Tutorium #8

16/12/2022

A message containing letters from `A-Z` can be **encoded** into numbers using the following mapping:

```
'A' -> "1"
'B' -> "2"
...
'Z' -> "26"
```

To **decode** an encoded message, all the digits must be grouped then mapped back into letters using the reverse of the mapping above (there may be multiple ways). For example, `"11106"` can be mapped into:

- `"AAJF"` with the grouping `(1 1 10 6)`
- `"KJF"` with the grouping `(11 10 6)`

Note that the grouping `(1 11 06)` is invalid because `"06"` cannot be mapped into `'F'` since `"6"` is different from `"06"`.

Given a string `s` containing only digits, return *the **number** of ways to **decode** it.*

```
def getNumberOfEncodingWays(text):
    # note: d(j) corresponds to the number if unique encodings from text[0:j]
    # hence: d(j) = (if text[j] != "0" ? d(j-1) : 0) + (if "09" < text[j-2:j] < "27" ? d(j-2) : 0)
    if (len(text) == 0) return 0;
    d = [0 * (len(text) + 1)]
    d[1] = 1

    for i in range(1, len(text) + 1):
      if (text[i] != "0"):
        d[i] += d[i-1]
      if (i > 1 and "09" < text[i-2:i] < "27"):
        d[i] += d[i-2]
    return d[len(text)]
```

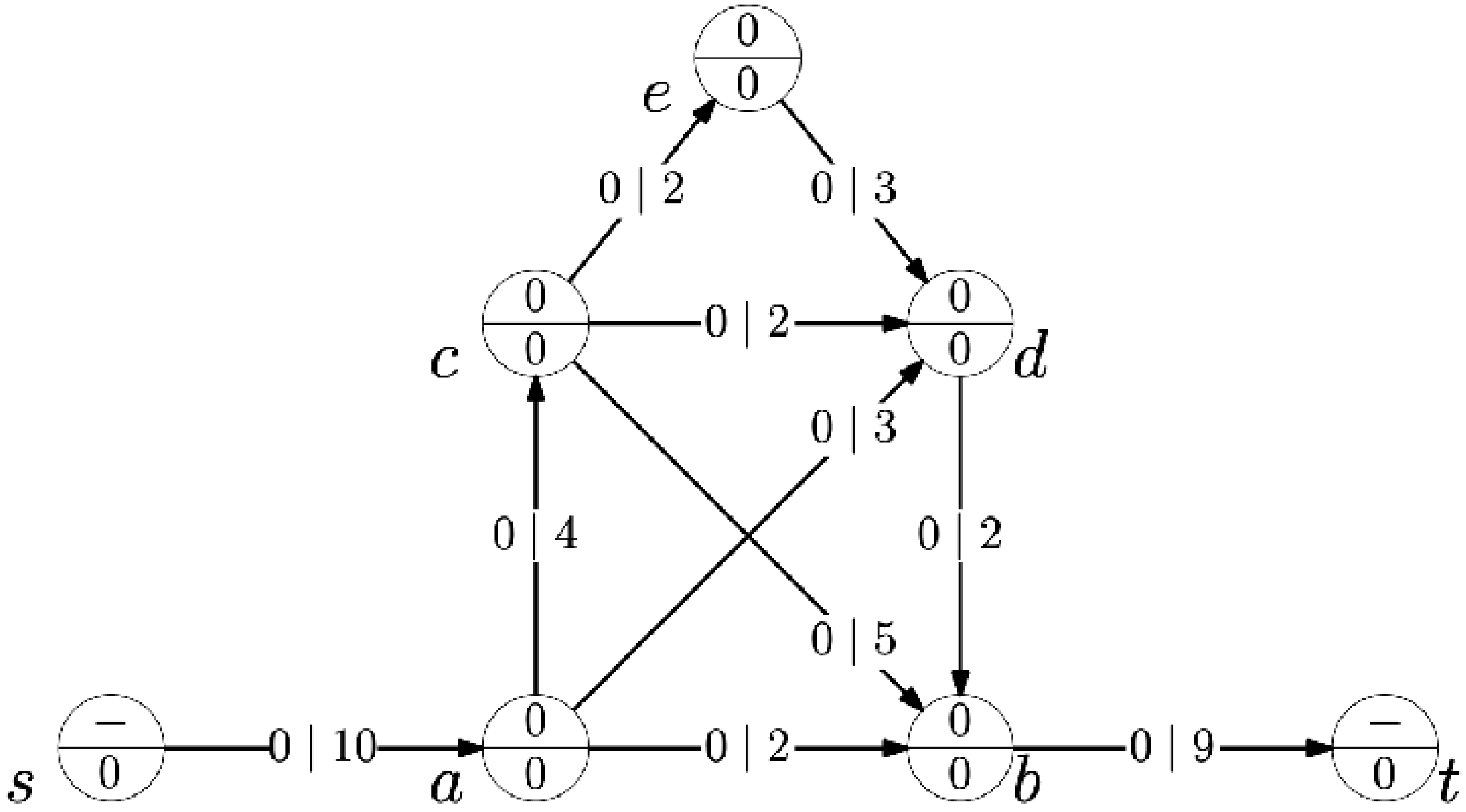
text = "226"

i	0	1	2
d	1	2	2 + 1

= 3

Problem 1 (8 points)

Given is the following flow network:



(das Haus vom Nikolaus mit Vorgarten und Hinterhof)

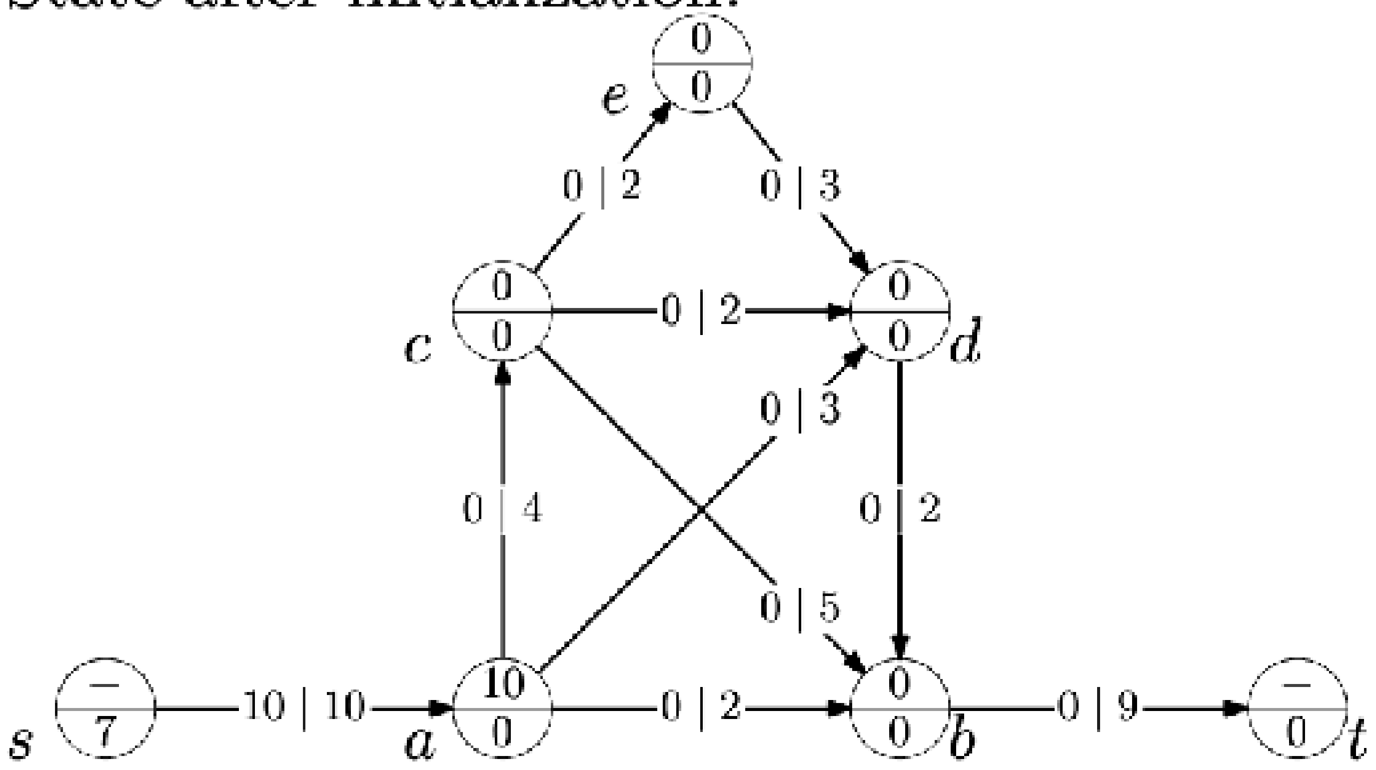
Node labels: level (below), excess (above)
Edge labels: flow (left), capacity (right)

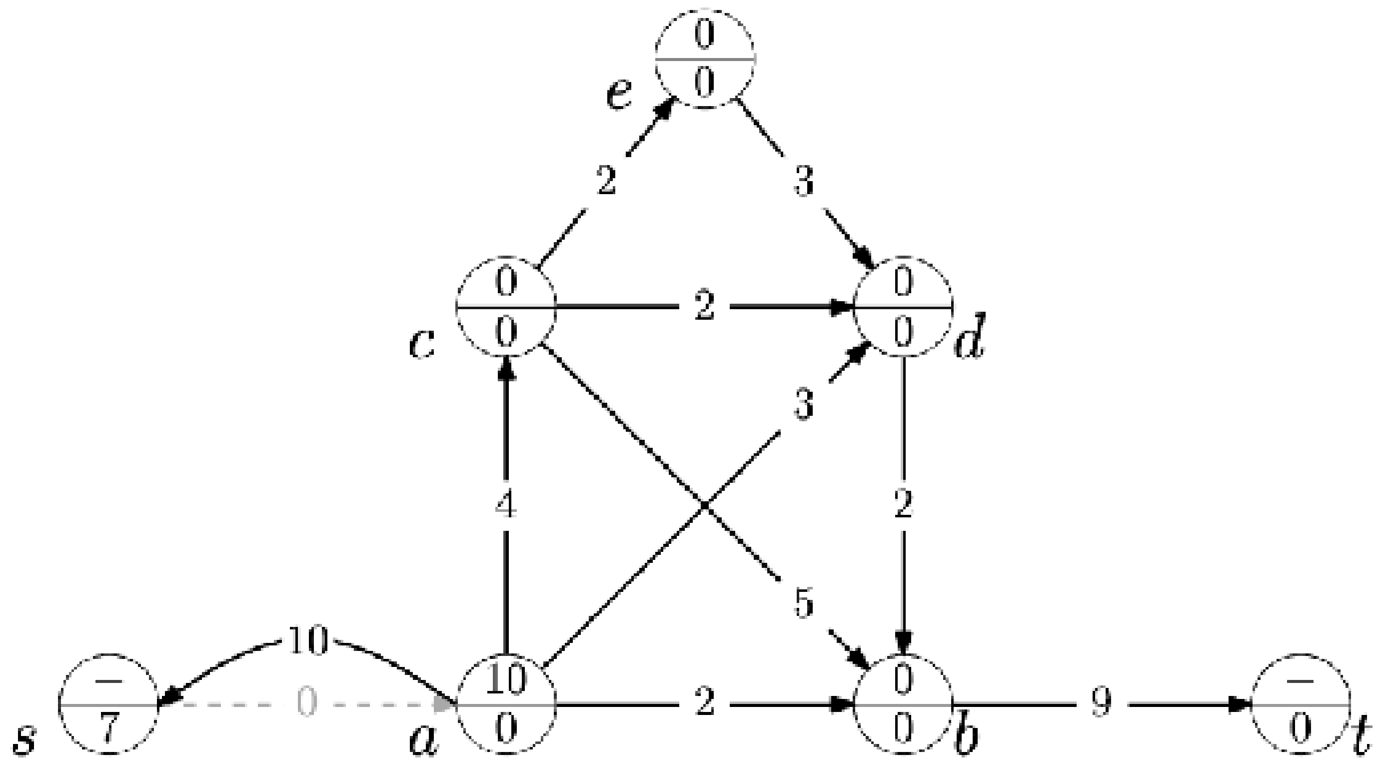
Compute the maximal flow from s to t using the generic *preflow-push* algorithm.

Solution:

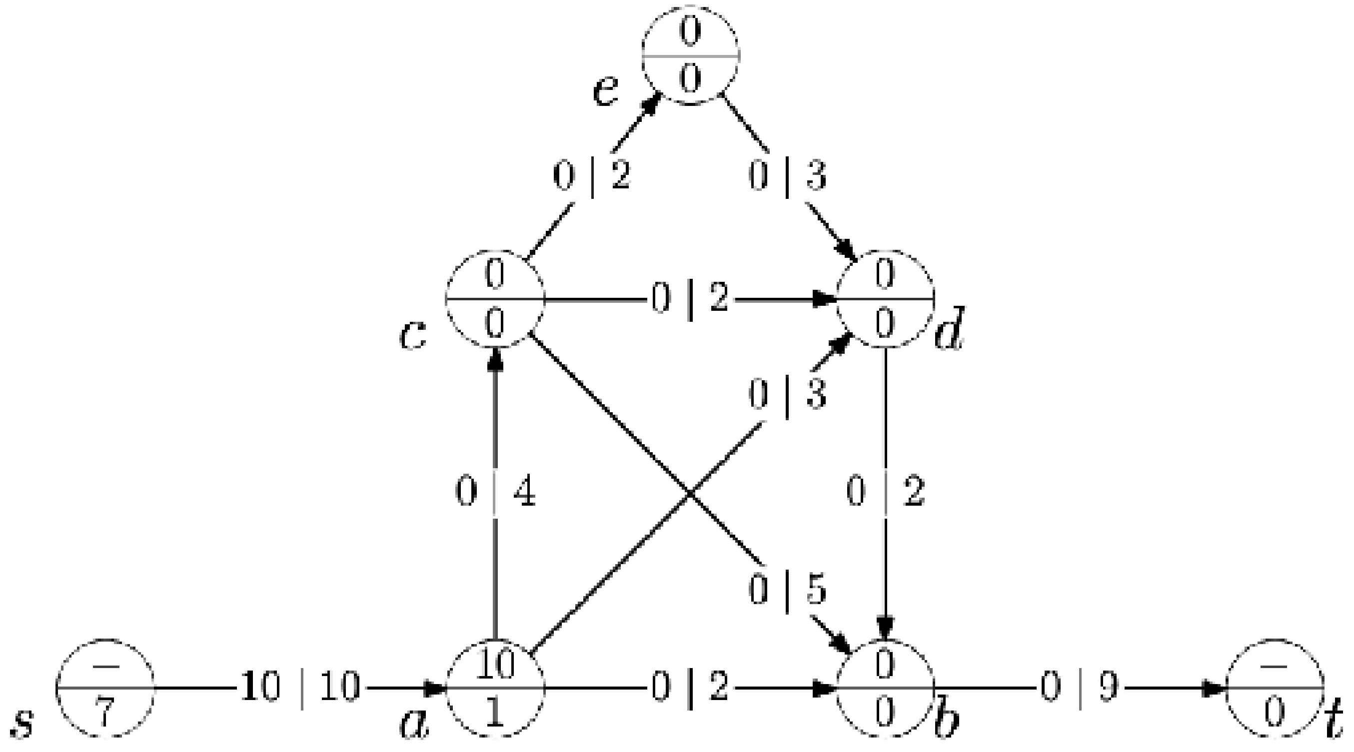
In the following, we execute the *preflow-push* algorithm from the lecture. Active nodes are chosen random. We stay at a node until its whole excess has been moved. This is not the fastest possibility! Left: we show that state of the flow network, right hand side: state of residual network.

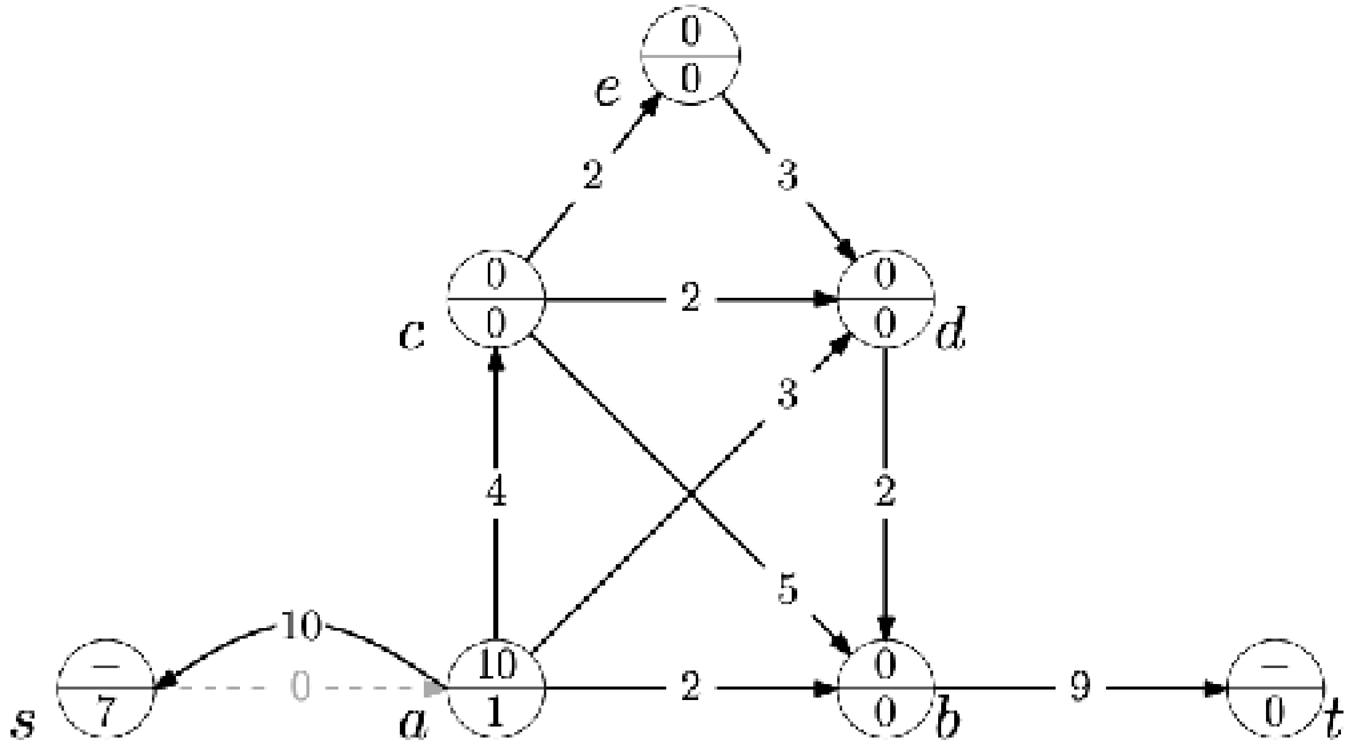
State after initialization:



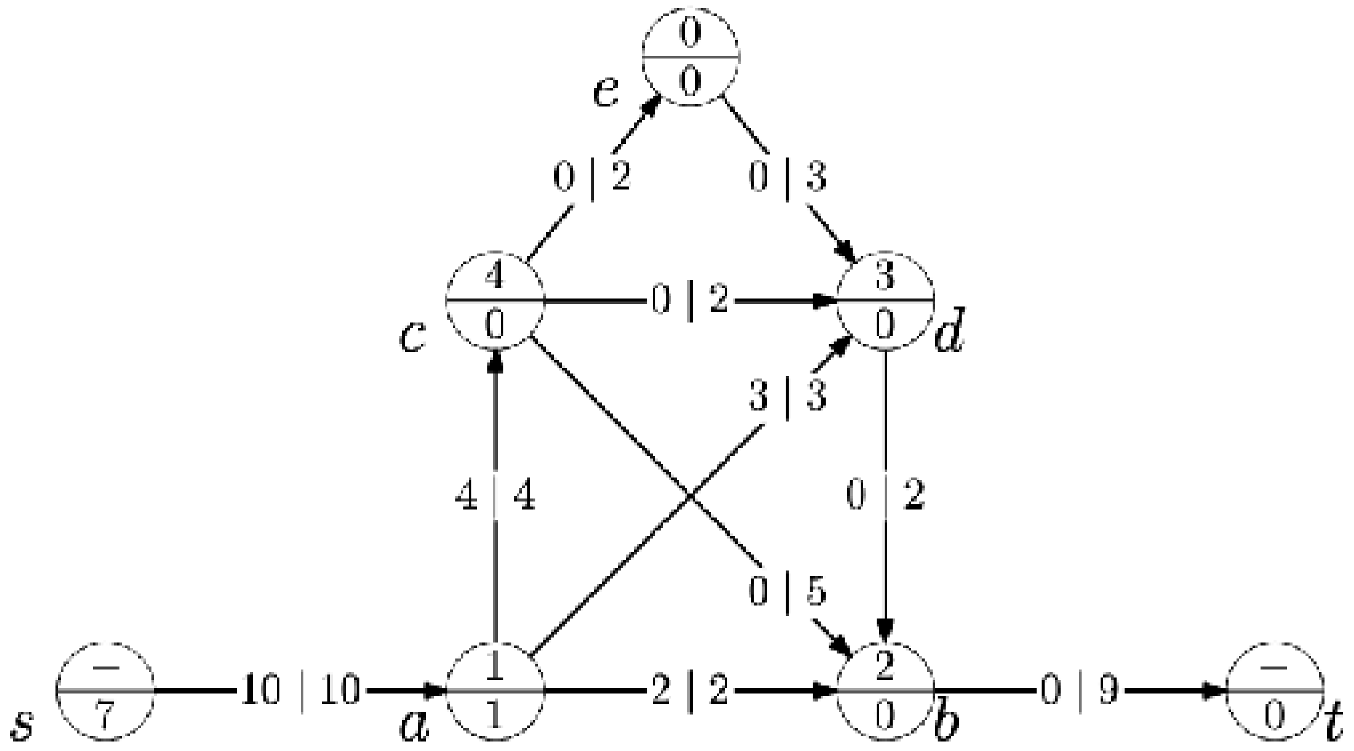


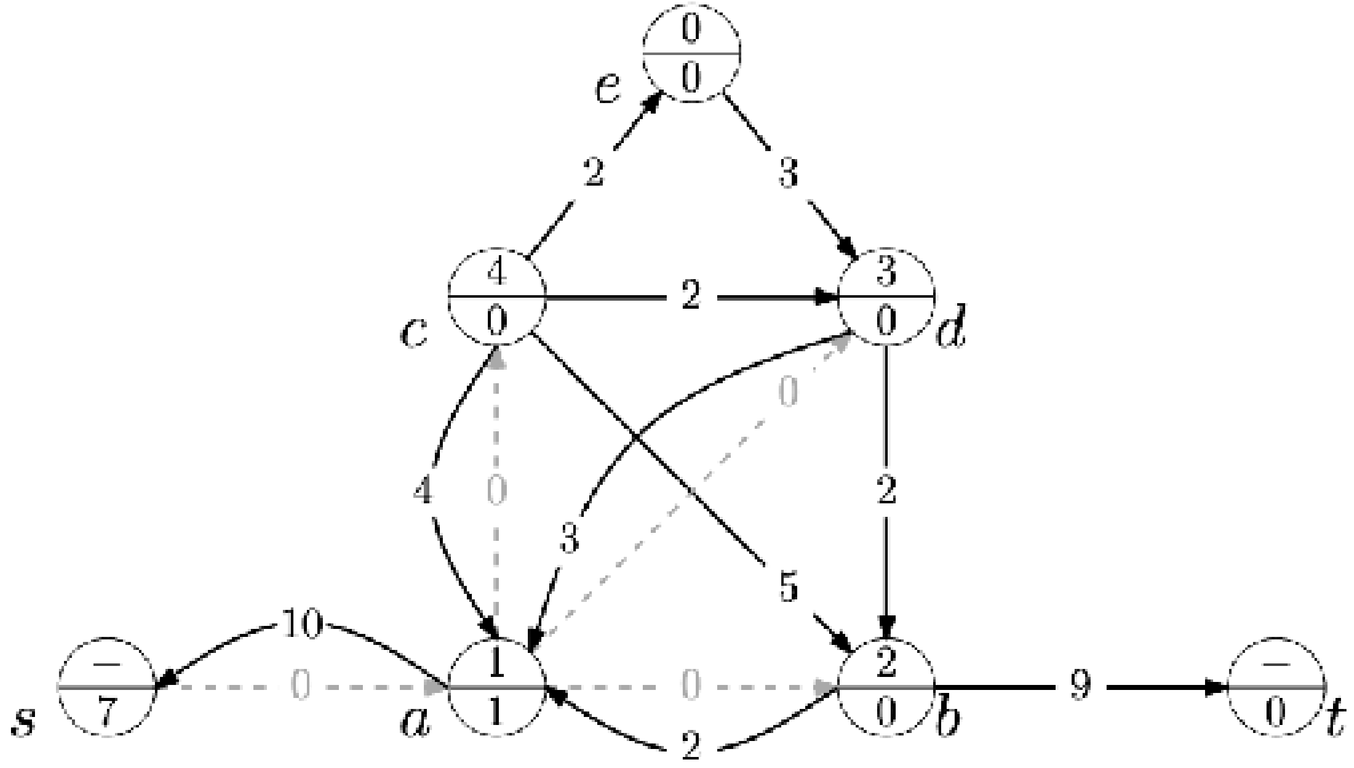
Level of a increased to 1:



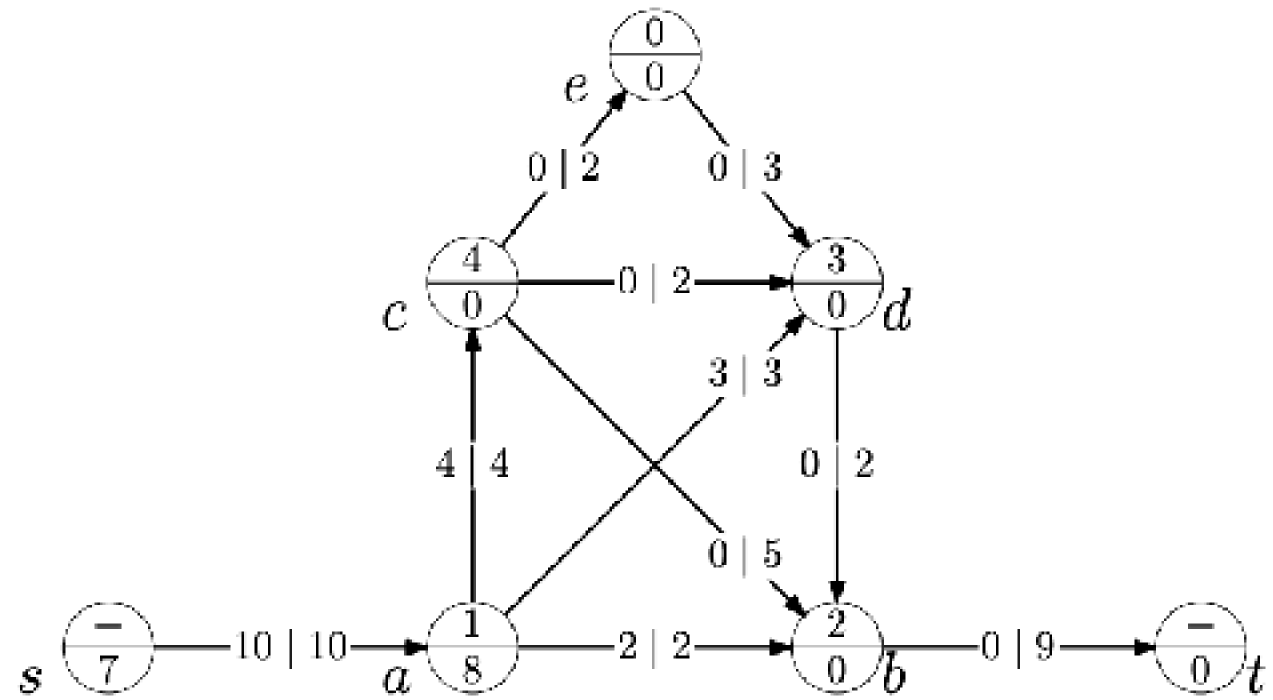


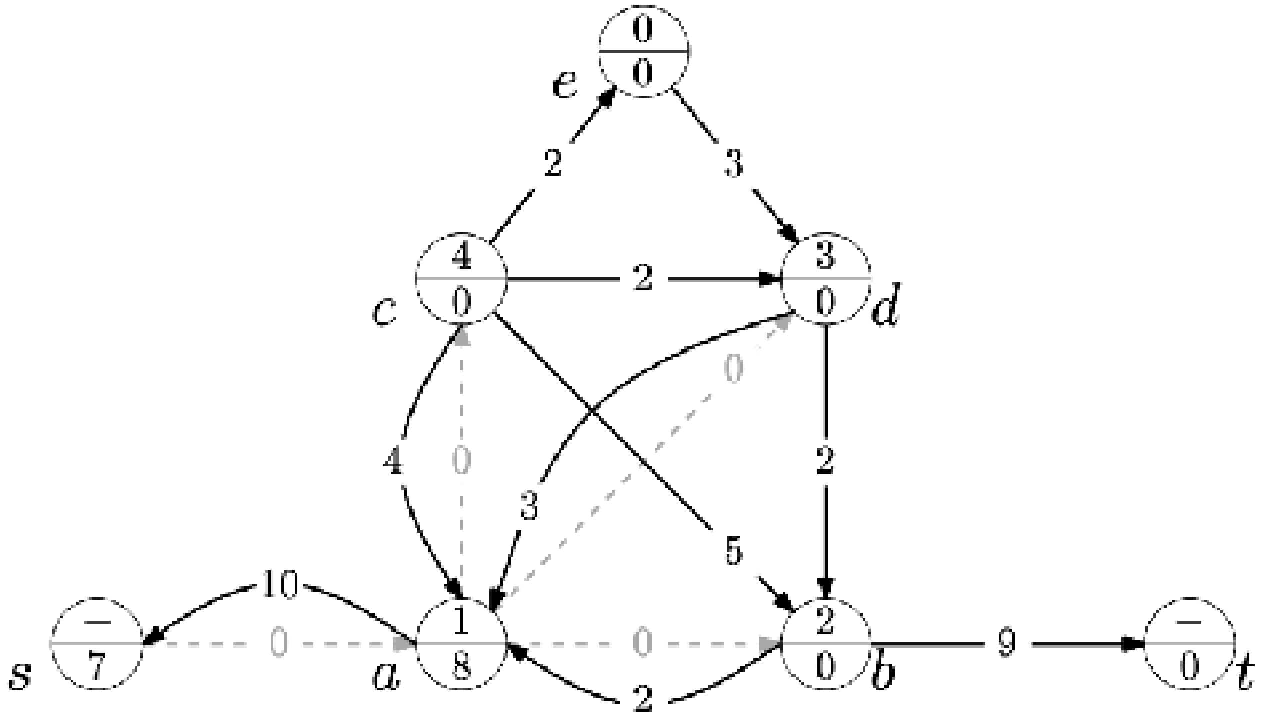
Flow send from a to b, c and d:



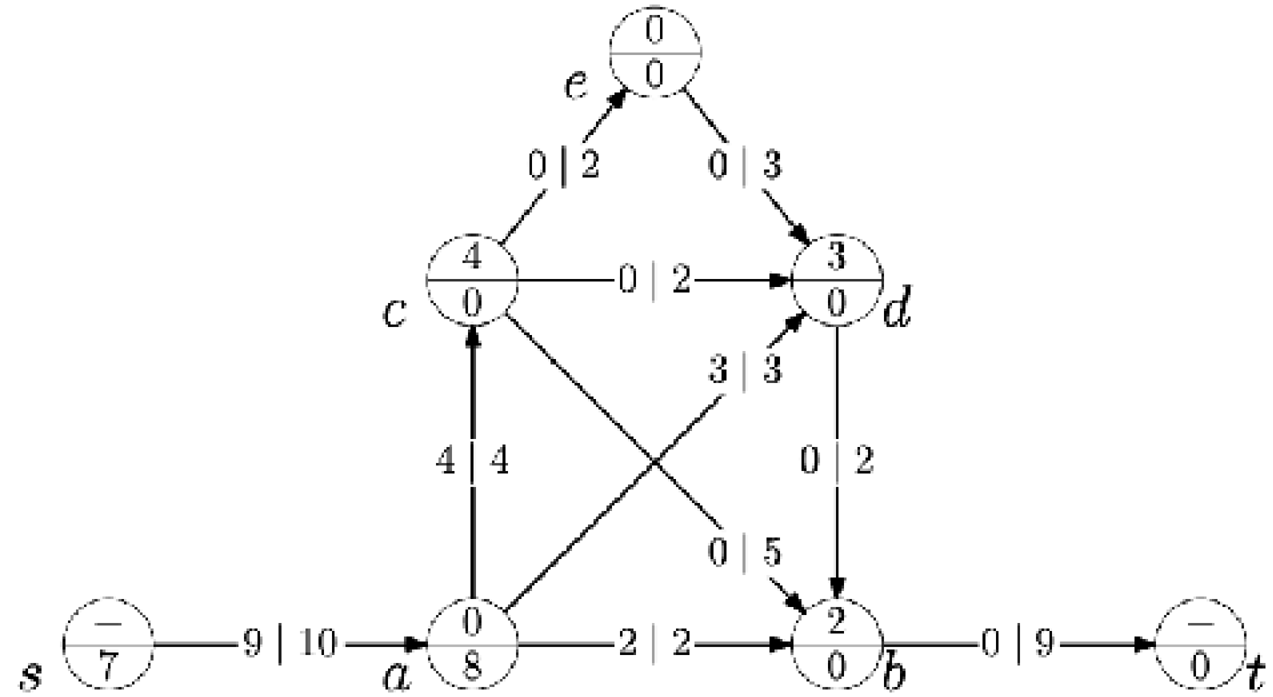


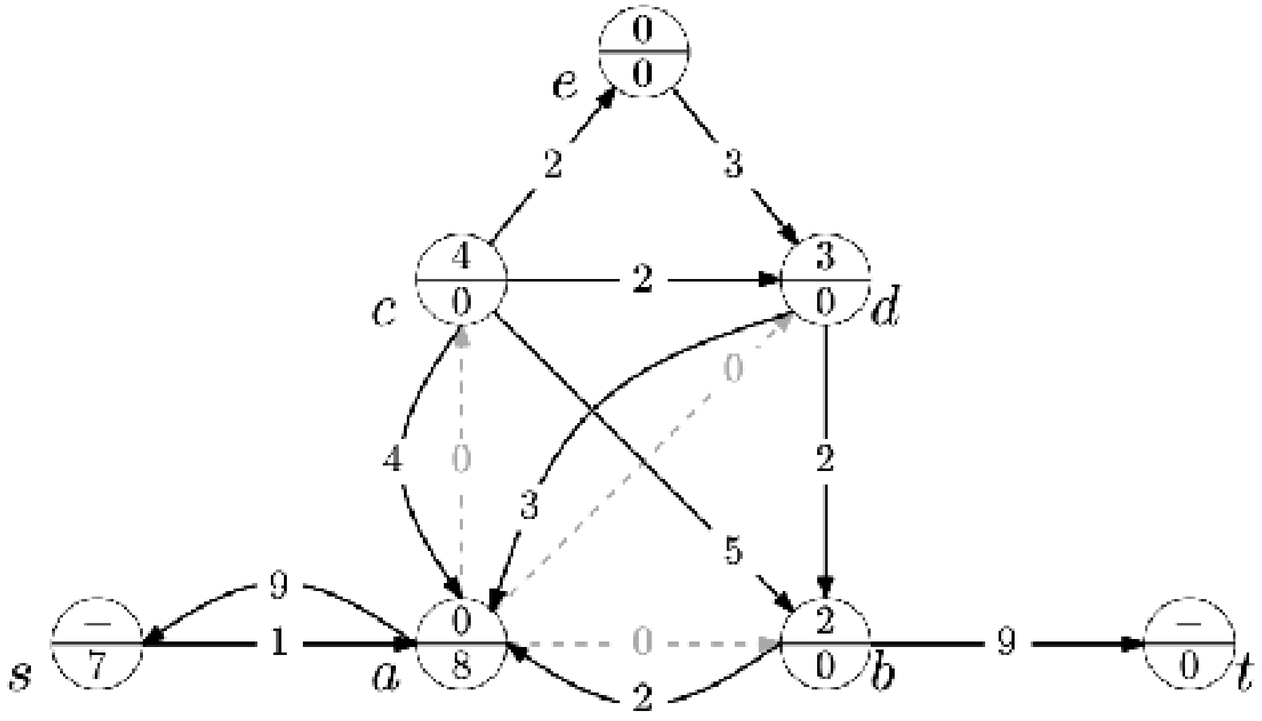
Level of node a increased to 8:



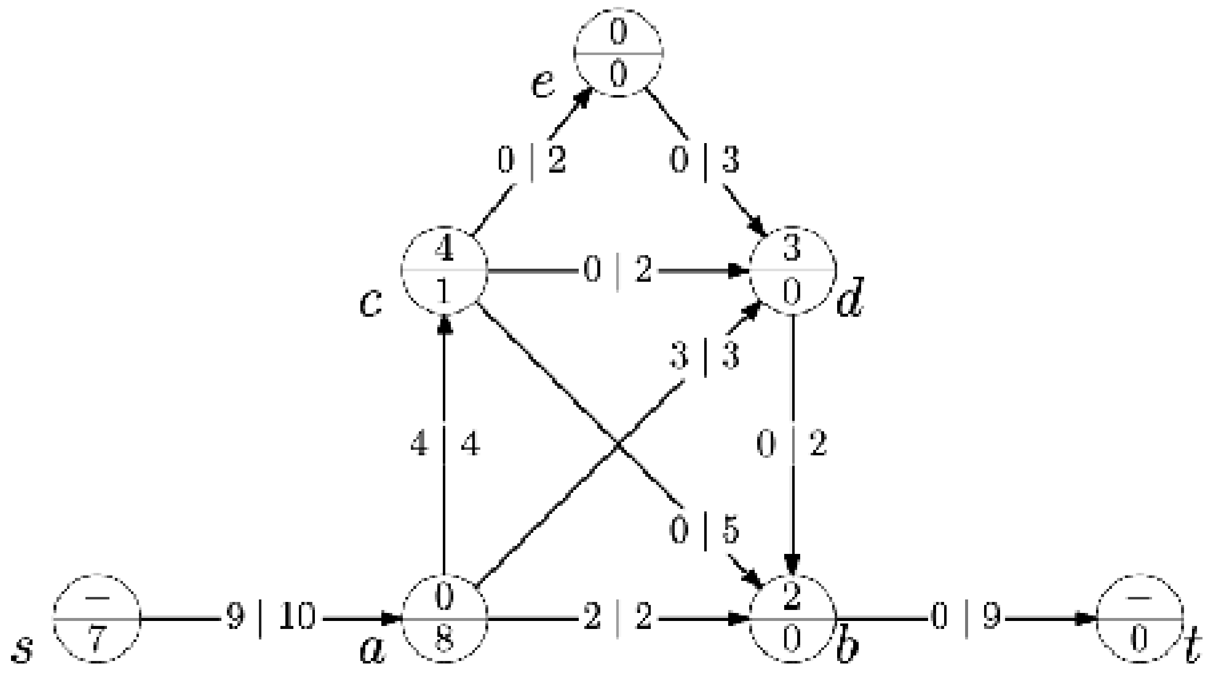


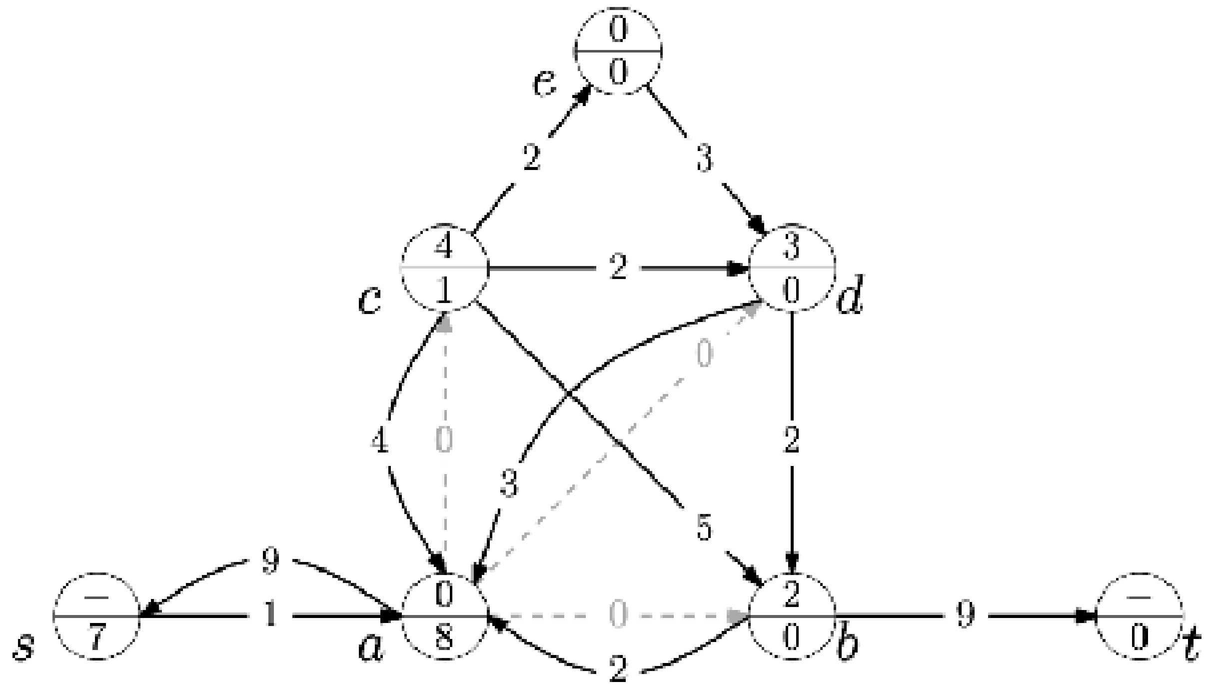
Flow sent from a to s:



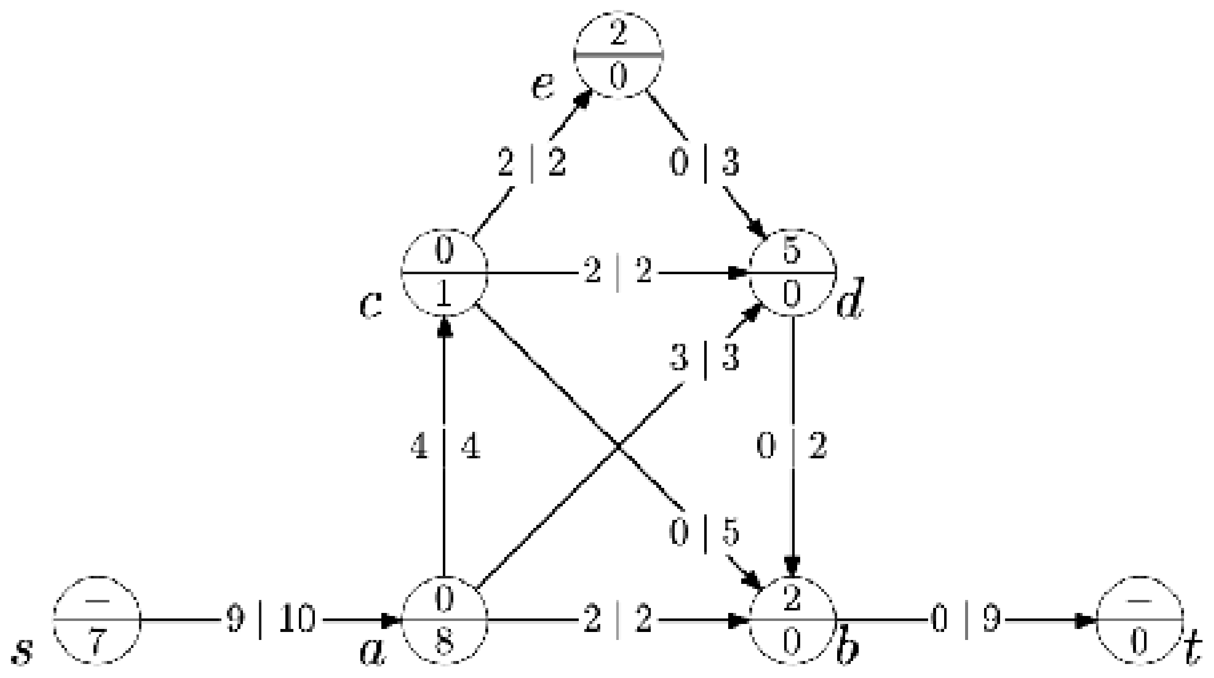


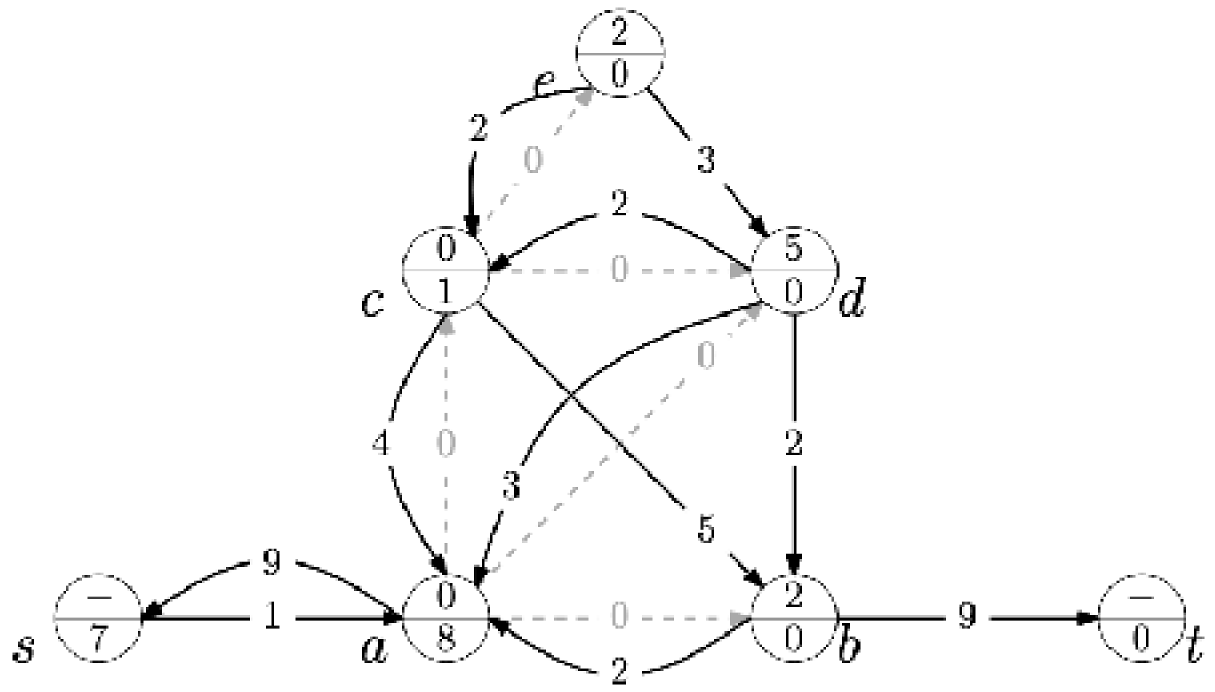
Level of node c increased to 1:



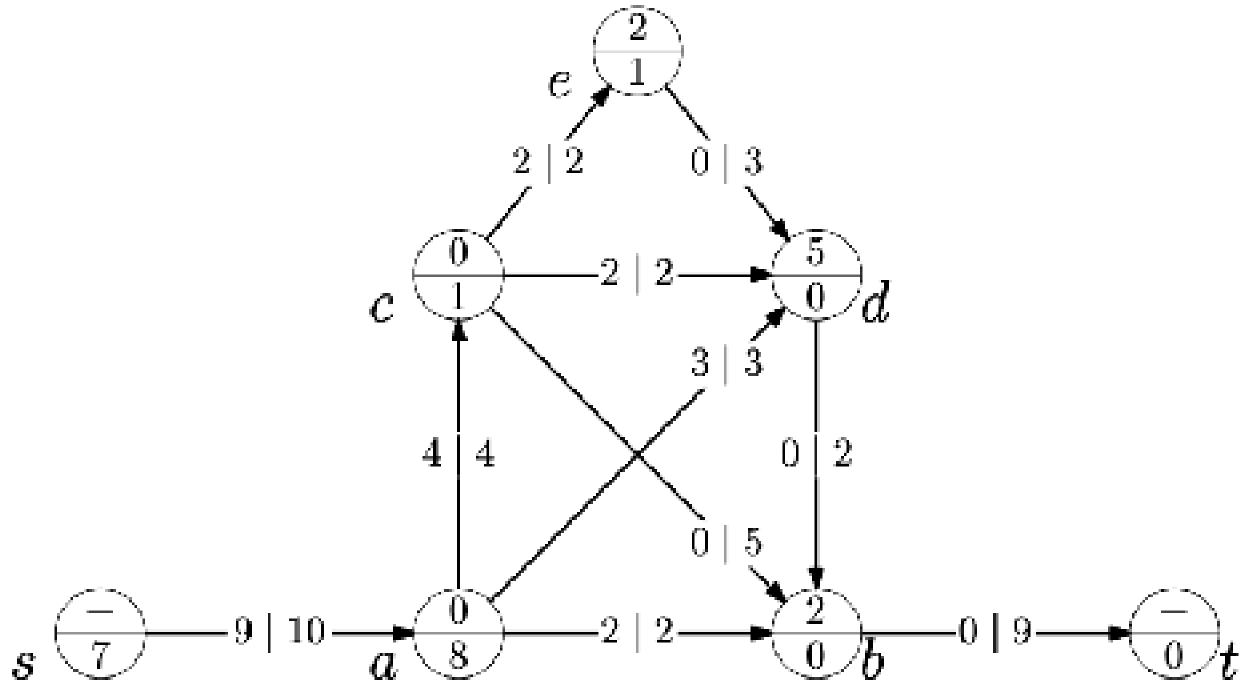


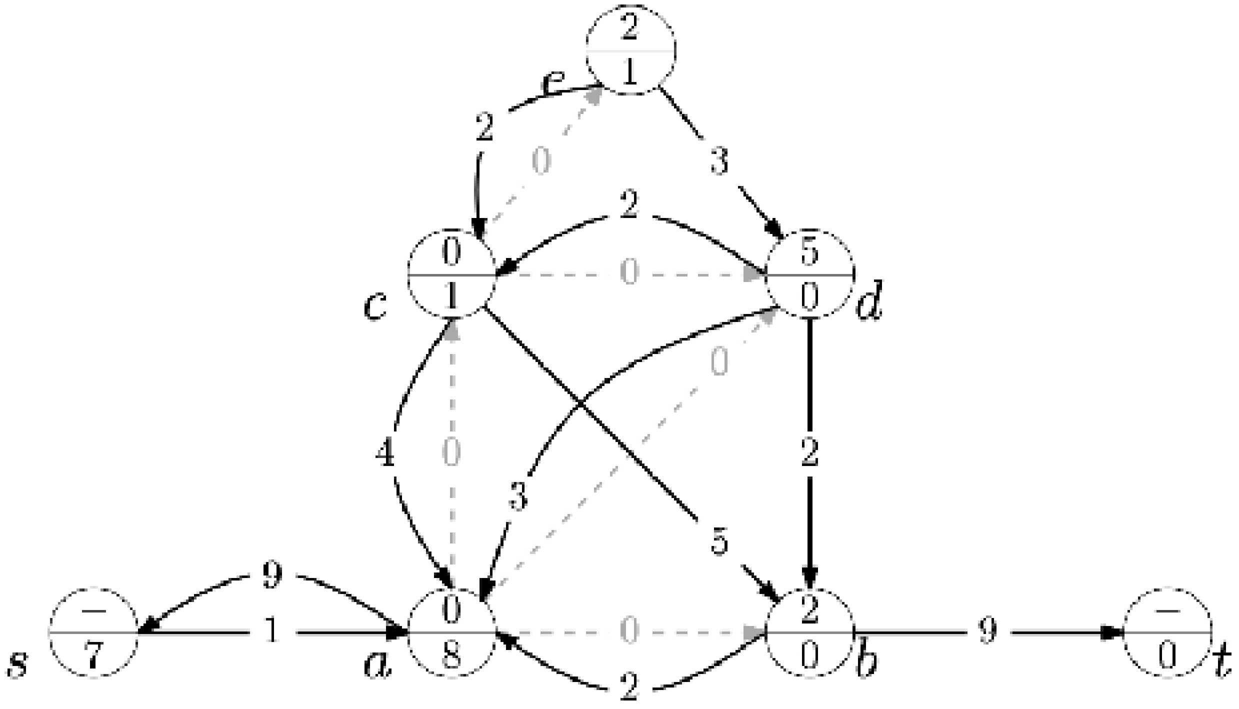
Flow send from c to d and e:



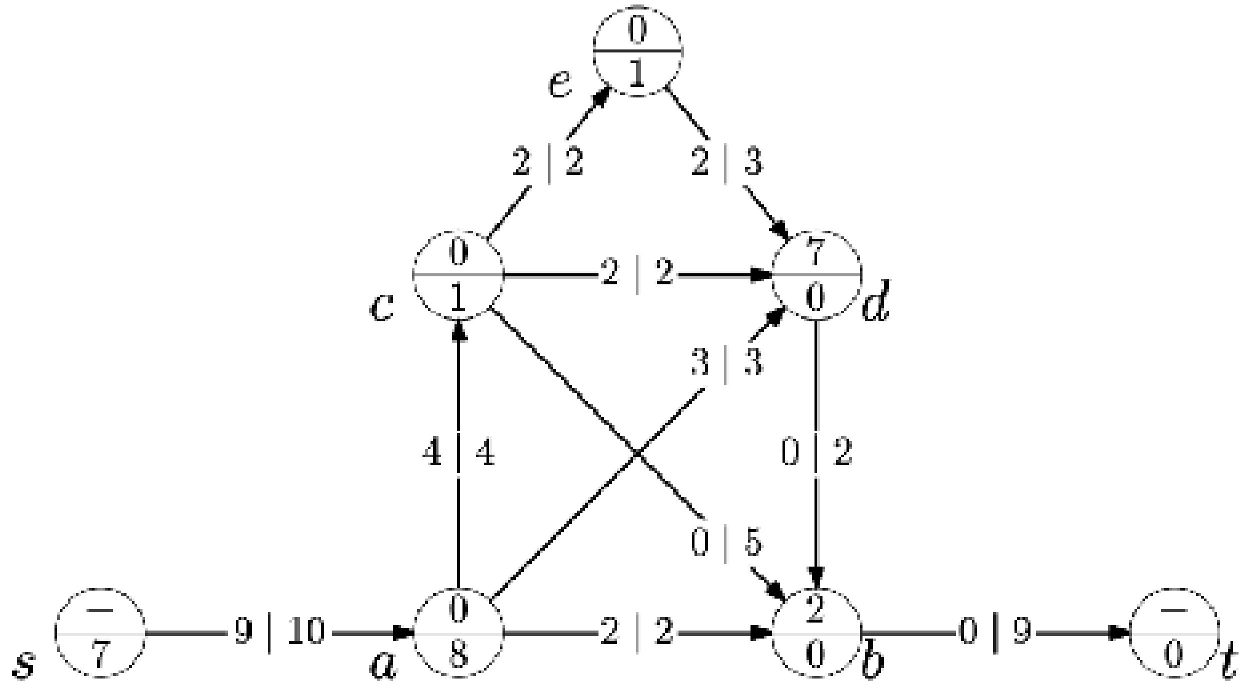


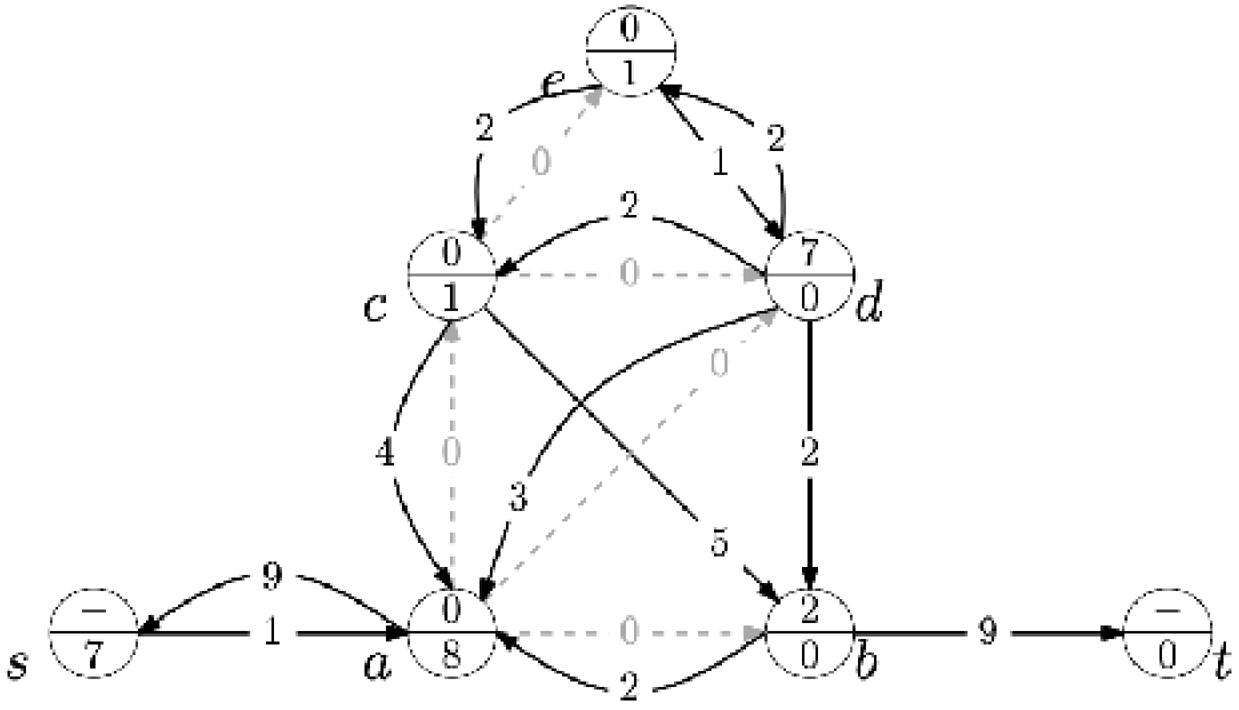
Level of e increased 1:



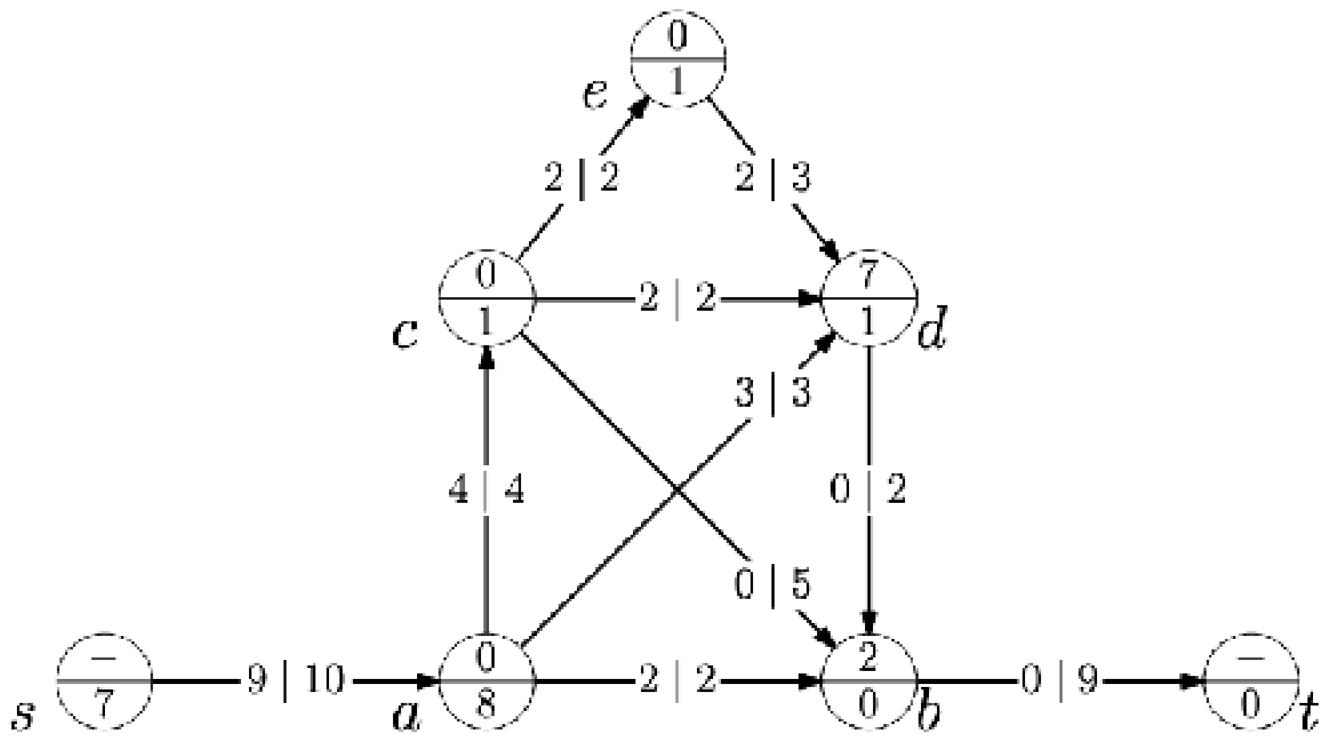


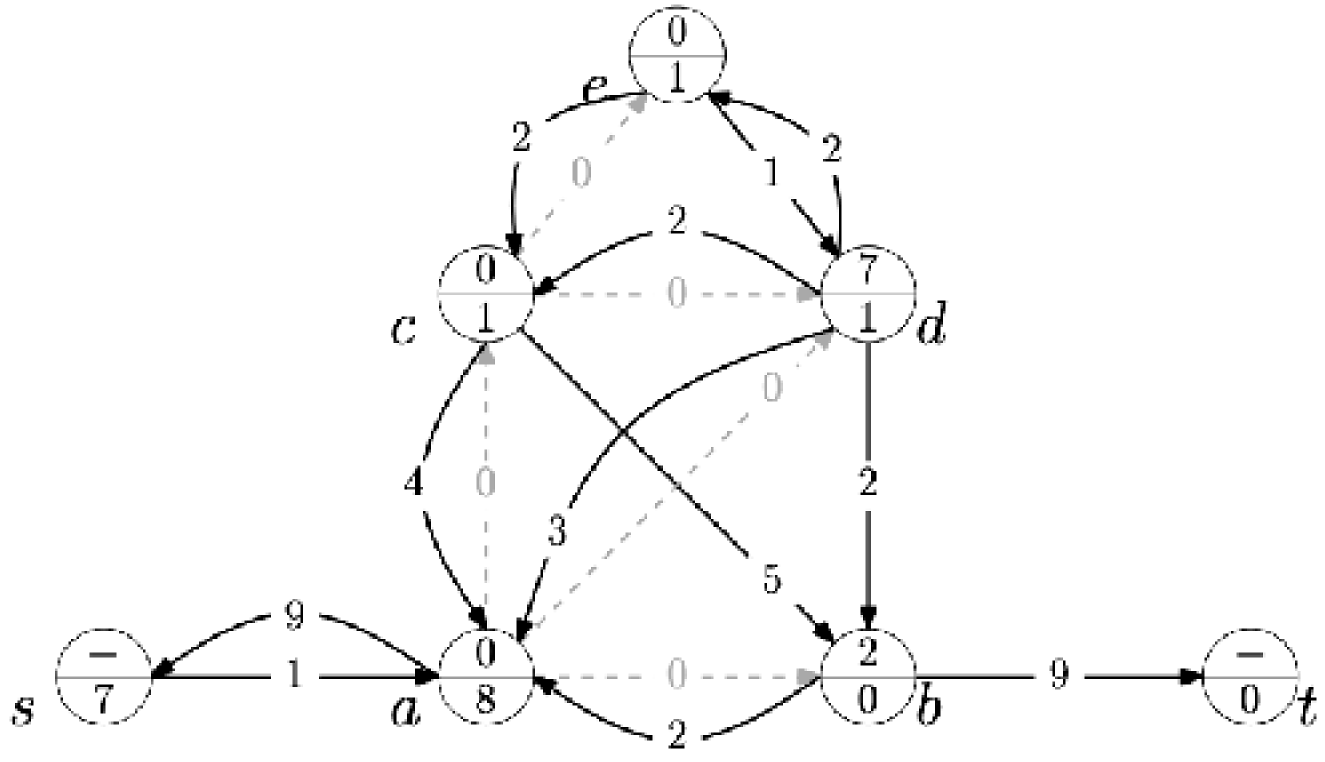
Flow sent from e to d:



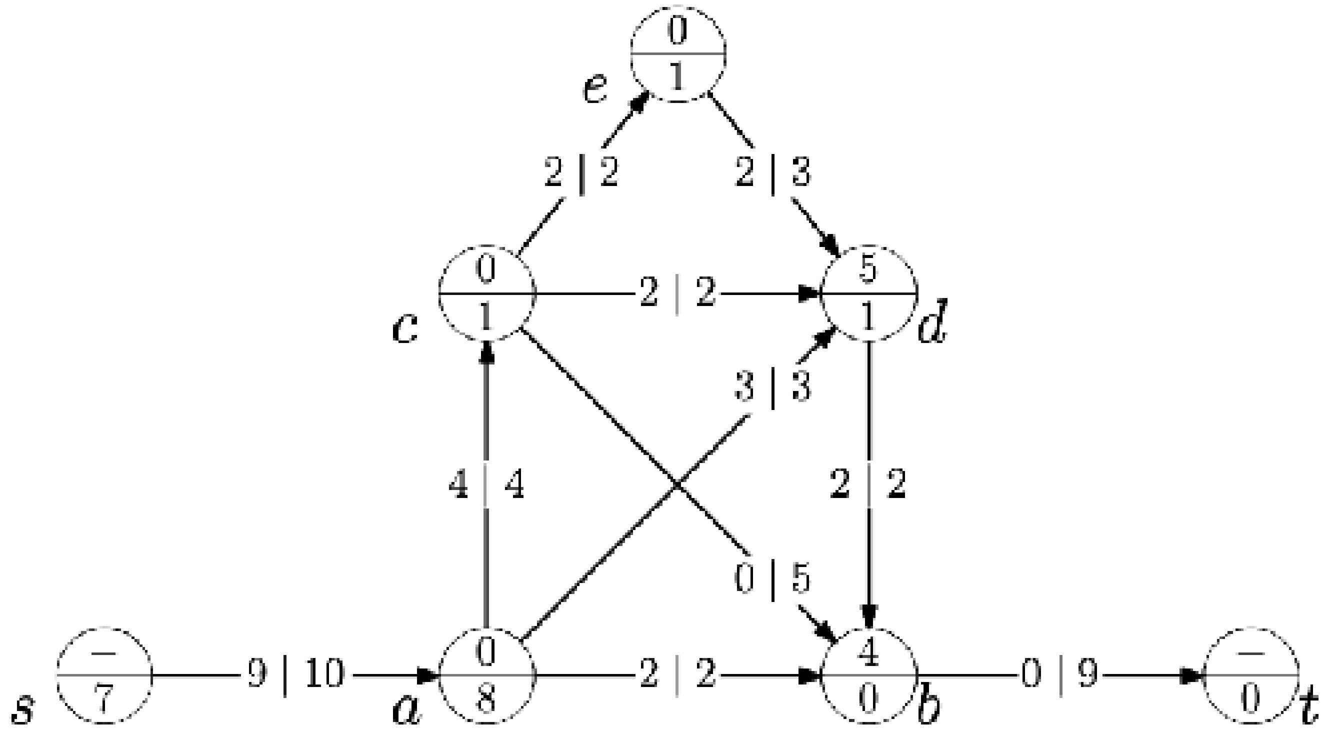


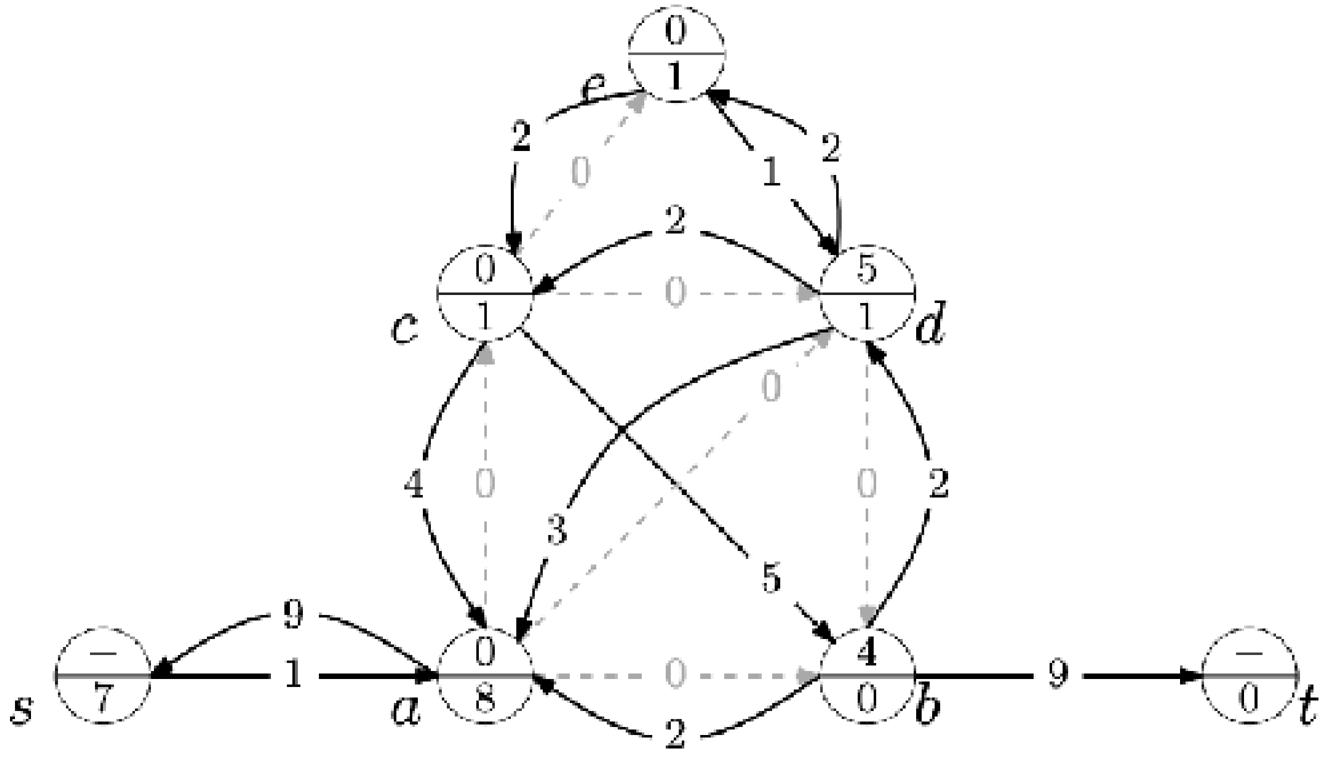
Level of d increased to 1:



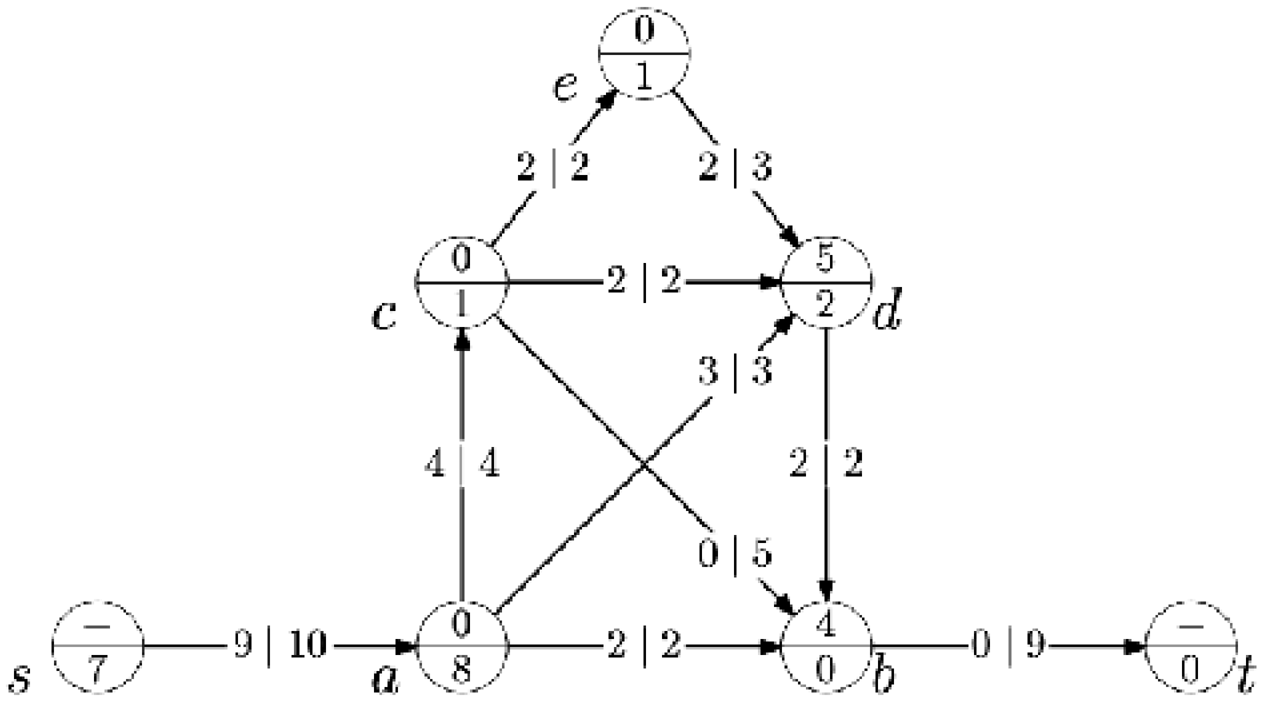


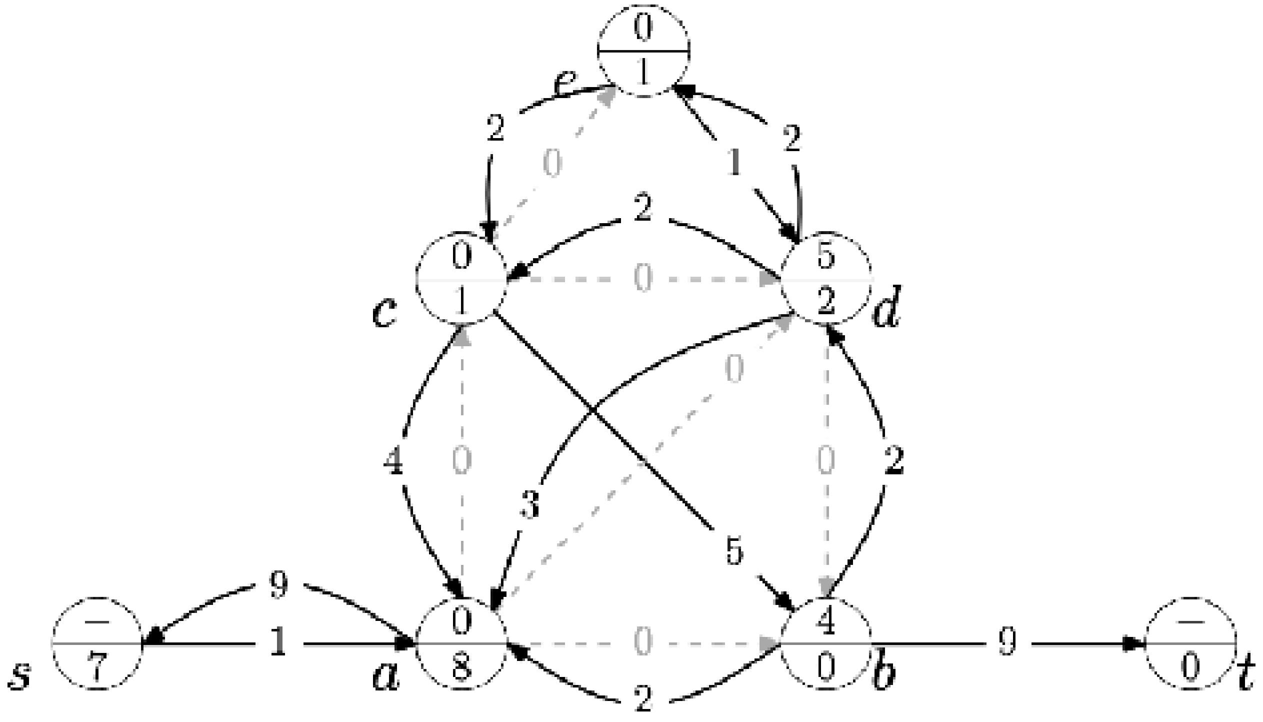
Flow send from d to b:



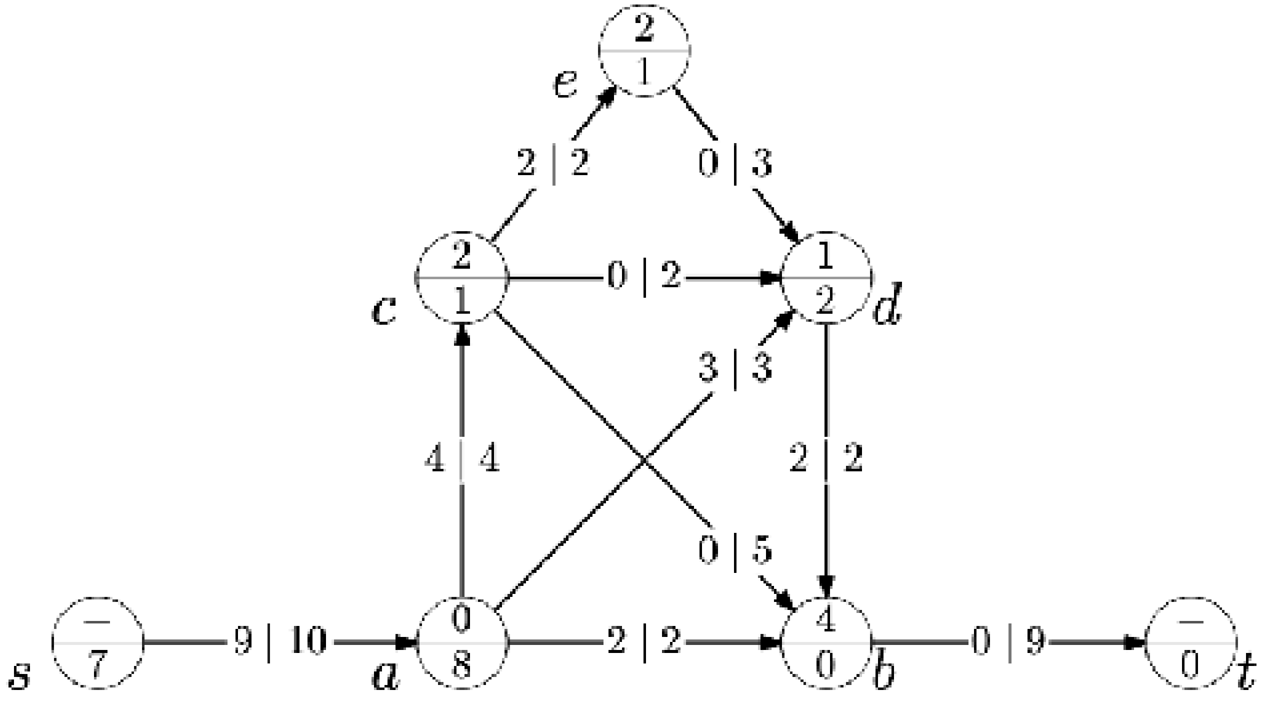


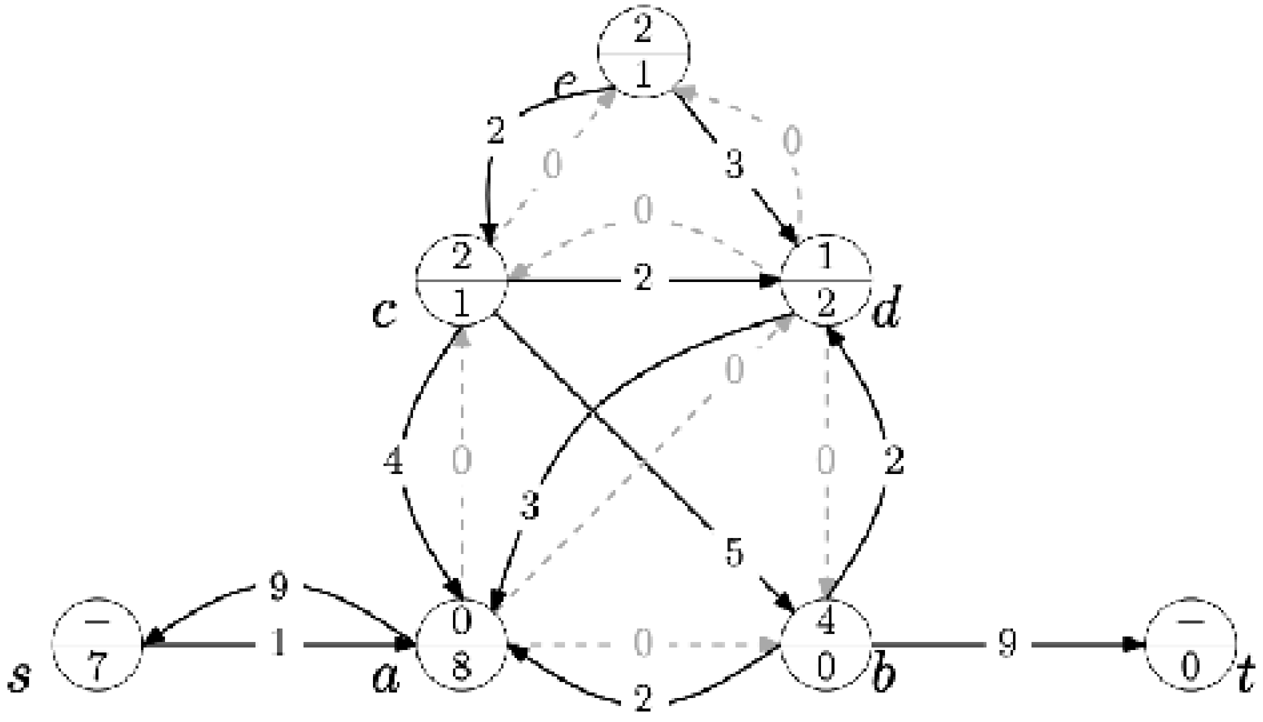
Level of d increased to 2:



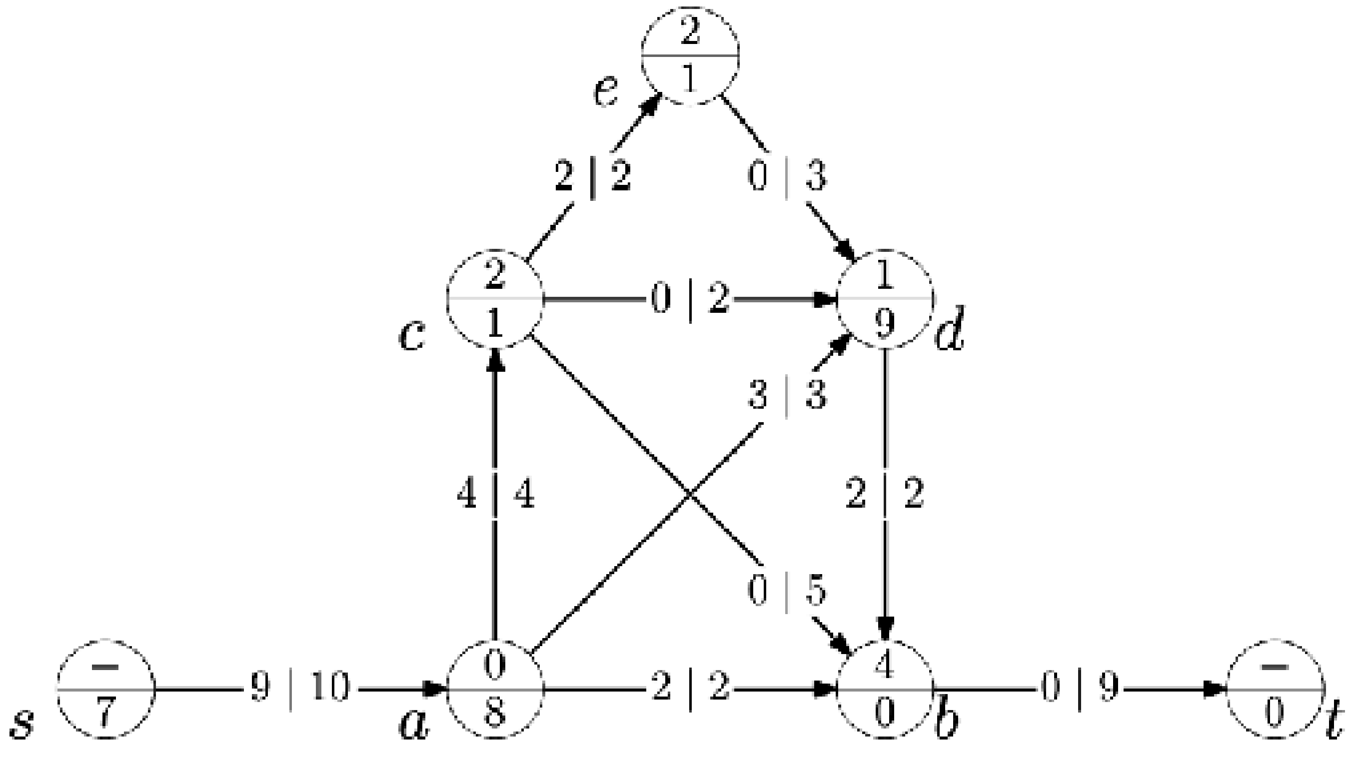


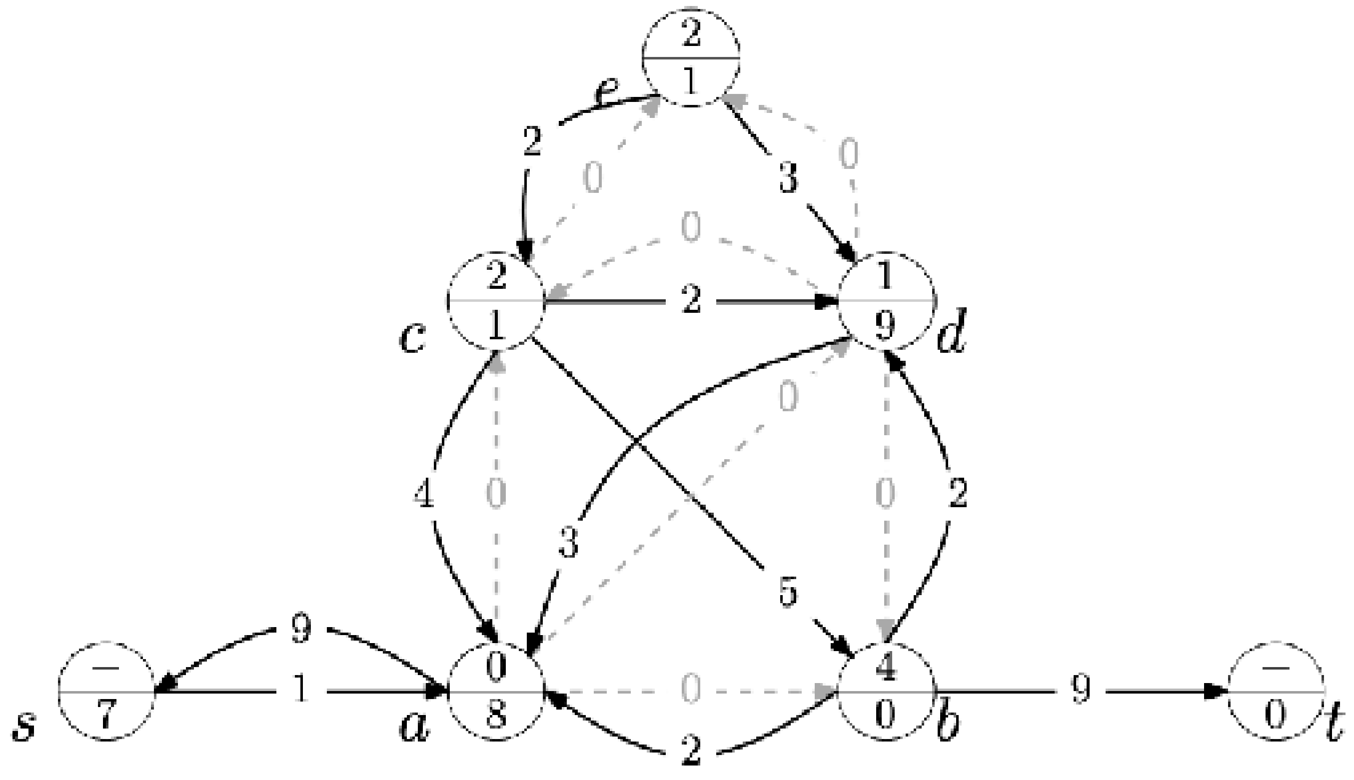
Flow sent from d to c and e:



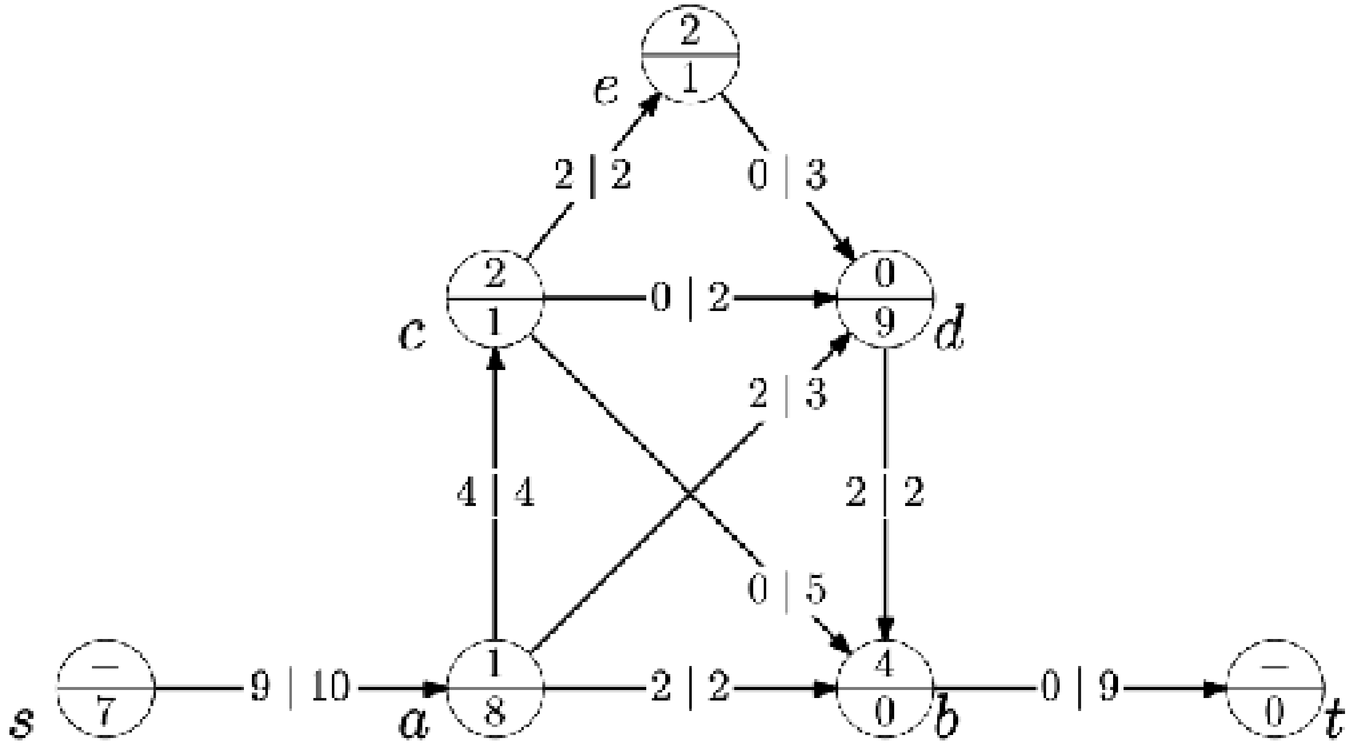


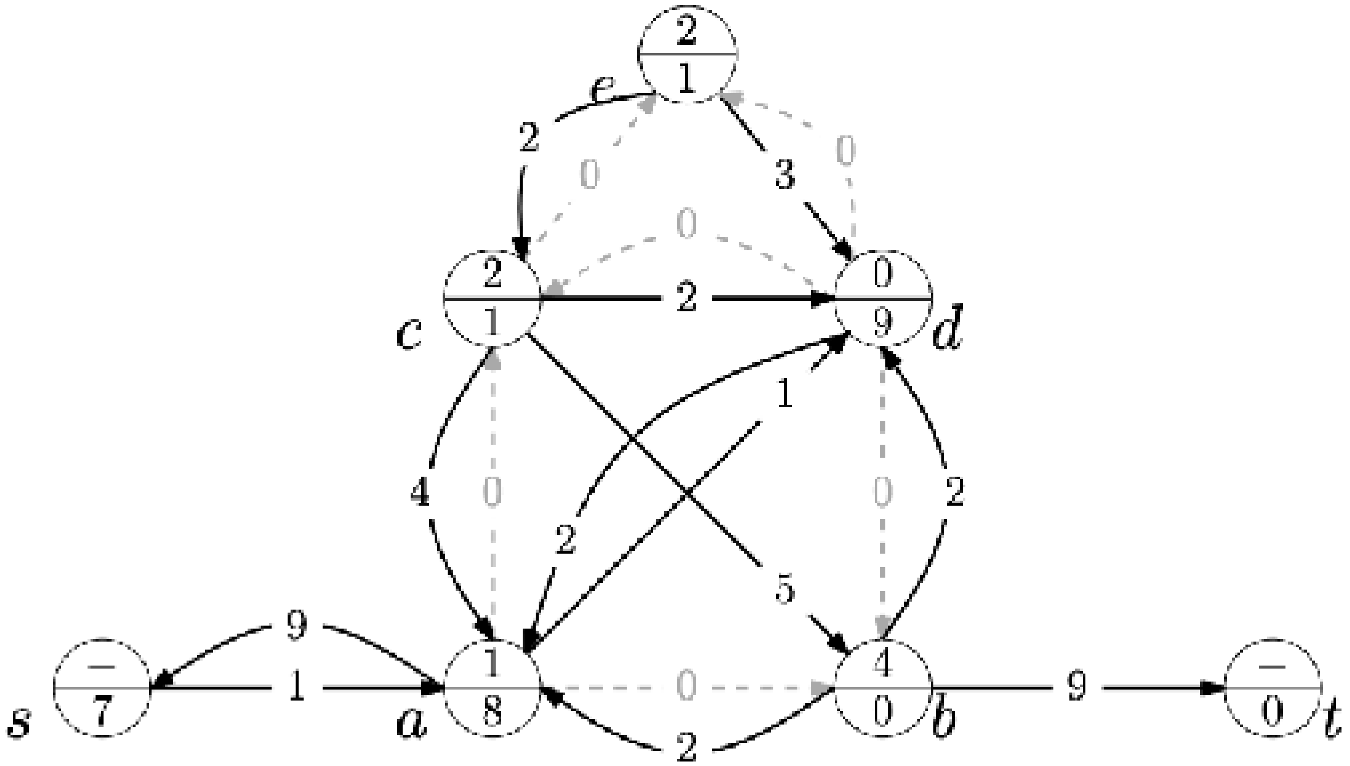
Level of d increased to 9:



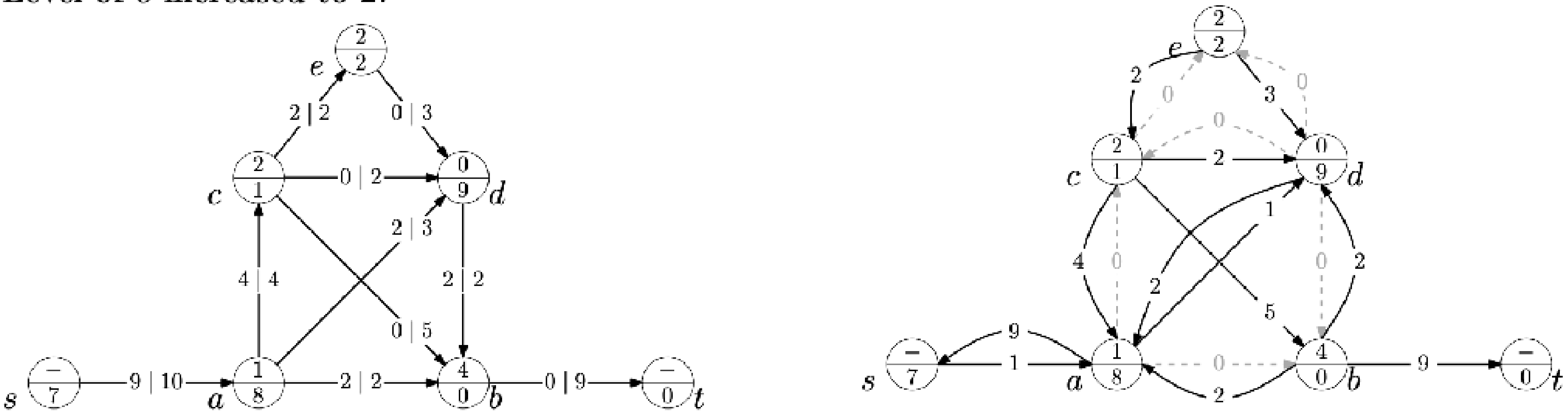


Flow sent from d to a:

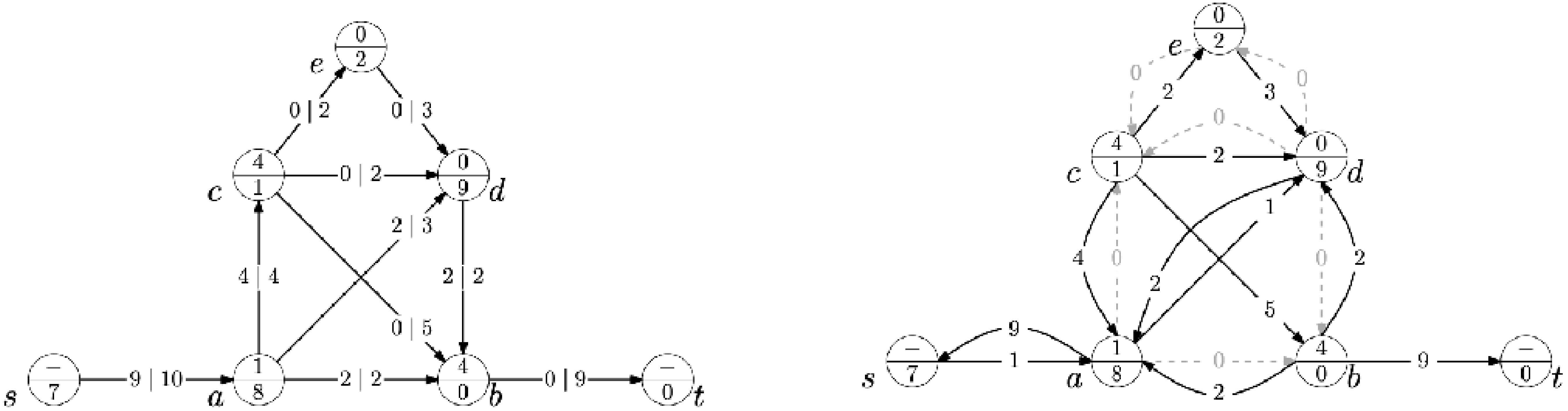




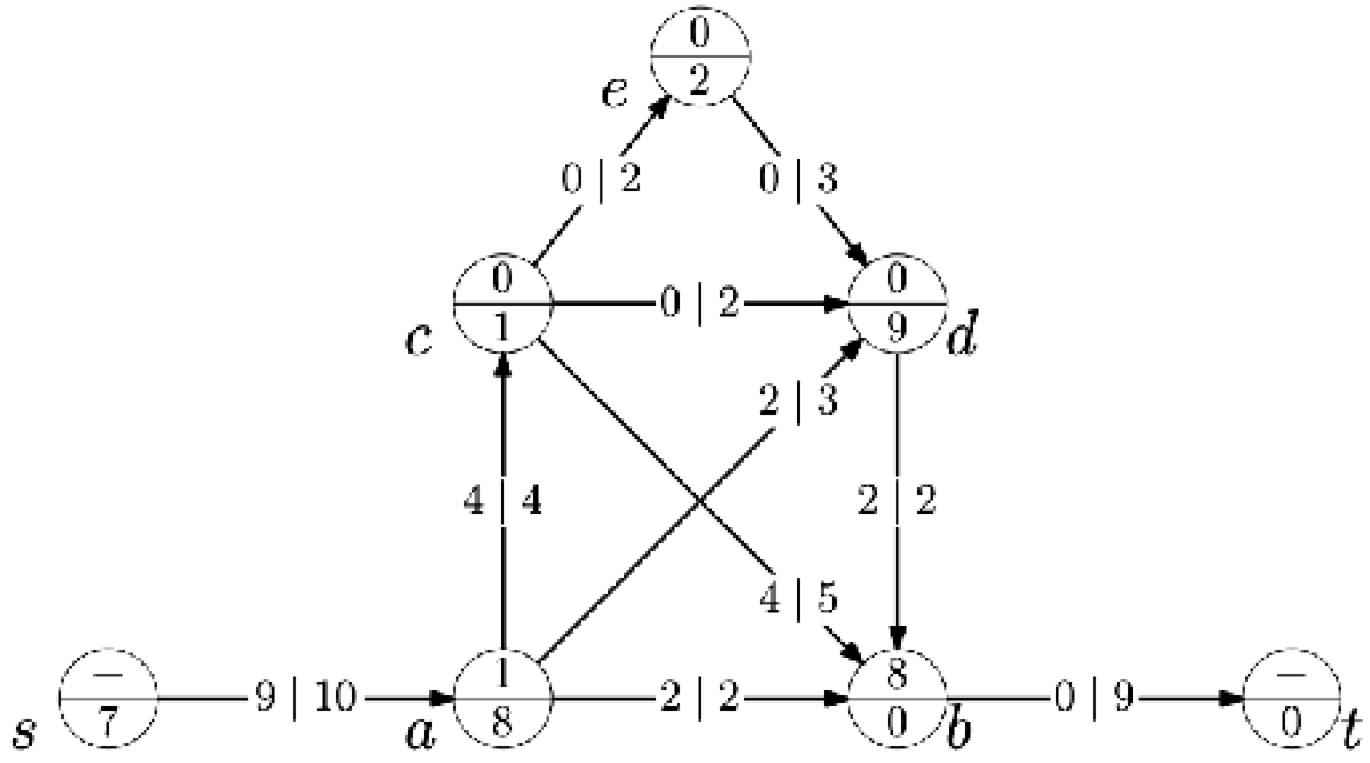
Level of e increased to 2:

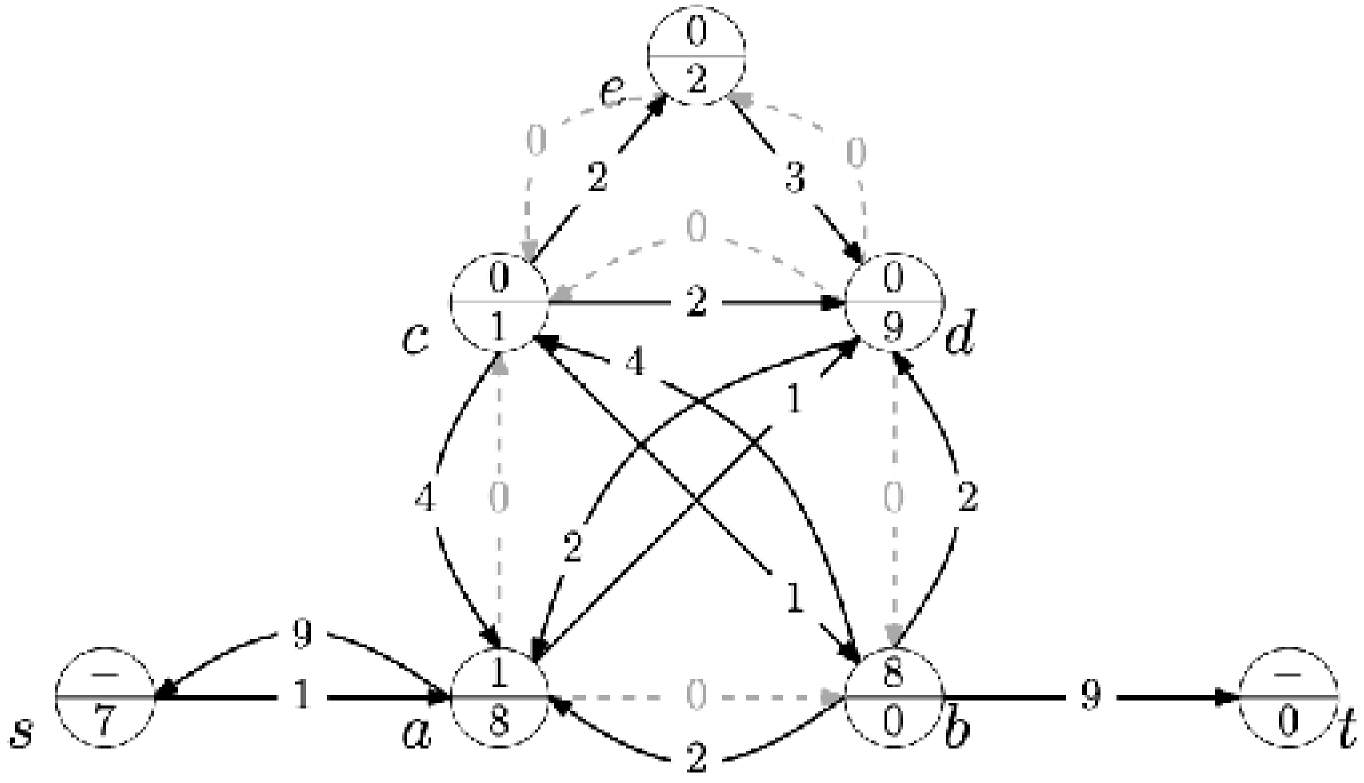


Flow sent from e to c:

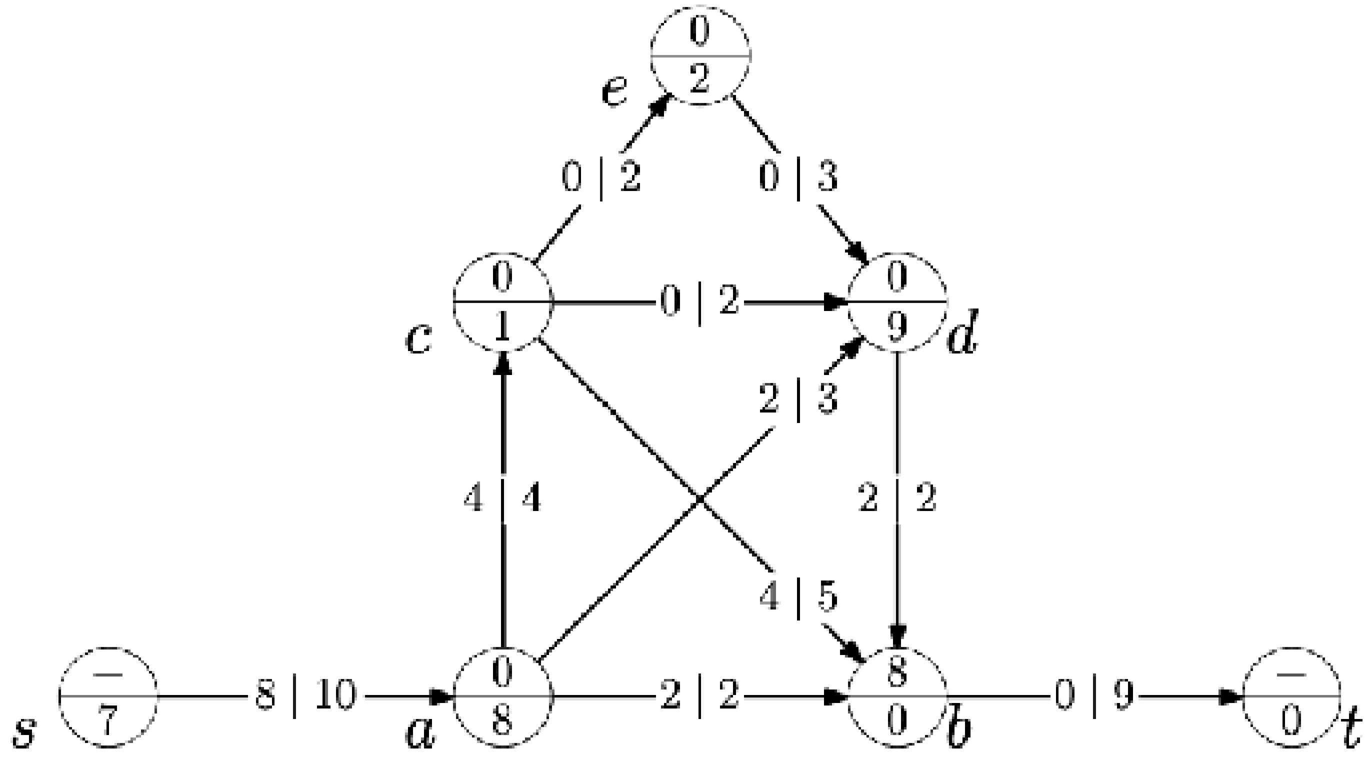


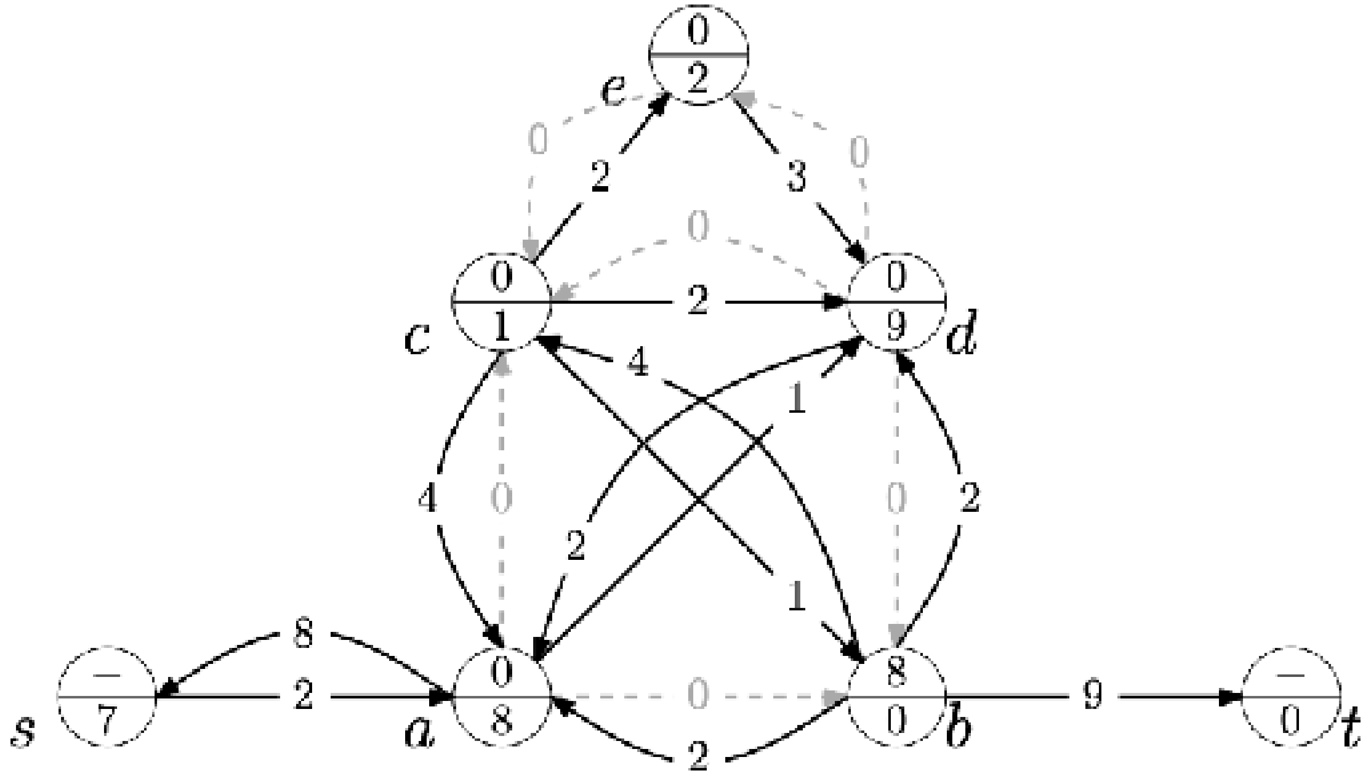
Flow sent from c to b:



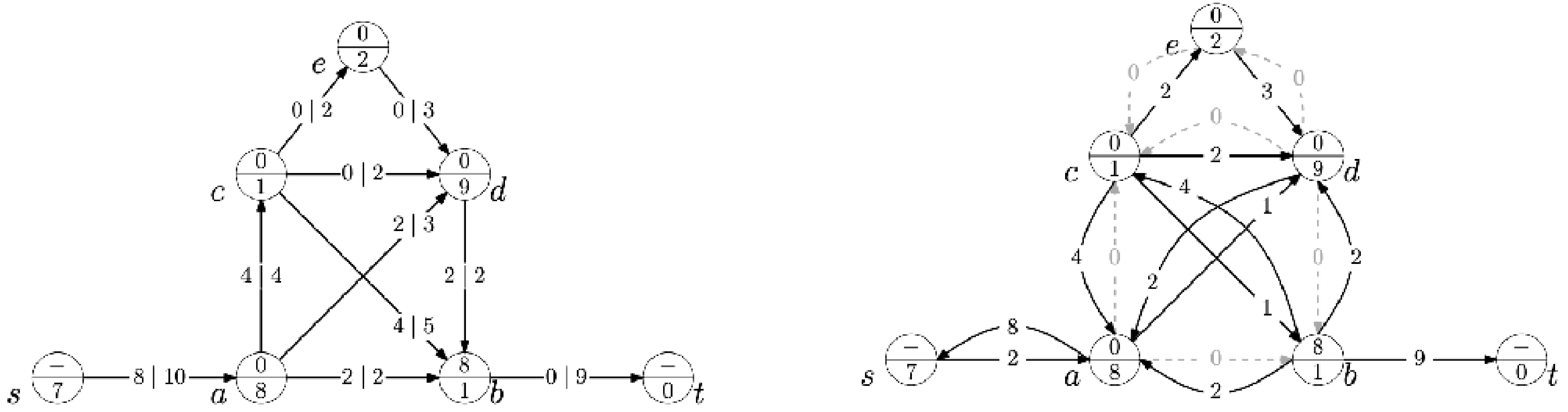


Flow send from a to s:

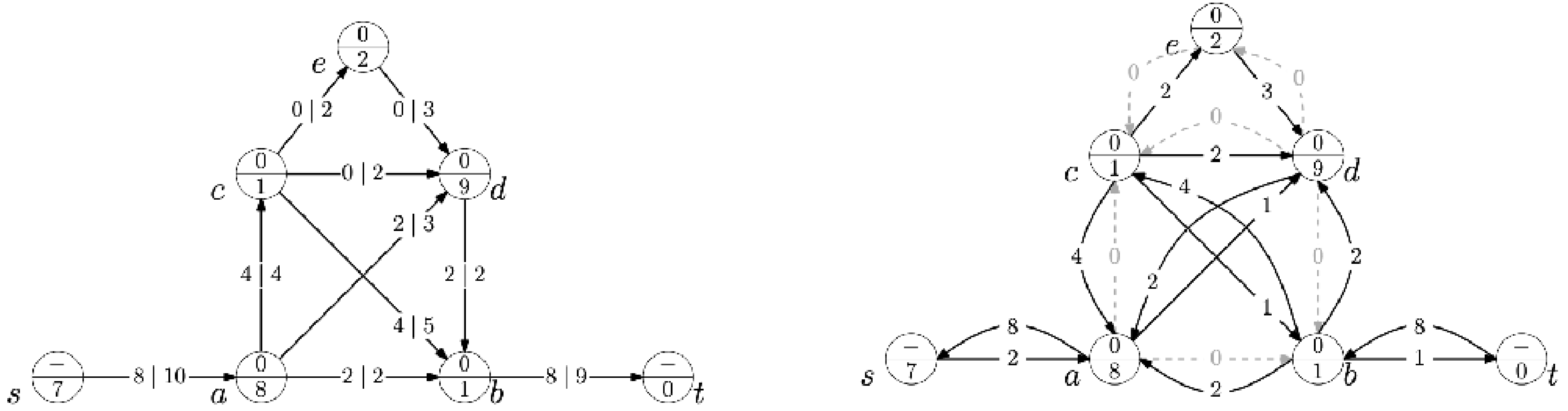




Level of b increased to 1:

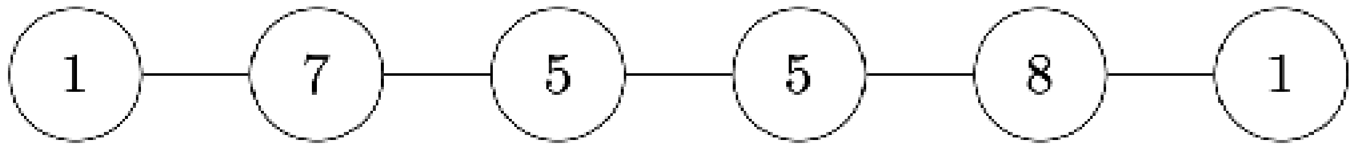


Flow send from b to t:



1. Write down the optimality principles that are important for dynamic programming.

2. Give a maximum independent set U for the following path by marking the nodes that belong to the set U. Moreover, state the overall weight of the set. The node weights are shown within the nodes.



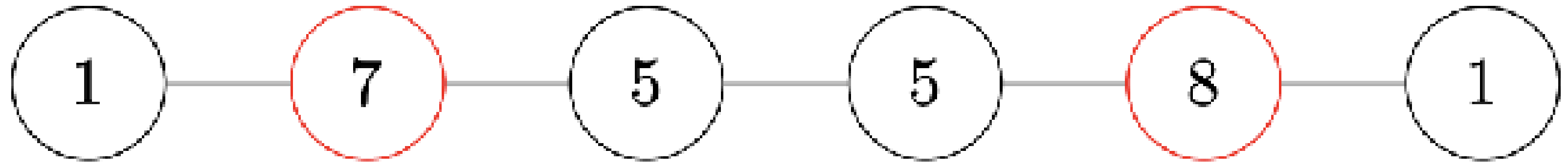
3. Let $m(\ell)$ be the overall weight of the maximum independent set U_ℓ on G_ℓ. Write down a recurrence equation for $m(\ell)$, by using the values $m(i)$ for $i < \ell$! Shortly explain why your equation works!

$$\begin{aligned} m(0) &= \quad 0 \\ m(1) &= \quad c(v_1) \\ m(\ell) &= \end{aligned}$$

4. Outline an algorithm that computes a maximum independent set U on a path $G = v_1 - \cdots - v_k$ having weights $c : V \to \mathbb{N}_{>0}$. Your algorithm has to **compute** and **output** the maximum independent set. The running time of the algorithm should be $O(k)$.

5. Outline a linear-time algorithm that computes a maximum independent set U on a **tree** with node weights $c : V \to \mathbb{N}_{>0}$. Your algorithm has to **compute** and **output** the maximum independent set.

1.
   - Optimal Substructure Property
   - Overlapping Subproblems
2. Optimal Independent Set weight: 15.



3. $m(l) = \max(m(l-2) + c(v_l), m(l-1))$

   As the graph is a path, vertex v_l is only connected to vertices v_{l-1} and v_{l+1}. Therefore, vertex v_l can be in any independent set of vertices 1 to l that does not contain v_{l-1}. If vertex v_{l-1} is part of the independent set, v_l can not be, as they are incident. Therefore the optimal solution for all vertices between 1 and l is the maximum of $m(l-2) + c(v_l)$ (v_l is part of optimal solution) and $m(l-1)$ (not part of optimal solution). The optimal solution for $m(1)$ will always contain v_1, therefore the initialization is correct.

4. We use the equation of 4.3 to compute $m(l)$ for each vertex in the path and return $m(n)$, which is the optimal solution for the whole path. We also use a decision variable x_i which indicates whether v_i is part of this partial solution. We then print all elements that are part of the maximum independent set by calling the algorith moutput(n).

$m(0) \leftarrow 0$;
$m(1) \leftarrow c(v_1)$;
$x_1 \leftarrow$ True;
for $l \leftarrow 2, \dots, n$ **do**
    $m(l) \leftarrow \max(m(l-2) + c(v_l), m(l-1))$;
    if $m(l-2) + c(v_l) \geq m(l-1)$ **then**
        $x_l \leftarrow$ True
    else
        $x_l \leftarrow$ False
    end
end

Algorithm 1: Compute m and x

if $l = 0$ **then**
    return;
else
    if x_l **then**
        print v_l;
        output$(l-2)$;
    else
        output$(l-1)$;
    end
end

Algorithm 2: output(l)

5. The optimal solution in a tree can be found by using the optimal solution of its subtrees. There are again two cases for a vertex v: either it is part of the maximum independent set or it is not. To find the maximum independent set of the subtree rooted in a vertex v, let $m_{\text{out}}(v)$ be the maximum independent set that does not contain v and $m_{\text{in}}(v)$ be the maximum independent set that contains v. They are defined as follows:

$$m_{in}(v) = c(v) + \textstyle\sum_{u \in \text{children}(v)} m_{\text{out}}(u)$$
$$m_{out}(v) = \textstyle\sum_{u \in \text{children}(v)} \max\{m_{\text{out}}(u), m_{\text{in}}(u)\}$$

We compute $m_{\text{out}}(v)$ and $m_{\text{in}}(v)$ for each vertex v from the leaves to the root of the tree and output the maximum of $m_{\text{out}}(\text{root})$ and $m_{\text{in}}(\text{root})$. We use decision variables x_i to denote whether v_i is in this solution. We then call treeOutput(root).

```
Q ← empty queue;
for v ← leaves do
    m(v) = m_in(v);
    x_v ← True;
    if parent(v) is undiscovered then
        Q.insert(parent(v))
    end
end
while Q not empty do
    v ← Q.front();
    m(v) = max(m_out(v), m_in(v));
    if m_in(v) ≥ m_out(v) then
        x_v ← True;
    else
        x_v ← False;
    end
    if parent(v) is undiscovered then
        Q.insert(parent(v))
    end
end
```

Algorithm 3: Compute m and x

```
if l is leaf then
    print v_l;
    return;
else
    if x_l then
        print v_l;
        for i ← children of l do
            for k ← children of i do
                treeOutput(k);
            end
        end
    else
        for i ← children of l do
            treeOutput(i);
        end
    end
end
```

Algorithm 4: treeOutput(l)