CS3230: Design and Analysis of Algorithms
Semester 2, 2019-20, School of Computing, NUS

Solutions of Assignment 2

Question 1: (5+10+5) Consider a connected graph $G = (V, E)$ with n nodes (i.e., $|V| = n$) and m edges (i.e., $|E| = m$). For any (non-trivial) partitioning of $V = V_1 \cup V_2$ (with $V_1 \cap V_2 = \emptyset$ and both V_1 and V_2 are non-empty) we call the set of all the edges having one endpoint in V_1 and other in V_2 a *cutset*. Now we would like to find a partition such that the corresponding cutset is of minimum size.

To find such a partition we would adopt the following randomized procedure (Procedure $\mathcal{A}$):

1. Pick an edge of G uniformly at random, and contract its two endpoints into a single node. Repeat this process until only two nodes remain.

2. Let u_1, u_2 be the two remaining nodes at the end. Let V_1 be the set of nodes that went into u_1 (while contracting edges), and similarly V_2 be the set of nodes that went into u_2. Output V_1 and V_2 as the partition.

Just to clarify, suppose the algorithm choose an edge $u - v$ to contract, then you may think of the new node as a set uv and all the edges (except $u - v$) incident on u and v in original will now incident on this new node. Note, contraction of edges may create multiple edges between two nodes. See Figure 1. Now answer the following questions.

(a) Show that the size of a minimum cutset is at most $\frac{2m}{n}$.

Answer: Without loss of generality assume that the graph is undirected. Let $d(v)$ denote the degree (in case of directed graph, sum of in-degree and out-degree) of a vertex v. Observe that $\sum_{v \in V} d(v) = 2m$ (because on the left size we count each edge twice). So by averaging argument there exists one vertex v such that $d(v) \leq 2m/n$.

Now if you remove all the edges incident on a vertex then you will get a non-trivial partitioning with cutset of size equal to the degree of that vertex. Hence size of minimum cutset is at most $2m/n$.

(b) Probability that the cutset corresponding to the output (partition) of Procedure $\mathcal{A}$ is of minimum size, is at least $\frac{2}{n(n-1)}$.

Answer: Let V_1 and V_2 be the partition corresponding to a minimum cut, and size of the minimum cut is s. Then by the statement of the previous question, probability that at the first step the algorithm will pick some edge from this particular cut is $s/m \leq 2/n$. The stated algorithm returns the right answer as long as it never picks an edge across the minimum cut. If it always picks a non-mincut edge, then this edge will connect two nodes on the same side of the cut, and so it is okay to collapse them together.

Each time an edge is collapsed, the number of nodes decreases by 1. Therefore,

$$\begin{aligned}
Pr[\text{Final cut is a minimum cut}] &= Pr[\text{The first selected edge is not in minimum cut}] \times \\
&\quad Pr[\text{The second selected edge is not in minimum cut}] \times \cdots \\
&\geq (1 - \frac{2}{n})(1 - \frac{2}{n-1})(1 - \frac{2}{n-2}) \cdots (1 - \frac{2}{3}) \\
&= \frac{n-2}{n} \cdot \frac{n-3}{n-1} \cdot \frac{n-4}{n-2} \cdots \frac{1}{3} \\
&= \frac{2}{n(n-1)}.
\end{aligned}$$

Note, the algorithm that you have just seen is the famous Karger's algorithm for finding global mincut.

(c) Repeat Procedure $\mathcal{A}$ for k times, and finally return the partition with minimum cutset size among k output partitions. Now this is your final algorithm. What should be the value

k you will choose such that your final algorithm outputs the partition with minimum cutset, with probability at least $2/3$? (Provide the detailed analysis.)

Answer: From the previous question we know that during a particular run of the algorithm probability that the output is not a minimum cut is at most $(1 - \frac{2}{n(n-1)})$. Now if you independently run the algorithm k times, then probability that at least one of the runs outputs a minimum cut is at least $1 - (1 - \frac{2}{n(n-1)})^k \geq 1 - e^{\frac{-2k}{n(n-1)}}$, which is at least $2/3$ for $k = \Theta(n^2)$. So the final algorithm runs in time $O(n^2 m)$ and outputs a min-cut with probability at least $2/3$.

Note, the algorithm that you have just seen is the famous Karger's algorithm for finding global mincut. There are some modifications of the given algorithm that lead to even faster algorithm which is referred as Karger-Stein Algorithm. Interested student may explore online on further improvements.

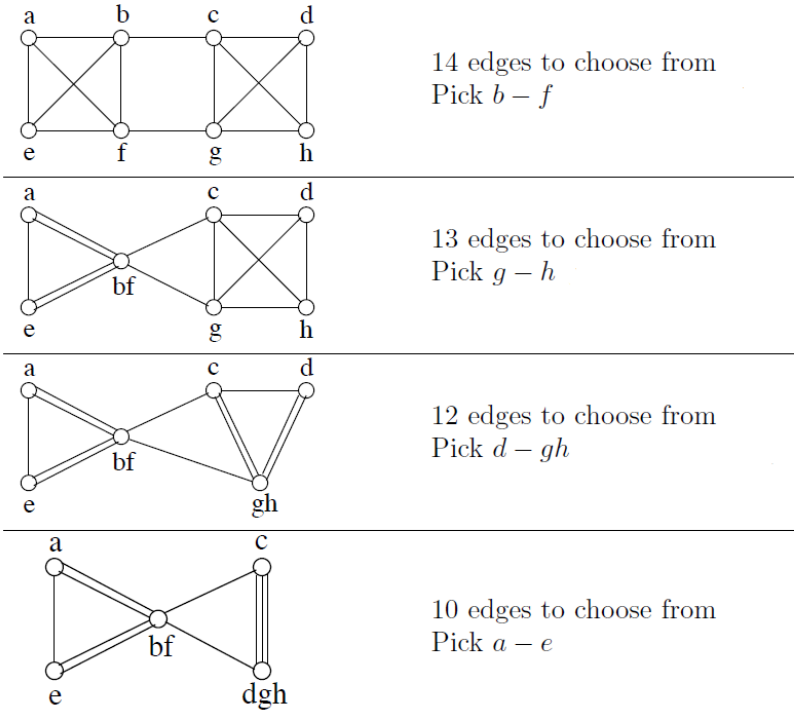

Figure 1: An example run of the algorithm

Question 2: (5+5) You are asked to maintain a list L while supporting the following two operations:

- Insert(x, L): Insert an integer x in the list L (cost is 1).
- ReplaceSum(L): Compute the sum of all the integers in L. Then remove all these integers, and finally just store the computed sum in L (cost is length of the list L).

(a) Use the accounting method to show that the amortized cost of both Insert and ReplaceSum operation is $O(1)$.

Answer: Charge \$2 for each insertion, among that \$1 will be used to insert and remaining \$1 will be stored in the bank. Now use the saving on the bank to delete each element.

At any point of time the bank balance is equal to the number of elements in the list, which cannot be negative. Clearly the amortized cost of Insert is $O(1)$.

During a ReplaceSum(L) operation let ℓ be the number of elements present in the list. Hence the bank balance is ℓ which is sufficient to delete all these ℓ elements. So the amortized cost of ReplaceSum(L) is again $O(1)$.

(b) Use the potential method to show that the amortized cost of both Insert and ReplaceSum operation is $O(1)$.

Answer: Consider the number of elements in the list as your potential function ϕ. At the beginning when the list is empty $\phi = 0$, and by definition at any point ϕ is number of elements in the list which is non-negative.

After each insertion operation number of elements increases by 1, and so potential difference is also 1. Actual cost of Insert is $O(1)$. Hence the amortized cost (=actual cost + potential difference) is also $O(1)$.

After each ReplaceSum(L) number of elements decreases by $\ell - 1$ (let ℓ be the number of elements in the list before the operation). So the potential difference is $1 - \ell$. Since actual cost is ℓ the amortized cost is again $O(1)$.

(Note, in the above analysis if somebody considers the actual cost of ReplaceSum(L) to be 2ℓ (which is more accurate) then he has to take potential function to be $2\times$ the number of elements in the list. The whole analysis remains the same.)

Question 3: (1+3+3+10+3) Let us consider the following *Online List Problem*. In this problem we are given a doubly linked list L containing n items (you may consider them to be distinct), and a sequence of m search operations $s_1, s_2, \cdots, s_m$ with the following restrictions:

- Each search operation s_t will look like: Given an item e_t you have to correctly say whether e_t is in the list L or not.
- The sequence of search operations will arrive in online fashion, i.e., at time t you will get a request for the operation s_t and you have to answer correctly. (One important thing to note, you cannot see future search queries.)
- You can access the linked list only through the HEAD pointer. So if at time[1] t you are asked to search an item e_t which is the third element in the list at that moment, then you have to pay 3 unit cost to find that element.
- During answering a search query you may update the list. However at any point you can only swap two neighboring elements in the list, and each such swap will cost you one unit. (So during any search operation if you will perform k swaps, you will have to pay additional k unit cost.)

Now keeping all these restrictions in mind, ideally we would like to design an algorithm that achieves minimum total cost.

Ooh, that must be a very difficult task! So to make your life simpler, I am not asking you to design an algorithm with the minimum cost. Instead I ask you to consider the following simple heuristic algorithm: Whenever we search an item in the list, if we find it we bring that in front of the list (by performing a sequence of swaps).

You may wonder since this is a extremely simple heuristic strategy, its total cost will no way be close to the minimum. However I have a different opinion. Suppose if all the m search operations were given in advance (instead of arriving in online fashion) then the minimum cost that one (optimum algorithm) could achieve is C_{opt}. I claim that the total cost of the above heuristic algorithm is just at most $4C_{opt}$. If you answer the following series of questions, you will also be able to prove this claim.

(a) During the search operation s_t suppose you are asked to search for the item e_t. Let r and r^* be the rank of e_t in the list maintained by the heuristic algorithm (denoted by L) and the optimum algorithm (denoted by L^*) respectively. Further suppose the optimum algorithm performs t^* swaps during the search operation s_t. What is the difference between the cost to perform s_t by the heuristic and the optimum algorithm?

Answer: The cost to search the item by the optimal algorithm is r^*, and so the overall actual cost is $r^* + t^*$. Similarly the overall actual cost for the heuristic algorithm is $r + (r-1) = 2r - 1$, where r is for the search and $r - 1$ for those many swaps. So the difference is $2r - r^* - t^* - 1$.

(b) Let us define the *distance* between two lists L and L' containing exactly the same items (but in different order) as number of pairs (x, y) such that relative ordering of x and y in L and L' are different. For example, if $L = 10, 20, 30, 40$ and $L' = 10, 30, 40, 20$, then the distance between them is 2 (since $(20, 30), (20, 40)$ are the pairs with different relative ordering in L and L'). Show that $\phi(t)$, defined as the twice the distance between L and L^* at time t, is a valid potential function.

[1]Do not get confused between time and cost. Here I use the term *time* to identify which operation you are going to serve at that particular moment, whereas the cost captures the complexity of each operation.

Answer: At the beginning both the lists (one maintained by the optimal algorithm and another maintained by the heuristic one) are same, and so $\phi = 0$. By the definition of the distance it can never be negative and hence at any point of time $\phi \geq 0$. So ϕ is a valid potential function.

(c) Show that during performing s_t by the heuristic algorithm, each swap changes the distance between L and L^* by at most one.

Answer: Suppose we are swapping items e_i and e_j where e_i appears before e_j. We know that such swap can happen only between two consecutive elements in the list. So relative ordering of only these two elements will change. Either the distance will increase by 1 (if e_i appears before e_j in the optimum list) or decrease by 1 (if e_i appears after e_j in the optimum list).

(d) Next prove that the change in potential from t-th time to $(t+1)$-th time is at most $4r^* + 2t^* - 2r - 2$.

Answer: Note, during each operation s_t we will move exactly one element e_t in the heuristic algorithm to the front. Since rank of e_t in the list of optimum algorithm is r^*, its movement can increase the distance by at most $r^* - 1$. Since r is the rank of e_t in the list maintained by the heuristic algorithm, this movement also leads to decrease in distance due to change in relative ordering with at least $r - r^*$ many elements (which appears before e_t in heuristic algorithm list but after the optimum algorithm list). t^* swaps by the optimum algorithm can increase the distance by at most t^*.

So the overall change in potential is at most $2(r^* - 1 + t^* - (r - r^*)) = 4r^* + 2t^* - 2r - 2$.

(e) Now complete the amortize analysis using the above potential function to show that the total cost of the above mentioned heuristic algorithm is at most $4C_{opt}$.

Answer: The amortized cost of operation s_t is actual cost + potential difference, which is at most $(2r - 1) + (4r^* + 2t^* - 2r - 2) \leq 4(r^* + t^*) = 4 \times$ Actual cost of optimal algorithm for performing s_t.

Hence the total cost of all the operations by the heuristic algorithm is at most $4C_{opt}$.

Note, what you have just seen is called the *competitive analysis* in the domain of online algorithm. In the above question you have finally shown that your online algorithm is 4-competitive.