CompSci 330 Design and Analysis of Algorithms
Assignment 4, Fall 2024 Duke University

TODO: Add your name(s) here

Due Date: October 7, 2024

How to Do Homework. We recommend the following three step process for homework to help you learn and prepare for exams.

1. Give yourself 15-20 minutes per problem to try to solve on your own, without help or external materials, as if you were taking an exam. Try to brainstorm and sketch the algorithm for applied problems. Don't try to type anything yet.
2. After a break, review your answers. Lookup resources or get help (from peers, office hours, Ed discussion, etc.) about problems you weren't sure about.
3. Rework the problems, fill in the details, and typeset your final solutions.

Typesetting and Submission. Your solutions should be typed and submitted as a single pdf on Gradescope. Handwritten solutions or pdf files that cannot be opened will not be graded. LaTeX[1] is preferred but not required. You must mark the locations of your solutions to individual problems on Gradescope as explained in the documentation. Any applied problems will request that you submit code separately on Gradescope to be autograded.

Writing Expectations. If you are asked to provide an algorithm, you should clearly and unambiguously define every step of the procedure as a combination of precise sentences in plain English or pseudocode. If you are asked to explain your algorithm, its runtime complexity, or argue for its correctness, your written answers should be clear, concise, and should show your work. Do not skip details but do not write paragraphs where a sentence suffices.

Collaboration and Internet. If you wish, you can work with a single partner (that is, in groups of 2), in which case you should submit a single solution as a group on Gradescope. You can use the internet, but looking up solutions or using LLMs is unlikely to help you prepare for exams. See the homework webpage for more details.

Grading. Theory problems will be graded by TAs on an S/I/U scale. Applied problems typically have a separate autograder where you can see your score. See the course policy webpage for details about homework grading.

[1] If you are new to LaTeX, you can download it for free at latex-project.org or you can use the popular and free (for a personal account) cloud-editor overleaf.com. We also recommend overleaf.com/learn for tutorials and reference.

Problem 1 (Decycle). A *decycler* $S \subseteq E$ of a connected, edge-weighted, undirected graph $G = (V, E)$ is a subset of edges that contains at least one edge from every cycle in the graph. If the edges of a decycler are removed from the graph, the resulting graph is acyclic. Design an $O((m + n) \log n)$-time algorithm to find a minimum-weight decycler in a connected, undirected, edge-weighted graph G with positive edge weights, where n and m are the number of vertices and edges in G, respectively. Justify the correctness of your algorithm and analyze its running time.

Solution 1 . **Algorithm.** Let $w(e)$ denote the weight of any edge $e \in E$. First, obtain a modified graph G' by multiplying each edge weight in G by -1, i.e., set $w'(e) = -w(e)$. Then compute a T be a minimum spanning tree of G'. Return the subset $S \subseteq E$ that are **not** in E as the desired minimum-weight decycler of G.

Correctness. Let S be any minimum-weight decycler of G, and let $G^- = (V, E \setminus S)$ be the the subgraph of G obtained by removing the edges of the decycler. We claim that G^- is a **maximum-weight** spanning tree of G. Assuming this holds, it then suffices to show that the minimum-weight spanning tree T of G' computed in our algorithm is a maximum-weight spanning tree of G, as our algorithm returns the edges of G not in this tree, which is a minimum-weight decycler by the previous claim.

We first argue G^- is a spanning tree of G. Indeed, by definition of decycler, G^- is acyclic. Furthermore, since the edges of G have positive weights and G is connected, if G^- is disconnected (and thus not a spanning tree) there is some edge $e \in S$ that can be added back to G^- while remaining acyclic. Then $S - \{e\}$ would be a decycler with less weight than minimum-weight decycler S, a contradiction. So G^- is acyclic and connected, i.e., a spanning tree of G.

Next, we argue G^- is a maximum-weight spanning tree of G. Let $w(\cdot)$ be the weight of any subgraph or set of edges. For sake of contradiction, suppose there is a spanning tree T of G with $w(G') < w(T)$. Then let $S' = \{e \in E \mid e \notin T\}$ be the edges not in T, which is a decycler. By definition of minimum-weight decycler S, we have

$$w(E) - w(S) = w(G') < w(T) = w(E) - w(S').$$

This implies $w(S') < w(S)$, which contradicts that S is a minimum-weight decycler.

Finally, we show that minimum-weight spanning tree T of G' is a maximum-weight spanning tree of G. By definition, $w'(T) = -w(T)$. For sake of contradiction, suppose there is a spanning tree T' of G with $w(T) < w(T')$. But then $w'(T') = -w(T') < -w(T) = w'(T)$, which implies $w'(T) > w'(T')$, which contradicts that T is a minimum-weight spanning tree of G'.

Runtime. Computing G' takes $O(n + m)$ time, and running, say, Prim's algorithm to obtain the MST T of G' takes $O((m+n) \log n)$ time. Finally, it takes $O(m)$ time to determine the set of edges not in T, which are returned. Thus the overall runtime is $O((m + n) \log n)$.

Problem 2 (Leafy Trees). Let $G = (V, E)$ be a connected, edge-weighted, undirected graph and let $S \subset V$ be a subset of vertices. A spanning tree T of G is *S-leafy* if all vertices of S are leaves of T. Design an $O((m+n)\log n)$-time algorithm that either computes an S-leafy spanning tree T of G of minimum total weight, or reports that no such tree exists, where n and m are the number of vertices and edges in G, respectively. Justify the correctness of your algorithm and analyze its running time. (Note that an S-leafy spanning tree may have leaves that are not in S as well.)

Solution 2 . Algorithm. Let G' be the subgraph of G induced by $V \setminus E$; let $G' = (V \setminus E, E')$ where $E' = \{uv \in E \mid u, v \in V \setminus S\}$. The edges E' of G' are where neither endpoint are vertices in S. We first check if G' is connected; if not, we report that no S-leafy spanning tree exists. Otherwise, we compute a minimum-weight spanning tree T' of G'. Then, for each vertex $v \in S$, we greedily attach it to T' as a leaf via its minimum-weight edge of the form $uv \in E$ where $u \notin S$. If, for some vertex $v \in S$ there is no such edge, we report that no S-leafy spanning tree exists. In the end, we return the spanning tree T of G obtained by attaching all vertices of S.

Correctness. We first claim that any S-leafy spanning tree T of G contains a minimum-weight spanning tree T' of G'. Assuming this claim, clearly our algorithm attaches each vertex of S to T' using minimum additional weight, which implies it is correct.

To prove the claim, let T' be the subgraph obtained by removing the leaves of S from T. Since only leaves are removed, T' remains connected on the vertices of $V \setminus S$. Indeed, the unique path in T between two vertices not in S must not include any vertices in S, and thus these paths remain in T. Clearly it is acyclic, so T' is a spanning tree of G'. For sake of contradiction, suppose there is a spanning tree T'' of G' with $w(T'') < w(T')$, where $w(\cdot)$ is the total weight of the given edge set or subgraph. Then let $E_S = \{uv \in E \mid u \in S, v \in V \setminus S\}$ be the edges of T that connect S as leaves in T. Then adding E_S to T'' results in an S-leafy spanning tree of G with weight $w(T'') + w(E_S) < w(T') + w(E_S) = w(T)$, which contradicts that T is a minimum-weight S-leafy spanning tree.

Runtime. Constructing G' takes $O(m+n)$ time by iterating through the vertices and edges of G. First checking if G' is connected takes $O(n+m)$ time using BFS, then computing MST T' of G' takes $O((n+m)\log n)$ time, if it exists. Finally, picking the minimum-weight edge to attach each vertex $v \in S$ to T', if possible, takes $O(m)$ overall. We conclude that the overall runtime is $O((n+m)\log n)$.

Problem 3 (Greyish Trees). Let $G = (V, E)$ be a connected, unweighted, undirected graph, where each edge is colored grey or blue, and let k be an integer. In this problem, you will describe a linear-time algorithm that either returns any spanning tree T of G that uses at most k blue edges, or reports that no such tree exists, where n and m are the number of vertices and edges in G, respectively.

(a) Reduce the problem to computing a minimum spanning tree (MST) in G by assigning appropriate weights to its edges. Describe how, after computing an MST T in G with your weights, T is used to answer the given problem of finding a spanning tree with at most k blue edges.

(b) Suppose you use Prim's algorithm on G with your edge weights to find the MST. Describe how to modify the algorithm specifically for inputs with edge weights of the kind you assigned so that it runs in only $O(n + m)$ time instead of $O((n + m) \log n)$ time.

Solution 3

(a) Assign weight 0 to all grey edges and 1 to all blue edges, then compute a minimum-weight spanning tree T in G. If the weight of T is at most k, then T itself is a tree with at most k blue edges, otherwise no such tree exists.

(b) The standard implementation of Prim's algorithm performs $O(n + m)$ priority-queue operations, where each takes $O(\log n)$ time. However, the graph we wish to run Prim's on has the special property of having only two different edge weights. Thus, we can replace the standard priority queue with a linked list, as follows. The standard priority queue supports insertions, decrease-key operations, and popping a minimum-priority element.

- To insert a vertex with priority (edge weight) 0, we add it to the front of the list, otherwise it has priority (edge weight) 1, and we add it to the end of the list. This takes $O(1)$ time.
- Instead of supporting the decrease-key operation, we simply insert the vertex with its priority to the list, even if it is already present in the queue. This takes $O(1)$ time.
- To pop a minimum-priority vertex from the list, we remove the head of the list in $O(1)$ time. Prim's algorithm uses this vertex, call it v, as a candidate for connecting it to the current tree with an edge. Since our queue may have the same vertex many times (due to our replacement for decrease-key), we need to check if v has already been connected, which can be checked in $O(1)$ by marking vertices when they are connected the first time.

Each of the three operations takes $O(1)$ time, instead of $O(\log n)$ with the standard priority queue. Each vertex is added to the list at most one time per each of its incident edges, and thus $O(n + m)$ list operations at $O(1)$-time each are performed throughout the algorithm. The overall runtime is thus $O(n + m)$.

Problem 4 (Applied). The town of Thirtyville is preparing for a large expansion. Naturally, the mayor of Thirtyville needs to decide how the electrical network for the town should be designed. Let $G = (V, E)$ be a connected undirected graph with positive edge weights $w : E \to \mathbb{R}^+$ representing the expanded town. The vertices in V represent locations and the edges in E represent potential power lines that can be built, where the weight $w(e)$ of an edge $e \in E$ is the cost in dollars to build that power line. Lastly, you are also given a parameter $z > 0$, which is the cost to build a power plant at any location $v \in V$.

Your goal is to select a subset of locations $P \subseteq V$ at which to build power plants **and** select a subset of power lines $L \subseteq E$ to build so that every location in V is reachable from a power plant $p \in P$ via the power lines in L. Furthermore, the total cost of the power plants and power lines must be minimized.

For example, a feasible solution with cost $z|V|$ is to set $P = V$ and $L = \varnothing$, meaning that a power plant is built at every location, which is valid because every location is trivially reachable from itself with a power plant. For another example, a feasible solution is to build a single power plant (pick any one location in V) and then pick the cheapest set of power lines that connect all locations it in some fashion. In between these extremes is the possibility to build more than one but less than $|V|$ power plants.

We represent graphs with vertices as integers from 0 to $n - 1$, where n is the number of vertices. Your task is to **design and implement** an algorithm that, given $G = (V, E)$ with real edge weights and positive parameter z, Your solution will need to have an empirical runtime that is within constant factors of an reference solution with $O((m + n) \log n)$ worst-case runtime, where n and m are the numbers of vertices and edges in G, respectively.

Language-specific details follow. You can use whichever of Python or Java you prefer. You will receive automatic feedback when submitting, and you can resubmit as many times as you like up to the deadline.

- **Python.** You should submit a file called `expansion.py` to the Gradescope item "Homework 4 (Python)." The file should define (at least) a top level function `expansion` that takes in an int n, the number of vertices, an float z, the cost to build a power plant, and a list `edges` of tuples `(int, int, float)` representing the edges, where `edges[i]` represents an edge from `edges[i][0]` to `edges[i][1]` with weight `edges[i][2]`. The function returns the minimum possible cost of power plants P and power lines L to connect all locations to power plants. The function header is:

```
def expansion(n:int, z:float, edges:list[(int, int, float)]):  -> float
```

- **Java.** You should submit a file called `Expansion.java` to the Gradescope item "Homework 4 (Java)." The file should define (at least) a top level function `Expansion` that takes in an int n, the number of vertices, a double z, the cost to build a power plant, and two arrays that represent the edges and their real weights: a 2D $m \times 2$ array `int[][] edges` and a m-length 1D array `double[] weights`, where `edges[i]` is the directed edges from `edges[i][0]` to `edges[i][1]` and its weight is `weight[i]`. The function returns the minimum possible cost of power plants P and power lines L to connect all locations to power plants. The method header is:

```
public double expansion(int n, float z, int[][] edges, double[] weights);
```

Solution 4 . There are many correct approaches. To motivate one, see that building a power line with cost z or greater is never preferable to building a power plant for z cost at one of its endpoints appropriately. Thus, we can remove all power lines with cost z or greater, then run Kruskal's algorithm. In the end, the spanning *forest* of G corresponds to paying z for each tree in the forest to open a power plant at any location in each tree, plus the cost to build all power lines in the trees. This solution is optimal and takes $O((m+n)\log n)$ time. (Prim's can be used instead, but it may need to be ran multiple times, each time starting at a disconnected vertex. The runtime is the same.)