Homework 4

CSOR W4231 - Analysis of Algorithms

Chandan Suri

UNI - CS4090

Collaborators: Arnavi Chedda (amc2476), Parth Jawale (pcj2105), Gursifath Bhasin (gb2760)

Submitted on -

April 18, 2022

Homework 4

Problem 1

(a)

Problem Description:
Input: A randomized algorithm using the k-th order statistic algorithm that returns the the median from a set S of n distinct integers. So, the input is just the set S of n distinct integers.
Output: Returns the median of S using this k-th order statistic algorithm.

Description of the Algorithm:
The original k-th order statistic algorithm finds the k^{th} smallest element in the set of elements S. Thus, to find out the median of the array, we basically need to find the middle element in the array of the elements involved when they are represented as ordered elements (ascending order). As we follow 1-based indexing here and the number of elements can be either even or odd, we need to consider the computation of median as follows:

1. If the number of elements in the array is even, then we don't exactly have a middle element, so, we would need to call the k-th order statistic function twice and return the mean of those values returned by the function calls.
2. If the number of elements in the array is odd, then we do have a middle element when we place the elements in ascending order (ordered) so we would need to call the k-th order statistic function once and return the value returned by the function.

Following the cases above, we would need to pass the value of k depending on the conditions stated below to get the median using the algorithm provided to us:

$$\text{Median}(S) = S[\frac{n+1}{2}] \text{ if n is odd}$$

$$\text{Median}(S) = \frac{S[\frac{n}{2}] + (S[\frac{n}{2}+1]))}{2} \text{ if n is even}$$

where, S = Ordered list of values
n = Number of values

Pseudocode of the Algorithm:

Algorithm 1: Algorithm to determine the median of S.

```
Function DetermineMedian($S$):
    $len \leftarrow |S|$;
    if $len \% 2 == 0$ then
        $first\_mid\_elem$ = $k$-th_order_statistic(S, $\frac{len}{2}$) ;
        $sec\_mid\_elem$ = $k$-th_order_statistic(S, $\frac{len}{2} + 1$) ;
        return $\frac{first\_mid\_elem+sec\_mid\_elem}{2}$;
    else
        return k-th_order_statistic(S, $\frac{len+1}{2}$);
```

This algorithm is correct as we are just using the function provided to us and using the formulation for calculating the median using that function.

(b)

To analyze the expected running time of the k-th order statistic method provided to us:
We can say that a sub-problem is of type j if its input consists of at most $n(\frac{3}{4})^j$ items but more than $n(\frac{3}{4})^{j+1}$ items. This has been provided to us by the hint and is coming from the probability theory. According to the probability theory, the Central Limit Theorem is related to the concept of a central element. In turn, this central element is the element which has at least a quarter of the elements larger than it and the same number of elements smaller than it. Precisely, this gives us the ratio of $\frac{3}{4}$ above.

Now, we need to find an element a_i such that it is the central element. Following this, at least a quarter of the elements will be thrown away. This would mean that the set of values will shrink by a factor of $\frac{3}{4}$ or lesser, and the current control in the recursive call will be returned which will end the current iteration.

As we know that half of the elements of the array are considered to be the good candidates to become a_i as quarter of the elements from the start and the end are not being taken according to the theory related to the central element. Therefore,

P(choosing an a_i from all set of values) $= \frac{1}{2}$

This , in turn, would mean that the expected number of iterations before a central element is found is at most 2 as the probability of finding such an element is half and in two iterations it should be considered at least once according to the probability theory.

Furthermore, let Y be a random variable which denotes the number of steps taken by the algorithm.

$$Y = Y_0 + Y_1 + Y_2 + ...$$

where Y_j is the expected number of steps spent in phase j. As we told above, when the algorithm is in phase j, the number of values in the set is at most $n(\frac{3}{4})^j$. As all the operations in the kth_order_statistic are constant time operations other than the loop where we add the elements in 2 new sets and the recursive call to the function itself. Thus, the number of steps required for one iteration in phase j is at most $cn(\frac{3}{4})^j$ for some constant c due to the looping structure in each iteration.

From probability theory, we know that the Expectation E[Y] is given by:

$$E[Y] = \sum_j j \cdot \Pr[Y = j]$$

Now, we need to fund the upper bound of the expected running time of the entire algorithm under consideration. For the same, we will have to sum up the expected time spent on every sub-problem or iteration. Also, as discussed before, the expected number of iterations in phase j is at most 2. Therefore,

$$E[Y] = \sum_j E\left[Y_j\right]$$
$$E[Y] \leq \sum_j 2cn\left(\frac{3}{4}\right)^j$$
$$E[Y] \leq 2cn\sum_j \left(\frac{3}{4}\right)^j$$

In the above formulation, we have taken out the $2cn$ which is basically a constant which can be outside the summation sign as that doesn't depend on phase j.

Also, in the equation above:
$\sum_j \left(\frac{3}{4}\right)^j$ is the formulation for the sum of a geometric progression. In the GP here, we have $a = 1$ and the common ratio $r = \frac{3}{4}$. Since j can potentially be very large (tending to ∞), the sum of this GP where each term if of the form ar^{n-1} is given by:

$$S = \frac{a}{1-r}; -1 < r < 1$$

NOTE: This is only applicable when we can converge to a sum and as the ratio here is in the bounds provided for the common ratio (r) above, we know that sum of the GP in our case will also converge. Thus,

$$S = \sum_{n=1}^{\infty} 1 \cdot (\frac{3}{4})^{j-1}$$
$$S = \frac{1}{1-\frac{3}{4}}$$
$$S = 4$$

Substituting this in the inequality for expected running time above, we get:

$$E[Y] \leq 2cn \cdot 4 \leq 8cn$$

Lastly, as we now know the upper bound for the expected running time, we can just drop the constants. Thus, the expected running time of this algorithm is of the form $O(n)$ (linear time).

Problem 2

Problem Description:
Input: We are given a flow network G = (V,E,s,t,c) with integer capacities, an integral max flow f^*, a residual graph G_{f^*}. Also, we are given the edge $e \in E$ whose capacity is reduced by 1 unit.
Output: Returns the new maximum flow for the new flow network with the updated edge and flow and also the updated residual graph.

Solution:
Description of the Algorithm:
When we reduce the capacity of one of the edges in the graph by 1 unit, the max flow is not bound to change. The cases would be as follows:

1. When $f(e) < c(e)$ where f denotes the flow and c denotes the capacity of an edge $e \in E$, if we reduce the capacity by 1, the max flow wouldn't change. This is because the flow wouldn't have to be reduced as the flow doesn't get more than the capacity even though the capacity is reduced by 1.

2. When the edge is saturated such that $f(e) = c(e)$ and we decrease the capacity by 1, then we would need to remove one unit of flow from s to t where s is the source and t is the sink that goes through the edge $e = (u, v)$.

Mainly, for the second case above, we will have to run our algorithm. For which, we have broken up the process in 2 parts as follows:

1. We would need to remove the flow from the s-t path that involves that edge $e = (u, v) \in E$ for which we had reduced the capacity by 1. For this purpose, we run the Breadth First Search (BFS) twice. Once, we start from s and go until u going through the edges that had flow greater than zero. For the second one, we start from v and go until t going though all the edges that had a flow greater than zero.
   Basically, for the path we found from s to u, we reduce the flow for all the edges by 1 unit and do the same for all the edges encountered in the path from v to t. After doing so, we have reduced the flow of all the edges in the s-t path.

2. Now, we will need to recover the max flow. We need to do so because we have a valid flow network but not necessarily the network (with the residual graph) with max flow. Now, we can just run the augment subroutine of the "Ford-Fulkerson's" algorithm to update the max flow and the residual graph. Only one iteration is needed here as we have just reduced 1 unit from the capacity for only 1 edge in the flow network.

Once, we have done the above two steps, we now have the new max flow, along with updated graph and the residual graph.
NOTE: I would also like to point out that the function "UpdateFlowsWithBFS" has been adapted from the "BFS" function taught in the class with some minor changes.

Pseudocode:

Algorithm 2: Returns the updated maximum flow after the capacity of one of the edges is reduced by 1 unit.

```
Function UpdateFlowsWithBFS($G = (V, E, s, t, c), start\_node, end\_node, f^*$):
    $discovered[V] \leftarrow 0$;
    $queue\ q$;
    $discovered[start\_node] \leftarrow 1$;
    $enqueue(q, start\_node)$;
    while $size(q) > 0$ do
        $u = dequeue(q)$;
        if $u == end\_node$ then
            $break$;
        for $(u, v) \in E$ do
            if $discovered[v] == 0$ then
                $discovered[v] = 1$;
                $enqueue(q, v)$;
                $f^*(u, v) = f^*(u, v) - 1$;
        end for
    end while
Function findUpdatedMaxFlow($G = (V, E, s, t, c), f^*, G_{f^*}, e = (u, v)$):
    if $f^*(e) == c(e) + 1$ then
        $UpdateFlowsWithBFS(G = (V, E), s, u, f^*)$;
        $UpdateFlowsWithBFS(G = (V, E), v, t, f^*)$;
        $path = simple\ s - t\ path\ in\ G_{f^*}$;
        $updated\_max\_flow = Augment(f^*, path)$;
        return $updated\_max\_flow$;
```

Correctness of the Algorithm:
First of all, we only run our algorithm if $f^*(e) == c(e) + 1$. This, in turn, updates the maximum flow. At first, we are running the BFS algorithm twice to update the flows for the edges in the s-t path. Basically, to conserve the flow, as we have already reduced the capacity and flow of one of edges in the s-t max flow path, we would need to update the flows of all the connections to and fro that updated edge e. We do the same, once for $s - u$ path and then for $v - t$ path. This basically just updates all the edges that have some connection to the updated edge and would be along the s-t path. We have adapted this code from BFS and thus, should run in an expected manner in order to find u when running from s, and to find t, when running from v. We have just added the code to update the flow to conserve the flow which is indeed correct.

Once the flow in the graph is a valid one, we will have to find the updated maximum flow. For this purpose, we just call the *Augment*() function. As this is a correct sub routine which is the part of the Ford-Fulkerson's algorithm and is known to update the residual graph and find the maximum flow if only one step is needed to find the maximum flow. As we had just updated one of the edges by reducing the capacity by 1, we only need one more iteration in order to find the max flow and update the residual graph accordingly. This routine runs in $O(m+n)$ time and returns the maximum flow in the network.

As we have correctly proven all the parts of our algorithm that uses tested algorithms like BFS and Ford-Fulkerson, we have indeed proven the correctness of our algorithm for our particular use case here.

Time Complexity Analysis: We will consider the two functions $UpdateFlowsWithBFS()$ and $findUpdatedMaxFlow()$ separately:
For the $UpdateFlowsWithBFS()$ function:
Lines 3-5 will take constant time as these are just assignment operations or union operations.
Also, we can consider line 2 to take $O(n)$ time as it creates an array with as many elements as the number of vertices (n) in the graph.
Lines 6 is a while loop that would run for n number of times in the worst case, if it's completely disconnected graph (albeit highly unlikely).
Other operations on the lines 7-9 and are just constant time operations like dequeue and comparison operations.
Line 10 has a for loop that goes over all the edges that start from the vertex u. If it runs for various vertices in the graph, it basically goes over all the edges of the graph just once because of the condition on the line 11 which is also a constant time operation.
Lines 12-14 are just constant time operations as these are just update and assignment operations.
Thus, this makes the for loop at line 10 take $O(m)$ time in worst case where m is the number of edges in the graph.
Also, the while loop at line 6 takes $O(m+n)$ time in worst case because of the for loop inside it.
This makes the worst case time complexity of this function to be $O(m+n)$.

For the $findUpdatedMaxFlow()$ function:
Line 18 is just a comparison operation.
Lines 19-20 are basically two calls to the BFS function which takes $O(n+m)$ time respectively.
Line 21 just assigns a simple s-t path in G_{f^*} to "path" variable which takes constant time.
In line 22 we just call Augment() function which takes $O(m+n)$ time in worst case, as discussed in class.
Line 23 is just a return statement which is a constant time operation.
Thus, $findUpdatedMaxFlow()$ function takes $O(m+n)$ time in the worst case.
Thus, our algorithm essentially takes $O(m+n)$ time (in the worst case) considering both the parts of the functionality as mentioned above.

Problem 3

Given/Input:
We are given a flow network with potentially multiple sources and sinks, which might have incoming and outgoing edges respectively.
Each node $v \in V$ has an integer supply s(v) with the following conditions:

1. if s(v) $>$ 0, v is a source.
2. if s(v) $<$ 0, v is a sink.

Also, let S be the set of source nodes and T be the set of sink nodes. Furthermore, a circulation with supplies is a function $f : E \longrightarrow R^+$ that satisfies:

1. capacity constraints: For each $e \in E$, $0 <= f(e) <= c(e)$.
2. supply constraints: For each $v \in V$, $f^{out}(v) - f^{in}(v) = s(v)$

(a)

Solution:
To derive a necessary condition for a feasible circulation with supplies to exist. (a decision problem): Firstly, I would like to point out that for a feasible circulation to exist here, the flow should be conserved.
Now, we are given that $f^{out}(v) - f^{in}(v) = s(v)$ where s is the supply for a vertex v. Supposing that there is a feasible circulation f, if we consider all the vertices in the graph:

$$\sum_{v \in V} s(v) = \sum_{v \in V} f^{out}(v) - f^{in}(v)$$

In the formulation above, every edge (u, v) appears twice: once as an outgoing edge from u and once as an incoming edge into v. Hence,

$$\sum_{v \in V} s(v) = 0$$

Furthermore, by the fact above, we know that summation of all the $f^{out}(v)$ must be equal to the summation of all the $f^{in}(v)$ as it follows from the above statement that:

$$\sum_{v \in V} f^{out}(v) - f^{in}(v) = 0$$

Distributing the flow and following the law of flow conservation, we can rewrite the above expression as:

$$\mid \sum_{v:s(v)>0} s(v) \mid = \mid \sum_{v:s(v)<0} s(v) \mid$$

We can also rewrite the above necessary condition for a feasible circulation with existence of the supplies as :

$$\sum_{v:s(v)>0} s(v) = - \sum_{v:s(v)<0} s(v)$$

Thus, we have found our final necessary condition as needed.

(b)

Solution:
To reduce the problem of finding a feasible circulation with supplies to the max flow problem, basically we need to convert this decision problem to a maximization one. To do so, we will follow the steps below to convert the original graph G accordingly into a graph $G^{'}$ as follows (the following steps will be in polynomial time altogether):

1. Add a super source sv^* and a super sink tv^*.
2. Add outgoing edges (from sv^* to sv)for each of the nodes in S ($sv \in S$), with the capacity $s(sv)$.
3. Add outgoing edges (from tv to tv^*) for each of the nodes in T ($tv \in T$), with capacity $-s(tv)$.

After this, we would run the "Ford-Fulkerson's" algorithm in the newly created graph $G^{'}$. Also, after following the above steps, a graph G would be converted to $G^{'}$ as shown in the figures below. The first figure shows the original graph G and the second one shows the updated graph $G^{'}$ with the super source node sv^* and super sink node tv^*. The nodes a, b and c were the source nodes with supplies greater than zero. While, the nodes d, f and g were the sink nodes with supplies less than zero. Thus, as you can see, we add the connections from the super source node to all the source nodes and from the sink nodes to the super sink node.

Time Complexity Explanation: In the step 1, we basically just add sv^* and tv^* to the set of vertices (V) in the graph. This must take constant time.
Furthermore, in step 2, we just try and find all the source nodes in the graph and add edges from the newly added super source node sv^* with the flow and capacity equal to the supply of the source node ($sv \in S$) in the graph. As for this, we might need to go over all the

Homework 4

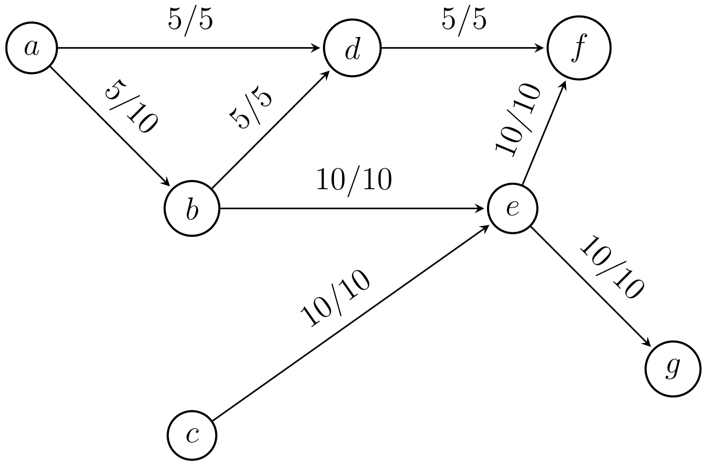

Figure 1: Original Graph G with multiple sources and sink

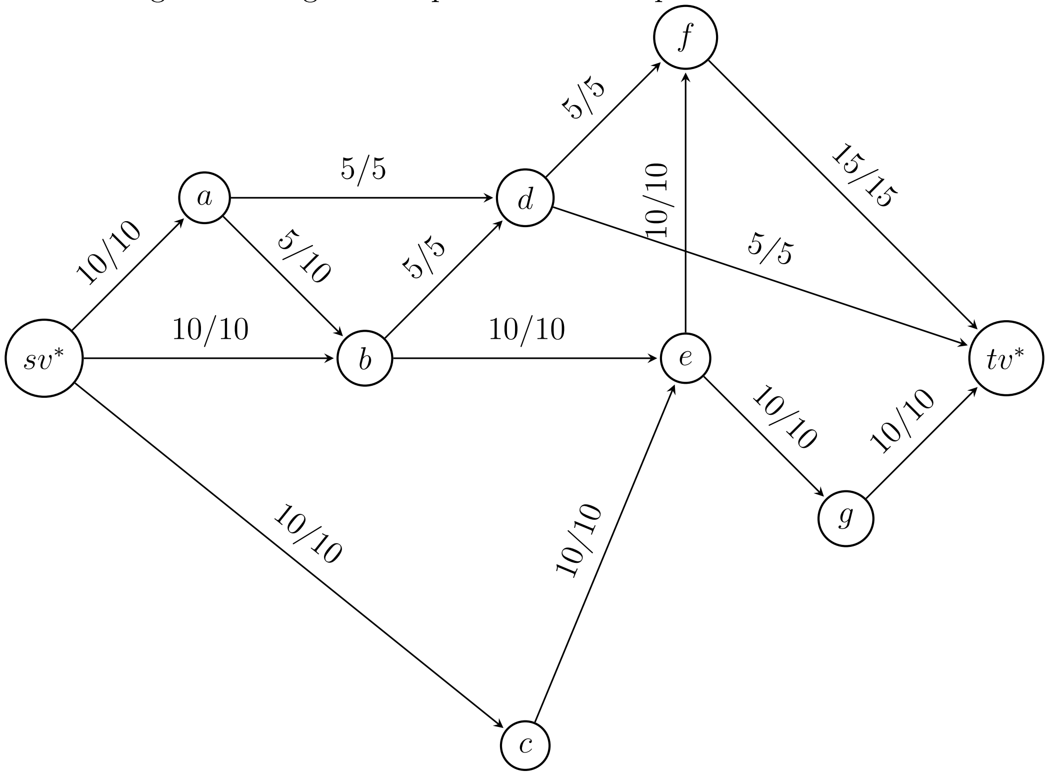

Figure 2: Updated Graph G' with super source and super sink nodes.

vertices and edges in the graph, this takes linear amount of time in terms of the number of edges and vertices in the graph.
Similarly, in step 3, we just try and find all the sink nodes in the graph and add edges from the sink nodes found to the newly added sink node tv^* with the flow and capacity equal to the supply of the corresponding sink node ($tv \in T$) in the graph. As for this, we might also need to go over all the vertices and edges in the graph, this also takes a linear amount of time in terms of the number of edges and vertices in the graph.
Finally, it takes polynomial time to run the Ford-Fulkerson's algorithm again wherein we try and find the max flow in the flow network formed $G^{'}$.
Thus, the final worst case running time for the complete algorithm is polynomial time as well.

Correctness of the Steps and Proof:
After following all the steps above, this should convert our decision problem into a maximization one now. To prove the correctness of this hypothesis, let's claim that there will be a circulation that satisfies all the supplies in the flow network G iff. there is a max flow of value $\sum_{sv \in S} s(sv)$ in $G^{'}$.
Following this and the feasible condition from the part a (by the law of flow conservation), we know that:

$$D = \sum_{sv \in S} s(sv) = -\sum_{tv \in T} s(tv)$$

Let's first prove the feasibility aspect. Suppose there is a feasible circulation f in G. Following this, in $G^{'}$ we can send a flow $s(sv)$ across the edge (sv^*, sv) for every $sv \in S$. Similarly, we can also send a flow $s(tv)$ across the edge (tv, tv^*) for every $tv \in T$. This should be a valid flow in new network $G^{'}$ as we aren't breaking any of the constraints mentioned.

As there is a single source and sink now and by the law of flow conservation along with the fact that max flow can be computed by the formulation mentioned above (D), the value of the flow will be D. The value of this flow can thus, be easily computed by calculating the capacity of the cuts involved here. Consequently, the value of D will be equal to the capacity of the cut $(sv^*, V - sv^*)$ and also equal to the capacity of the cut $(tv^*, V - tv^*)$. Therefore, it is maximum one as all those edges from the super source node or to the super sink node are saturated and by the applicability of the law of flow conservation.

Now, we will need to prove if we can find a maximum flow in the original flow network. For this, we will go in the opposite direction. Following from the explanation above, we can say that conversely supposing that there is a maximum flow of value D in $G^{'}$. Since, the value of the flow equals the value out of the super source node sv^* and also equal to the capacity of the cut $(sv^*, V - sv^*)$. This, in turn, is equal to D. We are also taking into consideration that every edge out of the super source node sv^* is saturated meaning that the flow of the edges equals the capacity of those corresponding edges.

Similarly, since the value of the flow equals the value into the super sink node tv^* and the capacity of the cut $(tv^*, V - tv^*)$. This, in turn, is also equal to D. We are also taking into consideration that every edge going into the super sink node tv^* is saturated meaning that the flow of the edges equals the capacity of those corresponding edges.

Consequently, this means that if we delete all the (sv^*, sv) and (tv, tv^*) edges that we had added at the start to convert the original flow network G into $G^{'}$, we would get the circulation f such that $f^{out}(v) - f^{in}(v) = s(v)$ for every node v in the network.

As we have reached the premise from the conclusion and vice-versa, hence, we have proved that the method proposed above would convert the decision problem into a maximization one.