CSE319: Modern Algorithm Design

Homework 2 Solutions

Submitted By: Divyajeet Singh (2021529) **Discussion Partner:** Siddhant Rai Viksit (2021565)

Solution 1.

Part (a)

In the naïve implementation, we regrow the tree (taking $O(m)$ time) after each price update. We do this for each of the $O(n)$ iterations in one phase. We modify the BFS procedure to ensure that the total work done in creating the search tree over $O(n)$ iterations of the phase is $O(m)$. Along with a BFS-queue Q, we also maintain a set S of currently '*un-tight*' edges, i.e.

$$S = \{(u, v) \in E \mid w(u, v) - p_u - p_v \neq 0, u \in Q\} \tag{1}$$

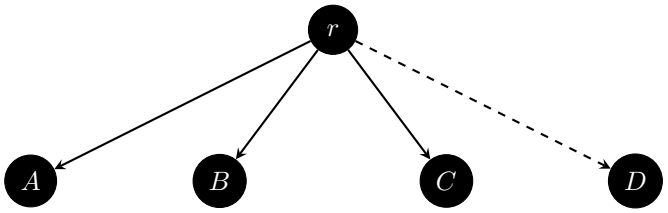

Figure 1: (r, A), (r, B), and (r, C) are tight and added to Q. (r, D) is un-tight and added to S.

The idea is as follows. At the start of the phase, add $r \in L$ to Q. While Q is not empty,

- Pop a node u from Q.
- For each edge $(u, v) \in E$, if $w(u, v) - p_u - p_v = 0$, add v to Q, otherwise, add (u, v) to S.

Once we get '*stuck*', we perform a price update as usual for all explored vertices in L and R. Next, we find the edges in S made tight by the price update[1]. Let these edges be $T_N = \{(u, v) \in S \mid w(u, v) - p_u - p_v = 0\}$. We add the vertices $V' = \{v \mid (u, v) \in T_N\}$ to Q and continue the BFS (as opposed to starting a BFS from scratch). We repeat this process until a good path P is found.
Since we grow one search tree per phase, each edge is considered at most twice (once while adding it to S and once while exploring it, if at all). Thus, the total work done searching for good paths over all the $O(n)$ iterations of one phase is now $O(m)$. This obviously excludes the work done to compute δs and update prices.

Part (b)

We know that the value of δ for a price update can be computed in $O(m)$ per iteration in each phase. For each phase, we also maintain a min-heap H, along with the BFS-queue Q and set S. The heap is structured as [Priority, Item].

$$H = \{[p_v, v] \mid (u, v) \in S\} \tag{2}$$

Just like the set S, the heap H is built incrementally. While BFS-ing, when we encounter an un-tight edge $e = (u, v)$, we add e to S and $[p_v, v]$ to H. As expected, we set[2]

$$[\delta, \_] \leftarrow \textsc{Extract-min}(H) \tag{3}$$

[1] Note that there can be multiple.
[2] For consistency with S, we need to $\textsc{Extract-min}(H)$ until $\textsc{Top}(H)$ (the minimum value in H) exceeds the first selected δ.

The time complexity of each extraction depends on the total number of elements k in the heap. There can be at most n^2 (non-distinct) elements in the heap during the phase. We do not need to worry about duplicates since the extraction time is proportional to $\log k$, and $\log n^2 = 2 \log n = O(\log n)$. Finally, there can be at most $O(m)$ inserts during the search tree expansion and at most $O(m)$ extractions[3] overall during the phase.
Thus, the total time spent in computing δs over the entire phase is $O(m \log n)$. This yields an $O(mn \log n)$ Hungarian Algorithm, as there can be at most $O(n)$ phases, and each phase can be implemented in $O(m + m \log n)$ time.

Solution 2.

Part (a)

We want to design a $\textsc{Poly}(n, k)$ algorithm to optimally solve the k-point facility problem optimally on an edge-weighted path (line graph) of n vertices. We do this with a 3-dimensional dynamic programming approach.

Dynamic Programming States

Let $\Phi[i, v, \kappa]$ be the minimum cost of placing a total of κ facilities on the first i vertices such that the last facility is placed at the vertex v. Then, the optimal cost of placing k facilities on the line graph is $\min_{k \le v \le n} \Phi[n, v, k]$.

Base Cases

Equation 4 describes the base cases.

$$\Phi[i, v, \kappa] = \begin{cases} \infty & v > i \textbf{ or } \kappa > v \\ \sum_{u=1}^{i} d_G(u, v) & \kappa = 1 \end{cases} \tag{4}$$

- The first case covers the situations where the last facility is placed at a vertex v after the first i vertices, or when the number of facilities κ is higher than the vertex v at which the last facility is placed.
- The second case covers the actual base case. We place $\kappa = 1$ facilities at a vertex v. The cost of this placement is the sum of the distances of the first i vertices from the vertex v.

Recurrence Relation

Equation 5 describes the recurrence relation.

$$\Phi[i, v, \kappa] = \begin{cases} \Phi[i-1, v, \kappa] + d_G(v, i) & v < i \\ \min_{1 \le u < i} \left\{ \Phi[m-1, u, \kappa - 1] + \sum_{w=m}^{i} d_G(w, i) \right\} & \text{otherwise} \end{cases} \tag{5}$$

where, in the second case, m is the leftmost vertex that is closer to i than u.

- The first case covers the situation when we do not place a facility at the current vertex i. This costs the same as placing κ facilities on the first $i - 1$ vertices with the last facility at v, and the distance between i and its closest facility at v.
- The second case covers the situation when we place a facility at the current vertex i. This arrangement costs placing $\kappa - 1$ facilities in the first $m - 1$ vertices with the last facility at some $1 \le u < i$ and the distance between i and all vertices beyond m, where m is the leftmost vertex that is closer to i than u. We take the minimum over u to find the optimal placement.

The Algorithm

Let the edge weights on the line graph be $W_E = \{w_1, w_2, \ldots, w_{n-1}\}$ from left to right, where $w_i = d_G(i, i+1)$ for $i \in [n-1]$. We calculate the prefix sum array $P[1:n]$ such that

$$P[i+1] = \sum_{j=1}^{i} w_j = \sum_{j=1}^{i} d_G(j, j+1) \quad \forall i \in [n-1] \tag{6}$$

[3] $O(n)$ extractions should be enough, but to take care of duplicates, we might need to extract all the elements inserted in H.

Using the prefix sum array P, we gain immediate access to the distance between any two vertices

$$d_G(u,v) = d_G(v,u) = P[v] - P[u] \quad \forall u < v \in [n] \tag{7}$$

We also need to calculate m for any given (i,j) pair. As per our requirement, $m = \textsc{Median}_G(i,j)$ suffices, which denotes the middle point between i and j with respect to the edge weights[4].
A recursive implementation of the algorithm is given in Algorithm 1. The final solution is given in Algorithm 2.

Algorithm 1 Recursive DP for the k-point Facility Problem on an edge-weighted path

```
1:  procedure DP(i, v, κ):
2:      if (i, v, κ) ∈ Φ then                          ▷ Value already computed
3:          return Φ[i, v, κ]
4:      else if v < i then                  ▷ Do not place any facility at i (Recurrence Case 1)
5:          Φ[i, v, κ] ← DP(i − 1, v, κ) + d_G(v, i)
6:          return Φ[i, v, κ]
7:      else                                ▷ Place a facility at i (Recurrence Case 2)
8:          φ ← ∞
9:          m_prev ← M[i − 1, i]
10:         s ← 0
11:         for u ← i − 1 to 1 do
12:             m ← M[u, i]
13:             for j ← m_prev to m do
14:                 s ← s + d_G(j, i)
15:             end for
16:             φ ← min(φ, DP(m − 1, u, κ − 1) + s)
17:             m_prev ← m
18:         end for
19:         Φ[i, v, κ] ← φ
20:         return Φ[i, v, κ]
21:     end if
22: end procedure
```

Algorithm 2 Solve the k-point Facility Problem on an edge-weighted path

```
1:  procedure k-POINT-FACILITY-LINE(G = (V, E), k):
2:      P ← PREFIX-SUM(W_E)                 ▷ W_E can be obtained by a modified BFS
3:      M ← PAIRWISE-MEDIANS(G)
4:      Φ ← {}                              ▷ An empty memoization table, indexed [i, v, κ]
5:      for v ← 1 to n do
6:          s ← 0
7:          for i ← 1 to n do
8:              s ← s + d_G(v, i)
9:              for κ ← 1 to k do
10:                 if v > i or κ > v then
11:                     Φ[i, v, κ] ← ∞          ▷ (Invalid) Base Case
12:                 end if
13:             end for
14:             Φ[i, v, 1] ← s                  ▷ (Valid) Base Case
15:         end for
16:     end for
17:     OPT ← ∞
18:     for v ← k to n do
19:         OPT ← min(OPT, DP(n, v, k))
20:     end for
21:     return OPT
22: end procedure
```

[4]We can pre-compute this in an array $M[i,j] = \textsc{Median}_G(i,j)$ in $O(n^2)$ time (see **Appendix**)

In Algorithm 2, we initalize the the base cases of the DP table Φ in $O(n^2k)$ time. By iterating over v in the outer loop, we can calculate a running sum to avoid recomputing the distances between i and v for each i.
Algorithm 1 has been written with a slightly non-trivial optimization. In Recurrence Case 2, one might want to compute the sum $\sum_{j=m}^{i} d_G(j,i)$ naïvely for as per 5, resulting in an effort of $O(n)$ for each u. Instead, we loop over u in reverse and maintain a running sum of the distances between j and i for each j from $m_{\textbf{prev}}$ to m. This saves duplicate calculations, as m decreases monotonically as u decreases - an overall total effort of $O(n)$.

Backtracking to find the set C

To find the actual set $C \subseteq V$, we need to backtrack through the table $\Phi[i, v, \kappa]$. We start at $\Phi[i = n, v = n, \kappa = k]$, and decrease v until we find the first $\Phi[i, v, \kappa] \neq \Phi[i, v-1, \kappa]$, and add this v to C. We then repeat this process from $\Phi[v-1, v-1, k-1]$ until we have k vertices in C. Again, the exact algorithm is given in the **Appendix**.

Time Complexity

There are a total of $O(n^2k)$ subproblems. Given the preprocessing and optimizations, we see that

- We spend an $O(n^2k)$ effort to initialize the base cases.
- A total of n states, where $i = v$, are computed in $O(n)$ time, adding another $O(n^2)$ effort.
- The rest of the states are computed in constant time, thanks to memoization.
- $O(n+k)$ time is spent backtracking to find the set C, if required.

Thus, the overall time complexity of the solution is $O(n^2k) = \textsc{Poly}(n,k)$. We could have still achieved a $\textsc{Poly}(n,k)$ runtime by removing the optimization tricks (resulting in a $O(n^3k)$ complexity), but *efficieny* :)

Part (b)

We are given an algorithm $\mathcal{A}$ that solves the k-point facility problem optimally on trees. We need to show that an algorithm that samples a tree T from an α-stretch distribution and runs $\mathcal{A}$ on T to get C_T ensures

$$\mathbb{E}_T[\Phi_T(C_T)] \leq \alpha \cdot \textsc{Opt} \tag{8}$$

Note that we calculate $\Phi_T(C_T)$ as opposed to $\Phi_G(C_T)$, since we solve the k-point facility problem on the tree T. We simply expand $\Phi_T(C_T)$ and use linearity of expectation.

$$\begin{aligned}
\mathbb{E}_T[\Phi_T(C_T)] &= \mathbb{E}_T\left[\sum_{v\in V} d_T(v, C_T)\right] \\
&= \sum_{v\in V} \mathbb{E}_T\left[\min_{c\in C_T} d_T(v,c)\right] \\
&= \sum_{v\in V} \mathbb{E}_T\left[d_T(v, c_v^*)\right] \qquad \text{where } c_v^* = \operatorname*{argmin}_{c\in C_T} d_T(v,c) \\
&\leq \sum_{v\in V} \alpha \cdot d_G(v, c_v^*) = \alpha \cdot \textsc{Opt}
\end{aligned} \tag{9}$$

where the inequality holds due to the α-stretch property of the distribution.

Part (c)

Now, we perform $L = O\left(\frac{\log n}{\epsilon}\right)$ independent runs of the above algorithm, to get the sets $C_1, C_2, \ldots, C_L$ from the trees $T_1, T_2, \ldots, T_L$ sampled from the α-stretch distribution. We want to bound the probability that $\Phi_{T^*}(C^*)$ exceeds the expected cost by a factor of $1+\epsilon$, where T^* is the tree that gives the set $C^* = \operatorname*{argmin}_{i\in[L]} \Phi_{T_i}(C_i)$.

We first find the probability that for any tree T_i,

$$\begin{aligned}
\mathbb{P}[\Phi_{T_i}(C_i) \geq (1+\epsilon)\cdot\alpha\cdot\textsc{Opt}] &\leq \frac{\mathbb{E}_{T_i}[\Phi_{T_i}(C_i)]}{(1+\epsilon)\cdot\alpha\cdot\textsc{Opt}} \qquad \text{by Markov's Inequality} \\
&\leq \frac{1}{1+\epsilon} \qquad \text{by the } \alpha\text{-stretch property}
\end{aligned} \tag{10}$$

Now, the probability that the cost of the minimizing tree-set C^* exceeds some value is the probability that the cost of all tree-sets exceeds that value. So, we have

$$\begin{aligned}\mathbb{P}[\Phi_{T^*}(C^*) \geq (1+\epsilon)\cdot\alpha\cdot\textsc{Opt}] &\leq \mathbb{P}\left[\bigcap_{i=1}^{L}\{\Phi_{T_i}(C_i) \geq (1+\epsilon)\cdot\alpha\cdot\textsc{Opt}\}\right] \\ &= \prod_{i=1}^{L}\mathbb{P}[\Phi_{T_i}(C_i) \geq (1+\epsilon)\cdot\alpha\cdot\textsc{Opt}] \qquad \text{by independence} \\ &\leq \left(\frac{1}{1+\epsilon}\right)^{L}\end{aligned} \tag{11}$$

Let $L = c\cdot\frac{\log n}{\epsilon}$ for some constant $c \in \mathbb{R}_{\geq 0}$. Then, we have

$$\begin{aligned}\mathbb{P}[\Phi_{T^*}(C^*) \geq (1+\epsilon)\cdot\alpha\cdot\textsc{Opt}] &\leq \left(\frac{1}{1+\epsilon}\right)^{c\cdot\frac{\log n}{\epsilon}} \\ &= (1+\epsilon)^{-c\cdot\frac{\log n}{\epsilon}} \\ &\leq (e^{\epsilon})^{-c\cdot\frac{\log n}{\epsilon}} \quad \text{since } 1+x \leq e^x\ \forall x\in\mathbb{R} \\ &= \frac{1}{e^{c\log n}} = \frac{1}{n^k} = \frac{1}{\textsc{Poly}(n)}\end{aligned} \tag{12}$$

where k absorbs the logarithmic conversion factor.

Part (d)

We want to show that the expected weight of a low-stretch tree with expected stretch α is at most $O(\alpha)$ times the weight of a minimum spanning tree, i.e.,

$$\mathbb{E}_{\textsc{Lst}}[w(\textsc{Lst})] \leq O(\alpha)\cdot w(\textsc{Mst}) \tag{13}$$

Claim 1. *Let T_1 and T_2 be any two spanning trees of a graph G. Then,*

$$\bigcup_{(u,v)\in T_1} P_{T_2}(u,v) = T_2 \tag{14}$$

where $P_T(x,y) = (v_1 \to v_2 \to \cdots \to v_k)$ denotes the path from x to y in a spanning tree T, with $v_1 = x$ and $v_k = y$.

Proof. Suppose T_2 is rooted at some $r \in T_2$. Then, every edge $\textsc{Par}_{T_2}(y) = (x,y)$ can be said to be the parent of the vertex y in T_2. Let the sub-tree of T_2 rooted at $v \in T_2$ be $\textsc{Sub}_{T_2}(v)$. Then, for any $(u,v) \in T_1$, we have either of the two cases

1. $v \in T_2 \setminus \textsc{Sub}_{T_2}(u)$, i.e. v is not in the sub-tree of u in T_2. Then, $\textsc{Par}_{T_2}(u) \in P_{T_2}(u,v)$.
2. $v \notin T_2 \setminus \textsc{Sub}_{T_2}(u)$, i.e. v is in the sub-tree of u in T_2. v is connected to all other vertices in this sub-tree. Then, there must be some $x \in \textsc{Sub}_{T_2}(u)$ and $y \in T_2 \setminus \textsc{Sub}_{T_2}(u)$, such that $(x,y) \in T_1$, otherwise T_1 would not be spanning. So, we have an edge $(x,y) \in T_1$ such that $y \in T_2 \setminus \textsc{Sub}_{T_2}(x)$, and thus, by case 1, $\textsc{Par}_{T_2}(u) \in P_{T_2}(x,y)$.

This means that each edge ('*parent*' of some vertex) in T_2 is covered by at least some edge in T_1. □

Claim 1 holds for any two general spanning trees. Now consider an Lst and an Mst of the graph G. We have by the α-stretch property of the Lst that

$$\begin{aligned} & & \mathbb{E}_{\textsc{Lst}}[d_{\textsc{Lst}}(x,y)] &\leq \alpha\cdot d_G(x,y) \qquad \forall\ x,y\in V \\ &\implies & \sum_{(x,y)\in\textsc{Mst}} \mathbb{E}_{\textsc{Lst}}[d_{\textsc{Lst}}(x,y)] &\leq \sum_{(x,y)\in\textsc{Mst}} \alpha\cdot d_G(x,y) \\ \implies & \sum_{(x,y)\in\textsc{Mst}}\ \sum_{(u,v)\in P_{\textsc{Lst}}(x,y)} \mathbb{E}_{\textsc{Lst}}[d_{\textsc{Lst}}(u,v)] &\leq \alpha\cdot w(\textsc{Mst}) \end{aligned} \tag{15}$$

By Claim 1 we have (where $T_1 = \textsc{Mst}$ and $T_2 = \textsc{Lst}$)

$$\bigcup_{(x,y)\in \textsc{Mst}} P_{\textsc{Lst}}(x,y) = \textsc{Lst}$$

$$\begin{aligned}\mathbb{E}_{\textsc{Lst}}[w(\textsc{Lst})] = \mathbb{E}_{\textsc{Lst}}\left[w\left(\bigcup_{(x,y)\in \textsc{Mst}} P_{\textsc{Lst}}(x,y)\right)\right] &\leq \sum_{(x,y)\in \textsc{Mst}} \mathbb{E}_{\textsc{Lst}}[w(P_{\textsc{Lst}}(x,y))] \\ &= \sum_{(x,y)\in \textsc{Mst}} \sum_{(u,v)\in P_{\textsc{Lst}}(x,y)} \mathbb{E}_{\textsc{Lst}}[d_{\textsc{Lst}}(u,v)]\end{aligned} \tag{16}$$

where the inequality follows due to the Union bound. By 15 and 16, we get

$$\mathbb{E}_{\textsc{Lst}}[w(\textsc{Lst})] \leq \alpha \cdot w(\textsc{Mst}) \tag{17}$$

Part (e)

Solution 3.

Part (a)

We need to show that for all $x, y \in V$, even if $(x,y) \notin E$,

$$d_G(x,y) \leq d_H(x,y) \leq \gamma \cdot d_G(x,y) \tag{18}$$

The first inequality holds trivially, since the γ-distance emulator H is a subgraph of G, so, $d_G(x,y) \leq d_H(x,y)$. The second inequality also holds trivially for $(x,y) \in E$ by the definition of γ-distance emulator. If $(x,y) \notin E$, then consider the shortest path $P^*_G(x,y) = (v_1 \to v_2 \to \ldots \to v_k)$ in G, where $v_1 = x$ and $v_k = y$, and each edge $(v_i, v_{i+1}) \in E$.

$$\begin{aligned} d_G(x,y) &= \sum_{i=1}^{k-1} d_G(v_i, v_{i+1}) \\ &\geq \sum_{i=1}^{k-1} \frac{1}{\gamma} \cdot d_H(v_i, v_{i+1}) \\ \implies \gamma \cdot d_G(x,y) &\geq \sum_{i=1}^{k-1} d_H(v_i, v_{i+1}) \end{aligned} \tag{19}$$

Since $H \subseteq G$, some edges $P^*_G(x,y)$ may not be in H. So, $P^*_G(x,y)$ may not be the shortest x-y path in H. This means that

$$d_H(x,y) \leq \sum_{i=1}^{k-1} d_H(v_i, v_{i+1}) \leq \gamma \cdot d_G(x,y) \tag{20}$$

Part (b)

By *Construction 1.*, we have $t = 4 \log n$ trees, T_1, T_2, …, T_t, sampled from a randomized α-stretch spanning tree distribution. We are given that the distance emulator H is the union of all these trees.

Subpart (i)

For any fixed edge $(x,y) \in E$, we bound the probability that the shortest x-y distance in some fixed tree T_i (i.e. a fixed i) is at least $2\alpha \cdot d_G(x,y)$.

$$\begin{aligned} \mathbb{P}[d_{T_i}(x,y) \geq 2\alpha \cdot d_G(x,y)] &\leq \frac{\mathbb{E}_{T_i}[d_{T_i}(x,y)]}{2\alpha \cdot d_G(x,y)} \qquad \text{by Markov's Inequality} \\ &\leq \frac{1}{2} \qquad \text{by the } \alpha\text{-stretch property} \end{aligned} \tag{21}$$

Now, since H contains all the edges present in all the trees, the probability that the shortest x-y distance in H is at least $2\alpha \cdot d_G(x,y)$ is the probability that the shortest x-y distance in all the trees is at least $2\alpha \cdot d_G(x,y)$. So,

$$\begin{aligned}\mathbb{P}[d_H(x,y) \geq 2\alpha \cdot d_G(x,y)] &\leq \mathbb{P}\left[\bigcap_{i=1}^{t}\{d_{T_i}(x,y) \geq 2\alpha \cdot d_G(x,y)\}\right] \\ &= \prod_{i=1}^{t}\mathbb{P}[d_{T_i}(x,y) \geq 2\alpha \cdot d_G(x,y)] \qquad \text{by independence} \\ &\leq \left(\frac{1}{2}\right)^t = 2^{-t} = \frac{1}{n^4}\end{aligned} \tag{22}$$

Subpart (ii)

We know that there always exists a low-stretch spanning tree distribution with a stretch of $\alpha = O(\log n \log\log n)$ for any graph G with n vertices[5]. So, let the stretch for a graph $G = (V,E)$ be $\alpha = c\log n \log\log n$ for some constant $c \in \mathbb{R}_{\geq 0}$. Then, we have

$$\mathbb{P}[d_H(x,y) \geq 2 \cdot c\log n \log\log n \cdot d_G(x,y)] \leq \frac{1}{n^4} \tag{23}$$

for a fixed edge $(x,y) \in E$. Now, we find the probability that no edge in E has a stretch greater than 2α.

$$\begin{aligned}\mathbb{P}[\forall\ (x,y) \in E, d_H(x,y) \leq 2\alpha \cdot d_G(x,y)] &= \mathbb{P}\left[\bigcap_{(x,y)\in E}\{d_H(x,y) \leq 2\alpha \cdot d_G(x,y)\}\right] \\ &= 1 - \mathbb{P}\left[\bigcup_{(x,y)\in E}\{d_H(x,y) \geq 2\alpha \cdot d_G(x,y)\}\right] \\ &\geq 1 - \sum_{(x,y)\in E}\mathbb{P}[d_H(x,y) \geq 2\alpha \cdot d_G(x,y)] \qquad \text{by Union Bound} \\ &\geq 1 - |E| \cdot \frac{1}{n^4} \geq 1 - \frac{n^2}{n^4} = 1 - \frac{1}{n^2}\end{aligned} \tag{24}$$

So, with probability at least $1 - \frac{1}{n^2}$, no edge in G has a stretch greater than $2c\log n \log\log n$. But that means H is a $(2c\log n\log\log n)$-distance emulator for G with probability at least $1 - \frac{1}{n^2}$. Moreover, the number of edges in H can be bounded by

$$|E_H| = \left|\bigcup_{i=1}^{t} E_{T_i}\right| \leq \sum_{i=1}^{t}|E_{T_i}| = t \cdot (n-1) = 4\log n \cdot (n-1) = O(n\log n) \tag{25}$$

Thus, H is an $O(n\log n)$-sized $O(\log n \log\log n)$-distance emulator for G with probability at least $1 - \frac{1}{n^2}$.

Part (c)

Subpart (i)

We are given a graph G with girth strictly greater than g. We work towards **Sub-subpart (C)**.

Sub-subpart (A)

Given that the average degree of the graph G is $\bar{d} = \frac{2m}{n}$. We show how to construct a feasible subset $S \subseteq V$ such that the induced subgraph $H := G[S]$ has minimum degree at least $\frac{\bar{d}}{2}$. Algorithm 3 describes the construction of such a subset S.

We only need to show that the algorithm terminates without emptying S completely. The removal of a vertex $v \in S$ with $d_G(v) < \frac{\bar{d}}{2}$, the number of edges goes down by at most $\frac{\bar{d}}{2} - 1 = \frac{m}{n} - 1$. Let us assume, for the sake of contradiction, that at the end of the algorithm, $S = \emptyset$. Then, the total number of edges removed would be at most

$$\sum_{v\in V}\frac{m}{n} - 1 = m - n < m \tag{26}$$

[5]**Mistake in question:** Lecture notes suggest a stretch of $O(\log n \log\log n)$ for general graphs instead of $O(n\log n\log\log n)$.

Algorithm 3 Constructing a feasible set $S \subseteq V$

```
procedure CONSTRUCT-FEASIBLE-SET(G = (V, E)):
    S ← V
    while ∃ v ∈ S with d_H(v) < d̄/2 do
        S ← S \ {v}
    end while
    return S
end procedure
```

This means that we have deleted all vertices, but some edges still remain in $G[S]$, which is absurd. Thus, an S produced by Algorithm 3 is feasible, and therefore such an S exists.

Sub-subpart (B)

In such a subgraph H, we show that for any $v \in H$,

$$\left|\textsc{Bfs}\left(H, v, \frac{g}{2}\right)\right| \geq \left(\frac{\bar{d}}{2} - 1\right)^{\left\lfloor \frac{g}{2} \right\rfloor} \tag{27}$$

where the girth of the graph is strictly greater than some g, and $\textsc{Bfs}(G, v, k)$ denotes the set of vertices reacheable from v in at most k steps in the graph G.
This is now an easy job. We know that the minimum degree of any vertex in H is $\frac{\bar{d}}{2}$. So, in the first hop from a vertex v, we can reach at least $\frac{\bar{d}}{2}$ new vertices. In the second hop, we discover at least $\frac{\bar{d}}{2} - 1$ vertices from each of the $\frac{\bar{d}}{2}$ vertices discovered in the first step. In each subsequent hop i, we encounter at least $n_{i-1} \cdot \left(\frac{\bar{d}}{2} - 1\right)$ new vertices, where n_{i-1} is the number of vertices discovered in the $(i-1)^{th}$ hop, and the -1 accounts for the vertex we came from. So, we have the total number of distinct vertices as

$$\begin{aligned}
\left|\textsc{Bfs}\left(H, v, \frac{g}{2}\right)\right| &= \frac{\bar{d}}{2} + \frac{\bar{d}}{2} \cdot \left(\frac{\bar{d}}{2} - 1\right) + \frac{\bar{d}}{2} \cdot \left(\frac{\bar{d}}{2} - 1\right)^2 + \cdots + \frac{\bar{d}}{2} \cdot \left(\frac{\bar{d}}{2} - 1\right)^{\left\lfloor \frac{g}{2} \right\rfloor - 1} \\
&= \frac{\bar{d}}{2} \cdot \frac{\left(\frac{\bar{d}}{2} - 1\right)^{\left\lfloor \frac{g}{2} \right\rfloor} - 1}{\frac{\bar{d}}{2} - 2} \\
&\geq \left(\frac{\bar{d}}{2} - 1\right)^{\left\lfloor \frac{g}{2} \right\rfloor} \qquad \text{since } \frac{x}{x-2} \geq 1 \ \forall \ x \geq 2
\end{aligned} \tag{28}$$

We can rest assured that the counted vertices are distinct by crucially using the fact that the girth of the graph is strictly greater than g, but the hops in all directions can achieve a maximum spread of $2 \cdot \left\lfloor \frac{g}{2} \right\rfloor \leq g$.

Sub-subpart (C)

At this point, we have a set S such that the induced subgraph $H := G[S]$ has minimum degree at least $\frac{\bar{d}}{2}$, and we have seen that the BFS-tree from any vertex $v \in H$ has at least $\left(\frac{\bar{d}}{2} - 1\right)^{\left\lfloor \frac{g}{2} \right\rfloor}$ vertices. Clearly, a BFS-tree of depth $\left\lfloor \frac{g}{2} \right\rfloor$ from any vertex $v \in H$ contains at most the total number of vertices in G. So, we have

$$\begin{aligned}
\left(\frac{\bar{d}}{2} - 1\right)^{\left\lfloor \frac{g}{2} \right\rfloor} &\leq n \\
\implies \frac{\bar{d}}{2} &\leq 1 + n^{\frac{1}{\left\lfloor \frac{g}{2} \right\rfloor}}
\end{aligned} \tag{29}$$

Note that since $\bar{d} = \frac{2m}{n}$, we can write[6]

$$m = \frac{n\bar{d}}{2} \leq n \cdot \left(1 + n^{\frac{1}{\left\lfloor \frac{g}{2} \right\rfloor}}\right) = n + n^{1 + \frac{1}{\left\lfloor \frac{g}{2} \right\rfloor}} = O\left(n + n^{1 + \frac{1}{\left\lfloor \frac{g}{2} \right\rfloor}}\right) \tag{30}$$

[6] Technically, $O\left(n + n^{1 + \frac{1}{\left\lfloor \frac{g}{2} \right\rfloor}}\right) = O\left(n^{1 + \frac{1}{\left\lfloor \frac{g}{2} \right\rfloor}}\right)$.

Subpart (ii)

We are given a variant of Kruskal's algorithm for $\alpha \geq 1$. We start with an empty H and add each edge $(x, y) \in E$ to H if $d_H(x, y) > \alpha \cdot d_G(x, y)$.

Sub-subpart (A)

We first show that at $\alpha = n - 1$, the described procedure becomes Kruskal's algorithm. We note that an edge $e_i = (x, y) \in E$ is included in H_i if

$$d_{H_{i-1}}(x, y) > (n - 1) \cdot d_G(x, y) \tag{31}$$

Let us analyze what happens when the algorithm encounters an edge $e_i = (x, y) \in E$. We land in one of two scenarios:

1. x and y are in different connected components of H_{i-1}. Then,

$$d_{H_{i-1}}(x, y) = \infty > (n - 1) \cdot d_G(x, y) \tag{32}$$

   satisfying the required condition. So, $H_i \leftarrow H_{i-1} \cup \{e_i\}$, i.e. x and y get connected in H_i. Note that when an edge gets added in the i^{th} iteration, $d_{H_j}(x, y) \leq (n - 1) \cdot d_G(x, y)$ for all $j \geq i$.

2. x and y are in the same connected component of H_{i-1}. By defintion of a connected component, there exists a path $P_{H_{i-1}}(x, y) = (v_1 \to v_2 \to \ldots \to v_k)$ in H_{i-1}, where $v_1 = x$ and $v_k = y$, and each edge $(v_j, v_{j+1}) \in H_{i-1}$.

$$\begin{aligned} d_{H_{i-1}}(x, y) &= \sum_{j=1}^{k-1} d_{H_{i-1}}(v_j, v_{j+1}) \\ &= \sum_{j=1}^{k-1} d_G(v_j, v_{j+1}) \\ &\leq k \cdot w^* \qquad \text{where } w^* = \max_{j \in [k-1]} d_G(v_j, v_{j+1}) \\ &\leq (n - 1) \cdot d_G(x, y) \end{aligned} \tag{33}$$

   where the last inequality holds because

   - There can be at most $n - 1$ edges in the path $P_{H_{i-1}}(x, y)$.
   - $w^* = w_{e_i} = d_G(x, y)$, since e_i must be the heaviest edge considered so far.

   This does not satisfy the required condition. So, $H_i \leftarrow H_{i-1}$.

The above two points combined show that an edge $e_i = (x, y) \in E$ is included in H_i if x and y are in different connected components of H_{i-1}. This is exactly the behavior of Kruskal's algorithm, as adding the exact edges that are in different connected components each time ensures that no cycles are formed.
By the equivalence thus proved, H is not only a subgraph, but also a spanning tree. The procedure ensures that $d_{H_{|E|}}(x, y) \leq (n - 1) \cdot d_G(x, y)$ for each edge $(x, y) \in E$, thus making H an $(n - 1)$-distance emulator. Moreover, since we showed in **Problem 3 (a)** that even if $(x, y) \notin E$, $d_{H_{|E|}}(x, y) \leq (n-1) \cdot d_G(x, y)$ by the distance emulator property, H is an $(n - 1)$-stretch spanning tree of G.

Sub-subpart (B)

Finally, we set $\alpha = O(\log n)$ for a graph whose girth is strictly more than $g = \Theta(\log n)$. Given g, we know that

$$m \leq c \cdot \left(n + n^{1 + \frac{1}{\left\lfloor \frac{\log n}{2} \right\rfloor}} \right) = c \cdot n \cdot \left(1 + n^{\frac{2}{\log n}} \right) = c \cdot k \cdot n = O(n) \qquad \text{where } k \geq 5 \tag{34}$$

with the help of the following equation

$$\lim_{n \to \infty} 1 + n^{\frac{2}{\log n}} = 1 + \lim_{n \to \infty} \left(2^{\log n} \right)^{\frac{2}{\log n}} = 1 + 2^2 = 5 \tag{35}$$

$H \subseteq G$ is an $O(\log n)$-distance emulator, because one of two things can happen when considering an edge $e_i = (x, y) \in E$:

1. $d_{H_{i-1}}(x, y) \leq O(\log n) \cdot d_G(x, y)$. In this case, we leave H_{i-1} unchanged, as the distance emulator property is satisfied for e_i.
2. $d_{H_{i-1}}(x, y) > O(\log n) \cdot d_G(x, y)$. So, e_i is added to H_{i-1} to obtain H_i. Now, $d_{H_i}(x, y) = d_G(x, y) \leq O(\log n) \cdot d_G(x, y)$, i.e. the distance emulator property will now remain satisfied for e_i.

Hence, H is an $O(\log n)$-distance emulator of G with $O(n)$ edges.

Appendix

Problem 2 (a)

Finding the Prefix Sum Array

The algorithm to find the prefix sum array of the array of weights $W[1 : n-1]$, where $W[i] = d_G(i, i+1)$, is given in Algorithm 4. We pay close attention to the indices, as $|W| = n - 1$ but $|P| = n$.

Algorithm 4 Computing the prefix sum array

```
1: procedure PREFIX-SUM(W[1 : n − 1]):
2:     P[1 : n] ← 0
3:     P[2] ← W[1]
4:     for i ← 3 to n do
5:         P[i] ← P[i − 1] + W[i]
6:     end for
7:     return P
8: end procedure
```

Finding Medians for all Pairs of Vertices

The median of the distances between all pairs of vertices (u, v) for $u < v$ on the edge-weighted line graph G can be computed in $O(n^2)$ time using Algorithm 5.

Algorithm 5 Computing the median of all pairs of vertices

```
1: procedure PAIRWISE-MEDIANS(G = (V, E)):
2:     M[1 : n, 1 : n] ← 0
3:     for i ← 1 to n do                                   ▷ for_out
4:         m ← i
5:         for j ← i + 1 to n do                           ▷ for_in
6:             while d_G(i, m) < d_G(m, j) do
7:                 m ← m + 1
8:             end while
9:             M[i, j] ← m
10:        end for
11:    end for
12:    return M
13: end procedure
```

Strictly speaking, $M[i, j]$ stores the first vertex m which is closer to j than to i, which is what we require. $\textbf{for}_{\textbf{out}}$ runs for a total of $O(n)$ iterations. For each i, m starts at i and moves to the right. At the end of each iteration of the $\textbf{for}_{\textbf{out}}$, m can be at most n. This means that m travels a distance of at most $O(n)$ over the $O(n)$ iterations of $\textbf{for}_{\textbf{in}}$ for each i. Therefore, for each i, the total work done is $O(n)$. Hence, the overall time complexity of the algorithm is $O(n^2)$.

Backtracking to find the set C

Algorithm 6 describes the backtracking procedure to find the set C, assuming the dynamic programming table $\Phi[i, v, \kappa]$ is ready.

Algorithm 6 Backtrack to find the set C

```
1: procedure BACKTRACK(Φ[1 : n, 1 : n, 1 : k]):
2:     C ← {}
3:     i, v, κ ← n, n, k
4:     while |C| ≠ k do
5:         while Φ[i, v, κ] = Φ[i, v − 1, κ] do
6:             v ← v − 1
7:         end while
8:         C ← C ∪ {v}
9:         i, v, κ ← v − 1, v − 1, κ − 1
10:    end while
11:    return C
12: end procedure
```

The procedure takes only $O(n + k)$ time to find the set C. This is because we do not visit all states $[i, v, \kappa]$ in the table. Instead, we visit at most n states, where some states are visited for $\kappa = 1$, some for $\kappa = 2$, and so on. Over the k iterations of the outer **while** loop, we visit n states.
Informally, we can imagine it like a 3-dimensional staircase, where we start from the bottom and climb up when we find a vertex to add to C. We can only move up exactly k times, but we might have to move left and back at most n times.