28 (100 PTS.) Skip distance.

Let $\mathsf{G} = (\llbracket n \rrbracket, \mathsf{E})$ be a connected undirected graph with positive distinct costs on its edges (the costs are real numbers), with n vertices and m edges, where $\llbracket n \rrbracket = \{1, 2, \ldots, n\}$. You can assume here that $n \leq m$. Here, the cost of an edge $e \in \mathsf{E}$ is denoted by $c(e)$.

28.A. (30 PTS.) Given a set $X \subseteq \mathsf{E}$, Let $C_1, \ldots, C_k$ be the connected components of the graph $\mathsf{J} = (\llbracket n \rrbracket, X)$. Consider the reduced graph $\mathsf{G}/X = (\llbracket k \rrbracket, \mathsf{E}')$, where there is an edge $i'j' \in \mathsf{E}'$, if and only if there exists an edge $ij \in \mathsf{E}$, such that $i \in \mathsf{V}(C_{i'})$ and $j \in \mathsf{V}(C_{j'})$. If $i'j' \in \mathsf{E}'$, then

$$c(i'j') = \min_{ij \in \mathsf{E}: i \in \mathsf{V}(C_{i'}), j \in \mathsf{V}(C_{j'})} c(ij).$$

Describe in detail an algorithm, as fast as possible, that computes the reduced graph G/X. You might need to use radix sort here[1]. A solution using hashing is worth no points at all.

Solution:

We compute the connected components of $\mathsf{J} = (\llbracket n \rrbracket, X)$ using **DFS**. This takes $O(n + m)$ time. We use **DFS** to propagate for each connected component of J its index (i.e., all the vertices in the ith connected component are assigned index i). For a vertex $u \in \mathsf{V}(\mathsf{G})$, let $\mathrm{idx}(u)$ be its ***index***.

For an each edge $uv \in \mathsf{E}$, we define

$$\gamma(uv) = \begin{cases} 0 & \mathrm{idx}(u) = \mathrm{idx}(v) \\ \min(\mathrm{idx}(u), \mathrm{idx}(v)) \cdot n + \max(\mathrm{idx}(u), \mathrm{idx}(v)) & \text{otherwise.} \end{cases}$$

Clearly, $\gamma(uv)$ be computed in constant time, and for any two edges uv, xy of G that connects two different connected components of J, we have that $\gamma(uv) \neq \gamma(xy)$. Furthermore the value of $\gamma(\cdot)$ is always in the range $1, \ldots, n^2$.

We scan the edges of E, copying only edges with non-zero γ value into a new set of edges Y. We sort the edges of Y by their γ value, using radix sort, which takes $O(m)$ time (here we are using that $m \geq n$). In the sorted list, all the edges that connect the same two connected components of J are consecutive. We break the list of edges into sublist, where each sublist all the edges have the same γ value.

The algorithm now initialize the graph $(\llbracket k \rrbracket, \emptyset)$. next, the algorithm scans the sorted list of edges. For each group, we know the two connected components i' and j' that these edges connect. The algorithm scans the group to find the min cost edge, and inserts into the new graph the edge $i'j'$ with the minimum cost computed. Clearly, the resulting graph is the desired graph G/X.

As for running time, running **DFS** twice takes $O(n+m)$ time. Radix sorting the edges takes $O(n+m)$ time, and creating the new graph takes $O(1)$ time, since inserting an edge takes constant time.

[1] Recall that radix sort allows you to sort n integer numbers that are in the range $1, \ldots, n^{O(1)}$ in $O(n)$ time (if you do not know what radix sort is, read the wikipedia page).

28.B. (20 PTS.) The ***skip*** price of a path π in G is $s(\pi) = \max_{e \in \pi} c(e)$. The ***skip distance*** for two vertices $i, j \in \mathsf{V}(\mathsf{G})$, is

$$s(i,j) = \min_{\pi:\text{path connecting } i \text{ to } j \text{ in } \mathsf{G}} s(\pi).$$

Given two vertices $u, v \in [\![n]\!]$, and a real number α, describe an algorithm, as fast as possible, that decides if $s(u,v) > \alpha$.

Solution:

The algorithm compute the set of edges

$$\mathsf{E}_{\leq \alpha} = \{e \in \mathsf{E} \mid c(e) \leq \beta\}$$

in linear time. If u and v are in the connected component of $([\![n]\!], \mathsf{E}_{\leq \alpha})$ then u and v are connected by a path of skip distance $\leq \alpha$. Otherwise, the skip distance between u and v is strictly larger than α, and the algorithm returns this result.

28.C. (50 PTS.) For a number β, let

$$\mathsf{E}_{\leq \beta} = \{e \in \mathsf{E} \mid c(e) \leq \beta\},$$

be the set of edges in G with cost at most β.

A natural algorithm for computing the skip distance between u and v is to pick some value α, and decide if $s(u,v) > \alpha$, and if so, the algorithm recurses on computing the skip distance between u' and v' in $\mathsf{G}' = \mathsf{G}/\mathsf{E}_{\leq \alpha}$, where u' and v' are the two meta vertices of G' that their original connected components contains u and v, respectively. Otherwise, if $s(u,v) \leq \alpha$, the algorithm computes the skip distance between u and v in the graph $([\![n]\!], \mathsf{E}_{\leq \alpha})$. If the graph has constant size, the algorithm computes the skip distance directly using brute force.

First prove that this algorithm is correct. Next, describe how to implement this algorithm efficiently, so that the resulting algorithm is as fast as possible (in particular, what is the right value of α to pick, and how do you compute it?). What is the running time of your algorithm as a function of n and m?

Solution:

The algorithm computes α to be the median cost of the edges in E. This takes $O(m)$ time using median selection. The algorithm next checks if $s(u,v) > \alpha$ using the algorithm from the previous part. If it is, then it recurses on G'. Otherwise, it recurses on the connected component of graph $([\![n]\!], \mathsf{E}_{\leq \alpha})$ that contains both u and v. When the graph has less than, say, 10 edges, the algorithm keep deleting edges in deceasing cost, till separating u and v. The cost of the edge separating u and v, is the skip distance between the two vertices.

Correctness.

Lemma 10.1. *The algorithm computes the skip distance between u and v correctly.*

Proof: The proof is by induction on the size of the graph (i.e., $n + m$). The base of induction is clear, as the reverse deletion algorithm described is clearly correct.

If $s(u,v) \leq \alpha$, then the algorithm recurses on a smaller graph, clearly the optimal path is not using any edge of cost $> \alpha$, and as such, by induction the result returned by the algorithm is correct.

We remain with the case that $s(u,v) \leq \alpha$. Consider the path π in G with the minimum skip price between u and v, and observe that it corresponds to a path between u' and v' in $\mathsf{G}' = \mathsf{G}/\mathsf{E}_{\leq\alpha}$, with the same maximum cost edge. Similarly, a path σ' between u' and v' in G', can be modified into a path in G, such that each new edge has price $\leq \alpha$, and the old edges are replace by their original edge in the original graph having the same cost. Thus, the skip distance between u' and v' in G' is the same as the skip distance between u and v in G in this case. Which, again, by induction implies the correctness of the algorithm. ■

Running time analysis. As for implementation, part (B) describes how to decide if $s(u,v) > \alpha$ in $O(n+m)$ time. Part (A) describes how to compute G' in the same time. We get the following recurrence

$$T(m) \leq O(m) + T(m/2)$$

and the solution for this recurrence is $O(m)$, which bounds the overall running time. Critically, the algorithm recurrences on the connected component that contains the two vertices of interest. This ensures that the number of edges is always (up to one) at least the number of vertices in the graph. We are also using here the fact that $m \geq n$ in the input graph.

29 (100 PTS.) More on MST.

29.A. (40 PTS.) THE SPANNING GRAPH THAT SURVIVES.

Let $\mathsf{G} = (\mathsf{V}, \mathsf{E})$ be a connected graph on n vertices and m edges, with unique costs on the edges. A subgraph H of G is a ***MST survivor***, if for any edge $e \in \mathsf{E}$, we have that $\mathsf{G} - e$ and $\mathsf{H} - e$ have the same minimum spanning forest, where $\mathsf{G} - e$ denotes the graph G after we delete the edge e from it. Present an algorithm, as fast as possible, that computes a MST survivor of G that has a minimum number of edges among all such graphs. How fast is your algorithm? Prove that your algorithm is correct.

Solution:

Let us start with the naive algorithm. We start by computing a MST T of G. For every edge $e \in T$, compute the MST of $\mathsf{G} - e$. This can be done in $O(nm)$ time with a bit of care, since you are looking for the cheapest edge in the cut that corresponds to e. Let $r(e)$ be this replacement edge. Clearly, T together with all the replacement edges is the minimum MST survivor graph, since we computed for every edge e for the MST what edge replaces it if e is deleted. Since the replacement edge is uniquely defined, this implies this algorithm is correct, and returns the correct result. Its running times $O(nm)$.

This solution is worth 30 points.

Somewhat faster algorithm. We start by computing a MST T of G. Next, we remove T from G, and compute an MST T' for the graph $\mathsf{G} - T$. Consider the graph formed by $T \cup T'$. We run the previous algorithm on $T \cup T'$ and the resulting algorithm is the MST survivor algorithm.

Lemma 10.2. *The graph $T \cup T'$ is a MST survivor graph.*

Proof: If the deleted edge e is not in T then there is nothing to prove. Otherwise, consider the deleted edge e, and let U be the set of vertices in one of the connected components of $T - e$.

If $\mathsf{G} - e$ is disconnected, then there is nothing to prove as $T - e$ spans both connected components correctly.

The edge e is the minimum cost edge in the cut $(U, \mathsf{V} \setminus U) = \{uv \in \mathsf{E} \mid u \in U, v \in \mathsf{V} \setminus U\}$. If we delete it, then the replacement edge, is the cheapest edge in $(U, \mathsf{V} \setminus U) - e$. Which is also the cheapest edge in this cut, for the graph $\mathsf{G} - T$ (since T contains only one edge in its cut). It follows that the replacement edge must be in T', which implies the claim that $T \cup T'$ is a MST survivor graph. ∎

This implies that $T \cup T'$ contains all the replacement edges, and the above algorithm computes them correctly. The running time of this algorithm is $O(n^2)$. A fast algorithm for this problem is probably possible, but that is more than enough.

What is false. It is not true that $T \cup T'$ is the desired graph. Consider the complete graph on the set of vertices v_1, v_2, v_3, v_4, where all edges have cost 1, except for the diagonals v_2v_4, v_1v_3 which have cost 2. The minimum MST survivor graph in this case is the cycle $v_1v_2v_3v_4v_1$, but $T \cup T'$ is the whole graph.

29.B. (40 PTS.) LINEAR TIME MST.

Some Nigerian prince, that needs your help, sent you as a gift a black box B and a graph G. The graph G has n vertices and m edges, and real positive distinct weights on its edges. The black box can computes the MSF (minimum spanning forest) of any graph with n' vertices, and at most $(3/2)n'$ edges, in $O(n')$ time (it will compute the minimum spanning forest of this graph if it is not connected). You can use the box B only on a graph with n' vertices, such that $n' \leq n$.

Present an algorithm that computes the MST of G, in $O(m)$ time, using this black box (you can safely assume that $m \geq n$). Prove the correctness of your algorithm.

Solution:

If one uses the black box with $n' = m$ (i.e., vertex padding), then one can just call the black box on the whole graph and be done. This was definitely not the intended solution, but because of a typo in an earlier version of this question, we will accept such a solution as a correct solution in this case.

Solution:

In the following, for a set of edges $X \subseteq \mathsf{E}$, let $\mathrm{mst}(X)$ denote the unique minimum spanning forest of the graph (V, X).

Algorithm. Break the edges of the graph into groups $E_1, \ldots, E_k$, where $|E_i| = n/2$, expect for the last group that might be smaller. Using the black box, computer the spanning forest of E_1, and let T_1 be this forest. For $i = 2, \ldots, k$, compute the MSF T_i of the graph $T_{i-1} \cup E_i$. This takes $O(n)$ time using the black box. Let T_k be the last MST computed. The algorithm outputs T_k as the desired MST.

running time. The black box is called $\lceil m/(n/2) \rceil$ times, which implies running time of $O(m + (m/(n/2))n) = O(m)$.

Correctness. We need the following lemma, that shows that we can add edges to the MST and recompute it to get a new bigger MST.

Lemma 10.3. *Let $X, Y \subseteq \mathsf{E}$ be two disjoint sets of edges. We have that $\mathrm{mst}\big(\mathrm{mst}(X) \cup Y\big) = \mathrm{mst}(X \cup Y)$.*

Proof: Consider an $e \in \mathrm{mst}(X \cup Y)$. There exists a cut $C = (S, (X \cup Y) \setminus S)$, such that e is the cheapest edge in this cut (since edge costs are all different). If $e \in X$, then $e \in (S \cap \mathsf{V}(X), \mathsf{V}(X) \setminus S) \subseteq C$, and e is the cheapest edge in this cut. Namely, e is an edge of $\mathrm{mst}(X)$. But then, it also an edge of $\mathrm{mst}(\mathrm{mst}(X) \cup Y)$. A similar argument applies if $e \in Y$. We conclude that $\mathrm{mst}(X \cup Y) \subseteq \mathrm{mst}(\mathrm{mst}(X) \cup Y)$. However, since $\mathrm{mst}(X) \cup Y \subseteq X \cup Y$, equality follows. ∎

We have that $T_1 = \mathrm{mst}(E_1)$, and assume inductively that $T_1 = \mathrm{mst}(E_1 \cup \cdots \cup E_i)$. We then have by the above lemma that

$$T_{i+1} = \mathrm{mst}(E_{i+1} \cup T_i) = \mathrm{mst}(E_{i+1}\mathrm{mst}(E_1 \cup \cdots \cup E_i)) = \mathrm{mst}(E_1 \cup \cdots \cup E_i \cup E_{i+1}).$$

But this implies that $T_k = \mathrm{mst}\big(\cup_{i=1}^{k} E_i\big) = \mathrm{mst}(\mathsf{E})$.

29.C. (20 PTS.) Assume you are given a union-find data-structure that can perform an operation in $O(1)$ time. Given a graph G with n vertices and m edges, where the weights on the edges are positive integers that are all $O(n^5)$, describe an algorithm, as fast possible, for computing the MST of G. What is the running time of your algorithm?

Solution:

Radix sort the edges by weights, and now run Kruskal algorithm. Since a union-find operation takes $O(1)$ by assumption, the overall running time of this algorithm is $O(n+m)$.

30 (100 PTS.) Not on MST.

You are given a directed graph $\mathsf{G} = (\mathsf{V}, \mathsf{E})$ with $2n$ vertices, and m edges. Here, the vertices appear in pairs (i.e., twins) x_i, y_i, for $i = 1, \ldots, n$. For a vertex u in this graph, let $\text{twin}(u)$ be the other vertex in the pair of u (thus, if $u = x_i$, then $\text{twin}(u) = y_i$).

There are no edges between twins in this graph. If an edge $u \to v$ appears in the graph, so does the edge $\text{twin}(v) \to \text{twin}(u)$. A subset of vertices $S \subseteq \mathsf{V}(\mathsf{G})$ is ***closed*** if $x \in S$, then all the vertices reachable from x in G are in S. A closed set is ***bad*** if it includes two vertices that are twins. A set of vertices $S \subseteq \mathsf{V}(\mathsf{G})$ is ***perfect*** if it is closed, $|S| = n$, and it includes exactly one vertex from each pair (i.e., it is not bad).

Describe a greedy algorithm, as fast as possible, that decides if there is a perfect set in the graph, and if so computes it. What is the running time of your algorithm? Prove the correctness of your algorithm.

(Hint: Compute the strong connected components of G, and analyze the meta graph of strong connected components of this graph. Add vertices to the output set using the meta graph, in a greedy fashion.)

Solution:

We need a property of this graph before we can present the algorithm.

Lemma 10.4. *Let C be a strong connected component of G. The graph G has another strong connected component C', such that $C' = \text{twin}(C) = \{\text{twin}(u) \mid u \in \mathsf{V}(C)\}$.*

Furthermore, if C is a sink of G^{SCC} then C' is a source in G^{SCC}.

Proof: Indeed, for any $u, v \in \mathsf{V}(C)$ there is a path from u to v in G $u = x_1 \to x_2 \to \cdots \to x_\beta = v$. But then, by the twin property of the edges of G, we have that $\text{twin}(v) = \text{twin}(x_\beta) \to \text{twin}(x_{\beta-1}) \to \cdots \to \text{twin}(x_1) = \text{twin}(u)$. Since there is also a path from v to u, the same argumentation implies a path from $\text{twin}(v)$ to $\text{twin}(u)$ in G. Namely, $\text{twin}(v)$ and $\text{twin}(u)$ are in the same connected component. This implies that $\text{twin}(C)$ is contained in some single strong connected component, say C'. Using the same argumentation, we get that $C = \text{twin}(\text{twin}(C)) \subseteq \text{twin}(C') \subseteq C$. Namely, C' and C are of the same size, and the claim follows.

As for the second claim, assume it is false, and there is an edge $u \to v \in \mathsf{E}$, with $v \in C'$, and $u \notin C'$. But then, by the twin edge property $\text{twin}(v) \to \text{twin}(u) \in \mathsf{E}$, and $\text{twin}(v) \in C$, and $\text{twin}(u) \notin C$. But that implies that C is not a sink, which is a contradiction. ■

Algorithm. Compute the strong connected components of G, and let $C_1, C_2, \ldots, C_k$ be these components. If there is a pair of vertices that both of them are in the same strong connected component, then the graph is bad, and there is no perfect solution. The algorithm outputs this and exit.

Otherwise, the algorithm computes the meta graph G^{SCC}. It greedily take a sink S_i of G^{SCC}, add its vertices to the output set S. Importantly, $\text{twin}(S_i)$ is also a strong connected component, and clearly, $\text{twin}(S_i)$ is a source of G. As such, the algorithm removes both S_i and $\text{twin}(S_i)$ from G. The algorithm repeats this peeling process till no vertices remain. It outputs the compuetd set as the desired perfect set.

Correctness. Clearly, the algorithm add a vertex s to the solution set, if and only if it throws $\text{twin}(s)$ away. It thus readily follows that what it outputs is indeed a perfect.

Running time. As for the algorithm, all it does is essentially scanning the topological ordering of G^{SCC} in reverse order, adding components to the set, and ignoring ones that their complements were already inserted. As such, We can easily implement this algorithm in $O(n+m)$ time.

What is this question, and why???

As some of you surely realized, this question is just asking you to develop the algorithm to solve 2SAT. You are given a 2SAT formula over n variables $x_1, \ldots, x_n$. Every clause is of the form $x \vee y$. We create a graph with $2n$ nodes, labeled $x_i = 0$ and $x_i = 1$, for $i = 1, \ldots, n$. Consider a clause $x_i \vee x_j$. If we pick $x_i = 0$ to the assignment, then we must pick $x_j = 1$ to the assignment. As such, we add an edge $(x_i = 0) \to (x_j = 1)$ (which is the logical implication, if you decided to assign zero to x_i, then you must assign 1 to x_j) to this graph. Similarly, we add the edge $(x_j = 0) \to (x_i = 1)$. We do this for all clause. Every clause give rise to two edges. As such, if the original formula has m clauses, the resulting graph has n variables and $2m$ clauses. The graph comply with the bizarre twin condition of the above graph.

Clearly, a perfect set in the resulting graph is a satisfying assignment. Using the above algorithm, one can compute a satisfying assignment in $O(n+m)$ time.

Awesome question, thanks for playing!