Tribhuvan University
Institute of Engineering
Purwanchal Campus
पूर्वाञ्चल क्याम्पस

4. Graph Algorithms

Pukar Karki
Assistant Professor

Single-Source Shortest Paths

- In a shortest-paths problem, we are given a weighted, directed graph G = (V, E).
- The weight w(p) of path p = (V_0, V_1…. V_k) is the sum of the weights of its constituent edges:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

Single-Source Shortest Paths

- We define the shortest-path weight δ(u, v)from u to v by

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \overset{p}{\leadsto} v\} & \text{if there is a path from } u \text{ to } v \\ \infty & \text{otherwise .} \end{cases}$$

Single-Source Shortest Paths: Variants

- The algorithm for the single-source problem can solve many other problems, including the following variants.

  Single-destination shortest-paths problem: Find a shortest path to a given destination vertex t from each vertex v.

  Single-pair shortest-path problem: Find a shortest path from u to v for given vertices u and v. If we solve the single-source problem with source vertex u, we solve this problem also.

  All-pairs shortest-paths problem: Find a shortest path from u to v for every pair of vertices u and v.

Relaxation

- The algorithms in this chapter use the technique of relaxation.
- For each vertex **v** $\in$ **V** , we maintain an attribute **v.d**, which is an upper bound on the weight of a shortest path from source **s** to **v**.
- We call **v.d** a shortest-path estimate.
- We initialize the shortest-path estimates and predecessors by the following **Θ(V)** time procedure.

Relaxation

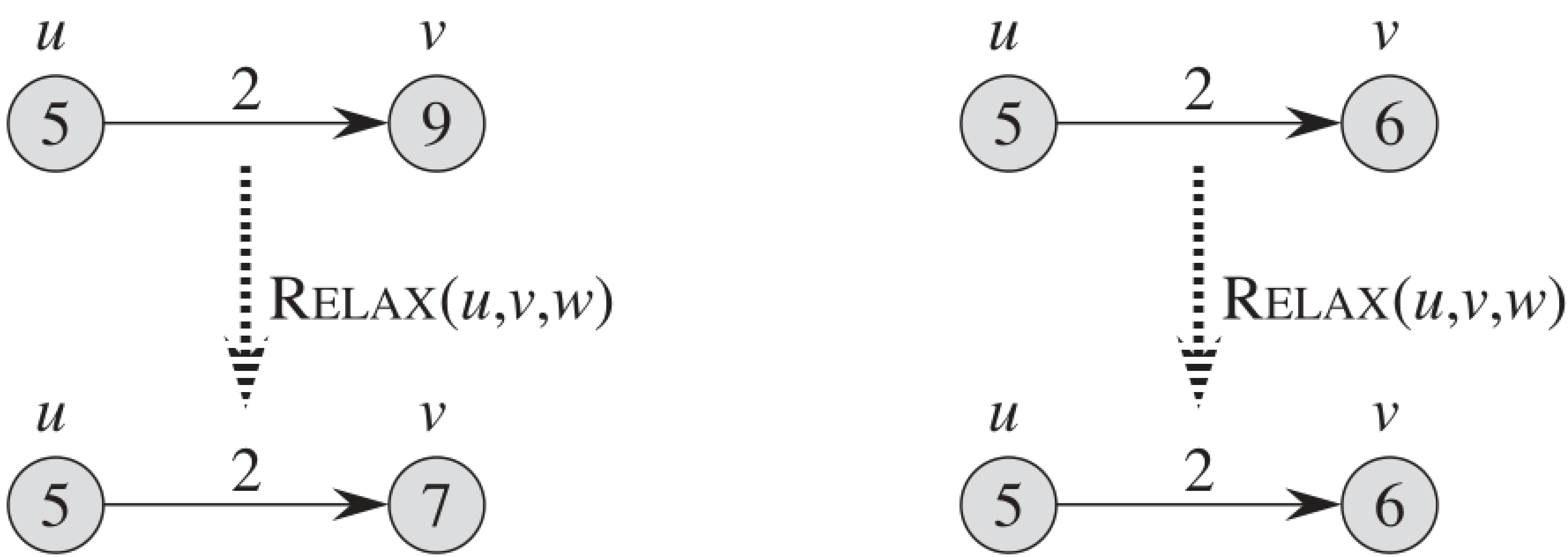
INITIALIZE-SINGLE-SOURCE(G, s)

```
for each vertex v ∈ G.V
    v.d = ∞
    v.π = NIL
s.d = 0
```

After initialization, we have **v.π = NIL** for all **v** ∈ **V** , **s.d = 0**, and **v.d = ∞** for **v** ∈ **V - {s}.**

Relaxation

- The process of relaxing an edge (u, v) consists of testing whether we can improve the shortest path to **v** found so far by going through **u** and, if so, updating **v.d** and **v.π**.



Relaxation

- The following code performs a relaxation step on edge **(u, v)** in **O(1)** time:

```
RELAX(u, v, w)
if v.d > u.d + w(u, v)
    v.d = u.d + w(u, v)
    v.π = u
```

The Bellman-Ford algorithm

- The Bellman-Ford algorithm solves the single-source shortest-paths problem in the general case in which edge weights may be negative.

- Given a weighted, directed graph G = (V, E) with source s , the Bellman-Ford algorithm returns a boolean value indicating whether or not there is a negative-weight cycle that is reachable from the source.

- If there is such a cycle, the algorithm indicates that no solution exists.

- If there is no such cycle, the algorithm produces the shortest paths and their weights.

The Bellman-Ford algorithm

- The algorithm returns TRUE if and only if the graph contains no negative-weight cycles that are reachable from the source.

```
BELLMAN-FORD(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
for i = 1 to |G.V| - 1
    for each edge (u, v) ∈ G.E
        RELAX(u, v, w)
for each edge (u, v) ∈ G.E
    if v.d > u.d + w(u, v)
        return FALSE
return TRUE
```

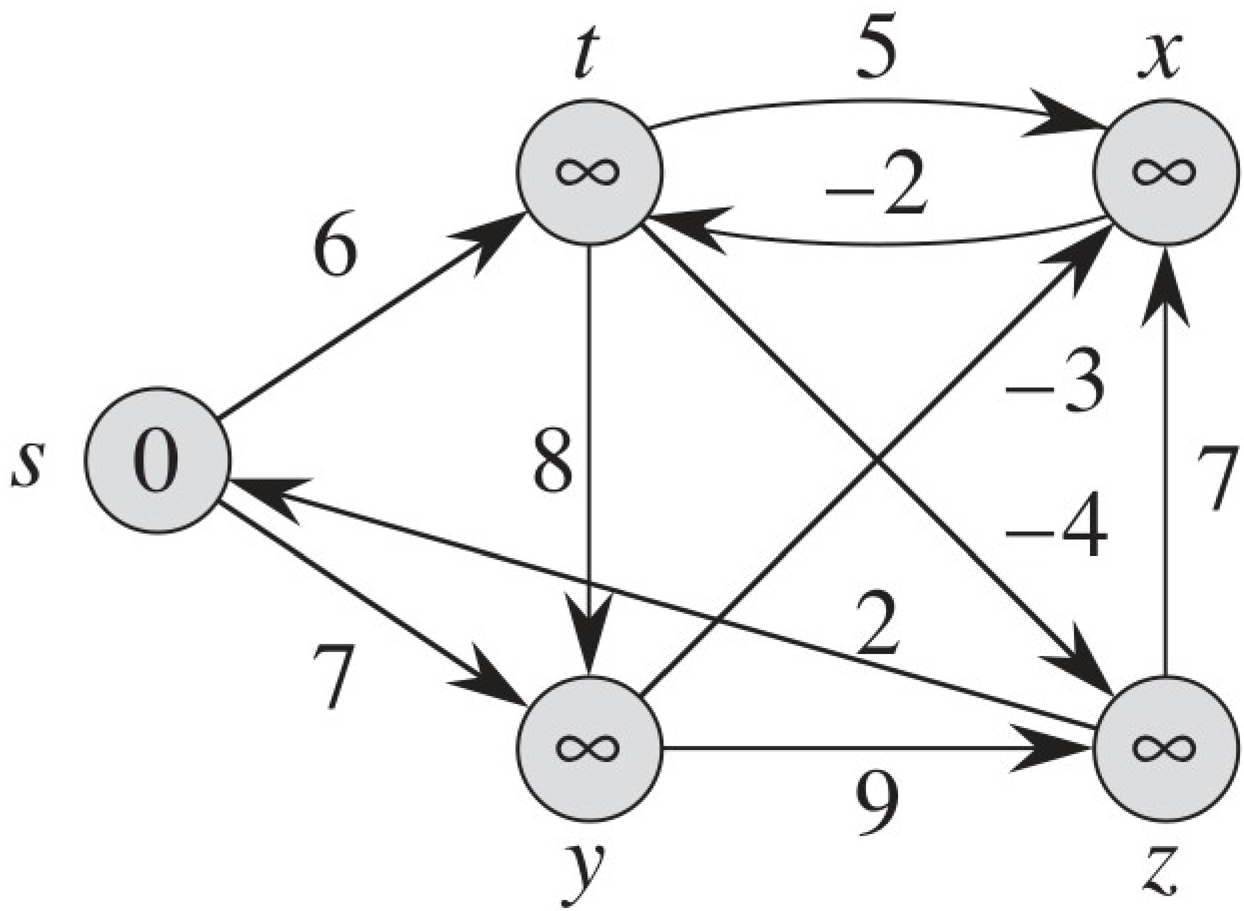
The Bellman-Ford algorithm

- The algorithm returns TRUE if and only if the graph contains no negative-weight cycles that are reachable from the source.

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| - 1
3      for each edge (u, v) ∈ G.E
4          RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6      if v.d > u.d + w(u, v)
7          return FALSE
8  return TRUE
```

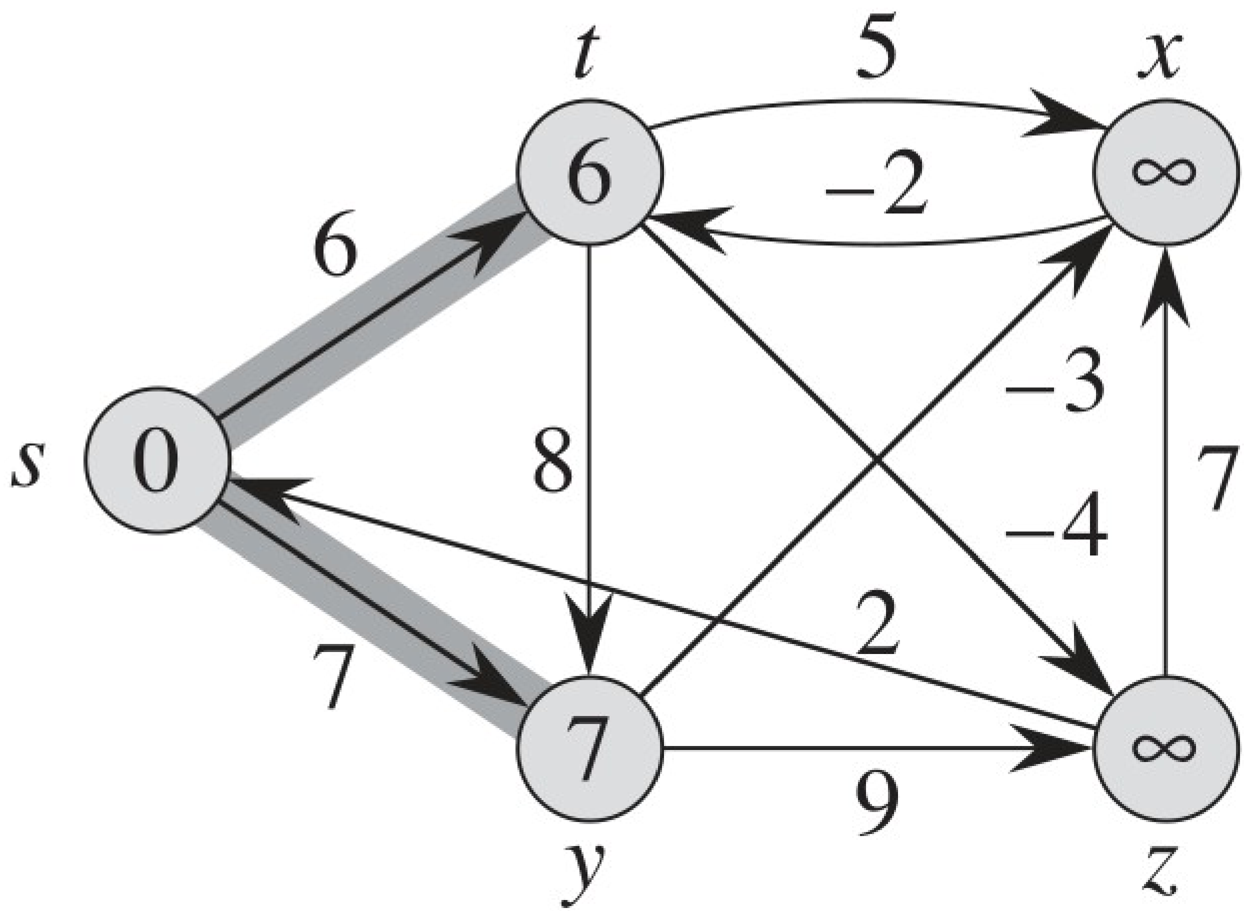
- The Bellman-Ford algorithm runs in time **O(V.E)**, since the initialization in line 1 takes Θ(V) time, each of the |V| - 1 passes over the edges in lines 2–4 takes Θ(E) time, and the for loop of lines 5-7 takes O(E) time.

The Bellman-Ford algorithm



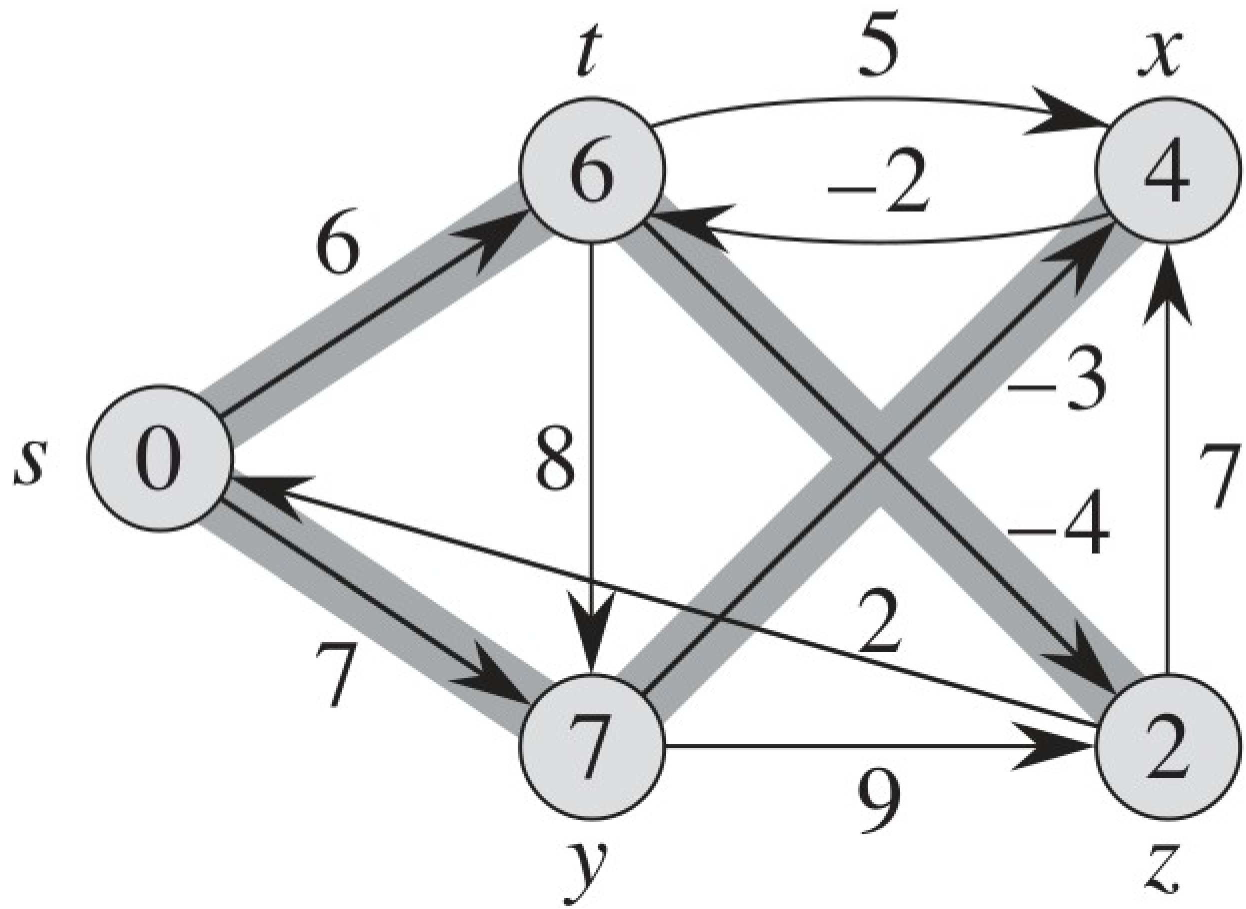
(a)

The Bellman-Ford algorithm



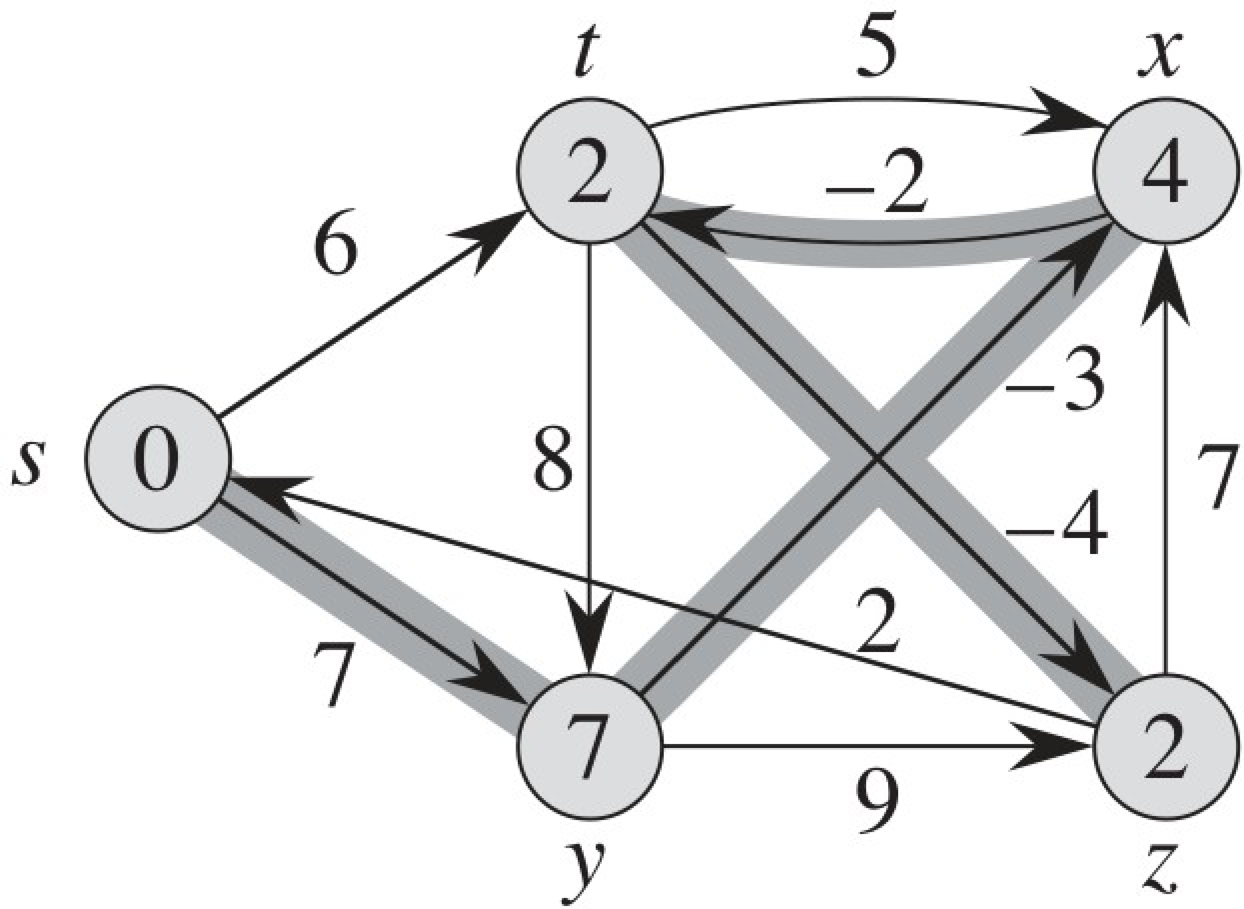
(b)

The Bellman-Ford algorithm



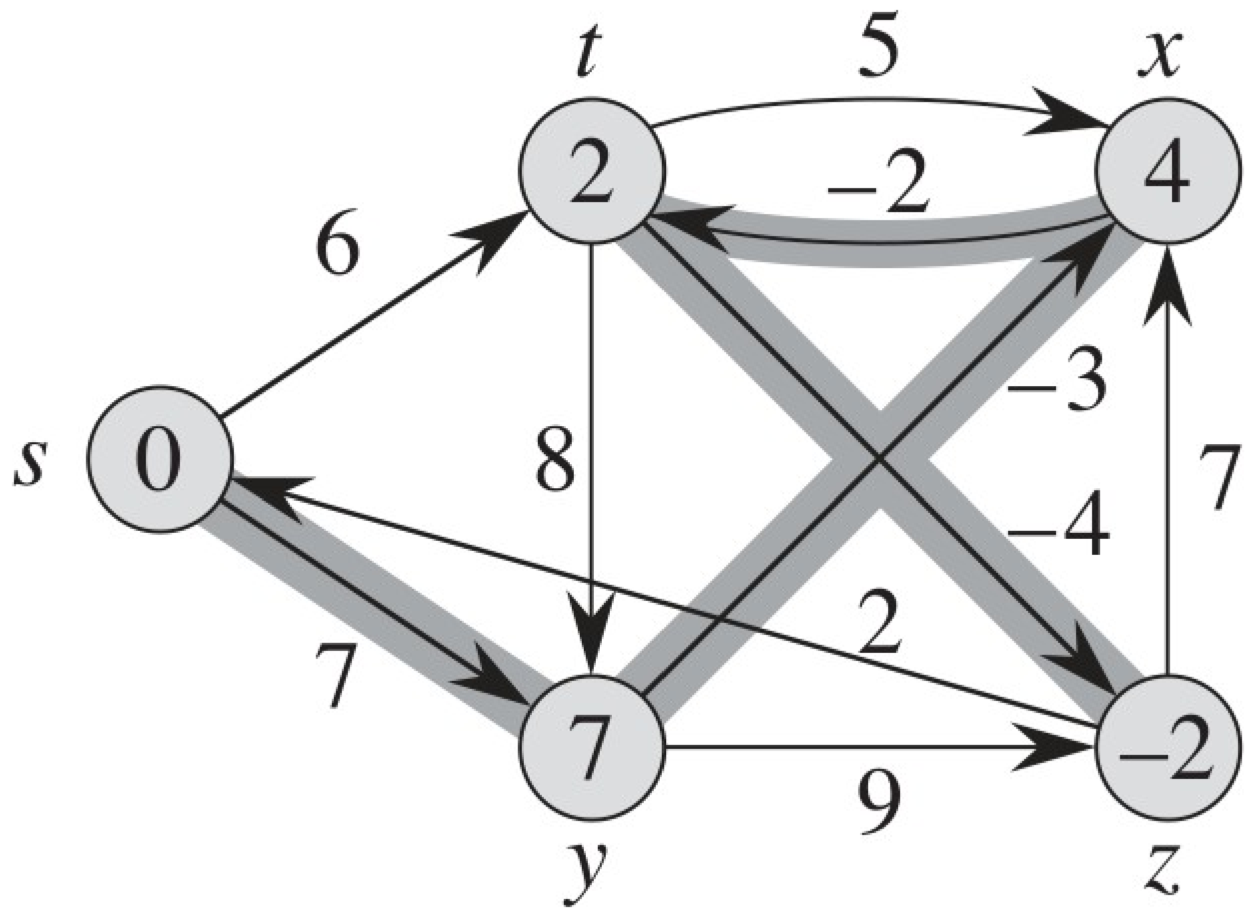
(c)

The Bellman-Ford algorithm



(d)

The Bellman-Ford algorithm



(e)

Dijkstra’s algorithm

- Dijkstra’s algorithm solves the single-source shortest-paths problem on a weighted, directed graph $G = (V, E)$ for the case in which all edge weights are non-negative.
- In this section, therefore, we assume that $w(u, v) \geq 0$ for each edge $(u, v) \in E$.
- As we shall see, with a good implementation, the running time of Dijkstra’s algorithm is lower than that of the Bellman-Ford algorithm.

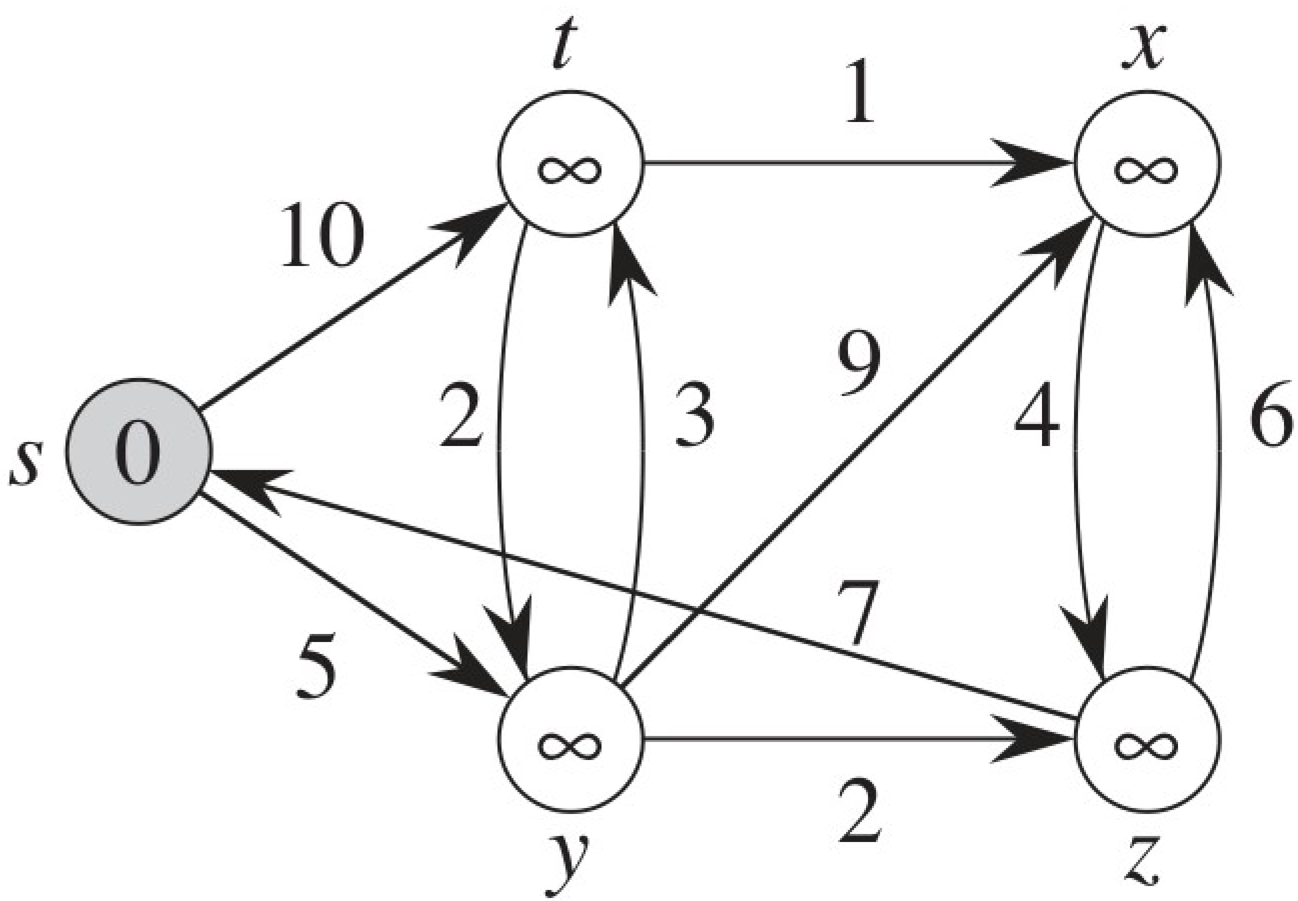
Dijkstra's algorithm

- Dijkstra's algorithm maintains a set S of vertices whose final shortest-path weights from the source s have already been determined.
- The algorithm repeatedly selects the vertex $u \in V - S$ with the minimum shortest-path estimate, adds u to S, and relaxes all edges leaving u.
- In the following implementation, we use a min-priority queue Q of vertices, keyed by their d values.

Dijkstra’s algorithm

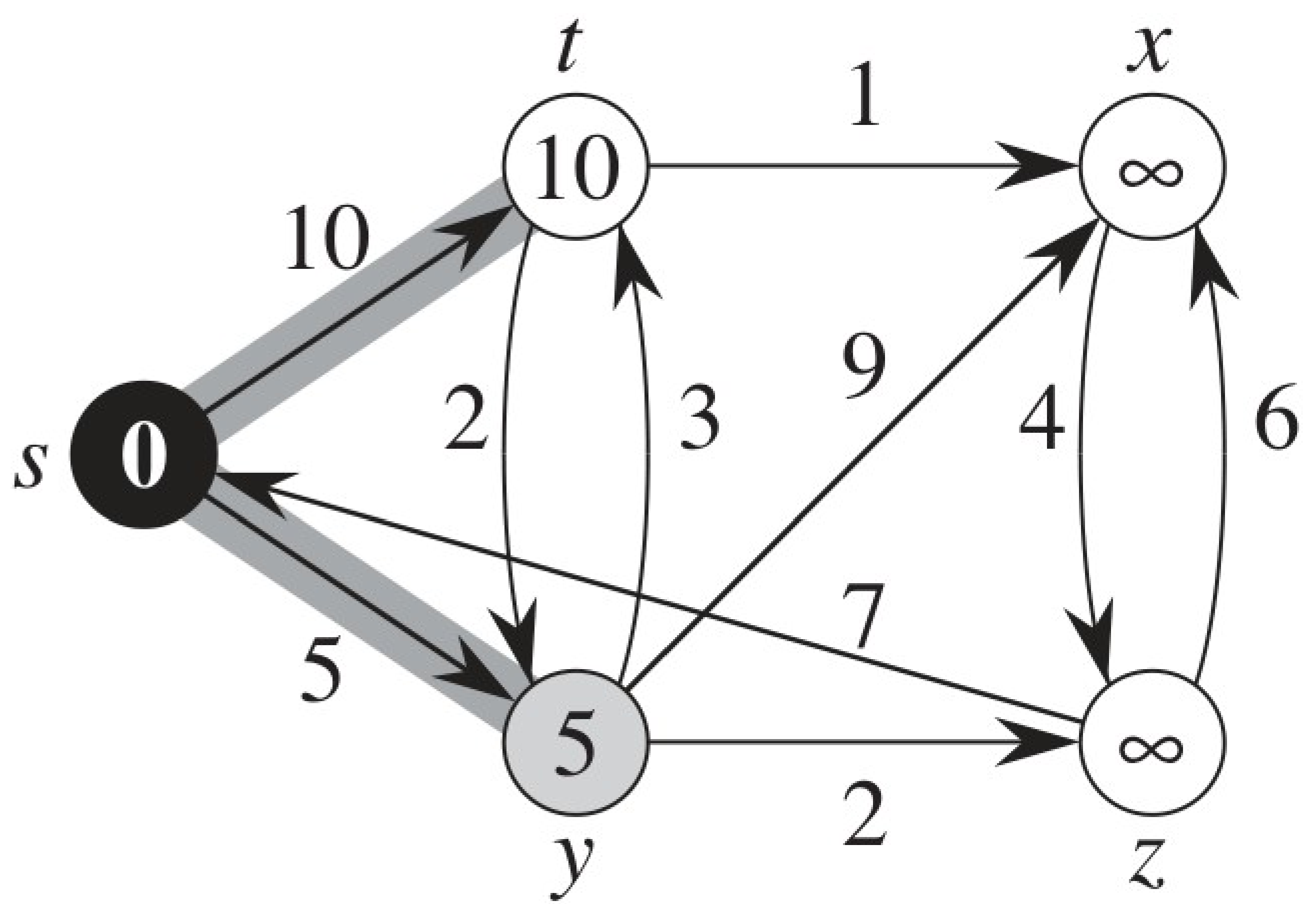
```
DIJKSTRA(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
S = ∅
Q = G.V
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    S = S ∪ {u}
    for each vertex v ∈ G.Adj[u]
        RELAX(u, v, w)
```

Dijkstra’s algorithm



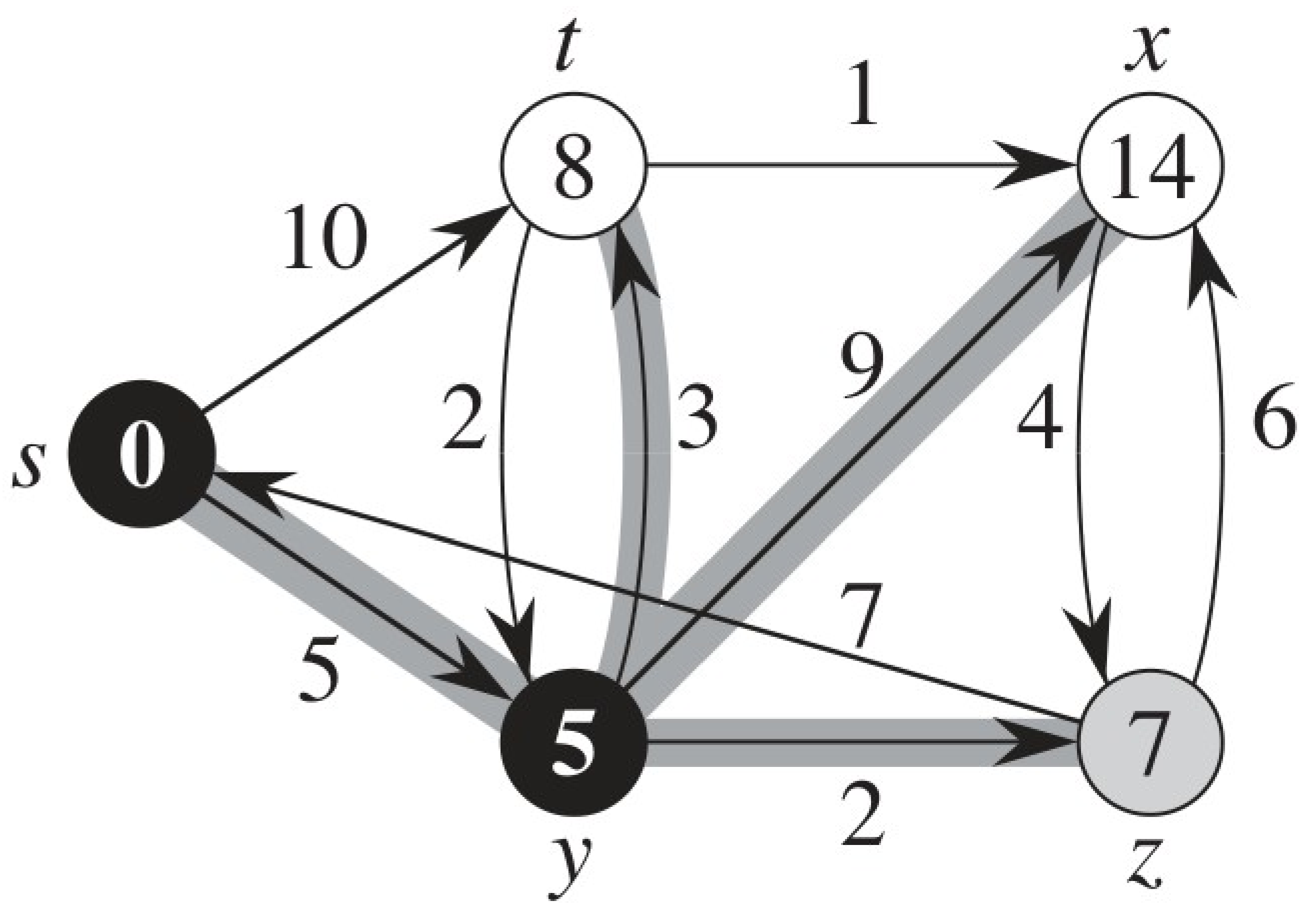
(a)

Dijkstra’s algorithm



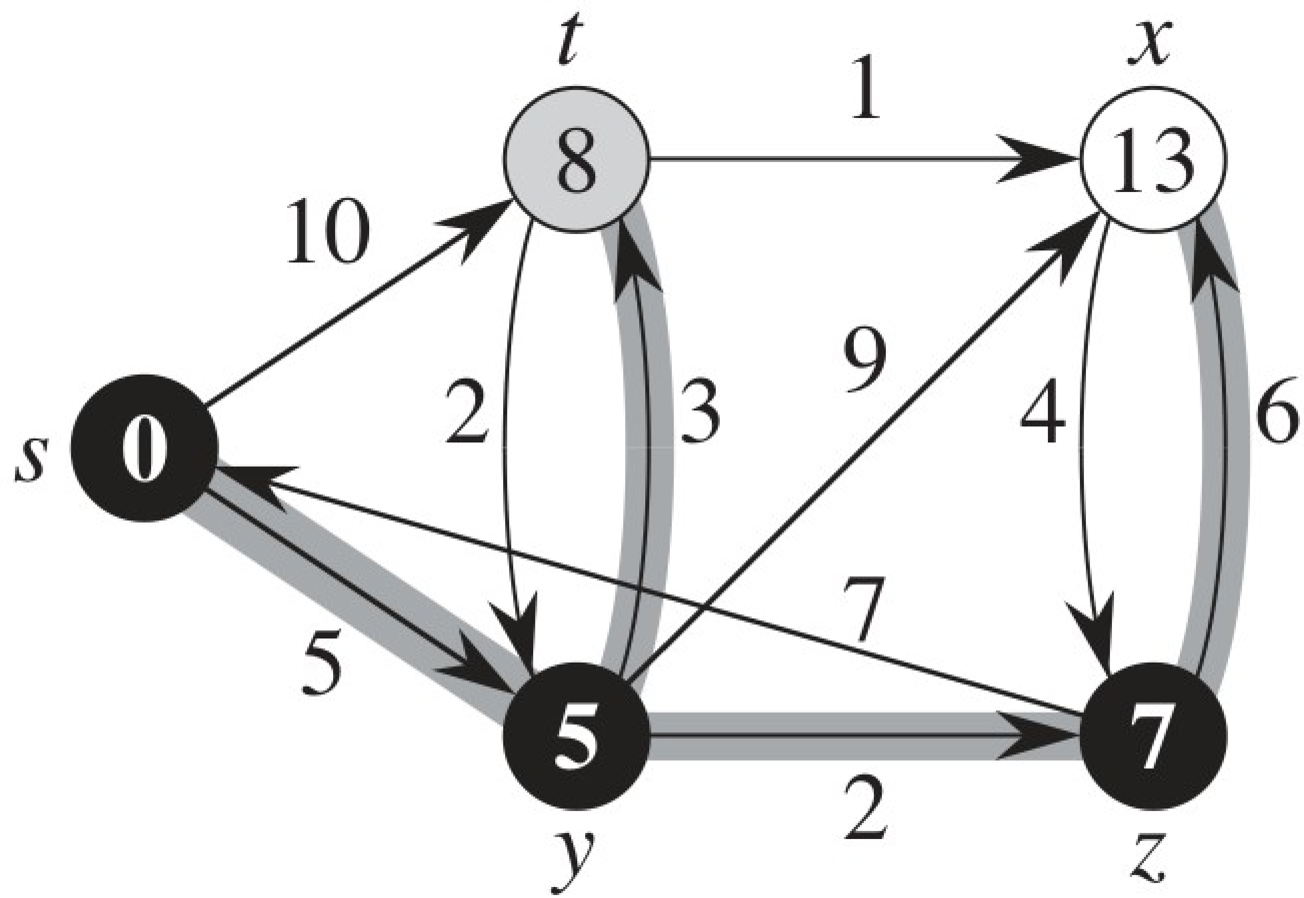
(b)

Dijkstra's algorithm



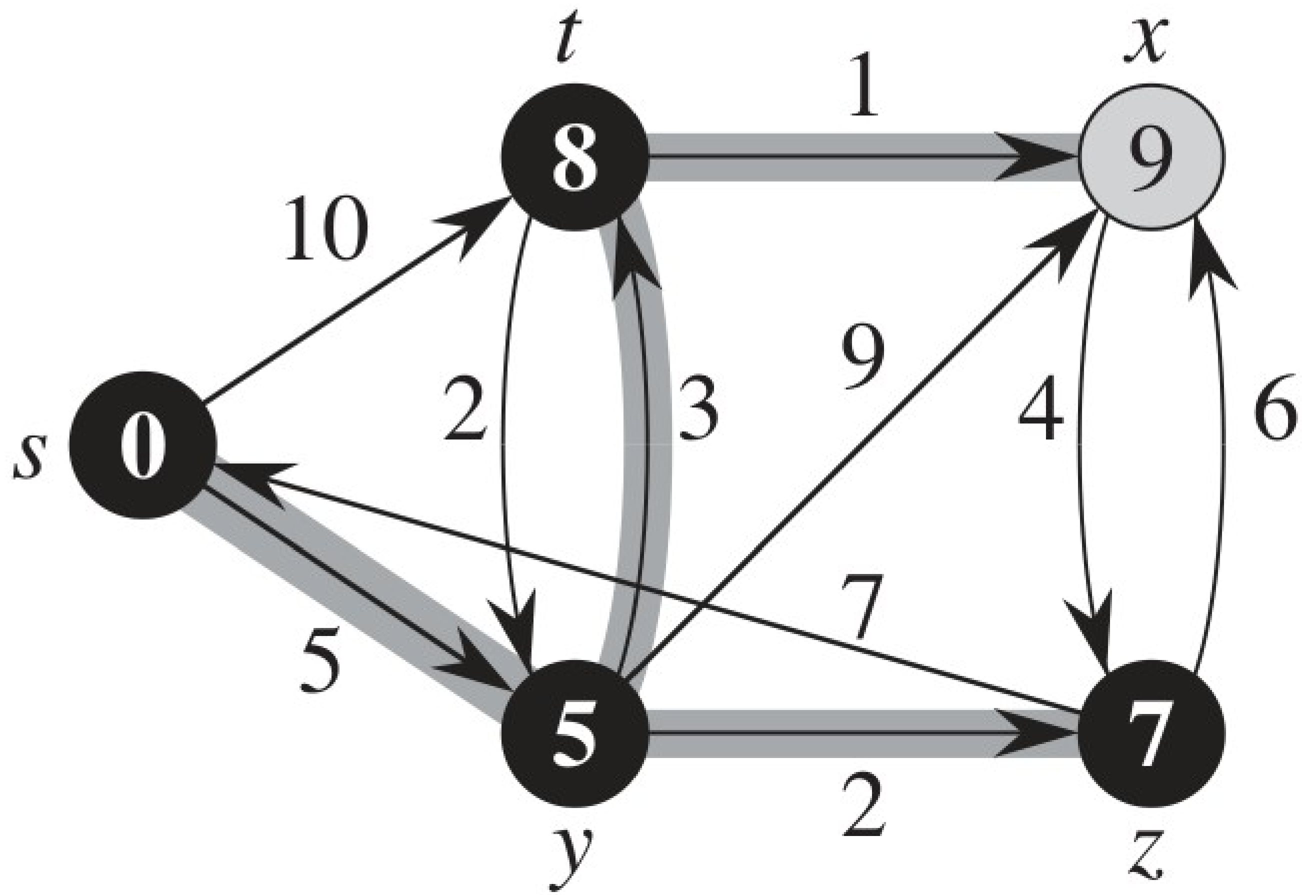
(c)

Dijkstra's algorithm



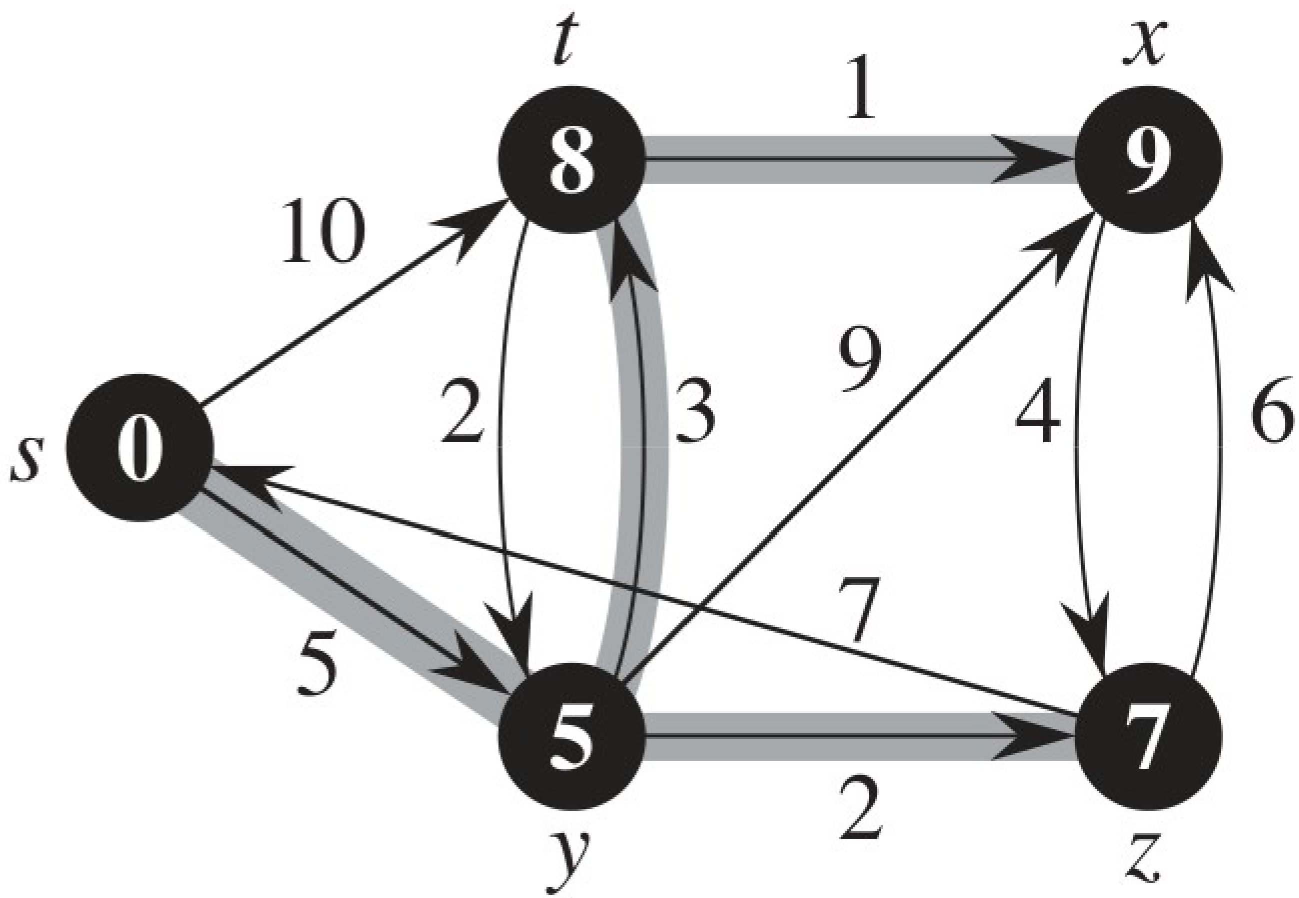
(d)

Dijkstra’s algorithm



(e)

Dijkstra’s algorithm



(f)

Dijkstra’s algorithm

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5      u = EXTRACT-MIN(Q)
6      S = S ∪ {u}
7      for each vertex v ∈ G.Adj[u]
8          RELAX(u, v, w)
```

- It maintains the min-priority queue Q by calling three priority-queue operations: INSERT (implicit in line 3), EXTRACT-MIN (line 5), and DECREASE-KEY (implicit in RELAX, which is called in line 8).
- The algorithm calls both INSERT and EXTRACT-MIN once per vertex.
- Because each vertex u ∈ V is added to set S exactly once, each edge in the adjacency list Adj[u] is examined in the for loop of lines 7–8 exactly once during the course of the algorithm.

Dijkstra’s algorithm

```
DIJKSTRA(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
S = ∅
Q = G.V
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    S = S ∪ {u}
    for each vertex v ∈ G.Adj[u]
        RELAX(u, v, w)
```

- Since the total number of edges in all the adjacency lists is |E|, this for loop iterates a total of |E| times, and thus the algorithm calls DECREASE-KEY at most |E| times overall.

Dijkstra’s algorithm

DIJKSTRA(G, w, s)

```
INITIALIZE-SINGLE-SOURCE(G, s)
S = ∅
Q = G.V
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    S = S ∪ {u}
    for each vertex v ∈ G.Adj[u]
        RELAX(u, v, w)
```

- The running time of Dijkstra’s algorithm depends on how we implement the min-priority queue.

- Consider first the case in which we maintain the min-priority queue by taking advantage of the vertices being numbered 1 to |V|.
- We simply store v.d in the v^{th} entry of an array.
- Each INSERT and DECREASE-KEY operation takes O(1) time, and each EXTRACT-MIN operation takes O(V) time (since we have to search through the entire array), for a total time of

  $$O(V^2 + E) = O(V^2).$$

Dijkstra’s algorithm

```
DIJKSTRA(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
S = ∅
Q = G.V
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    S = S ∪ {u}
    for each vertex v ∈ G.Adj[u]
        RELAX(u, v, w)
```

- The running time of Dijkstra’s algorithm depends on how we implement the min-priority queue.
- If the graph is sufficiently sparse, we can improve the algorithm by implementing the min-priority queue with a binary min-heap.
- Each EXTRACT-MIN operation then takes time O(lg V).
- As before, there are |V| such operations.
- The time to build the binary min-heap is O(V).
- Each DECREASE-KEY operation takes time O(lg V), and there are still at most |E| such operations.
- The total running time is therefore O((V + E) lg V), which is O(E lg V) if all vertices are reachable from the source.

All-Pairs Shortest Paths

- Consider the problem of finding shortest paths between all pairs of vertices in a graph.
- We are given a weighted directed graph G = (V, E).
- We wish to find, for every pair of vertices u, v $\in$ V , a shortest (least-weight) path from u to v, where the weight of a path is the sum of the weights of its constituent edges.

All-Pairs Shortest Paths

- We can solve an all-pairs shortest-paths problem by running a single-source shortest-paths algorithm |V| times, once for each vertex as the source.
- If all edge weights are non-negative, we can use Dijkstra’s algorithm.
- If we use the linear-array implementation of the min-priority queue, the running time is

$$O(V^3 + VE) = O(V^3)$$

All-Pairs Shortest Paths

- The binary min-heap implementation of the min-priority queue yields a running time of O(VE lg V), which is an improvement if the graph is sparse.
- If the graph has negative-weight edges, we cannot use Dijkstra's algorithm. Instead, we must run the slower Bellman-Ford algorithm once from each vertex.
- The resulting running time is $O(V^2E)$, which on a dense graph is $O(V^4)$.

All-Pairs Shortest Paths

- Unlike the single-source algorithms, which assume an adjacency-list representation of the graph, most of the algorithms in this section use an adjacency-matrix representation.
- For convenience, we assume that the vertices are numbered 1, 2, …. |V|, so that the input is an n x n matrix W representing the edge weights of an n-vertex directed graph G = (V, E).
- That is, W = (w_{ij}), where

$$w_{ij} = \begin{cases} 0 & \text{if } i = j \ , \\ \text{the weight of directed edge } (i, j) & \text{if } i \neq j \text{ and } (i, j) \in E \ , \\ \infty & \text{if } i \neq j \text{ and } (i, j) \notin E \ . \end{cases}$$

All-Pairs Shortest Paths

- We allow negative-weight edges, but we assume for the time being that the input graph contains no negative-weight cycles.
- The tabular output of the all-pairs shortest-paths algorithms presented in this chapter is an n x n matrix $D = (d_{ij})$, where entry d_{ij} contains the weight of a shortest path from vertex i to vertex j

The Floyd-Warshall Algorithm

- Floyd-Warshall algorithm, runs in $\Theta(V^3)$ time and uses the notion of dynamic programming.
- This algorithm can be used to compute all pair shortest path.
- This algorithm works even if some of the edges have negative weights.

The Floyd-Warshall Algorithm

- Suppose, G = (V, E) is a weighted graph.
- Let W be the adjacency matrix of G.
- Let D^k be a n x n matrix such that $D^k(i, j)$ contains the weight of the shortest path from vertex i to vertex j using vertices 1,2, … k as intermediate vertices.
- We compute D^k as follows

$D^k = D^{k-1}(i, j)$ when k is not an intermediate vertex.

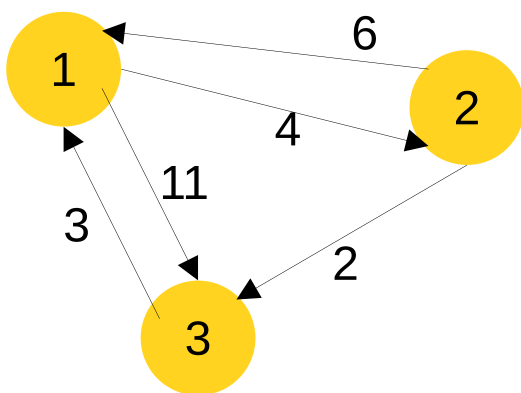
$D^k = D^{k-1}(i, k) + D^{k-1}(k, j)$ when k is an intermediate vertex

- The matrix D^n gives us all pair shortest path.

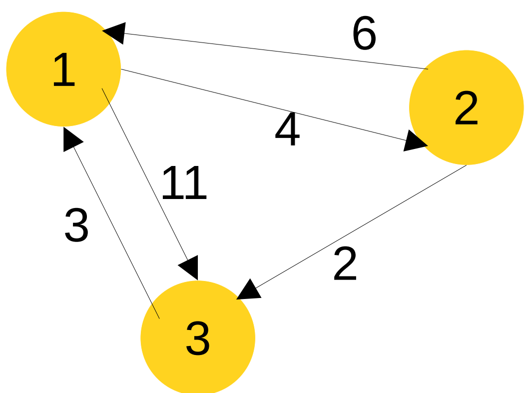
$D^k = \textbf{min[}D^{k-1}(i, j), D^{k-1}(i, k) + D^{k-1}(k, j)\textbf{]}$

The Floyd-Warshall Algorithm

Q)Find the shortest path from source vertex to every other vertices.



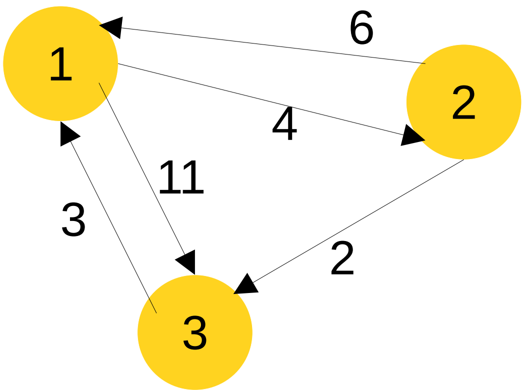
The Floyd-Warshall Algorithm



- The adjacency matrix can be computed as

W/D^0	1	2	3
1	0	4	11
2	6	0	2
3	3	inf	0

The Floyd-Warshall Algorithm



W/D°	1	2	3
1	0	4	11
2	6	0	2
3	3	inf	0

- When vertex 1 is an intermediate vertex, D^1 can be computed as

  $D^1(1, 1) = 0$

  $D^1(1, 2) = \min[D^0(1, 2), D^0(1, 1) + D^0(1, 2)] = \min[4, 0+4] = 4$

  $D^1(1, 3) = \min[D^0(1, 3), D^0(1, 1) + D^0(1, 3)] = \min[11, 0+11] = 11$

  $D^1(2, 1) = \min[D^0(2, 1), D^0(2, 1) + D^0(1, 1)] = \min[6, 6+0] = 6$

  $D^1(2, 2) = \min[D^0(2, 2), D^0(2, 1) + D^0(1, 2)] = \min[0, 6 +10] = 0$

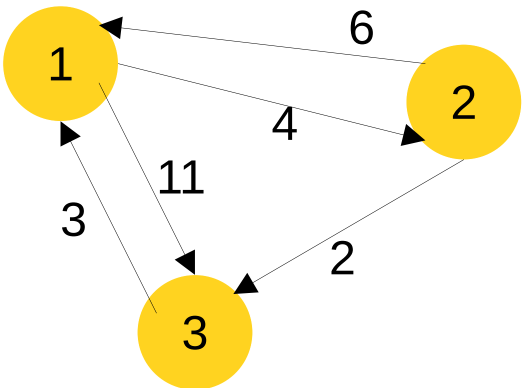
  $D^1(2, 3) = \min[D^0(2, 3), D^0(2, 1) + D^0(1, 3)] = \min[2, 6+11] = 2$

  $D^1(3, 1) = \min[D^0(3, 1), D^0(3, 1) + D^0(1, 1)] = \min[3, 3 + 0] = 3$

  $D^1(3, 2) = \min[D^0(3, 2), D^0(3, 1) + D^0(1, 2)] = \min[\text{inf}, 3+4] = 7$

  $D^1(3, 3) = \min[D^0(3, 3), D^0(3, 1) + D^0(1, 3)] = \min[0, 3+11] = 0$

The Floyd-Warshall Algorithm



D^1	1	2	3
1	0	4	11
2	6	0	2
3	3	7	0

- When vertex 2 is an intermediate vertex, D^2 can be computed as

  $D^2(1, 1) = 0$

  $D^2(1, 2) = \min[D^1(1, 2), D^1(1, 2) + D^1(2, 2)] = \min[4, 4+0] = 4$

  $D^2(1, 3) = \min[D^1(1, 3), D^1(1, 2) + D^1(2, 3)] = \min[11, 4+2] = 6$

  $D^2(2, 1) = 6$

  $D^2(2, 2) = 0$

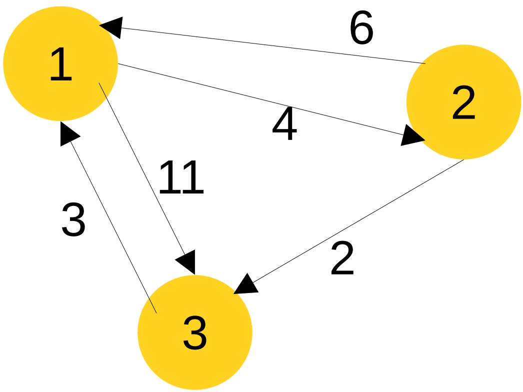
  $D^2(2, 3) = 2$

  $D^2(3, 1) = \min[D^1(3, 1), D^1(3, 2) + D^1(2, 1)] = \min[3, 7 + 6] = 3$

  $D^2(3, 2) = 7$

  $D^2(3, 3) = 0$

The Floyd-Warshall Algorithm



D^2	1	2	3
1	0	4	6
2	6	0	2
3	3	7	0

- When vertex 3 is an intermediate vertex, D^3 can be computed as

  $D^3(1, 1) = 0$

  $D^3(1, 2) = \min[D^2(1, 2), D^2(1, 3) + D^2(3, 2)] = \min[4, 6+7] = 4$

  $D^3(1, 3) = 6$

  $D^3(2, 1) = \min[D^2(2, 1), D^2(2, 3) + D^2(3, 1)] = \min[6, 2 + 3] = 5$

  $D^3(2, 2) = 0$

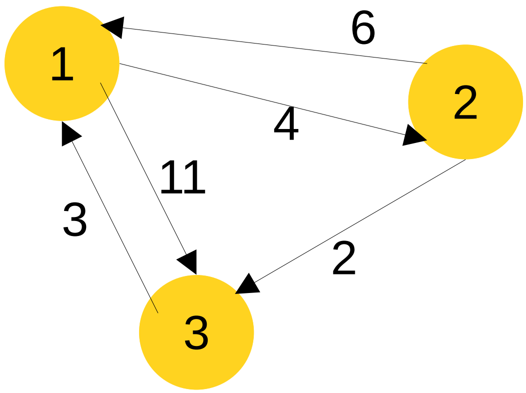
  $D^3(2, 3) = 2$

  $D^3(3, 1) = 3$

  $D^3(3, 2) = 7$

  $D^3(3, 3) = 0$

The Floyd-Warshall Algorithm



D^3	1	2	3
1	0	4	6
2	5	0	2
3	3	7	0

The Floyd-Warshall Algorithm

- Let W be a matrix that contains weight of each edges of G, n be the number of nodes.

The Floyd-Warshall Algorithm

FLOYD-WARSHALL(W)

$n = W.rows$
$D^{(0)} = W$
for $k = 1$ **to** n
    let $D^{(k)} = \left(d_{ij}^{(k)}\right)$ be a new $n \times n$ matrix
    for $i = 1$ **to** n
        for $j = 1$ **to** n
            $d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$
return $D^{(n)}$

The Floyd-Warshall Algorithm

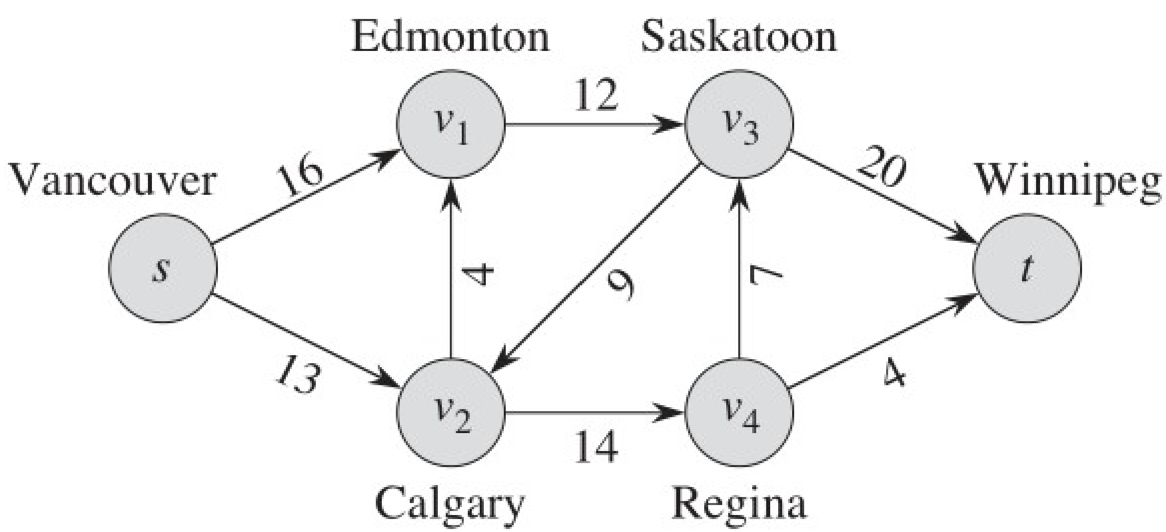
FLOYD-WARSHALL(W)

1 $n = W.rows$
2 $D^{(0)} = W$
3 **for** $k = 1$ **to** n
4   let $D^{(k)} = \left(d_{ij}^{(k)}\right)$ be a new $n \times n$ matrix
5   **for** $i = 1$ **to** n
6   **for** $j = 1$ **to** n
7             $d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$
8 **return** $D^{(n)}$

- The running time of the Floyd-Warshall algorithm is determined by the triply nested for loops of lines 3–7. Because each execution of line 7 takes O(1) time, the algorithm runs in time $\Theta(n^3)$.
- The code is tight, with no elaborate data structures, and so the constant hidden in the notation is small.
- Thus, the Floyd-Warshall algorithm is quite practical for even moderate-sized input graphs.

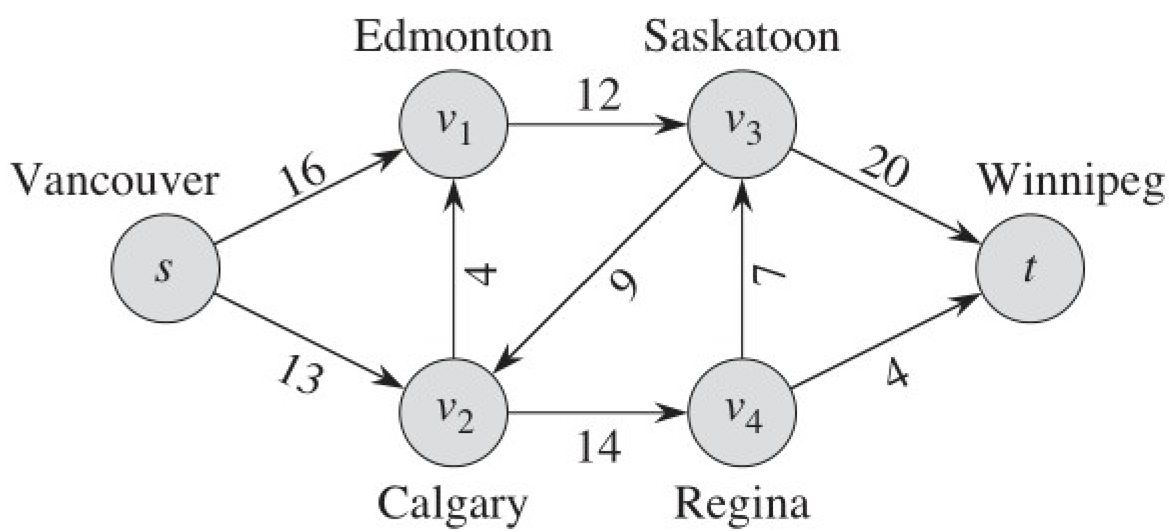
Flow Networks

- Imagine a material coursing through a system from a source, where the material is produced, to a sink, where it is consumed.
- The source produces the material at some steady rate, and the sink consumes the material at the same rate.
- The "flow" of the material at any point in the system is intuitively the rate at which the material moves.



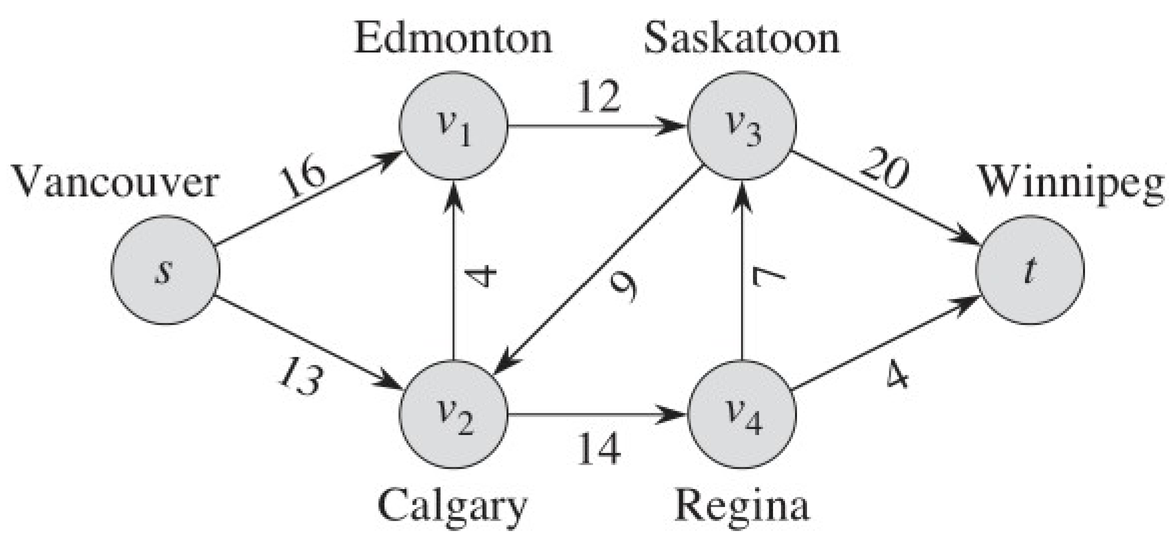
Flow Networks

- Flow networks can model many problems, including liquids flowing through pipes, parts through assembly lines, current through electrical networks, and information through communication networks.



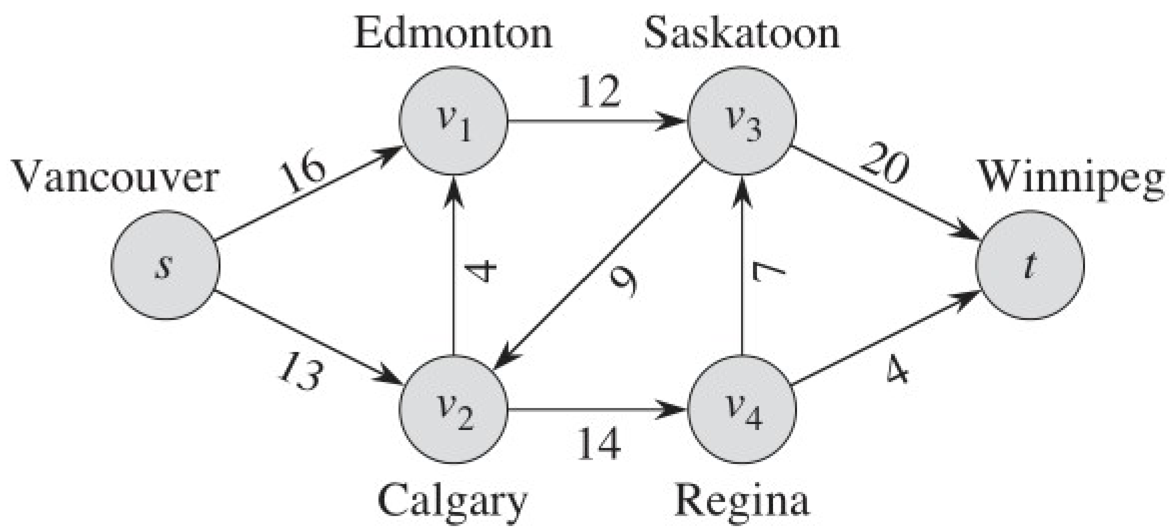
Flow Networks

- A flow network G = (V, E) is a directed graph in which each edge (u, v) $\in$ E has a non-negative capacity c(u, v) $\geq$ 0.
- We further require that if E contains an edge (u,v), then there is no edge (v, u) in the reverse direction.



Flow Networks

- We distinguish two vertices in a flow network: a source s and a sink t.



Flow Networks

- Let G = (V, E) be a flow network with a capacity function c.
- Let s be the source of the network, and let t be the sink.

- A flow in G is a real-valued function f : V x V ⟶ R that satisfies the following two properties:

Capacity constraint: For all $u, v \in V$, we require $0 \leq f(u, v) \leq c(u, v)$.

Flow conservation: For all $u \in V - \{s, t\}$, we require

$$\sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v) .$$

When $(u, v) \notin E$, there can be no flow from u to v, and $f(u, v) = 0$.

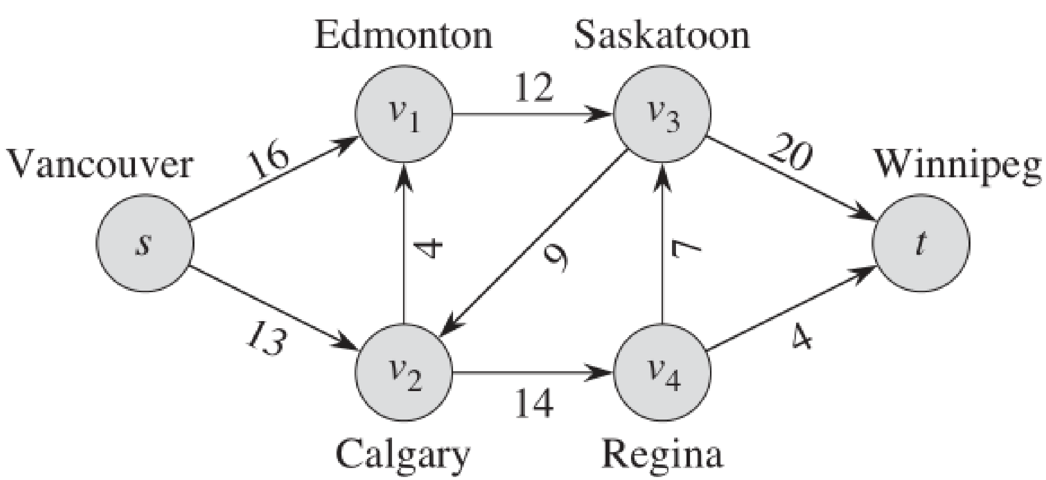
Flow Networks

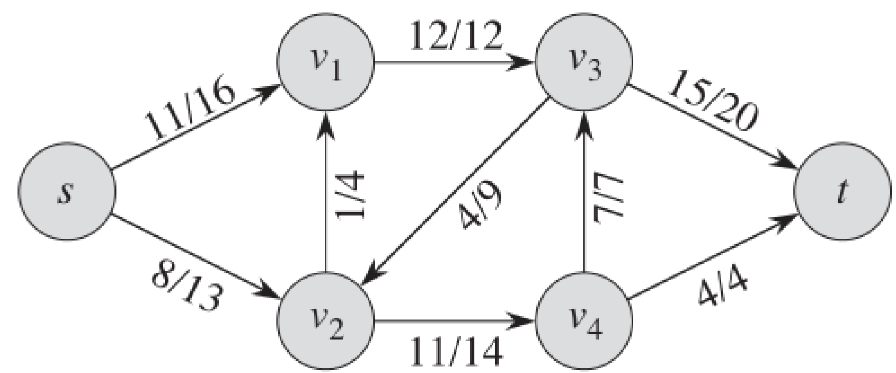
Capacity constraint: For all $u, v \in V$, we require $0 \le f(u, v) \le c(u, v)$.

Flow conservation: For all $u \in V - \{s, t\}$, we require

$$\sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v) .$$

When $(u, v) \notin E$, there can be no flow from u to v, and $f(u, v) = 0$.

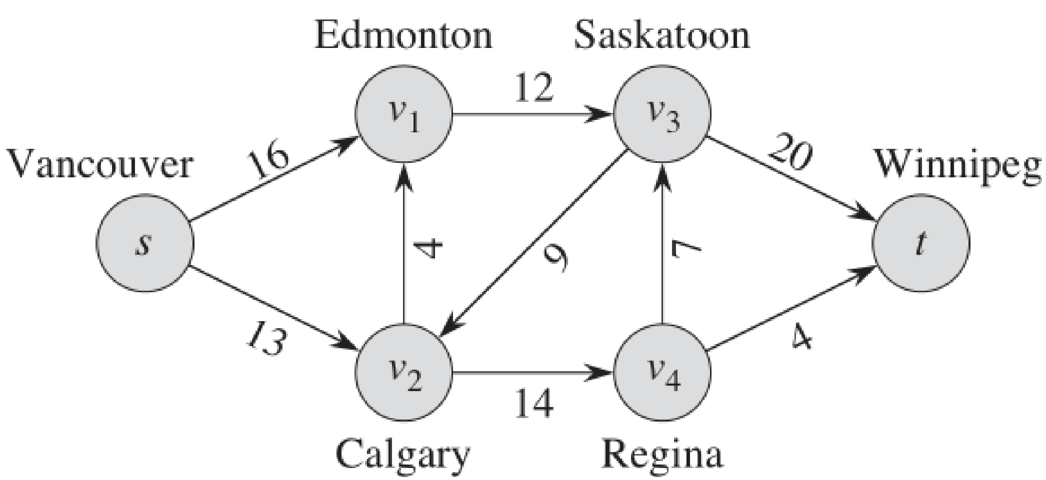


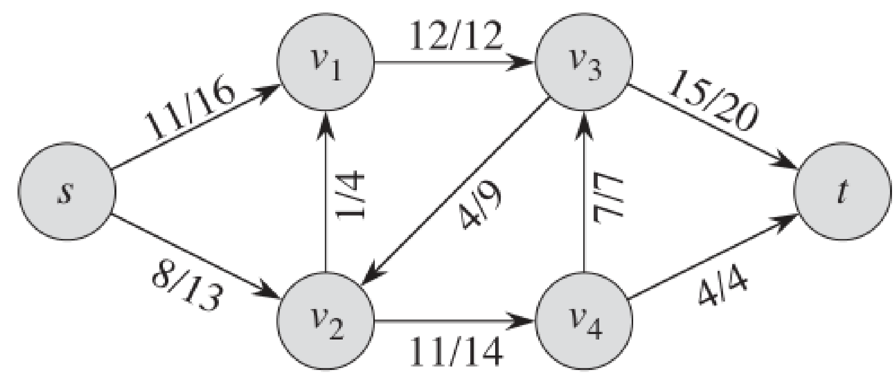


Flow Networks

- The capacity constraint simply says that the flow from one vertex to another must be non-negative and must not exceed the given capacity.

- The flow-conservation property says that the total flow into a vertex other than the source or sink must equal the total flow out of that vertex—informally, “flow in equals flow out.”





Maximum Flow

- In the maximum-flow problem, we are given a flow network G with source s and sink t, and we wish to find a flow of maximum value.

The Ford-Fulkerson method

- Ford-Fulkerson method is used for solving the maximum-flow problem.
- We call it a “method” rather than an “algorithm” because it encompasses several implementations with differing running times.
- The Ford-Fulkerson method depends on three important ideas that transcend the method and are relevant to many flow algorithms and problems:

  - residual networks,

  - augmenting paths, and

  - cuts.

The Ford-Fulkerson method

```
FORD-FULKERSON-METHOD(G, s, t)
1  initialize flow f to 0
2  while there exists an augmenting path p in the residual network G_f
3      augment flow f along p
4  return f
```

The Ford-Fulkerson method

Residual Networks

- Intuitively, given a flow network G and a flow f , the residual network G_f consists of edges with capacities that represent how we can change the flow on edges of G.
- An edge of the flow network can admit an amount of additional flow equal to the edge's capacity minus the flow on that edge.
- If that value is positive, we place that edge into G_f with a "residual capacity" of

$$c_f(u,v) = c(u,v) - f(u,v)$$

The Ford-Fulkerson method

Residual Networks

- As an algorithm manipulates the flow, with the goal of increasing the total flow, it might need to decrease the flow on a particular edge.
- In order to represent a possible decrease of a positive flow f(u, v) on an edge in G, we place an edge (v, u) into G_f with residual capacity

$c_f(v, u) = f(u, v)$ that is, an edge that can admit flow in the opposite direction to (u, v), at most canceling out the flow on (u, v).

The Ford-Fulkerson method

Residual Networks

- Formally, suppose that we have a flow network G = (V, E) with source s and sink t.
- Let f be a flow in G, and consider a pair of vertices u, v ∈ V.
- We define the residual capacity $c_f(u, v)$ by

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & \text{if } (u,v) \in E , \\ f(v,u) & \text{if } (v,u) \in E , \\ 0 & \text{otherwise} . \end{cases}$$

The Ford-Fulkerson method

Residual Networks

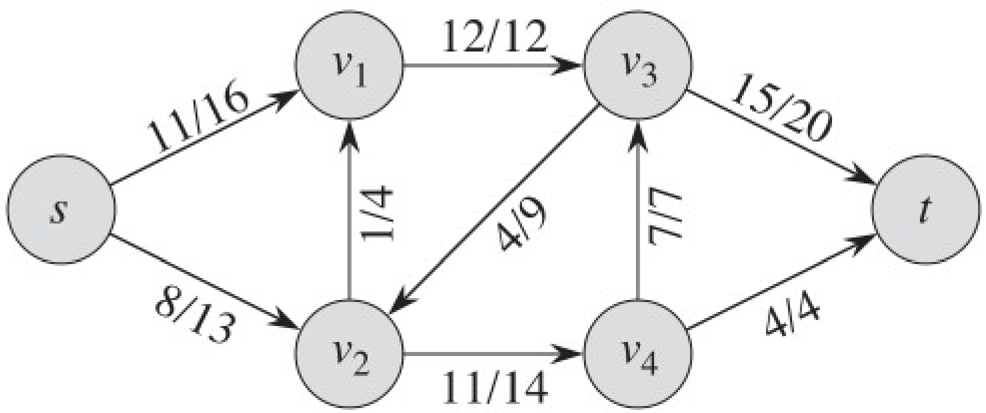
- Given a flow network G = (V, E) and a flow f , the residual network of G induced by f is $G_f = (V, E_f)$, where

$$E_f = \{(u, v) \in V \times V : c_f(u, v) > 0\}$$

- That is, as promised above, each edge of the residual network, or residual edge, can admit a flow that is greater than 0.

The Ford-Fulkerson method

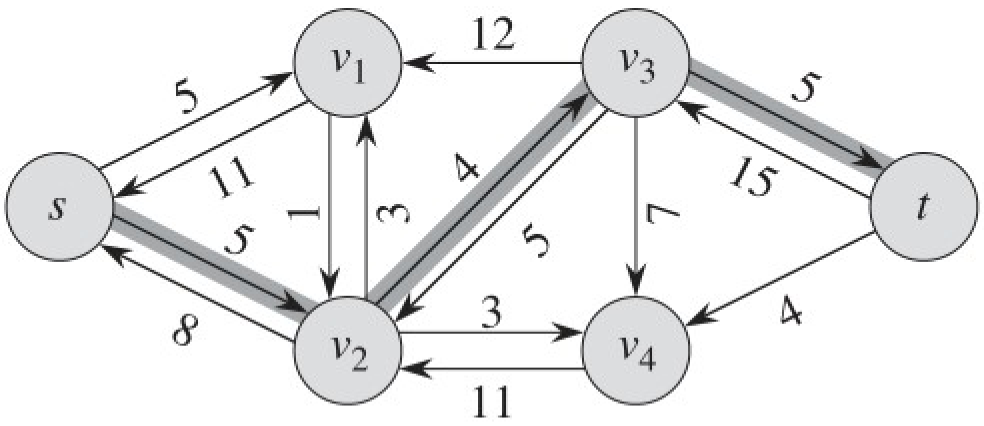
Residual Networks



(a)

The Ford-Fulkerson method

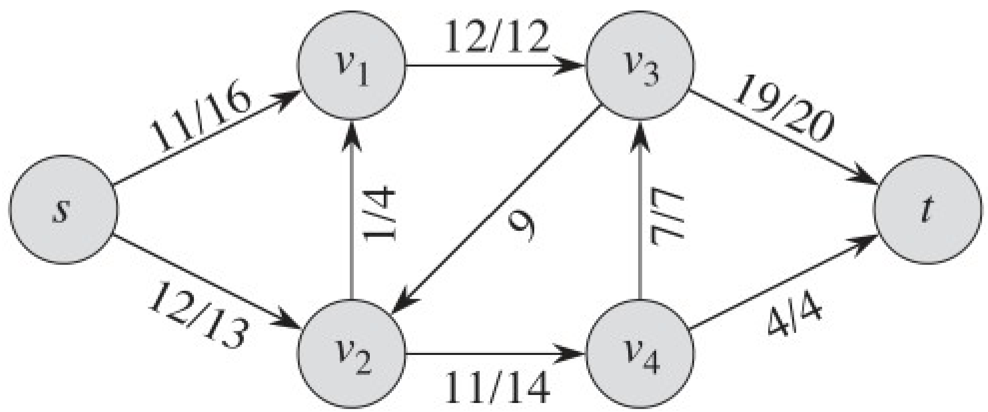
Residual Networks



(b)

The Ford-Fulkerson method

Residual Networks



(c)

The Ford-Fulkerson method

Residual Networks

- A flow in a residual network provides a roadmap for adding flow to the original flow network.
- If f is a flow in G and f' is a flow in the corresponding residual network G_f, we define $f \uparrow f'$, the augmentation of flow f by f', to be a function from V x V to R, defined by

$$(f \uparrow f')(u, v) = \begin{cases} f(u, v) + f'(u, v) - f'(v, u) & \text{if } (u, v) \in E , \\ 0 & \text{otherwise .} \end{cases}$$

The Ford-Fulkerson method

Augmenting Paths

- Given a flow network $G = (V, E)$ and a flow f , an augmenting path p is a simple path from s to t in the residual network G_f .
- By the definition of the residual network, we may increase the flow on an edge (u, v) of an augmenting path by up to $c_f(u, v)$ without violating the capacity constraint on whichever of (u, v) and (v, u) is in the original flow network G.

The Ford-Fulkerson method

Augmenting Paths

- We call the maximum amount by which we can increase the flow on each edge in an augmenting path p the residual capacity of p, given by

$$c_f(p) = \min\{c_f(u, v) : (u, v) \text{ is on } p\}$$

The Ford-Fulkerson method

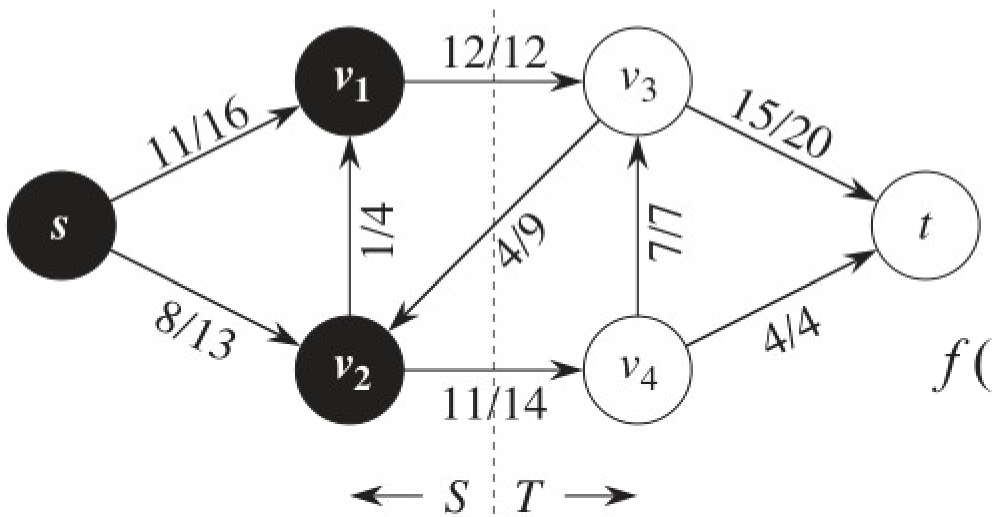
Cuts of flow networks

- The Ford-Fulkerson method repeatedly augments the flow along augmenting paths until it has found a maximum flow.
- The max-flow min-cut theorem, tells us that a flow is maximum if and only if its residual network contains no augmenting path.

The Ford-Fulkerson method

Cuts of flow networks

- A cut (S, T) of flow network G = (V, E) is a partition of V into S and T = V - S such that s ∈ S and t ∈ T.



- The net flow f (S, T) across the cut (S, T) is defined to be

$$f(S,T) = \sum_{u \in S} \sum_{v \in T} f(u,v) - \sum_{u \in S} \sum_{v \in T} f(v,u)$$

- The capacity of the cut (S, T) is

$$c(S,T) = \sum_{u \in S} \sum_{v \in T} c(u,v) .$$

- A cut (S, T) where S = {s, v_1, v_2} and T = {v_3, v_4, t}.
- The capacity is c(S, T) D= 26
- The net flow across (S, T) is f (S, T) = 19

The Ford-Fulkerson method

Cuts of flow networks

- A minimum cut of a network is a cut whose capacity is minimum over all cuts of the network.

The Ford-Fulkerson method

Max-flow min-cut theorem

- If f is a flow in a flow network G = (V, E) with source s and sink t, then the following conditions are equivalent:

  1. f is a maximum flow in G.

  2. The residual network G_f contains no augmenting paths.

  3. $|f| = c(S, T)$ for some cut (S, T) of G.

The Ford-Fulkerson method

The basic Ford-Fulkerson algorithm

FORD-FULKERSON(G, s, t)

for each edge $(u, v) \in G.E$
    $(u, v).f = 0$
while there exists a path p from s to t in the residual network G_f
    $c_f(p) = \min\{c_f(u, v) : (u, v) \text{ is in } p\}$
    for each edge (u, v) in p
        if $(u, v) \in E$
            $(u, v).f = (u, v).f + c_f(p)$
        else $(v, u).f = (v, u).f - c_f(p)$

The Ford-Fulkerson method

The basic Ford-Fulkerson algorithm

FORD-FULKERSON(G, s, t)

1 **for** each edge $(u, v) \in G.E$
2 $\quad (u, v).f = 0$
3 **while** there exists a path p from s to t in the residual network G_f
4 $\quad c_f(p) = \min\{c_f(u, v) : (u, v) \text{ is in } p\}$
5 $\quad$ **for** each edge (u, v) in p
6 $\quad\quad$ **if** $(u, v) \in E$
7 $\quad\quad\quad (u, v).f = (u, v).f + c_f(p)$
8 $\quad\quad$ **else** $(v, u).f = (v, u).f - c_f(p)$

- The running time of FORD-FULKERSON depends on how we find the augmenting path p in line 3.
- If we choose it poorly, the algorithm might not even terminate: the value of the flow will increase with successive augmentations, but it need not even converge to the maximum flow value.

The Ford-Fulkerson method

The basic Ford-Fulkerson algorithm

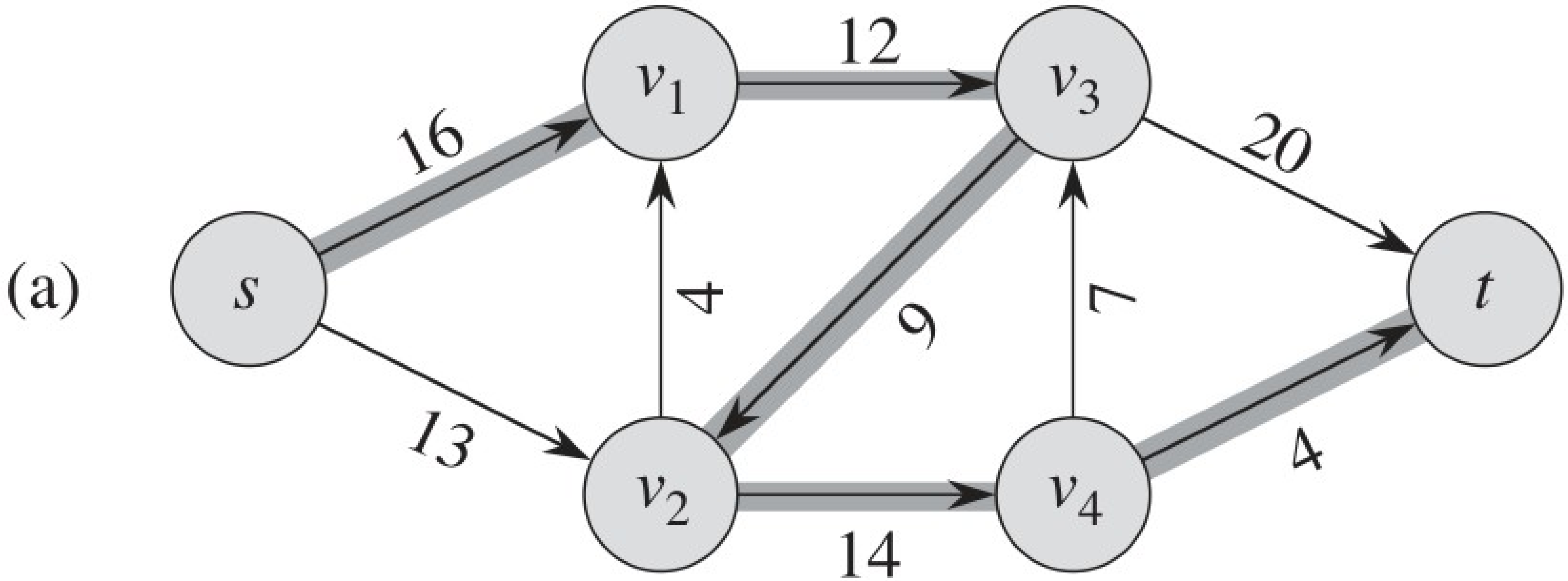
FORD-FULKERSON(G, s, t)

for each edge $(u, v) \in G.E$
    $(u, v).f = 0$
while there exists a path p from s to t in the residual network G_f
    $c_f(p) = \min\{c_f(u, v) : (u, v) \text{ is in } p\}$
    for each edge (u, v) in p
        if $(u, v) \in E$
            $(u, v).f = (u, v).f + c_f(p)$
        else $(v, u).f = (v, u).f - c_f(p)$

- If we find the augmenting path by using a breadth-first search , however, the algorithm runs in polynomial time.

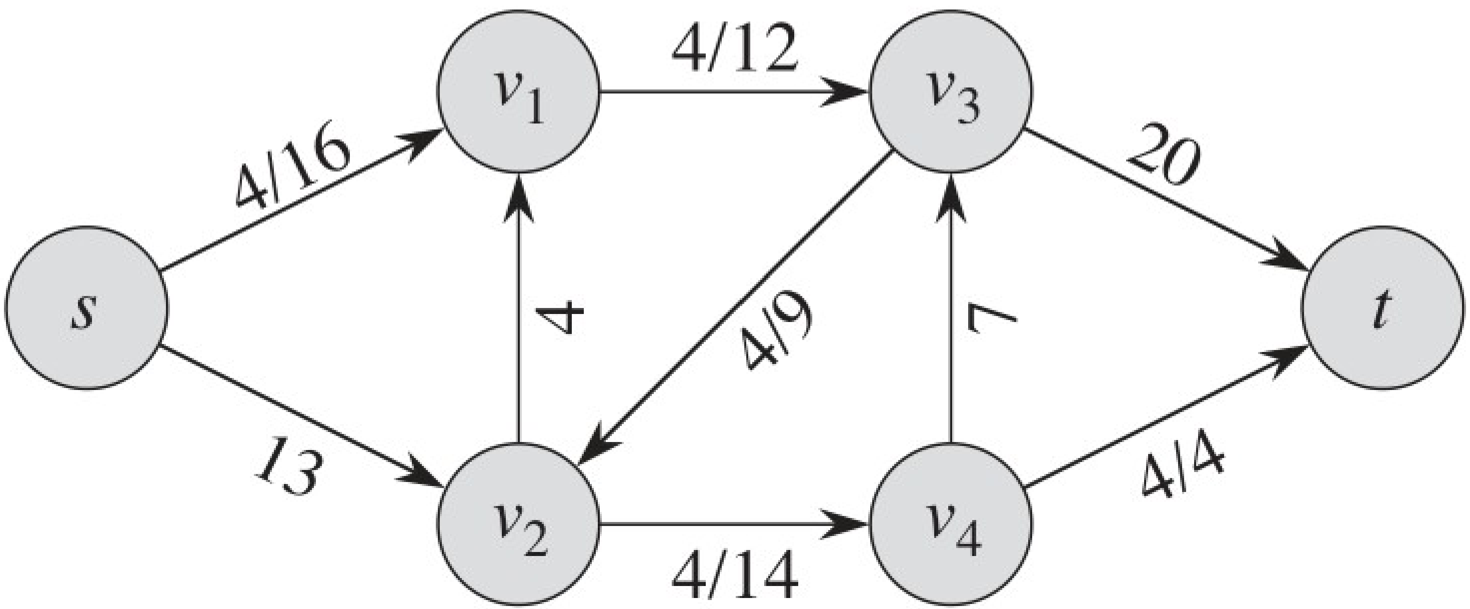
The Ford-Fulkerson method

The basic Ford-Fulkerson algorithm



The Ford-Fulkerson method

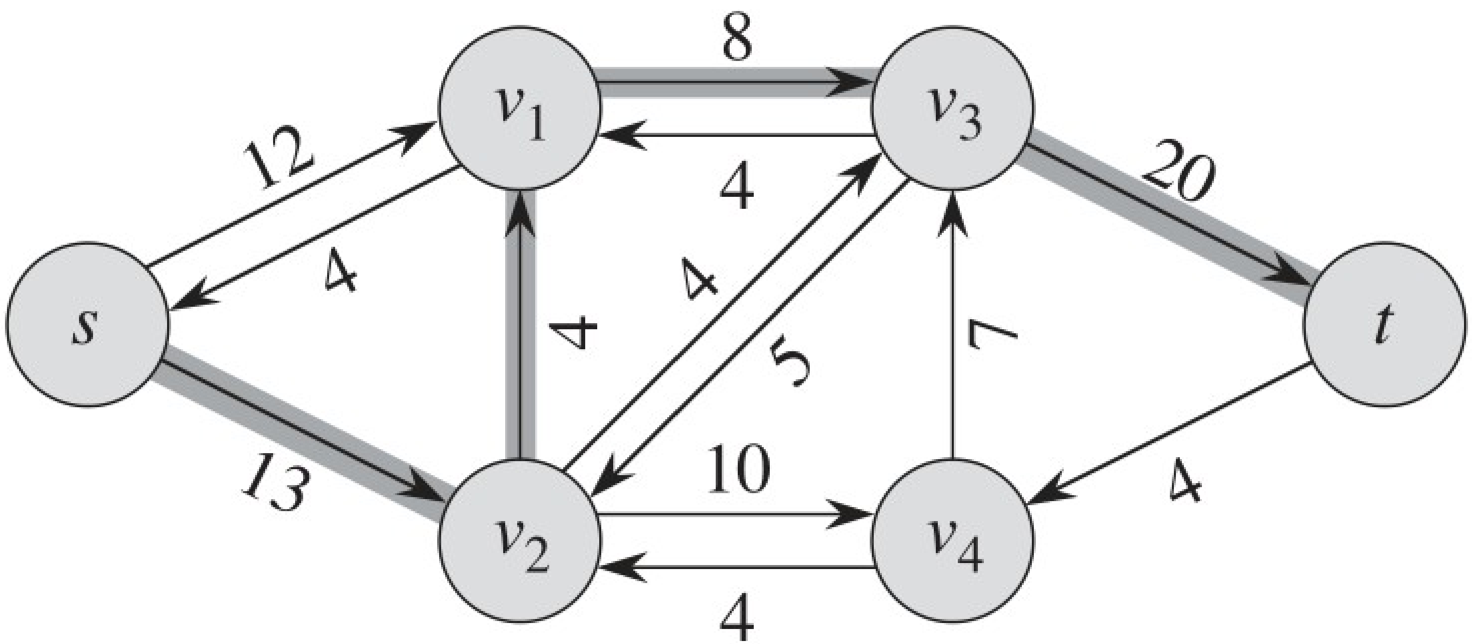
The basic Ford-Fulkerson algorithm



The Ford-Fulkerson method

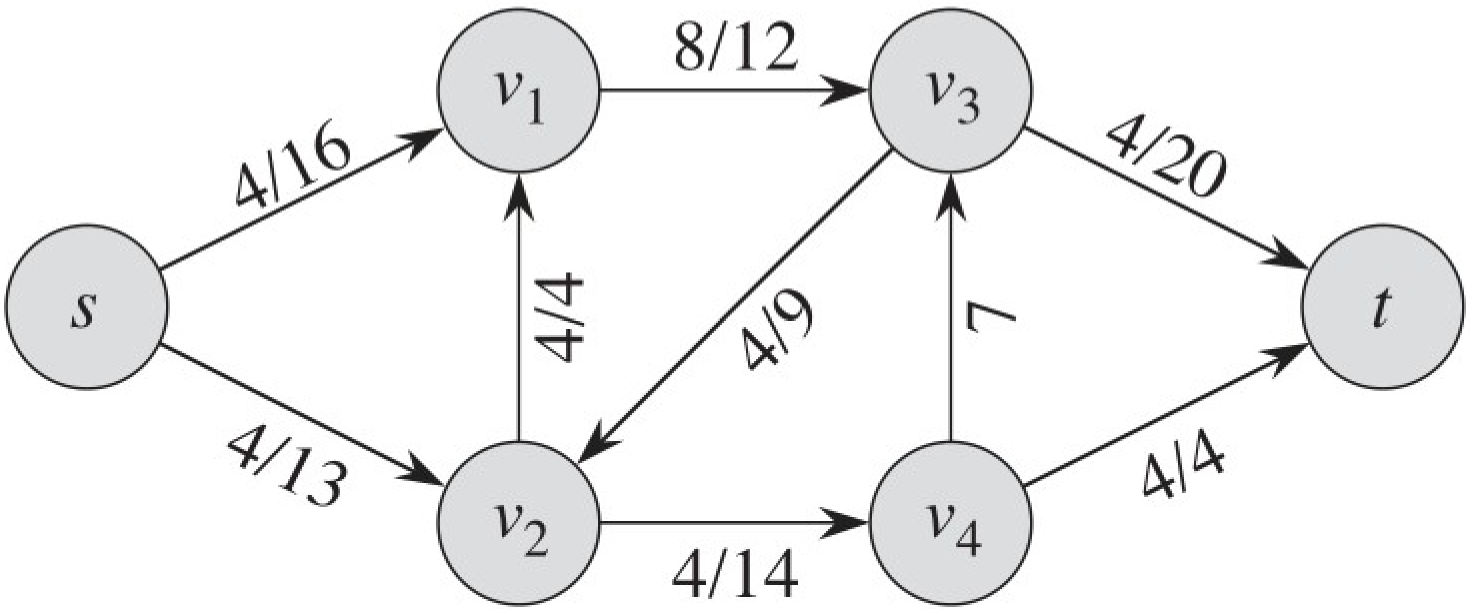
The basic Ford-Fulkerson algorithm

(b)



The Ford-Fulkerson method

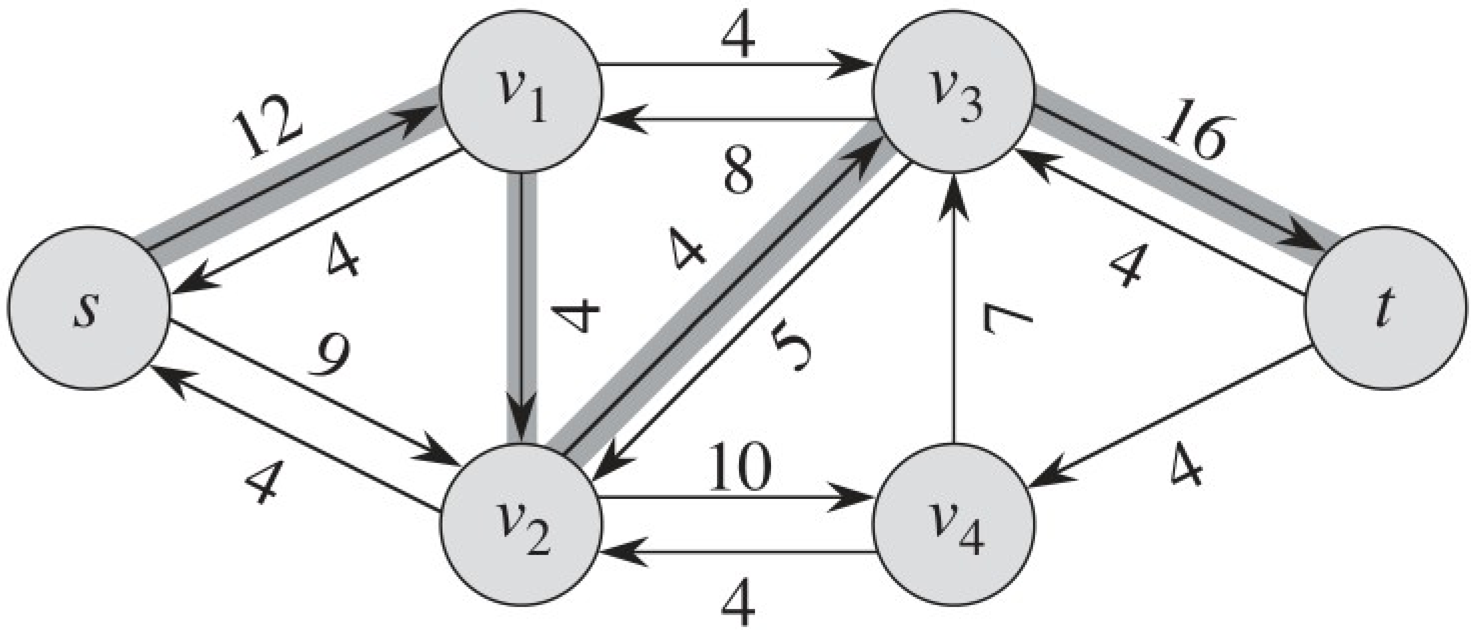
The basic Ford-Fulkerson algorithm



The Ford-Fulkerson method

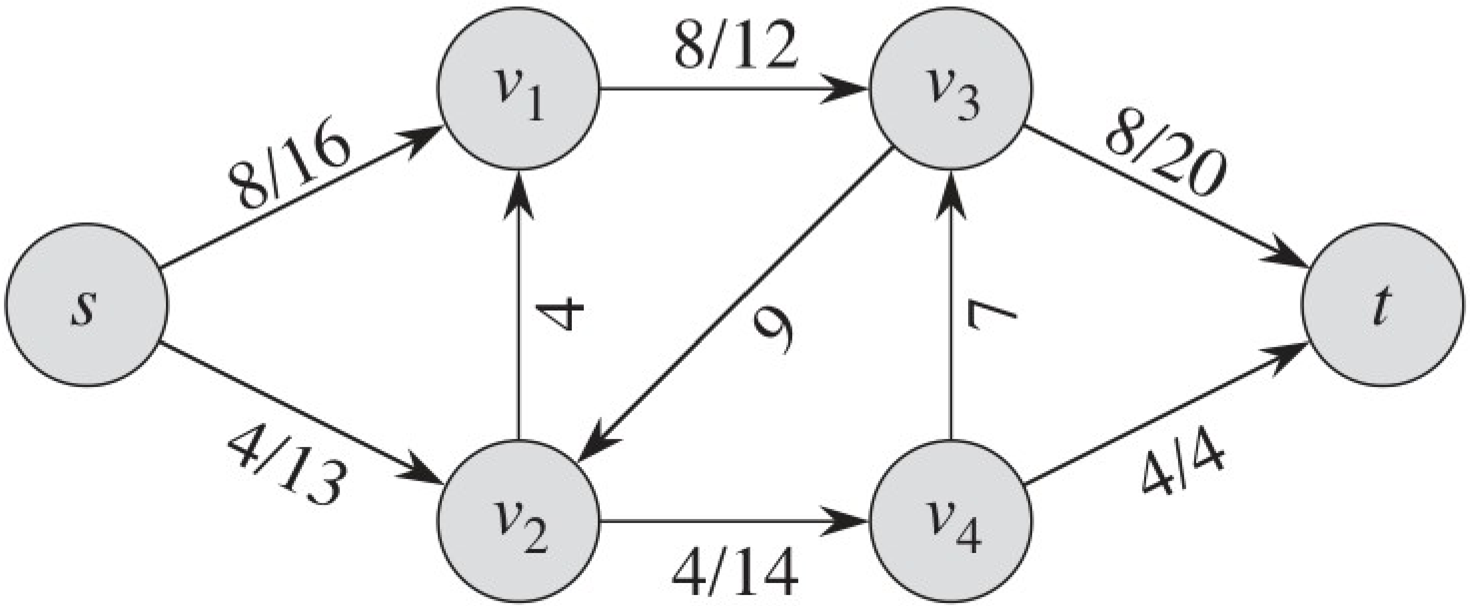
The basic Ford-Fulkerson algorithm

(c)



The Ford-Fulkerson method

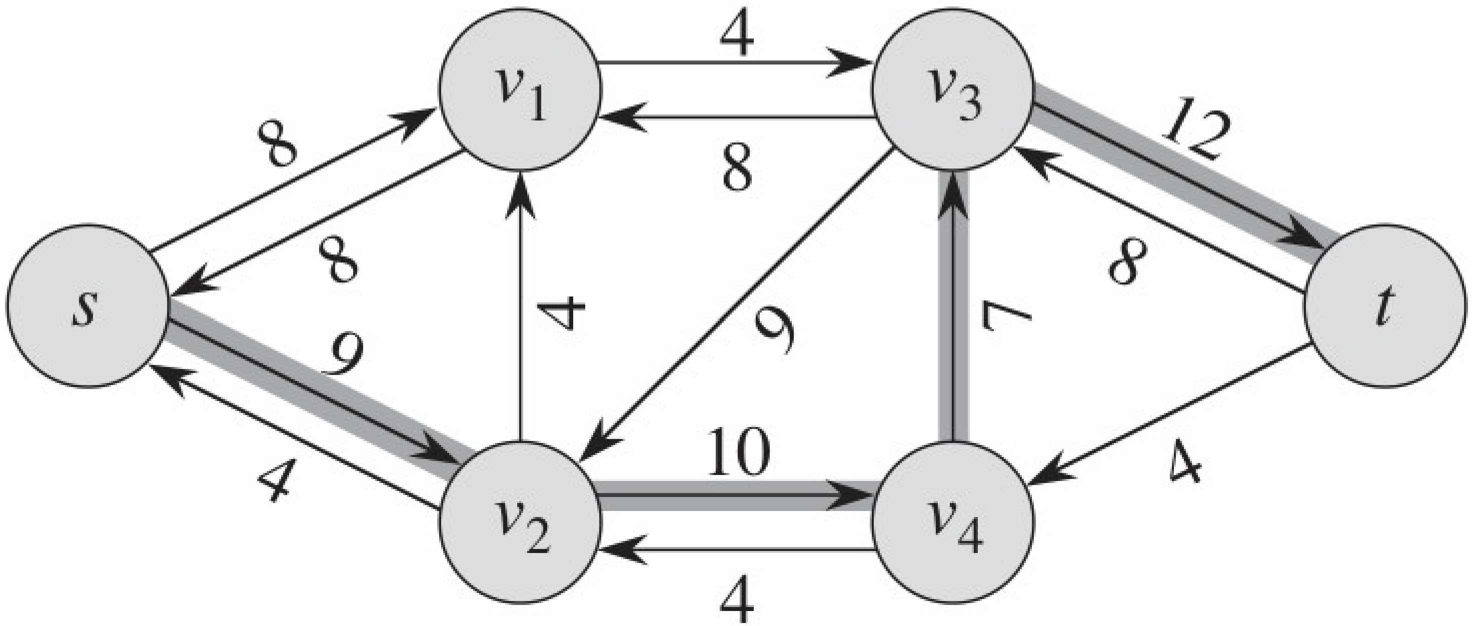
The basic Ford-Fulkerson algorithm



The Ford-Fulkerson method

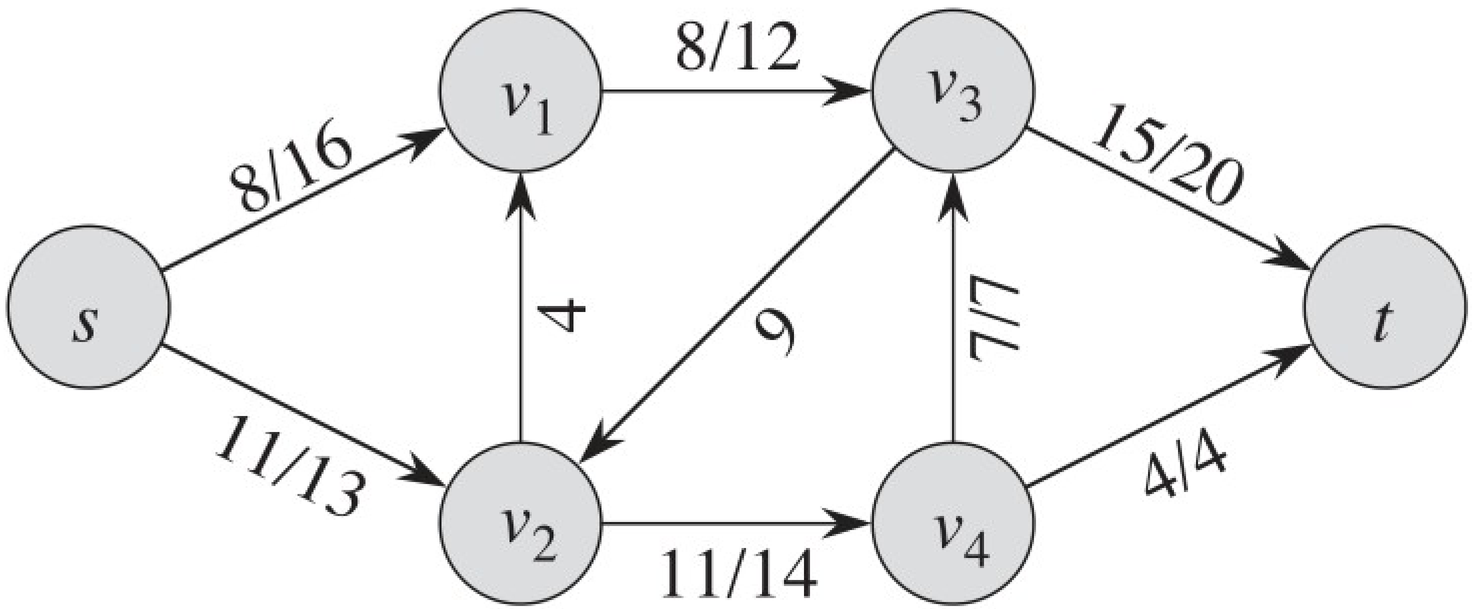
The basic Ford-Fulkerson algorithm

(d)



The Ford-Fulkerson method

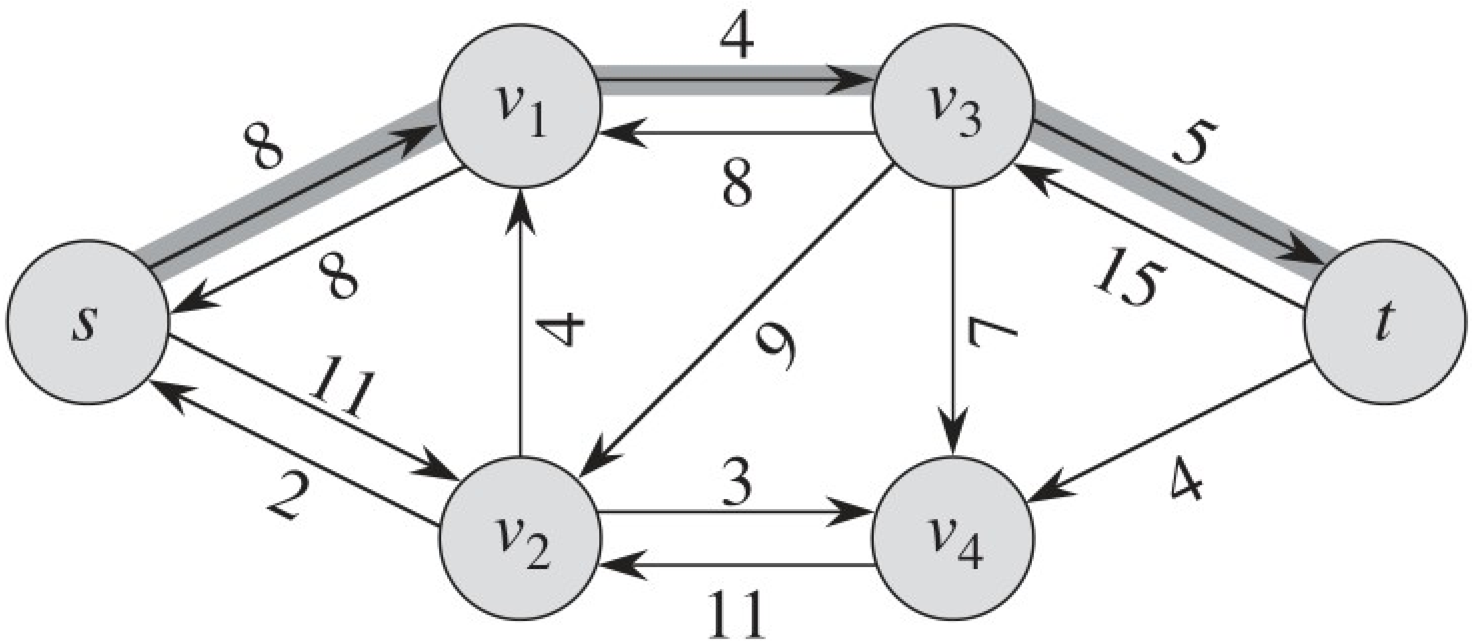
The basic Ford-Fulkerson algorithm



The Ford-Fulkerson method

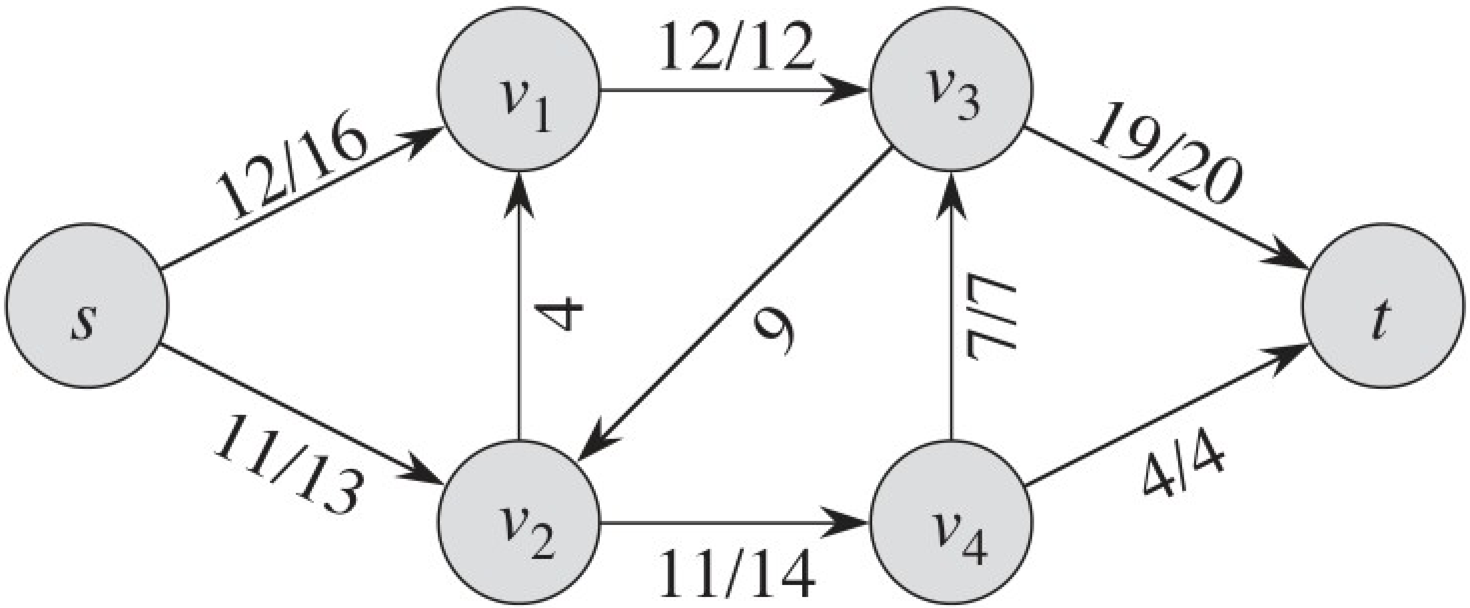
The basic Ford-Fulkerson algorithm

(e)



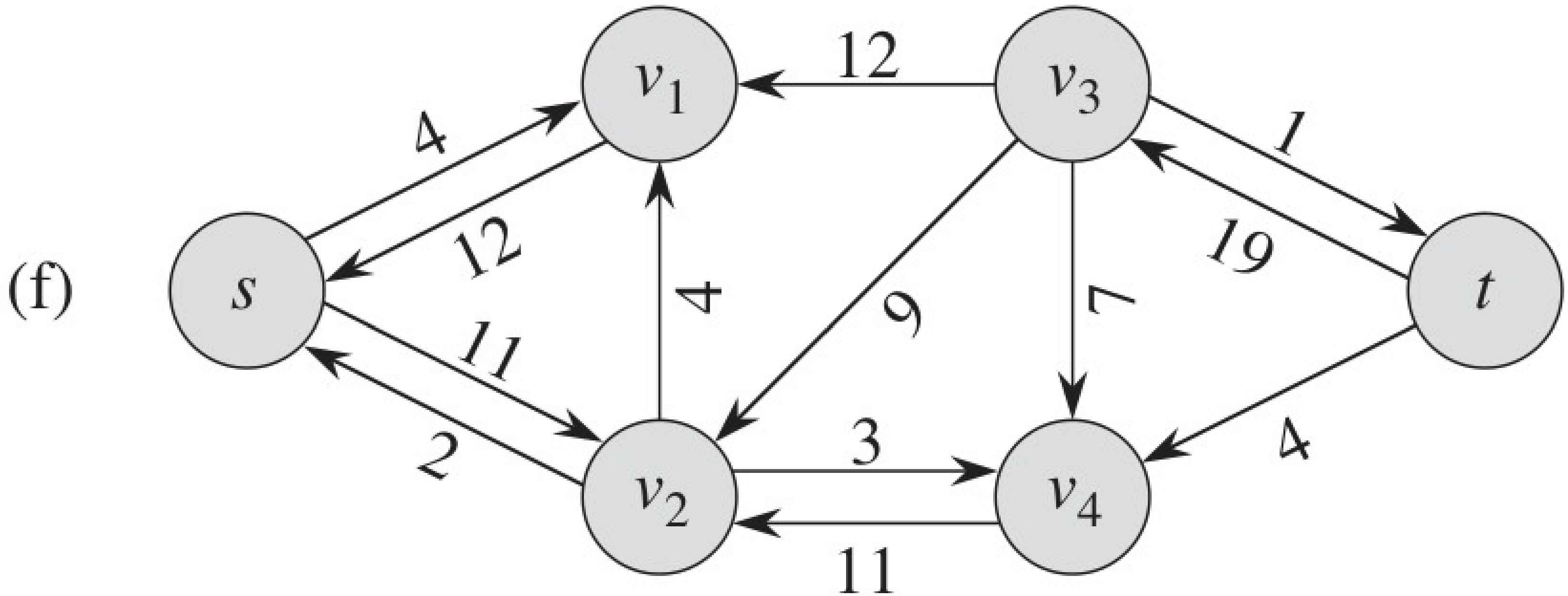
The Ford-Fulkerson method

The basic Ford-Fulkerson algorithm



The Ford-Fulkerson method

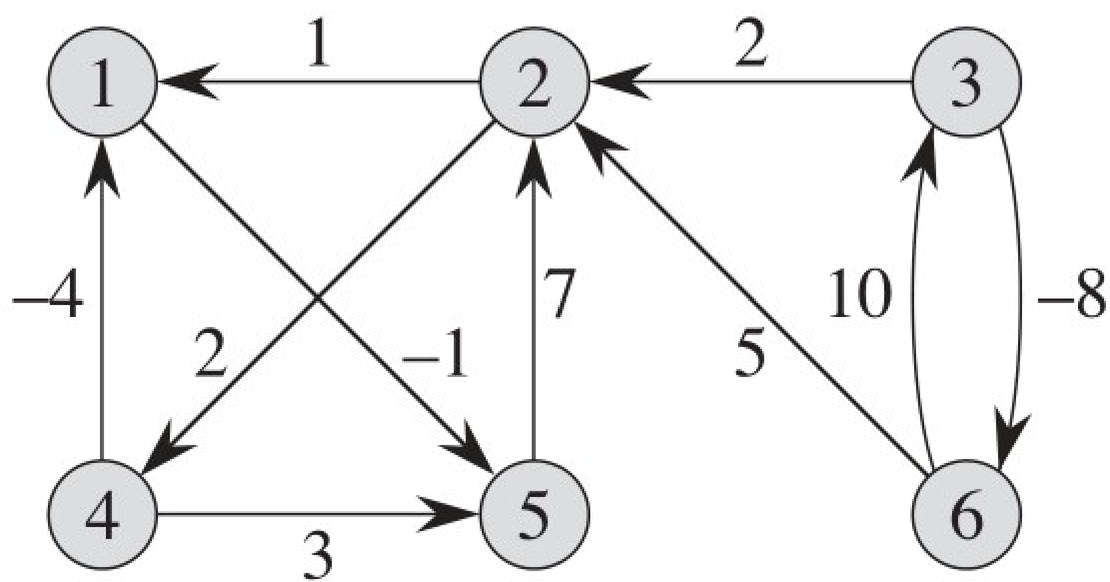
The basic Ford-Fulkerson algorithm



The value of the maximum flow found is 23.

Review Questions

1) Run the Bellman-Ford algorithm on the directed graph in the given figure using vertex 1 as the source. Show the d and π values after each pass.



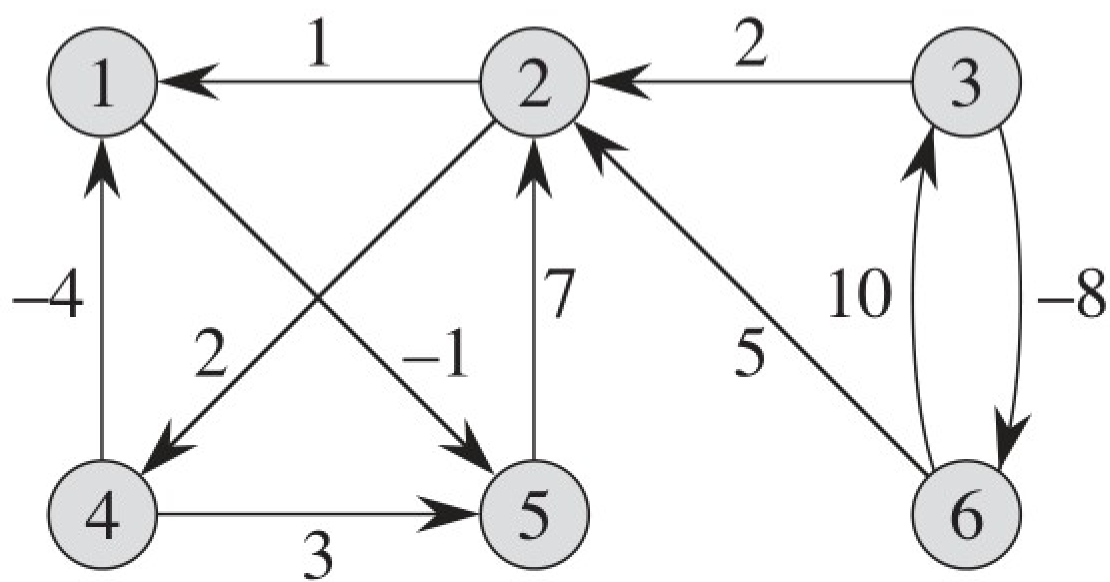
Review Questions

2) Give a simple example of a directed graph with negative-weight edges for which Dijkstra's algorithm produces incorrect answers.

3) Professor Ram has written a program that he claims implements Dijkstra's algorithm. The program produces v.d and v.π for each vertex $v \in V$. Give an $O(V + E)$ time algorithm to check the output of the professor's program. It should determine whether the d and π attributes match those of some shortest-paths tree. You may assume that all edge weights are non-negative.

Review Questions

4. Run the Floyd-Warshall algorithm on the weighted, directed graph in the given figure. Show the matrix $D^{(k)}$ that results for each iteration of the outer loop.



Review Questions

5. Run the Ford-Fulkerson algorithm to compute the maximum flow in the following network. Also, comment on the run time of this algorithm.