Diego Escobedo March 31, 2020

6.046 Problem Set 5

Collaborators: *Temi Omitoogun, Daniel Adebi, Timmy Xiao*

Problem 1

(A)

`Algorithm:` We are going to store the graph in a union-find data structure. However, we are going to start with only the vertices without considering any edges. We initialize the union-find by performing MAKE-SET$(v) \forall v \in V$, the set of all vertices. Now, we will sequentially consider the edges, i.e. we will be considering the last element of E_k for $k = 1, 2, 3.....$ When we consider an edge, say edge $e_k = (i, j)$, we will first check whether FIND-SET(i) equals FIND-SET(j). If it does, then we exit and claim that we found the smallest k such that a cycle exists in G_k. If it doesn't, then we perform UNION(i, j). If we eventually end the stream of edges and we have not found a cycle, we claim that there is no cycle specified by those specific edges.

`Correctness:` We are maintaining the invariant that there are NO cycles in the graph. Initially, it is fairly obvious why there are no cycles: there are no edges. However, when we add a single edge, then there are two cases. Firstly, if we add an edge between two elements that have the same representative element, then I claim that this will give us the cycle. Because we have been maintaining the invariant and unioning anything that didn't violate this invariant (aka storing the 'straight line'/'tree' that they form as a connected component in the union find), now we find ourselves with a 'straight line'/'tree' and an edge that makes this straight line into a loop. Secondly, if they have different representative elements, then the whole 'network' that i and j are connected to are totally distinct (and by our invariant, do not contain cycles). However, a cycle requires that we return to the element we started the cycle with, and if our invariant is true then there is no way to draw a single edge between sets that will create a cycle between two totally disjoint sets of elements. Therefore, our algorithm is correct.

`Runtime/Space:` We know from lecture that because of our improvements, we only ever have $O(|V|)$ edges. Combined with the fact that we are storing $|V|$ vertices, we have that we use $O(2|V|) = O(|V|)$ space. In terms of runtime, we are performing a constant number of amortized constant time operations at each step of k, so the algorithms runtime depends on k. Since we know that once $|E| > |V| - 1$ we are guaranteed to have a cycle, we can safely say that the worst-case running time is $O(|V|)$.

(B)

IMPORTANT FACT: Before we begin, let us recall the fact from 6.042 that bipartite graphs cannot have a cycle with an odd number of vertices. This stems from the fact that if it had an odd number of vertices, the cycle would have to be 'closed off' by connecting two vertices in the same 'side' of the graph (since the cycle has to alternate between sides). This further raises the concept of 'coloring' a graph, as we did for red-black trees in 6.006. In theory, we should be able to color a bipartite graph with 2 colors, meaning every time a new edge arises, the two vertices it connects should have a different color. This being said, let us approach the problem:

`Algorithm:` We are going to store the graph in a union-find data structure. However, we are going to augment the nodes with two properties: node.*color* (initialized at red for all nodes), and node.$children\_flipped$ (initialized at False for all nodes). Additionally, we are going to slightly change our behavior of FIND-SET(v). The only thing that we are going to change is that when traversing up the parents to find the representative elements, if a given ancestor node has node.$children\_flipped$ set as True, we will flip the color of the current node. This has the effect that if there is an even number of ancestor nodes, the color will not change, and if there is an odd number, it will change. This modified find-set will be performed recursively for all its ancestors too.

Then, we are going to start performing our algorithm from part A. We are going to start with only the vertices without considering any edges. We initialize the union-find by performing MAKE-SET(v)$\forall v \in V$, the set of all vertices. Now, we will sequentially consider the edges in an arbitrary order, i.e. we will be considering the edge e_k for $k = 1, 2, 3..., |E|$. When we consider an edge, say edge $e_k = (i, j)$, we will perform FIND-SET(i) an FIND-SET(j) and act based on the following cases:

- **Find-Set**(i) **equals Find-Set**(j) **AND** i.*color* **equals** j.*color*: We exit and claim that the graph is not bipartite.
- **Find-Set**(i) **equals Find-Set**(j) **AND** i.*color* **does not equal** j.*color*: Do nothing and keep on iterating through the edges.
- **Find-Set**(i) **does not equal Find-Set**(j) **AND** $Rep[i]$.*color* **does not equal** $Rep[j]$.*color*: Perform a normal UNION(i, j)
- **Find-Set**(i) **does not equal Find-Set**(j) **AND** $Rep[i]$.*color* **equals** $Rep[j]$.*color*: w.l.o.g., assume we are unioning $Rep[i]$ into $Rep[j]$. So, change $Rep[i]$.*color* to the opposite color and $Rep[i]$.$children\_flipped$ to True, and then perform a normal UNION(i, j).

If we eventually get through all the edges and we have not found an edge that triggers the first condition, we claim that the graph is bipartite.

`Correctness:` For the correctness, I am going to analyze the correctness of a much simpler version of the above algorithm. However, we will see that the added complexity is purely for the sake of runtime and will not affect the actual outcome. Imagine we do the same as above but we ignore the whole part about flipped children. Additionally, we have the invariant that in a tree, all the nodes are colored such that a node's direct children all have a different color from it (starts off trivially true because there are no children). Instead of doing a flip children, when we are joining $Rep[i]$ to $Rep[j]$, if they are the same color, we simply flip the colors of every node in the tree. In terms of runtime, we can see that this has the unfortunate effect of creating an $O(|V||E|)$ runtime, which we obviously can't do. However, keeping track of the flipped children is simply a way of delaying the expensive operation until we can 'package' it in with the FIND-SET. This is very similiar to what we did in pset 3 to keep track of ratios. Thus, if we can prove that this version of the problem is correct, we would prove that the version in the answer is correct. Proving that this is corect is fairly simple. We know that if we try to connect two nodes that have the same color, then the graph will no longer be 2-colorable, because tehre is not way to adjust the colors to make sure they are always alternating. This is equivalent to creating an odd numbered cycle, which violates the bipartite principle. Therefore, because we are maintaining the invariant of no loops as we add edges, when we find an edge that forces us to violate this invariant, we know this edge prevents it from being a bipartite graph. If no edge does so, then we know it is possible to 2-color this graph!

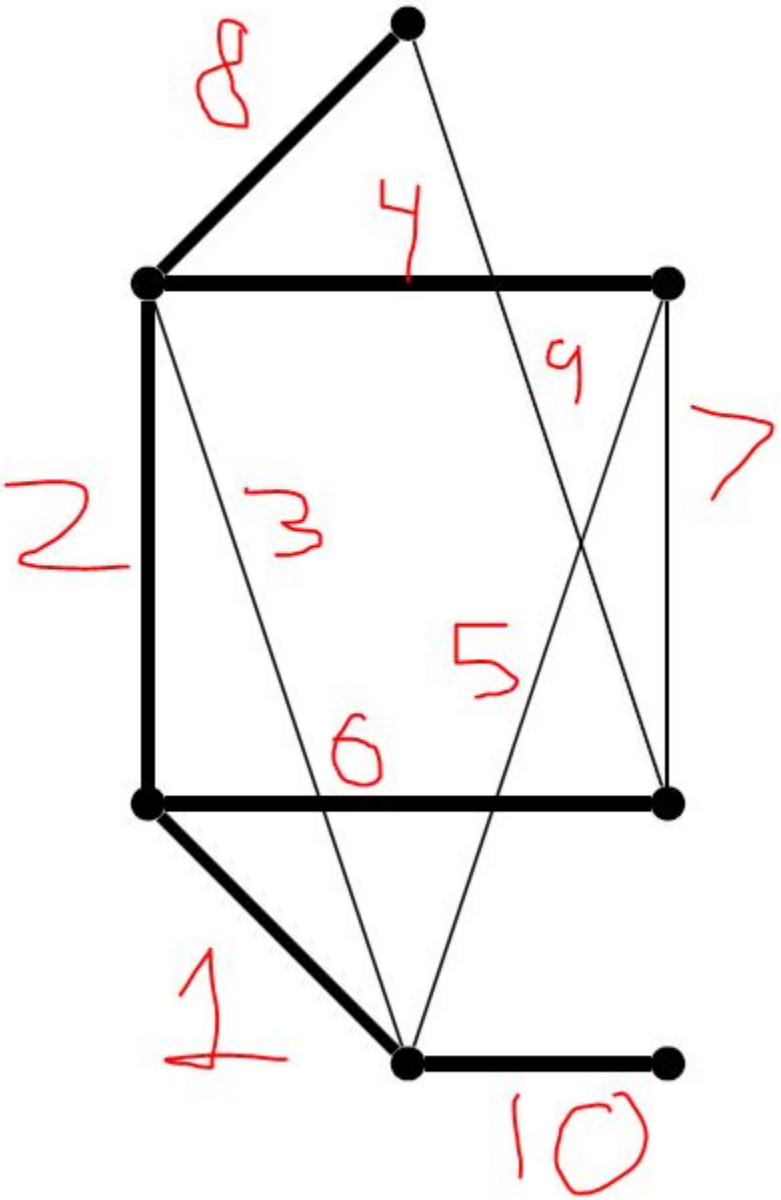
`Runtime/Space:` In the worst case, we are looking at all $|E|$ edges. So, let us consider how much work we could possibly do with each edge:

- We perform two FIND-SET operations, both of which are performed in amortized $O(1)$ time. If an edge falls under this case, we will need to perform $2 * O(1) = O(1)$ work.
- We perform two FIND-SET operations, both of which are performed in amortized $O(1)$ time. If an edge falls under this case, we will need to perform $2 * O(1) = O(1)$ work.
- We perform two FIND-SET operations and a UNION operation, all of which are performed in amortized $O(1)$ time. If an edge falls under this case, we will need to perform $3 * O(1) = O(1)$ work.
- We perform two FIND-SET operations and a UNION operation, plus the constant time work of changing a single node's properties, all of which are performed in amortized $O(1)$ time. If an edge falls under this case, we will need to perform $3 * O(1) + O(1) = O(1)$ work.

Thus, since for each edge we are performing worst case $O(1)$ amortized work, the entire algorithm runs in $|E| * O(1) = O(|E|)$ time, which beats our bound of $O(|E|logV)$. Keep in mind that the amortized constant term is in actuality $O(\alpha(|V|))$, so it is actually $O(|E|\alpha(|V|))$.

In terms of space, as we proved in part A, a union-find only ever requires $O(V)$ space, so we are good on that end.

Problem 2



Problem 3

(A)

`Algorithm:` First, let us consider how many characters we have to consider. Since it is a ring, the worst case senario is that the word s of length m is composed by the 'last' letter we look at, followed by the first m-1 letters. Therefore, if we choose an arbitrary starting point, we need to look at the next n+m-1 = $O(m+n)$ characters in the ring. Now, let us find polynomial representations for the n+m characters we will need to look at (denoted as A) and the length m target string (denoted as B). Assume that the two symbols we found being used in the ring are represented by p and q. Thus, for A, we say its coefficient $a_i = 1$ if the i^{th} character of A is p, and -1 if it is q, $\forall i \in [0, m+n]$. We do the same thing for b, and say its coefficient $b_i = 1$ if the i^{th} character of A is p, and -1 if it is q, $\forall i \in [0, m]$. Thus, we get:

$$a(x) = a_0 + a_1 x + ... + a_{n+m-1} * x^{m+n-1}$$
$$b(x) = b_{m-1} + b_{m-2} x + ... + b_0 * x^{m-1}$$

Now, we compute $c(x) = a(x) * b'(x)$ where $b'(x)$ is the same polynomial but with reversed order of coefficients, using FFT. Then, we return the set $\{k s.t. c_k = m\}$.

`Correctness:` This is the same exact problem as 1.4 from Recitation 5. Please see the proof tehre for correctness. In this case, since we want the exact string, we would leave the hamming distance d=0.

`Runtime/Space:` Converting the strings into A and B takes $O(m+n)$, FFT is $O((m+n)log(m+n))$, and finding the coefficients of the right size is $O(m+n)$. so, in total, it is $O((m+n)log(m+n))$.

(B)

`Algorithm:` First, let us consider how many characters we have to consider. It's the same setup as above. Now, let us find polynomial representations for the n+m characters we will need to look at (denoted as A) and the length m target string (denoted as B). However, since we now have $|\Sigma|$ different alphabets, our approach will be to construct two polynomials and perform FFT on each one. Therefore, We will have $|\Sigma|$ different polynomials, represented by a_σ and b_σ for σ in {1,2,3,...,$|\Sigma|$}.

Thus, for a_σ, we say its coefficient $a_i = 1$ if the i^{th} character of A is $\Sigma[\sigma]$ or a wildcard, and 0 otherwise $\forall i \in [0, m+n]$. We do the same thing for b_σ. Thus, we get:

$$a_\sigma(x) = a_0 + a_1 x + ... + a_{n+m-1} * x^{m+n-1}$$
$$b_\sigma(x) = b_{m-1} + b_{m-2} x + ... + b_0 * x^{m-1}$$

Now, we compute $c_\sigma(x) = a_\sigma(x) * b'_\sigma(x)$ where $b'_\sigma(x)$ is the same polynomial but with reversed order of coefficients, using FFT. So, given what our coefficients were, we know that for a given $c_\sigma(x)$, its coefficient at position k, $c_\sigma(x)_k$, it can take on a value x from 0 to m, where x represents the number of 'agreements' that our target string had with the substring of the ring ending at position k. Recall, an 'agreement' can only happen in $c_\sigma(x)$ if 1. the character in the target string is $\Sigma[\sigma]$ and so is the character in the ring string or 2. the character in the target string is $\Sigma[\sigma]$ and the ring string character is a wildcard. So, after we have run this FFT process for all possible σ, we can define $c(x) = \sum_{k \in \sigma} c_{\sigma k}(x)$. Therefore, the coefficient c_k is the sum of all the agreements of the substrings that ended at k for every letter in the alphabet. Therefore, we should return all k for which c_k is greater than or equal to m, because if it is equal to m, then it means that we had enough agreements over the course of all the letters in the alphabet to perfectly match a string that ended at that point, and we should return that as a location where the string ended.

`Correctness:` This is the same exact thing as above, with a couple key differences. We are now using the AND instead of the XNOR, which we successfully accounted for by using 1 and 0 instead of 1 and -1. Secondly, we are doing this for each of the letters in the alphabet, which means we are trying to solve the problem individually first for each letter, then simply summing up our answers. Since the 'agreements' are non overlapping, this means that we are accounting for all the possible different letters in the alphabet without double counting.

`Runtime/Space:` Converting the strings into A and B takes $O(m+n)$, FFT is $O((m+n)log(m+n))$, and finding the coefficients of the right size is $O(m+n)$. So, in total, it is $O((m+n)log(m+n))$ per letter of the alphabet. Since we have $|\Sigma|$ letters, total running time is $O(|\Sigma|(m+n)log(m+n))$.

(C) This is going to be there exact same as above. The only changes are going to be that now all the wildcards are going to be on B. Now is when we run the risk of double counting, because the wildcard on the target text could trigger an 'agreement' multiple times for a single string without actually being valid. Therefore, we are going to instead return all k for which c_k is greater than or equal to $m + |\Sigma|$. The correctness and runtime analyses are the same except for the few differences I detailed.