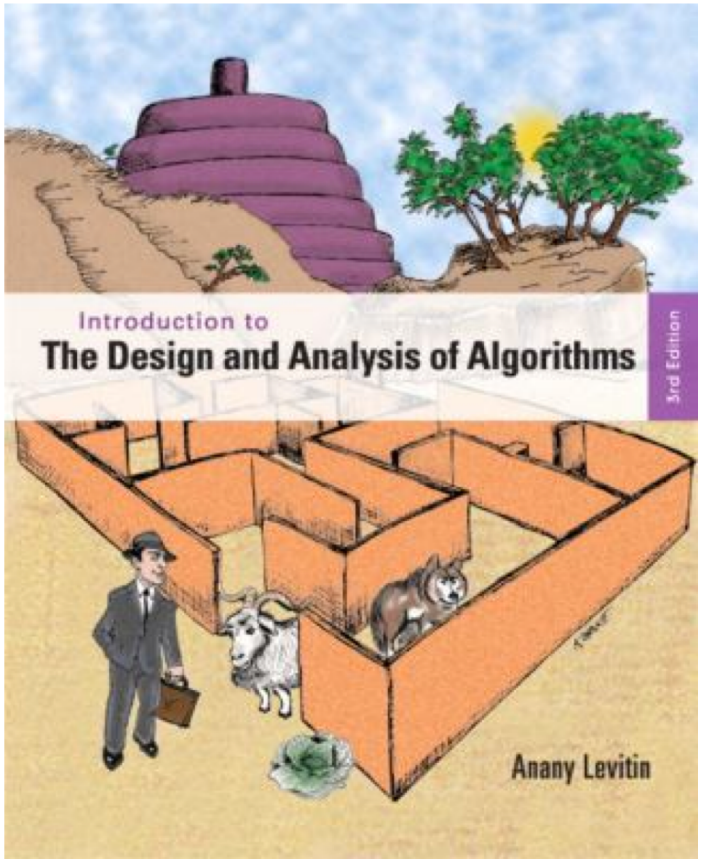
Chapter 10

Iterative Improvement



Iterative Improvement

Algorithm design technique for solving optimization problems

- Start with a feasible solution
- Repeat the following step until no improvement can be found:
  - change the current feasible solution to a feasible solution with a better value of the objective function
- Return the last feasible solution as optimal

Note: Typically, a change in a current solution is "small" (*local search*)

Major difficulties

- For some problem, finding an initial solution may require as much effort as solving the problem

- It is not always clear what changes should be allowed in a feasible solution so that we can check efficiently whether the current solution is locally optimal and, if not, replace it with a better one

- Local optimum vs. global optimum

Problems that can be solved using Iterative Improvement

- Linear programming
  - ➢simplex method
- Maximum flow problem
  - Ford-Fulkerson algorithm
- Maximum matching of graph vertices
- Gale-Shapley algorithm for the stable marriage problem
- Local search heuristics

Linear Programming

Linear programming (LP) problem is to optimize a linear function of several variables subject to linear constraints:

maximize (or minimize) $c_1 x_1 + \ldots + c_n x_n$

subject to $a_{i\,1} x_1 + \ldots + a_{in} x_n \leq$ (or $\geq$ or $=$) b_i, $i = 1,\ldots,m$
$x_1 \geq 0, \ldots, x_n \geq 0$

The function $z = c_1 x_1 + \ldots + c_n x_n$ is called the *objective function*;

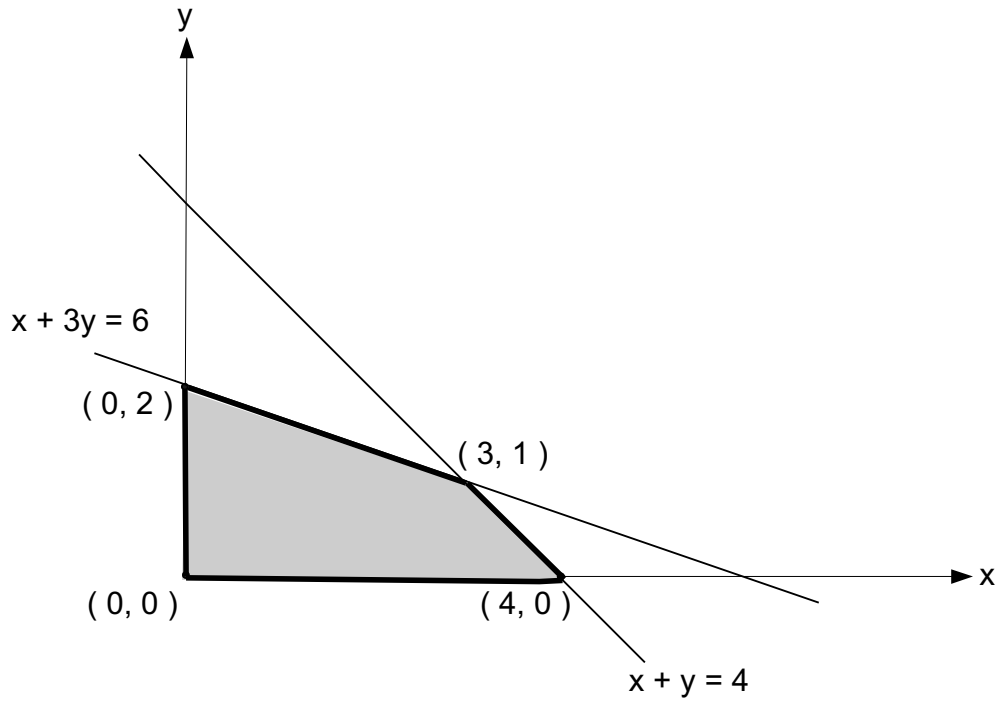
constraints $x_1 \geq 0, \ldots, x_n \geq 0$ are called *nonnegativity constraints*

Example

maximize $3x + 5y$
subject to $x + y \le 4$
$x + 3y \le 6$
$x \ge 0,\ y \ge 0$

A ***feasible solution*** to this problem is any point (x, y) that satisfies all the constraints of the problem

Feasible region is the set of points defined by the constraints



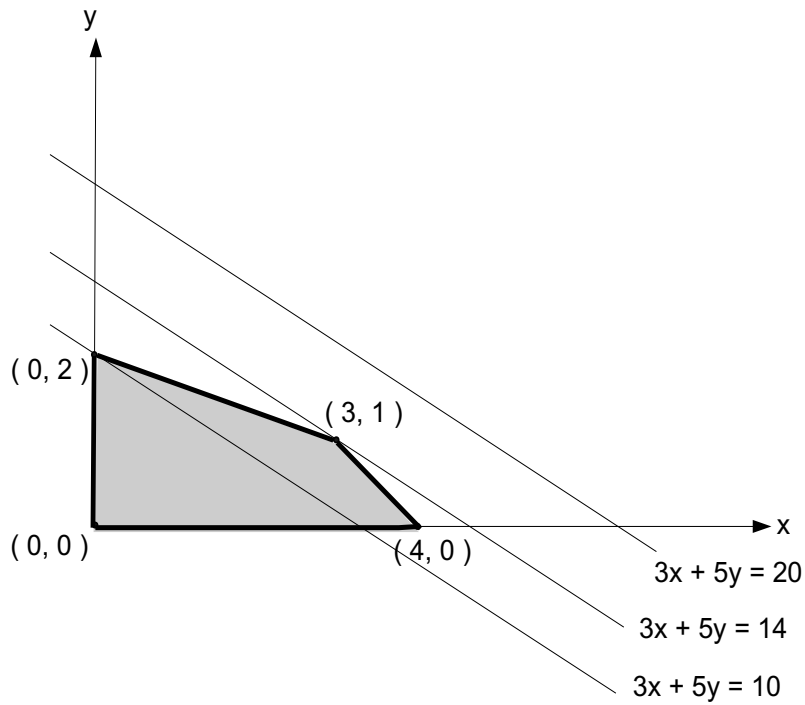
Geometric solution

maximize $3x + 5y$

subject to $x + y \leq 4$

$x + 3y \leq 6$

$x \geq 0,\ y \geq 0$



Optimal solution: x = 3, y = 1

Question

- Does every linear programming problem have an optimal solution that can be found at a vertex of its feasible region?

- If the constraints include two contradictory requirements, there can be no points in the problem's feasible region. Linear programming problems with the empty feasible region are called ***infeasible***.

Geometric solution

- If the feasible region of a linear programming problem is **unbounded**, its objective function may or may not attain a finite optimal value on it.

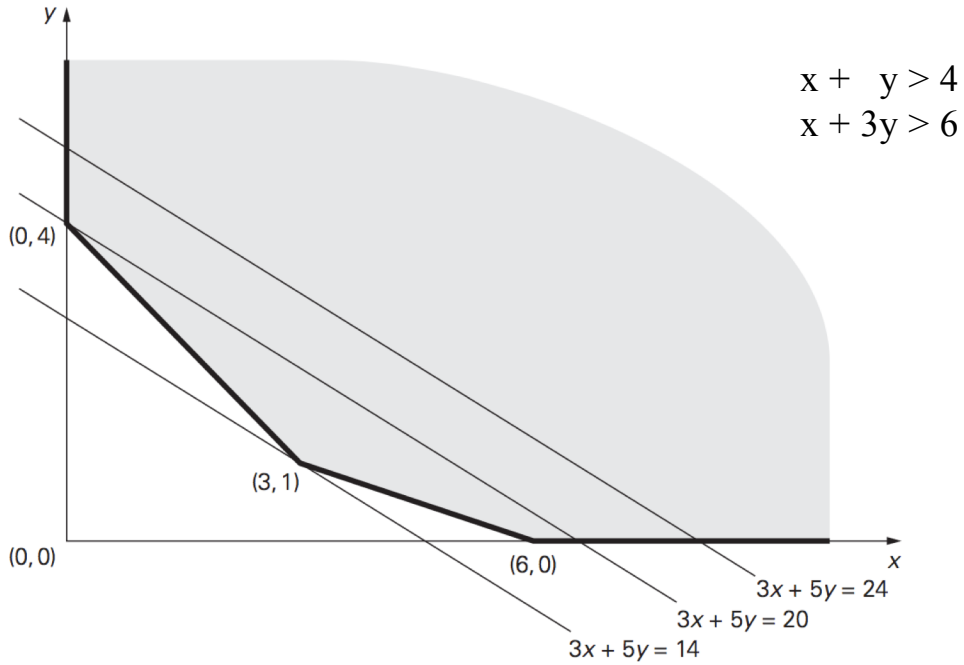


FIGURE 10.3 Unbounded feasible region of a linear programming problem with constraints $x + y \geq 4$, $x + 3y \geq 6$, $x \geq 0$, $y \geq 0$, and three level lines of the function $3x + 5y$.

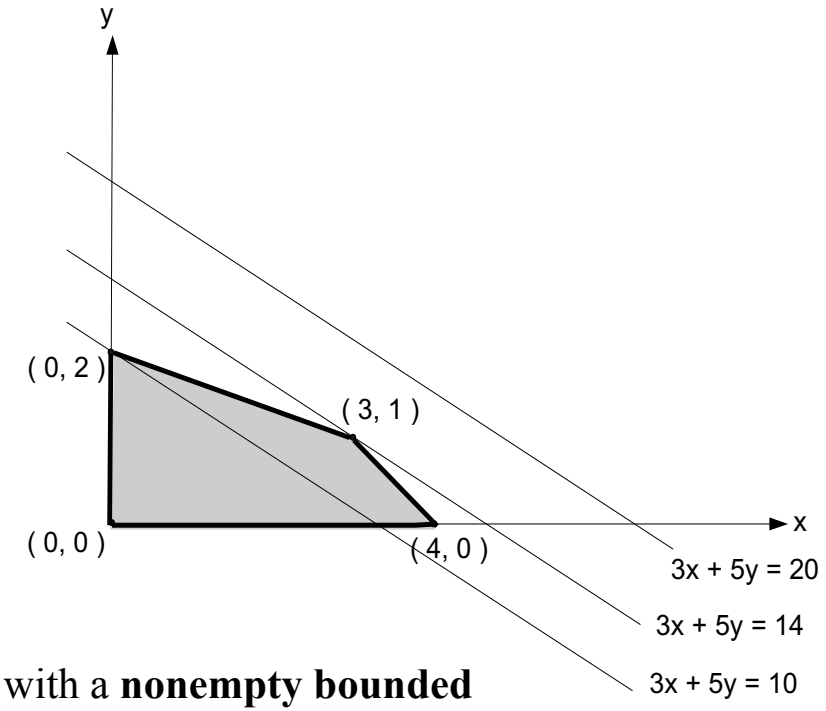
3 possible outcomes in solving an LP problem

- has a finite optimal solution, which may not be unique
- *unbounded*: the objective function of maximization (minimization) LP problem is unbounded from above (below) on its feasible region
- *infeasible*: there are no points satisfying all the constraints, i.e. the constraints are contradictory

THEOREM (*Extreme Point Theorem*)

For a typical linear programming problem:

- Its feasible region has a finite number of vertices, called ***extreme points***
- An optimal solution to a linear programming problem can be found at one of the extreme points of its feasible region.



Extreme Point Theorem Any LP problem with a **nonempty bounded** feasible region has an optimal solution; moreover, an optimal solution can always be found at an *extreme point* of the problem's feasible region.

The Simplex Method

- The classic method for solving LP problems;
  one of the most important algorithms ever invented

- Invented by George Dantzig in 1947

- Based on the iterative improvement idea:

  Generates a sequence of adjacent points of the problem's feasible region with improving values of the objective function until no further improvement is possible

The Simplex Method (cont.)

- Steps:
  - Start by identifying an extreme point of the feasible region.
  - Then check whether one can get an improved value of the objective function by going to an adjacent extreme point.
    - ✓ If it is not the case, the current point is optimal—stop;
    - ✓ if it is the case, proceed to an adjacent extreme point with an improved value of the objective function.
  - After a finite number of steps, the algorithm will either reach an extreme point where an optimal solution occurs or determine that no optimal solution exists.

Standard form of LP problem

- must be a maximization problem
- all constraints (except the nonnegativity constraints) must be in the form of linear equations
- all the variables must be required to be nonnegative

Thus, the general linear programming problem in standard form with m constraints and n unknowns ($n \geq m$) is

maximize $c_1 x_1 + \ldots + c_n x_n$

subject to $a_{i\,1} x_1 + \ldots + a_{in} x_n = b_i, \quad i = 1,\ldots,m,$
$x_1 \geq 0, \ldots, x_n \geq 0$

Every LP problem can be represented in such form

Transforming a LP into an equivalent problem in standard form

- Minimum problem can be replaced by maximizing the same objective function with all its coefficients c_j replaced by $-c_j$, j = 1, 2, . . . , n

- If a constraint is given as an inequality, it can be replaced by an equivalent equation by adding a ***slack variable*** representing the difference between the two sides of the original inequality.

- A negative variable x_j can be replaced by the difference between two new nonnegative variables: $x_j = x' - x''$, $x' \geq 0$, $x'' \geq 0$.

Example

maximize $3x + 5y$
subject to $x + y \le 4$
$x + 3y \le 6$
$x \ge 0, \quad y \ge 0$

maximize $3x + 5y + 0u + 0v$
subject to $x + y + u = 4$
$x + 3y + v = 6$
$x \ge 0, \ y \ge 0, \ u \ge 0, \ v \ge 0$

Variables u and v, transforming inequality constraints into equality constrains, are called ***slack variables***

Basic feasible solutions

A ***basic solution*** to a system of ***m*** linear equations in ***n*** unknowns ($n \geq m$) is obtained by setting ***n – m*** variables to 0 and solving the resulting system to get the values of the other ***m*** variables. The variables set to 0 are called ***nonbasic***; the variables obtained by solving the system are called ***basic***.

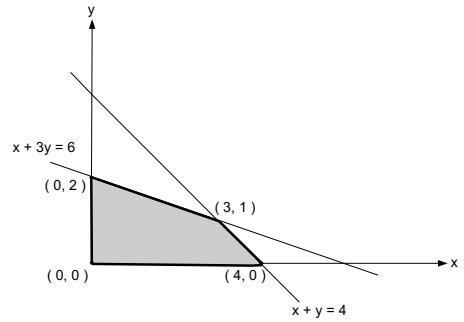
A **basic solution** is called ***feasible*** if all its (basic) variables are nonnegative.

Example $x + y + u = 4$

$x + 3y + v = 6$

(0, 0, 4, 6) is **basic feasible solution**

(*x*, *y* are **nonbasic**; *u*, *v* are **basic**)



There is a 1-1 correspondence between extreme points of LP's feasible region and its basic feasible solutions.

Simplex Tableau

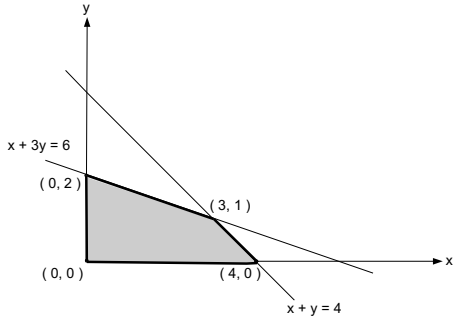
An extreme point can be represented by a ***simplex tableau***, a table storing the information about the basic feasible solution corresponding to the extreme point.

maximize $z = 3x + 5y + 0u + 0v$

subject to $x + y + u = 4$

$x + 3y + v = 6$

$x \geq 0,\ y \geq 0,\ u \geq 0,\ v \geq 0$



	x	y	u	v	
basic variable u	1	1	1	0	4
basic variable v	1	3	0	1	6
objective row	−3	−5	0	0	0

basic variables

basic feasible solution

(0, 0, 4, 6)

objective row

value of z at (0, 0, 4, 6)

Objective row

The last row of a simplex tableau is called the ***objective row***.

It is initialized by the coefficients of the objective function with their signs reversed (in the first n columns) and the value of the objective function at the initial point (in the last column).

On subsequent iterations, the objective row is transformed the same way as all the other rows.

The objective row is used by the simplex method to check whether the current tableau represents an optimal solution:

- it does if all the entries in the objective row—except, possibly, the one in the last column—are nonnegative.
- If this is not the case, any of the negative entries indicates a nonbasic variable that can become basic in the next tableau.

Simplex Tableau (cont.)

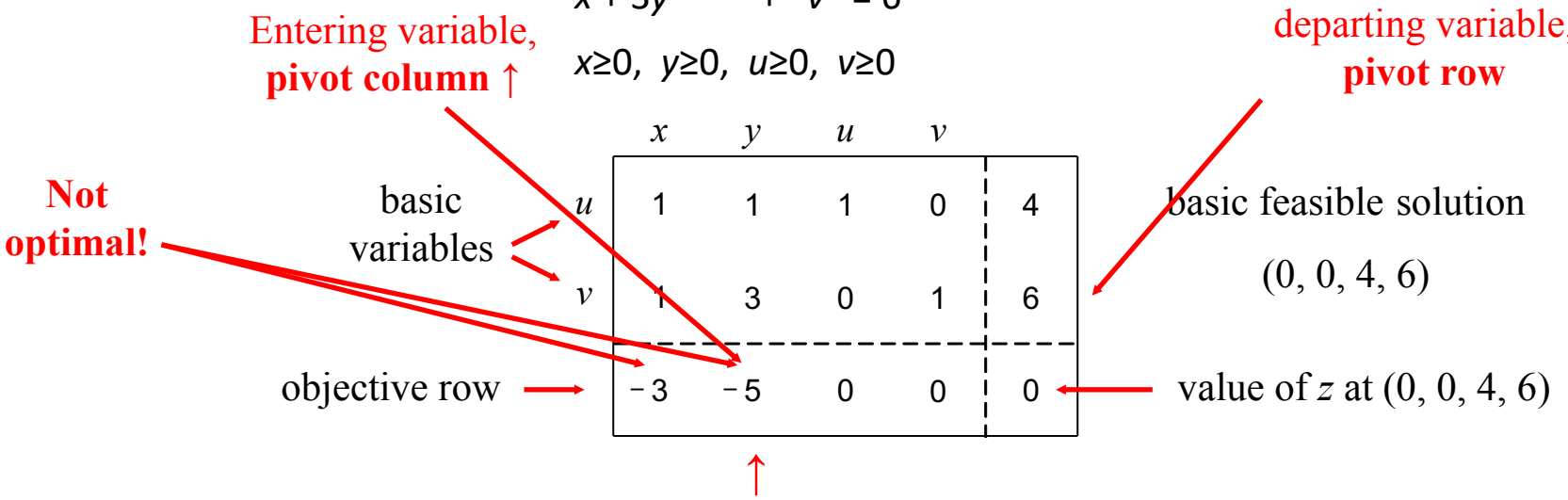
The negative value in the x-column signals the fact that we can increase the value of the objective function z = 3x + 5y + 0u + 0v by increasing the value of the x-coordinate in the current basic feasible solution (0, 0, 4, 6).

maximize $z = 3x + 5y + 0u + 0v$

subject to $x + y + u = 4$

$x + 3y + v = 6$

$x \geq 0,\ y \geq 0,\ u \geq 0,\ v \geq 0$



basic variables	x	y	u	v	
u	1	1	1	0	4
v	1	3	0	1	6
objective row	−3	−5	0	0	0

Transforming a Tableau

- Divide all the entries in the pivot row by its entry in the pivot column.

$$\overleftarrow{\text{row}}_{\text{new}}: \quad \frac{1}{3} \quad 1 \quad 0 \quad \frac{1}{3} \quad 2$$

- Subtract from each of the other rows, including the objective row, the new pivot row multiplied by the entry in the pivot column of the row in question.

$$\text{row} - c \cdot \overleftarrow{\text{row}}_{\text{new}}$$

where c is the row's entry in the pivot column

$$\text{row } 1 - 1 \cdot \overleftarrow{\text{row}}_{\text{new}}: \quad \frac{2}{3} \quad 0 \quad 1 \quad -\frac{1}{3} \quad 2$$

$$\text{row } 3 - (-5) \cdot \overleftarrow{\text{row}}_{\text{new}}: \quad -\frac{4}{3} \quad 0 \quad 0 \quad \frac{5}{3} \quad 10$$

- Replace the label of the pivot row by the variable's name of the pivot column.

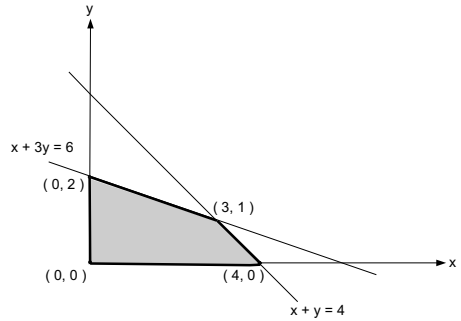
Example of Simplex Method Application

maximize $z = 3x + 5y + 0u + 0v$

subject to $x + y + u = 4$

$x + 3y + v = 6$

$x \geq 0,\ y \geq 0,\ u \geq 0,\ v \geq 0$



	x	y	u	v	
u	1	1	1	0	4
← v	1	3	0	1	6
	−3	−5	0	0	0

basic feasible sol.
(0, 0, 4, 6)

$z = 0$

	x	y	u	v	
← u	$\frac{2}{3}$	0	1	$-\frac{1}{3}$	2
y	$\frac{1}{3}$	1	0	$\frac{1}{3}$	2
	$-\frac{4}{3}$	0	0	$\frac{5}{3}$	10

basic feasible sol.
(0, 2, 2, 0)

$z = 10$

	x	y	u	v	
x	1	0	$\frac{3}{2}$	$-\frac{1}{2}$	3
y	0	1	$-\frac{1}{2}$	$\frac{1}{2}$	1
	0	0	2	1	14

basic feasible sol.
(3, 1, 0, 0)

$z = 14$

Outline of the Simplex Method

Step 0 [Initialization] Present a given LP problem in standard form and set up initial tableau.

Step 1 [Optimality test] If all entries in the objective row are nonnegative — stop: the tableau represents an optimal solution.

Step 2 [Find entering variable] Select (the most) negative entry in the objective row. Mark its column to indicate the entering variable and the pivot column.

Step 3 [Find departing variable] For each positive entry in the pivot column, calculate the θ-ratio by dividing that row's entry in the rightmost column by its entry in the pivot column. (If there are no positive entries in the pivot column — stop: the problem is unbounded.) Find the row with the smallest θ-ratio, mark this row to indicate the departing variable and the pivot row.

Step 4 [Form the next tableau] Divide all the entries in the pivot row by its entry in the pivot column. Subtract from each of the other rows, including the objective row, the new pivot row multiplied by the entry in the pivot column of the row in question. Replace the label of the pivot row by the variable's name of the pivot column and go back to Step 1.

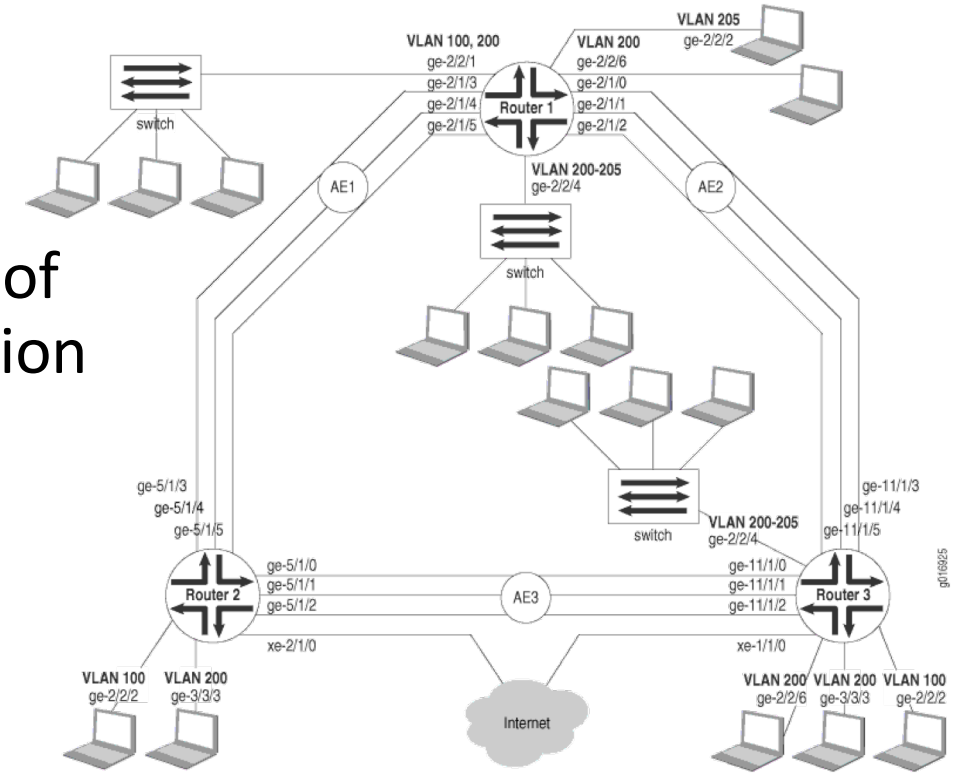
Notes on the Simplex Method

- Finding an initial basic feasible solution may pose a problem
- Theoretical possibility of cycling
- Typical number of iterations is between m and $3m$, where m is the number of equality constraints in the standard form
- Worse-case efficiency is exponential
- More recent *interior-point algorithms* such as *Karmarkar's algorithm* (1984) have polynomial worst-case efficiency and have performed competitively with the simplex method in empirical tests

Maximum Flow Problem

Problem of maximizing the flow of a material through a transportation network

- pipeline system
- communications
- transportation networks



Maximum Flow Problem

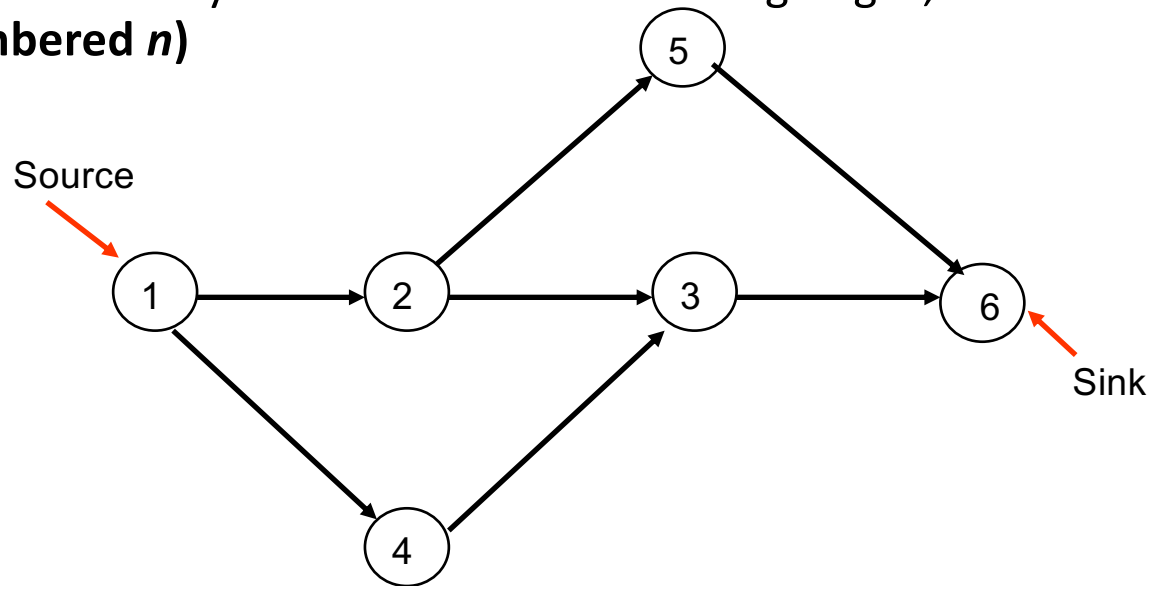
Represented by a connected weighted digraph with *n* vertices numbered from 1 to *n* with the following properties:

- contains exactly one vertex with no entering edges, called the ***source*** **(numbered 1)**
- contains exactly one vertex with no leaving edges, called the ***sink*** **(numbered *n*)**

Maximum Flow Problem

Represented by a connected weighted digraph with *n* vertices numbered from 1 to *n* with the following properties:

- contains exactly one vertex with no entering edges, called the ***source*** **(numbered 1)**
- contains exactly one vertex with no leaving edges, called the ***sink*** **(numbered *n*)**



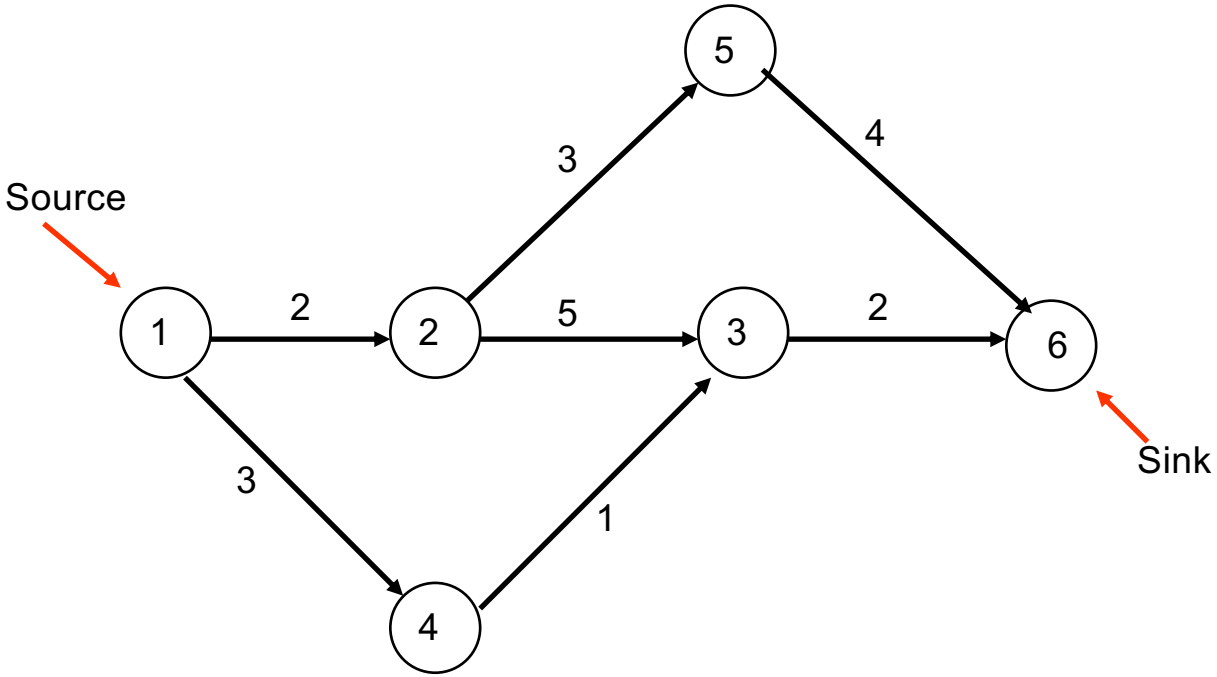
Definition of Flow Network

Represented by a connected weighted digraph with *n* vertices numbered from 1 to *n* with the following properties:

- contains exactly one vertex with no entering edges, called the ***source*** **(numbered 1)**
- contains exactly one vertex with no leaving edges, called the ***sink*** **(numbered *n*)**
- has **positive** integer weight u_{ij} on each directed edge (*i, j*), called the ***edge capacity***, indicating the upper bound on the amount of the material that can be sent from *i* to *j* through this edge

A digraph satisfying these properties is called a ***flow network*** or simply a ***network***

Example of Flow Network



Definition of a Flow

A *flow* is an assignment of real numbers x_{ij} to edges (i, j) of a given network that satisfy the following:

- ***flow-conservation requirements***
  The total amount of material entering an intermediate vertex must be equal to the total amount of the material leaving the vertex

$$\sum_{j:\,(j,\,i)\,\in\,E} x_{ji} \;=\; \sum_{j:\,(i,\,j)\,\in\,E} x_{ij} \quad \text{for } i = 2,3,\ldots,n\text{-}1$$

- ***capacity constraints***

$$0 \le x_{ij} \le u_{ij} \quad \text{for every edge } (i, j) \in E$$

Flow value and Maximum Flow Problem

Since no material can be lost or added to by going through intermediate vertices of the network, the total amount of the material leaving the source must end up at the sink:

$$\sum_{j:\,(1,j)\,\in\,E} x_{1j} \;=\; \sum_{j:\,(j,n)\,\in\,E} x_{jn}$$

The ***value*** of the flow is defined as the total outflow from the source (= the total inflow into the sink).

The ***maximum flow problem*** is to find a flow of the largest value (maximum flow) for a given network.

Maximum-Flow Problem as LP problem

Maximize $v = \sum_{j:\,(1,j) \in E} x_{1j}$

subject to

$$\sum_{j:\,(j,i) \in E} x_{ji} - \sum_{j:\,(i,j) \in E} x_{ij} = 0 \quad \text{for } i = 2, 3, \ldots, n\text{-}1$$

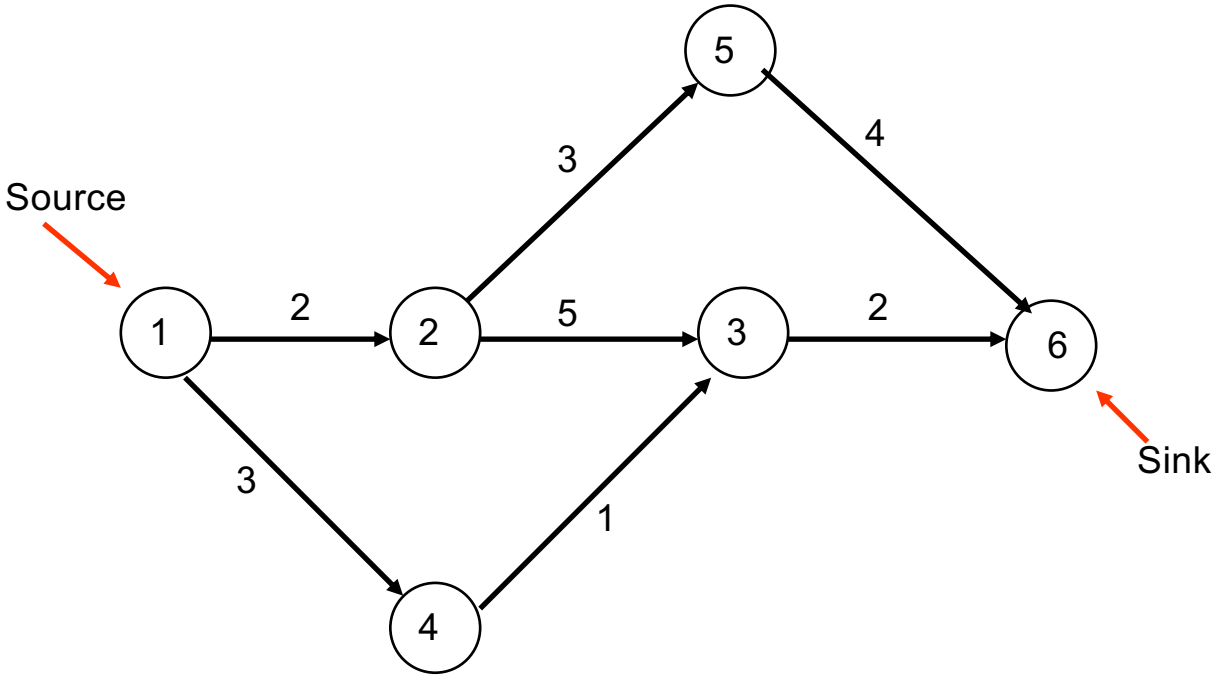
$$0 \le x_{ij} \le u_{ij} \quad \text{for every edge } (i,j) \in E$$

Augmenting Path (Ford-Fulkerson) Method

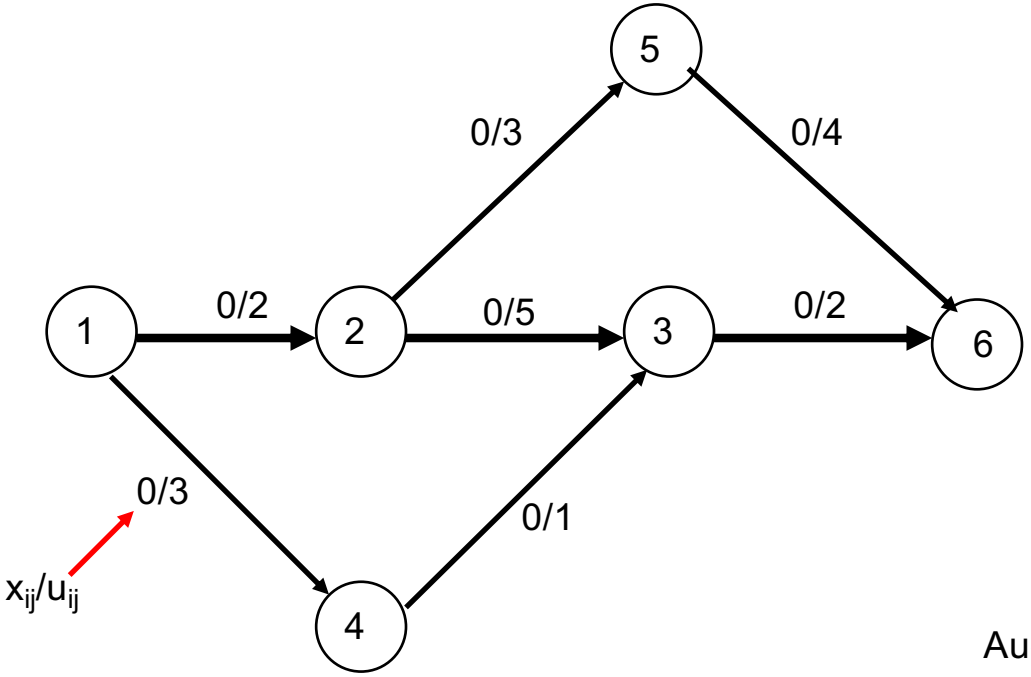
Employ iterative improvement idea as follows

- Start with the zero flow ($x_{ij} = 0$ for every edge)
- On each iteration, try to find a ***flow-augmenting path*** from source to sink, a path along which some additional flow can be sent
- If a flow-augmenting path is found, adjust the flow along the edges of this path to get a flow of increased value and try again
- If no flow-augmenting path is found, the current flow is maximum

Example 1

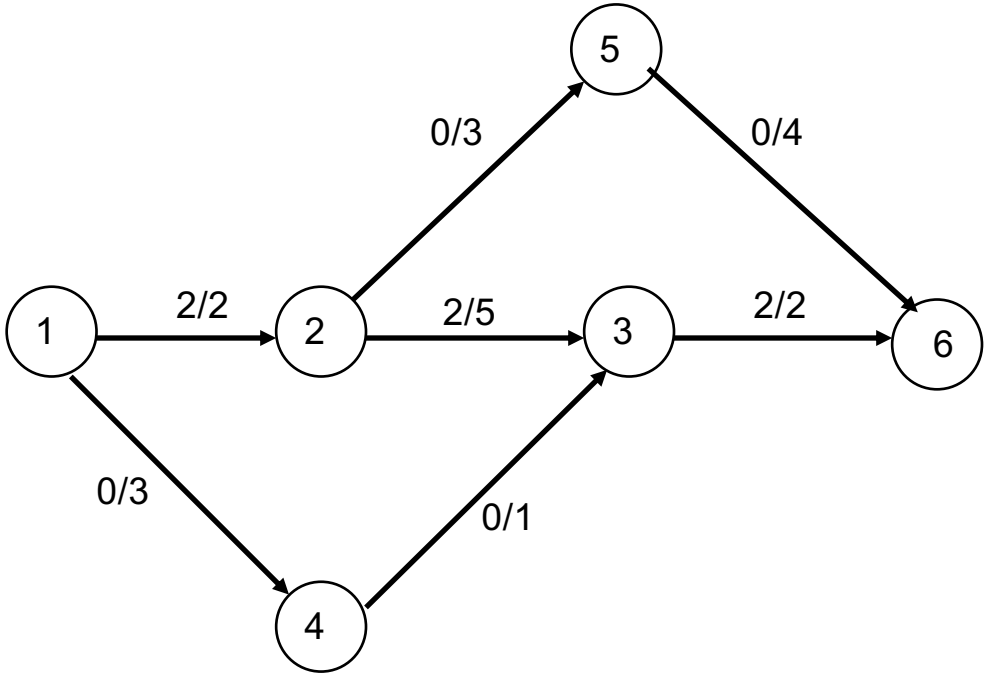


Example 1

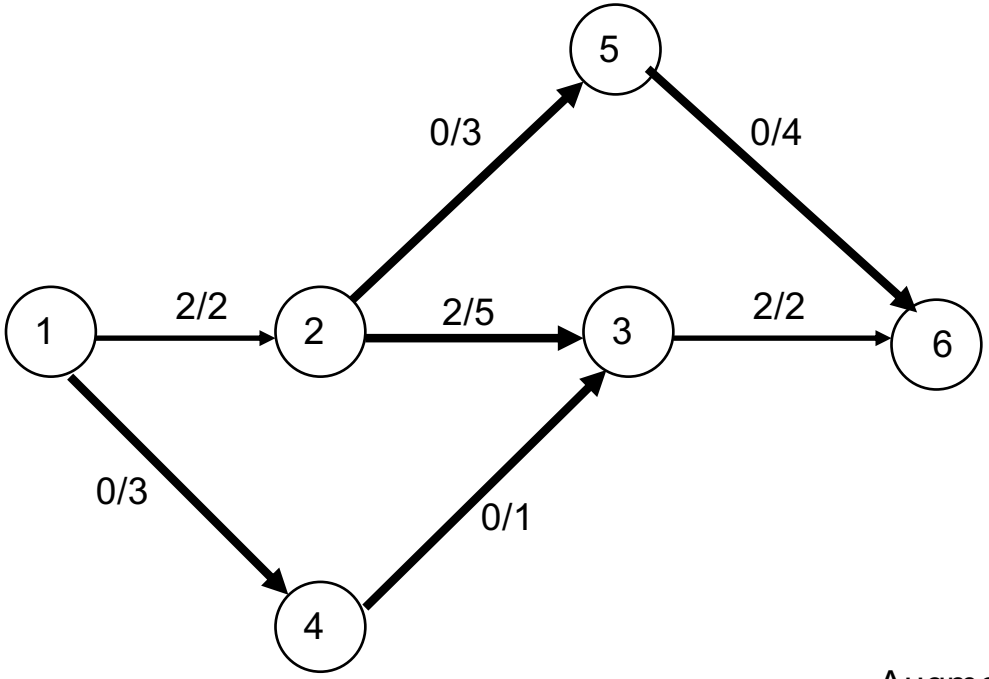


Augmenting path:
1→2 →3 →6

Example 1

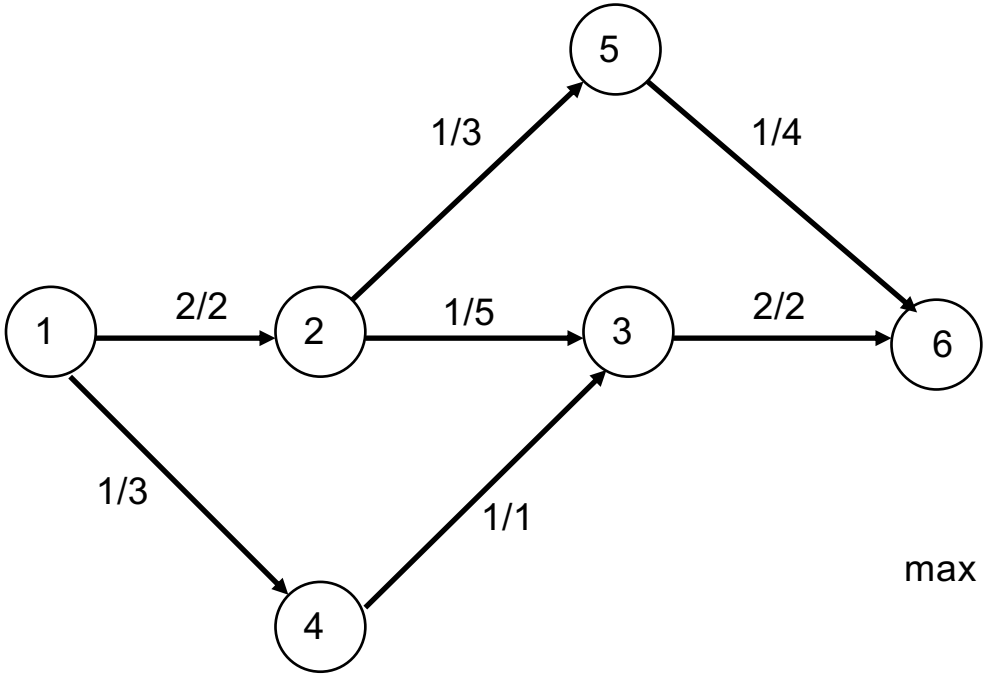


Example 1 (cont.)



Augmenting path:
1 →4 →3←2 →5 →6

Example 1 (maximum flow)



max flow value = 3

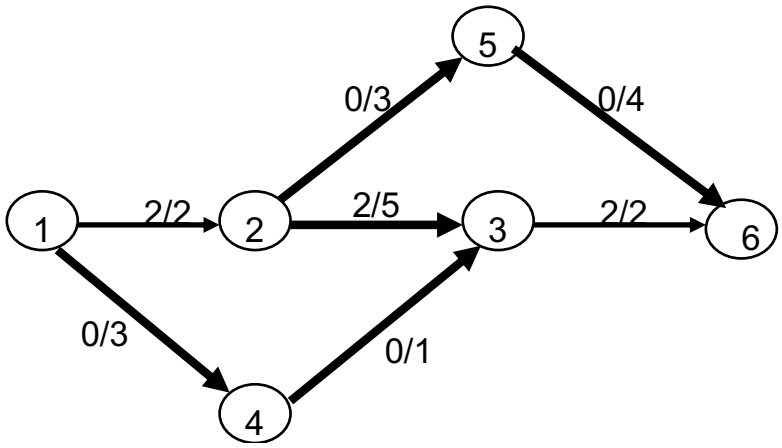
Finding a flow-augmenting path

Consider paths from source to sink in the underlying undirected graph in which any two consecutive vertices *i*, *j* are either:

- connected by a directed edge (*i* to *j*) with some positive unused capacity $r_{ij} = u_{ij} - x_{ij}$
  - known as *forward edge* (→)

OR

- connected by a directed edge (*j* to *i*) with positive flow x_{ji}
  - known as *backward edge* (←)



Finding a flow-augmenting path

Consider paths from source to sink in the underlying undirected graph in which any two consecutive vertices i, j are either:

- connected by a directed edge (i to j) with some positive unused capacity $r_{ij} = u_{ij} - x_{ij}$
  - known as *forward edge* (→)

OR

- connected by a directed edge (j to i) with positive flow x_{ji}
  - known as *backward edge* (←)

If a flow-augmenting path is found, the current flow can be increased by r units by increasing x_{ij} by r on each forward edge and decreasing x_{ji} by r on each backward edge, where

r = min {r_{ij} on all forward edges, x_{ji} on all backward edges}

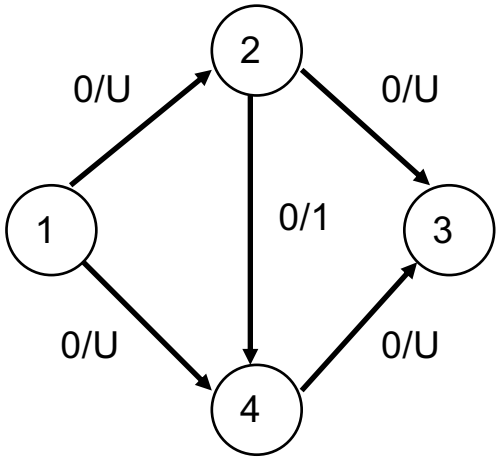
Finding a flow-augmenting path (cont.)

- Assuming the edge capacities are integers, *r* is a positive integer
- On each iteration, the flow value increases by at least 1
- Maximum value is bounded by the sum of the capacities of the edges leaving the source; hence the augmenting-path method has to stop after a finite number of iterations
- The final flow is always maximum, its value doesn't depend on a sequence of augmenting paths used

Performance degeneration of the method

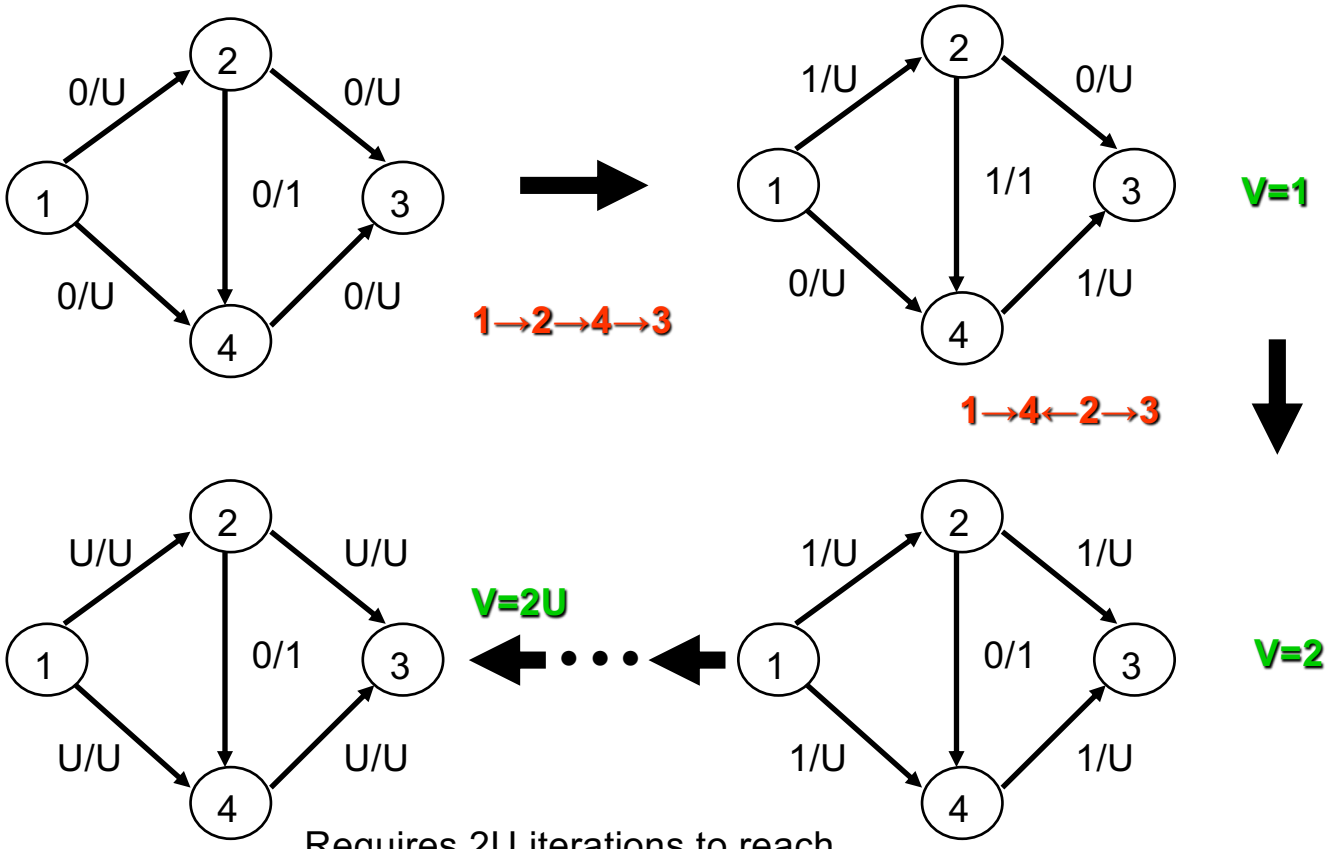
- The augmenting-path method doesn't prescribe a specific way for generating flow-augmenting paths
- Selecting a bad sequence of augmenting paths could impact the method's efficiency

Example 2



U = large positive integer

Example 2 (cont.)



Requires 2U iterations to reach maximum flow of value 2U

Shortest-Augmenting-Path Algorithm (Dinic's algorithm, Edmonds–Karp algorithm)

- Generate augmenting path with the least number of edges by **BFS** as follows.
- Starting at the source, perform BFS traversal by marking new (unlabeled) vertices with two labels:
  - **first label** – indicates the amount of additional flow that can be brought from the source to the vertex being labeled
  - **second label** – indicates the vertex from which the vertex being labeled was reached, with "+" or "–" added to the second label to indicate whether the vertex was reached via a forward or backward edge

Vertex labeling

- The **source** is always labeled with **∞,-**

- All other vertices are labeled as follows:
  - If unlabeled vertex *j* is connected to the front vertex *i* of the traversal queue by a directed edge from *i* to *j* with positive unused capacity $r_{ij} = u_{ij} - x_{ij}$ (**forward edge**), vertex *j* is labeled with $\boldsymbol{l_j, i^+}$, where $l_j = \min\{l_i, r_{ij}\}$

  - If unlabeled vertex *j* is connected to the front vertex *i* of the traversal queue by a directed edge from *j* to *i* with positive flow x_{ji} (**backward edge**), vertex *j* is labeled $\boldsymbol{l_j, i^-}$, where $l_j = \min\{l_i, x_{ji}\}$

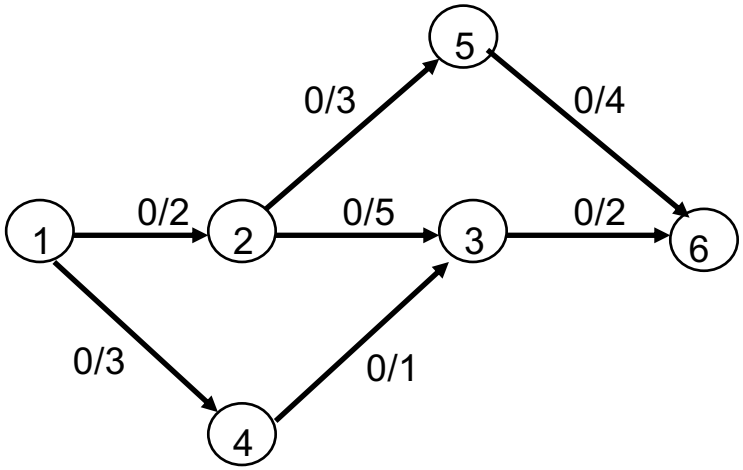
Vertex labeling (cont.)

- If the **sink** ends up being labeled, the current flow can be augmented by the amount indicated by the sink's first label

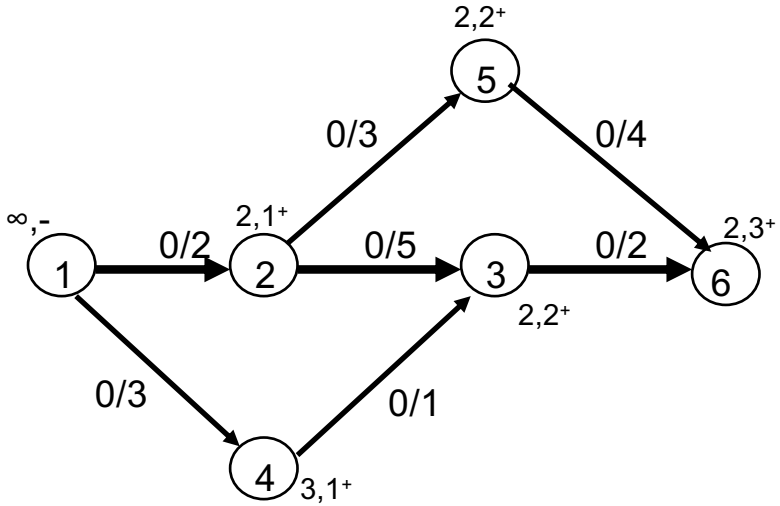
- The augmentation of the current flow is performed along the augmenting path traced by following the vertex second labels from sink to source; the current flow quantities are increased on the forward edges and decreased on the backward edges of this path

- If the sink remains unlabeled after the traversal queue becomes empty, the algorithm returns the current flow as maximum and stops

Example: Shortest-Augmenting-Path Algorithm

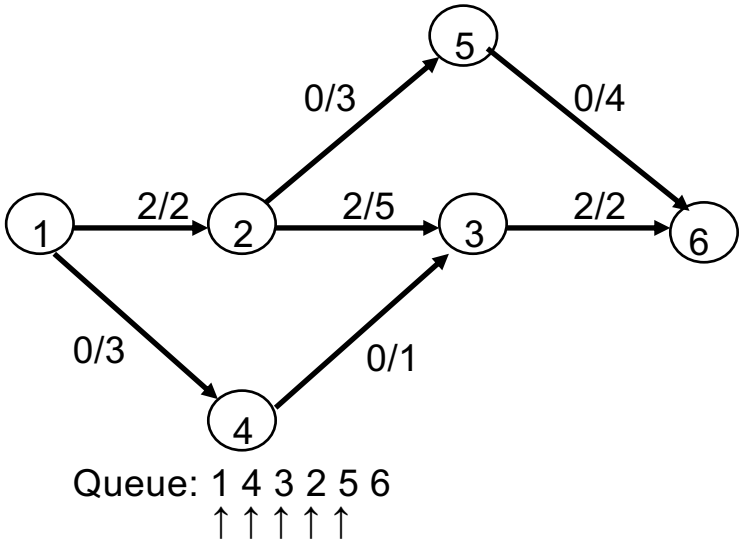


Queue: 1 2 4 3 5 6
↑ ↑ ↑ ↑

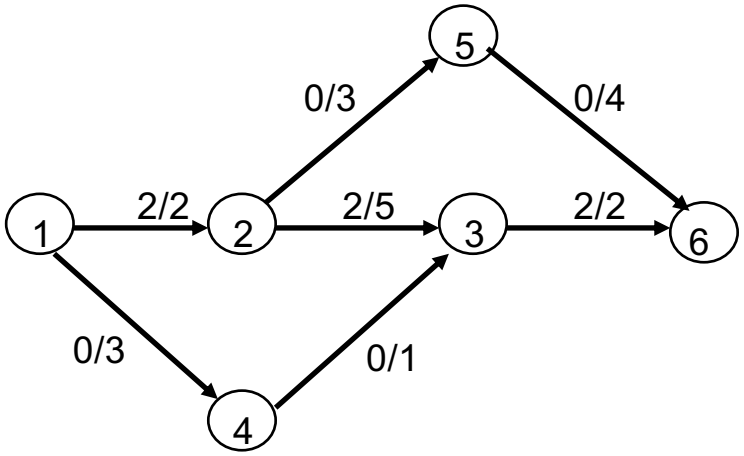


Augment the flow by 2 (the sink's first label) along the path 1→2→3→6

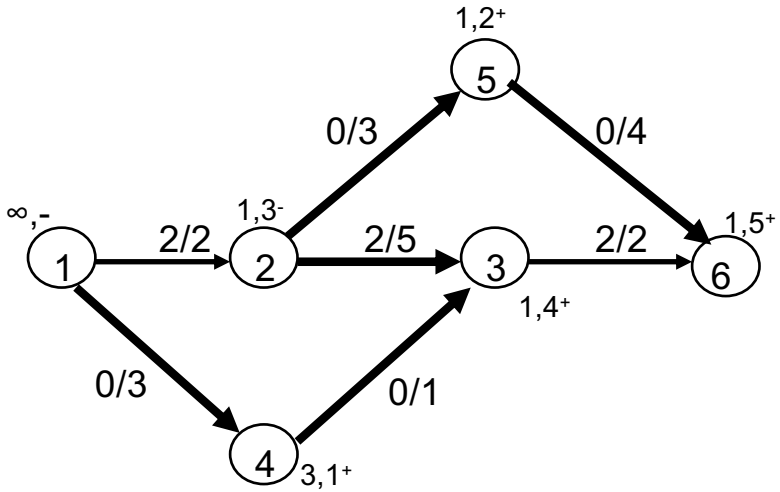
Example (cont.)



Example (cont.)

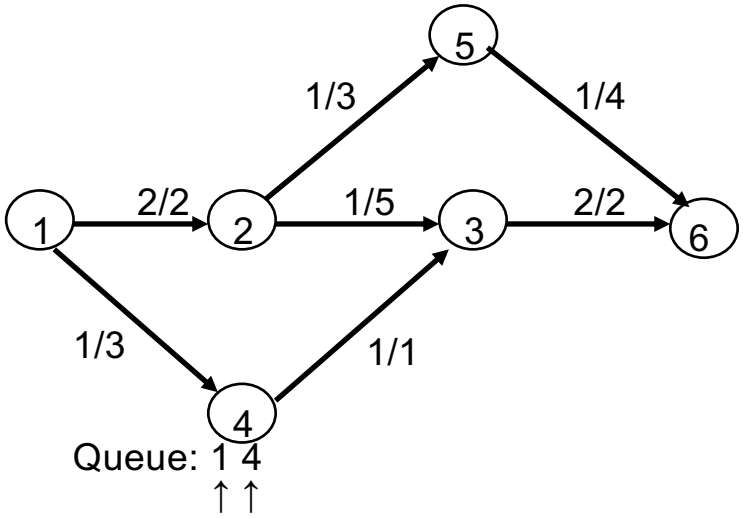


Queue: 1 4 3 2 5 6
↑ ↑ ↑ ↑ ↑

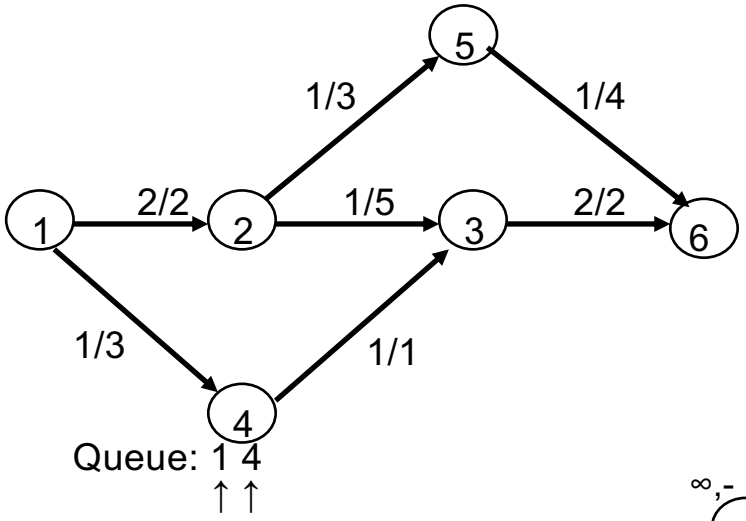


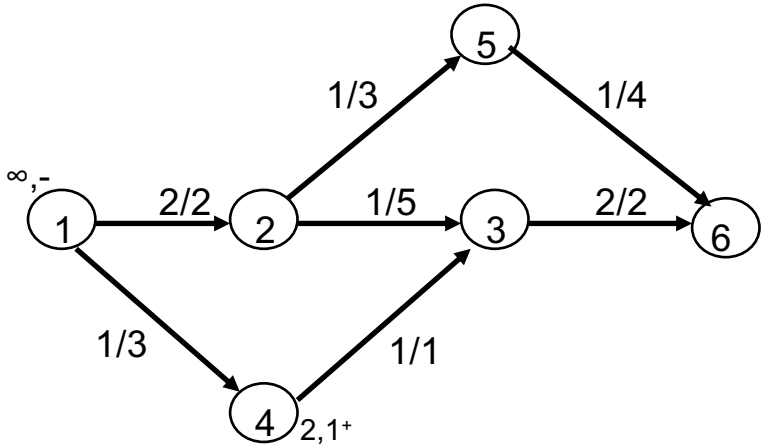
Augment the flow by 1 (the sink's first label) along the path 1→4→3←2→5→6

Example (cont.)



Example (cont.)





No augmenting path (the sink is unlabeled)
the current flow is maximum

Shortest-augmenting-path algorithm

Input: A network with single source 1, single sink n, and
positive integer capacities u_{ij} on its edges (i, j)

Output: A maximum flow x

assign $x_{ij} = 0$ to every edge (i, j) in the network

label the source with $\infty, -$ and add the source to the empty queue Q

while not *Empty*(Q) **do**

    $i \leftarrow$ *Front*(Q); *Dequeue*(Q)

    for every edge from i to j **do** //forward edges

        if j is unlabeled

            $r_{ij} \leftarrow u_{ij} - x_{ij}$

            if $r_{ij} > 0$

                $l_j \leftarrow \min\{l_i, r_{ij}\}$; label j with l_j, i^+

                Enqueue(Q, j)

    for every edge from j to i **do** //backward edges

        if j is unlabeled

            if $x_{ji} > 0$

                $l_j \leftarrow \min\{l_i, x_{ji}\}$; label j with l_j, i^-

                Enqueue(Q, j)

    if the sink has been labeled

        //augment along the augmenting path found

        $j \leftarrow n$ //start at the sink and move backwards using second labels

        while $j \neq 1$ //the source hasn't been reached

            if the second label of vertex j is i^+

                $x_{ij} \leftarrow x_{ij} + l_n$

            else //the second label of vertex j is i^-

                $x_{ji} \leftarrow x_{ji} - l_n$

            $j \leftarrow i$

        erase all vertex labels except the ones of the source

        reinitialize Q with the source

return x //the current flow is maximum

Time Efficiency

- The number of augmenting paths needed by the shortest-augmenting-path algorithm never exceeds $nm/2$, where n and m are the number of vertices and edges, respectively
- Since the time required to find shortest augmenting path by breadth-first search is in $O(n+m)=O(m)$ for networks represented by their adjacency lists, the time efficiency of the shortest-augmenting-path algorithm is in $\mathbf{O(nm^2)}$ for this representation
- More efficient algorithms have been found that can run in close to $\mathbf{O(nm)}$ time, but these algorithms don't fall into the iterative-improvement paradigm

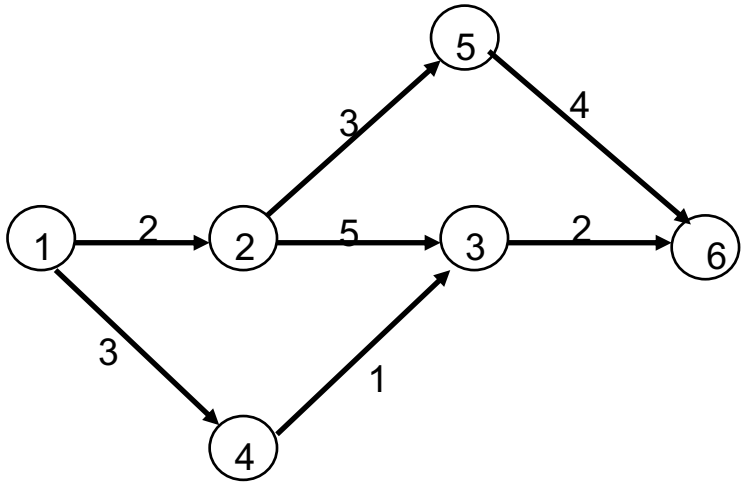
Definition of a Cut

Let X be a set of vertices in a network that includes its source but does not include its sink, and let $\overline{X}$, the complement of X, be the rest of the vertices including the sink. The ***cut*** induced by this partition of the vertices is the set of all the edges with a tail in $\overline{X}$ and a head in X.

Capacity of a cut is defined as the sum of capacities of the edges that compose the cut.

- We'll denote a cut and its capacity by $C(X,\overline{X})$ and $c(X,\overline{X})$

Examples of network cuts



If X = {1} and $\overline{X}$ = {2,3,4,5,6}, C(X,$\overline{X}$) = {(1,2), (1,4)}, c = 5

If X ={1,2,3,4,5} and $\overline{X}$ = {6}, C(X,$\overline{X}$) = {(3,6), (5,6)}, c = 6

If X = {1,2,4} and $\overline{X}$ ={3,5,6}, C(X,$\overline{X}$) = ??, c = ??

Minimum cut

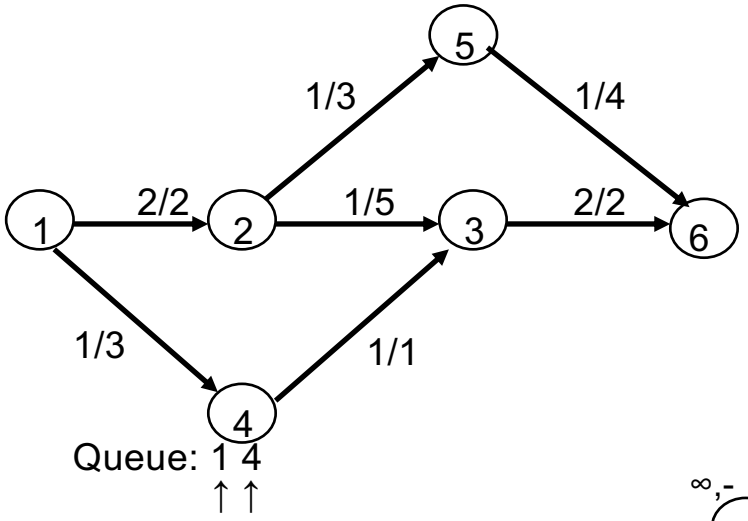
- ***Minimum cut*** is a cut of the smallest capacity in a given network

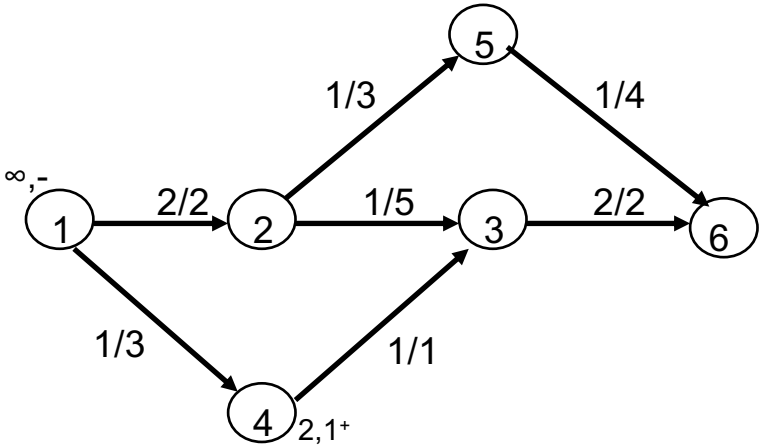
- **Max-Flow Min-Cut Theorem:** The value of maximum flow in a network is equal to the capacity of its minimum cut

Finding a minimum cut

- The shortest augmenting path algorithm yields both a maximum flow and a minimum cut:
  - maximum flow is the final flow produced by the algorithm
  - minimum cut is formed by all the edges from the labeled vertices to unlabeled vertices on the last iteration of the algorithm
  - all the edges from the labeled to unlabeled vertices are full, i.e., their flow amounts are equal to the edge capacities, while all the edges from the unlabeled to labeled vertices, if any, have zero flow amounts on them

Recaps of previous example





No augmenting path (the sink is unlabeled)
the current flow is maximum