Midterm

Contents

1 Problem 1

1.1 Statement

Let $G = (V, E)$ be a complete undirected graph with edge costs satisfying triangle inequality and let k be a positive integer. The problem is to partition V into sets $V_1, \ldots, V_k$ so as to minimize the costliest edge between two vertices of the same set, i.e.

$$\min \max_{1 \le i \le k} \max_{u,v \in V_i} cost(u, v)$$

Give a factor 2 approximation algorithm for this problem together with a tight example.

1.2 Solution

As per a clarification, we have $cost(u, u) = 0$. By the triangle inequality, we have $cost(u, v) + cost(v, u) = cost(u, u) \geq 0$, and since edges are undirected, $cost(u, v) = cost(v, u) \geq 0$.

We assume that $k \leq n$, since otherwise by the pigeonhole principle, at least one of the sets must be empty, and hence the costliest edge between two vertices of the same set won't be defined at all.

We consider the following greedy algorithm:

```
function ApproxPartition($G = (V, E)$, $cost : V \times V \to \mathbb{R}$)
    Let $c[1 \ldots k]$ be an array initialized to all 0s (except $c[1]$, which is 1)    ▷ This stores the "center" of the $i^{th}$ partition (not a center in the traditional sense)
    Let $partition[1 \ldots n]$ be an array initialized to all 1s.    ▷ This stores the partition id of the $i^{th}$ vertex
    Let $vertices[1 \ldots k]$ be an array of linked lists (all but $vertices[1]$ are empty, with $vertices[1]$ initially having all the vertices).
    for $i$ from 2 to $k$ do
        $index \leftarrow -1$
        $distance \leftarrow 0$
        for $p$ from 1 to $i - 1$ do
            for $v$ in $vertices[p]$ do
                if $distance < cost(v, c[p])$ then
                    $index \leftarrow v$
                    $distance \leftarrow cost(v, c[p])$
                end if
            end for
        end for
        Remove $index$ from $vertices[partition[index]]$
        $partition[index] \leftarrow i$
        Insert $index$ into $vertices[partition[index]]$
        $c[i] \leftarrow index$
        for $p$ from 1 to $i - 1$ do
            for $v$ in $vertices[p]$ do
                if $cost(v, c[p]) < cost(v, index)$ then
                    Remove $v$ from $vertices[p]$
                    $partition[v] \leftarrow i$
                    Insert $v$ into $vertices[i]$
                end if
            end for
        end for
    end for
    return $vertices$
end function
```

The high level description of the algorithm is as follows. Initially the only partition consists of all vertices. Suppose we are at iteration i. Consider a vertex which is farthest from the "center" (assigned by the algorithm, not the true center) of its partition. Remove it from this partition, and make a new partition which has this vertex as its assigned "center", and stores only this vertex (for now). Then for each vertex which is closer to this vertex than the "center" of its own partition, move it from its old partition to this partition. Then after $k - 1$ such iterations, we get a partition of V into k sets, and we return it.

Claim 1.1

In the case when $n = k$, the answer returned by the algorithm is optimal.

Proof. Since at any step of the algorithm, there exist two distinct vertices in the same partition by the pigeonhole principle, we never pick $c[p]$ for a partition p at any point. Hence at each step of the algorithm, we do two things:

1. Increase the number of partitions by 1.
2. Don't empty any partition.

Since in the end, we have $k = n$ partitions in this case, and the cost is 0, we are done (since no cost is negative). $\square$

Now suppose $n > k$. Let v be the vertex that would have been picked had we run the algorithm for another iteration (i.e., the vertex which our algorithm would have chosen as the "center" for a new $(k+1)^{th}$ partition).

Suppose this was in a partition p earlier, whose center was $c[p] = x$. Note that since we move a vertex from a partition to another partition if and only if the distance to the "center" of its partition reduces, we can see that the distance from a vertex to the "center" of its partition never increases as the algorithm progresses.

Claim 1.2

In the solution produced by the algorithm, the weight of the costliest edge between two vertices of the same set is at most $2 \cdot cost(v, x)$.

Proof. Now note that v (due to the choice of v) has distance to x as the largest distance from a vertex to a center. So, for any vertex u, we have the distance to its partition's center to be at most $cost(v, x)$. Using the triangle inequality on the triangle $u, u', c[partition[u]]$ for u, u' in the same partition, we get the distance between them (i.e., $cost(u, u')$) bounded above by $cost(u, c[partition[u]]) + cost(u', c[partition[u']]) \leq 2 \cdot cost(v, x)$, and we are done. $\square$

Claim 1.3

$\mathbf{OPT} \geq cost(v, x)$.

Proof. Note that as we mentioned earlier, for each vertex v, $cost(v, c[partition[v]])$ never increases as the algorithm progresses. We claim the following:

Claim 1.4

Suppose the center of the partition which had $c[i]$ before it was moved into partition i was $a[i]$. Then $cost(c[j], c[l]) \geq a$ for all $1 \leq j \neq l \leq i$.

Proof. We do an induction on i. For $i = 1$ and $i = 2$ (when $i = 2$ is valid) there is nothing to prove. Now suppose $i > 2$. Note that whenever we move a vertex from a partition to a new partition, it is done only if the distance to the center of the new partition is smaller than the distance from its old partition's center. Hence, we have $cost(c[i], c[j]) \geq a$ for all $1 \leq j < i$. Now note that at each step, since $cost(v, c[partition[v]])$ never increases, the maximum of this quantity over all v doesn't increase either. This precisely gives us $a[i] \leq a[i-1]$. Using the inductive hypothesis, we are done. $\square$

This shows that all $k+1$ centers are at a distance of at least $cost(v, x)$ from each other. Since there are only k sets in $\mathbf{OPT}$, two of them must be in the same partition by the pigeonhole principle, so in that partition, the costliest edge must have weight at least $cost(v, x)$. $\square$

Combining these two claims, we have $\mathbf{ALG} \leq 2 \cdot \mathbf{OPT}$, and we are done.

For a tight example, consider the following:

Let $n = 4$ and $k = 2$, an array $a[1 \ldots 4] = [2, 1, 3, 4]$, and (the metric) $cost(u, v) = |a[u] - a[v]|$. This satisfies the triangle inequality (since $|a-b|+|b-c| \geq |a-c|$ for all $a, b, c \in \mathbb{R}$), and the algorithm partitions it into $\{1,2,3\}$ and $\{4\}$, which gives $\mathbf{ALG} = 2$, but the optimal way is to split it as $\{1,2\}$ and $\{3,4\}$ with $\mathbf{OPT} = 1$.

To argue that even if we change the $<$ sign to the $\leq$ sign anywhere, we can't get better than a $2-$approximation with this algorithm, we tweak the example slightly to $a[4] = 4 + \varepsilon$ for some small ε. Then the algorithm gives us the same partition again, with $\mathbf{ALG} = 2$, and the optimal partition is also the same, with $\mathbf{OPT} = 1 + \varepsilon$. This can be made arbitrarily close to 2, so we are done.

2 Problem 2

2.1 Statement

Let x^* be an optimum solution to the linear relaxation of the integer program for minimum vertex cover in $G = (V, E)$. Suppose $x_i^* \in \{0, 1/2, 1\}, 1 \le i \le n$ where x_i is the variable associated with $v_i \in V$. Further assume that the vertices of G are colored with 4 colors such that every pair of adjacent vertices have different colors. How will you round the solution x^* to obtain a 1.5-approximation for the minimum vertex cover. Give an algorithm and analyse its guarantee.

2.2 Solution

Since the statement mentions that we are already given such a colouring, we will assume that the input additionally contains the colouring information for all vertices in the graph.

Without loss of generality, let the colours be red, blue, green, yellow. Our algorithm will be as follows:

1. Find a colour c with the most number of vertices coloured with that colour which have $x_i^* = \frac{1}{2}$ (i.e., c is a majority colour among vertices v_i which have $x_i^* = \frac{1}{2}$).
2. Initialize set S with the empty set.
3. For each vertex v_i in V, add v_i to S if and only either
   - $x_i^* = 1$, or
   - $x_i^* = \frac{1}{2}$ and the colour of v_i is not c

   (i.e., round an x_i^* with value $\frac{1}{2}$ to 1 if and only if the colour of v_i is not c, else round it down to 0).
4. Return S.

For the sake of convenience, we suppose that the colour c that was picked is blue.

Claim 2.1
S is a vertex-cover.

Proof. Suppose there is an edge which has not been covered. Then there exists an edge (v_i, v_j) such that after rounding the solution from x^* to $\overline{x}$, we have $\overline{x}_i + \overline{x}_j < 1$. Since both of these are integers, this can happen iff both $\overline{x}_i, \overline{x}_j$ are 0. In the case that either of x_i^* and x_j^* is 0, the other variable must have been 1, which couldn't have been rounded down. And if either of them is 1, it isn't rounded down anyway, so the inequality is still satisfied.

Hence, the only possibility is that both of x_i^*, x_j^* are $\frac{1}{2}$, and both of $\overline{x}_i, \overline{x}_j$ are both rounded down to 0. This is only possible if both v_i, v_j were blue, which is impossible since the condition on the colouring guarantees that v_i, v_j must have had opposite colours.

This is a contradiction, and hence, S is a valid vertex cover. $\square$

Claim 2.2
$|S| \le \frac{3}{2} \cdot \mathbf{OPT}$

Proof. Let the set of vertices with $x_i^* = 1$ be A, the set with $x_i^* = 0$ be B, and the set with $x_i^* = \frac{1}{2}$ be C. Let D be the subset of C which consists of vertices which have colour blue.

Note that $\mathbf{OPT} \ge \mathbf{OPT}_{LP} = |A| + \frac{1}{2}|C|$, since the LP is a relaxed version of the original problem.

Now note that since blue is a majority colour among vertices in C, it must have at least $\frac{1}{4}|C|$ vertices.

Now consider the following inequalities:

$$\begin{aligned}
\mathbf{ALG} &= |A| + |C| - |D| \\
&\leq |A| + \frac{3}{4} \cdot |C| \\
&\leq \frac{3}{2} \cdot |A| + \frac{3}{4} \cdot |C| \\
&= \frac{3}{2} \cdot \left(|A| + \frac{1}{2} \cdot |C| \right) \\
&= \frac{3}{2} \cdot \mathbf{OPT}_{LP} \\
&\leq \frac{3}{2} \cdot \mathbf{OPT}
\end{aligned}$$

This completes the proof. $\square$

3 Problem 3

3.1 Statement

We are given a directed acyclic graph $G = (V, E)$, vertices $s, t \in V$, edge-costs $c : E \to \mathbb{R}^+$, edge-lengths $l : E \to \mathbb{R}^+$, and a length bound L. Give a full polynomial time approximation scheme (FPTAS) to find the minimum cost path from s to t of length at most L. Hint: First give a dynamic program to solve the problem assuming the edge-costs are integers.

3.2 Solution

Firstly, we give a dynamic program that works if all the edge weights c_i are integer weights.

```
function MinCostPath($G = (V, E)$, $s$, $t$, $n = |V|$, $m = |E|$, $c[1 \dots m]$, $l[1 \dots m]$, $L$)
    Let $ord$ be a topologically sorted order of vertices in $G$
        ▷ i.e., $ord[i]$ denotes the $i^{th}$ vertex in this topologically sorted order. We can maintain a permutation
    that gives us, given a vertex $v$, the index $i$ such that $ord[i] = v$, and this can be constructed in linear time.
    Let $C \leftarrow \max_i c[i]$
    Let $minLen[1 \dots n][0 \dots Cn]$ be a matrix initialized to $\infty$
                                                  ▷ For our purposes, initializing it to $L + 1$ works as well.
        ▷ $minLen[i][j]$ will represent the minimum length of a path from $ord[i]$ to $k$ with cost at most $j$
    Let $q$ be the index such that $ord[p] = t$.
    Let $p$ be the index such that $ord[q] = s$.
    if $p > q$ then
        return failure
    end if
    for $i$ from 0 to $Cn$ do
        $minLen[q][i] \leftarrow 0$
    end for
    for $i$ from $q - 1$ down to $p$ do
        for $j$ from 0 to $Cn$ do
            if $j \neq 0$ then
                $minLen[i][j] \leftarrow minLen[i][j - 1]$
            end if
            for each edge $(ord[i], ord[k])$ (with id $r$) with cost $c[r] \leq j$ do
                $minLen[i][j] \leftarrow \min(minLen[i][j], minLen[k][j - c[r]] + l[r])$
            end for
        end for
    end for
    if $minLen[p][Cn] > L$ then
        return failure
    end if
    Let $c'$ be the minimum index with $minLen[p][c'] \leq L$.
    Let $l'$ be $minLen[p][c']$
    Let $path \leftarrow$ list consisting only of $p$
    while $p \neq q$ do
        Let $(p, r)$ be an edge with cost $f$ and length $g$, and $minLen[r][c' - f] = l' - g$.
        $c' \leftarrow c' - f$
        $l' \leftarrow l' - g$
        $p \leftarrow r$
        Append $p$ to $path$
    end while
    return $path$
end function
```

Claim 3.1

This algorithm returns failure if there is no path with length $\leq L$, else it returns a minimum cost with length at most L.

Proof. Note that since ord is a topological ordering of the vertices, any edge joins a vertex v_i to v_j only if $i < j$.

Firstly, we claim that $minLen[i][j]$ does indeed satisfy the property in line 7, after the nested iteration for j in the corresponding iteration for i is completed.

For the case $i = q$, it is fairly straightforward, since the empty path has length L and all paths have non-negative lengths so we can't do better.

Now suppose this is true for all $i' > i$. We show that this is true for i as well. Now consider any path with cost x and length y, starting at $ord[i]$. Then there must be an edge from $ord[i]$ to some $ord[k]$ with index r and cost $c[r]$ and length $l[r]$ such that the cost of the path from $ord[k]$ to t is $x - c[r]$ and the length is $y - l[r]$.

Fix some j. Then this holds for each path with cost $\leq j$ as well. Moreover, we must have the cost of the remaining path be non-negative (since edge costs are non-negative integers). So, it is sufficient to fix an edge and consider the minimum length of a path from $ord[k]$ to t with cost at most $j - c[r]$ (since path lengths are additive), and then take the minimum over all edges coming out of $ord[i]$, which completes the proof of the claim.

Now suppose there is no path with length $\leq L$ between s and t. Then we have $minLen[p][c] > L$ for all $c \in [0 \dots Cn]$, so we return failure in this case.

Now suppose there is indeed such a path. Then the number of edges in the path are at most $n - 1$, so the path cost is bounded above by $C(n-1) < Cn$. Hence, by the claim, we have that the minimum c for which $minlen[p][c]$ is $\leq L$ is indeed the minimum cost which is required (and such a c exists since we have assumed that such a path exists, so $minLen[p][cost(path)] \leq L$ by the claim). The rest of the algorithm is just construction of the path from the $minLen$ values, which is a straightforward simulation of walking on the path. □

Note that this algorithm uses Cn as an upper bound on the cost of **OPT**, and there is implicitly a lower bound of 1 since weights belong to $\mathbb{R}^+$, and the only case where the answer is 0 is the case when $s = t$, which is trivial. We will try to iteratively find better and better lower bounds so that we can get a better algorithm.

Firstly note that we can change the order of computation of our dynamic program from row major to column major order, by noting that (i, j) is computed strictly after $(i-1, *)$, and it depends only on smaller j. By allocating memory only for the columns we need (this can be done by swapping rows and columns and the coordinates in our algorithm), we will get another algorithm which would run in $O((n+m) \cdot \textbf{OPT})$. Call this the **good** algorithm.

To reduce the upper bound, we will try to binary search on it (with the initial range being $[1, Cn]$, where C is the max cost edge length).

For the binary search, we will use the following approximate predicate for a given $0 < r < 1$, given an upper bound B to test:

1. Remove all edges with cost $> B$, and replace the cost of edge e with $\lfloor cost \cdot n/(Br) \rfloor$
2. For c in 0 to $\frac{n}{r}$:
   - Compute the $minLen$ values for cost $= c$ (this corresponds to a column) using the good algorithm (we can store the results for the previous runs as well).
   - If we have $minLen[p][c] \leq L$, return false.
3. Return true

Claim 3.2

If this predicate returns false, then $\textbf{OPT} \leq (1+r)B$, else $\textbf{OPT} > B$.

Proof. Suppose we return false. Then we have a path in the rounded instance with cost c, and since the number of edges on the path can be at most n, and the maximum length we have shaved off an edge by rounding to be Br/n (comparing the original costs to the costs we get when we multiply the costs in the instance considered in the algorithm with Br/n), we have the path cost at most $B + n \cdot Br/n = B(1+r)$.

If we return true, this means that even in the relaxed instance (after scaling and rounding), there is no solution with cost $\leq \frac{n}{r}$, i.e., in the relaxed instance where we scale back after rounding, there is no solution with cost $\leq \frac{n}{r} \cdot \frac{Br}{n} = B$, so even for the original problem, there is no solution with cost B, which completes the proof. □

Hence using this algorithm, we can choose $r = \sqrt{2} - 1$, and run this algorithm while doing a binary search on **OPT**, while the ratio between the upper bound and the lower bound is more than 2. It will always lead to a reduction of the ratio between by at least a factor of $\sqrt{2}$ each time, i.e., a reduction of log of the upper bound by at least $\frac{1}{2}$ each time. The time taken by the predicate to run is $O(n(n+m)/r) = O(n(n+m))$ (since r is a fixed constant equal to $\sqrt{2}-1$), so we will end up with an $O(n(n+m)\log(Cn))$ algorithm till we get an upper bound which is at most two times the lower bound.

Note that since $\log C$ is the size of C in the input, this is polynomial time in input.

Hence using this algorithm (which in turn depends on the good algorithm), we have finally found an upper bound which is at most twice **OPT** (since lower bound is $\leq$ **OPT**), and that too in polynomial time, as a single iteration of the good algorithm takes time polynomial in n and $1/r$.
Now our FPTAS will be as follows:

1. Let C be an upper bound on the answer found from the algorithm (this is different from the definition of C used above).
2. Multiply all edge costs with $\frac{2n}{\varepsilon C}$, and round them up.
3. Run the algorithm on this modified instance (and in the algorithm, replace the Cn in the algorithm by $\frac{2n}{\varepsilon}$ that we have chosen, since this is now the upper bound on the optimal path cost), and return the output of this algorithm.

Claim 3.3

This is a $(1+\varepsilon)$ approximation.

Proof. Consider the optimal solution (under the condition that a solution exists). Note that our algorithm also returns a solution, since we haven't changed L or the edge lengths, or the structure of the graph. Suppose the edge costs on the path were $x_1, \ldots, x_k$. Then in the modified instance, it corresponds to $\lceil \frac{2nx_1}{C\varepsilon} \rceil, \ldots \lceil \frac{2nx_k}{C\varepsilon} \rceil$. Suppose the edge costs on the path returned by our algorithm are $y_1, \ldots, y_K$. Then in the modified instance, the edge costs are $\lceil \frac{2ny_1}{C\varepsilon} \rceil, \ldots \lceil \frac{2ny_K}{C\varepsilon} \rceil$.

We then have the following:

$$\begin{aligned}
\sum_{i=1}^{K} y_i &\leq \frac{C\varepsilon}{2n} \cdot \sum_{i=1}^{K} \left\lceil \frac{2ny_i}{C\varepsilon} \right\rceil \\
&= \mathbf{OPT}_{modified} \\
&\leq \frac{C\varepsilon}{2n} \cdot \sum_{i=1}^{k} \left\lceil \frac{2nx_i}{C\varepsilon} \right\rceil \\
&\leq \frac{C\varepsilon}{2n} \cdot \sum_{i=1}^{k} \left(1 + \frac{2nx_i}{C\varepsilon} \right) \\
&= \mathbf{OPT} + \frac{kC\varepsilon}{2n} \\
&\leq \mathbf{OPT} + \frac{k \cdot \mathbf{OPT}}{n} \\
&\leq \mathbf{OPT} \cdot (1+\varepsilon)
\end{aligned}$$

□

Claim 3.4

The running time of this algorithm is $O(n(n+m)/\varepsilon)$

Proof. Note that the time taken by the algorithm is
$O((n+m) \cdot (\text{upper bound on OPT for the instance passed to the algorithm}))$, which is $O((n+m) \cdot \frac{2n}{\varepsilon})$, and we are done. □

This shows that we indeed have a FPTAS for this final algorithm.

4 Problem 4

4.1 Statement

Consider the following problem arising in communication networks. The network consists of a cycle on n nodes. Some set C of calls is given: each call has an originating node and a destination node on this cycle. Each call can be routed either clockwise or anticlockwise around the cycle and the objective is to route the calls so that the maximum load on any link (edge of the cycle) is minimised. The load on a link is the number of calls routed through it. Give a 2-approximation algorithm for this problem.

4.2 Solution

We will assume that the originating and the destination nodes are distinct in each call (since they don't contribute to the load of any link).

We will use LP relaxation for this problem.

We number the nodes from 0 to $n-1$ in an anticlockwise order, and let link i ($0 \le i < n$) be between nodes i and $(i+1) \pmod{n}$.

Let call k have the originating node o_k and the destination node d_k.

Without loss of generality, we can assume $o_k < d_k$, since a clockwise routing for a call with originating and destination nodes o_k, d_k respectively contributes to the same links that an anticlockwise routing for a call with originating and destination nodes d_k, o_k respectively does (so, before running the algorithm, we swap these if needed, and given a final assignment, we can check if the original call is from a node with a lower number to a node with a higher number, and if it is not the case, then we simply change the direction of the call).

Now note that for a call o_k, d_k, routing it anticlockwise contributes a load of 1 to each link i with $o_k \le i < d_k$.

We will set up an integer program as follows. Let the variable $x_{i,0}$ be 1 iff the link i is routed anticlockwise, and 0 otherwise. Let the variable $x_{i,1}$ be 1 iff the link i is routed clockwise, and 0 otherwise. Finally, for the answer, we make a variable y.

First we construct an integer program as follows:

$$\begin{aligned}
&\text{Minimize } y \text{ subject to the constraints}\\
&\forall\, 1 \le i \le |C| : x_{i,0} + x_{i,1} = 1\\
&\forall\, 1 \le i \le |C| : x_{i,0}, x_{i,1} \in \{0,1\}\\
&\forall\, 0 \le i < n : \sum_{(o_k,d_k)\in C, o_k \le i < d_k} (-1)x_{k,0} + \sum_{(o_k,d_k)\in C, o_k > i \vee i \ge d_k} (-1)\cdot x_{k,1} + y \ge 0
\end{aligned}$$

The first and the second constraints tell us that we need to assign at least one direction to these. The last constraints say that all loads should be at most y.

Moreover, given a solution to this integer program, we trivially have an assignment which has maximum load $\le y$.

We now relax it to the following linear program:

$$\begin{aligned}
&\text{Minimize } y \text{ subject to the constraints}\\
&\forall\, 1 \le i \le |C| : x_{i,0} + x_{i,1} \ge 1\\
&\forall\, 1 \le i \le |C| : x_{i,0} \ge 0\\
&\forall\, 1 \le i \le |C| : x_{i,1} \ge 0\\
&\forall\, 0 \le i < n : \sum_{(o_k,d_k)\in C, o_k \le i < d_k} (-1)\cdot x_{k,0} + \sum_{(o_k,d_k)\in C, o_k > i \vee i \ge d_k} (-1)\cdot x_{k,1} + y \ge 0
\end{aligned}$$

Note that if there is a solution to the LP with $x_{i,j} > 1$, we can replace it by 1 while keeping y the same and not violating any constraint, since we don't decrease the LHS of any inequality of the fourth type by this operation, and the second and third types of inequalities are trivially satisfied, and any inequality of the first type in which this variable appears would be satisfied since the other variable is ≥ 0 due to the second and the third inequality.

Our algorithm (after the above simplifications) would be:

1. Solve the LP so formed above to get a solution x^*, and replace x_i^* by 1 if it exceeds 1.
2. For each link i, if $x_{i,0}^* \ge \frac{1}{2}$, set $\overline{x}_{i,0} = 1$ and $\overline{x}_{i,1} = 0$, and otherwise, do the opposite.

3. Corresponding to the integer solution $\overline{x}$, construct a mapping from calls to their direction of routing (also performing the check with the original mapping as described earlier).
4. Return this routing scheme.

Note that by our algorithm, we guarantee that the first two constraints of the integer program are always satisfied.

Claim 4.1
For all valid i, j, $\overline{x}_{i,j} \leq 2 \cdot x^*_{i,j}$.

Proof.

1. If $x^*_{i,0} \geq \frac{1}{2}$, we have $\overline{x}_{i,0} = 1 = 2 \cdot \frac{1}{2} \leq 2 \cdot x^*_{i,0}$, and $\overline{x}_{i,1} = 0 \leq 2 \cdot x^*_{i,1}$.
2. Otherwise, the first constraint tells us that $x^*_{i,1} \geq \frac{1}{2}$, so the same analysis as the previous case holds.

$\square$

Note that since this is a relaxed version of the integer program, we have $\mathbf{OPT} \geq y^*$.

Claim 4.2
This algorithm is a 2-approximation.

Proof.
Let **ALG** be the maximum load on a link in the routing done by the algorithm. Note that

$$\begin{aligned}
\mathbf{ALG} &= \max_{0 \leq i < n} \left(\sum_{(o_k,d_k) \in C, o_k \leq i < d_k} \overline{x}_{k,0} + \sum_{(o_k,d_k) \in C, o_k > i \vee d_k \leq i} \overline{x}_{k,1} \right) \\
&\leq \max_{0 \leq i < n} \left(\sum_{(o_k,d_k) \in C, o_k \leq i < d_k} 2 \cdot x^*_{k,0} + \sum_{(o_k,d_k) \in C, o_k > i \vee d_k \leq i} 2 \cdot x^*_{k,1} \right) \\
&= 2 \max_{0 \leq i < n} \left(\sum_{(o_k,d_k) \in C, o_k \leq i < d_k} x^*_{k,0} + \sum_{(o_k,d_k) \in C, o_k > i \vee d_k \leq i} x^*_{k,1} \right) \\
&\leq 2y^* \\
&\leq 2 \cdot \mathbf{OPT}
\end{aligned}$$

Here the first inequality comes from our previous observation, the second comes from the fact that x^*, y^* is a feasible solution of the LP, and the last comes from our previous observation that this LP is a relaxed version of the integer program. $\square$

Claim 4.3
This is a polynomial-time algorithm.

Proof. The time taken by this algorithm is dominated by the time taken to solve the LP (if we use a standard LP solver), which is at most cubic in the number of constraints and variables, i.e., it is $O((n + |C|)^3)$, which is polynomial in the input size. $\square$

5 Problem 5

5.1 Statement

Consider the following problem.

There is a set U of n nodes, which we can think of as users (e.g.,these are locations that need to access a service, such as a Web server). You would like to place servers at multiple locations. Suppose you are given a set S of possible sites that would be willing to act as locations for the servers. For each site $s \in S$, there is a fee $f_s \geq 0$ for placing a server at that location. Your goal will be to approximately minimize the cost while providing the service to each of the customers. So far this is very much like the Set Cover Problem: The places s are sets, the weight of sets is f_s, and we want to select a collection of sets that covers all users.

There is one extra complication: Users $u \in U$ can be served from multiple sites, but there is an associated cost d_{us} forserving user u from site s. When the value d_{us} is very high, we do not want to serve user u from site s; and in general the service cost d_{us} serves as an incentive to serve customers from "nearby" servers whenever possible.

So here is the question, which we call the Facility Location Problem: Given the sets U and S, and costs f and d, you need to select a subset $A \subseteq S$ at which to place servers (at a cost of $\sum_{s\in A} f_s$), and assign each user u to the active server where it is cheapest to be served, $\min_{s\in A} d_{us}$.

The goal is to minimize the overall cost $\sum_{s\in A} f_s + \sum_{u\in U} \min_{s\in A} d_{us}$. Give an $H(n)$-approximation for this problem. (Note that if all service costs d_{us} are 0 or infinity, then this problem is exactly the Set Cover Problem: f_s is the cost of the set named s, and d_{us} is 0 if node u is in set s, and infinity otherwise.)

5.2 Solution

We assume that d_{us} is non-negative. Consider the following algorithm:

1. Let X be the set of uncovered users, initially U.
2. While X is non-empty:
   - For each size i in $\{1, \ldots, |X|\}$, choose the site s_i which has the minimum value of
   $$\frac{f_i + \sum_{u\in X, u\in \text{ the cheapest } i \text{ uncovered users to } s_i} d_{us_i}}{i}$$
   and find the value of i which maximizes this (note that the cheapness is relative to a set, and is given by d_{us_i}).
   - Add s_i to the answer, and remove all uncovered elements that s_i covers, from X.
3. Return the answer.

Note that this does give a valid assignment, since we cover all users, and this is indeed a polynomial time algorithm.

We claim that this is equivalent to solving the following instance of set cover using the following algorithm:

Instance:

1. The universal set is U.
2. There are 2^U copies of any site $s \in S$, with each copy being associated to some subset W of U, and the cost of this copy being $f_s + \sum_{u\in W} d_{us}$, and this copy covers only elements in W. We define $W(copy)$ to be the set associated with a copy.

Algorithm:

1. Let X be U.
2. While X is non-empty:
   - Let c be some copy of a site with the minimum ratio $\frac{cost(c)}{|W(c)\cap X|}$
   - Add c to the answer, and remove all uncovered elements that c covers from X.
3. Return the answer.

Indeed, given a fixed size of $|W(s_i) \cap X|$, we need to minimize the cost of adding s_i, which is f_i plus the cost of covering some subset of elements of U. This is minimized when the subset of elements covered $W(s_i)$ is $X \cap W(s_i)$ itself. Hence, the greedy step is the same.

Now we show that solving this instance of set cover is equivalent to the original problem, and then we will show that this instance of set cover gives a $H(n)$ approximation for the general weighted set cover problem, which will imply this bound as well.

Claim 5.1

Solving this instance of set cover is equivalent to solving the Facility Location Problem.

Proof. Suppose there is a set cover with a cost C. Suppose in that set cover, some two set copies correspond to intersecting subsets of U. Then since d_{us} is non-negative, we can remove the intersection from one of the set copies and get a cost at most C. After all such possible reductions, we get a partition of U (each partition corresponding to a set that covers elements in it) with a cost at most C (since in that partition, we reassign each element to the closest site, which has a cost d_{us} which is at most the contribution of this element to C).

Now suppose there is a solution to the original problem with cost C'. Then there is a solution to the set cover which has cost precisely C' – for this, for each user, we assign it to the closest site, and consider, for each site $s \in S$, the copy in the family of sets which corresponds to covering precisely those users whose closest site is s.

All in all, if we have a min-cost solution for either problem, we can easily construct a min-cost solution for the other problem, which is what our algorithm does as well. $\square$

Claim 5.2

This algorithm gives an $H(n)$-approximation for the general weighted cover.

Proof. The proof closely mirrors the proof of the case when the weights of sets are the same (i.e., when the cost of a solution is proportional to the number of sets taken into account).

Suppose that we assign $cost(c)/(|X \cap W(c) \cdot |H(n))$ to each uncovered element when it is covered for the first time (i.e., to each element in $X \cap W(c)$). Then the total weight we give to all elements is $\frac{\mathbf{ALG}}{H(n)}$.

Now consider any copy c, and let $w = W(c)$, and suppose $w = \{u_1, \ldots, u_k\}$, and elements are covered in this order. Then, in an optimal solution, the cost of u_i is at most $\frac{cost(c)}{(k-i+1)\cdot H(n)}$, since otherwise, we could have picked the copy c at the iteration u_i was covered instead to get a better solution. So the total cost of this set is at most $\frac{cost(c)}{H(n)} \sum_{i=1}^{k} = \frac{cost(c) \cdot H(k)}{H(n)} \leq cost(c)$.

Now note that the sum of weights given to all elements is at most the sum of weights counted in the labelled union of the sets in the optimal solution (since each element is counted at least once), which is precisely the total cost of all sets chosen in the optimal solution. Hence, we have $\mathbf{OPT} \geq \frac{\mathbf{ALG}}{H(n)}$, which gives us an $H(n)$-approximation to this instance of set cover. $\square$

Since the universal sets for the set cover problem was U, we get an $H(|U|) = H(n)$ approximation for the original problem, we are done.