CSC265 F18: Assignment 6

Due: December 6, by midnight

Guidelines: (read fully!!)

- Your assignment solution must be submitted as a *typed* PDF document. Scanned handwritten solutions, solutions in any other format, or unreadable solutions will **not** be accepted or marked. You are encouraged to learn the LaTeX typesetting system and use it to type your solution. See the course website for LaTeX resources. Solutions typed using LaTeX receive 2 bonus marks.
- Your submission should be no more than 6 pages long, in a single-column US Letter or A4 page format, using at least 9 pt font and 1 inch margins.
- To submit this assignment, use the MarkUs system, at URL `https://markus.teach.cs.toronto.edu/csc265-2018-09`
- This is a *group assignment.* This means that you can work on this assignment with *at most one other* student. You are *strongly encouraged* to work with a partner. Both partners in the group should work on and arrive at the solution together. Both partners receive the same mark on this assignment.
- Work on all problems together. For each problem, one of you should write the solution, and one should proof-read and revise it. The first page of your submission must list the *name*, *student ID*, and *UTOR email address* of both group members. It should also list, for each problem, which group member wrote the problem, and which group member proof-read and revised it.
- You **may not** consult any other resources except: your partner; your class notes; your textbook and assigned readings. *Consulting any other resource, or collaborating with students other than your group partner, is a violation of the academic integrity policy!*
- You may use any data structure, algorithm, or theorem previously studied in class, or in one of the prerequisites of this course, by just referring to it, and without describing it. This includes any data structure, algorithm, or theorem we covered in lecture, in a tutorial, or in any of the assigned readings. Be sure to give a *precise reference* for the data structure/algorithm/result you are using.
- Unless stated otherwise, you should justify all your answers using rigorous arguments. Your solution will be marked based both on its completeness and correctness, and also on the clarity and precision of your explanation.

Question 1. (14 marks)

Consider an ADT which maintains a collection of disjoint subsets of the set $[n] = \{1, \dots, n\}$, and supports the operations

- $\textsc{Split}(i)$ splits the set S containing i into two sets: $S_{\leq} = \{j \in S : j \leq i\}$ and $S_{>} = \{j \in S : j > i\}$;
- $\textsc{Find-Min}(i)$ returns the minimum element of the set S containing i.

Suppose that you are given as input a sequence $\sigma = (\sigma_1, \dots, \sigma_m)$ of operations, where each σ_t is equal to either $\textsc{Find-Min}(i)$ or $\textsc{Split}(i)$ for some $i \in [n]$. Assume that before σ_1, we start with a single set $S = [n]$. Assume further that, for each $i \in [n-1]$, $\textsc{Split}(i)$ is called at most once in σ, and that it's never called for $i = n$. Design an algorithm that, given σ as input, computes the value that must be returned by the ADT on each call of $\textsc{Find-Min}(i)$. More precisely, your algorithm must return an array $R[1\,..\,m]$, so that if $\sigma_t = \textsc{Find-Min}(i)$, then $R[t]$ is the value returned by this instance of $\textsc{Find-Min}(i)$, and if $\sigma_t = \textsc{Split}(i)$, then $R[t] = \textsc{Nil}$. Your algorithm must run in worst-case time complexity $O((n + m) \log^* n)$.

As an example, let $n = 5$, and assume that σ is the sequence

$$\textsc{Find-Min}(3), \textsc{Split}(2), \textsc{Find-Min}(3), \textsc{Find-Min}(1), \textsc{Split}(3), \textsc{Find-Min}(4).$$

Then you must return $R = [1, \textsc{Nil}, 3, 1, \textsc{Nil}, 4]$. After the first split, the sets maintained by the ADT are $\{1, 2\}, \{3, 4, 5\}$, and after the second split, they are $\{1, 2\}, \{3\}, \{4, 5\}$.

Describe your algorithm in clear and precise English, and justify why it runs in the required worst-case time complexity and why it correctly computes the array R. Be precise about what data structures you use.

Remark: *Your do **not** need to implement a data structure for the above ADT. You can use the fact that the entire sequence σ is given to your algorithm.*

[Solution]

We first initialize an array $P[1\,..\,n]$, all of whose entries are initially 0. We scan σ and for each i such that for some t we have $\sigma_t = \textsc{Split}(i)$, we record $P[i] = 1$. Let i_1 be the smallest integer such that $P[i_1] = 1$; let i_2 be the second smallest such integer, etc, and let k be the total number of such integers, i.e. the number of calls to $\textsc{Split}$. We claim that after σ_m is executed, the collection of sets maintained by the ADT is $S_1 = \{1, \dots, i_1\}$, $S_2 = \{i_1 + 1, \dots, i_2\}$, …, $S_{k+1} = \{i_k + 1, \dots, n\}$. This follows by an easy induction on the number of splits.

We initialize a disjoint set forest data structure with the sets $S_1, \dots, S_{k+1}$. Moreover, for each set, we will maintain a field *min* storing the minimum element of the set: you can assume that this field is stored with the root of the tree representing the set. Initially, we can just create a tree for every set, with one of its elements (say the minimum) chosen as the root, and all other elements as children. Initialize the rank field of every node in the tree to 0, except the root, whose rank is initialized to 1. Initialize the *min* field to the minimum. Using the array P, this takes time $O(n)$.

For the rest of the algorithm, we will assume that the disjoint set forest is implemented using the union by rank and the path compression heuristics. Moreover, whenever we take the union of two sets S and S', we update the *min* of the new set to be the minimimum of the *min* fields of the two sets. This does not affect the asymptotic complexity of the union.

We then start processing the operations in σ in reverse order, starting from σ_m. When processing operation σ_t we do the following. If the operation is $\textsc{Find-Min}(i)$, then we call $\textsc{Find}(i)$ in the disjoint set forest and record the *min* field of the set containing i in $R[t]$. If σ_t is $\textsc{Split}(i)$, then we call $\textsc{Union}(i, i + 1)$ in the disjoint set forest. This finishes the description of the algorithm.

The correctness of the algorithm follows from the claim that after processing σ_t, the sets maintained by the disjoint set forest are the same as the sets maintained by the ADT when the operations $\sigma_1, \dots, \sigma_{t-1}$ have been executed. Observe first that, by a straightforward induction, at any point in time the sets maintained by the ADT are all contiguous, i.e. every set has the form $\{a, a + 1, \dots, b\}$. We will prove our claim by induction on the number of operations processed by our algorithm. The base case is the claim above that

$S_1, \ldots, S_{k+1}$ are maintained by the ADT after executing σ_m. The inductive step is trivial if σ_t is Find-Min. Otherwise, we have that the set $S = \{i-a, i-a+1, \ldots, i, i+1, \ldots, i+b\}$ is split by $\sigma_t = \text{Split}(i)$ into $\{i-a, i-a+1, \ldots, i\}$ and $\{i+1, \ldots, i+b\}$. Therefore, the collection of sets maintained by the ADT before σ_t is executed is the same as the sets maintained after it is executed, but with the sets containing i and $i+1$ merged. Since this is exactly what we do when processing σ_t, the inductive step follows.

For the running time analysis, we first recall from above that creating the initial disjoint set forest takes time $O(n)$. Then the running time of the rest of the algorithm is dominated by m calls to disjoint set forest operations, one for each σ_t. As shown in lecture, these m operations take time $O(m \log^* n)$ in the worst case, giving total running time of $O(n + m \log^*(n))$.

Question 2. (14 marks)

Let $G = (V, E)$ be an undirected connected graph on the nodes $V = \{1, \ldots, n\}$, with edges $E = \{e_1, \ldots, e_m\}$, and for any $i \in \{1, \ldots, m\}$ define the graph $G_i = (V, E_i)$, $E_i = \{e_1, \ldots, e_i\}$. Moreover, G_0 is the totally disconnected graph $G_0 = (V, \emptyset)$. There exists a unique $i^* \in \{1, \ldots, m\}$ such that G_{i^*} is connected, and G_{i^*-1} has two connected components.

Assume the edges $e_1, \ldots, e_m$ are given as an array $A[1\,..\,m]$, where $A[i]$ contains the tuple of vertices $e_i = (u, v)$, $u, v \in \{1, \ldots n\}$. Give an algorithm that, when given as input the integer n, and the array A, outputs the i^* defined above. The algorithm should run in worst-case time complexity $O(m)$.

Describe your algorithm in clear and precise English, and justify why it runs in the required worst-case time complexity and why it correctly computes i^*.

Hint: Do a binary search for i^* and use some graph search.

[Solution]

As in the hint, we use binary search. Initially, let us set $\ell = \lceil m/2 \rceil$. We will keep an array $C[1\,..\,n]$ to indicate the connected component of each node i. We initialize all entries of this array to 0. First, we construct the adjacency list Adj of G_ℓ from $A[1\,..\,\ell]$. To do this, we create an array $Adj[1\,..\,n]$ and initialize its entries to the Nil pointer. Then we go through $A[1\,..\,\ell]$ and for each edge $e_i = (u, v)$, stored in $A[i]$, we add v to the front of the linked list stored in $Adj[u]$, and we also add u to the front of the linked list stored in $Adj[v]$. This takes time $O(n + \ell) = O(m)$, as $\ell \geq m/2 \geq (n-1)/2$ (since G is connected).

Then we use Adj to run BFS on G_ℓ, starting from node 1. When the BFS is completed, we go through all vertices and for each vertex v whose distance from 1 (as computed by BFS) is finite, we record 1 in $C[v]$. Since the distance of a node from v is finite if and only if they are in the same connected component, these are all the vertices in 1's connected component.

If any vertex u had infinite distance from 1, then we restart BFS from u, and after the BFS is completed, we record $C[v] = 2$ for any v which has finite distance from u. We continue these BFS runs, each started from a node which so far has had infinite distance to the previous start nodes, until we exhaust all nodes, and all entries of C are nonzero. This way we have labeled each node with its connected component, and for any two nodes u and v, $C[u] = C[v]$ if and only if they are in the same connected component in G_ℓ. The maximum entry in C indicates the number of connected components in G_ℓ.

If G_ℓ has one connected component (i.e. is connected), we must have that $i^* \leq \ell$. If $\ell \leq n-1$, then $G_{\ell-1}$ is disconnected and we return $i^* = \ell$. Otherwise, we run BFS on $G_{\ell-1}$ to check if it is connected, and, if it is not, we output ℓ. If$G_{\ell-1}$ is connected, we recurse on V and $E = \{e_1, \ldots, e_{\ell-1}\}$.

If G_ℓ has $c > 1$ connected components, we must have $i^* > \ell$. Then we create a new graph to recurse on. Define the vertex set $V' = \{1, \ldots, c\}$, and the edge set $E' = \{e'_1, \ldots, e'_{m-\ell}\}$ using the following rule: if edge e_i, $\ell < i \leq m$, goes between a vertex in connected component a and a vertex in connected component b of G_ℓ, then set $e'_{i-\ell} = (a, b)$. This can create self-loops and parallel edges, but this is not a problem for our algorithm (or for BFS). The new sequence of edges can be created from $e_{\ell+1}, \ldots, e_m$ in time $O(m)$ using the array C. Then we recurse on V' and $e'_1, \ldots, e'_{m-\ell}$. If the recursive call returns j^*, then we return $i^* = j^* + \ell$.

The correctness of the algorithm is implied by the claim that the graph $G'_k = (V', \{e'_1, \ldots, e'_k\})$ is connected if and only if $G_{\ell+k}$ is connected. On one hand, we can transform any path P in $G_{\ell+k}$, into a path in G'_k as follows. If the i-th edge (u_i, v_i) in P connects a node u_i which lies in the connected component numbered a during our BFS of G_ℓ, and a node v_i which lies in component numbered b, then we can replace it with an edge (a, b) of G'_k if $a \neq b$, or drop it if $a = b$. In the case $a \neq b$, the edge (a, b) is an edge of G'_k by the definition of the graph, and because (u_i, v_i) cannot be an edge of G_ℓ since its two end points are in different connected components in G_ℓ. This transformation implies that if $G_{\ell+k}$ is connected, then there must be a path between any two connected components a and b of G_ℓ, so there must be a path in G'_k between any two nodes a and b, and G'_k is connected too.

Conversely, we can take any path P' of G'_k and transform it into a path P in $G_{\ell+k}$. To do this, take any edge (a_i, b_i) in P' and add to P any edge between two nodes u_i and v_i, where u_i is in the connected component numbered a_i in G_ℓ, and v_i is in the connected component b_i in G_ℓ. Such an edge must exist in $G_{\ell+k}$, or (a_i, b_i) would not be an edge in G'_k. This does not quite create a path, as for two consecutive edges (u_i, v_i) and (u_{i+1}, v_{i+1}) it may not be the case that $v_i = u_{i+1}$. However, v_i and u_{i+1} must be in the same connected component in G_ℓ (or P' would not be a path in G'), so there is a path between them in G_ℓ, and, therefore, also in $G_{\ell+k}$. We simply pick one such path and add its edges to P. With this, we have completed the definition of P. This transformation shows that, if G'_k is connected, then for any two connected components a and b of G_ℓ, there is a path in $G_{\ell+k}$ between a node in a and a node in b. Since all nodes in the same component of G_ℓ are connected in $G_{\ell+k}$, this clearly implies that $G_{\ell+k}$ is also connected. We have now proved our claim that G'_k is connected if and only if $G_{\ell+k}$ is, and, therefore, our algorithm is correct.

For the running time analysis, note the algorithm takes time $O(m)$ before the recursive call, and in each case we recurse on at most $\lceil m/2 \rceil$ many edges. Also we never recurse on a graph with fewer than $n-1$ edges. So, we have a recursive algorithm whose worst-case running time satisfies the recurrence $T(m) \leq T(\lceil m/2 \rceil) + Cm$, where C is a constant independent of m. It is easy to verify that this recurrence solves to $T(m) = O(m)$.