CS 577: Introduction to Algorithms — Greedy Review Problems

Problems

1. Let $G = (V, E)$ be a connected undirected graph. We will call a subgraph (V, E') *special* if it is connected and has exactly one cycle.

   (a) Show that a subgraph is special iff it can be obtained by taking a spanning tree and adding one edge.

   (b) Let G have non-negative weights on its edges. Suppose all edge weights are distinct. Describe and analyze efficient algorithm that finds a minimum-weight special subgraph of G, assuming there is one.

2. Suppose we have a connected graph $G = (V, E)$. Each edge e has a time-varying edge cost given by a function $f_e(t) = a_e t + b_e$ where $f_e(t)$ gives the cost of the edge at time t. Observe that the set of edges constituting the minimum spanning tree of G may change over time, and that the cost of the minimum spanning tree of G can be expressed as a function of the time t. Denote this function $cost_G(t)$.

   Give an algorithm that takes as input: – the graph G; – the values $\{(a_e, b_e) : e \in E\}$; – an interval $[\ell, r]$; and returns a value of the time $t \in [\ell, r]$ at which the minimum spanning tree has minimum cost, $\min_{t \in [\ell,r]} cost_G(t)$. Your algorithm should run in time polynomial in the number of nodes and edges of the graph G, where you may assume that arithmetic operations on numbers can be done in constant time.

3. Given a connected graph $G = (V, E)$ with weights w_e on edges and a spanning tree T, define the width of T to be the weight of the maximum weight edge in T.

   (a) Given a target width W, develop a linear time algorithm to determine whether G contains a spanning tree of width at most W.

   (b) Develop a linear time algorithm to find the smallest W such that there exists a spanning tree T with width W in G. You may assume that all edge weights are distinct.

   (Hint: Use your algorithm from part (a) as a subroutine, and do a sort of "binary search" over W.)

4. Consider a path between two vertices s and t in an undirected weighted graph G. The *bottleneck length* of this path is the maximum weight of any edge in the path. The *bottleneck distance* between s and t is the minimum bottleneck length of any path from s to t. (If there are no paths from s to t, the bottleneck distance between s and t is ∞.)

   Describe and analyze an efficient algorithm to compute the bottleneck distance between every pair of vertices in an arbitrary undirected weighted graph. Assume that no two edges have the same weight.