CSE 347 Analysis of Algorithms March 27, 2019

Lecture 18 Summary

Instructor: Brendan Juba

1. The basic Contraction Algorithm solves the global Min-Cut problem with probability $1/\binom{|V|}{2}$.
   (a) When we *contract* an edge (u, v) of our (undirected) (multi)graph G, we replace the vertices u and v with a new vertex w such that for each edge (u, x) or (v, x) incident to u or v (but not joining u and v), we add a (w, x) edge. (In particular, e.g., if both u and v had an edge to x, *both* of these add a (w, x) edge.)
   (b) The contraction algorithm is: $|V| - 2$ times, pick a random edge and contract it. Finally return the sets of vertices corresponding to the two remaining vertices.
   (c) The algorithm succeeds if and only if we never pick one of the edges of a minimum cut, since then the two supernodes must correspond to the sides of this cut.
   (d) If the minimum cut had size k, then the cut induced by each of these "supernodes" and the rest of the graph must have at least k edges crossing, i.e., each has degree at least k; therefore, the number of edges on the ith iteration is at least $(|V| - (i-1))k/2$.
   (e) Therefore, the probability of *not* picking an edge of the cut on the ith iteration, given that we didn't pick one previously, is at least $1 - \frac{2}{|V|-(i-1)}$. Thus, the probability that we never pick such an edge is at least $\prod_{i=1}^{|V|-2}(1 - \frac{2}{|V|-(i-1)}) = \frac{2}{|V|(|V|-1)}$
2. We can take an algorithm for an optimization problem that succeeds with (polynomially) small probability, like the contraction algorithm, and "amplify" its correctness to probability $1 - \delta$
   (a) If we find an optimal solution with probability ϵ, and we repeat ℓ times and take the best solution, then we only fail to find an optimal solution with probability $(1 - \epsilon)^\ell$.
   (b) Therefore, here, $\epsilon = 1/\binom{|V|}{2}$, and so if we pick $\ell = \frac{1}{\binom{|V|}{2}} \ln \frac{1}{\delta}$, (using $1 + x \leq e^x$ for all x), we find that the probability of failing to find a minimum cut is at most δ, i.e., the probability of succeeding is at least $1 - \delta$.
   (c) A similar trick can amplify correctness of a decision algorithm using the Chernoff bound.
3. Good data structures for maintaining the "supernode" sets are only good in an "amortized" sense – that is, we only bound the total time for a sequence of operations, not the worst-case time per operation.
   (a) As a simpler example of this "amortized analysis," consider the following array-based implementation of a stack: we initially allocate an array of size ℓ, and maintain a "top" index. On pop, if top is 0, we return NULL, and otherwise we decrement top and return the element in index top. On push, if top is less than $\ell + 1$, we put the new element in the top index and increment it. If top is equal to $\ell + 1$, we first allocate a new array of size 2ℓ, copy the old array over to the new one, and then execute push as before.

(b) We can show that this takes constant *amortized time* per operation as follows:

i. We introduce a *potential function* $\Phi_t \geq 0$ for the data structure after each tth operation. We take as convention that $\Phi_0 = 0$. We can think of Φ_t as the balance of a "bank account" – on cheap operations, we deposit a little extra into the account, which we withdraw to cover expensive operations.

ii. On each push that does not allocate a new array, we put an extra \$2 into the bank ($\Phi_t = \Phi_{t-1} + 2$). Thus, in total, we suppose both pop and these "cheap" push operations can be covered by a cost of \$3.

iii. Let's suppose now that the "expensive" push costs $\$\ell$ to copy the ℓ elements.

iv. Notice, when we last allocated the array, since we double the size each time, the array had at most $\ell/2$ elements in it. Therefore, we had at least $\ell/2$ cheap push operations since then, and so our bank account must contain at least $\$\ell$ to cover the cost of this expensive push, so we take $O(1)$ amortized time in total.

v. In general, if we spend T_t time on the tth operation and $T_t + (\Phi_t - \Phi_{t-1}) \leq C$, then our amortized time is at most C per operation.

(c) For the Contraction algorithm (and Kruskal's Minimum Spanning Tree algorithm) we needed a "Union Find" data structure that supports the following operations

i. MakeSet(v): returns the "name" of a set containing simply the element v

ii. Find($\{v\}$): returns the "name" of the set *currently* containing the set that originally contained v. Note that we can use this to check if two elements are contained in the same set by testing if Find($\{u\}$)=Find($\{v\}$).

iii. Union(S_1, S_2): replaces the sets S_1 and S_2 with a new set $S_1 \cup S_2$ and returns the name of this new set (i.e., contracting two supernodes or joining two trees from a spanning forest into a single tree in Kruskal's algorithm).

(d) There is a relatively simple implementation of Union Find that uses $O(1)$ time for both MakeSet and Find, and takes $O(\log n)$ amortized time over n Union operations. This is adequate to make Kruskal's algorithm competitive with Prim's algorithm (for example).

i. We represent sets by linked lists, with an extra field that points to the head of the list and an extra field that, in the head of the list, contains the size of the list.

ii. The head of the list will be the "name" of the set. So, MakeSet and Find are easily seen to be $O(1)$ time with this setup.

iii. The Union operation is performed as follows. We take the head of the longer list as the head of the union set; we update its size to the sum of the sizes of the two lists, and walk the *shorter* list, updating each element to point to this new head. At the end, we attach the old second node of the larger set to the end of the list. The head's next field is updated to point to the old head of the smaller set. Next time we'll see the amortized analysis of this operation.