THE MAXIMUM-FLOW PROBLEM

Maximum Flow Problem

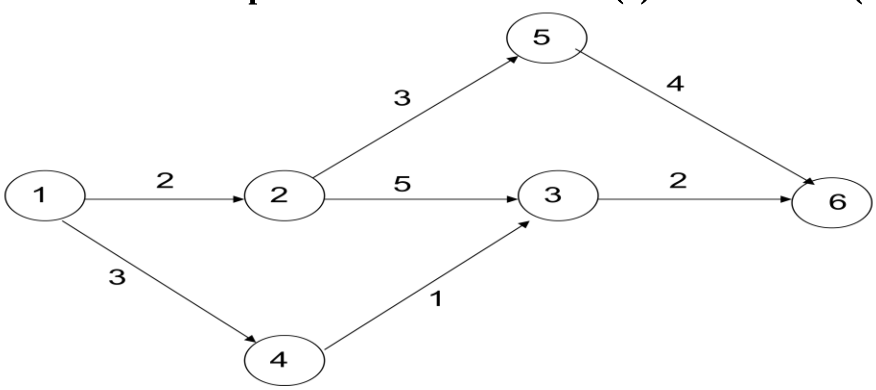
Problem of maximizing the flow of a material through a transportation network (e.g., pipeline system, communications or transportation networks)

Formally 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*** (numbered1)
- 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 an network.

Example of Flow Network Node (1) = source Node(6) = sin



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

- *capacity constraints*

$0 \le x_{ij} \le u_{ij}$ 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

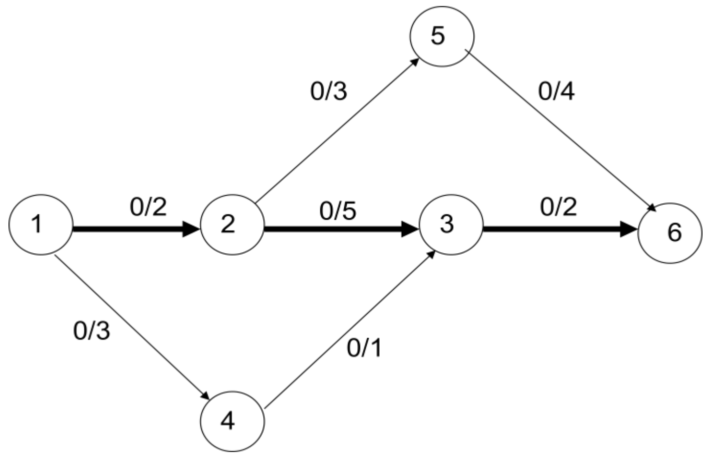
$$\sum_{j: (j,i) \in E} x_{ji} - \sum_{j: (i,j) \in E} x_{ij} = 0 \quad \text{for } i = 2, 3, \ldots, n-1$$

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

Augmenting Path (Ford-Fulkerson) Method

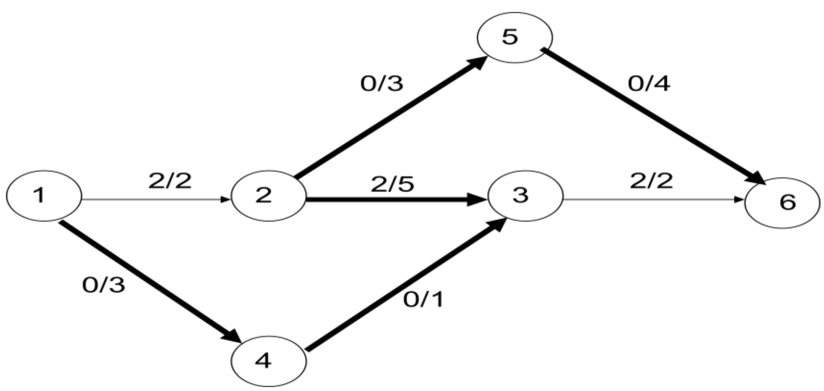
- Start with the zero flow ($x_{ij} = 0$ for every edge).
- On each iteration, try to find a *flow-augmenting path* from source to sink, which a path along which some additional flow can be sent.
- If a flow-augmenting path is found, adjust the flow along the edges of this path to get a flow of increased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

Example 1



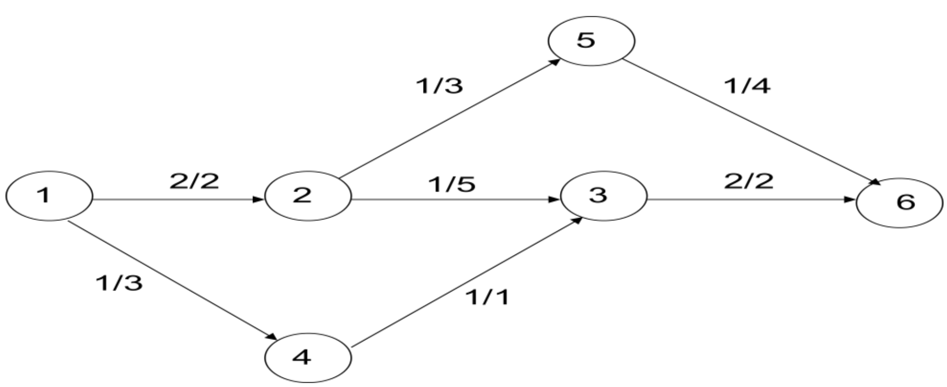
Augmenting path: 1‹2 ‹3 ‹6

x_{ij}/u_{ij}



Augmenting path: 1 ‹4 ‹3›2 ‹5 ‹6

Example 1 (maximum flow)



Finding a flow-augmenting path

To find a flow-augmenting path for a flow x, 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

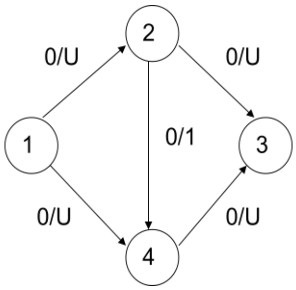
r = min {r_{ij} on all forward edges, x_{ji} on all backward edges}

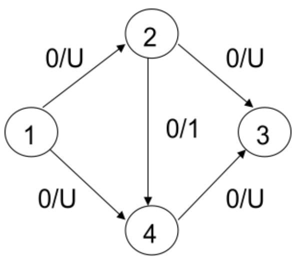
- Assuming the edge capacities are integers, *r* is a positive integer
- On each iteration, the flow value increases by at least1
- Maximum value is bounded by the sum of the capacities of the edges leaving the source; hence the augmenting-path method has to stop after a finite number of iterations
- The final flow is always maximum; its value doesn't depend on a sequence of augmenting paths used

Performance degeneration of the method

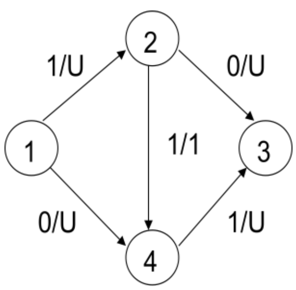
- The augmenting-path method doesn't prescribe a specific way for generating flow- augmenting paths
- Selecting a bad sequence of augmenting paths could impact the method's efficiency.

1.Example

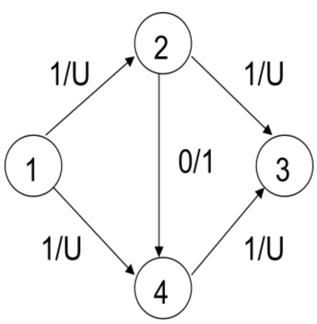




1→2→4→3

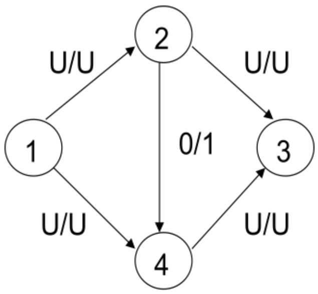


1→4←2→3 **V=1**



V=2

…



V=2U

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

Shortest-Augmenting-Path 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:

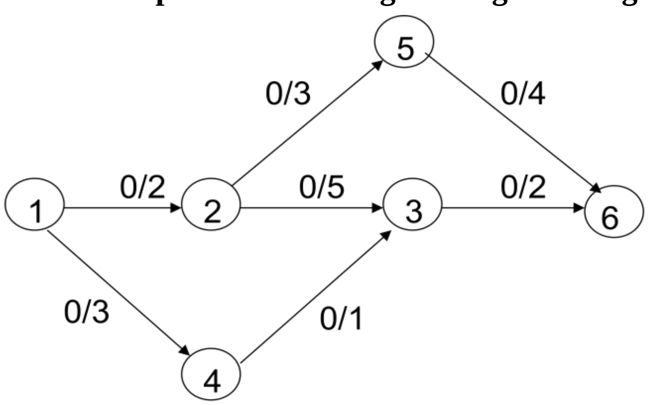
- first label – indicates the amount of additional flow that can be brought from the source to the vertex being labeled
- second label – indicates the vertex from which the vertex being labeled was reached, with "+" or "–" added to the second label to indicate whether the vertex was reached via a forward or backward edge

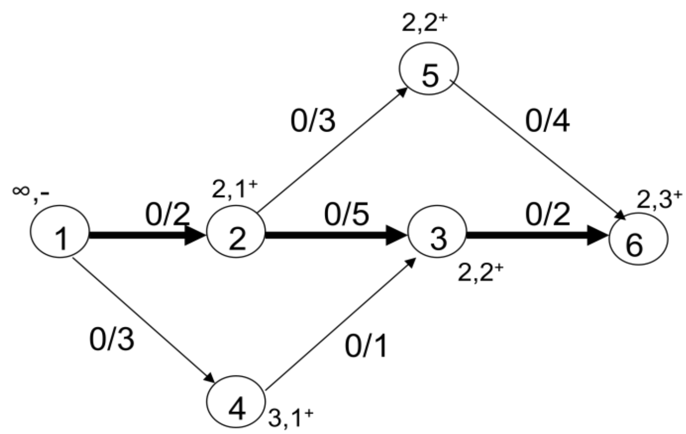
Vertex labeling

- 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 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 l_j, i^-, where $l_j = \min\{l_i, x_{ji}\}$

- 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

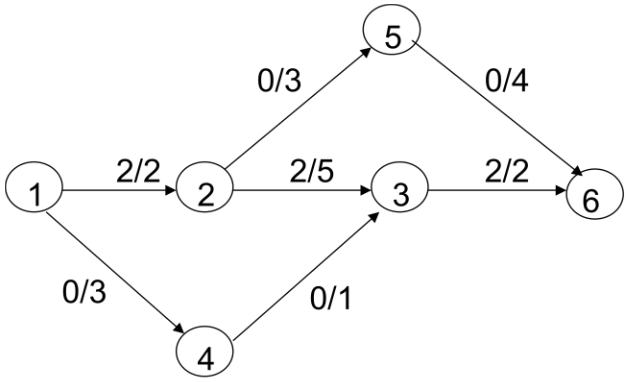


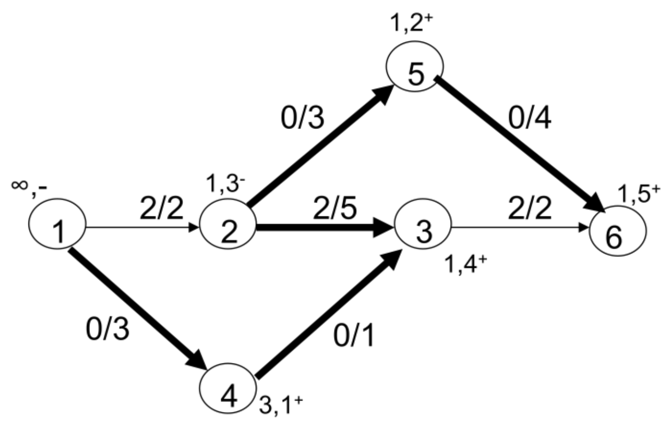


Queue: 1 2 4 3 5 6

‡ ‡ ‡ ‡

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

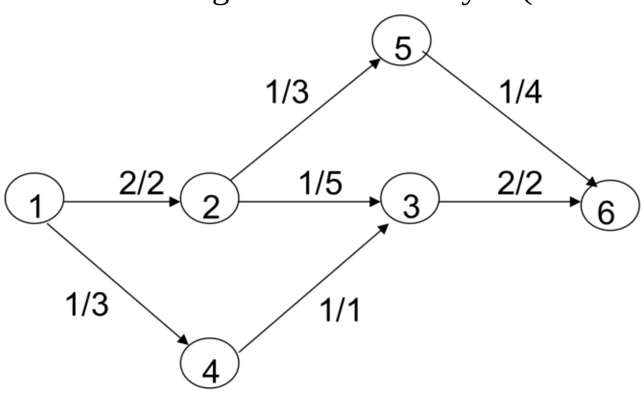


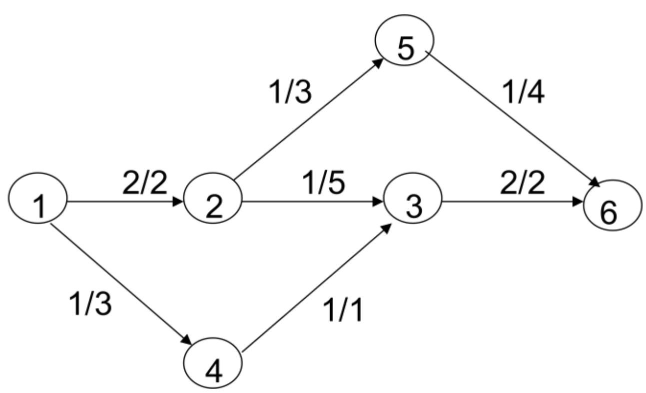


Queue: : 1 4 3 2 5 6

‡ ‡ ‡ ‡ ‡

Augment the flow by 1 (the sink’s first label) along the path 1‹4‹3›2‹5‹6





Queue: 14

‡‡

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

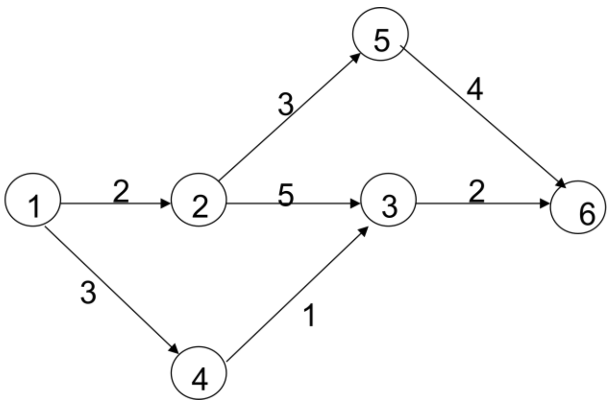
Definition of a Cut

Let X be a set of vertices in a network that includes its source but does not include its sink, and let 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 X and a head in X.

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

- →e'll denote a cut and its capacity by C(X,X) and c(X,X)
- Note that if all the edges of a cut were deleted from the network, there would be no directed path from source to sink
- *Minimum cut* is a cut of the smallest capacity in a given network

Examples of network cuts



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

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

If X = {1,2,4} and X ={3,5,6}, C(X,X) = {(2,3), (2,5), (4,3)}, c = 9

Max-Flow Min-Cut Theorem

1. The value of maximum flow in a network is equal to the capacity of its minimum cut
2. 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.

ALGORITHM *Shortest Augmenting Path(G)*

//Implements the shortest-augmenting-path algorithm

//Input: A network with single source 1, single sink *n*, and positive integer capacities *uij* on

// its edges *(i, j)*

//Output: A maximum flow *x*

assign *xij*= 0 to every edge *(i, j)*in the network

label the source with ∞, − and add the source to the empty queue *Q*

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

i ›*Front(Q)*; *Dequeue(Q)*

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

if *j* is unlabeled

r_{ij}›u_{ij}−x_{ij}

if r_{ij}>0

lj›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* isunlabeled

if x_{ji}>0

l_j›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 ›*n* //start at the sink and move backwards using second labels
while *j* ≠ 1 //the source hasn't been reached
if the second label of
vertex *j*is*i*+
x_{ij}›x_{ij}+l_n
else //the second label of
vertex *j* is *i*– x_{ij}›x_{ij}–l_n
j ›*i*; *i* ›the vertex indicated by *i*'s
second label 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 $O(nm^2)$ for this representation.
- More efficient algorithms have been found that can run in close to O(*nm*) time, but these algorithms don't fall into the iterative-improvement paradigm.