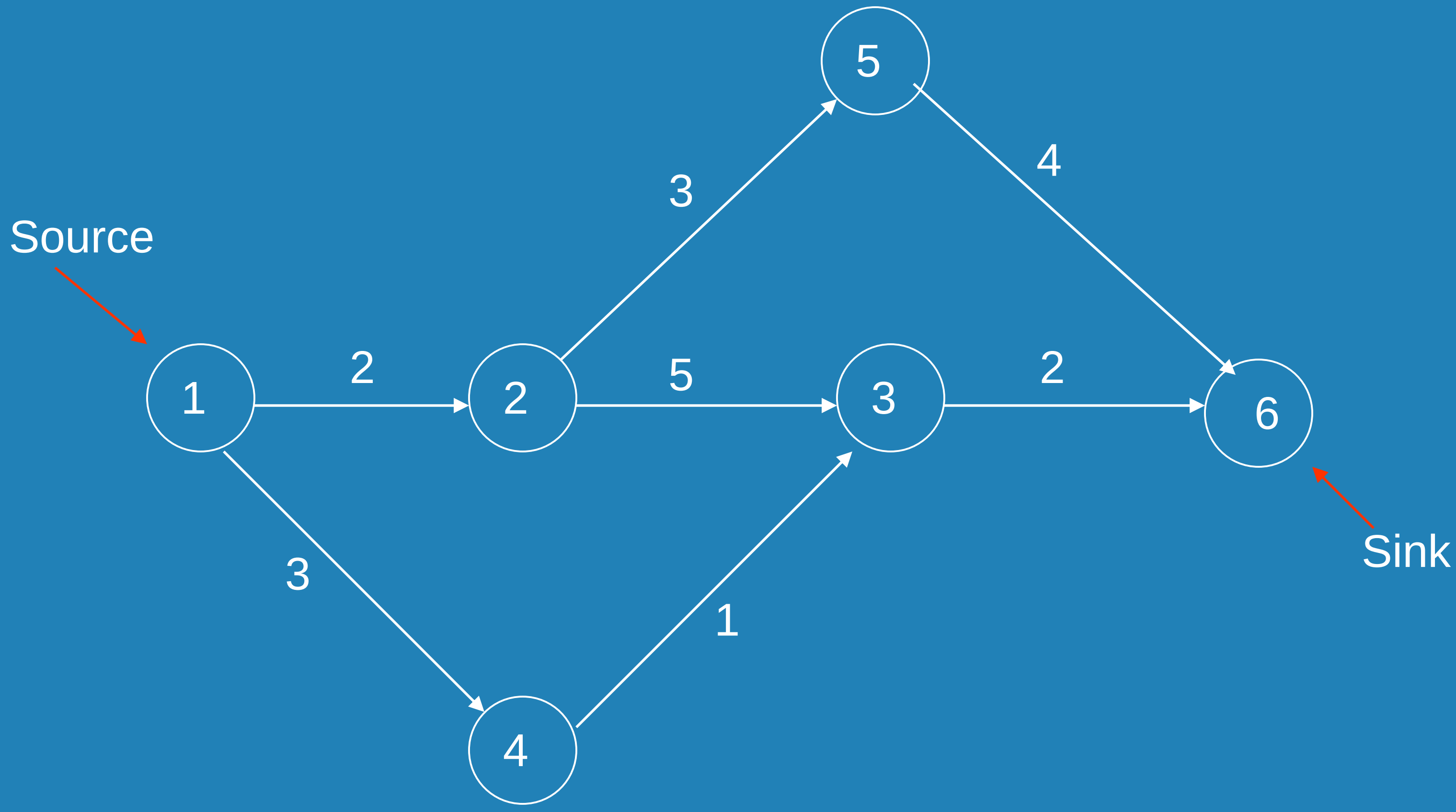
Maximum Flow Problem

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* (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

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

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

- *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:

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

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

- **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**

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

In order to implement and analyze the Ford-Fulkerson method, we need to introduce several additional concepts.

Residual flow

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

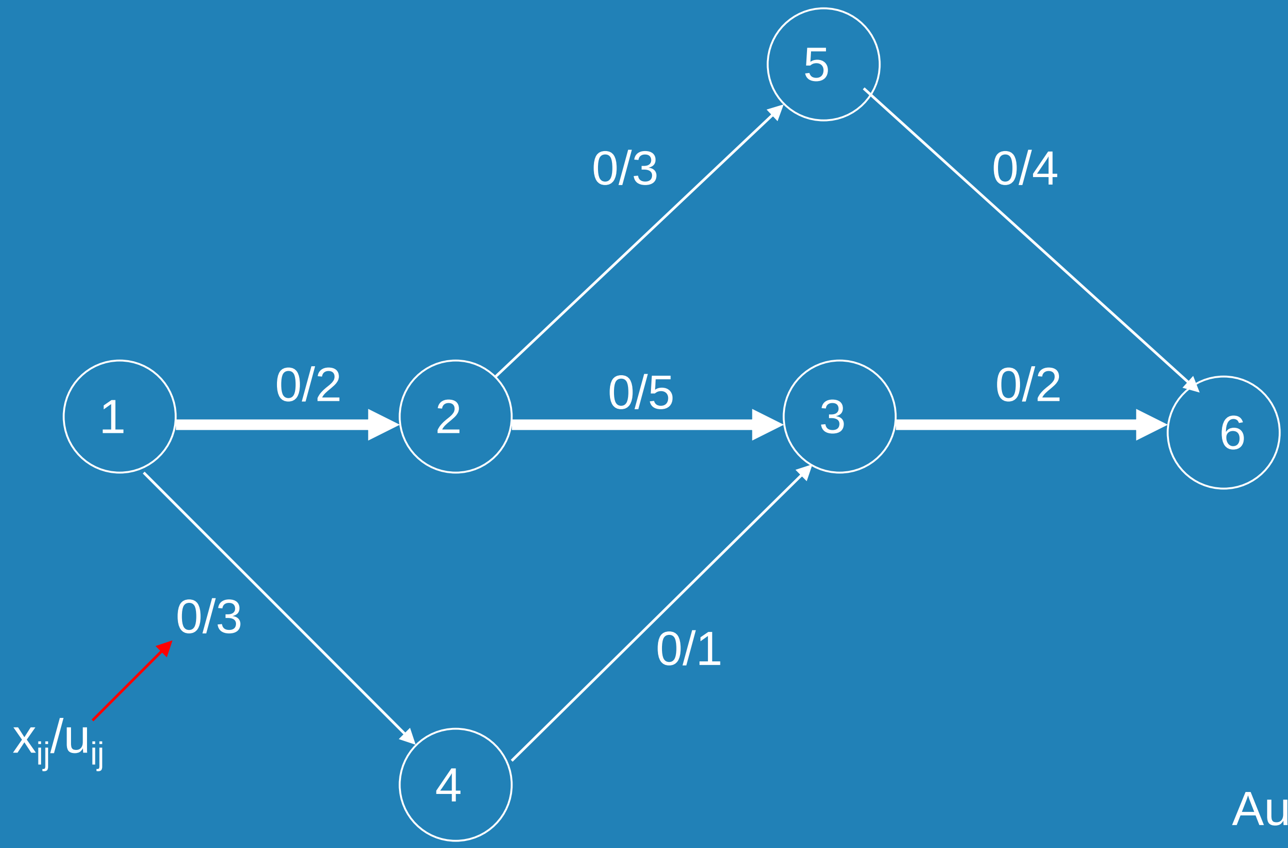
These reverse edges in the residual network allow an algorithm to send back flow it has already sent along an edge. Sending flow back along an edge is equivalent to decreasing the flow on the edge

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

Augmenting path

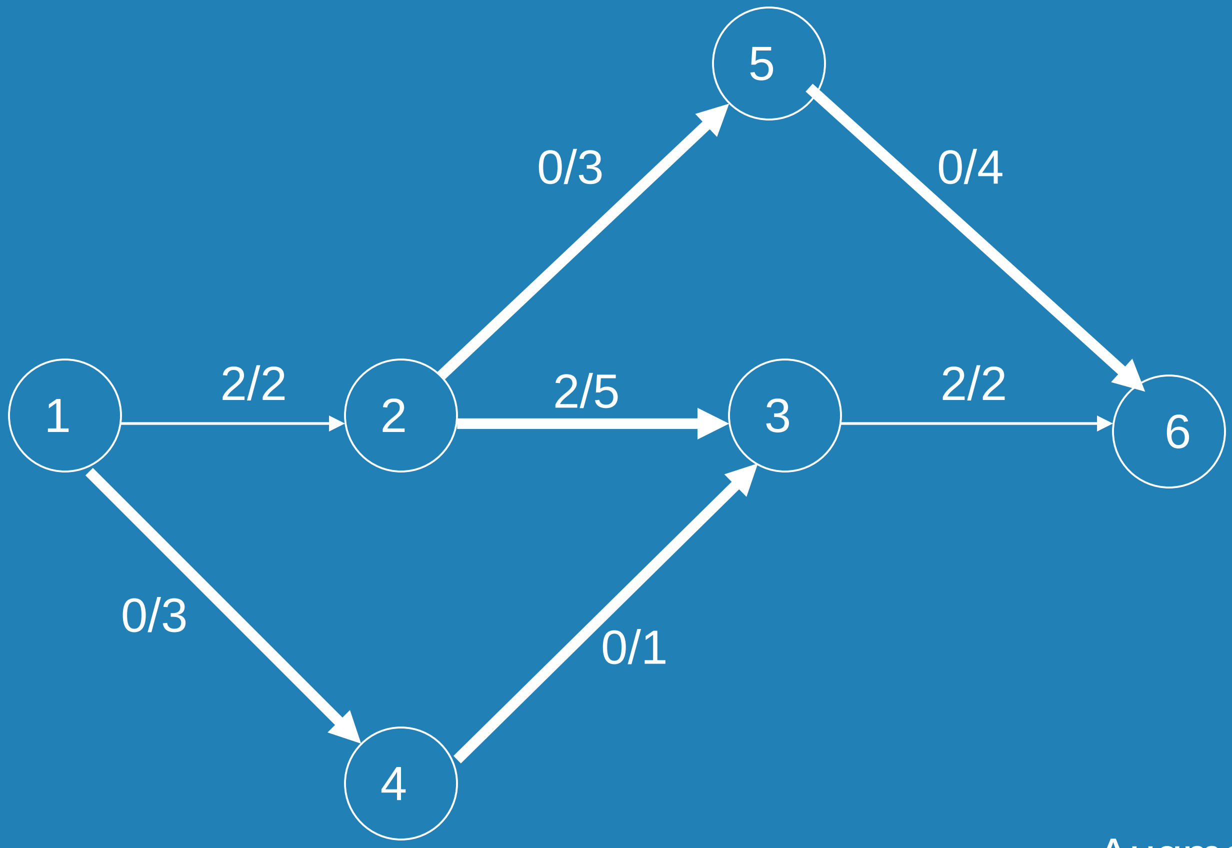
Given a flow network $G = (V, E)$ and a flow f, an ***augmenting path*** p is a simple path from s to t in the residual network G_f. By the definition of the residual network, we may increase the flow on an edge (u, v) of an augmenting path by up to $c_f(u, v)$ without violating the capacity constraint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

Example 1



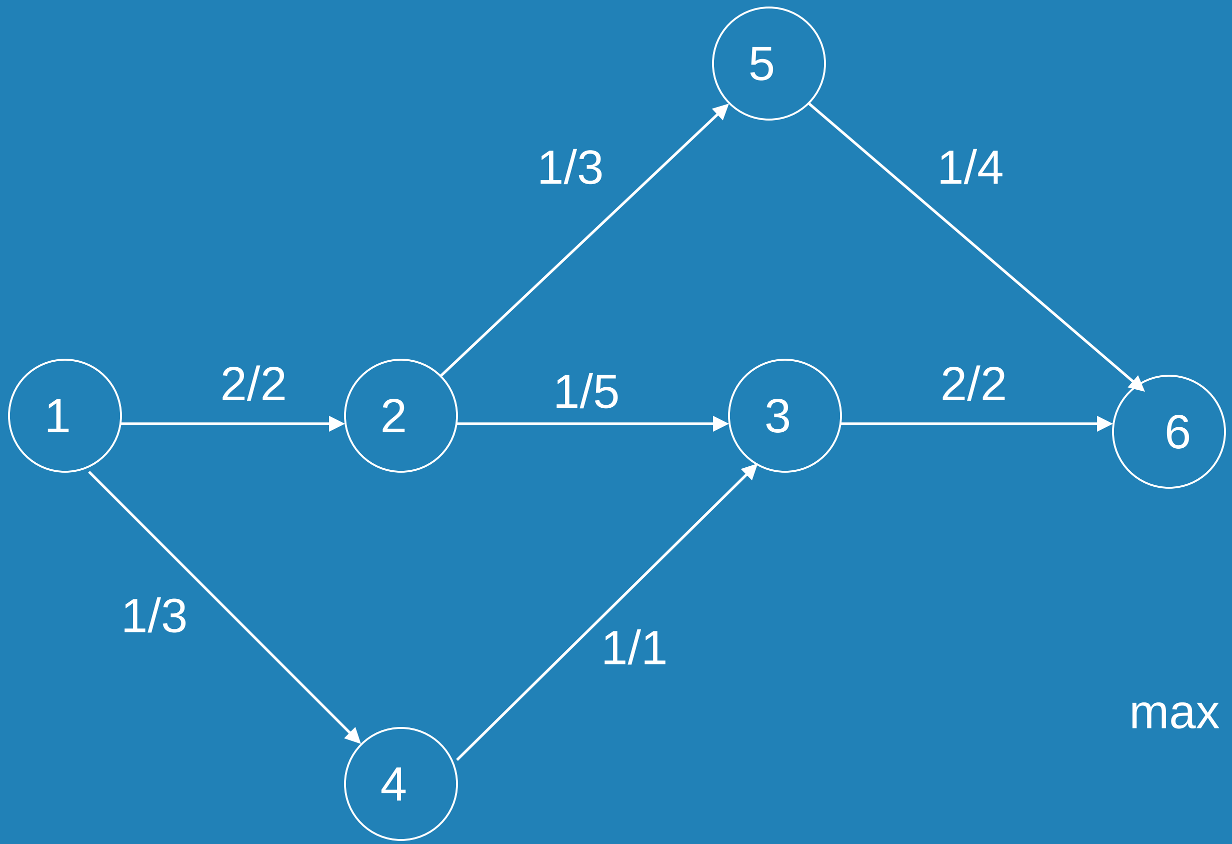
Augmenting path:
1→2 →3 →6

Example 1 (cont.)



Augmenting path:
1 →4 →3←2 →5 →6

Example 1 (maximum flow)



max flow value = 3

Finding a flow-augmenting path

To find a flow-augmenting path for a flow x, consider paths from source to sink in the underlying <u>undirected</u> 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

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

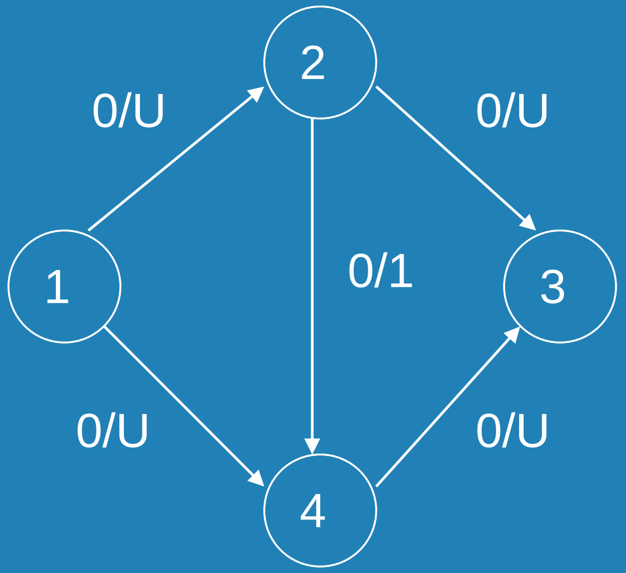
Finding a flow-augmenting path (cont.)

- 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

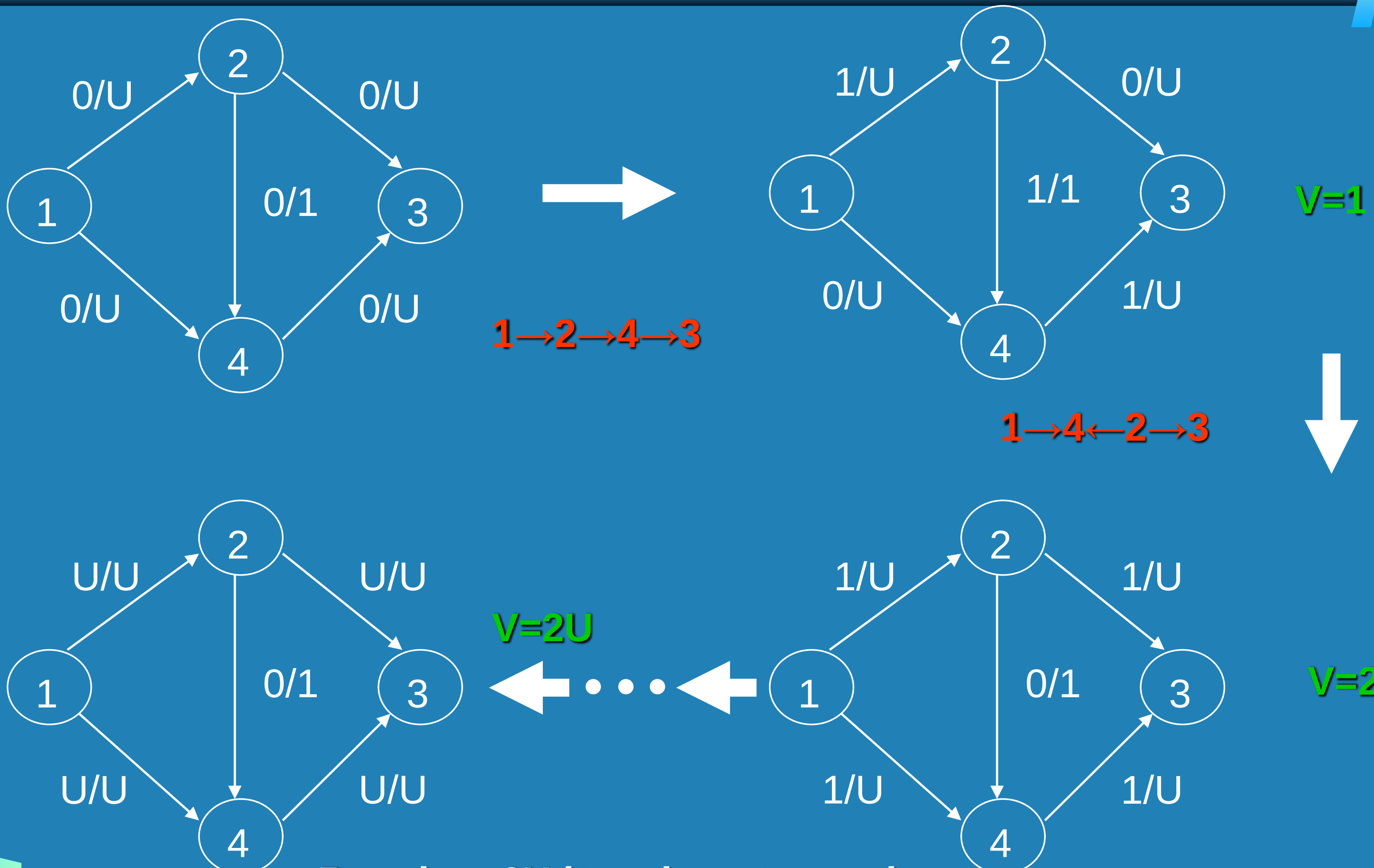
- The augmenting-path method doesn't prescribe a specific way for generating flow-augmenting paths
- Selecting a bad sequence of augmenting paths could impact the method's efficiency

Example 2



U = large positive integer

Example 2 (cont.)



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

Shortest-Augmenting-Path Algorithm

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

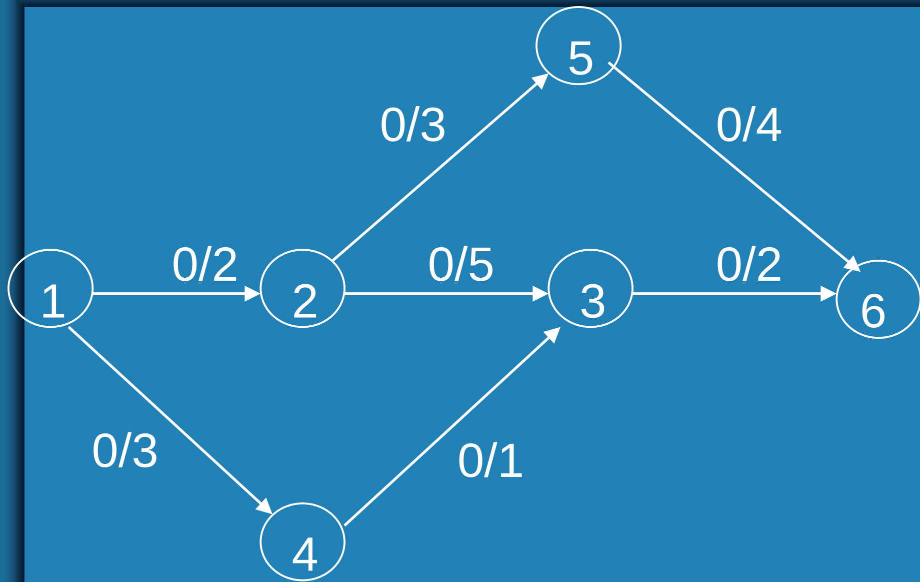
Vertex labeling

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

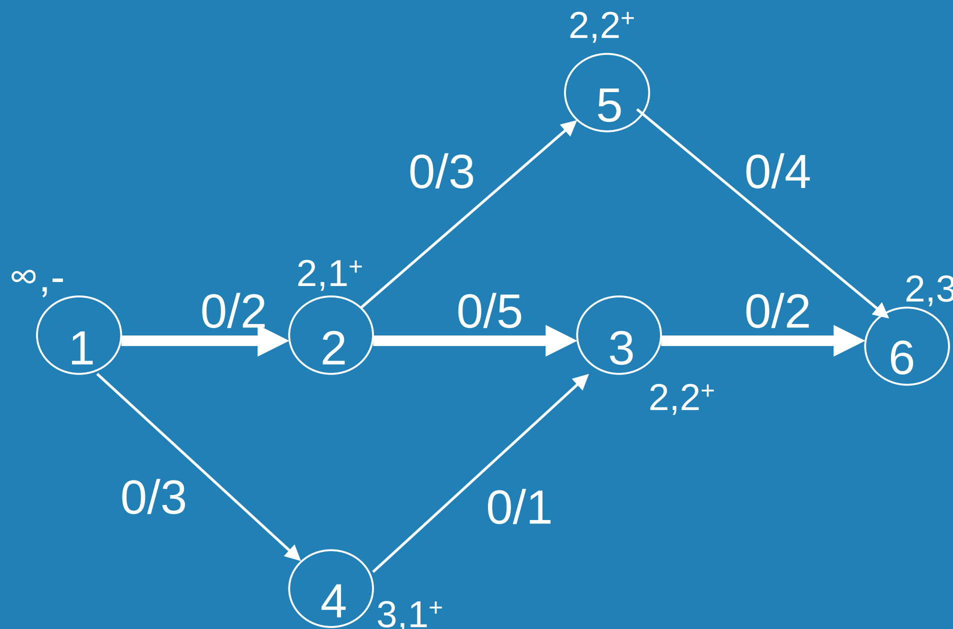
Vertex labeling (cont.)

- 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

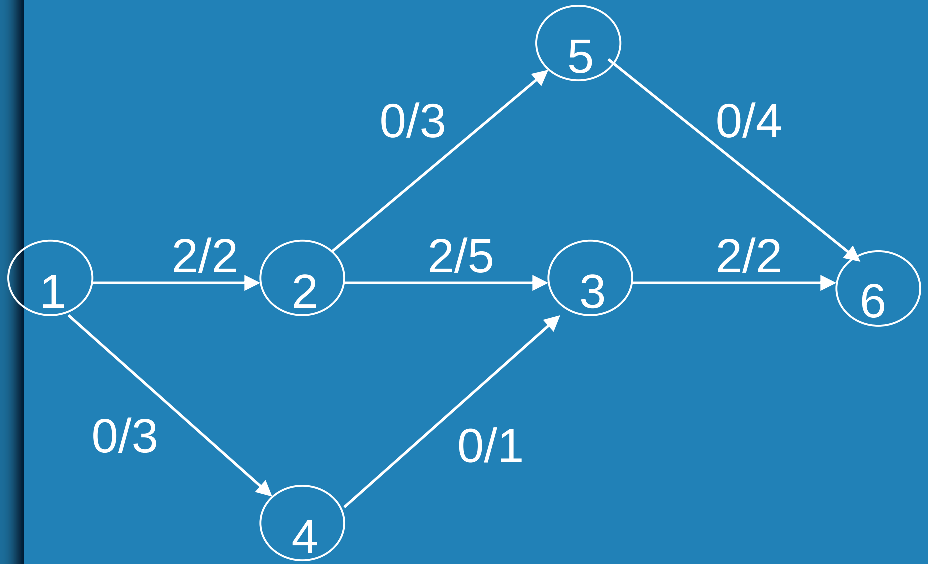


Queue: 1 2 4 3 5 6
↑ ↑ ↑ ↑

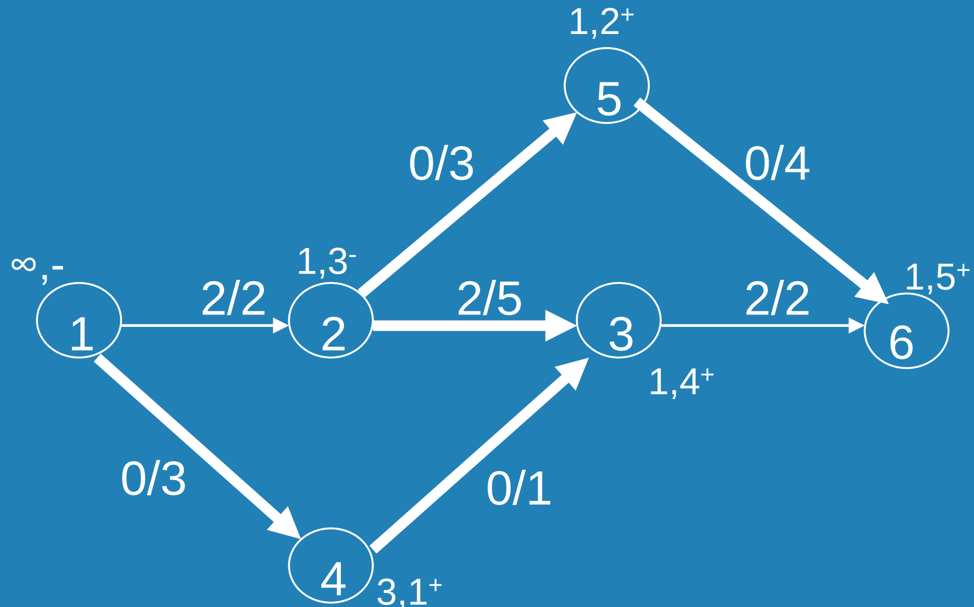


Augment the flow by 2 (the sink's first label) along the path 1→2→3→6

Example (cont.)

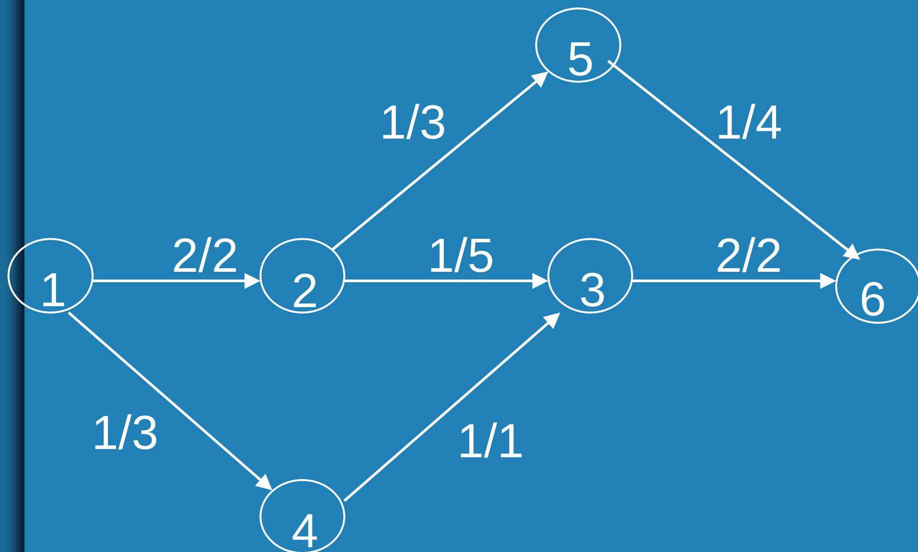


Queue: 1 4 3 2 5 6
↑ ↑ ↑ ↑ ↑

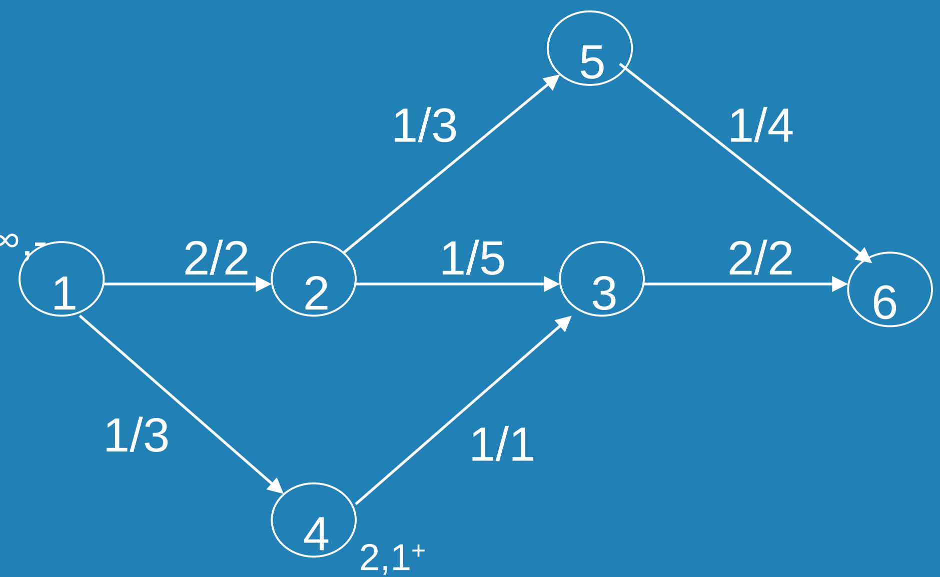


Augment the flow by 1 (the sink's first label) along the path 1→4→3←2→5→6

Example (cont.)



Queue: 1 4
↑ ↑



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

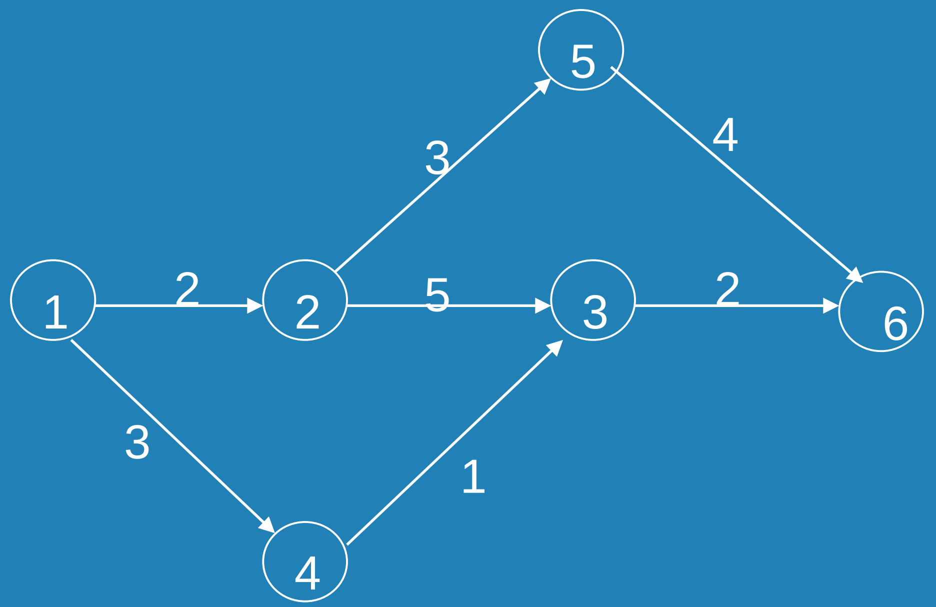
Definition of a Cut

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

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

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 $\overline{X} =\{3,5,6\}$, $C(X,\overline{X}) = \{(2,3), (2,5), (4,3)\}$, $c = 9$

Max-Flow Min-Cut Theorem

- The value of maximum flow in a network is equal to the capacity of its 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

Theorem 26.6 (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.

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 ∞, − 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
                l_j ← 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 ← 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_ji ← x_ji − l_n
            j ← 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

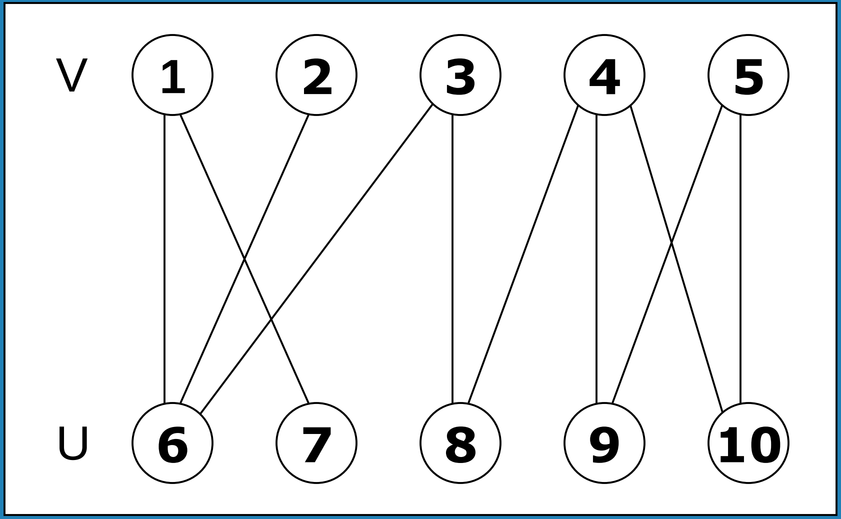
- The number of augmenting paths needed by the shortest-augmenting-path algorithm never exceeds $nm/2$, where n and m are the number of vertices and edges, respectively

- 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 $O(nm^2)$ 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

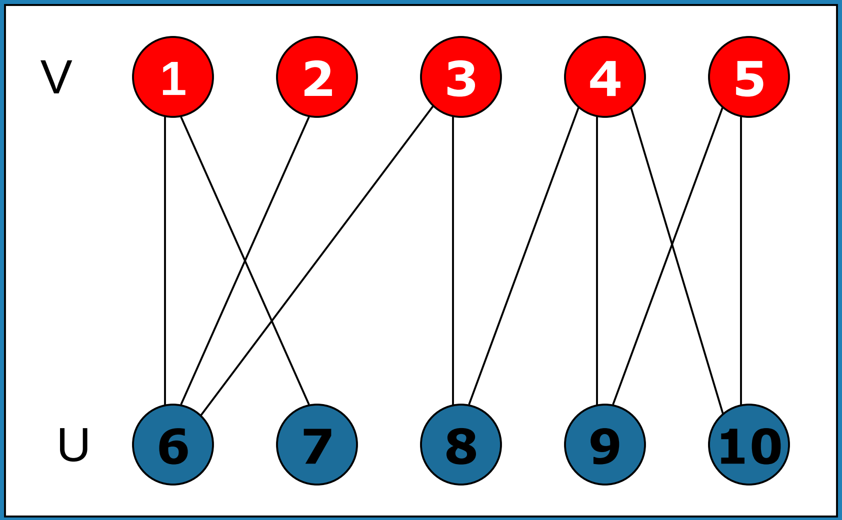
Bipartite Graphs

- *Bipartite graph*: a graph whose vertices can be partitioned into two disjoint sets V and U, not necessarily of the same size, so that every edge connects a vertex in V to a vertex in U
- A graph is bipartite if and only if it does not have a cycle of an odd length



Bipartite Graphs (cont.)

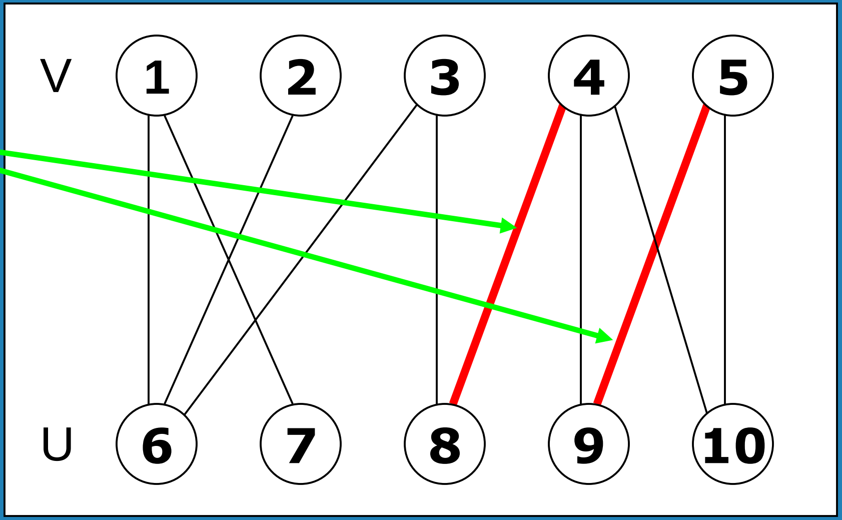
A bipartite graph is *2-colorable*: the vertices can be colored in two colors so that every edge has its vertices colored differently



Matching in a Graph

A *matching* in a graph is a subset of its edges with the property that no two edges share a vertex

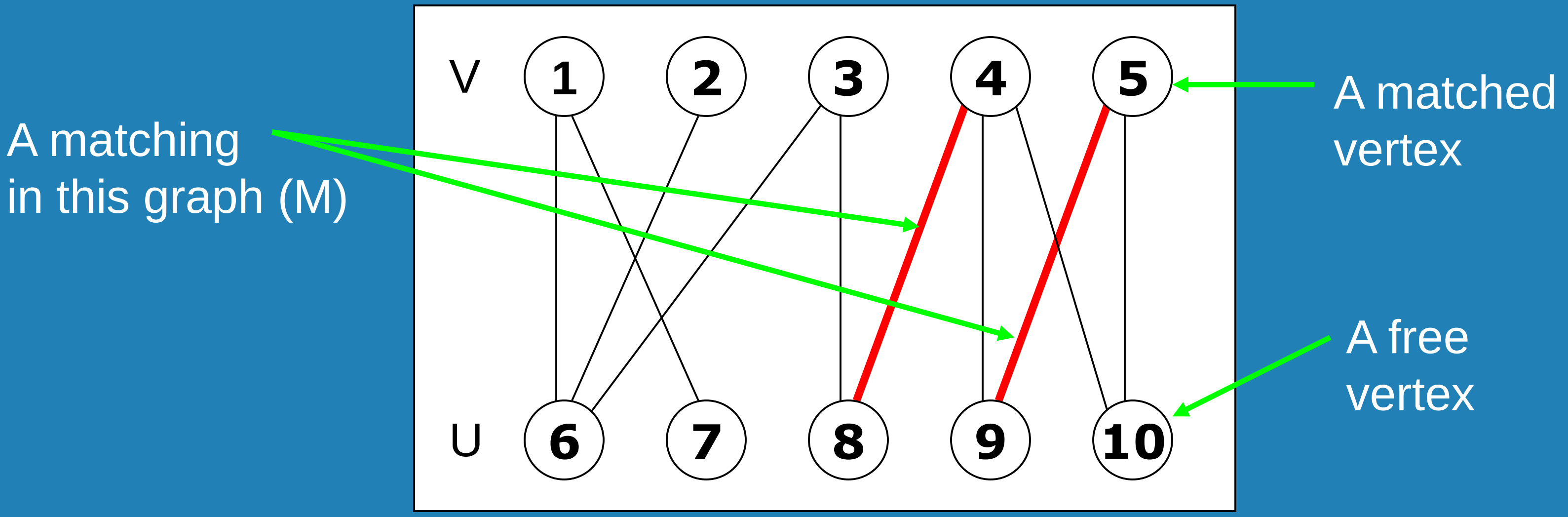
a matching
in this graph
M = {(4,8), (5,9)}



A *maximum* (or *maximum cardinality*) *matching* is a matching with the largest number of edges

- always exists
- not always unique

Free Vertices and Maximum Matching



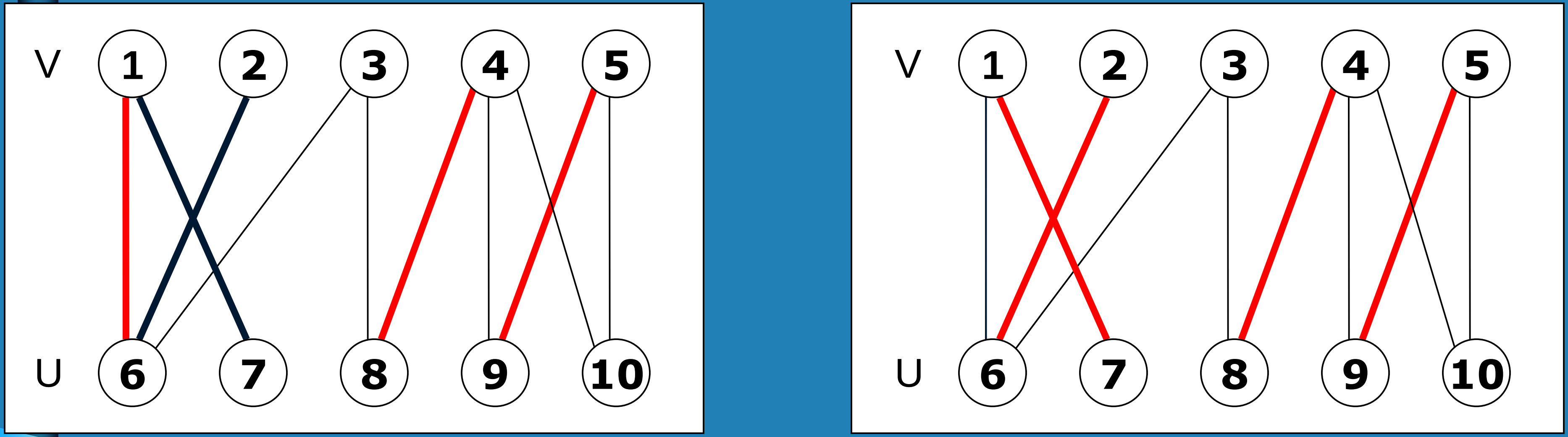
For a given matching M, a vertex is called *free* (or *unmatched*) if it is not an endpoint of any edge in M; otherwise, a vertex is said to be *matched*

- **If every vertex is** matched, **then M is a maximum matching**
- **If there are** unmatched **or** free **vertices, then M may be able to be improved**
- **We can immediately increase a matching by adding an edge connecting two free vertices (e.g., (1,6) above)**

Augmenting Paths and Augmentation

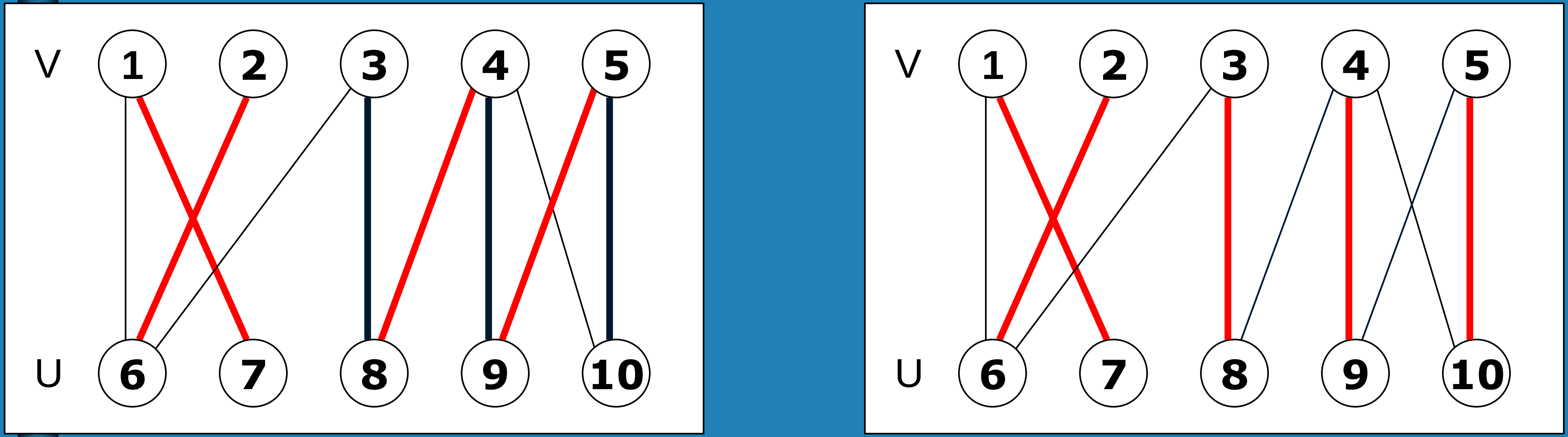
An *augmenting path* for a matching M is a path from a free vertex in V to a free vertex in U whose edges alternate between edges not in M and edges in M

- **The length of an augmenting path is always odd**
- **Adding to M the odd numbered path edges and deleting from it the even numbered path edges increases the matching size by 1 (*augmentation*)**
- **One-edge path between two free vertices is special case of augmenting path**



Augmentation along path 2,6,1,7

Augmenting Paths (another example)



Augmentation along
3, 8, 4, 9, 5, 10

- **Matching on the right is maximum (*perfect* matching)**
- **<u>Theorem</u> A matching M is maximum if and only if there exists no augmenting path with respect to M**

Augmenting Path Method (template)

- Start with some initial matching
  - e.g., the empty set
- Find an augmenting path and augment the current matching along that path
  - e.g., using breadth-first search like method
- When no augmenting path can be found, terminate and return the last matching, which is maximum

BFS-based Augmenting Path Algorithm

- Initialize queue Q with all free vertices in one of the sets (say V)

- While Q is not empty, delete front vertex *w* and label every unlabeled vertex *u* adjacent to *w* as follows:

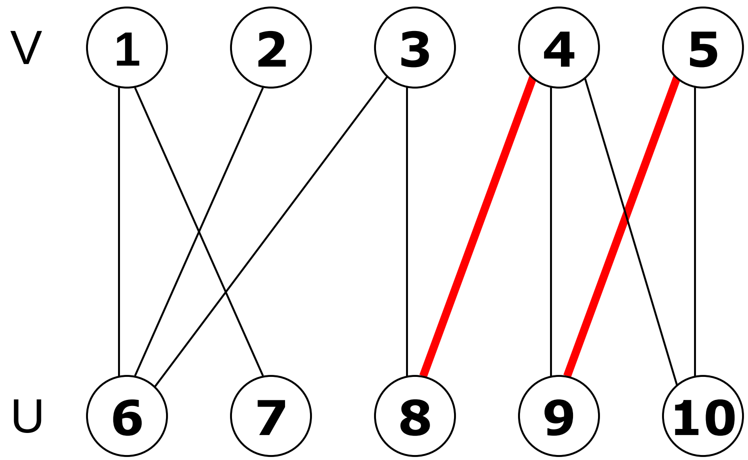
  Case 1 (*w* is in V)
  If *u* is free, augment the matching along the path ending at *u* by moving backwards until a free vertex in V is reached. After that, erase all labels and reinitialize Q with all the vertices in V that are still free
  If *u* is matched (not with *w*), label *u* with *w* and enqueue *u*

  Case 2 (*w* is in U) Label its matching mate *v* with *w* and enqueue *v*

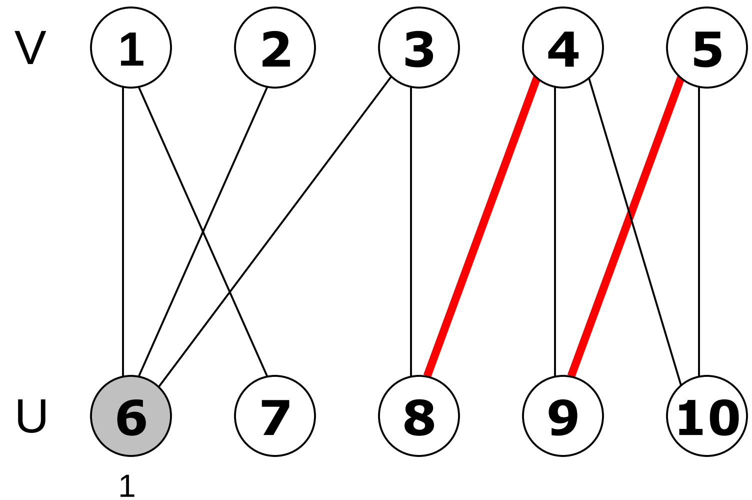
- After Q becomes empty, return the last matching, which is maximum

Example (revisited)

Initial Graph



Resulting Graph



Queue: 1 2 3

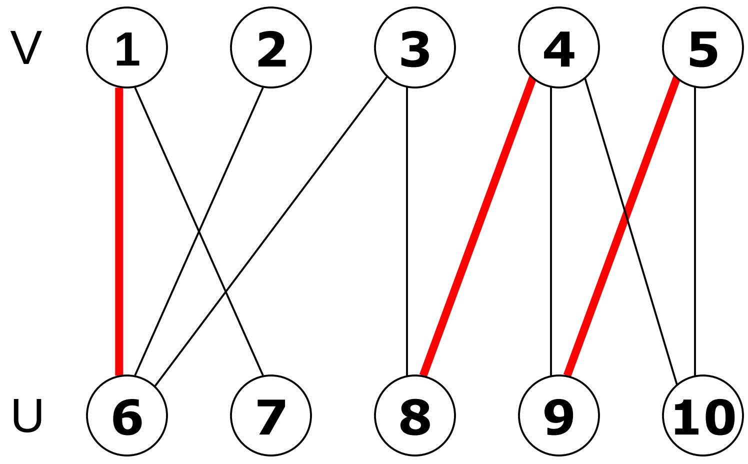
Queue: 1 2 3

Augment from 6

Each vertex is labeled with the vertex it was reached from. Queue deletions are indicated by arrows. The free vertex found in U is shaded and labeled for clarity; the new matching obtained by the augmentation is shown on the next slide.

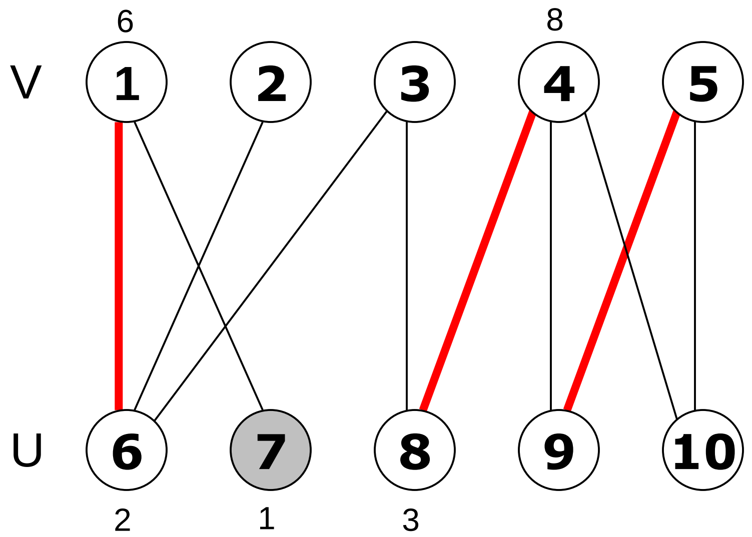
Example (cont.)

Initial Graph



Queue: 2 3

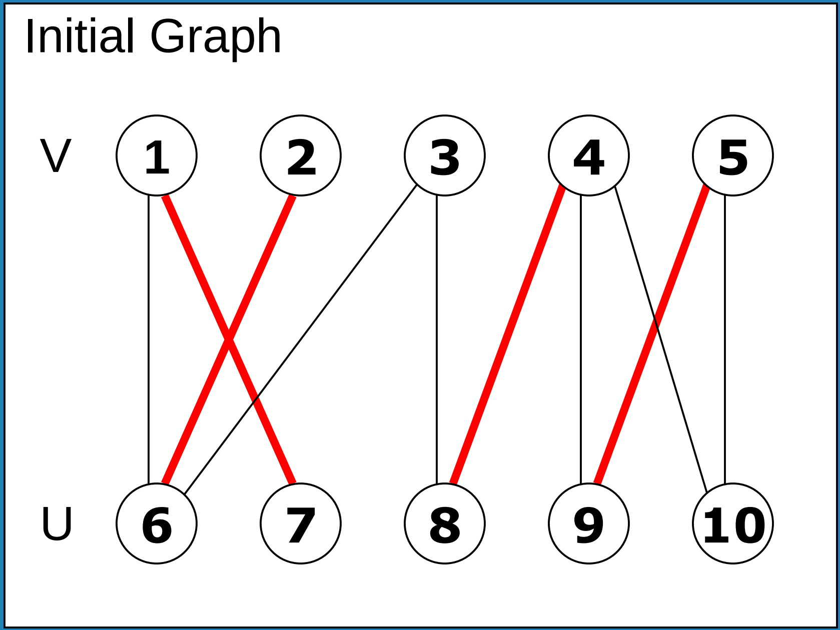
Resulting Graph

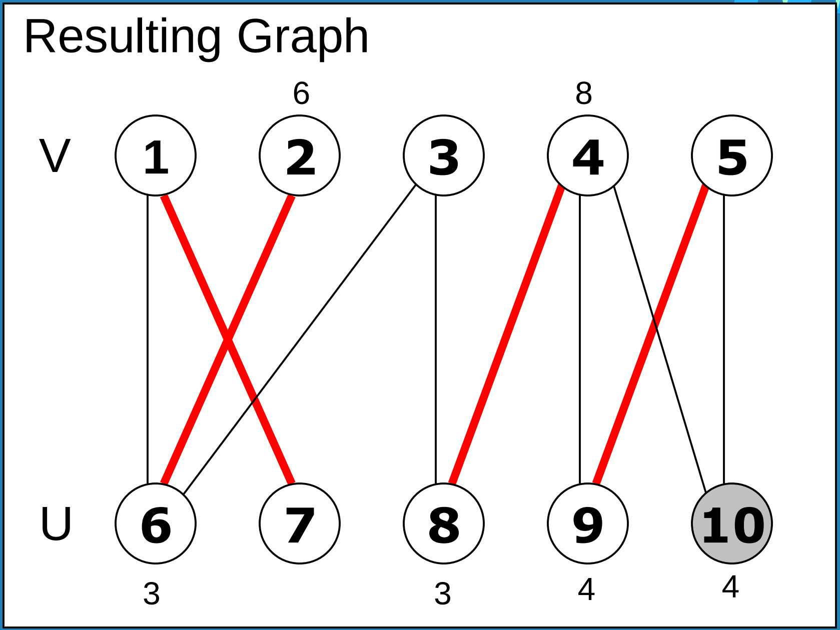


Queue: 2 3 6 8 1 4

Augment from 7

Example (cont.)



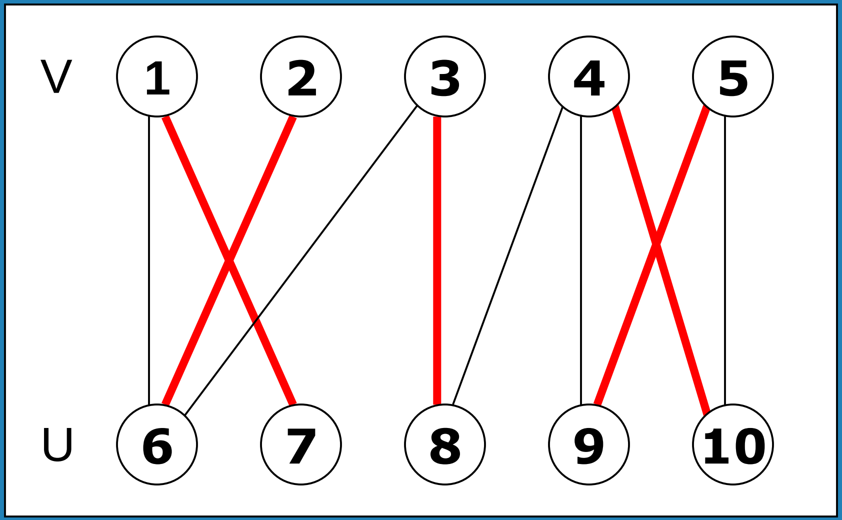


Queue: 3

Queue: 3 6 8 2 4 9

Augment from 10

Example: maximum matching found



maximum matching

- This matching is maximum since there are no remaining free vertices in V (the queue is empty)
- Note that this matching differs from the maximum matching found earlier

Maximum-matching algorithm for bipartite graphs

Input: A bipartite graph $G = \langle V, U, E \rangle$

Output: A maximum-cardinality matching M in the input graph

```
initialize set M of edges with some valid matching (e.g., the empty set)
initialize queue Q with all the free vertices in V (in any order)
while not Empty(Q) do
    w ← Front(Q);  Dequeue(Q)
    if w ∈ V
        for every vertex u adjacent to w do
            if u is free
                //augment
                M ← M ∪ (w, u)
                v ← w
                while v is labeled do
                    u ← vertex indicated by v's label;   M ← M − (v, u)
                    v ← vertex indicated by u's label;   M ← M ∪ (v, u)
                remove all vertex labels
                reinitialize Q with all free vertices in V
                break //exit the for loop
            else //u is matched
                if (w, u) ∉ M and u is unlabeled
                    label u with w
                    Enqueue(Q, u)
    else //w ∈ U (and matched)
        label the mate v of w with "w"
        Enqueue(Q, v)
return M //current matching is maximum
```

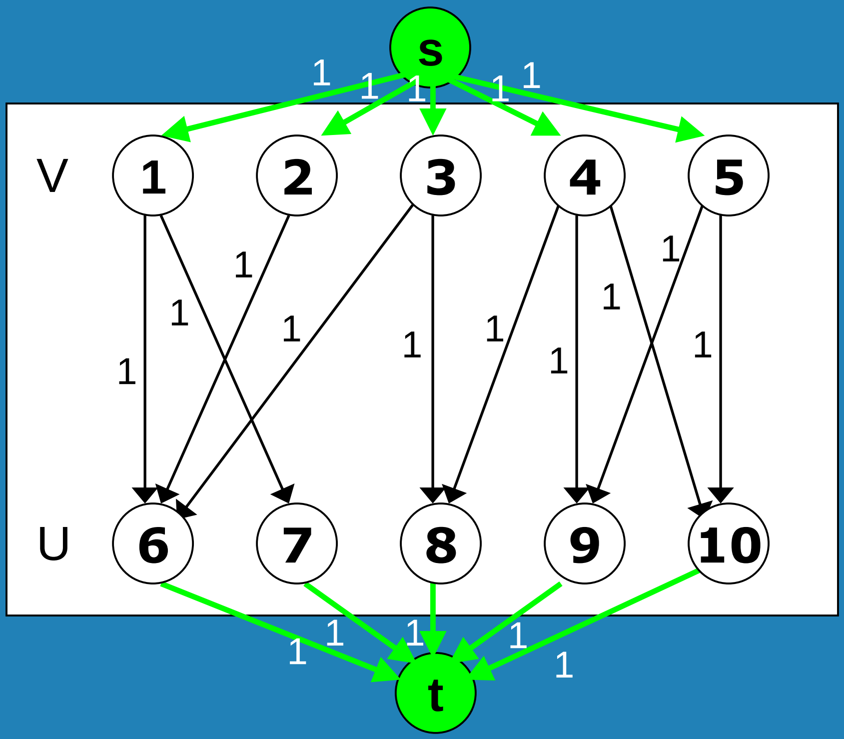
Notes on Maximum Matching Algorithm

- Each iteration (except the last) matches two free vertices (one each from V and U). Therefore, the number of iterations cannot exceed $\lfloor n/2 \rfloor + 1$, where n is the number of vertices in the graph. The time spent on each iteration is in $O(n+m)$, where m is the number of edges in the graph. Hence, the time efficiency is in $O(n(n+m))$

- This can be improved to $O(\mathrm{sqrt}(n)(n+m))$ by combining multiple iterations to maximize the number of edges added to matching M in each search

- Finding a maximum matching in an arbitrary graph is much more difficult, but the problem was solved in 1965 by Jack Edmonds

Conversion to Max-Flow Problem



- Add a source and a sink, direct edges (with unit capacity) from the source to the vertices of V and from the vertices of U to the sink
- Direct all edges from V to U with unit capacity

Stable Marriage Problem

- There is a set Y = $\{m_1,\ldots,m_n\}$ of n men and a set X = $\{w_1,\ldots,w_n\}$ of n women. Each man has a ranking list of the women, and each woman has a ranking list of the men (with no ties in these lists).
- A *marriage matching* M is a set of n pairs (m_i, w_j).
- A pair (m, w) is said to be a *blocking pair* for matching M if man m and woman w are not matched in M but prefer each other to their mates in M.
- A marriage matching M is called *stable* if there is no blocking pair for it; otherwise, it's called *unstable*.
- The *stable marriage problem* is to find a stable marriage matching for men's and women's given preferences.

Instance of the Stable Marriage Problem

An instance of the stable marriage problem can be specified either by two sets of preference lists or by a ranking matrix, as in the example below.

men's preferences

	1st	2nd	3rd
Bob:	Lea	Ann	Sue
Jim:	Lea	Sue	Ann
Tom:	Sue	Lea	Ann

women's preferences

	1st	2nd	3rd
Ann:	Jim	Tom	Bob
Lea:	Tom	Bob	Jim
Sue:	Jim	Tom	Bob

ranking matrix

	Ann	Lea	Sue
Bob	2,3	1,2	3,3
Jim	3,1	1,3	2,1
Tom	3,2	2,1	1,2

{(Bob, Ann) (Jim, Lea) (Tom, Sue)} is unstable

{(Bob, Ann) (Jim, Sue) (Tom, Lea)} is stable

Stable Marriage Algorithm (Gale-Shapley)

Step 0 Start with all the men and women being free

Step 1 While there are free men, arbitrarily select one of them and do the following:

Proposal The selected free man m proposes to w, the next woman on his preference list

Response If w is free, she accepts the proposal to be matched with m. If she is not free, she compares m with her current mate. If she prefers m to him, she accepts m's proposal, making her former mate free; otherwise, she simply rejects m's proposal, leaving m free

Step 2 Return the set of n matched pairs

Example

Free men:
Bob, Jim, Tom

	Ann	Lea	Sue
Bob	2,3	$\boxed{1,2}$	3,3
Jim	3,1	1,3	2,1
Tom	3,2	2,1	1,2

Bob proposed to Lea
Lea accepted

Free men:
Jim, Tom

	Ann	Lea	Sue
Bob	2,3	$\boxed{1,2}$	3,3
Jim	3,1	1,3	2,1
Tom	3,2	2,1	1,2

Jim proposed to Lea
Lea rejected

Example (cont.)

Free men:
Jim, Tom

	Ann	Lea	Sue
Bob	2,3	$\boxed{1,2}$	3,3
Jim	3,1	1,3	$\boxed{2,1}$
Tom	3,2	2,1	1,2

Jim proposed to Sue
Sue accepted

Free men:
Tom

	Ann	Lea	Sue
Bob	2,3	$\boxed{1,2}$	3,3
Jim	3,1	1,3	$\boxed{2,1}$
Tom	3,2	2,1	<u>1,2</u>

Tom proposed to Sue
Sue rejected

Example (cont.)

Free men:
Tom

	Ann	Lea	Sue
Bob	2,3	1,2	3,3
Jim	3,1	1,3	$\boxed{2,1}$
Tom	3,2	$\boxed{2,1}$	1,2

Tom proposed to Lea
Lea replaced Bob with Tom

Free men:
Bob

	Ann	Lea	Sue
Bob	$\boxed{2,3}$	1,2	3,3
Jim	3,1	1,3	$\boxed{2,1}$
Tom	3,2	$\boxed{2,1}$	1,2

Bob proposed to Ann
Ann accepted

Analysis of the Gale-Shapley Algorithm

- **The algorithm terminates after no more than n^2 iterations with a stable marriage output**

- **The stable matching produced by the algorithm is always *man-optimal*: each man gets the highest rank woman on his list under any stable marriage. One can obtain the *woman-optimal* matching by making women propose to men**

- **A man (woman) optimal matching is unique for a given set of participant preferences**

- **The stable marriage problem has practical applications such as matching medical-school graduates with hospitals for residency training**