601.435 / 601.635 Approximation Algorithms **Spring 2026**
Homework #2 **Due:** February 26, 2026, 9:00am

Reminder: you may work in groups of up to three people, but must write up solutions entirely on your own. Collaboration is limited to discussing the problems – you may not look at, compare, reuse, etc. any text from anyone else in the class. Please include your list of collaborators on the first page of your submission. Many of these problems have solutions which can be found on the internet – please don't look. You can of course use the internet (including the links provided on the course webpage) as a learning tool, but don't go looking for solutions.

Please include proofs with all of your answers, unless stated otherwise.

1 Min-cost bounded path (50 points)

Suppose are are given a directed acyclic graph $G = (V, E)$ with $|V| = n$ and $|E| = m$, a cost function $c : E \to \mathbb{N}$, a length function $\ell : E \to \mathbb{N}$, two nodes $s, t \in V$, and a length bound L. Our goal is to compute the minimum cost $s \to t$ path whose length is at most L. In other words, find the path P from s to t with $\sum_{e \in P} \ell(e) \le L$ which minimizes $\sum_{e \in P} c(e)$.

(a) (25 points) Give a pseudopolynomial-time exact algorithm for this problem (i.e., an algorithm which solves the problem but runs in time polynomial in n, m and $\max_{e \in E} c(e), \max_{e \in E} \ell(e)$).

Solution: For all $v \in V$ and $k \ge 0$, let $f(k, v)$ denote the length of the minimum-length $s - v$ path of cost at most k. Then since the minimum-length $s - v$ path of cost at most k must go through some final edge (u, v), we get the following (recursive) equation for f:

$$f(k,v) = \begin{cases} 0 & \text{if } v = s \\ \infty & \text{if } v \ne s \text{ and } k = 0 \\ \min_{u:(u,v)\in E} \left(\ell(u,v) + f(k - c(u,v), u)\right) & \text{otherwise} \end{cases}$$

It is easy to see that this equation is correct via a proof by contradiction. Clearly the equation is correct for the first two cases. Otherwise, suppose that there is some v, k so that $f(k,v) < \min_{u:(u,v)\in E} (\ell(u,v) + f(k - c(u,v), u))$, and let k be the smallest such value so that some such k, v exist. Then the minimum length path from s to v of cost at most k (call it P) goes through some final edge (w, v). Let P' denote the prefix of P ending at w. Note that P' has cost equal to $c(P) - c(w, v) \le k - c(w, v)$. Then

$$\begin{aligned} \ell(P') = \ell(P) - \ell(w,v) &< \min_{u:(u,v)\in E} \left(\ell(u,v) + f(k - c(u,v), u)\right) - \ell(w,v) \\ &\le \ell(w,v) + f(k - c(w,v), w) - \ell(w,v) = f(k - c(w,v), w) \end{aligned}$$

So P' is an $s - w$ path of cost at most $k - c(w, v)$ with total length less than $f(k - c(w, v), w)$. This contradicts the definition of f. Thus $f(k,v) \ge \min_{u:(u,v)\in E} (\ell(u,v) + f(k - c(u,v), u))$. On the other hand, clearly $f(k,v) \le \min_{u:(u,v)\in E} (\ell(u,v) + f(k - c(u,v), u))$ since for any such u the definition of $f(k - c(u, v), u)$ guarantees a path to v of the appropriate cost. Thus $f(k,v) = \min_{u:(u,v)\in E} (\ell(u,v) + f(k - c(u,v), u))$.

So now we know that this equation for f is correct. Now we claim that a dynamic program can fill in a table for f. Let $C = \max_{e\in E} c(e)$. Since all subproblems for $f(k, v)$ involve cost bounds $k' < k$, we can go bottom-up from $k = 0$ up to $k = nC$. So we can fill in this table in time number of entries $\times$ time per entry. There are clearly at most $n \cdot nC = n^2C$ table entries, and each can be computed in time $O(n)$, so we have a total running time of $O(n^3C)$.

Once the table is filled in, we can do a binary search on k to find the smallest k so that $f(k, t) \le L$. This will, by the definition of f, be the minimum cost necessary for an L-length bounded path. We can then use standard backtracking on the DP table to find the actual path. This all takes time at most polynomial in nC, m, and n, and hence the total running time is pseudopolynomial.

(b) (25 points) Using your algorithm from the previous part, give an FPTAS (a $(1+\varepsilon)$-approximation which runs in time polynomial in the size of the instance and $1/\varepsilon$, i.e., polynomial in $n, m, \log(\max_{e\in E} c(e)), \log(\max_{e\in E} \ell(e))$, and $1/\varepsilon$). Hint: Think about "guessing" the maximum cost of any edge that appears in the optimal path.

Solution: Let P^* be the optimal solution. There are at most $m < n^2$ distinct edge costs, so we will assume that we know the value $A = \max_{e\in P^*} c(e)$. Slightly more formally, we can try all different m possibilities, and take the best solution we find. If we can get a $(1 + \epsilon)$-approximation for each guess, then on the correct guess we will give a $(1 + \epsilon)$-approximation, so if a different guess gives an even better solution then we are still within $(1 + \epsilon)OPT$.

So assume that we know A. We first remove every edge e from the graph for which $c(e) > A$ (note that if A is correct then no such edge can be in the optimal solution). Let $c'(e) = \left\lceil \frac{c(e)}{A\epsilon/n} \right\rceil$. Our algorithm is simple: we run the exact algorithm from the previous part on the new instance in which everything is the same except we have costs c' rather than c, and use the solution that it finds.

Let's first analyze the running time. Note that $\max_{e\in E} c'(e) \le \frac{A}{A\epsilon/n} \le n/\epsilon$, so from our analysis of part a we get that the running time is at most $O(n^3 \cdot (n/\epsilon)) = O(n^4/\epsilon)$ (for a total running time of at most $O(n^6/\epsilon)$).

If the algorithm does not return any path, then there is no path of length at most L from s to t, so our guess for A was too low. Otherwise, if our guess for A is correct, let P be the path which our algorithm returns. Clearly P is feasible (is from s to t and has length at most L) since we did not change the lengths at all (just the costs).

So now we just need to analyze the approximation ratio. We get that

$$\begin{aligned}\sum_{e\in P} c(e) &\le \frac{A\epsilon}{n}\sum_{e\in P} c'(e) \le \frac{A\epsilon}{n}\sum_{e\in P^*} c'(e) \le \frac{A\epsilon}{n}\sum_{e\in P^*}\left(\frac{c(e)}{A\epsilon/n} + 1\right) = \sum_{e\in P^*} c(e) + |P^*|\frac{A\epsilon}{n}\\ &\le OPT + A\epsilon \le (1+\epsilon)OPT,\end{aligned}$$

where we used the definition of c', the fact that P is optimal for costs c', the definition of c' again, the fact that any min-cost path has at most n edges (since costs are nonnegative), and the fact that $A \le OPT$.

2 Min-degree Steiner Tree (50 points)

The Min-degree Steiner tree problem is the same as the Steiner tree problem, except instead of minimizing total cost our goal is to minimize the maximum degree. More formally, we are given an undirected graph $G = (V, E)$ and a subset $D \subseteq V$ of terminals. The goal is to find a tree T in G which spans all of D (but not necessarily all of V) and which minimizes the maximum degree.

Show how to modify the local search algorithm from class for min-degree spanning tree to get a local search algorithm for min-degree Steiner tree that runs in polynomial time and returns a tree with maximum degree at most $2\Delta^* + \log n$ (where Δ^* is the maximum degree of the optimal tree). If you're off by $+O(1)$ (like we were in class) that's OK, i.e., a bound of the form $2\Delta^* + \log n + c$ is OK for constant c.

Hints:

(1) You will need to slightly redefine a u-improvement, since adding a single edge might not create a fundamental cycle anymore. What kind of structure "acts" in a way equivalent to a non-tree edge in spanning trees?

(2) You might want to use the following structural graph theoretic result: in any tree with n nodes in which all non-leaves have degree at least d, the number of leaves is at least $\frac{d-2}{d-1}n$. You may use this without proof, although it's a good idea to convince yourself that it's true.

(3) You will also have to change the potential function when analyzing the running time. How can you change it so that the same basic idea from class works?

Solution: Given a graph $G = (V, E)$ and a tree $T = (V', E')$ which is a subgraph of G, define a "tree-avoiding path" to be a path in G between two nodes $u, v \in V'$ such that no internal nodes in the path intersect the tree. More formally, P is a tree-avoiding $u - v$ path if $P \cap V' = \{u, v\}$ and P is a $u - v$ path in G. Note that every tree-avoiding path P induces one cycle in $T \cup P$, which we call the fundamental cycle of P.

We will define a u-improvement similarly to how we defined it for the spanning tree setting, but with tree-avoiding paths in place of non-tree edges. In particular, for a tree avoiding $v - w$ path P, we will say that $\{u, P\}$ is a u-improvement if $u \in T$, there is an edge $\{u, x\} \in E(T)$ that is on the fundamental cycle of P, and $\max(d_{T'}(v), d_{T'}(w)) \le d_{T'}(u) = d_T(u) - 1$ (where T' is the tree obtained by adding P and removing $\{u, x\}$ from T).

We will use the following algorithm, which is similar to the spanning tree algorithm analyzed in class but with the new idea of u-improvement as well as an additional pruning step. We start with an arbitrary minimal steiner tree T (a steiner tree where removing any edge makes it infeasible). Then we do the following while loop. While there exists some u where a u-improvement exists and $d_T(u) \ge \Delta(T) - \log n$:

(a) Do the u-improvement

(b) Prune T so that it is minimal (removing any edge results in two trees, neither of which is a feasible steiner tree).

As in the spanning tree algorithm, we will define a locally optimal tree (a LOT) to be a minimal feasible tree T in which there are no u-improvements for any u with $d_T(u) \ge \Delta(T) - \log n$. Let T^* be the optimal steiner tree, with maximum degree Δ^*.

Lemma 1 *Let T be a LOT. Then $\Delta(T) \leq 2\Delta^* + \lceil \log n \rceil$.*

Proof: We follow the same basic outline as in class. Let S_i denote the nodes with degree at least i in T. Then as shown in class (Claim 6.3.7 in the lecture notes), there is some $i \geq \Delta(T) - \log n$ such that $|S_{i-1}| \leq 2|S_i|$. Fix this i. Let E_i^T denote the edges of T that are incident on nodes in S_i. Then the proof of Claim 6.3.9 from class is still true, so we know that $|E_i^T| \geq (i-1)|S_i| + 1$. Let $\mathcal{C}$ denote the set of all components of $T \setminus E_i^T$, and let $\mathcal{C}'$ be the components of $T \setminus E_i^T$ *other* than the singleton components consisting of the elements of S_i. Hence $|\mathcal{C}| = |E_i^T| + 1 = (i-1)|S_i| + 2$, while $|\mathcal{C}'| = |\mathcal{C}| - |S_i| = (i-2)|S_i| + 2$.

Let Q_i denote vertices in V which, in $T - E_i^T$, are incident on tree-avoiding paths between different components in $\mathcal{C}$, or on a tree edge between different components (i.e., an edge in E_i^T). In other words, every tree-avoiding path or edge between components in $\mathcal{C}$ has both endpoints in Q_i.

Claim 2 *Let $x, y \in Q_i$ be in different components of $T - E_i^T$ (components in $\mathcal{C}$) where there is either a tree-avoiding path or an edge between x and y. Then either x or y is in S_{i-1}.*

Proof: Suppose there is a tree edge between x and y. Then since x and y are in different components of $T - E_i^T$, this edge must be in E_i^T, and thus at least one of x, y is in S_i and thus in S_{i-1}.

On the other hand, suppose that there is a tree-avoiding path P between x and y. Then by the definition of Q_i, there is some vertex u with $d_T(u) \geq i$ on the path from x to y in T. If $d_T(x)$ and $d_T(y)$ are both less than $i-1$, then (u, P) would be a u-improvement, and by definition of i we know that $d_T(u) \geq \Delta(T) - \log n$. Thus T would not be a LOT, which is a contradiction to the definition of T. ∎

So now we know that for every edge or tree-avoiding path between two different components of $T - E_i^T$, at least one endpoint is in S_{i-1}. We can now use a similar argument to Lemma 6.3.5 from the lecture notes to provide a lower bound on Δ^*. In particular, consider the components $\mathcal{C}'$ of $T - E_i^T$ excluding the singletons of S_i. Divide these into two types: Let $\mathcal{T}$ be the set of *terminal components*, which have at least one terminal in them, and let $\mathcal{S}$ be the *steiner components* which have no terminals. Note that since we do the pruning step, all of the leaves of T are terminals. So every component in $\mathcal{C}'$ which contains a leaf of T is a terminal component.

Now consider the following auxiliary tree T'. We first arbitrarily root T. Note that every component $c \in \mathcal{C}'$ is a subtree. We say that one component c is the parent of another component c' if the path from the root of c' to c only goes through ancestors of that root, and if all of the nodes on that path are in S_i. To create T', we create a vertex for every component of $\mathcal{C}'$, and we add an edge $\{c, c'\}$ between two of these vertices if one component is the parent of the other. Note that this is a tree on $|\mathcal{C}'|$ nodes in which every internal node has degree at least $i-1$. Hence the number of leaves in this tree (by the hint) is at least

$$\frac{i-3}{i-2}|\mathcal{C}'| \geq \frac{i-3}{i-2}\left((i-2)|S_i| + 2\right) = (i-3)|S_i| + 2i - 4$$

Each leaf in T' correspond to a a component in $\mathcal{C}'$ with at least one leaf in it, and hence at least one terminal, and hence a terminal component. So $|\mathcal{T}| \geq (i-3)|S_i| + 2i - 4$.

Now we note that Lemma 6.3.5 from the lecture notes still holds for this particular set: each set in $\mathcal{T}$ has one terminal (and no terminal is in more than one of these components), so the optimal tree must use at least $|\mathcal{T}| - 1$ edges to connect these terminals together, and since the terminals are in different components of $\mathcal{C}$ then each of these edges must have at least one endpoint in S_{i-1} by the previous claim. Thus

$$\Delta^* \geq \frac{|\mathcal{T}|-1}{|S_{i-1}|} \geq \frac{(i-3)|S_i|+2i-5}{2|S_i|} \geq \frac{i-3}{2} \geq \frac{\Delta(T)-\log n-3}{2}.$$

Rearranging, we get that $\Delta(T) \leq 2\Delta^* + \log n + 3$. ■

Lemma 3 *This algorithm completes in polynomial time.*

Proof: We first analyze the running time of each iteration, and then analyze the number of iterations. In each iteration, we need to check for every pair of nodes whether there is a tree-avoiding path between them, and if so whether the fundamental cycle it causes has a node of degree at least i. This can be done in polynomial time by removing all tree nodes from the graph and running any path-finding algorithm (e.g., DFS or BFS) to see if the two nodes are still connected.

So the only question is the number of iterations. We will use the following potential function:

$$\Phi(v) = \begin{cases} 1 & \text{if } d_T(v) \leq 2 \\ 3^{d_T(v)-2} & \text{if } d_T(v) \geq 3 \end{cases}$$
$$\Phi(T) = \sum_{v \in V} \Phi(v)$$

Note that Steiner nodes which do not appear in T have degree 0 in T, so still contribute 1 to the potential of T.

Now it is not hard to see that the analysis from class still basically holds. We know that $\Phi(T) \geq n$ (since every node contributes at least one to the potential), and we know that $\Phi(T) \leq n \cdot 3^n$. If we make a u-improvement for some u with $d_T(u) = i \geq \Delta(T) - \log n$, then $\Phi(u)$ decreases by $3^{i-2} - 3^{i-3} = 2 \cdot 3^{i-3}$. The potential stays the same at all of the nodes in the tree-avoiding path except the endpoint (since the internal nodes go from degree 0 to degree 2, so still have potential 1). The two endpoints *increase* the potential by at most $3^{i-3} - 3^{i-4} = 2 \cdot 3^{i-4}$. Hence the total decrease in the potential is at least

$$\begin{aligned} 2 \cdot 3^{i-3} - 4 \cdot 3^{i-4} &= \frac{2}{3}3^{i-2} - \frac{4}{9}3^{i-2} = \frac{2}{9}3^{i-2} \geq \frac{2}{9} \cdot 3^{\Delta(T)-\log n-2} = \frac{2}{9 \cdot 3^{\log n}}3^{\Delta(T)-2} \\ &= \frac{2}{9n^{\log 3}}3^{\Delta(T)-2} \geq \frac{2}{9n^2} \cdot \frac{1}{n}\Phi(T) = \frac{2}{9n^3}\Phi(T) \end{aligned}$$

Thus the potential drops by a $(1 - \frac{2}{9n^3})$ multiplicative factor in each iteration. Now (as we showed in class) this means that after at most $\frac{9}{2}n^4 \ln n$ iterations the potential is at most n. Thus the algorithm is finished after at most that many iterations. ■