ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institute of Theoretical Computer Science
Bernd Gärtner, Rasmus Kyng, Angelika Steger, David Steurer, Emo Welzl

This is a draft version. After we have graded your assignments, we will publish an improved version based on the feedback that we have obtained from what you have written.

Solution 1

(a) If an edge e is *not* in $\mathrm{MSF}(E' \cup \{e\})$ it must be T-heavy: Otherwise, adding it to $\mathrm{MSF}(E') = T$ and removing a heavier edge from the created cycle would improve T, a contradiction. In our algorithm, we choose a random subset E' of exactly r edges. Let e be a uniformly random edge from E, chosen independently from the choice of E'.
Now note that e is uniformly random among $E' \cup e$, which has either size r or $r+1$. Therefore, the probability of e being in $\mathrm{MSF}(E' \cup \{e\})$ is at most n/r, as $|\mathrm{MSF}(E' \cup \{e\})| < n$ and $|E' \cup \{e\}| \geq r$. As a random edge is not T-heavy with probability at most n/r, the expected number of edges that are not T-heavy (the edges in L) is at most mn/r, concluding the proof.

(b) Let $p \in [0, 1]$ be an arbitrary constant probability. As in the proof of the original Randomized Minimum Spanning Tree algorithm in the script, we want to show that the runtime of Modified Randomized MST is bounded by $C(n + m)$ for some $C > 0$. We can assume that the random choice of the subset E' can be performed in $C_R(n+m)$ for some C_R. We only need to consider the two recursive calls, and everything else can be hidden in the constant C.
For the first recursive call, there are $n/8$ vertices, and $r \leq pm + 1$ edges. For the second recursive call, there are also $n/8$ vertices. Using subtask (a), we can conclude that there are at most $n/8 \cdot m/r \leq n/8 \cdot m/(pm) = n/(8p)$ edges.
If $(n/8+pm+1)+(n/8+n/(8p)) < (n+m)$, we can thus conclude that the algorithm indeed runs in time $C(n + m)$. For which values of p can we do this? For one, we need to have $pm < m$, which holds for any $p < 1$. Second, we need $n/8 + n/8 + n/(8p) < n$, which holds for any $8p > 4/3$, i.e., $p > 1/6$. We conclude that for any $1/6 < p < 1$, Modified Randomized MST runs in linear time.
Note that we ignored the additive $+1$. For any $1/6 < p < 1$, there exists constant n_0, m_0, such that the necessary inequality holds for all $n > n_0$, or $m > m_0$. The runtime of the algorithm for $n \leq n_0$ and $m \leq m_0$ can be considered a constant.

Solution 2

(a) We first note that for any $k - 1$ distinct vertices, the sum of their degrees must be at least the size of the minimum k-cut. Otherwise, removing their incident edges would be a smaller k-cut. Let C be some minimum k-cut of size $|C| = m$. We thus sum up over all subsets of $k-1$

vertices: $\sum_{v_1<\ldots<v_{k-1}} d(v_1)+\ldots+d(v_{k-1}) \geq \binom{n}{k-1}m$. By doing this, we include the degree of each vertex exactly $\binom{n-1}{k-2}$ many times. Thus,

$$\sum_v d(v) \geq \frac{\binom{n}{k-1}}{\binom{n-1}{k-2}}m = \frac{mn}{(k-1)}.$$

Therefore, $|E| \geq \frac{mn}{2(k-1)}$, and the chance of contracting an edge of C is at most $\frac{m}{\frac{mn}{2(k-1)}} = \frac{2(k-1)}{n}$ and the chance of C surviving (and thus $\mu(G,k)$ being unchanged) is at least $1-\frac{2(k-1)}{n}$.

(b) Let $f(k) = 2k-2$. The probability of BasicMin-k-Cut finding the correct minimum k-cut value is thus at least

$$\frac{n-2k+2}{n}\cdot\frac{n-2k+1}{n-1}\cdot\ldots\cdot\frac{1}{2k-1} = \frac{\prod_{i=1}^{2k-2} i}{\prod_{i=n-2k+3}^{n} i} \geq \frac{(2k-2)!}{n^{2k-2}}$$

Thus, if we run BasicMin-k-Cut $N = n^{2k-2}/((2k-2)!)$ times, the probability of failure is at most e^{-1} and thus the probability of success is at least $1/2$. Each run of the algorithm requires $O(n^2)$ time for the contractions, and $O(f(k)^2k^{f(k)}) =: h(k)$ for trying out all min-cuts among the final $f(k)$ vertices. Thus, in total, the algorithm requires $O(n^{2k-2}\cdot(n^2+h(k)))$ time, which is $O(n^{2k}+n^{2k-2}h(k))$ or more simply, $O(n^{2k}h(k))$.

(c) We can use the algorithm from (b) as our A_0, giving us $r(0) = 2k = 6 \leq 9$. We use the same approach as on page 12 of the lecture notes. The probability of the minimum cut being the same after RandomContract is at least

$$\frac{n-4}{n}\cdot\ldots\cdot\frac{t-3}{t+1} = \frac{t(t-1)(t-2)(t-3)}{n(n-1)(n-2)(n-3)}$$

As then A_i finds the correct mincut in this graph with probability at least $1/2$, we set $N = 2\frac{n(n-1)(n-2)(n-3)}{t(t-1)(t-2)(t-3)}$. The runtime of A_{i+1} is thus at most

$$O(N\cdot(n^2+t^{r(i)})) = O(\frac{n^4}{t^4}\cdot(n^2+t^{r(i)})$$

We set $t = n^{2/r(i)}$ to make both summands equal and thus get

$$O(n^{4-8/r(i)}n^2) = O(n^{6-8/r(i)})$$

as the runtime of A_{i+1}, yielding the desired recurrence.

Solution 3

(a) An edge from v to its parent is part of the shortest path between a and b if and only if one of a and b is part of the subtree rooted at v, and the other one is not. There are $w(v)\cdot(n-w(v))$ such pairs a,b. Each edge $v,p(v)$ is thus part of $w(v)\cdot(n-w(v))$ shortest paths. Summing up this quantity over all edges is equivalent to summing up the length of the shortest path for all pairs.

(b) $E[P_n] = \sum_{r=1}^{n} E[P_n|\text{root}=r]P[\text{root}=r] = \frac{1}{n}\sum_{r=1}^{n} E[P_n|\text{root}=r]$.
Given that the root is r, we can split our shortest paths as follows: Shortest paths between two vertices in the left subtree sum up to P_{r-1} in expectation. Similarly, paths between two vertices in the right subtree sum up to P_{n-r} in expectation.

The paths between all nodes in the left subtree and the root of the left subtree are part of $n-r+1$ shortest paths (with one endpoint outside of the subtree) each. The expected sum of these paths is x_{r-1}. Similarly on the right subtree, the sum of paths to the root of the subtree is x_{n-r}, and these are present in r shortest paths (with one endpoint outside of the subtree). Finally, we count how often the edges incident to the root are part of a shortest path. The left such edge is part of $(r-1)(n-r+1)$ shortest paths (by (a)), and the right edge is part of $(r(n-r))$ shortest paths. Summing this all up, we get the required formula.

(c) We compute np_n and $(n-1)p_{n-1}$.

$$np_n = 2\sum_{i=0}^{n-1} p_i + \sum_{i=0}^{n-1}(2n-2i)x_i + \sum_{i=1}^{n}(2(ni-i^2+i)-n-1)$$

$$(n-1)p_{n-1} = 2\sum_{i=0}^{n-2} p_i + \sum_{i=0}^{n-2}(2n-2-2i)x_i + \sum_{i=1}^{n-1}(2(ni-i^2)-n)$$

$$np_n - (n-1)p_{n-1} = 2p_{n-1} + 2\sum_{i=0}^{n-1} x_i + \sum_{i=1}^{n}(2i-1) - n$$

$$p_n = \frac{(n+1)p_{n-1} + 2\sum_{i=0}^{n-1} x_i + \sum_{i=1}^{n}(2i-1) - n}{n}$$

$$p_n = \frac{n+1}{n}p_{n-1} + \frac{2}{n}(\sum_{i=0}^{n-1} x_i) + (n-1)$$

Solution 4

(a) For each polygon, store the upper and lower envelope in sorted order. For the query, binary search through the layers. For each layer, determine whether the query point is inside/outside of the polygon by searching for the x-coordinate of the query point in the upper and lower envelope, and checking the relevant segment for being above/below the query point. Each binary search requires $O(\log n)$ queries, and the checks can be done in constant time. Query time is thus $O(\log^2 n)$. The space needed is constant per segment, and thus overall storage space is $O(n)$.
To preprocess, assuming that the layers of the onion are already given in sorted order, the upper and lower envelope can be extracted in linear time by walking along each polygon. If the onion layers are not given, we have to add the time needed to compute the onion, which is (as a very rough estimate) computing the convex hull of n points n times, $O(n^2 \log n)$.

(b) Sort all x-coordinates. For each x-interval between two points, sort the line segments crossing that interval from negative to positive y-coordinate (these cannot intersect). Between any two segments, store the onion depth. For a query point, binary search for the x-interval, then binary search for the correct area between two segments. Both binary searches are among at most $2n$ elements, and thus both take $O(\log n)$, leading to overall $O(\log n)$ query time. The storage needed is $O(n^2)$, as for each of the $O(n)$ x-intervals, we store $O(n)$ y-areas with constant amount of information each.
To preprocess, assuming that the layers of the onion are already given in sorted order, we can sort all points in $O(n \log n)$. Then, the segments have to be sorted in y-direction, again incurring at most $O(n \log n)$.

Alternative solution:
One might also want to use trapezoidal decomposition from the lecture. Each segment of the convex hulls is added as a segment to the data structure. With each segment is stored the index of the corresponding layer, and whether the segment is in the upper or lower envelope of that convex hull. By answering "segment above" queries, we can then figure out the onion depth: If the segment above is in the lower envelope of layer i, the query point is in layers $0, \ldots, i-1$, but not i. Similarly, if the segment above is in the upper envelope of layer i, the query point is in layers $0, \ldots, i$, but not $i+1$.
As shown in the lecture notes, the trapezoidal decomposition requires $O(n)$ space and $O(\log n)$ query time, but these bounds both only hold *in expectation*. Using this approach to solve both (a) and (b) *with certainty* requires some more arguments. As the preprocessing time is unlimited, it is possible, but requires arguments we will not go into in detail for now.