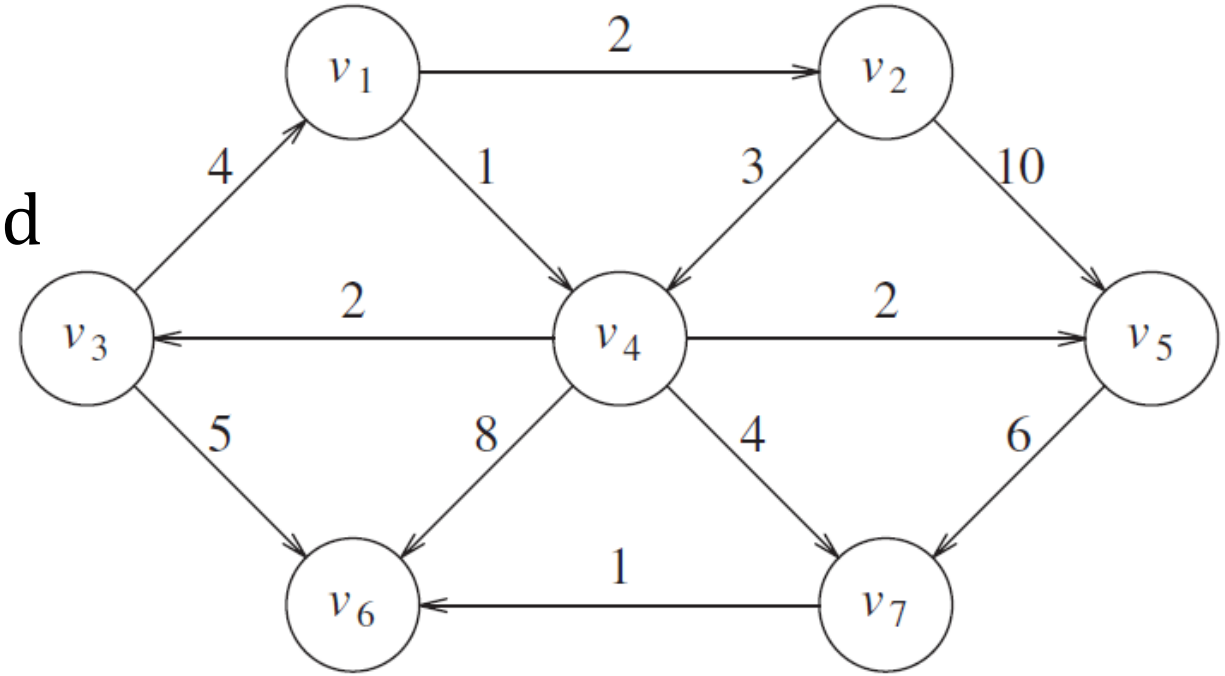



**Figure 9.8** A directed graph $G$

# Single-Source Shortest-Path Algorithms

- The shortest unweighted path between these vertices is 2

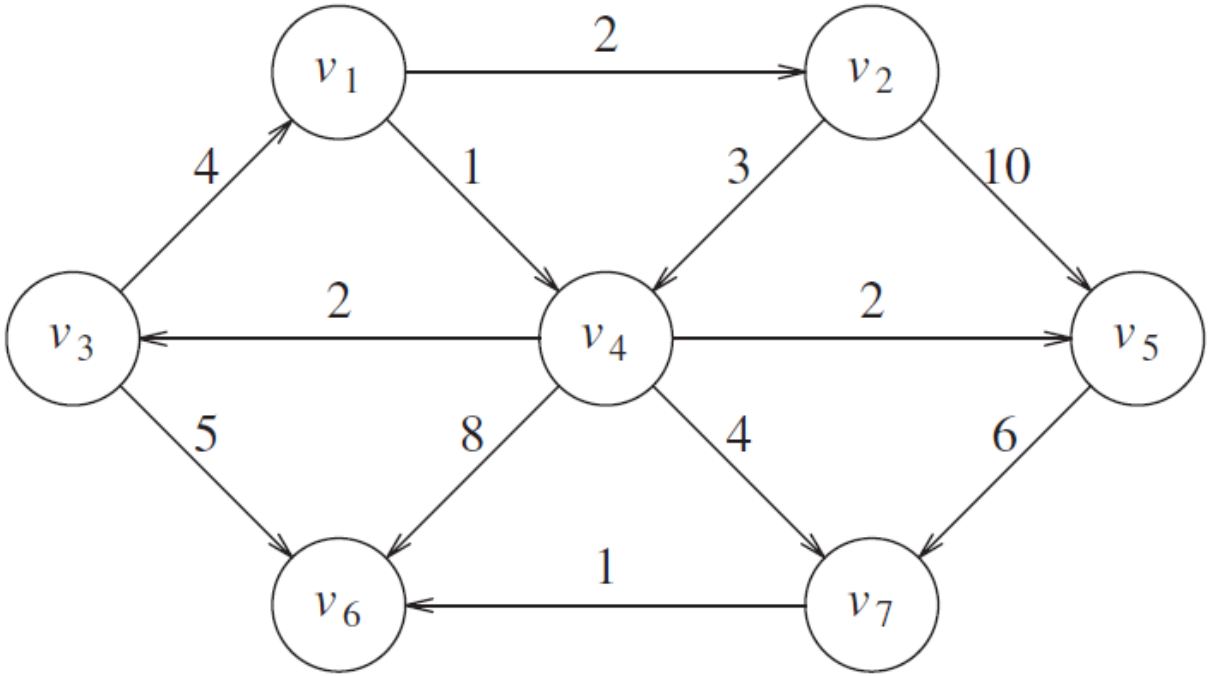



**Figure 9.8** A directed graph G

# 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**.

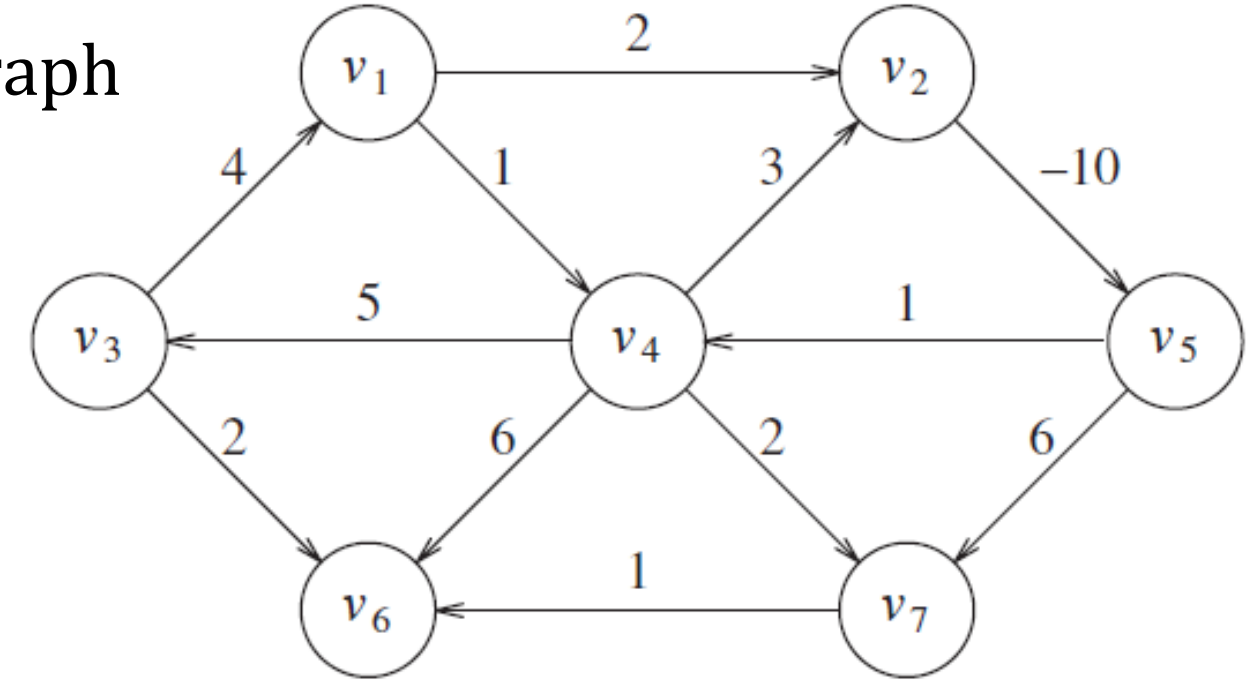



**Figure 9.9** A graph with a negative-cost cycle

# 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.

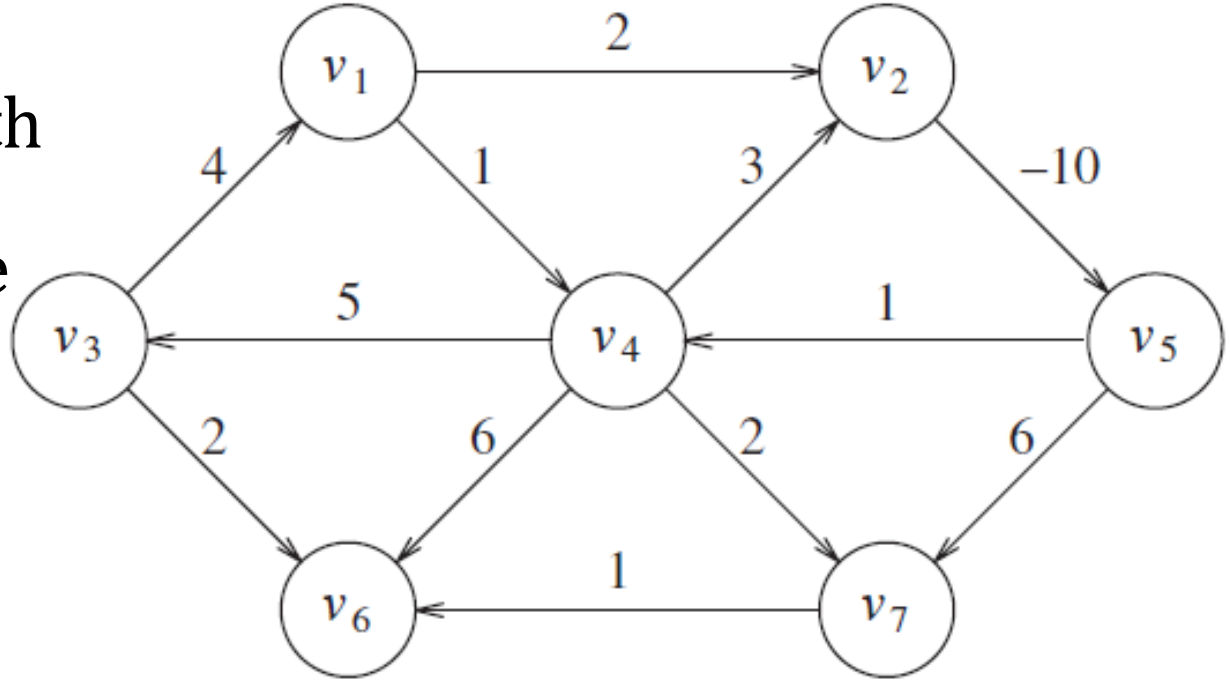


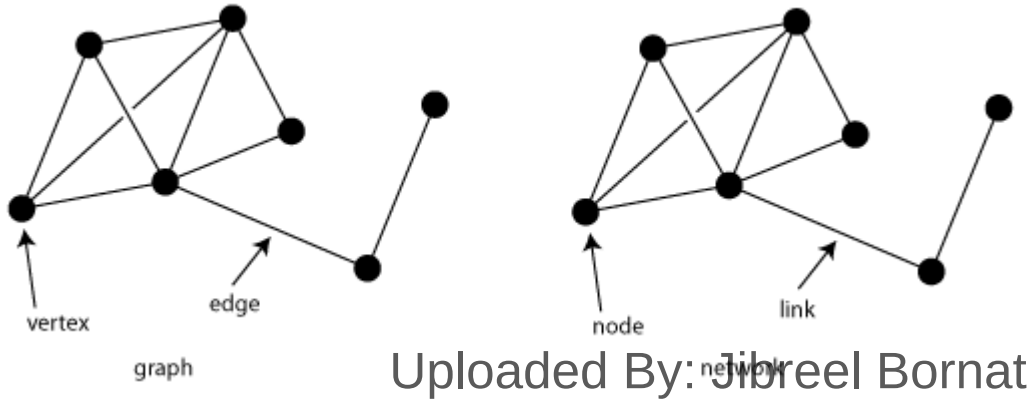

**Figure 9.9** A graph with a negative-cost cycle

# 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

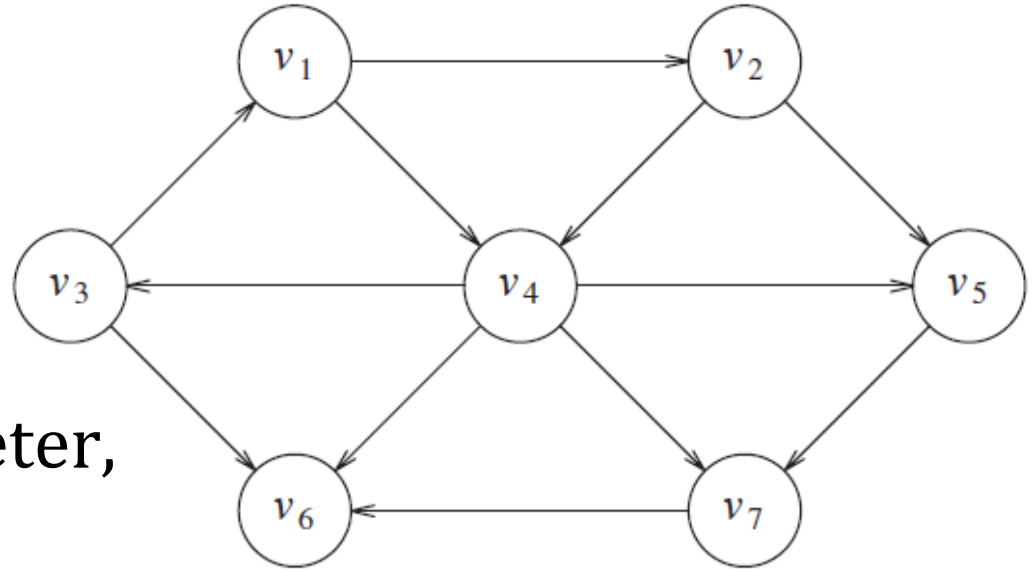



**Figure 9.10** An unweighted directed graph $G$

- 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

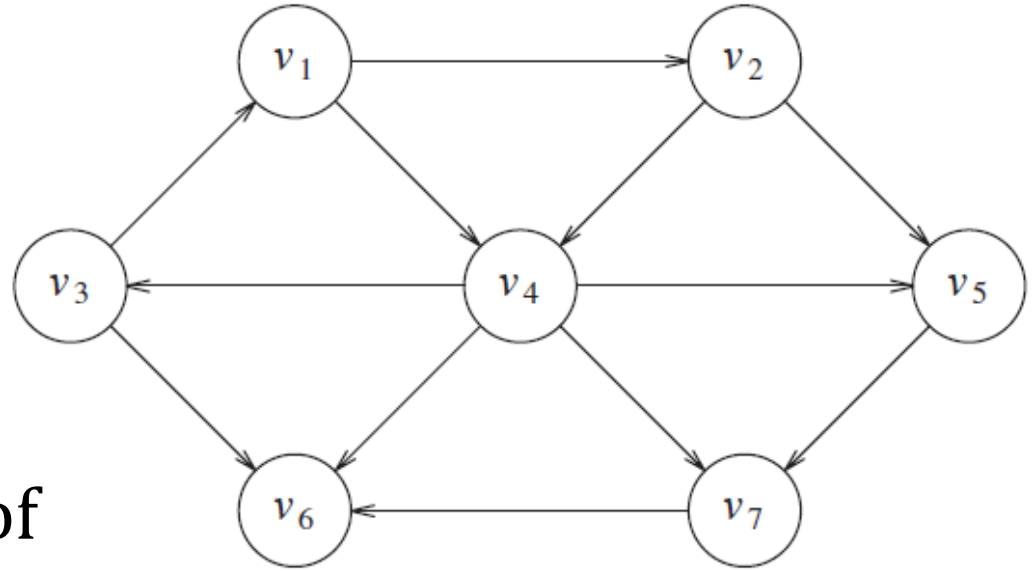



**Figure 9.10** An unweighted directed graph $G$

- 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.

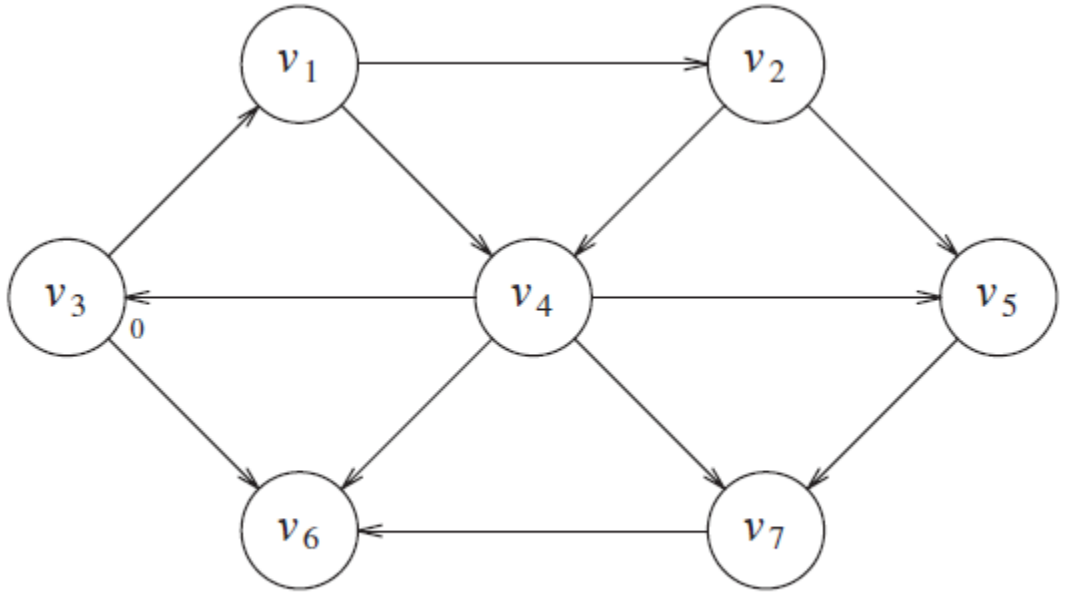



**Figure 9.11** Graph after marking the start node as reachable in zero edges

# 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.

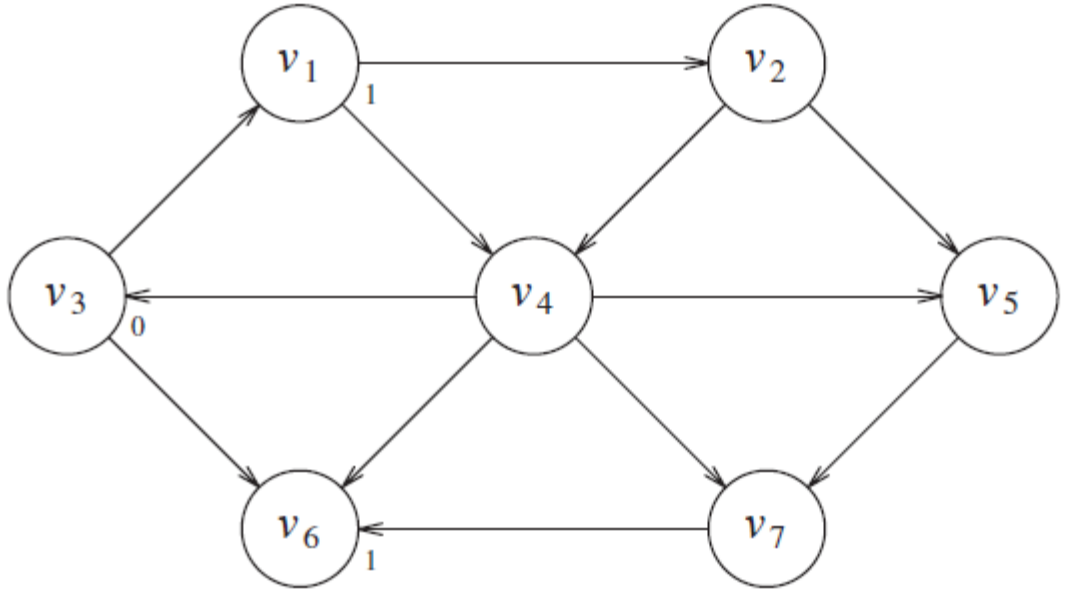



**Figure 9.12** Graph after finding all vertices whose path length from s is 1

# 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.

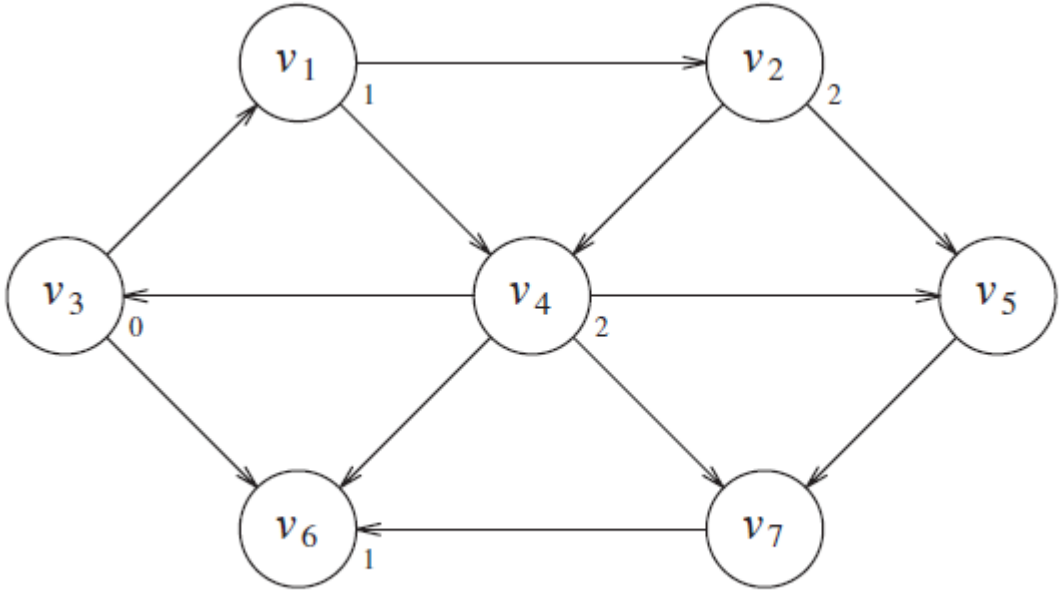



**Figure 9.13** Graph after finding all vertices whose shortest path is 2

# 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.

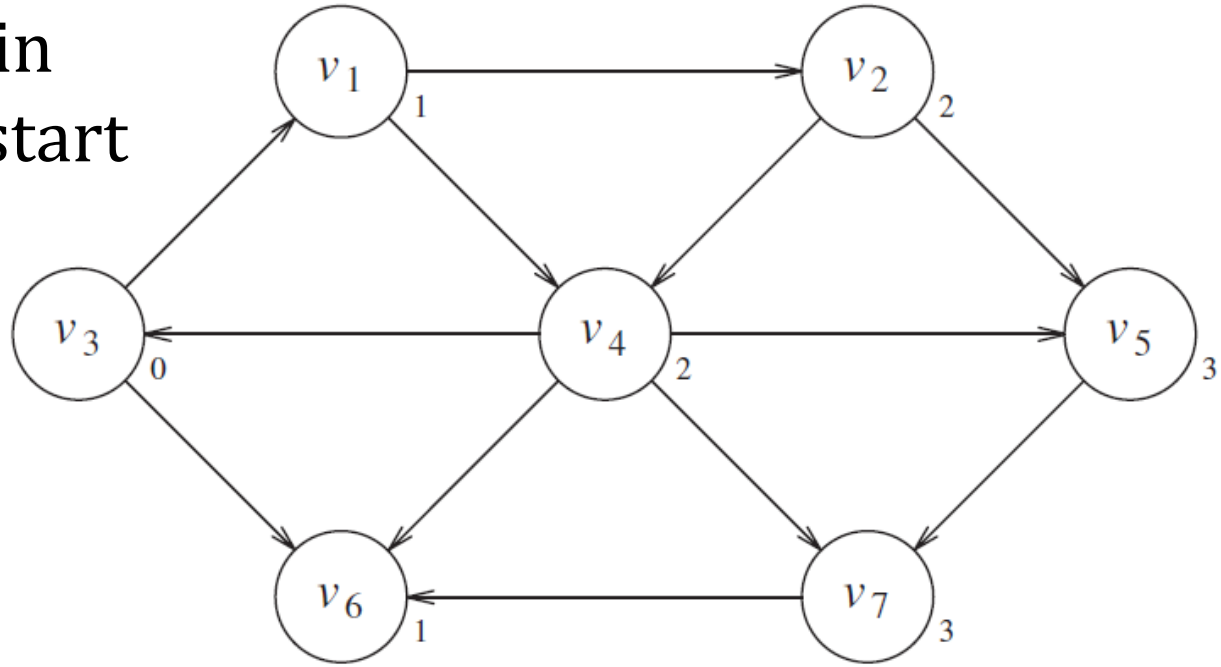



**Figure 9.14** Final shortest paths

# 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

# BFS - Algorithm

1. Store source vertex S in a queue and mark as processed
2. *while* queue is not empty
   Read vertex v from the queue
   *for* all neighbors w:
   If w is not processed Mark as processed
   Append in the queue Record the parent of w to be v (necessary only if we need the shortest path tree)

Time complexity?

# BFS - Algorithm

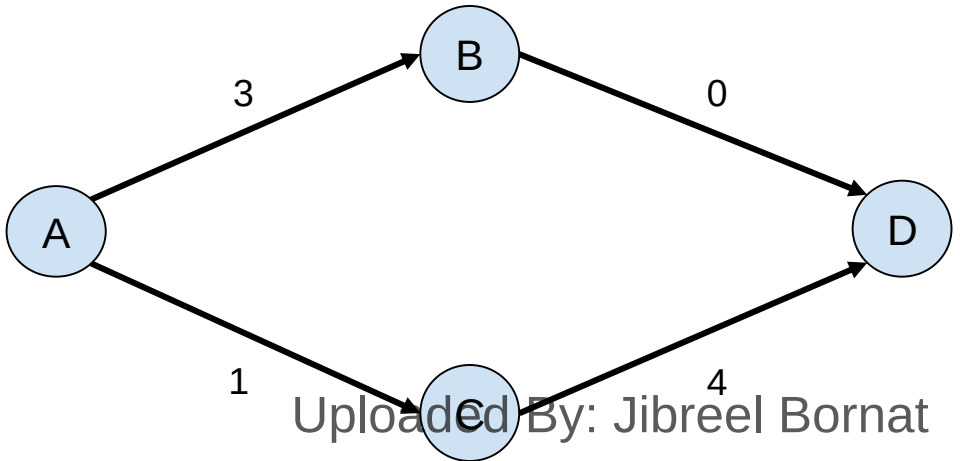
1. Store source vertex S in a queue and mark as processed
2. *while* queue is not empty
   Read vertex v from the queue
   *for* all neighbors w:
      If w is not processed Mark as processed
      Append in the queue Record the parent of w to be v

Time complexity? O(|V|+|E|)

# Dijkstra's Algorithm

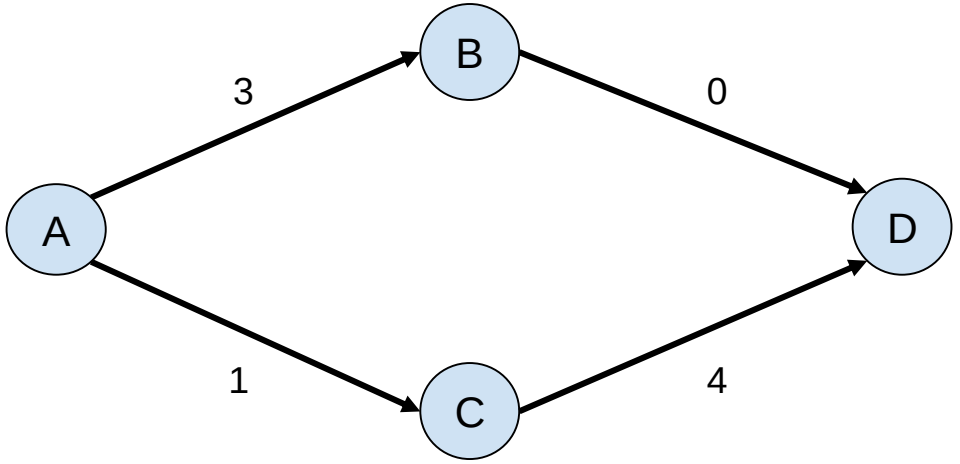
# Dijkstra's Algorithm

- If the graph is weighted, the problem becomes harder, but we can still use the ideas from the unweighted case.
  - Known/unknown and a tentative distance for each node
- 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.

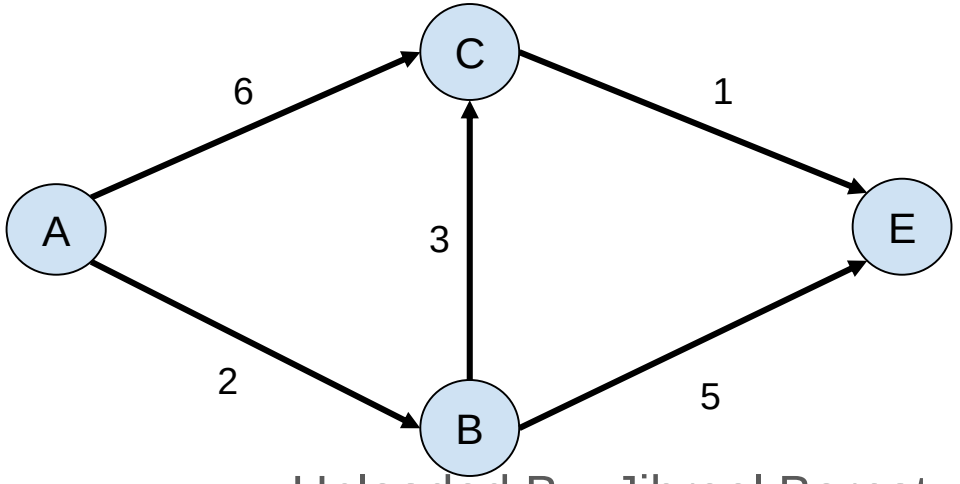


# Dijkstra's Algorithm

- 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

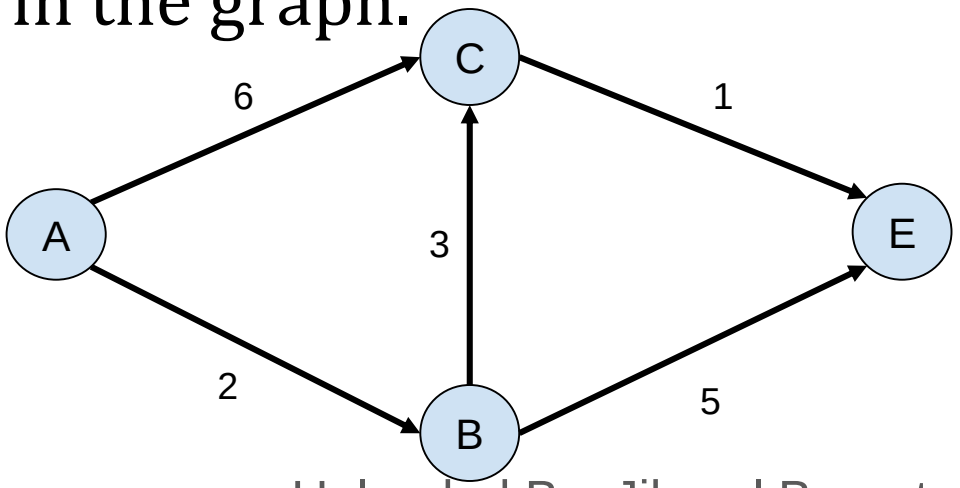
- Dijkstra's algorithm computes the cost of the shortest path from a starting vertex to all other vertices in the graph.
- Consider the following graph: Starting point 'A', destination 'E'.
- If we run this using the BFS, we will end-up with the cost of 7 (6+1)
- We aim at finding the destination is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



# Dijkstra's Algorithm

- 4-basic steps for Dijkstra's algorithm:

1. Find the vertex with the minimal cost. This is the vertex you can get to in the least amount of time.
2. Update the costs of the neighbor vertices.
3. Repeat until this is done for every vertex 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 $v$.

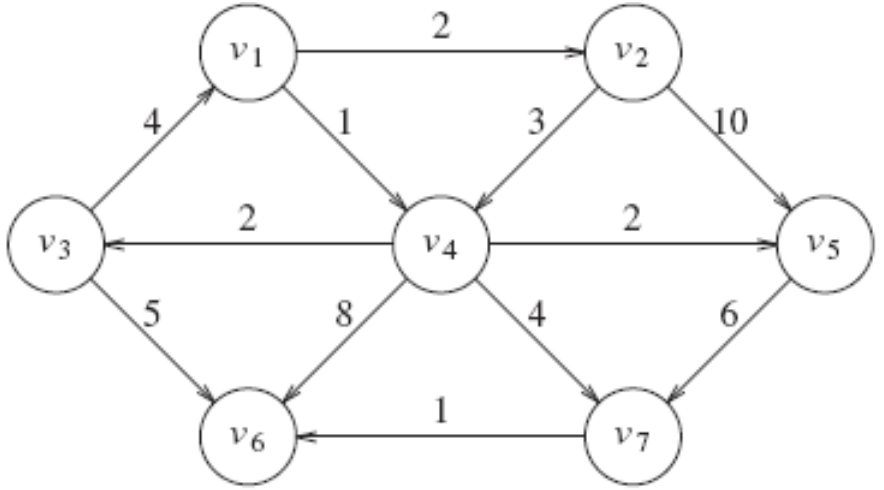
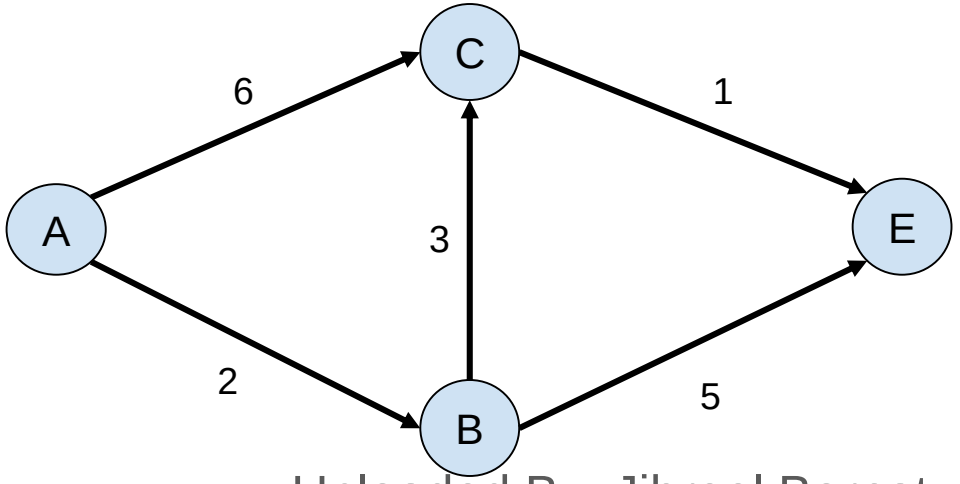



**Figure 9.20** Undirected graph G (again)

# Dijkstra's Algorithm

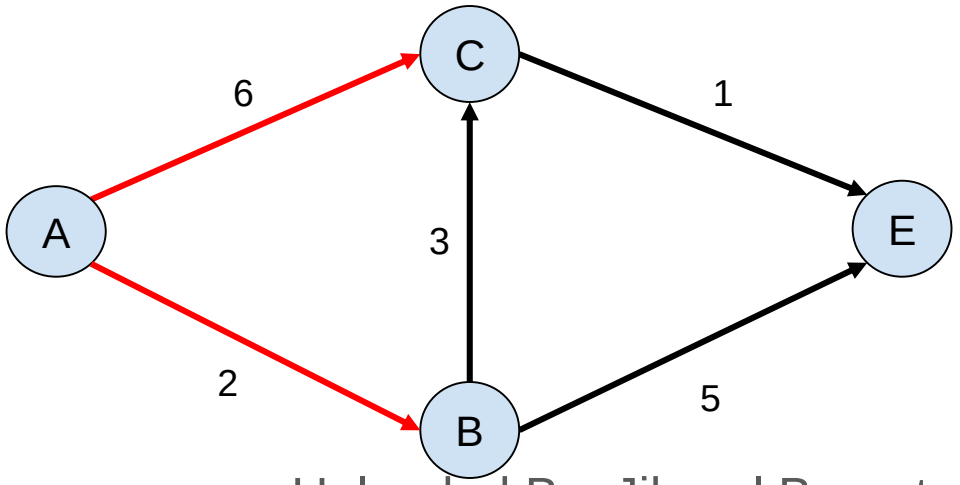
- **Step 1:** Find the node with the minimal cost.
- We are standing at the starting node 'A'. 'B' will take 6; and 'C' will take 2. We don't know the rest yet.
- As we don't know how long it will take to reach the destination, we will put it infinity.



# 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.

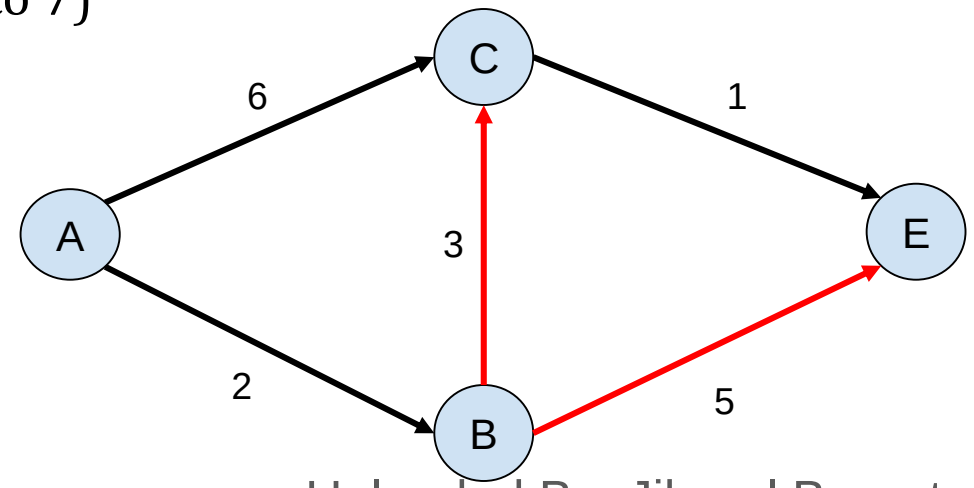
| Node | Cost to Node |
|---|---|
| B | 2 |
| C | 6 |
| E | ∞ |



# 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)

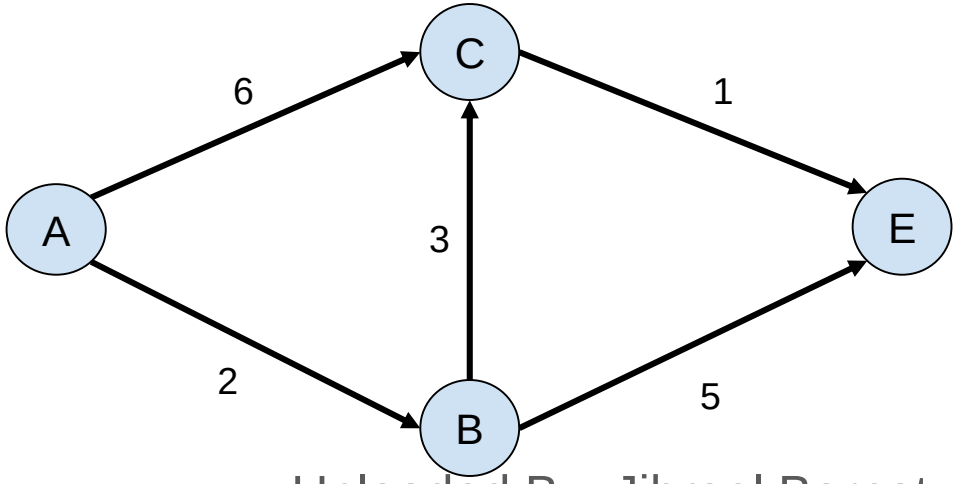
| Node | Cost to Node |
|---|---|
| B | 2 |
| C | ~~6~~ 5 |
| E | ~~∞~~ 7 |



# 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.

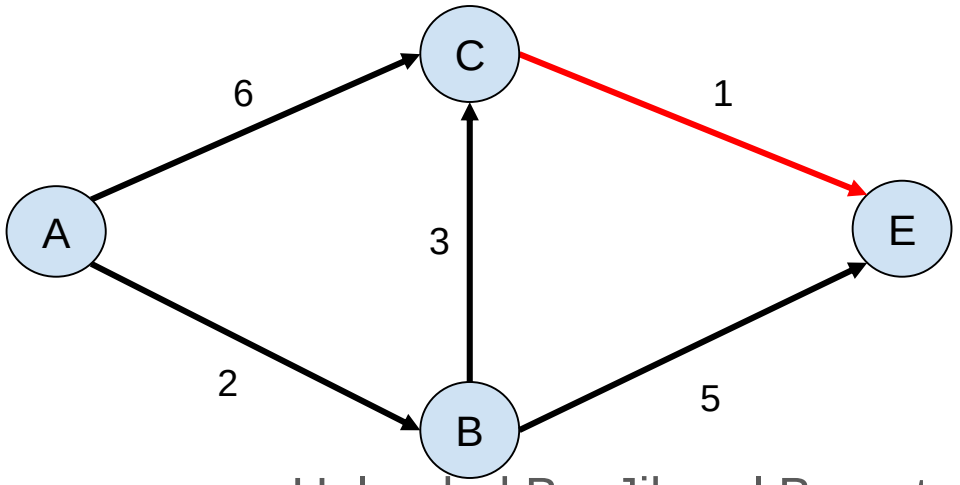
| Node | Cost to Node |
|---|---|
| B | 2 |
| C | 5 |
| E | 7 |



# Dijkstra's Algorithm

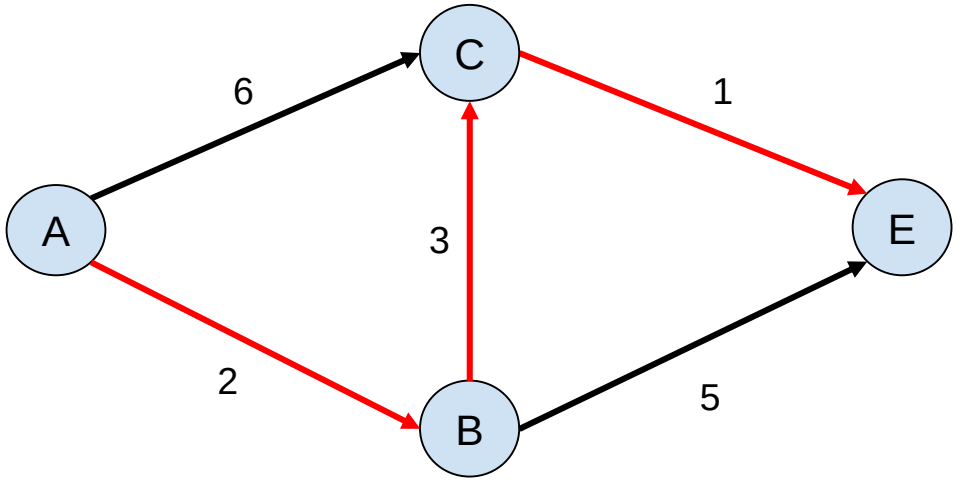
- Step 2 again: Update the cost of C's neighbours.
- We run Dijkstra's algorithm for every node (you don't need to run it
- for the finish node).
- At this point, you know
  - It takes 2 minutes to get to node B.
  - It takes 5 minutes to get to node C.
  - It takes 6 minutes to get to the destination.

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

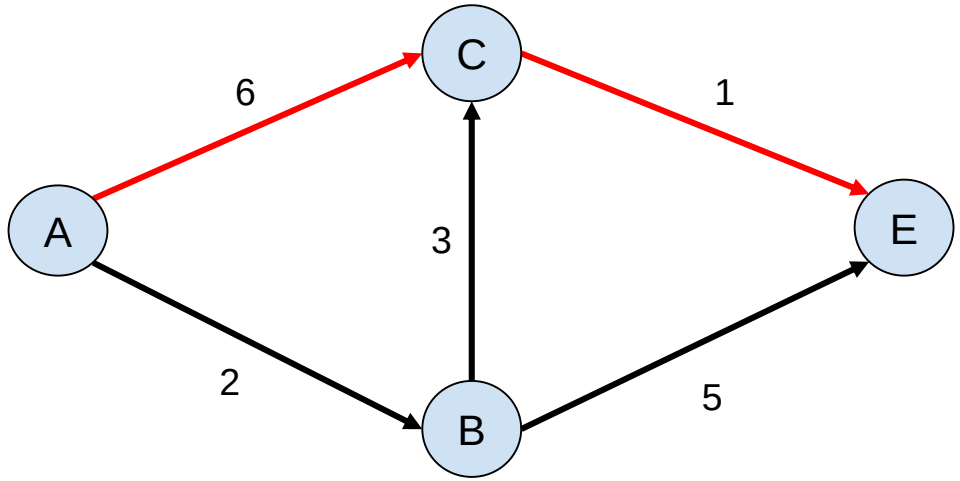


# Dijkstra's Algorithm

- So the final path is

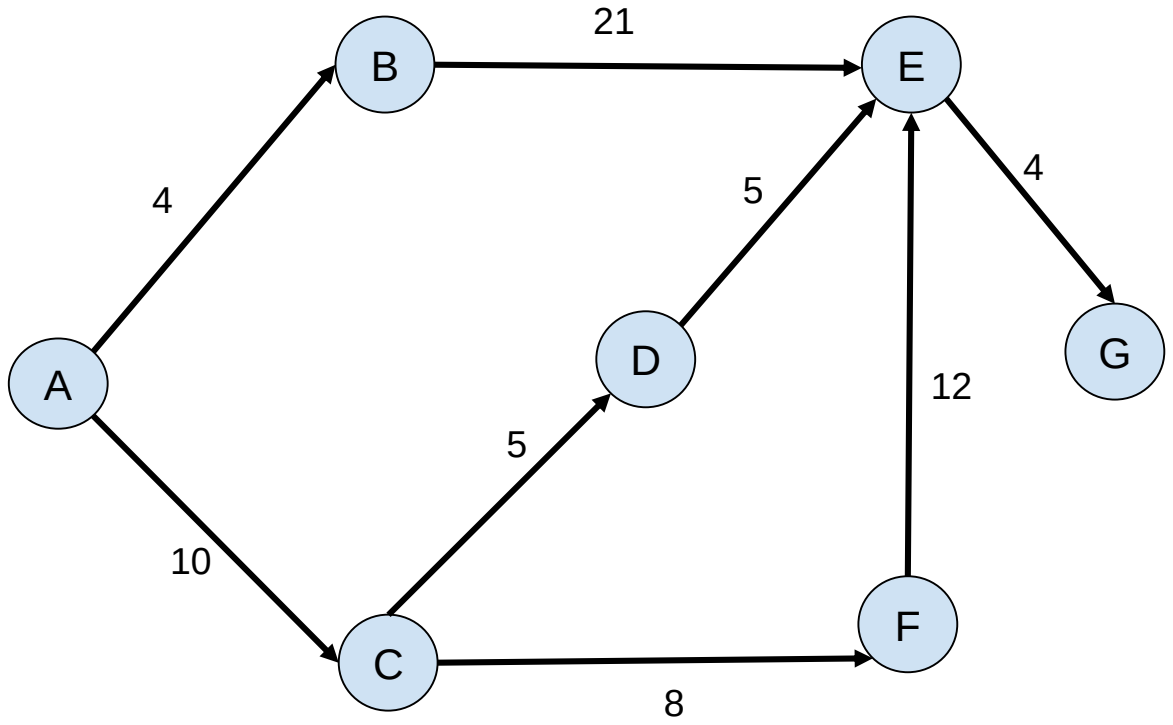



- BFS wouldn't have found this as the shortest path, because it has three segments.
- And there's a way to get from the start to the destination in two segments.



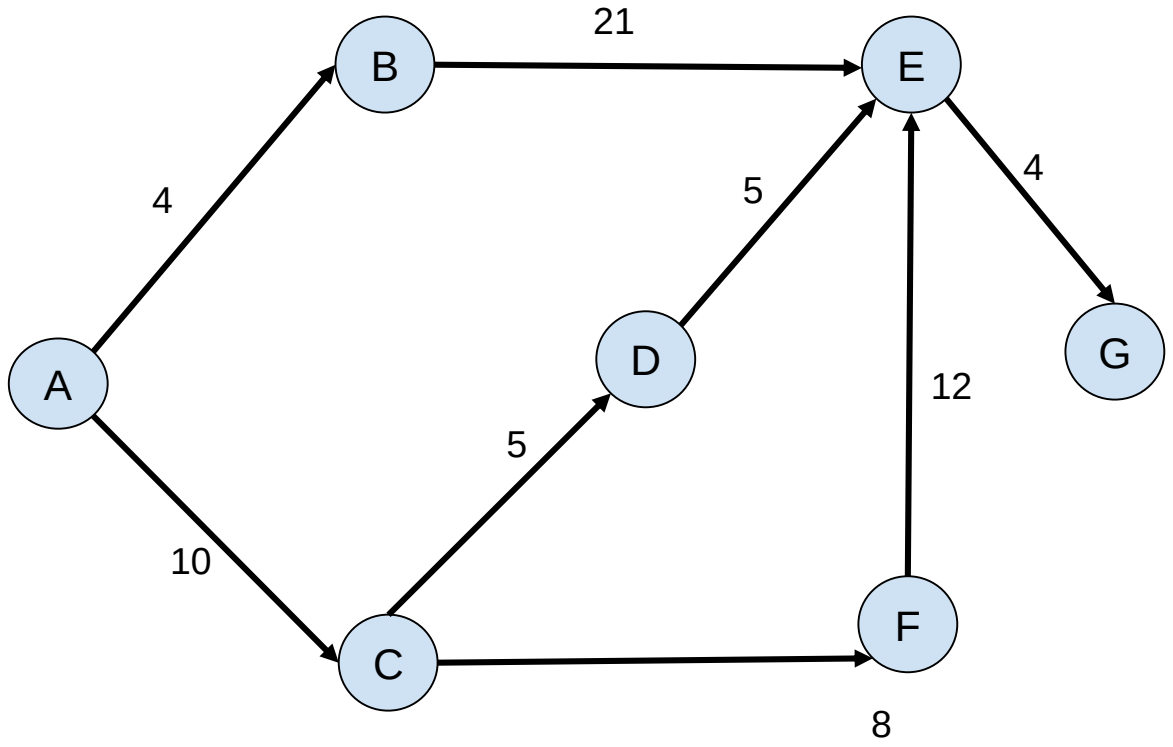
# Dijkstra's Algorithm - Example

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



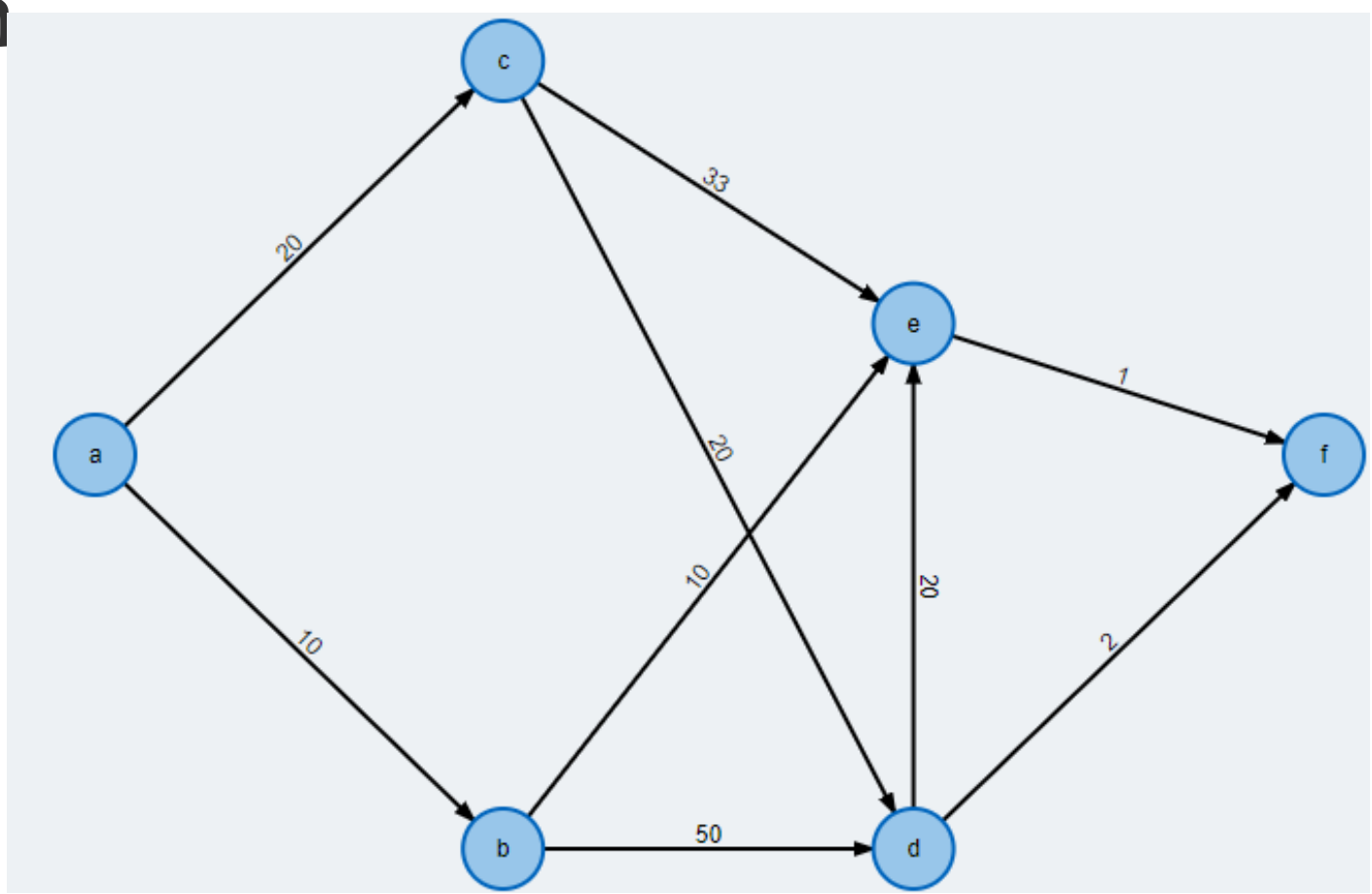
# Dijkstra's Algorithm - Example

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



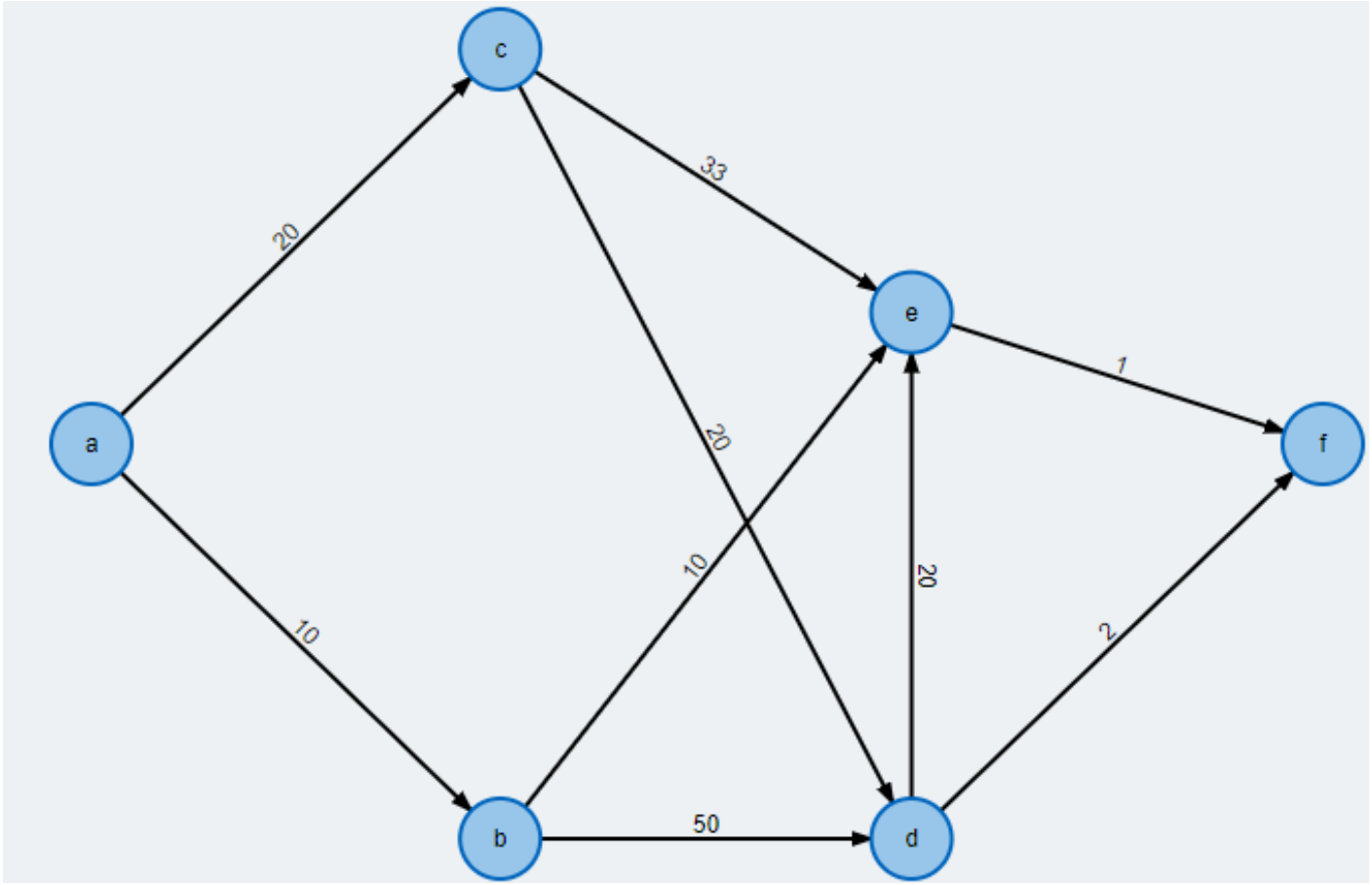
# Dijkstra's Algorithm

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



# Dijkstra's Algorithm

| Node | Final Cost |
|---|---|
| A | 0 |
| B | 10 |
| C | 20 |
| D | 40 |
| E | 20 |
| F | 21 |



# Dijkstra's Algorithm

Maintain 2 sets of vertices:

S -> a list contains the vertices that we have already discovered (we already found it's shortest path)
Q -> contains all vertices. Every vertex we are done with, then we will remove it from here.

Q-> priority queue and using heap sort. The priority is based on the min. distance.

# 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

# Graphs with Negative Edge Costs

- Edges with negative weights can create negative weight cycles (a cycle that will reduce the total path distance by coming back to the same vertex).
- If the graph has negative edge costs, then Dijkstra's algorithm will not work.
- Bellman-Ford algorithm solves the single-source shortest path when there may be negative weights in the graph.

# Graphs with Negative Edge Costs

- 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
- Bellman-Ford algorithm works by overestimating the length of the path from the starting vertex to all other vertices.
- Then it iteratively relaxes those estimates by finding new paths that are shorter than the previously overestimated paths.

# Graphs with Negative Edge Costs

- The only difference with Dijkstra is that Bellman-Ford is capable of handling negative weights whereas Dijkstra Algorithm can only handle positives
- In terms of cost, Dijkstra is better. The run-time of Bellman-Ford is O(V.E)
- N-1 iterations should ensure that the shortest path is reached. For example, if we have 6 vertices, then we need 5 iterations.
- In each iteration it will update all edges. Meaning, in each iteration it will examine all edges
  - Go through the nodes one-by-one, examine the outgoing edges and update accordingly

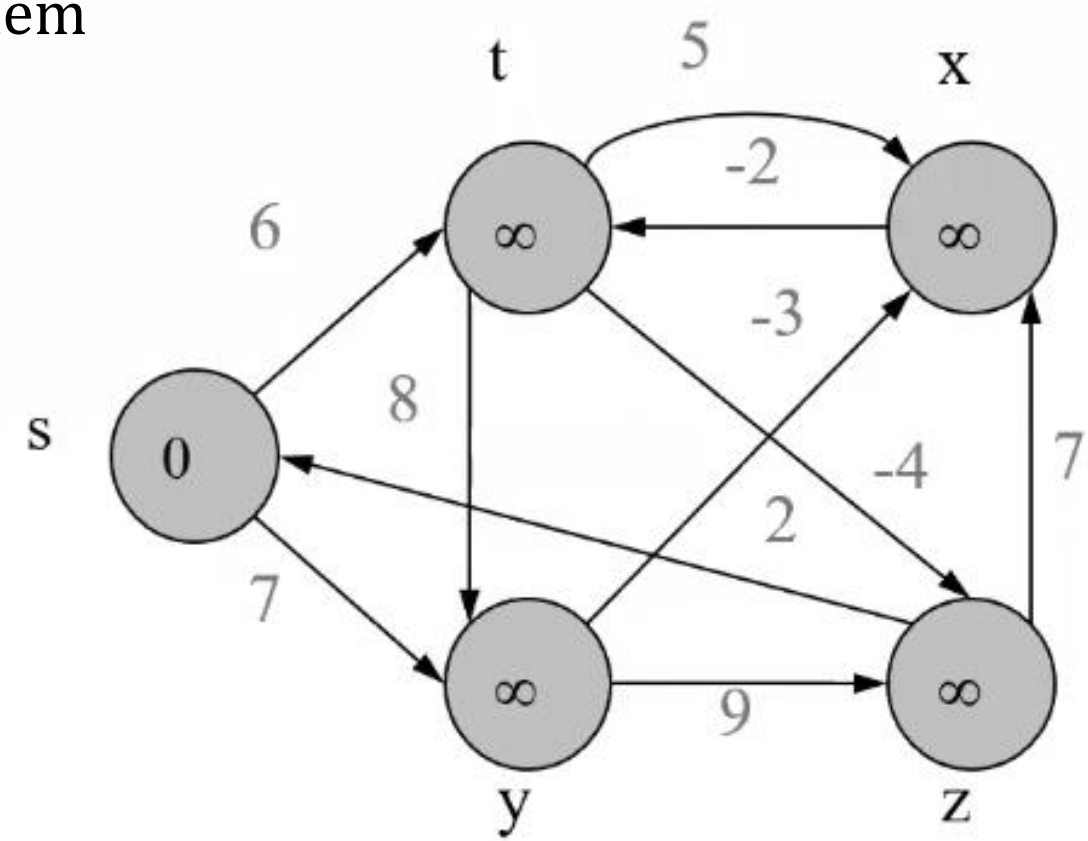
# Graphs with Negative Edge Costs

```
function bellmanFord(G, S)
  for each vertex V in G
    distance[V] <- infinite
      previous[V] <- NULL
  distance[S] <- 0

  for each vertex V in G
    for each edge (U,V) in G
      tempDistance <- distance[U] + edge_weight(U, V)
      if tempDistance < distance[V]
        distance[V] <- tempDistance
        previous[V] <- U

  for each edge (U,V) in G
    If distance[U] + edge_weight(U, V) < distance[V}
      Error: Negative Cycle Exists

  return distance[], previous[]
```

# Graphs with Negative Edge Costs

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

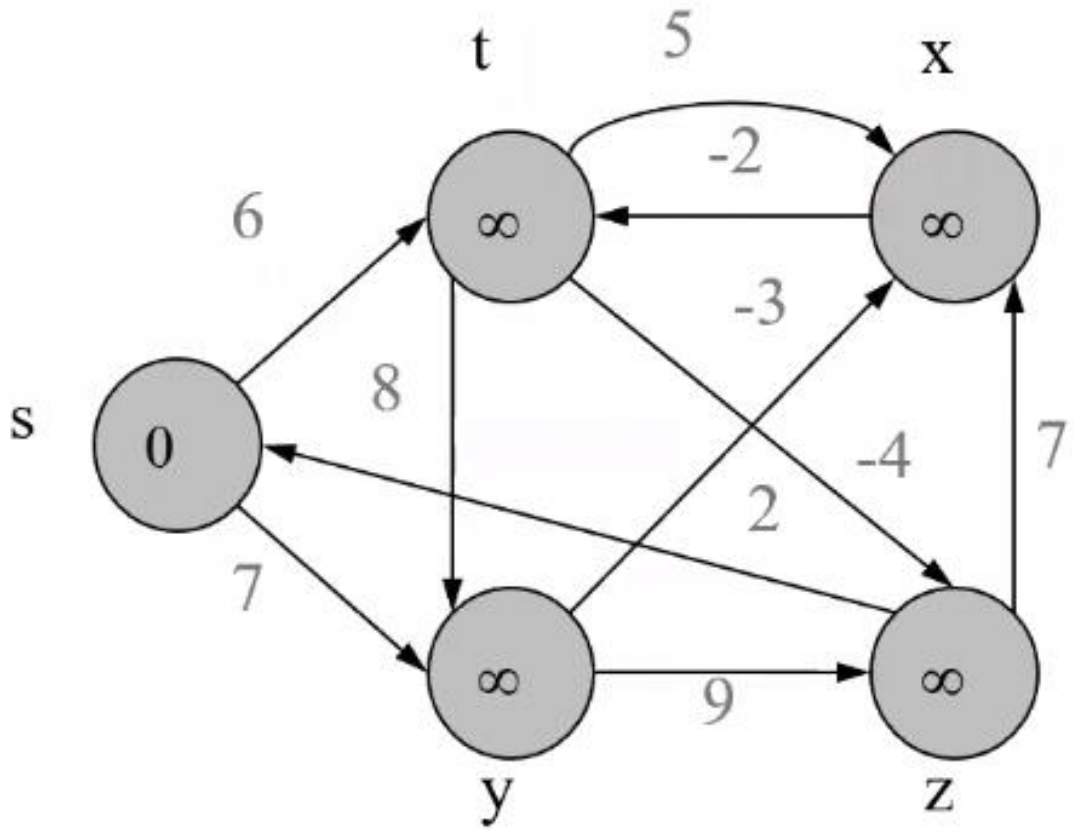
| Node | Initial. |
|---|---|
| s | 0 |
| t | ∞ |
| y | ∞ |
| x | ∞ |
| z | ∞ |



# Graphs with Negative Edge Costs

- The adjacent of s are y and t.

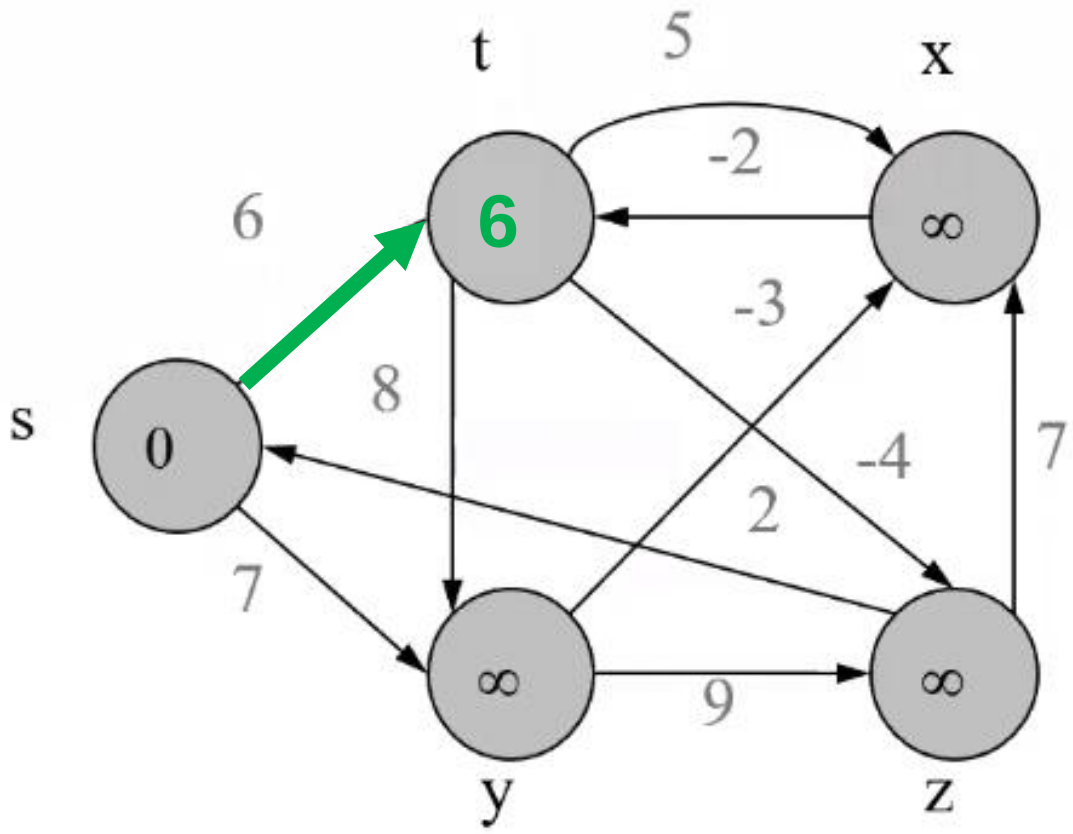
| Node | Initial. | Iter. 1 |
|---|---|---|
| s | 0 | |
| t | ∞ | |
| y | ∞ | |
| x | ∞ | |
| z | ∞ | |



# Graphs with Negative Edge Costs

- The adjacent of s are y and t.

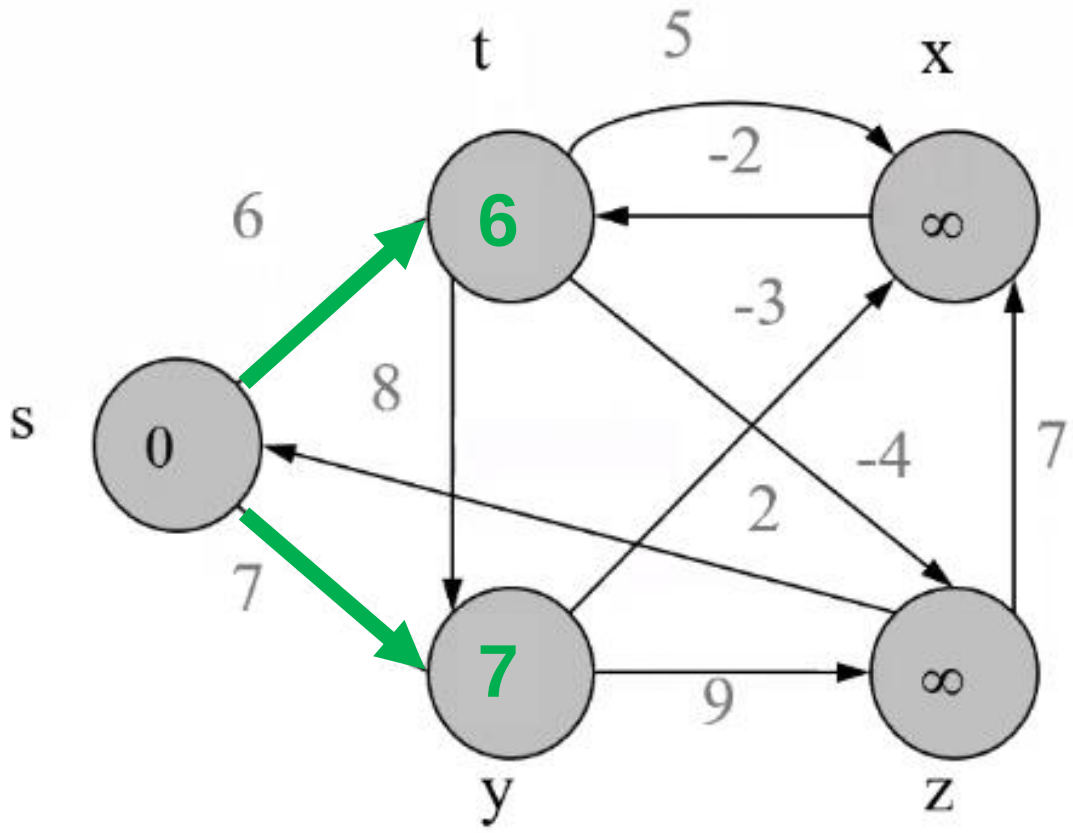
| Node | Initial. | Iter. 1 |
|---|---|---|
| s | 0 | |
| t | ∞ | 6 |
| y | ∞ | |
| x | ∞ | |
| z | ∞ | |



# Graphs with Negative Edge Costs

- The adjacent of s are y and t.

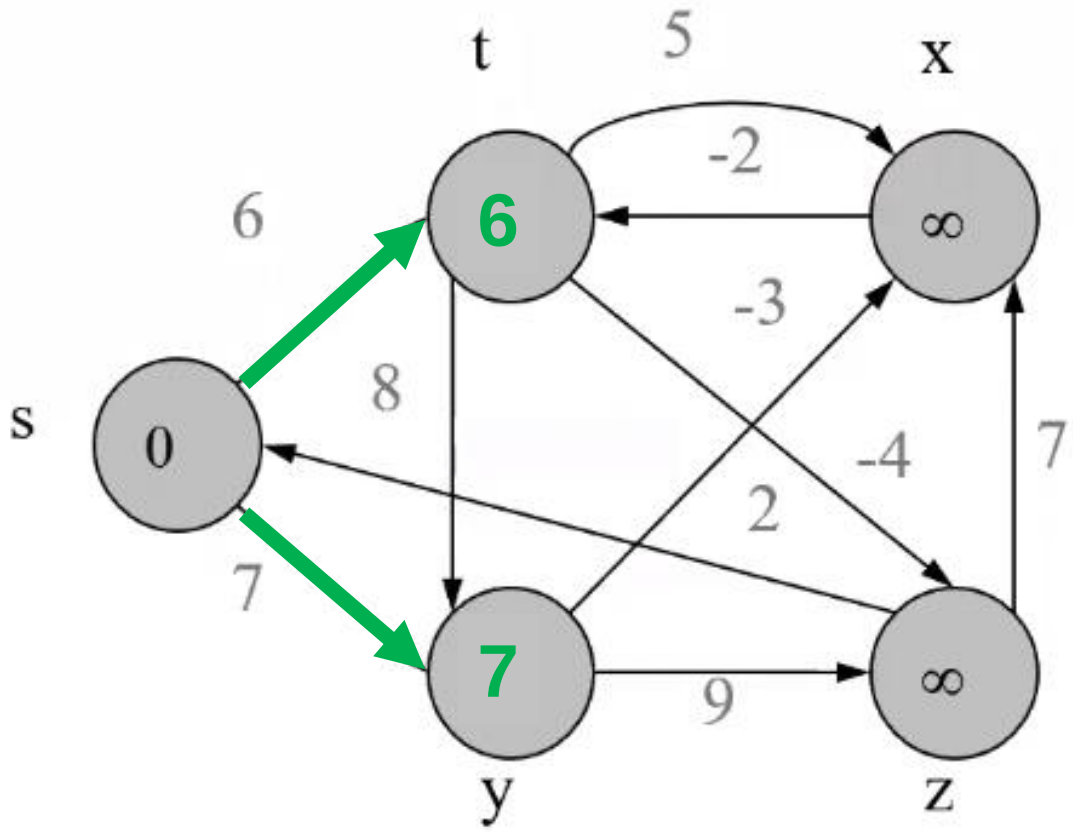
| Node | Initial. | Iter. 1 |
|---|---|---|
| s | 0 | |
| t | ∞ | 6 |
| y | ∞ | 7 |
| x | ∞ | |
| z | ∞ | |



# Graphs with Negative Edge Costs

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

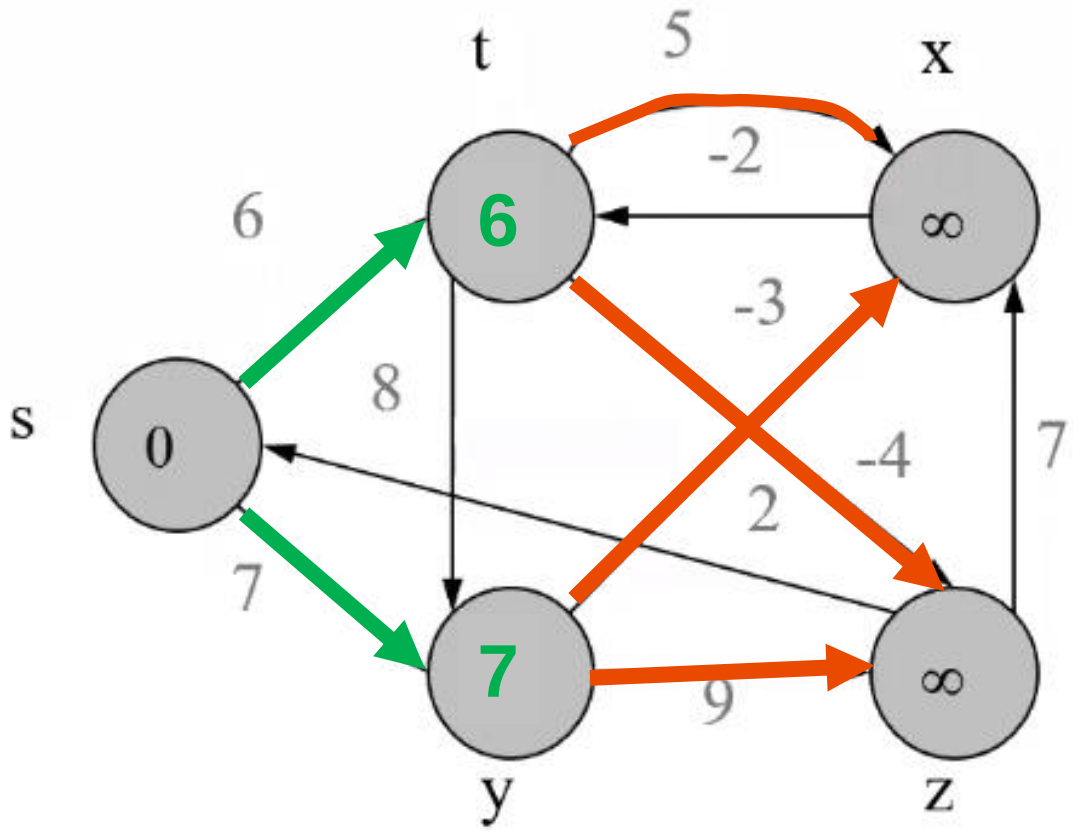
| Node | Initial. | Iter. 1 | Iter. 2 |
|---|---|---|---|
| s | 0 | 0 | |
| t | ∞ | 6 | |
| y | ∞ | 7 | |
| x | ∞ | ∞ | |
| z | ∞ | ∞ | |



# Graphs with Negative Edge Costs

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

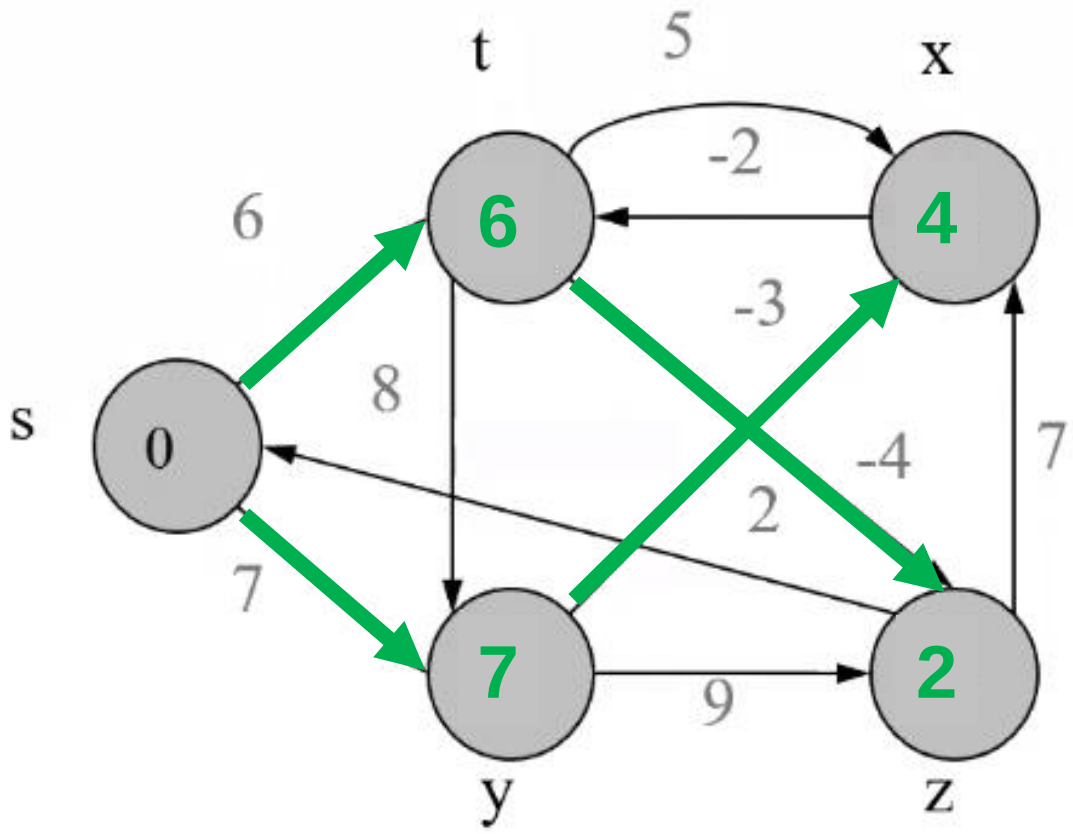
| Node | Initial. | Iter. 1 | Iter. 2 |
|---|---|---|---|
| s | 0 | 0 | |
| t | ∞ | 6 | |
| y | ∞ | 7 | |
| x | ∞ | ∞ | |
| z | ∞ | ∞ | |



# Graphs with Negative Edge Costs

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

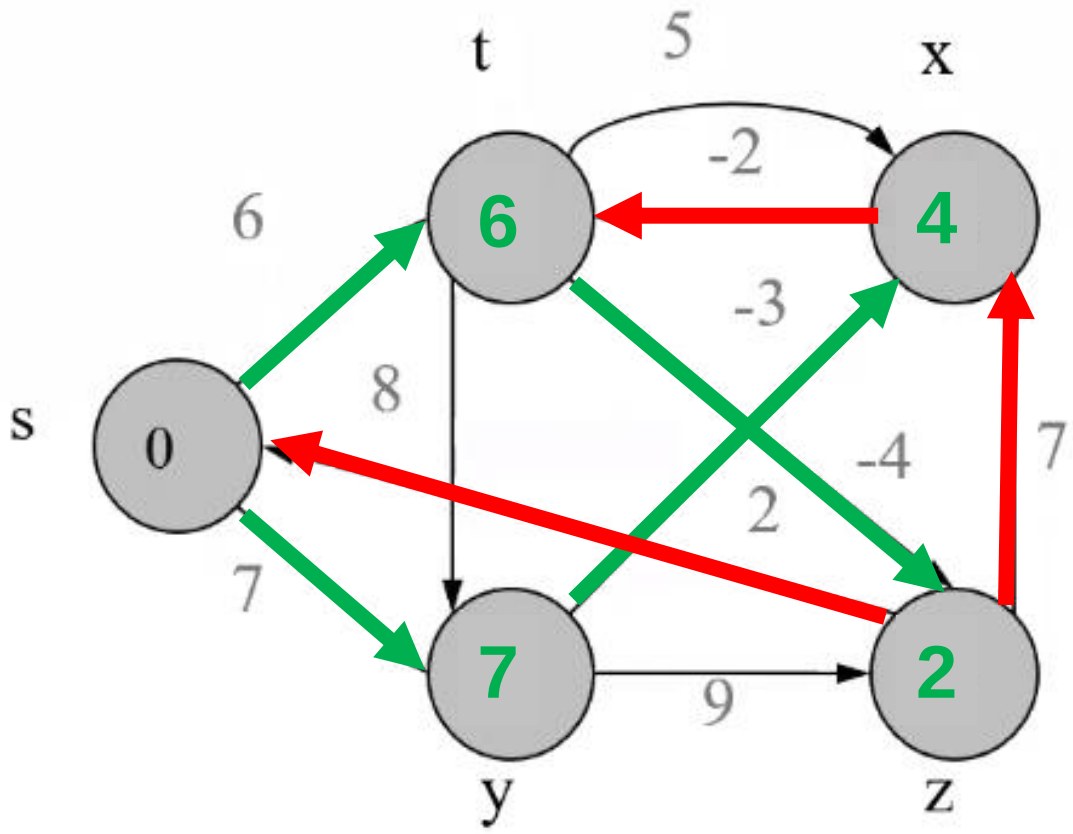
| Node | Initial. | Iter. 1 | Iter. 2 | Iter. 3 |
|---|---|---|---|---|
| s | 0 | 0 | 0 | |
| t | ∞ | 6 | 6 | |
| y | ∞ | 7 | 7 | |
| x | ∞ | ∞ | 4 | |
| z | ∞ | ∞ | 2 | |



# Graphs with Negative Edge Costs

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

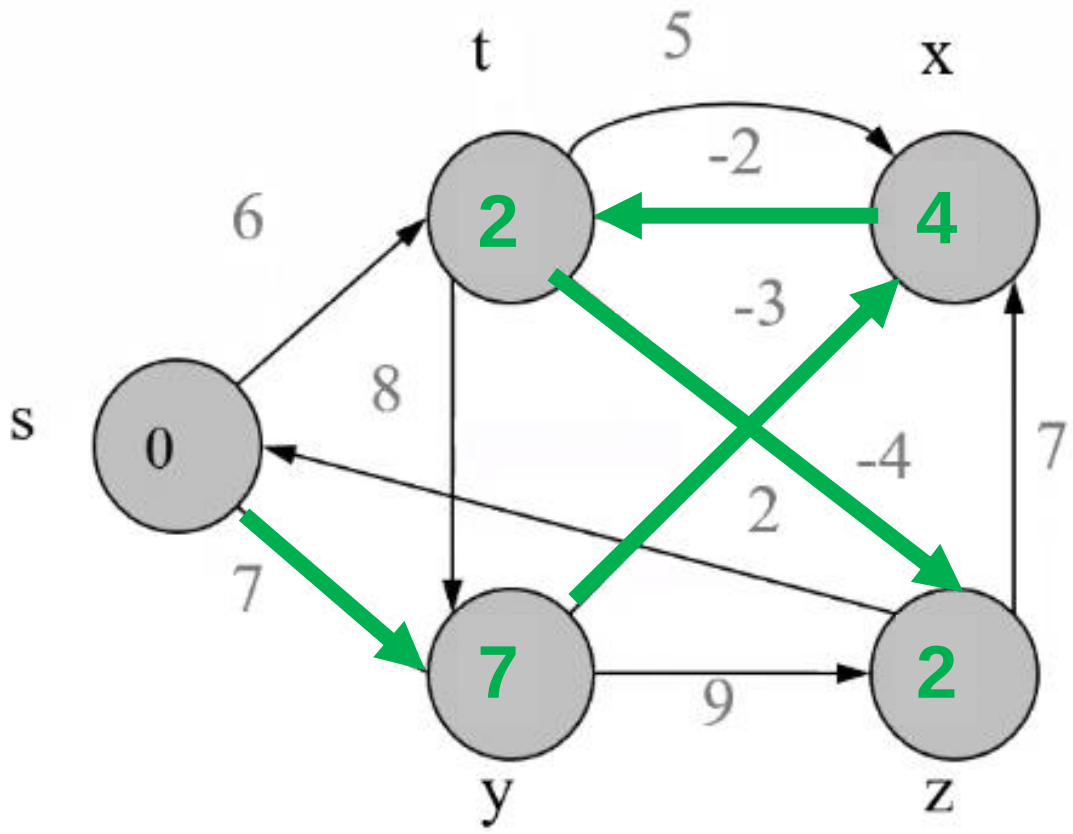
| Node | Initial. | Iter. 1 | Iter. 2 | Iter. 3 |
|---|---|---|---|---|
| s | 0 | 0 | 0 | |
| t | ∞ | 6 | 6 | |
| y | ∞ | 7 | 7 | |
| x | ∞ | ∞ | 4 | |
| z | ∞ | ∞ | 2 | |



# Graphs with Negative Edge Costs

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

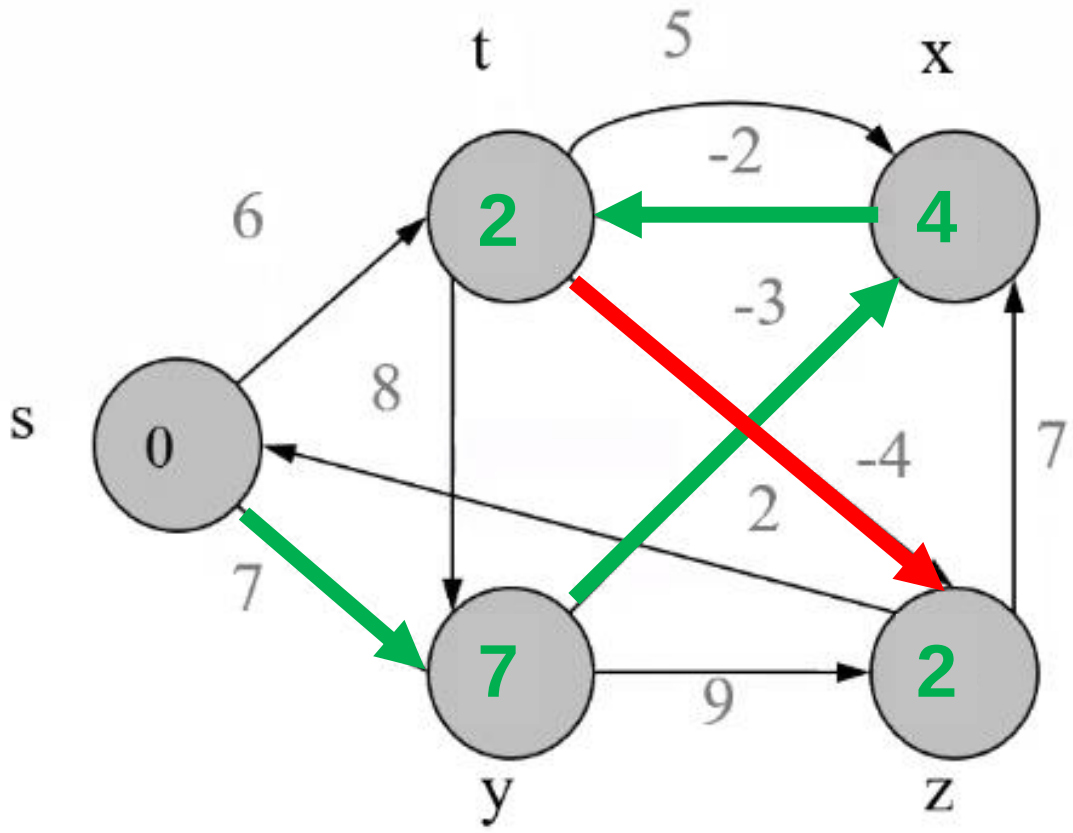
| Node | Initial. | Iter. 1 | Iter. 2 | Iter. 3 | |
|---|---|---|---|---|---|
| s | 0 | 0 | 0 | 0 | |
| t | ∞ | 6 | 6 | 2 | |
| y | ∞ | 7 | 7 | 7 | |
| x | ∞ | ∞ | 4 | 4 | |
| z | ∞ | ∞ | 2 | 2 | |



# Graphs with Negative Edge Costs

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

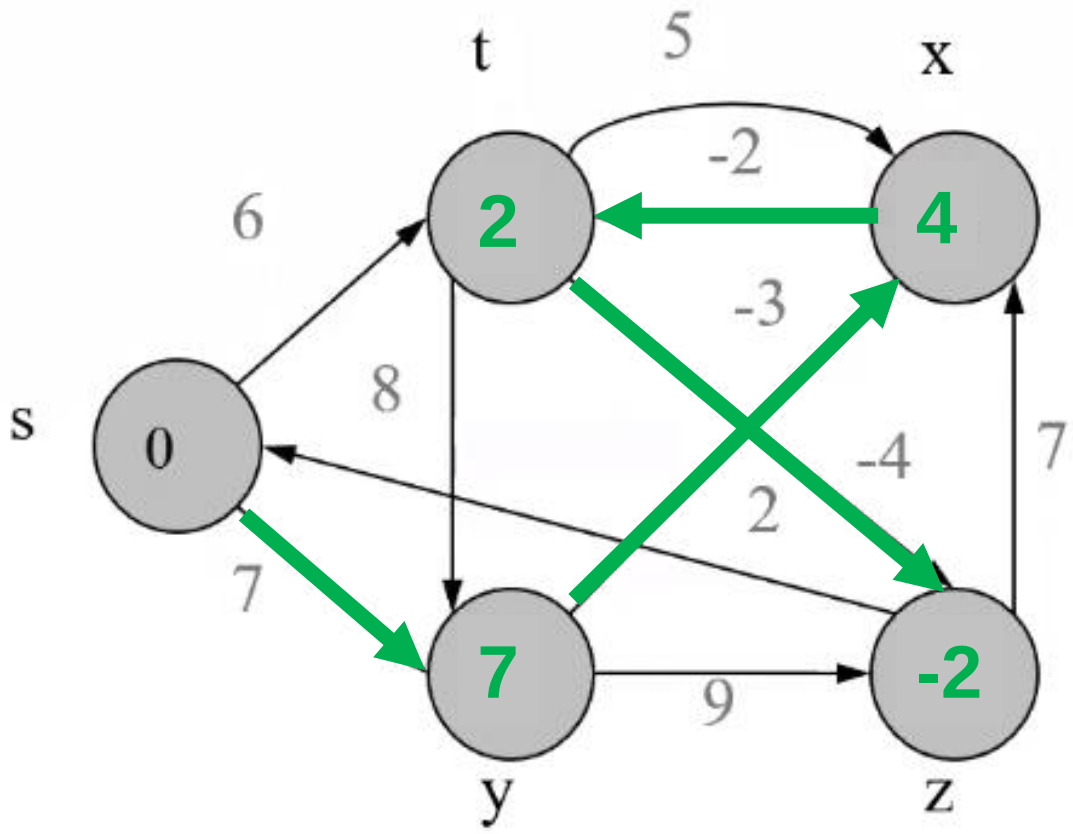
| Node | Initial. | Iter. 1 | Iter. 2 | Iter. 3 | |
|---|---|---|---|---|---|
| s | 0 | 0 | 0 | 0 | |
| t | ∞ | 6 | 6 | 2 | |
| y | ∞ | 7 | 7 | 7 | |
| x | ∞ | ∞ | 4 | 4 | |
| z | ∞ | ∞ | 2 | 2 | |



# Graphs with Negative Edge Costs

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

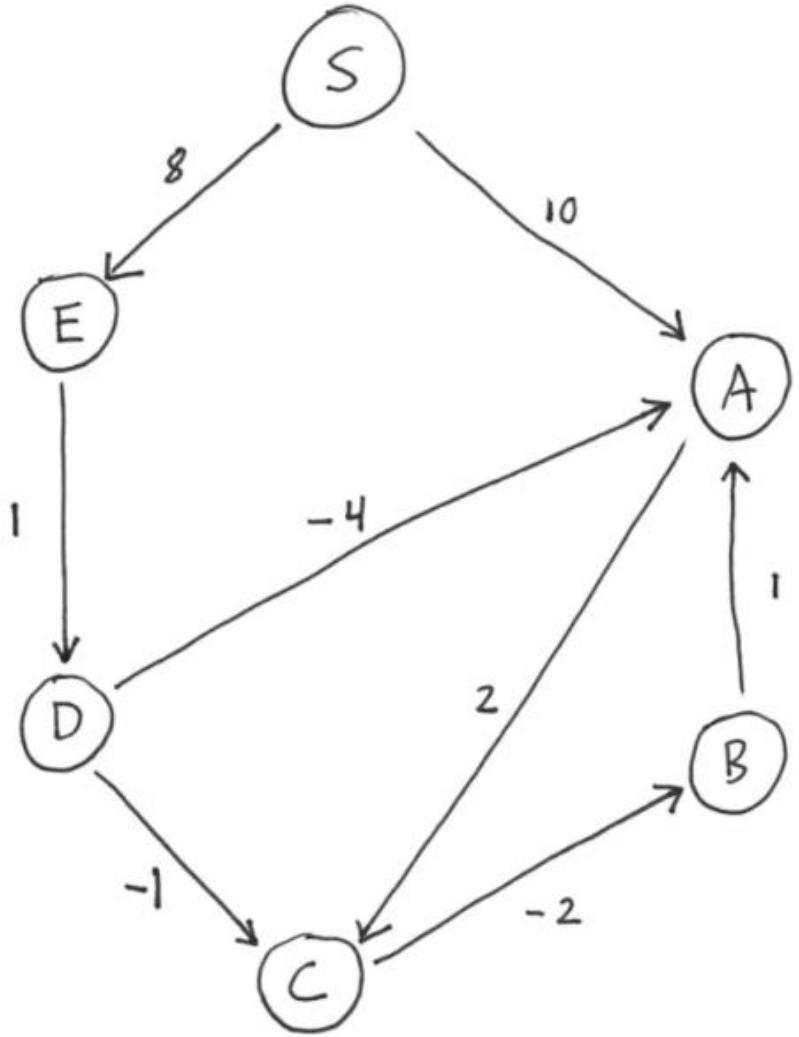
| Node | Initial. | Iter. 1 | Iter. 2 | Iter. 3 | Iter. 4 |
|---|---|---|---|---|---|
| s | 0 | 0 | 0 | 0 | 0 |
| t | ∞ | 6 | 6 | 2 | 2 |
| y | ∞ | 7 | 7 | 7 | 7 |
| x | ∞ | ∞ | 4 | 4 | 4 |
| z | ∞ | ∞ | 2 | 2 | -2 |



# Graphs with Negative Edge Costs

- In each iteration, we need to examine all edges
- This can be done by going through each vertex and examine its outgoing edges

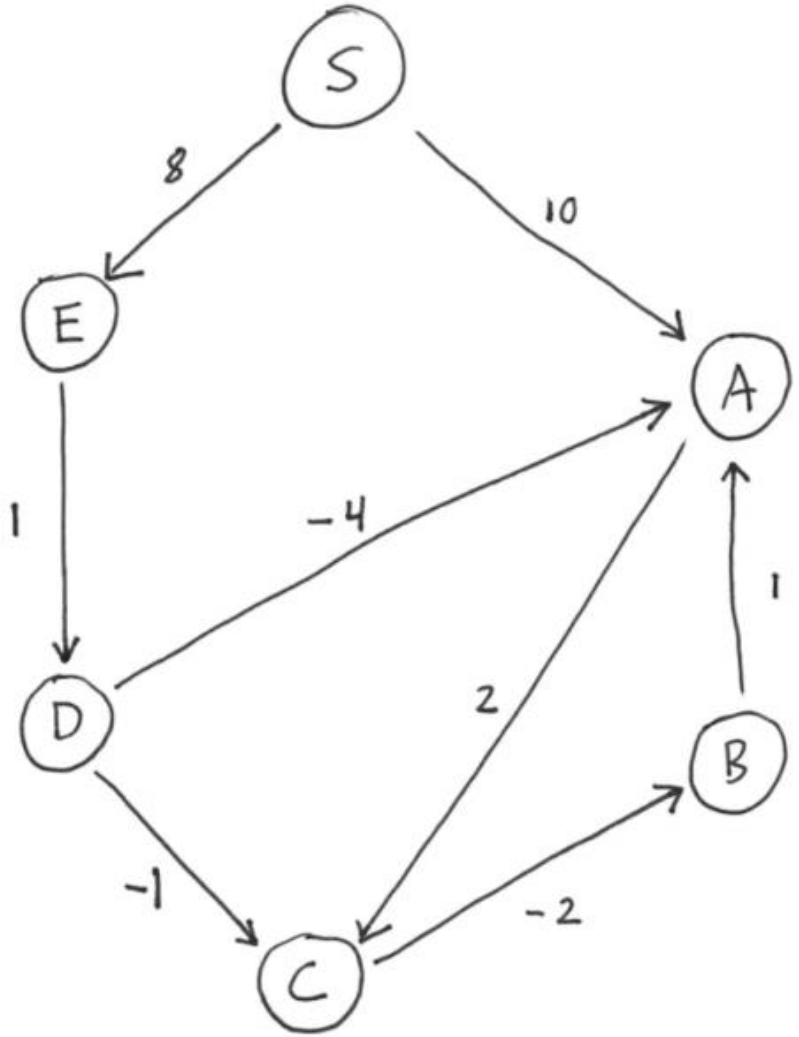
| Node | Initial. | 1st iter. |
|---|---|---|
| S | 0 | |
| A | ∞ | |
| B | ∞ | |
| C | ∞ | |
| D | ∞ | |
| E | ∞ | |



# Graphs with Negative Edge Costs

- In each iteration, we need to examine all edges
- This can be done by going through each vertex and examine its outgoing edges

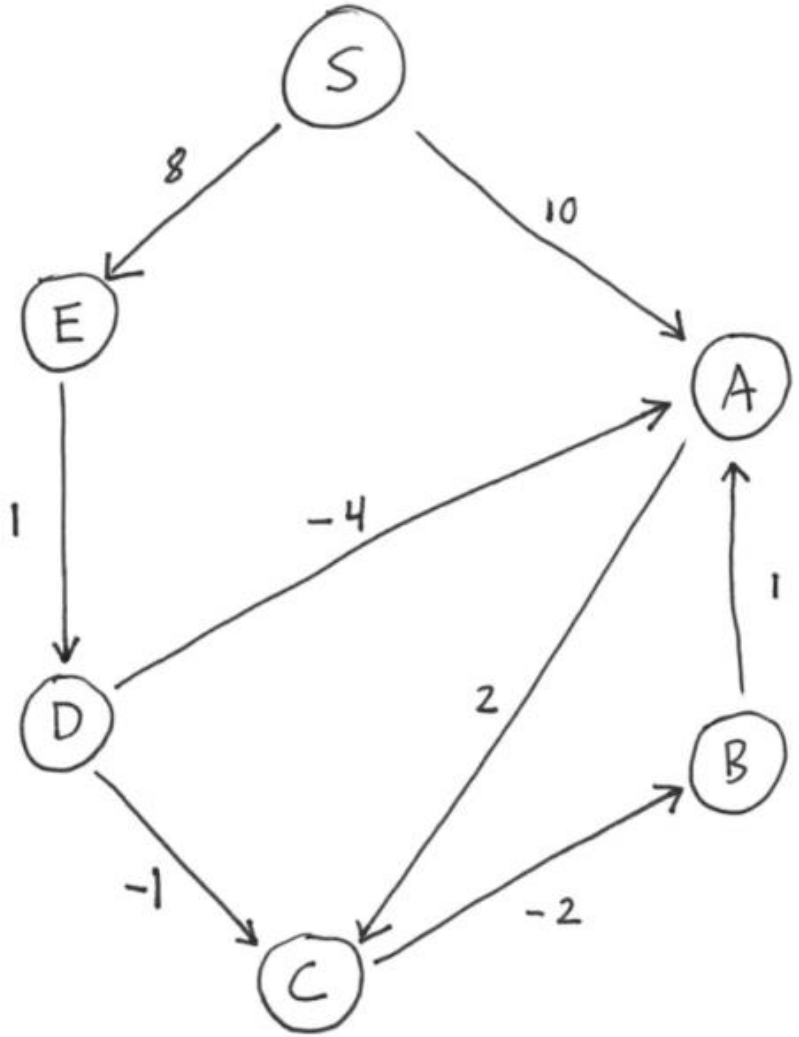
| Node | Initial. | 1st iter. | 2nd iter. |
|---|---|---|---|
| S | 0 | 0 | |
| A | ∞ | 10 | |
| B | ∞ | 10 | |
| C | ∞ | 12 | |
| D | ∞ | 9 | |
| E | ∞ | 8 | |



# Graphs with Negative Edge Costs

- In each iteration, we need to examine all edges
- This can be done by going through each vertex and examine its outgoing edges

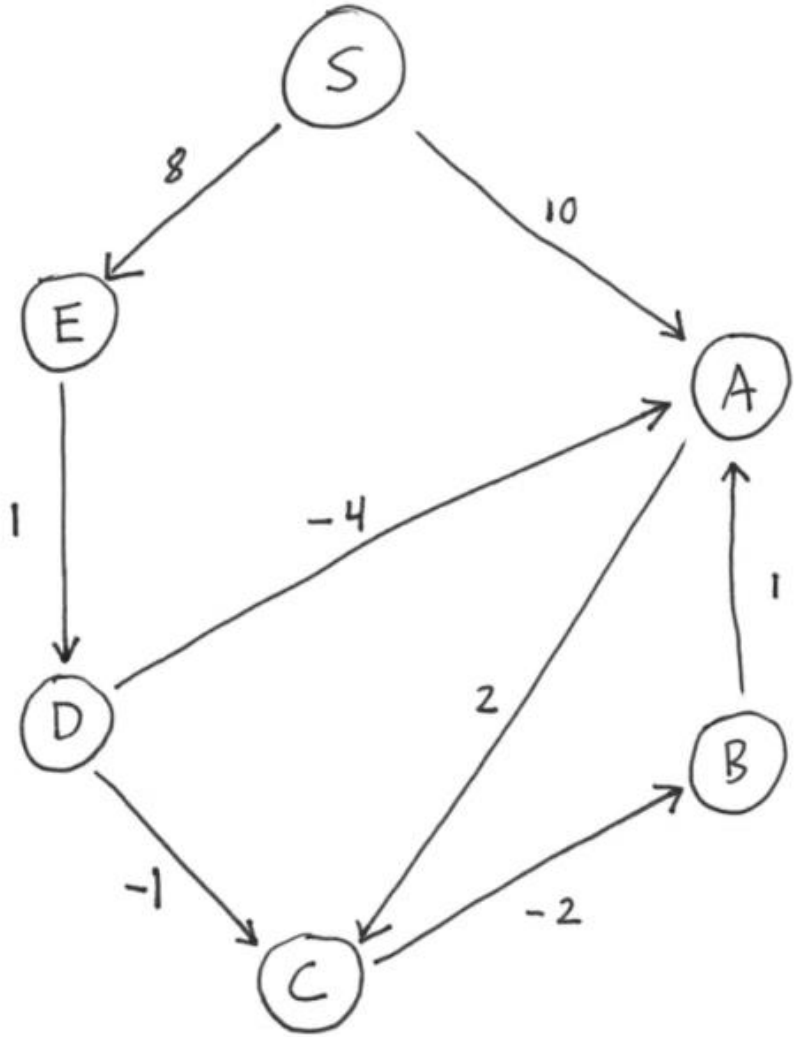
| Node | Initial. | 1st iter. | 2nd iter. | 3rd iter. |
|---|---|---|---|---|
| S | 0 | 0 | 0 | |
| A | ∞ | 10 | 5 | |
| B | ∞ | 10 | 10 | |
| C | ∞ | 12 | 8 | |
| D | ∞ | 9 | 9 | |
| E | ∞ | 8 | 8 | |



# Graphs with Negative Edge Costs

- In each iteration, we need to examine all edges
- This can be done by going through each vertex and examine its outgoing edges

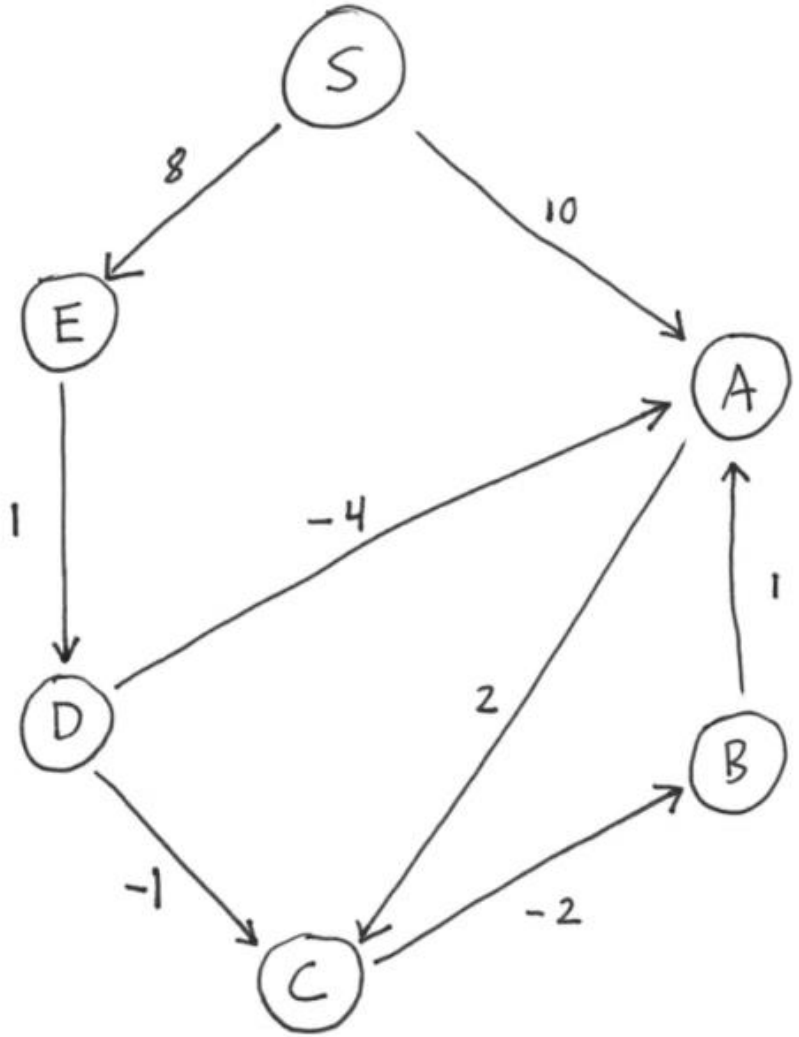
| Node | Initial. | 1st iter. | 2nd iter. | 3rd iter. | 4th iter. |
|---|---|---|---|---|---|
| S | 0 | 0 | 0 | 0 | |
| A | ∞ | 10 | 5 | 5 | |
| B | ∞ | 10 | 10 | 5 | |
| C | ∞ | 12 | 8 | 7 | |
| D | ∞ | 9 | 9 | 9 | |
| E | ∞ | 8 | 8 | 8 | |



# Graphs with Negative Edge Costs

- Since there is no improvement from iteration 3 to iteration 4, then we can stop before going to iteration 5 to improve efficiency

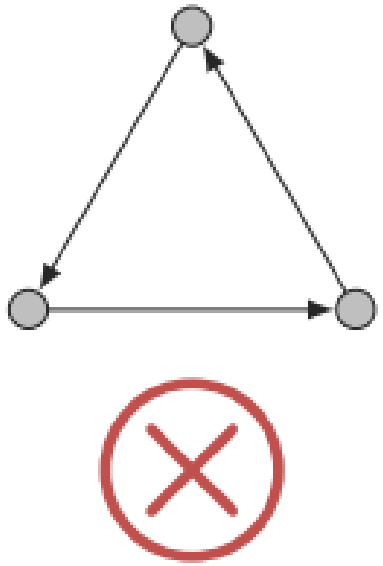
| Node | Initial. | 1st iter. | 2nd iter. | 3rd iter. | 4th iter. |
|---|---|---|---|---|---|
| S | 0 | 0 | 0 | 0 | 0 |
| A | ∞ | 10 | 5 | 5 | 5 |
| B | ∞ | 10 | 10 | 5 | 5 |
| C | ∞ | 12 | 8 | 7 | 7 |
| D | ∞ | 9 | 9 | 9 | 9 |
| E | ∞ | 8 | 8 | 8 | 8 |

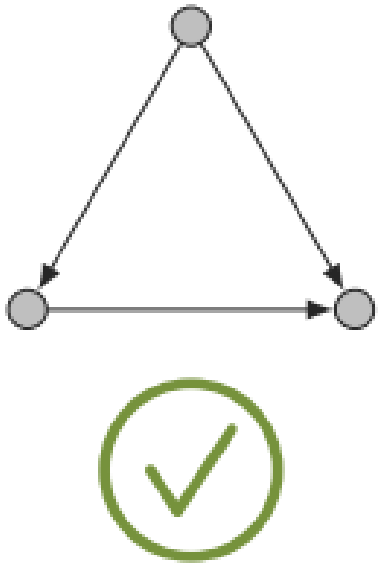


# Directed Acyclic Graphs

# Acyclic Graphs

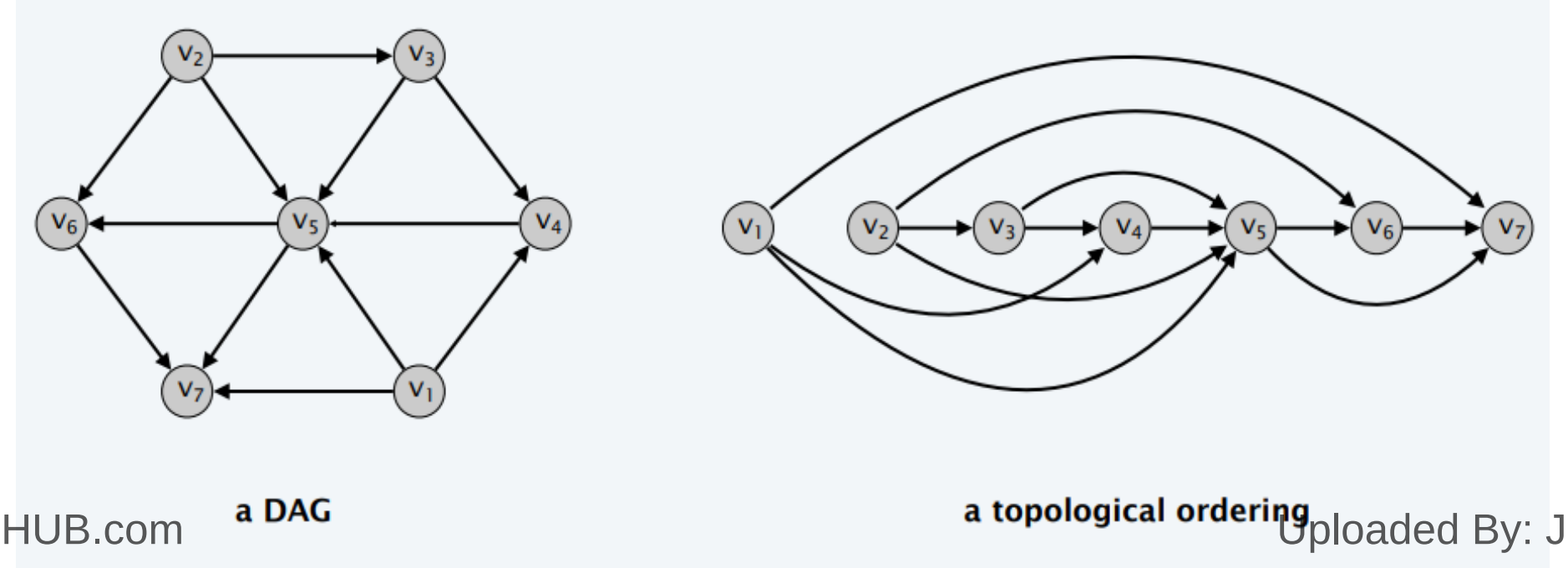
- We want to find the shortest path in acyclic graph (Directed Acyclic Graph)
- DAG contains no cycles





# 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

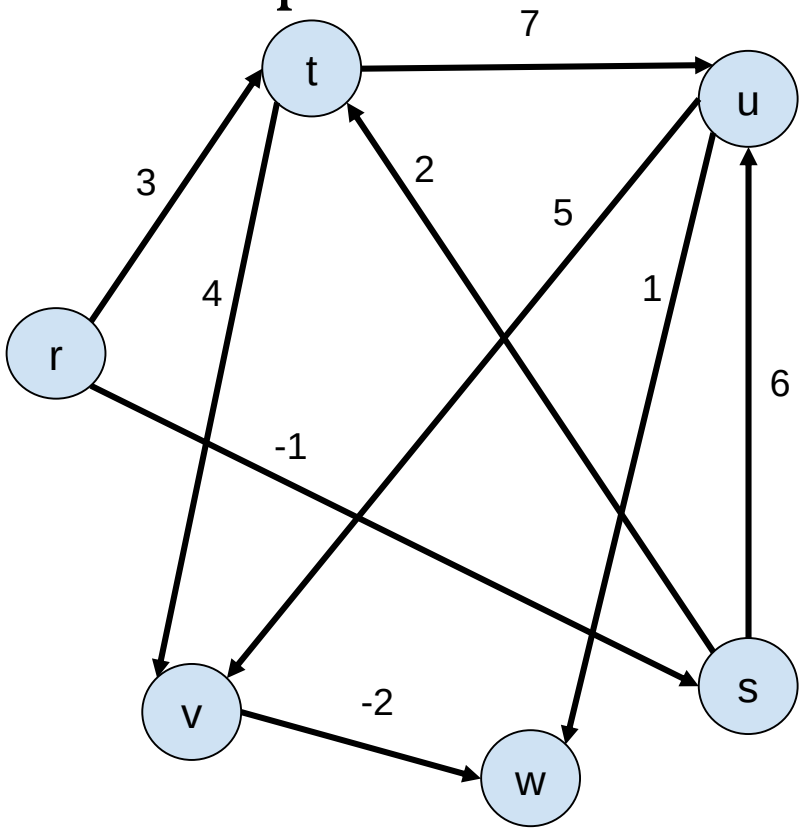


# Acyclic Graphs

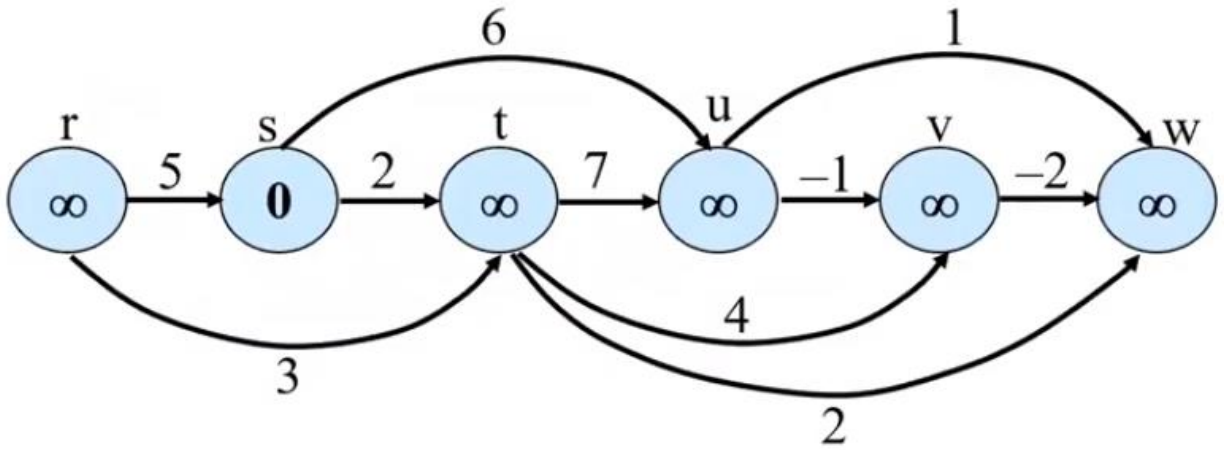
- Precedence constraints: Edge ($v_i$ , $v_j$ ) means task $v_i$ must occur before $v_j$
- Examples of DAG
  - Course prerequisite graph: course $v_i$ must be taken before $v_j$
  - Compilation: module $v_i$ must be compiled before $v_j$
  - Pipeline of computing jobs: output of job $v_i$ needed to determine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

# Acyclic Graphs

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




DAG

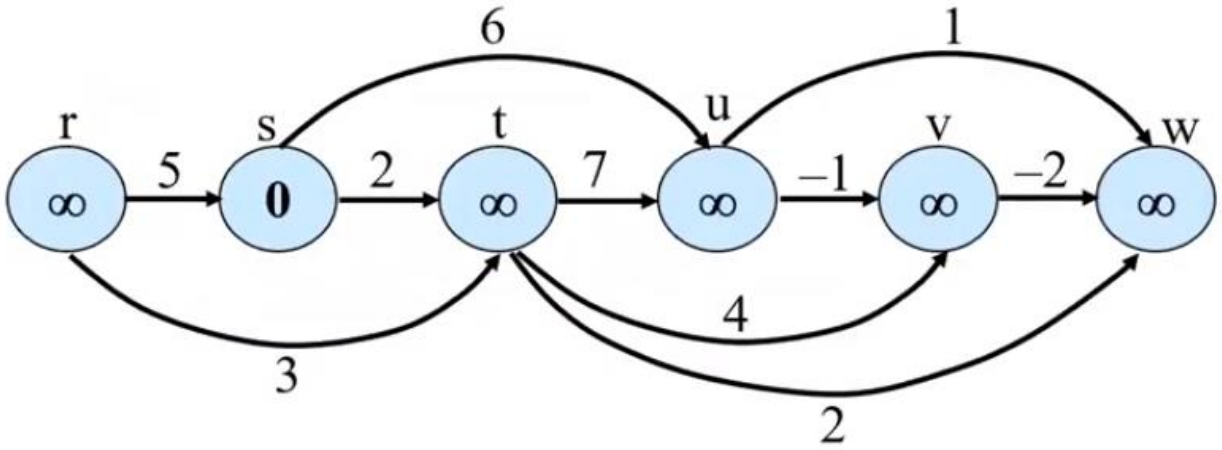



Topologically sorted

# 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)

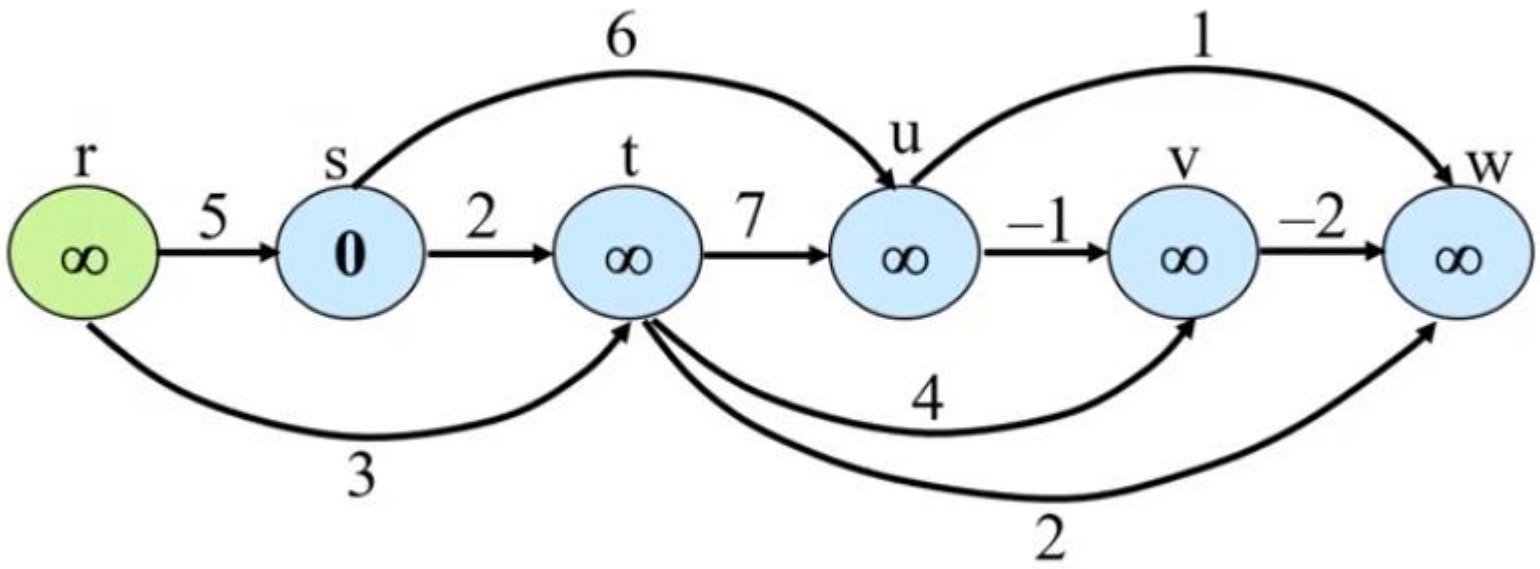
# Acyclic Graphs




- 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

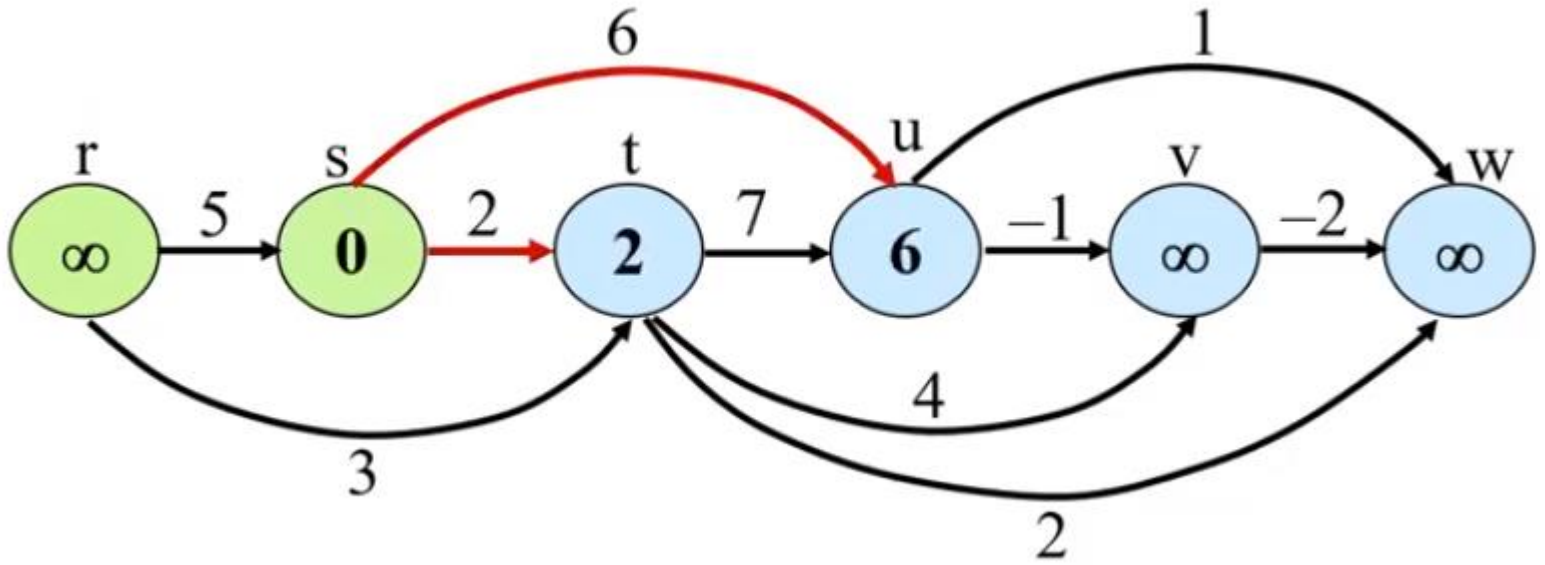
- So the first iteration, r = ∞




- 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, 6 is less than ∞, so update u and the predecessor is s

# Acyclic Graphs

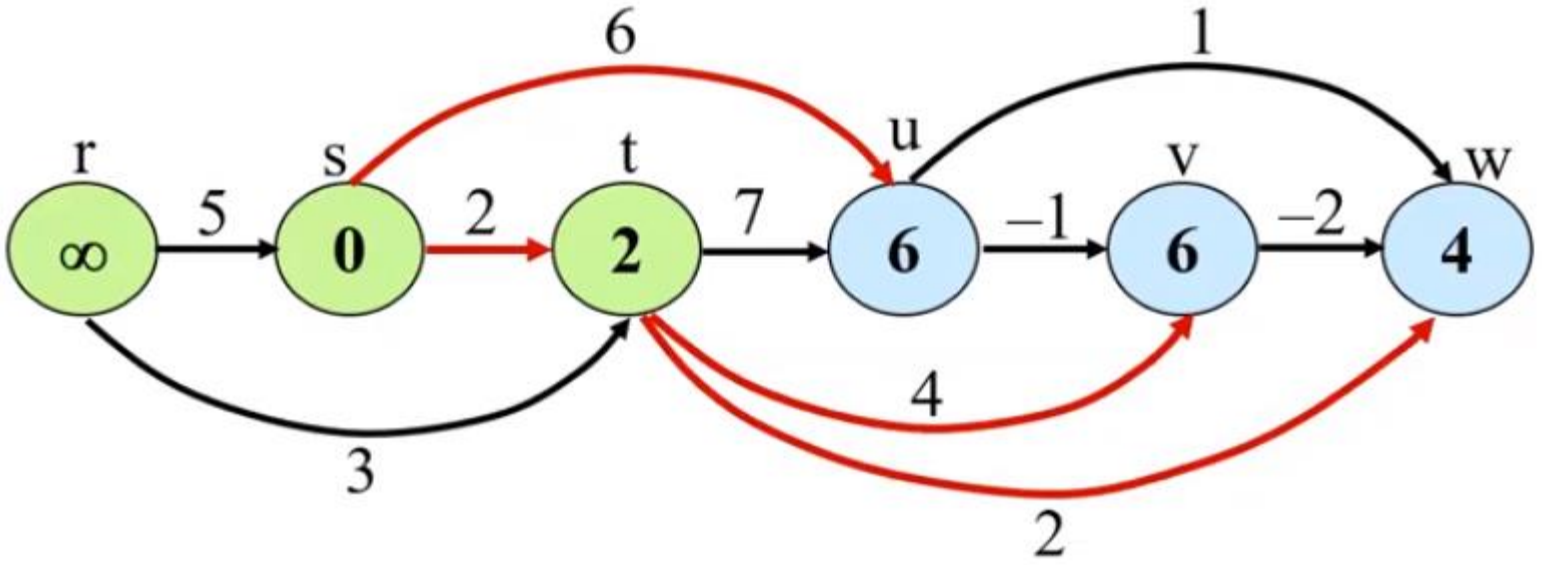
- Next iteration, check the adjacents of t




- 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

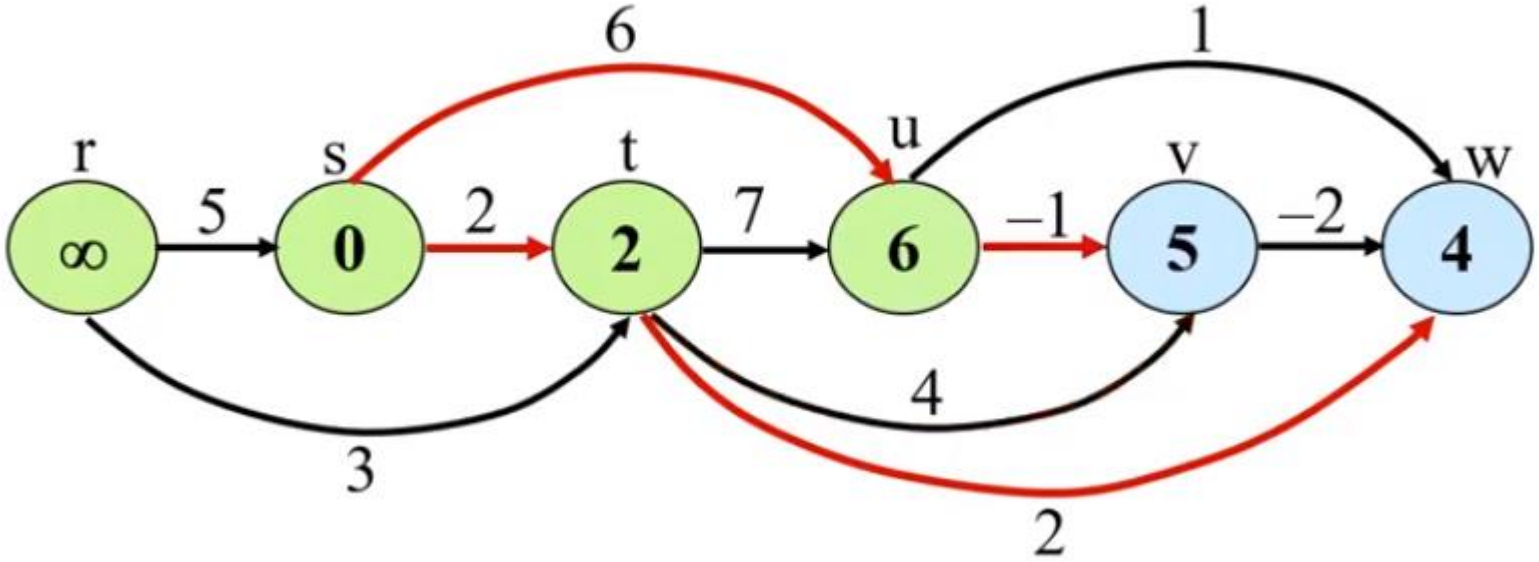
- Next iteration, check the adjacents of u




- 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

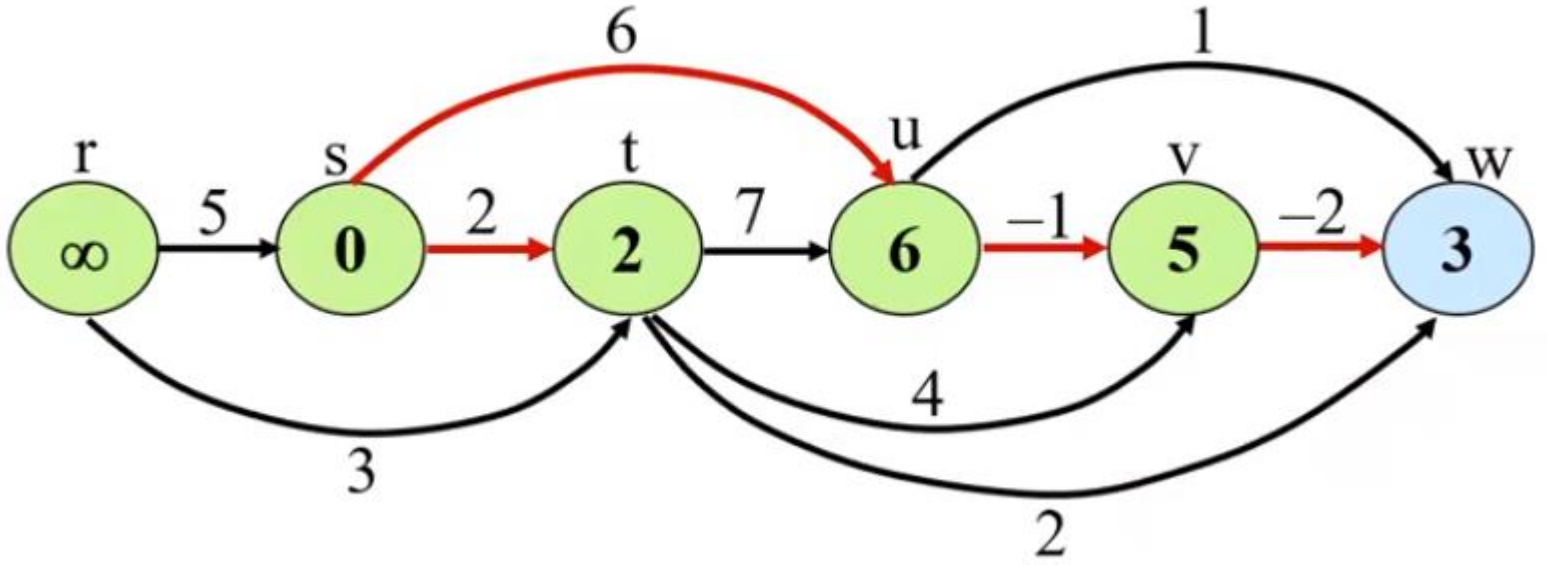
- Next iteration, check the adjacents of v




- 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