# Homework 4

Obliviate

April 2021

## 1 Solution:

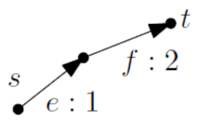
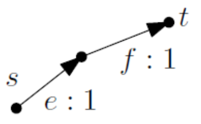
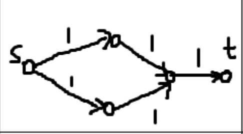
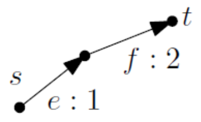
Since there exists a flow from $s$ to $r$ of value $k$, the maximum flow from $s$ to $r$ is at least $k$, which implies that the capacity of any $s$-$r$ cut is at least $k$. The same holds for $r$-$t$ cut. Since any $s$-$t$ cut is either an $s$-$r$ cut or an $r$-$t$ cut, the capacity of it is at least $k$. Equivalently, the maximum flow from $s$ to $t$ is at least $k$, which implies that there is a flow from $s$ to $t$ of value $k$.

## 2 Solution:

1.
(i)always full: e
(ii)optionally full: b c d f g h i
(iii)never full: a

2.
(i)always crossing: e
(ii)optionally crossing:
(iii)never crossing: a b c d f g h i

## 3 Solution:

| The edge $e$ is: | $x$: always full | $y$: optinally full | $z$: never full |
|---|---|---|---|
| $x'$: always crossing |  | impossible | impossible |
| $y'$: optionally crossing |  | impossible | impossible |
| $z'$: never crossing | impossible |  |  |
| | | | |

All we need to prove is that an edge $e$ crosses at least one of minimum cut if and only it's full in every maximum flow.

*Proof.*

(Necessity) Suppose that $e$ is a crossing edge in a minimum $s$-$t$ cut $S$. Then for every maximum flow $f$,

$$\sum_{u\in S}\sum_{v\in V\backslash S} c(u,v) = \text{cap}(S, V\backslash S) = \text{val}(f) = \sum_{u\in S}\sum_{v\in V\backslash S} f(u,v)$$

Since $f(u,v) \le c(u,v)$, $f(u,v)$ must be equal to $c(u,v)$. Hence $e$ is full in $f$.

(Sufficiency) Suppose that $e$ is full in every maximum flow, of which value is denoted by $F$. We replace $c(e)$ by $c(e) - \varepsilon$, where $\varepsilon$ is a sufficiently small number, to construct a new network.

Then in the new network, the value of maximum flow must be $F - \varepsilon$, otherwise $e$ cannot be full in every maximum flow of original network.

By the maxflow-mincut theorem, the minimum cut of the new network is also $F - \varepsilon$. Since $\varepsilon$ is sufficiently small, every minimum cut of the new network must contain $e$. Then it's easy to see that every minimum cut of the new network is also a minimum cut of the original network, which implies that $e$ is a crossing edge in at least one minimum cut of original network. □

**4** Solution:

Suppose $c(e_1) \le c(e_2) \le \cdots \le c(e_m)$.

We use an array $vis[]$ to denote whether a vertex is reachable from $s$. More specifically, $vis[x] = True$ means $x$ is reachable from $s$ and $vis[x] = False$ otherwise.

In the $i$-th iteration, we add $e_{m-i+1}$ to the graph and consider changes of $vis[]$. When $e_i.from$ is reachable from $s$ and $e_i.to$ isn't, we should update $vis$ value of $e_i.to$ and other vertices which are reachable from $e_i.to$. We use DFS(or BFS) to ensure for every vertex $x$ we only update $vis[x]$(from False to True) at most once in the whole algorithm. Our recursive algorithm returns when $vis[t] = True$. And the answer is the cost of last edge we have added.

The correctness of this algorithm is obvious. Suppose the answer of MCP is $c^*$. That is to say, $c^*$ is the maximum value satisfying that $s$ can reach $t$ by using edges with $c(e_i) \ge c^*$.

In DFS, we visit a vertex if and only if we will change its $vis$ value. We visit an edge if and only if we just changed $vis[e_i.from]$. The time complexity of BFS is $O(n+m)$ and our total complexity is obviously $O(n+m)$.

The pseudocode can be found at the end of the assignment.

**5** Solution:

We use $len_i$ to denote the number of candidates for bottleneck before the $i$-th iteration. Obviously, $len_1 = m$. In the $i$-th iteration $i$, we apply median of medians recursively to divide the edge set into $2^{2^{k_i}}$ parts of approximately equal sizes and label them in increasing order. Then we replace the capacities of edges in each set by its corresponding set label and use the algorithm in Exercise 4 to determine which division $c^*$ is in. $len_{i+1} = len_i / 2^{2^{k_i}}$. Finally, there will be only one possible value of $c^*$ left.

The time complexity of median of medians in the $i$-th iteration is $t(i) = O(len_i \times 2^{k_i})$.

Let $k_1 = 0$. Then $t(1) = O(len_1 \times 2^{k_1}) = O(m)$. If $k_{i+1} = 2^{k_i} + k_i$, then $t(i+1) = O(len_{i+1} \times 2^{k_{i+1}}) = O(len_i \times 2^{k_i}) = t(i)$. We can prove $t(i) = O(m)$ by induction.

There are $O(\log^* m)$ iterations because $k$ increase exponentially. The total time is $O(m \log^* m)$. It's better than $O(m \log\log m)$.

The pseudocode can be found at the end of the assignment.

**6** Solution:

See Exercise 5.

**Algorithm 1:** Edges Sorted MCP

```
Function Initialize():
    Clear all edges in the graph;
    for $i=1$ to $n$ do
        vis[i] ← False;
    end
    vis[s] ← True;
Function DFS($pos$):
    vis[pos] ← True;
    for $y$ : out edges of $pos$ do
        if $vis[y.to] = False$ then
            DFS($y.to$);
        end
    end
Function Solve($edge$):
    AddEdge($edge.from$, $edge.to$);
    if $vis[edge.from] = True$ and $vis[edge.to] = False$ then
        DFS($edge.to$);
    end
Function EdgesSortedMCP():
    Initialize();
    for $i=m$ to $1$ do
        Solve($e[i]$);
        if $vis[t] = True$ then
            return $e[i].c$ ;
        end
    end
```

**Algorithm 2:** MCP

```
Function FindDivision(divisions, OldEdges):
    Initialize();
    for edge e in OldEdges do
        Solve(e);
    end
    for division d in divisions do
        for edge e in d do
            Solve(e);
        end
        if $vis[t] = True$ then
            return d, OldEdges;
        end
        for edge e in d do
            Append e to OldEdges;
        end
    end
Function MCP():
    E ← $\{e_1, e_2, \cdots, e_m\}$;
    OldEdges ← $\emptyset$;
    k ← 0;
    while |E| > 1 do
        divisions ← {E};
        for i= 1 to $2^k$ do
            nextDivisions ← $\emptyset$;
            for division w in divisions do
                u ← MedianOfMedians(w);
                Append $\{c(e_i) \geq u, e_i \in w\}$ to nextDivisions;
                Append $\{c(e_i) < u, e_i \in w\}$ to nextDivisions;
            end
            divisions ← nextDivisions;
        end
        E, OldEdges ← FindDivision(divisions, OldEdges);
        k ← $k + 2^k$;
    end
    return the capacity of edge in E;
```