CSE 2202

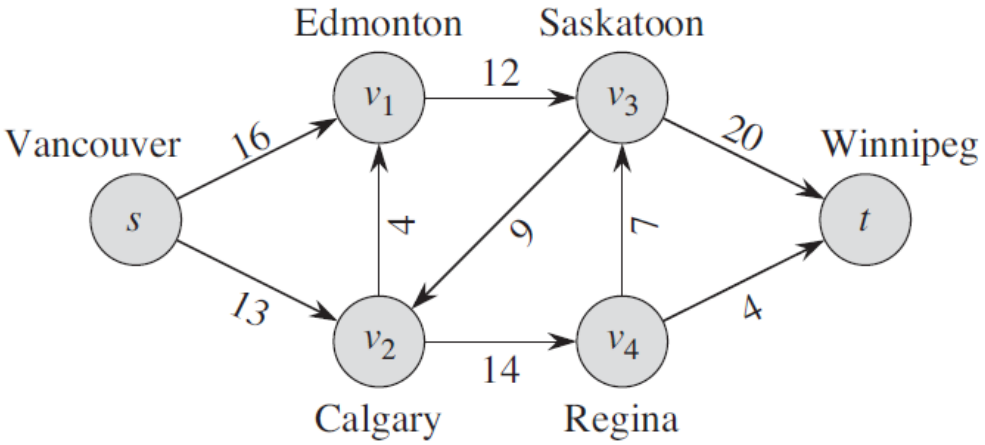
Design and Analysis of Algorithms – I

Network Flow

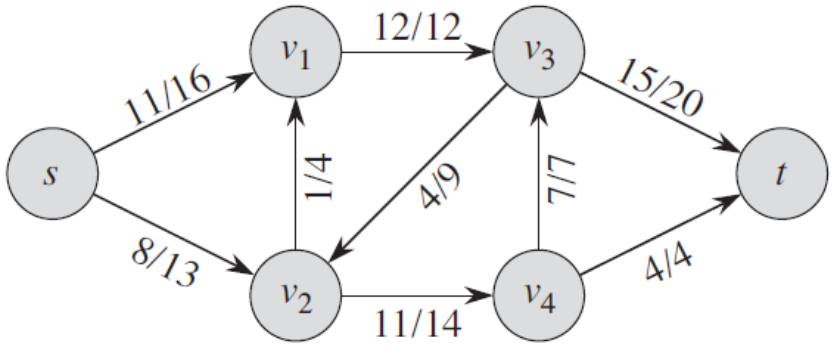
Network Flow

- Instance:
  - A Network is a directed graph G
  - Edges represent pipes/conduits/channels that carry flow
  - Each edge $\langle u,v \rangle$ has a maximum capacity $c_{\langle u,v \rangle}$
  - A source node s out which flow leaves
  - A sink node t in which flow arrives

Network Flow



(a)



(b)

A ***flow*** in G is a real-valued function $f : V \times V \to \mathbb{R}$ that satisfies the following two properties:

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

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

"flow in equals flow out."

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

Network Flow

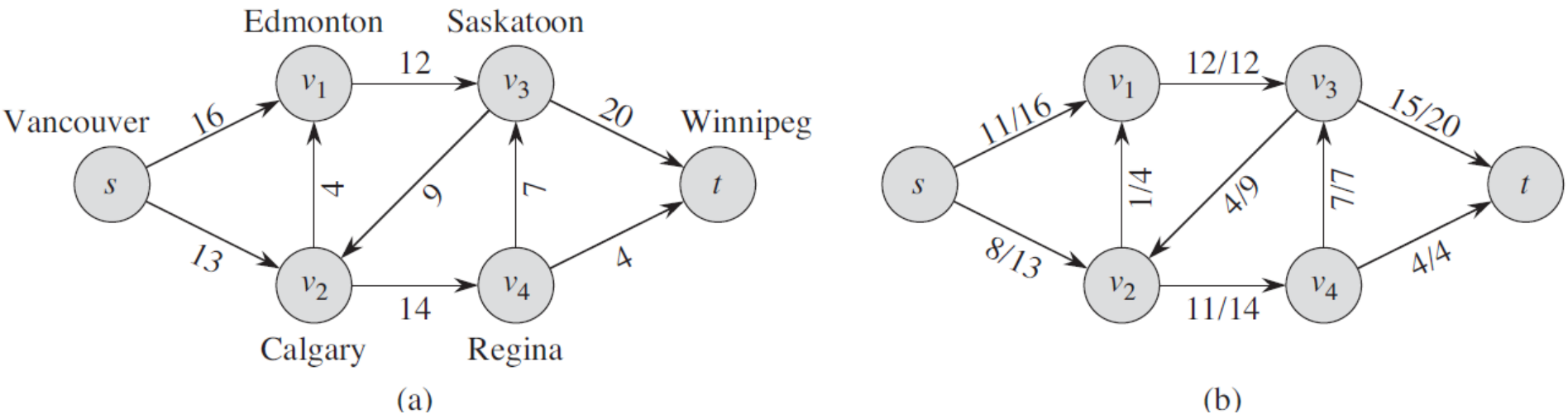
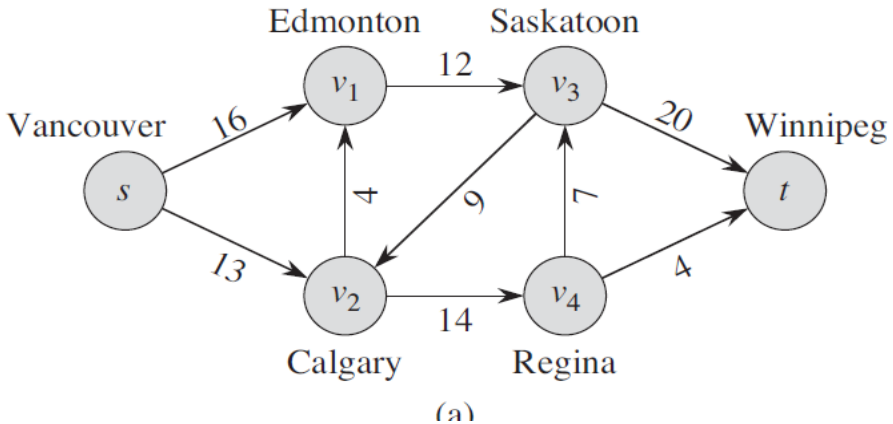


Figure 26.1 **(a)** A flow network $G = (V, E)$ for the Lucky Puck Company's trucking problem. The Vancouver factory is the source s, and the Winnipeg warehouse is the sink t. The company ships pucks through intermediate cities, but only $c(u, v)$ crates per day can go from city u to city v. Each edge is labeled with its capacity. **(b)** A flow f in G with value $|f| = 19$. Each edge (u, v) is labeled by $f(u, v)/c(u, v)$. The slash notation merely separates the flow and capacity; it does not indicate division.

Network Flow



We call the nonnegative quantity $f(u, v)$ the flow from vertex u to vertex v. The ***value*** $|f|$ of a flow f is defined as

$$|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s) , \tag{26.1}$$

that is, the total flow out of the source minus the flow into the source. (Here, the $|\cdot|$ notation denotes flow value, not absolute value or cardinality.) Typically, a flow network will not have any edges into the source, and the flow into the source, given by the summation $\sum_{v \in V} f(v, s)$, will be 0. We include it, however, because when we introduce residual networks later in this chapter, the flow into the source will become significant. 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.

Modeling Problems with antiparallel edges

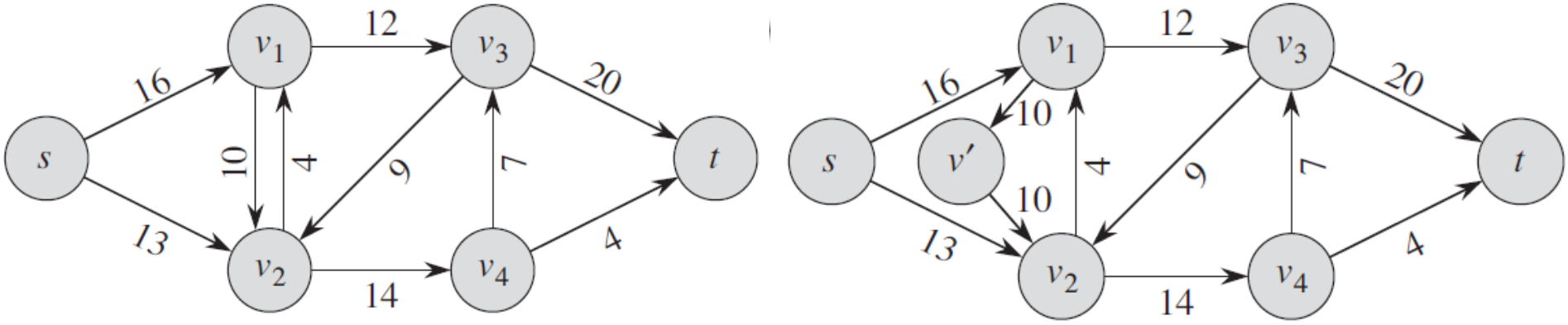
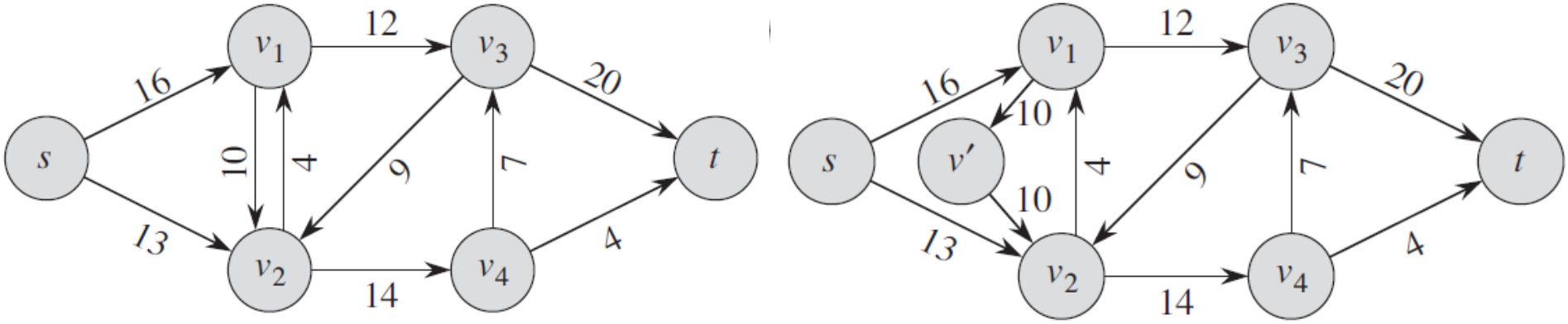


Figure 26.2 Converting a network with antiparallel edges to an equivalent one with no antiparallel edges. **(a)** A flow network containing both the edges (v_1, v_2) and (v_2, v_1). **(b)** An equivalent network with no antiparallel edges. We add the new vertex v', and we replace edge (v_1, v_2) by the pair of edges (v_1, v') and (v', v_2), both with the same capacity as (v_1, v_2).

Suppose that the trucking firm offered Lucky Puck the opportunity to lease space for 10 crates in trucks going from Edmonton to Calgary. It would seem natural to add this opportunity to our example and form the network shown in Figure 26.2(a). This network suffers from one problem, however: it violates our original assumption that if an edge $(v_1, v_2) \in E$, then $(v_2, v_1) \notin E$. We call the two edges (v_1, v_2) and (v_2, v_1) ***antiparallel***. Thus, if we wish to model a flow problem with antiparallel edges, we must transform the network into an equivalent one containing no antiparallel edges

Modeling Problems with antiparallel edges



- We also set the **capacity of both new edges** to the capacity of the original edge. The resulting network satisfies the property that if an edge is in the network, the reverse edge is not.

- We see that a real-world flow problem might be **most naturally modelled** by a network with antiparallel edges.
- It will be **convenient** to disallow antiparallel edges, however, and so we have a straightforward way to convert a network containing antiparallel edges into an equivalent one with no antiparallel edges.

Networks with multiple sources and sinks

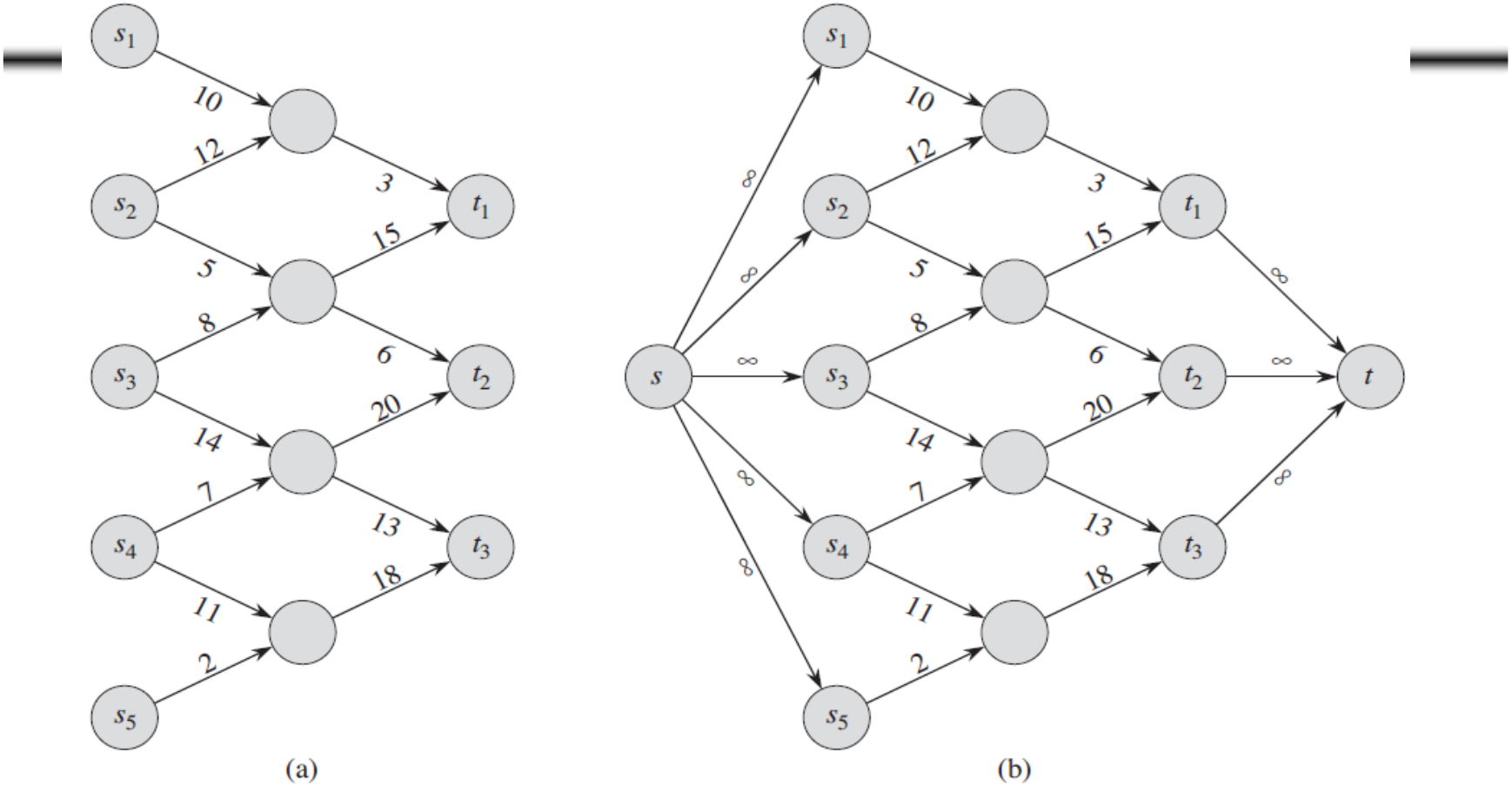


Figure 26.3 Converting a multiple-source, multiple-sink maximum-flow problem into a problem with a single source and a single sink. **(a)** A flow network with five sources $S = \{s_1, s_2, s_3, s_4, s_5\}$ and three sinks $T = \{t_1, t_2, t_3\}$. **(b)** An equivalent single-source, single-sink flow network. We add a supersource s and an edge with infinite capacity from s to each of the multiple sources. We also add a supersink t and an edge with infinite capacity from each of the multiple sinks to t.

The Ford-Fulkerson Method

We call **the Ford-Fulkerson Method** a “method” rather than an “algorithm” because it encompasses several implementations with differing running times.

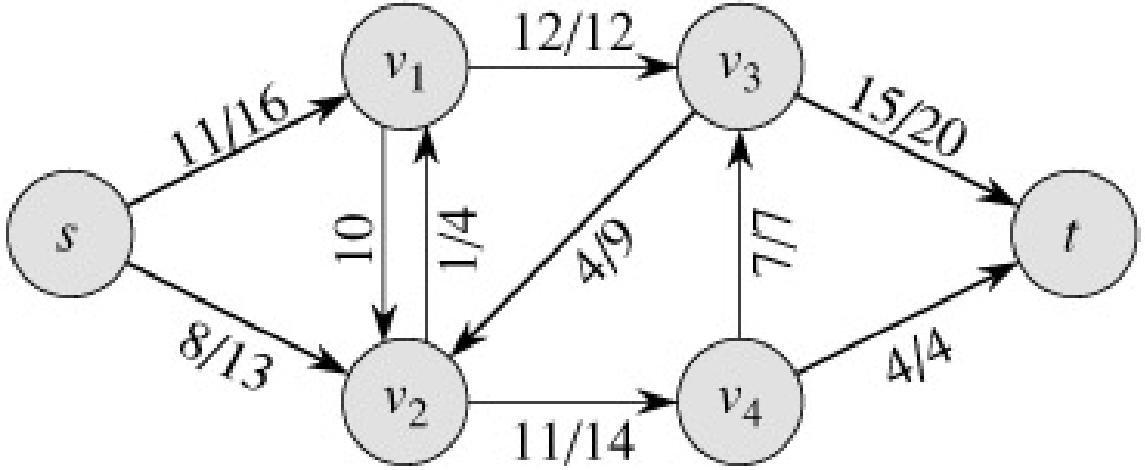
This depends on three important ideas that transcend the method and are relevant to many flow algorithms and problems:

- **Residual networks**
- **Augmenting paths**
- **Cuts**

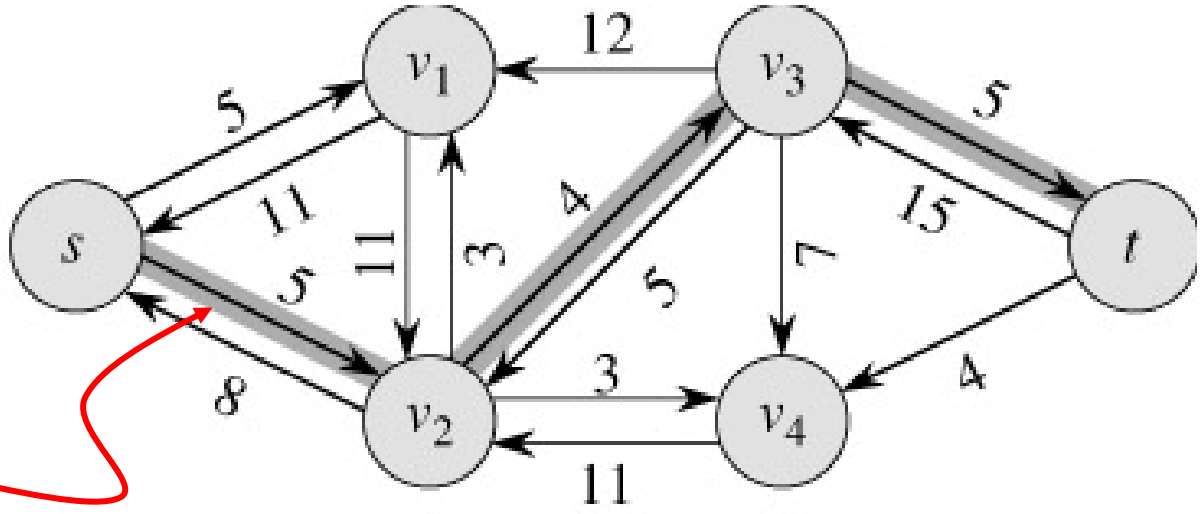
- Over the course of the algorithm, flow is monotonically increased.
- Augmenting paths are simply any path from the source to the sink that can currently take more flow.
- So, there are times when a path from the source to the sink can take on more flow, and that is an augmenting path.

Example of residual capacities

Network:



Residual Network:



Augmenting path

The Residual Networks

Residual Network: Given a flow network and a flow assignment, the residual network (often denoted as G_f) is constructed as follows:

- For each edge (u, v) in the original network with flow $f(u, v)$ and capacity $c(u, v)$, there will be:
  - A forward edge in G_f from u to v with capacity $c(u, v) - f(u, v)$. This represents how much more flow we can send from u to v.
  - A backward edge in G_f from v to u with capacity $f(u, v)$. This represents how much flow we can "undo" or send back.

Imagine you have a flow network with an edge (u, v) that has a capacity of, say, 10 units. If, in our current flow assignment, we are sending 7 units of flow from node u to node v along this edge, then in the residual network, we can introduce a "backward" edge from v to u to represent the possibility of "undoing" or "pushing back" some or all of that flow.

The Residual Networks

Why is this Useful?: Let's say, in the process of finding the maximum flow, we determine that it would be beneficial to reroute some flow that was previously sent from u to v. The backward edge allows us to do just that. By "pushing" flow back along the backward edge, we effectively reduce the flow on the original edge (u, v).

Augmenting Path Through Backward Edge: If an augmenting path in the residual network uses a backward edge, it means we're effectively reducing the flow on the corresponding forward edge in the original flow network.

The Residual Networks

- Let's assume there is a direct path from the source s to u, then u to v, and finally v to the sink t in the flow network.
- In a given flow assignment, 7 units of flow are sent from u to v.
- In the residual network, there exists a backward edge from v to u with a capacity of 7.
- If we find an augmenting path that goes from s to u, then uses the backward edge to go from v to u, and then goes from u back to t, this implies that we are redirecting 7 units of flow that originally went from s to t via v so that it now bypasses v entirely. This can increase the overall flow from s to t if there was previously some bottleneck preventing maximum flow through v.

The 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.

More 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 $u, v \in V$. We define the ***residual capacity*** $c_f(u, v)$ by

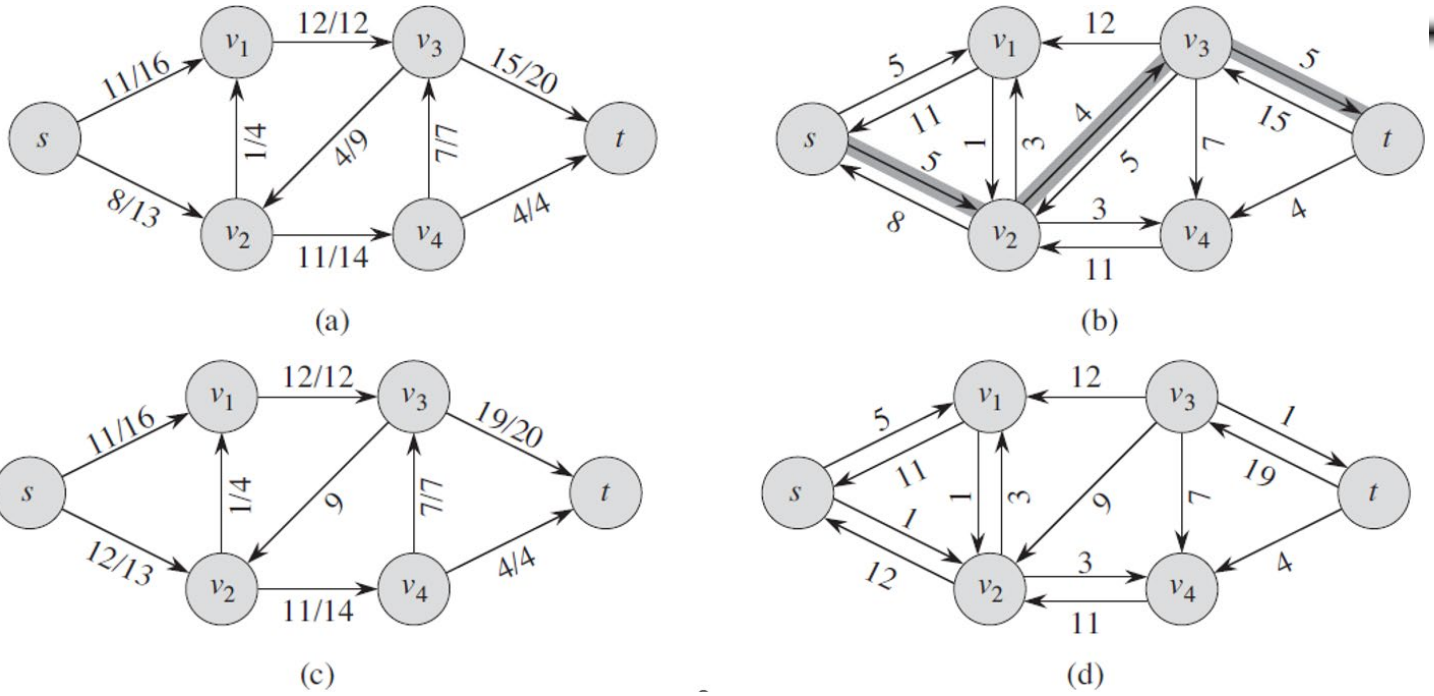
$$c_f(u, v) = \begin{cases} c(u, v) - f(u, v) & \text{if } (u, v) \in E\ , \\ f(v, u) & \text{if } (v, u) \in E\ , \\ 0 & \text{otherwise}\ . \end{cases} \tag{26.2}$$

Because of our assumption that $(u, v) \in E$ implies $(v, u) \notin E$, exactly one case in equation (26.2) applies to each ordered pair of vertices.

The only edges of G that are in G_f are those that can admit more flow; those edges (u, v) whose flow equals their capacity have $c_f(u, v) = 0$, and they are not in G_f.

The Residual Networks

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



The residual network G_f may also contain edges that are not in G

Pushing flow on the reverse edge in the residual network is also known as ***cancellation***

- For example, if we send 5 crates of hockey pucks from u to v and send 2 crates from v to u, we could equivalently (from the perspective of the final result) just send 3 creates from **u to v** and none from **v to u**.

- Cancellation of this type is crucial for any maximum-flow algorithm.

The Residual Networks

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

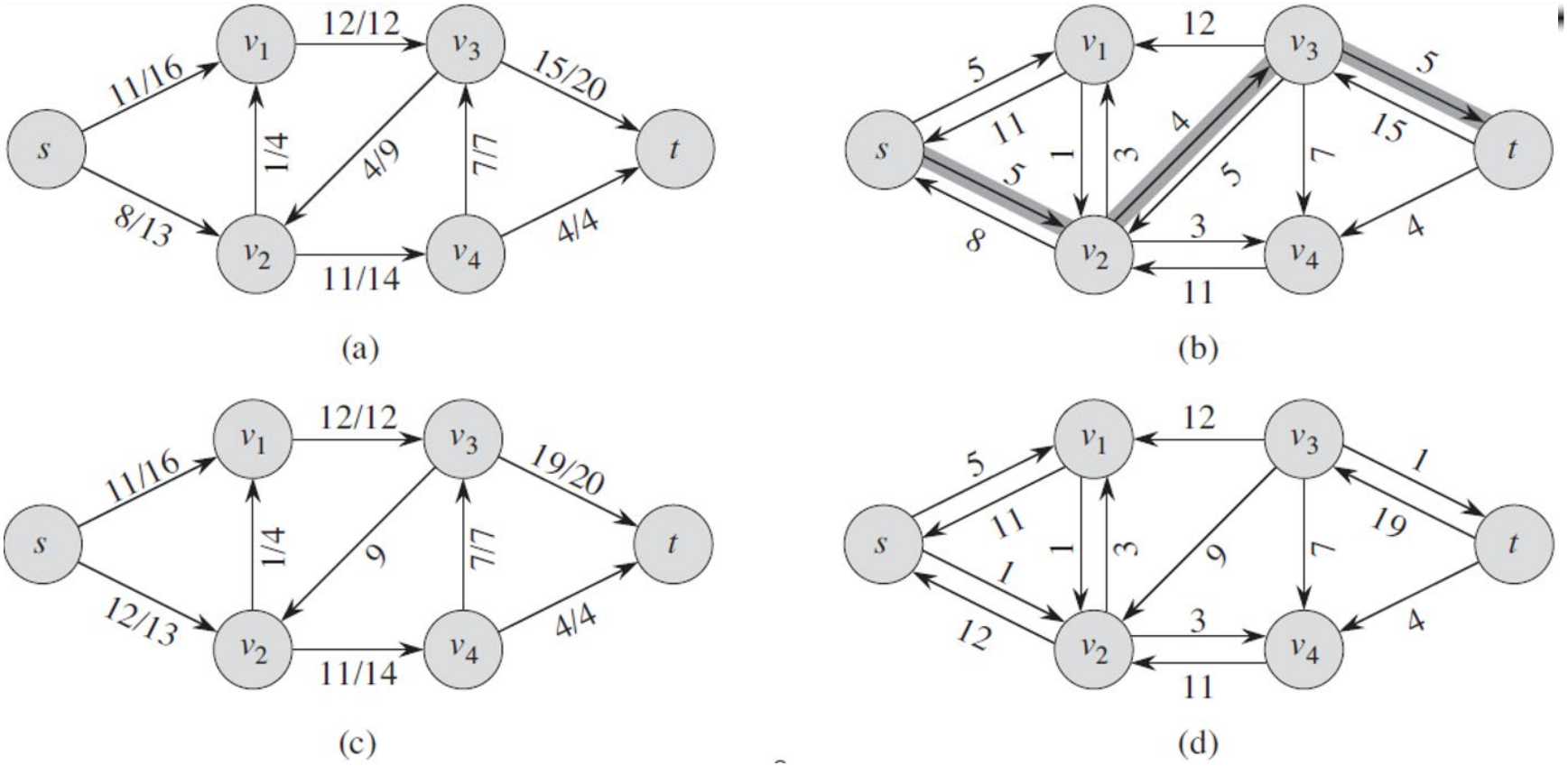


Figure 26.4 **(a)** The flow network G and flow f of Figure 26.1(b). **(b)** The residual network G_f with augmenting path p shaded; its residual capacity is $c_f(p) = c_f(v_2, v_3) = 4$. Edges with residual capacity equal to 0, such as (v_1, v_3), are not shown, a convention we follow in the remainder of this section. **(c)** The flow in G that results from augmenting along path p by its residual capacity 4. Edges carrying no flow, such as (v_3, v_2), are labeled only by their capacity, another convention we follow throughout. **(d)** The residual network induced by the flow in (c).

Why do we need residual networks?

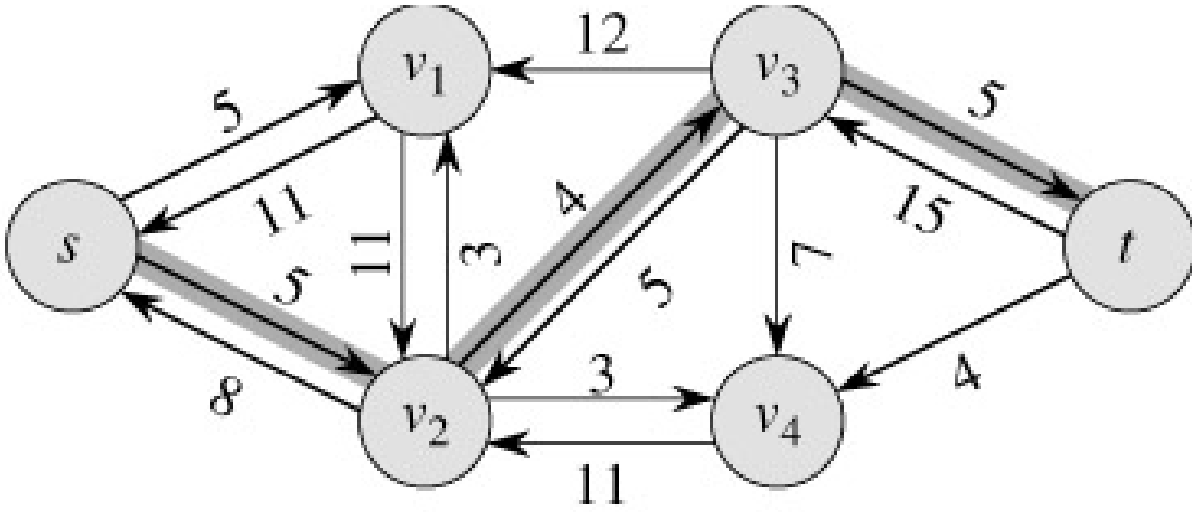
- Residual networks allow us to reverse flows if necessary.
- If we have taken a bad path then residual networks allow one to detect the condition and reverse the flow.
- A bad path is one which overlaps with too many other paths.

Augmenting Paths

- An augmenting path p is a simple path from s to t on the residual network.
- We can put more flow from s to t through p.
- We call the maximum capacity by which we can increase the flow on p the residual capacity of p.

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

Augmenting Paths - example



- The residual capacity of p = 4.
- Can improve the flow along p by 4.

The basic Ford-Fulkerson algorithm

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
```

FORD-FULKERSON(G, s, t)

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

The basic Ford-Fulkerson algorithm

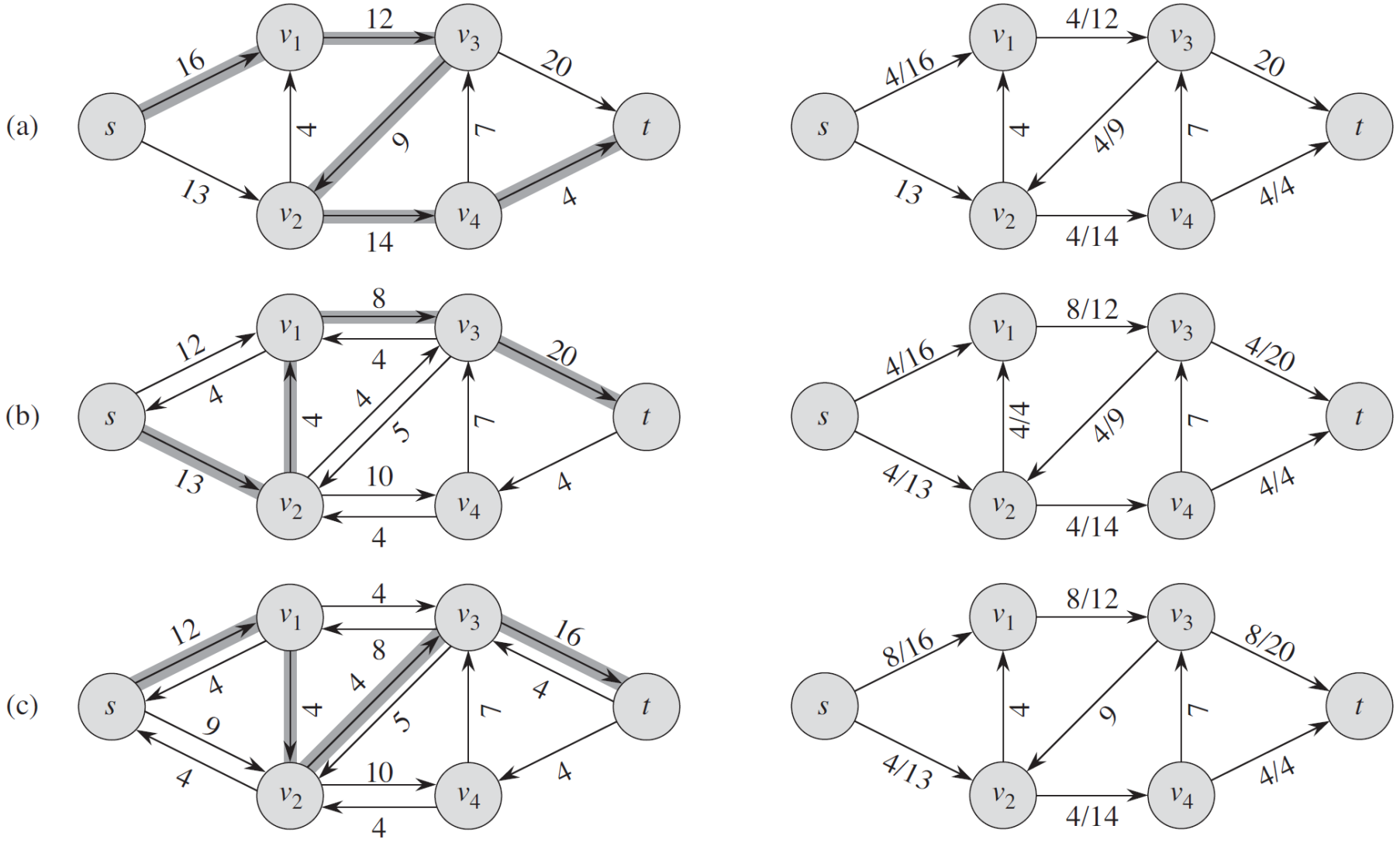
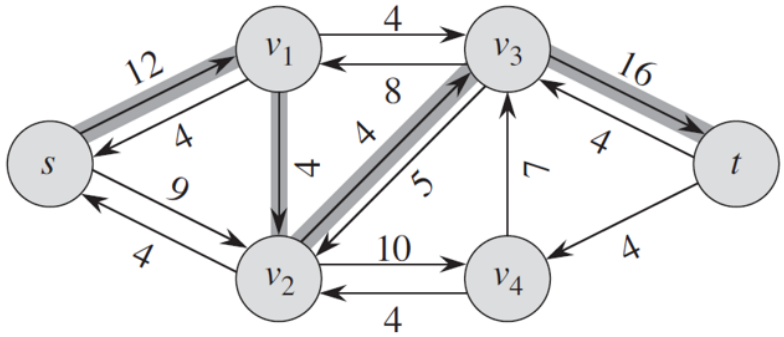
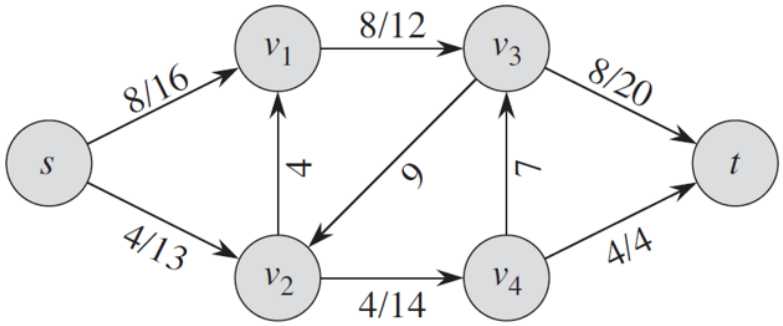


Figure 26.6 The execution of the basic Ford-Fulkerson algorithm. (a)–(e) Successive iterations of the **while** loop. The left side of each part shows the residual network G_f from line 3 with a shaded augmenting path p. The right side of each part shows the new flow f that results from augmenting f by f_p. The residual network in (a) is the input network G.

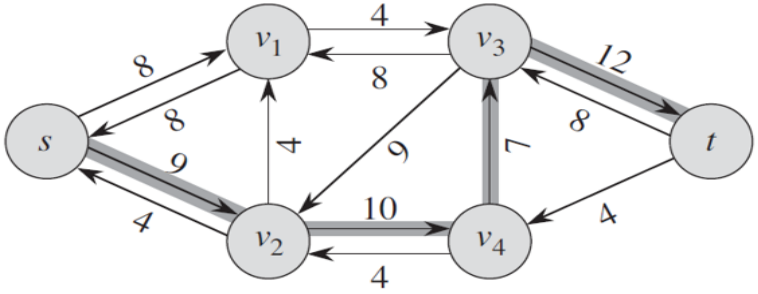
The basic Ford-Fulkerson algorithm

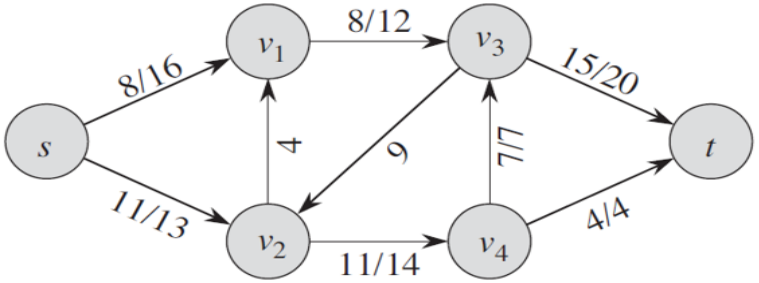
(c)



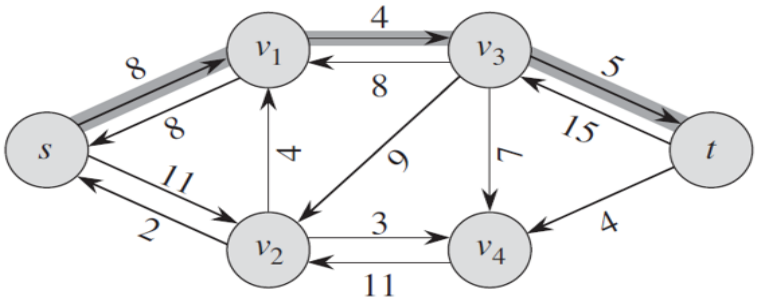


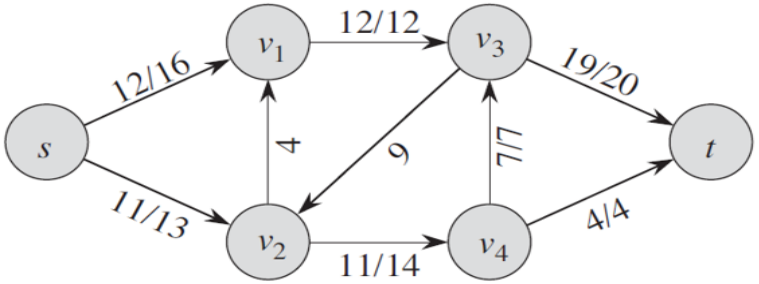
(d)



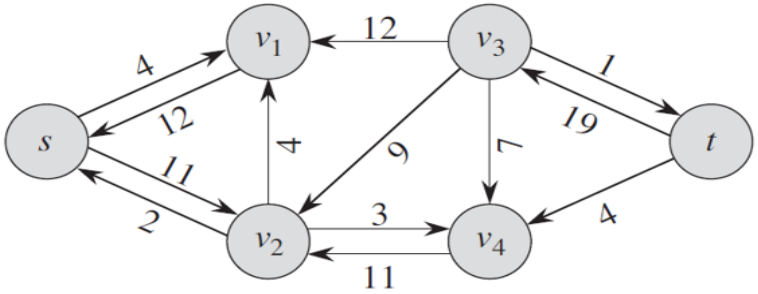


(e)





(f)



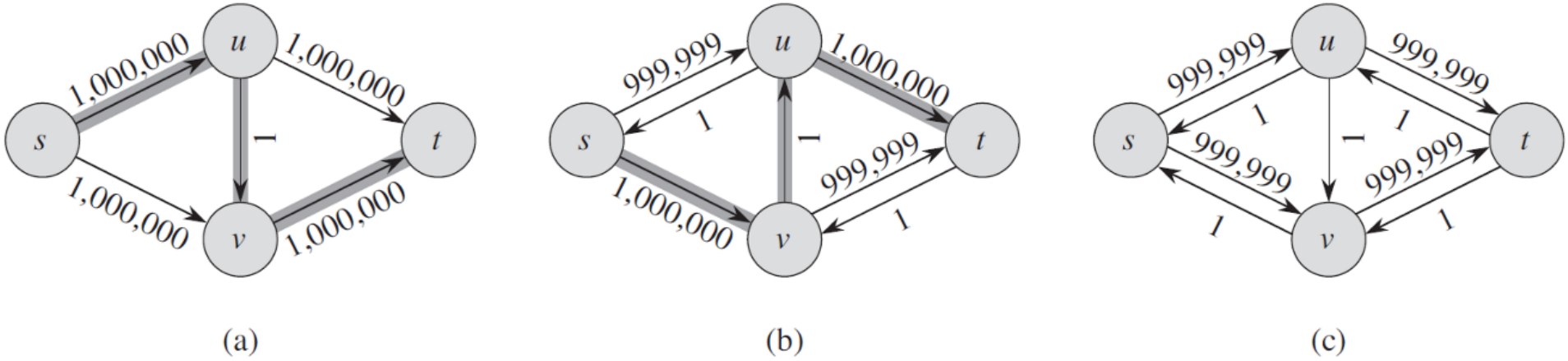
The basic Ford-Fulkerson algorithm

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

In practice, the maximum-flow problem often arises with integral capacities. If the capacities are rational numbers, we can apply an appropriate scaling transformation to make them all integral. If f^* denotes a maximum flow in the transformed network, then a straightforward implementation of FORD-FULKERSON executes the **while** loop of lines 3–8 at most $|f^*|$ times, since the flow value increases by at least one unit in each iteration.

FORD-FULKERSON algorithm $O(E\ |f^*|)$.

|f*|=2,000,000



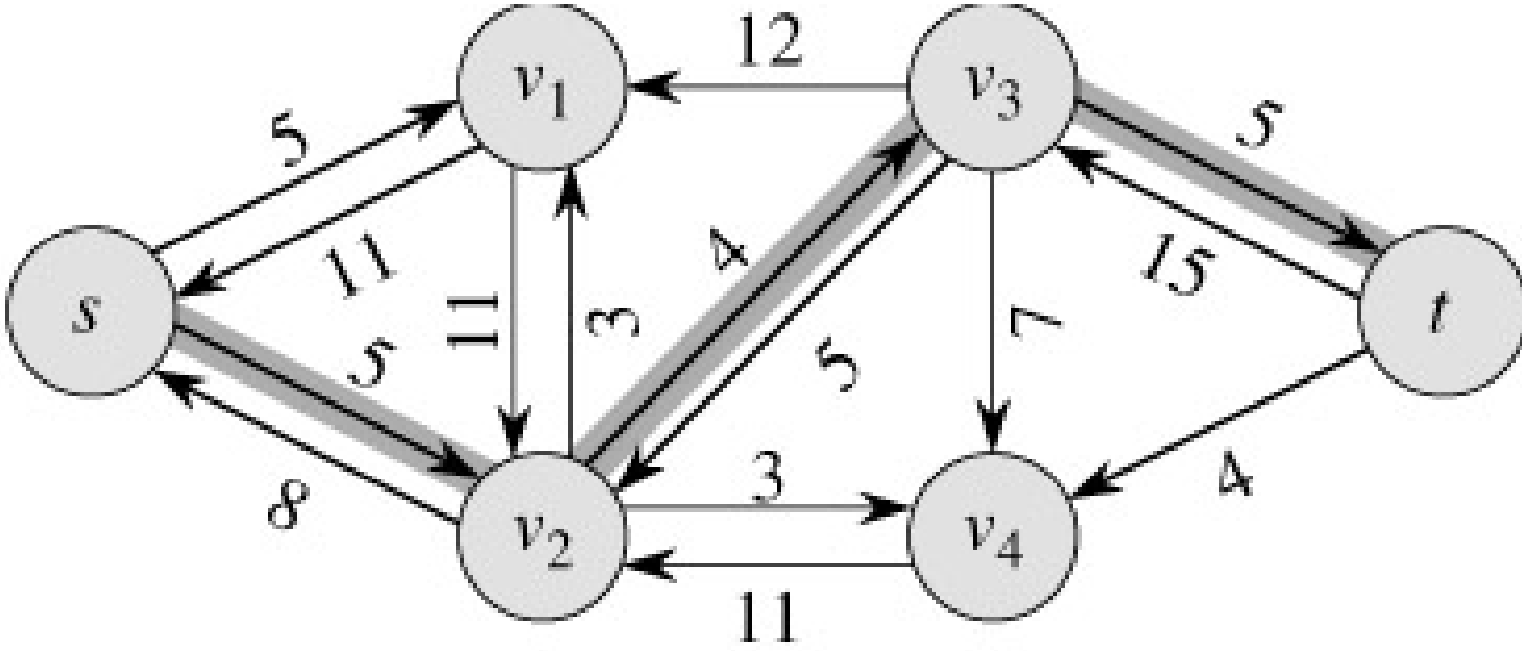
(a) (b) (c)

Figure 26.7 **(a)** A flow network for which FORD-FULKERSON can take $\Theta(E\ |f^*|)$ time, where f^* is a maximum flow, shown here with $|f^*| = 2{,}000{,}000$. The shaded path is an augmenting path with residual capacity 1. **(b)** The resulting residual network, with another augmenting path whose residual capacity is 1. **(c)** The resulting residual network.

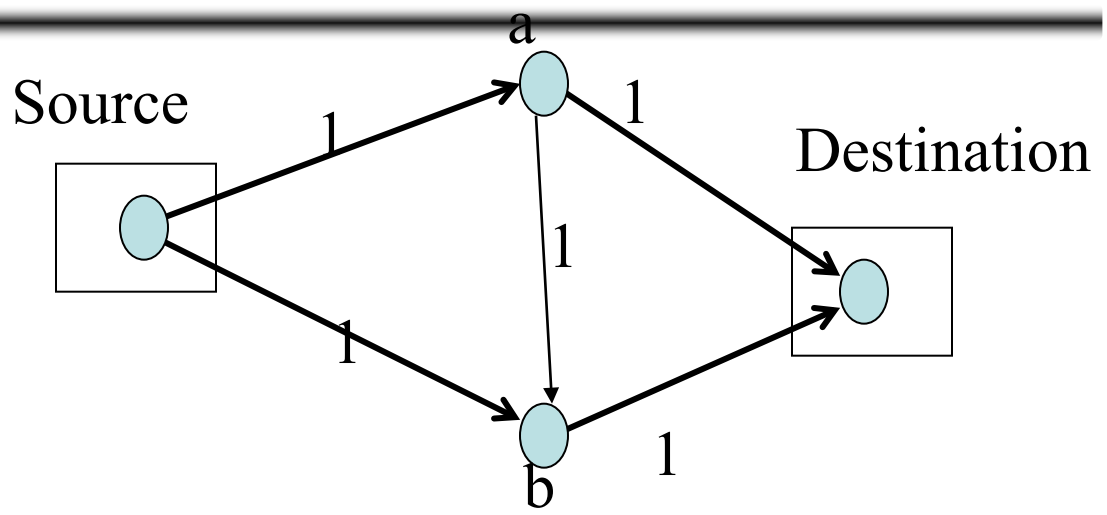
In practice, the maximum-flow problem often arises with integral capacities. If the capacities are rational numbers, we can apply an appropriate scaling transformation to make them all integral. If f^* denotes a maximum flow in the transformed network, then a straightforward implementation of FORD-FULKERSON executes the **while** loop of lines 3–8 at most $|f^*|$ times, since the flow value increases by at least one unit in each iteration.

The residual network

- The edges of the residual network are the edges on which the residual capacity is positive.



Example



Paths source, a, destinations and source, b destination gives a flow of 2 units.

Path source, a, b, destination overlaps with both the optimal paths.

If we initially choose source, a, b, destination as our path, then no greedy strategy will be able to augment the network flow any further (unless we use residual edges which allows recovery)

Verify how we recover in spite of the initial bad choice, if we use the residual network to augment flows.

Ford-Fulkerson method, with details

```
Ford-Fulkerson(G,s,t)
1  for each edge (u,v)∈G.E do
2     f(u,v) = f(v,u) = 0
3  while ∃ path p from s to t in residual network G_f do
4     c_f = min{c_f(u,v): (u,v)∈p}
5     for each edge (u,v) in p do
6        f(u,v) = f(u,v) + c_f
7        f(v,u) = -f(u,v)
8  return f
```

The algorithms based on this method differ in how they choose *p* in step 3.

Time Analysis I

- A complete analysis establishing which specific method is best is a complex task, however, because their running times depend on
  - The number of augmenting paths needed to find a maxflow
  - The time needed to find each augmenting path

Analysis

FORD-FULKERSON(G, s, t)

1 **for** each edge $(u, v) \in E[G]$
2 **do** $f[u, v] \leftarrow 0$
3 $f[v, u] \leftarrow 0$

} O(E)

4 **while** there exists a path p from s to t in the residual network G_f
5 **do** $c_f(p) \leftarrow \min\{c_f(u, v) : (u, v) \text{ is in } p\}$
6 **for** each edge (u, v) in p
7 **do** $f[u, v] \leftarrow f[u, v] + c_f(p)$
8 $f[v, u] \leftarrow -f[u, v]$

} O(E)

Summary of Ford-Fulkerson's Method

- Start with an initial feasible flow (often, zero flow).
- Find an **augmenting path** in the **residual graph**.
- Augment flow along this path.
- Repeat until no augmenting paths are left.

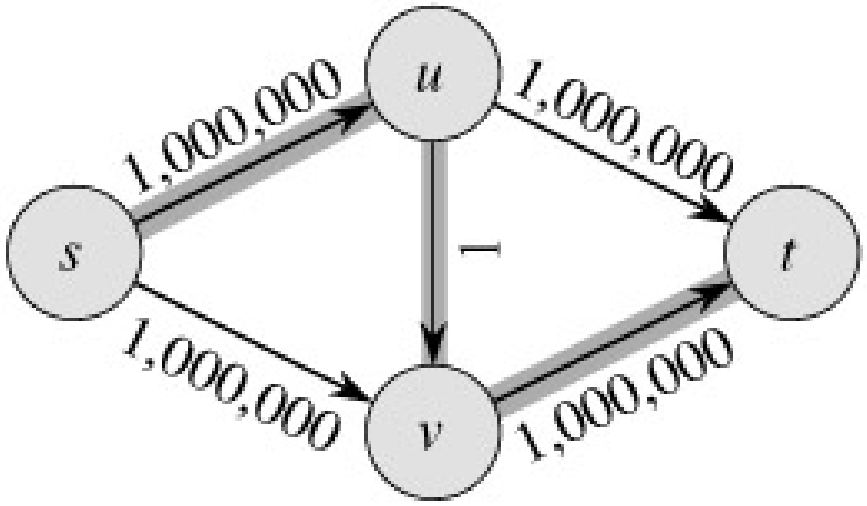
```
FORD-FULKERSON(G, s, t)
1  for each edge (u, v) ∈ E[G]                                   } O(E)
2      do f[u, v] ← 0
3         f[v, u] ← 0
4  while there exists a path p from s to t in the residual network G_f
5      do c_f(p) ← min {c_f(u, v) : (u, v) is in p}              } O(E)
6         for each edge (u, v) in p
7             do f[u, v] ← f[u, v] + c_f(p)
8                f[v, u] ← −f[u, v]
```

Analysis

- If capacities are all integer, then each augmenting path raises $|f|$ by ≥ 1.
- If max flow is f^*, then need $\leq |f^*|$ iterations
  - **Hence, the time is $O(E|f^*|)$.**
- Note that this running time is not polynomial in input size. It depends on $|f^*|$, which is not a function of $|V|$ or $|E|$.
- If capacities are rational, can scale them to integers.
- If irrational, FORD-FULKERSON might never terminate!

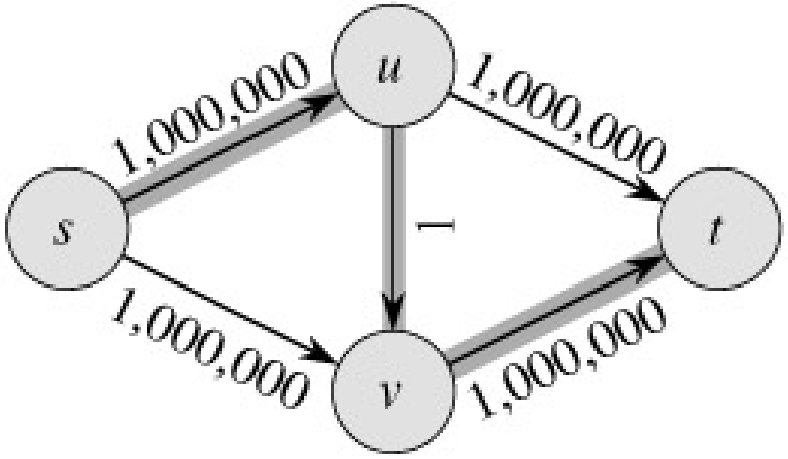
The basic Ford-Fulkerson Algorithm

- With time O (E |f*|), the algorithm is not polynomial.
- This problem is real: Ford-Fulkerson may perform very badly if we are unlucky:

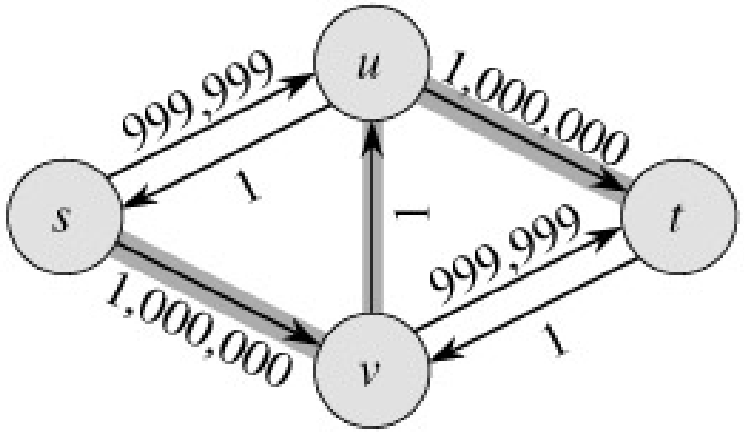


|f*|=2,000,000

Run Ford-Fulkerson on this example

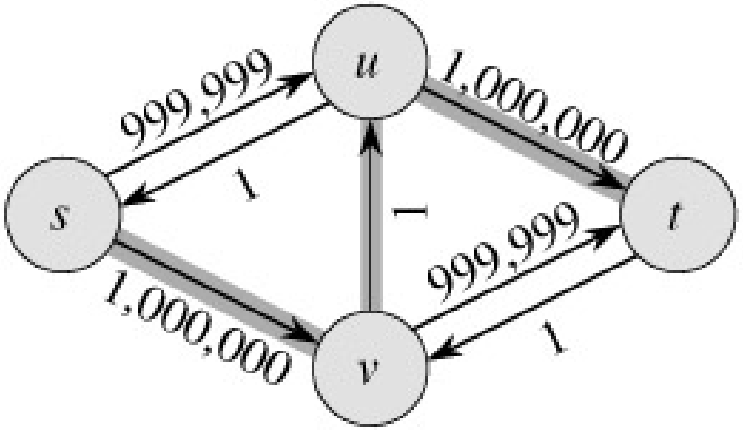


Augmenting Path

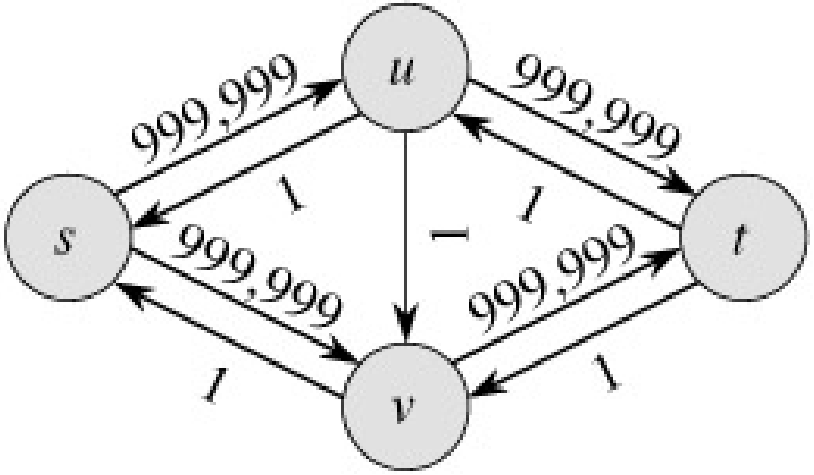


Residual Network

Run Ford-Fulkerson on this example

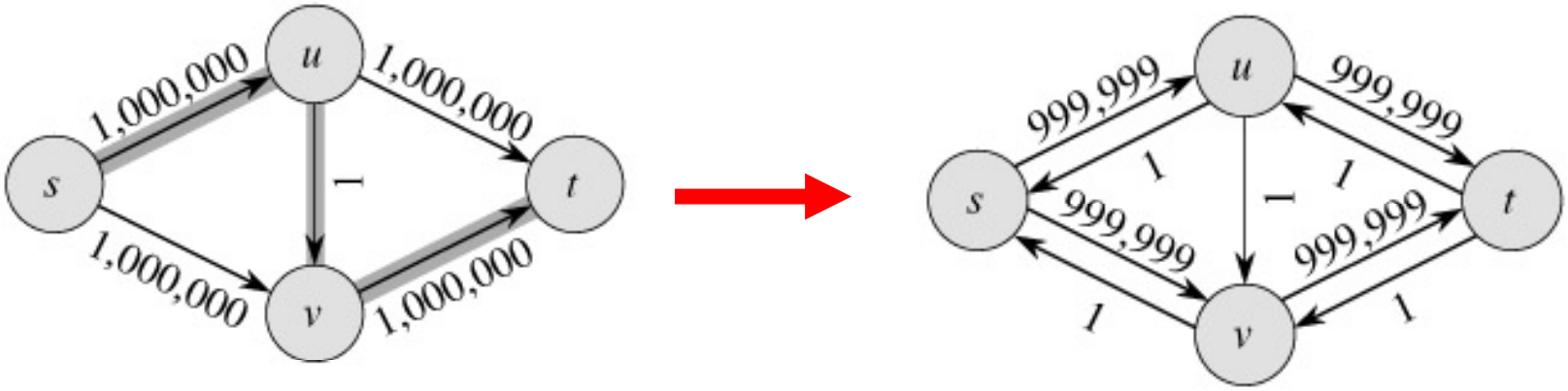


Augmenting Path



Residual Network

Run Ford-Fulkerson on this example



- Repeat 999,999 more times…

The Edmonds-Karp Algorithm

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

Edmonds-Karp is essentially a specific implementation of the Ford-Fulkerson method with a well-defined strategy for selecting augmenting paths

Shortest Augmenting Paths: Instead of just picking any augmenting path, Edmonds-Karp always chooses the shortest augmenting path (in terms of the number of edges). This selection is done using a Breadth-First Search (BFS) on the residual graph.

Why Shortest path in Edmonds-Karp Algorithm?

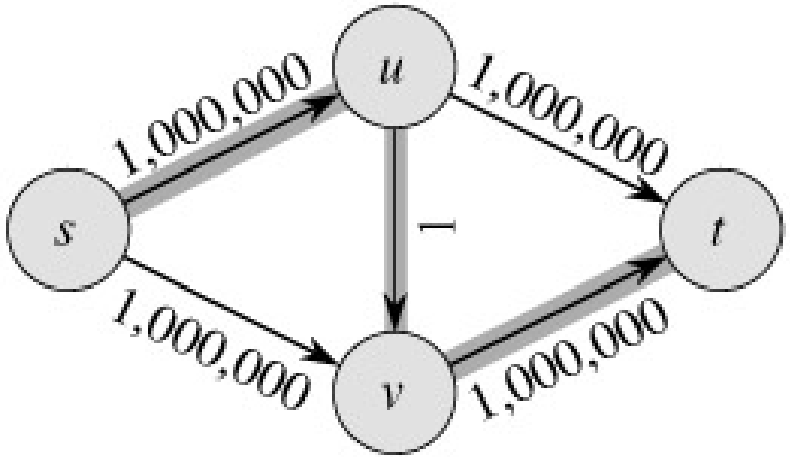
Termination

- By always picking the shortest path, Edmonds-Karp avoids the issue of entering an infinite loop where there are cycles in the residual graph don't increase the overall max-flow.
- By always picking the shortest path, Edmonds-Karp avoids this pitfall, ensuring termination.

Efficiency:

- By always augmenting along the shortest path, the "distance" (number of edges) from the source to any node in the residual graph **increases monotonically**.
- This property ensures that **each edge gets saturated** (i.e., its capacity in the residual graph drops to zero) in a bounded number of iterations.

The Edmonds-Karp Algorithm - example



- Edmonds-Karp's algorithm runs only 2 iterations on this graph.

Time Complexity

```
FORD-FULKERSON(G, s, t)
1  for each edge (u, v) ∈ E[G]
2      do f[u, v] ← 0
3         f[v, u] ← 0
4  while there exists a path p from s to t in the residual network G_f
5      do c_f(p) ← min {c_f(u, v) : (u, v) is in p}
6         for each edge (u, v) in p
7             do f[u, v] ← f[u, v] + c_f(p)
8                f[v, u] ← −f[u, v]
```

- By always choosing the shortest augmenting path, we can show that the capacities of the edges in the residual network will increase monotonically.
- As a result, any edge can become a bottleneck at most a limited number of times, bounding the number of iterations.

Time Complexity

```
FORD-FULKERSON(G, s, t)
1  for each edge (u, v) ∈ E[G]
2      do f[u, v] ← 0
3         f[v, u] ← 0
4  while there exists a path p from s to t in the residual network G_f
5      do c_f(p) ← min {c_f(u, v) : (u, v) is in p}
6         for each edge (u, v) in p
7             do f[u, v] ← f[u, v] + c_f(p)
8                f[v, u] ← −f[u, v]
```

- Let, total number of flow augmentations performed by Edmonds-Karp algorithm is O(VE)
- BFS to find the augmented path – O(E)
- So, Total running time is $O(VE^2)$

- This polynomial-time bound is a result of the algorithm's approach to always choosing the shortest augmenting path.

Conditions

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 augmented paths.
3. $|f| = c(S,T)$ for some cut (S,T) (a min-cut).

It is a flow since there is no augmented paths It is maximum since the sink is not reachable from the source

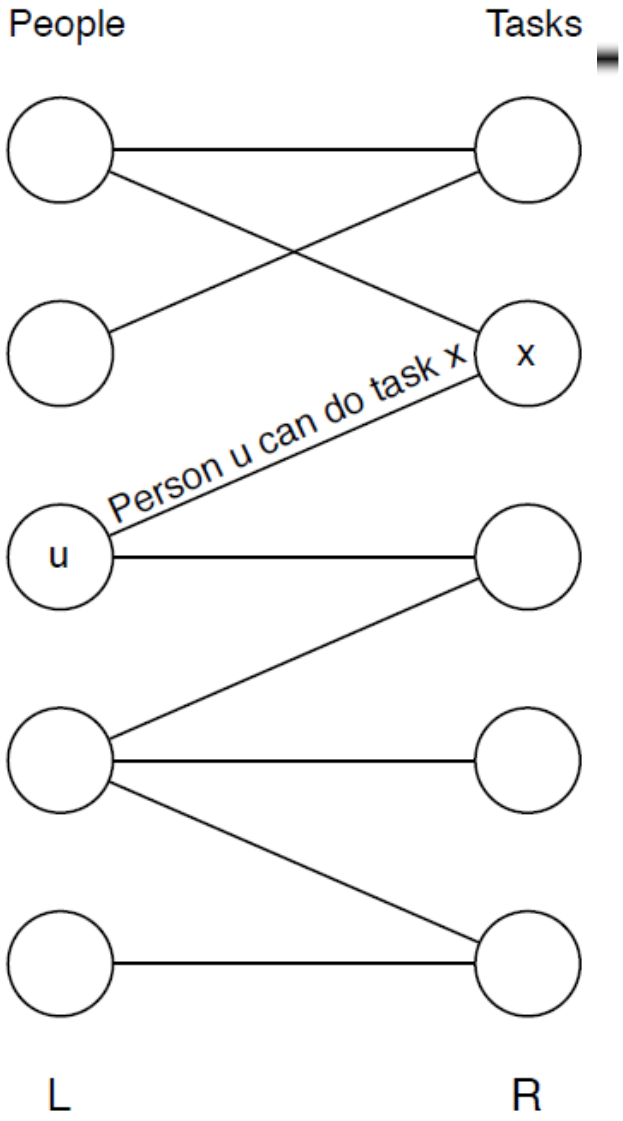
Theorem 26.6 (Max-flow min-cut theorem)

Maximum Bipartite Matching

- The **network flow problem** is itself interesting.
- But even more interesting is how you can use it to solve many problems that don't involve flows or even networks.

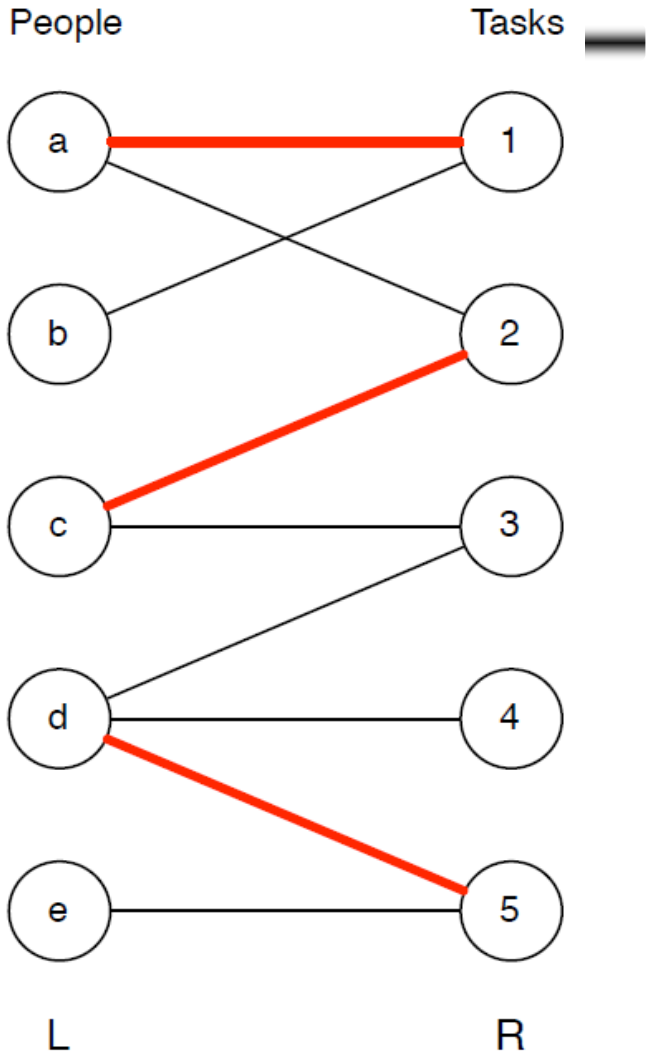
Bipartite Graphs

- Suppose we have a set of people L and set of jobs R.
- Each person can do only some of the jobs.
- Can model this as a bipartite graph $\rightarrow$



Bipartite Matching

- A **matching** gives an assignment of people to tasks.
- Want to get as many tasks done as possible.
- So, want a **maximum matching**: one that contains as many edges as possible.
- (This one is not maximum.)



Maximum Bipartite Matching

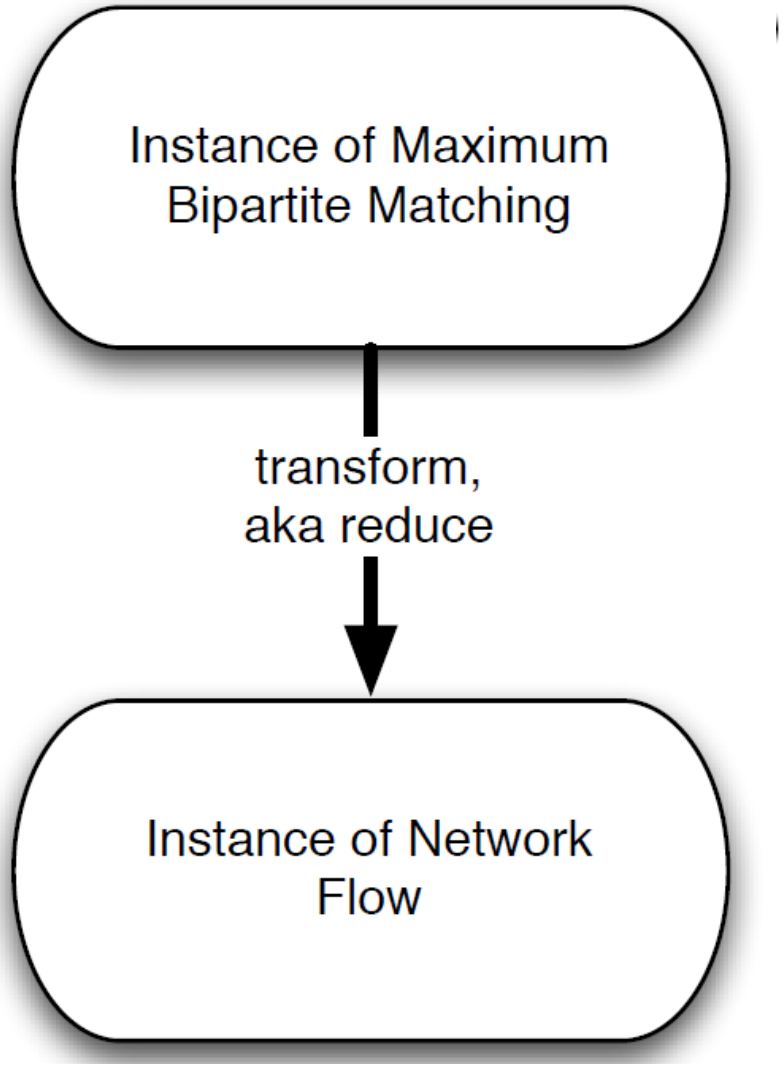
Maximum Bipartite Matching

Given a bipartite graph $G = (A \cup B, E)$, find an $S \subseteq A \times B$ that is a matching and is as large as possible.

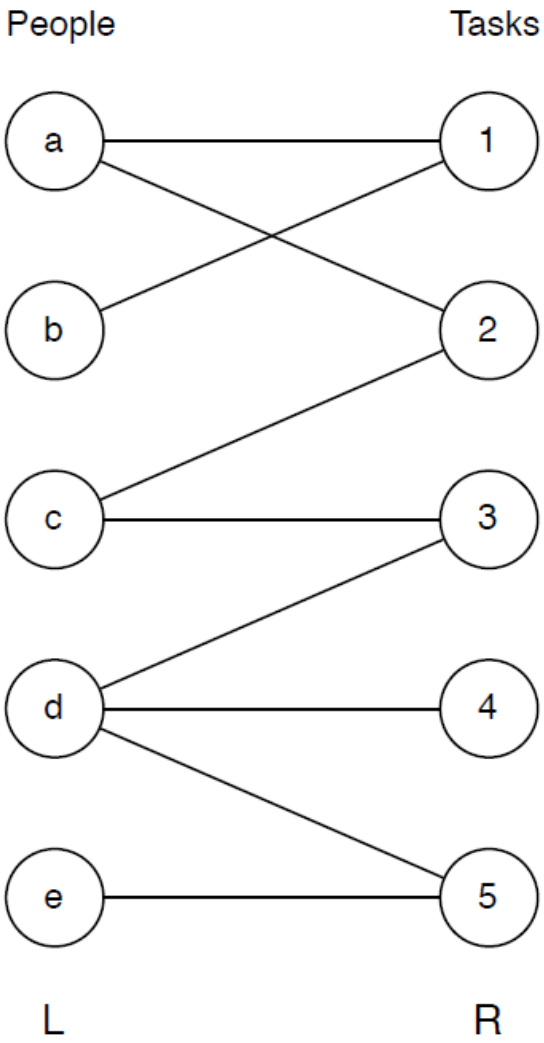
- We're given A and B so we don't have to find them.
- S is a **perfect matching** if every vertex is matched.
- *Maximum* is not the same as *maximal*: greedy will get to maximal.

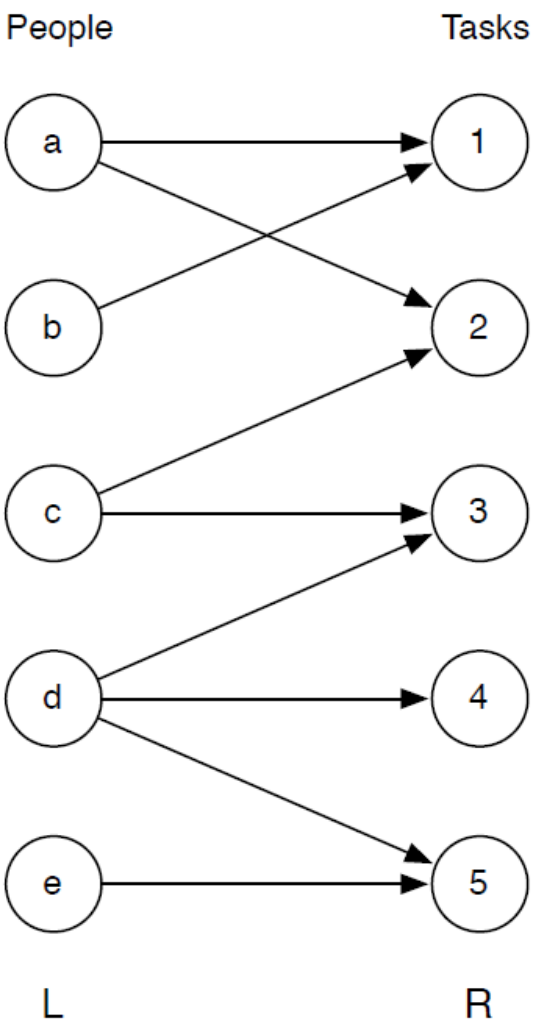
Reduce

- Given an instance of bipartite matching,
- Create an instance of network flow.
- Where the solution to the network flow problem can easily be used to find the solution to the bipartite matching.

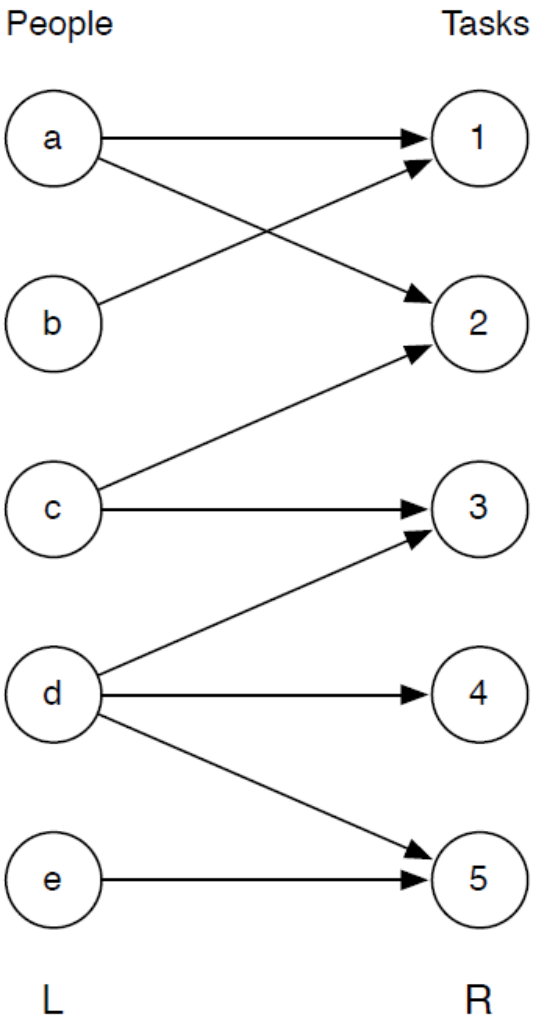


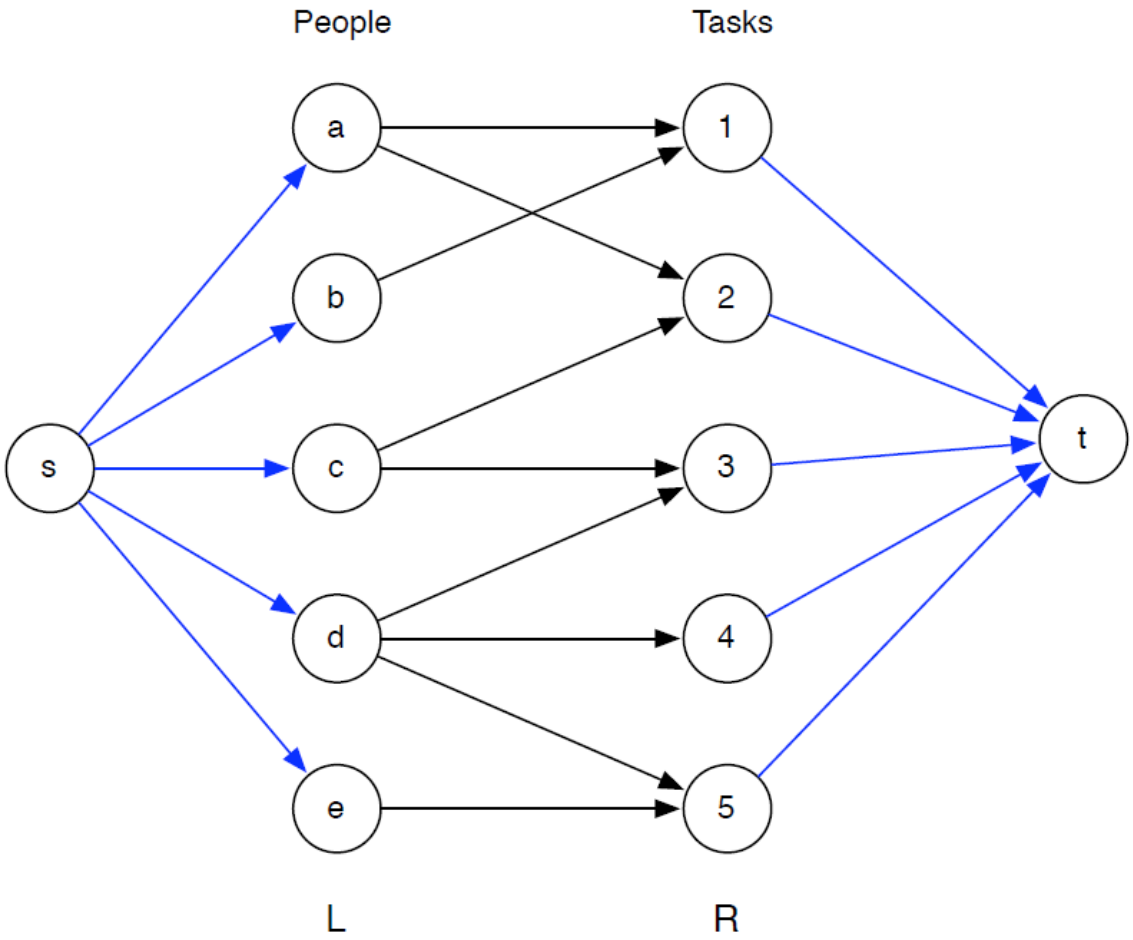
Reducing Bipartite Matching to Net Flow



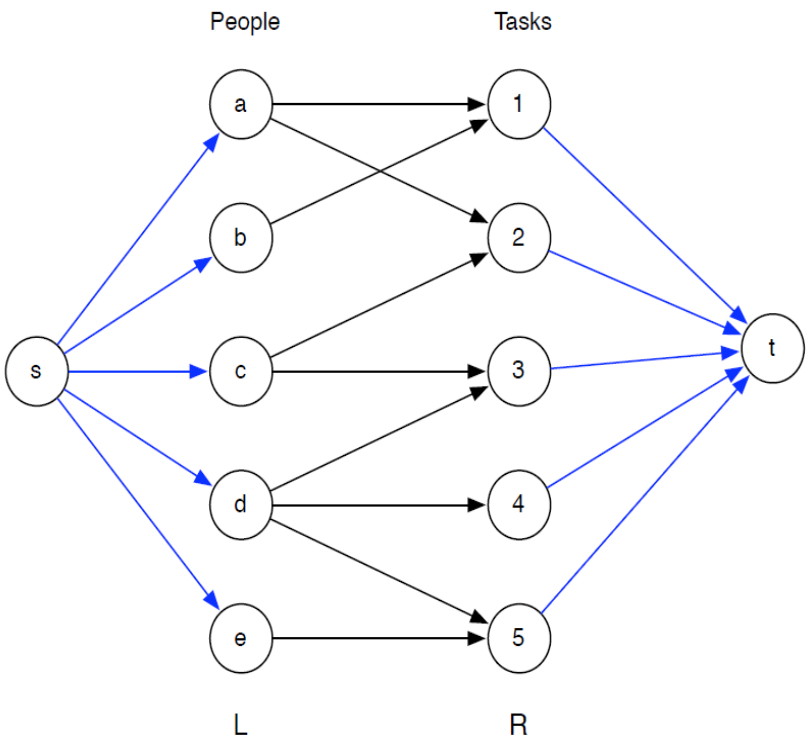


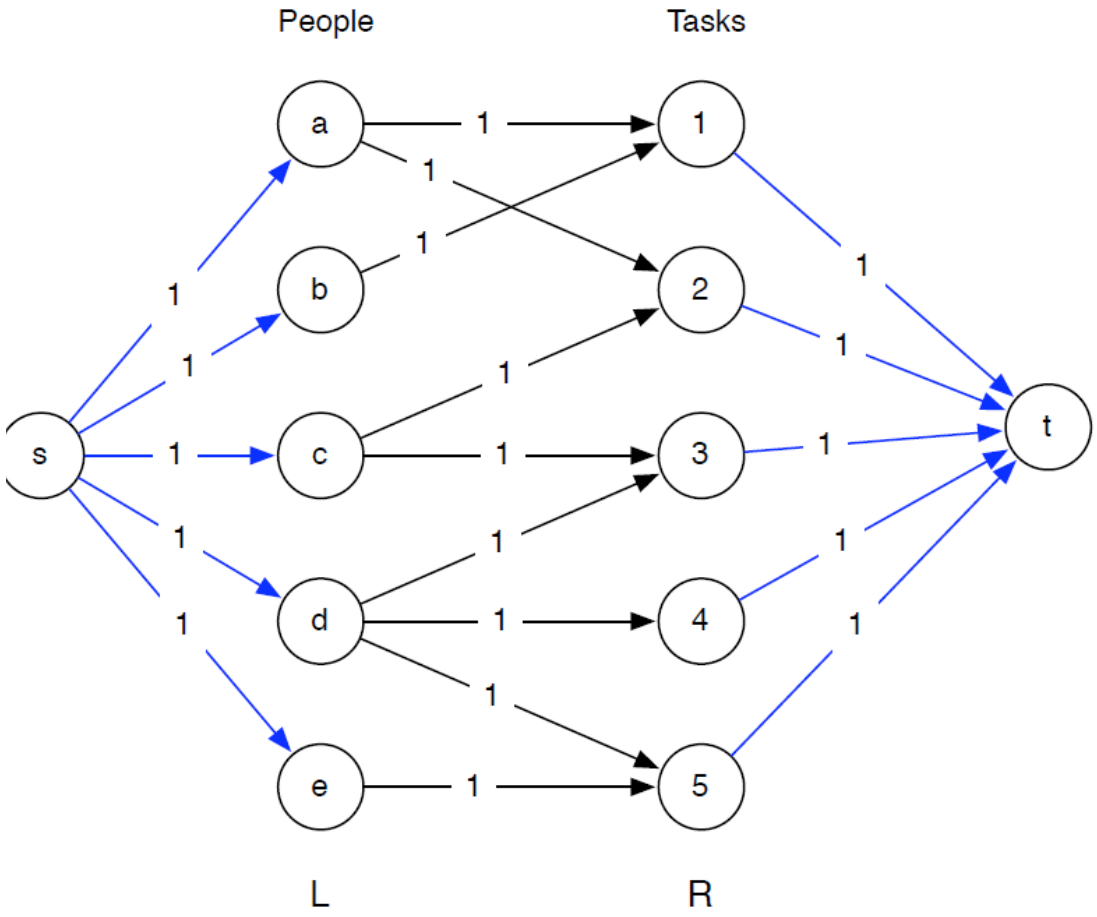
Reducing Bipartite Matching to Net Flow



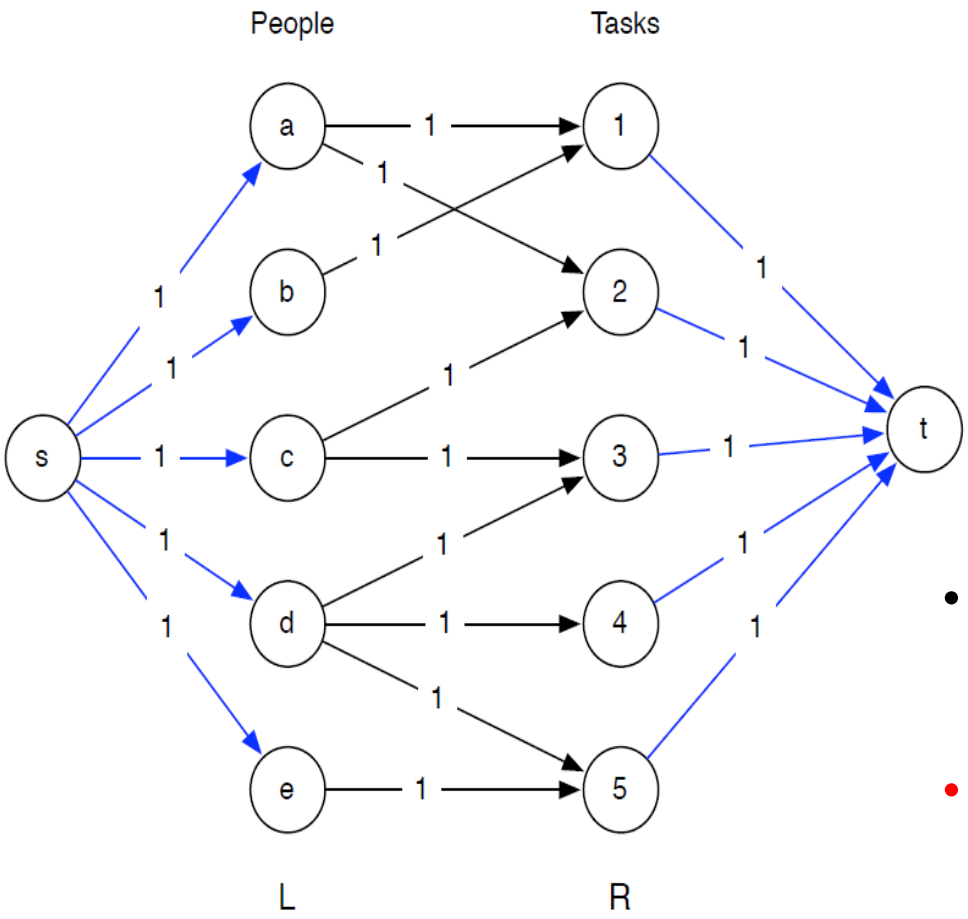


Reducing Bipartite Matching to Net Flow



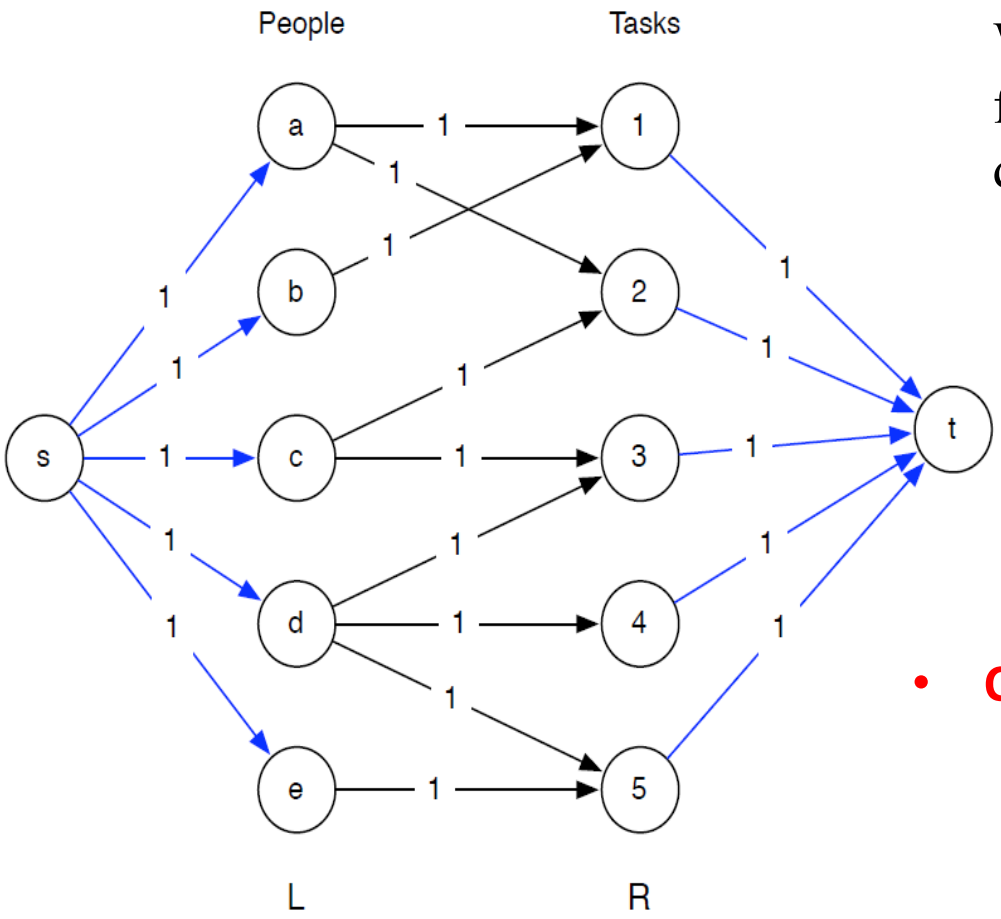


Reducing Bipartite Matching to Net Flow



- **Capacity = 1 from source to person**
  - Each person can do at most 1 task and no more
- If it is more than 1, then we can have people who can do multiple tasks.

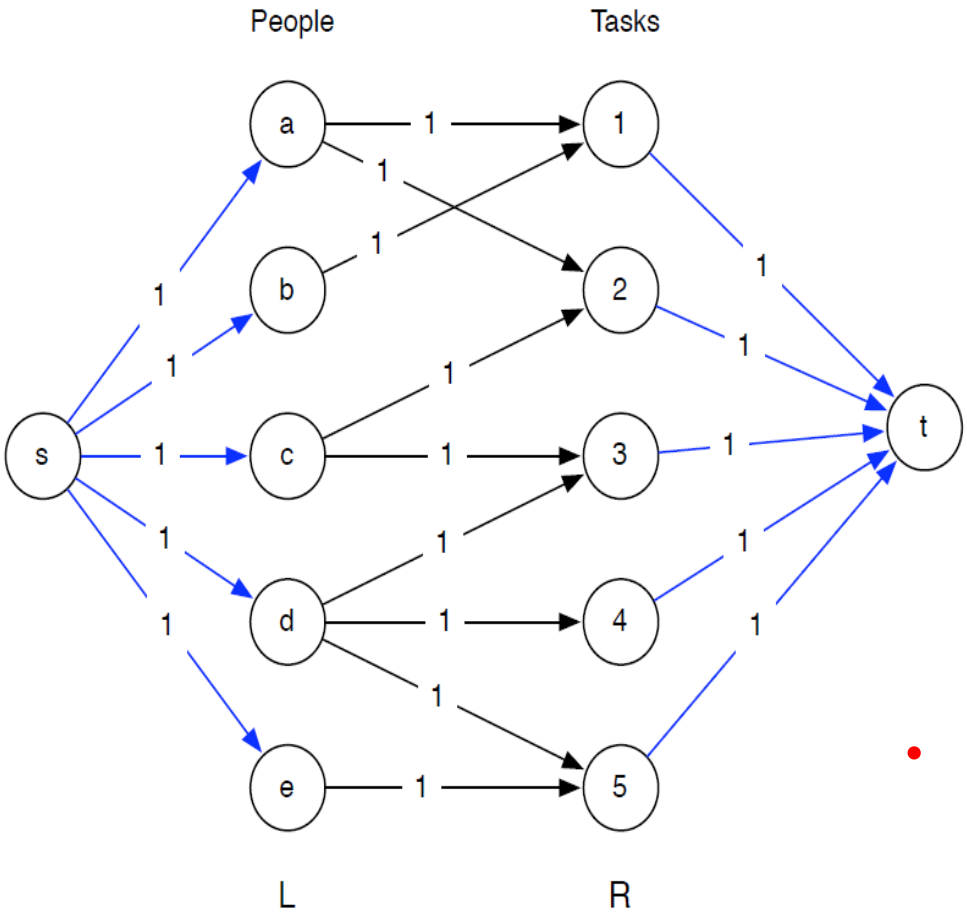
Reducing Bipartite Matching to Net Flow



What do we need to change in the flow network to allow a task to be done multiple time?

- **Capacity > 1 from that task to sink t**
  - Now those tasks can be done multiple times

Reducing Bipartite Matching to Net Flow

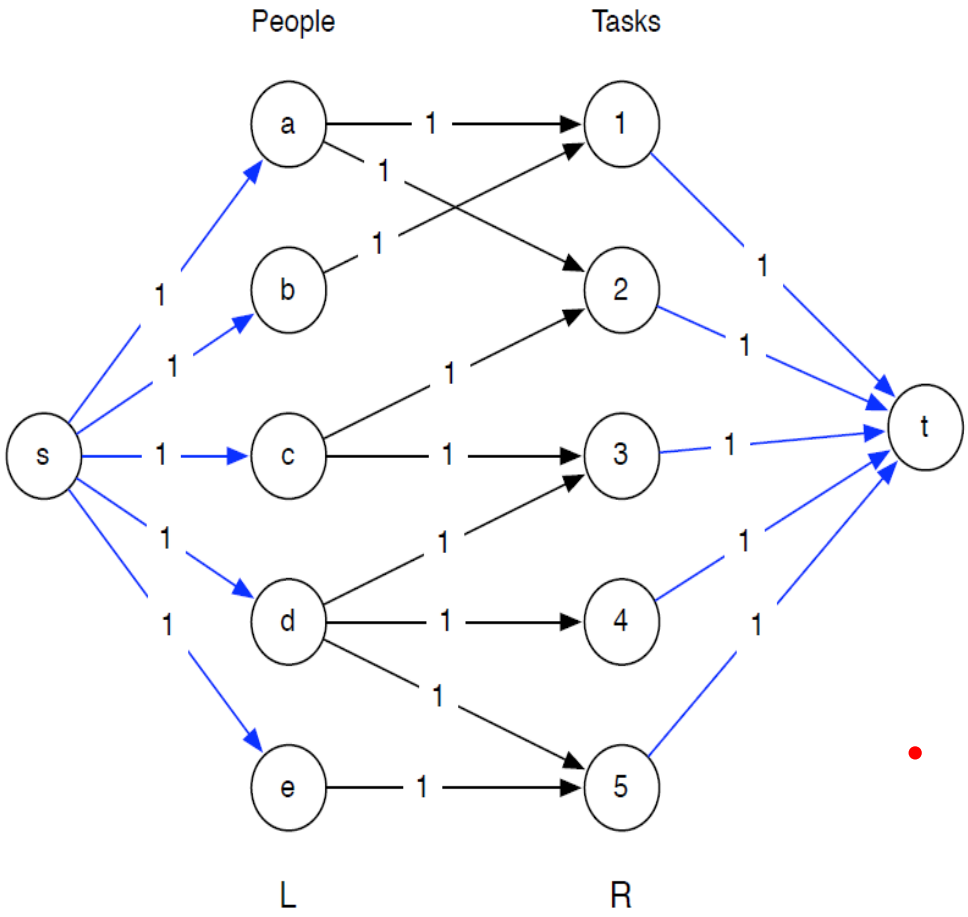


What do we need to change in the flow network to allow a task to be done multiple time?

- **Capacity > 1 from that task to sink t**
  - Now those tasks can be done multiple times

- Now, if we have **capacity more than 1** both in outgoing edges of S and incoming edges of t, we can have same tasks be selected multiple times by a single person

Reducing Bipartite Matching to Net Flow

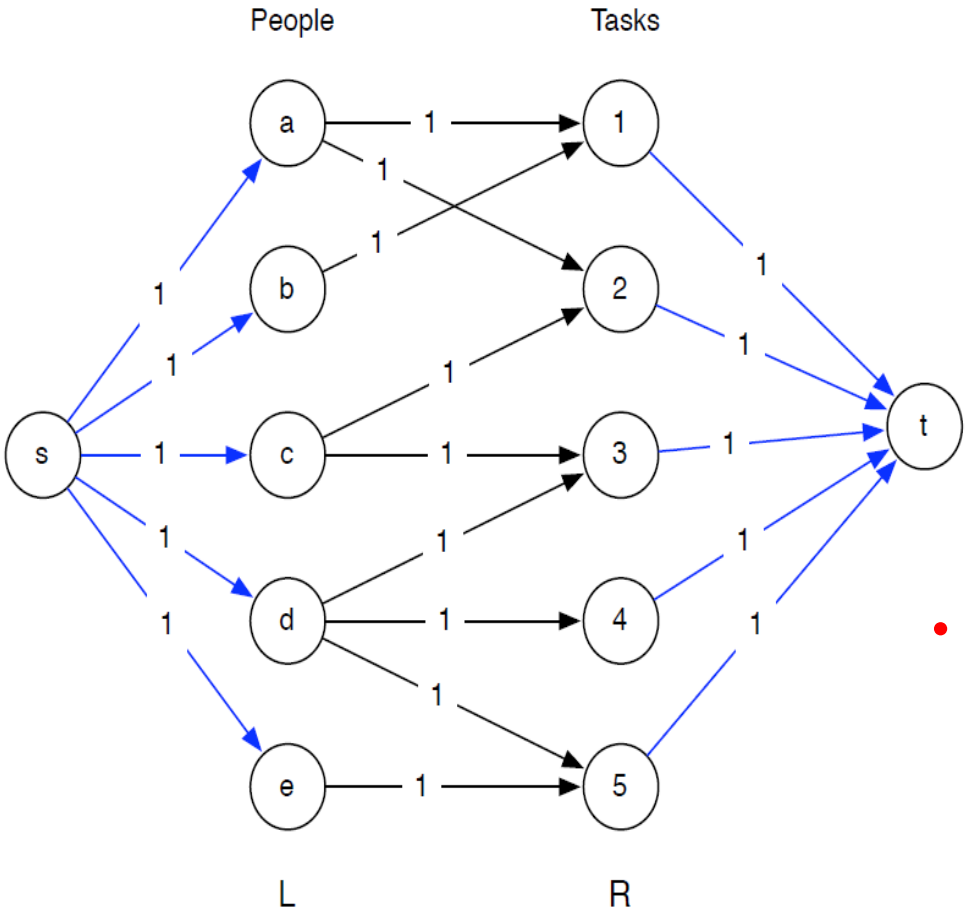


What do we need to change in the flow network to allow a task to be done multiple time?

- **Capacity > 1 from that task to sink t**
  - Now those tasks can be done multiple times

- Flow values on the edge from tasks to the sink indicates **the number of times** a task will be completed by different people.

Reducing Bipartite Matching to Net Flow

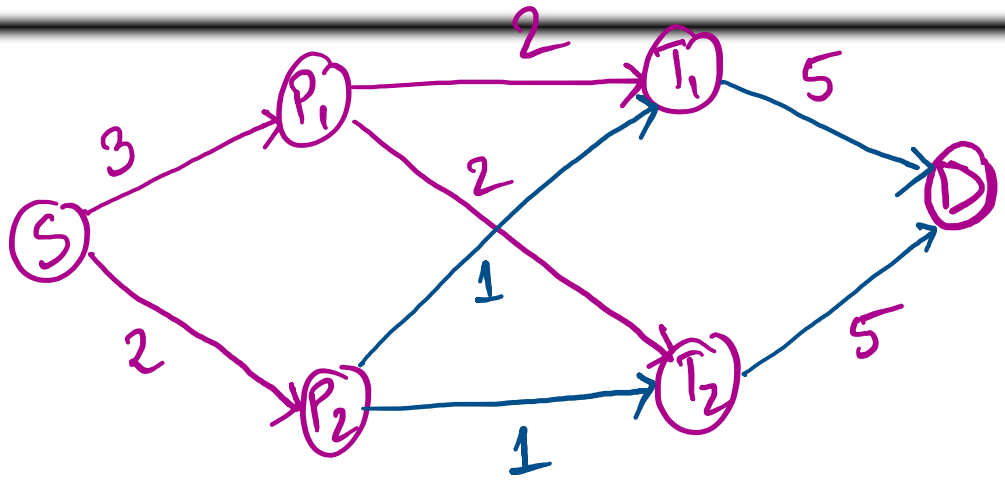


What do we need to change in the flow network to allow a task to be done multiple time?

- **Capacity > 1 from that task to sink t**
  - Now those tasks can be done multiple times

- Now, a Person is allowed complete a task once: capacity 1 in **people to task edges**

- Even though there are options of one task be completed more than once.
- Make the capacity 3 from c to task 2 to allow person C to do that task 3 times.


S
P1
P2
T1
T2
D
3
2
2
2
1
1
5
5