
Problem Set 5 Solutions

This problem set contains **4 problems**.

This problem set is due after the break **at 10:00pm** on **Monday, March 30, 2020**.

Important: Since we significantly extended the due date for this problem set, *you will NOT be able to use any late days for this assignment.* Solutions will be posted early on *March 31st* so that you have time to review the solutions prior to our midterm.

EXERCISES (NOT TO BE TURNED IN)

Streaming

- Devise a streaming algorithm that keeps track of the average and standard deviation of a stream of integers.

Fast Fourier Transform

- Do Exercise 30.1-1 in CLRS on page 905.
- Do Exercise 30.1-4 in CLRS on page 905.
- Do Exercise 30.2-4 in CLRS on page 914.

Minimum Spanning Tree

- Do Exercise 23.1-5 in CLRS (pg. 629)
- Do Exercise 23.2-8 in CLRS (pg. 637)

Problem 5-1. An Enemy of My Enemy [50 points]

Fakebook Inc. is a social media website for the less amicable, where instead of making friends, users can designate "enemies." These relationships are encoded in an undirected graph $G = (V, E)$, where V represents the set of users, and two users v_i and v_j are enemies if and only if $(v_i, v_j) \in E$ (indeed, the enemy relationship must be mutually acknowledged).

Seryl, a social analyst, wants to determine whether the old adage "an enemy of an enemy is a friend" applies to Fakebook's social network. In particular, Seryl wants to determine whether we can partition the vertices into two sets $V_1, V_2 \subseteq V$ of "mutual friends", i.e., where no two users in the same set have an edge between them. In other words, Seryl wants to determine if G is bipartite.

Due to a recent surge in hostility, the graph G has suddenly become *dense*, such that $|E| = \omega(|V|)$ (or equivalently, $|V| = o(|E|)$). Unfortunately, this makes it infeasible to store all of G in memory. Your task is to help Seryl by coming up with an efficient algorithm to determine whether G is bipartite, given a stream of edges and only $O(|V|)$ space.

Note: For the sake of simplicity, assume each vertex can be represented in $O(1)$ space by an integer in $\{1, ..., |V|\}$ (i.e. ignore the bit-complexity of storing integers).

(a) [15 points] **Warm Up:** Let E_k represent the first k edges in the stream, and let $G_k = (V, E_k)$. Devise an algorithm that uses $O(|V|)$ space and returns the smallest k such that there exists a cycle in G_k. Be sure to specify which data structures you are using and analyze their space complexity.

Solution:

When designing a space-bounded streaming algorithm, key is to understand the minimum information needed to carry out the task. To do so, one needs to define a notion of a "state" that encapsulates the information that we have at any point in time.

In this case, we know that we can stop once we detect a cycle. So, if we haven't stopped, we know that G_k is acyclic and, therefore, G_k is a collection of trees (a forest). Do we need to remember all the edges seen so far? On close examination, we see that the only relevenat information needed is the set of connected components of G_k. Indeed, if the new edge e falls within the same connected component, then we know that e closes some cycle. Otherwise, e either creates a new component (if e is an isolated edge), or it joins two existing components.

We can use the UNION-FIND data structure to keep track of the components. Space is linear in the size of the universe ($O(|V|)$), and time is $O(|E| + |V| \log(|V|)$, or $O(|E| + |V|\alpha(|V|))$ with the path compression and union by rank optimizations.

(b) [35 points] Devise an algorithm that determines whether or not G is bipartite using only $O(|V|)$ space. For full credit, your algorithm should run in $O(|E| \log |V|)$.

Hint: Consider modifying your algorithm from (a) to detect when certain properties of bipartite graphs are violated.

Solution:

We follow the same idea as in part (a). Notice that once G_k is deemed to be non-bipartite, then any subsequent graphs are also non-bipartite. Therefore, the task is to find the first time when G_k stops being bipartite. As before, if we haven't stopped, it means that G_k is a collection of connected bipartite graphs.

What is the minimum amount of information needed to carry out the task? Do we need to keep all previous edges? Let's try to see what happens if we only keep the partitions (A_i, B_i) (for $i = 1, \ldots, l$) of each of the connected bipartite graphs, where l is the number of connected components. That is, every previous edge e connects a node from A_i to a node from B_i for some i. What happens when we process a new edge e? If, for some i, e lies within A_i or within B_i, then e closes an odd cycle and the graph becomes non-bipartite. If e connects A_i with B_i for some i, then we can through away e as it does not bring any new relevant information. If e connects two previously disconnected bipartite graphs, then we need to join the sets properly. Say e connects A_i with B_j for $i \neq j$. In this case we can join the two bipartite graphs into $(A_i \cup A_j, B_i \cup B_j)$. If e connects A_i with A_j, then we can join the two components to form $(A_i \cup B_j, A_j \cup B_i)$. Notice that, in both cases, we keep the invariant that all previous edges connect A_i with B_i for some i. Finally, if the edge e is an isolated edge, then we increase l and create a new bipartite graph.

How do we keep track of the collection (A_i, B_i), for $i = 1, \ldots, l$? The problem looks similar to that of keeping a set of disjoint subsets (which is easily handled by UNION-FIND). In our case, however, subsets come in pairs. To accommodate this new requirement, we modify the UNION-FIND implementation by having the representative of each subset point to the representative of its pair. To see how to update the information, we look at one of the cases. The rest are handled in a similar manner. Assume that the new edge connects A_i with A_j. As explained before, in this case we need to form the new sets $(A_i \cup B_j, A_j \cup B_i)$. We join both A_i with B_j and A_j with B_j using the largest rank heuristic. We then have the surviving representatives point to each other.

As in (a), the space used is $O(|V|)$ and the time complexity is $O(|E|+|V|\alpha(|V|))$ with the usual optimizations for UNION-FIND.

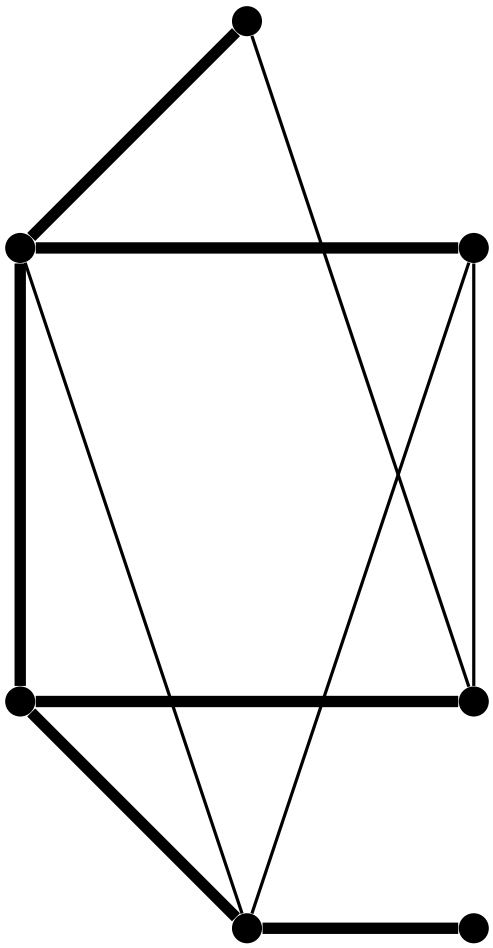
Problem 5-2. Build Your Own MST [15 points] Label the edges of the graph below with weights from the set

$$\{\mathbf{1}, \mathbf{2}, 3, \mathbf{4}, 5, \mathbf{6}, 7, \mathbf{8}, 9, \mathbf{10}\}$$

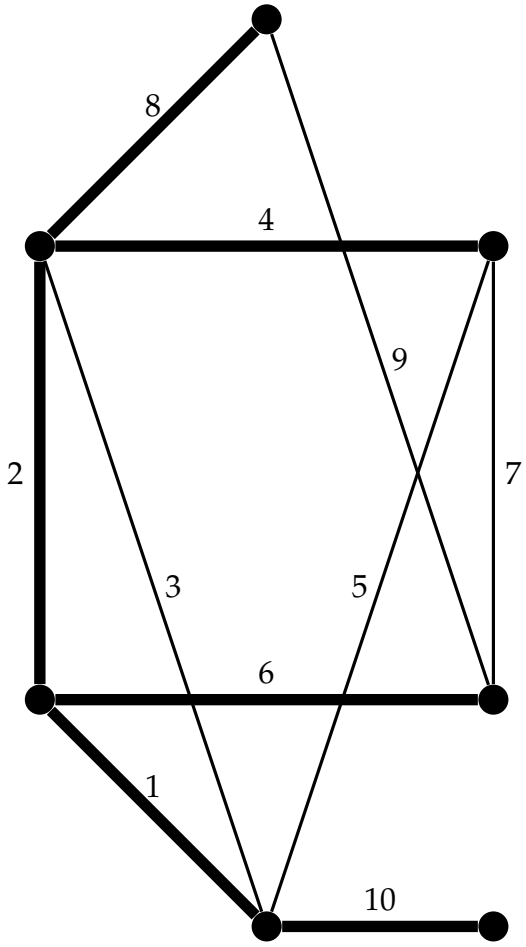
so that:

- each weight is used *exactly* once;
- the thick edges in the graph below have the weights that are in bold face above;
- the thick edges form an MST in the resulting graph.

No justification is needed (nor will it be considered).



Solution:



Looking at the list of edge weights, we note that all the weights are distinct; thus, once we assign the weights to edges, the corresponding MST will be unique. Also, the heaviest weight (10) is supposed to be in the MST, and while the two lightest edges (1,2) *are* in the MST, the third lightest edge (3) is *not*. The cycle property tells us that the heaviest unique edge in a cycle can not be in any MST. Here that implies that 10 can not be the weight of any edge in a cycle; because there is only one edge not in a cycle in the given graph, we assign 10 to that edge. How do deal with the 3? If it is not in the MST, and all remaining edges are in cycles, then it must be the heaviest edge in whatever cycle it is in. A little thought reveals that it must be in a three-edge cycle with 1 and 2 as the other weighted edges in the cycle. The placement shown, or switching the 1 and 2, are the only ways to accomplish the goal. We also need to exclude 5, 7, and 9 from the MST. To exclude the 5, we put it in cycle with 1, 2, and 4. To exclude the 7 we put it in a cycle with 2, 4, and 6. To exclude the 9, we put it in a cycle with 2, 6, and 8.

Problem 5-3. The Fellowship of the Ring [25 points]

Balbo Biggins has found three rings covered in mysterious inscriptions. Each ring has a continuous line of n characters drawn from an alphabet Σ.

(a) [10 points] One of the rings seems to only use two of the symbols of Σ. Balbo knows it contains a length-m word s, but wants to find where s appears in the ring. Develop an algorithm that uses FFT to find all occurrences of s in the ring in $O((n+m)\log(n+m))$ time.

Solution:

This problem requires ideas very similar to those in the solution to the Binary Strings problem in Recitation 5.

The difference here is that one of the strings is a ring, and we need to deal with this. But this is easy to do. All we need to do is cut the ring in an arbitrary position and replicate the first m characters at the end to get a string of length $n+m$. We now have a straightforward matching problem.

Without loss of generality, let us denote $a, b \in \Sigma$ as the two symbols in the alphabet that appear on the ring. We represent the word s as a polynomial $S(x)$ of degree $m-1$. Let s_i represent the ith character of s. Define $S(x) = \sum_i [[s_i = a]]x^i - \sum_i [[s_i = b]]x^i$. That is, we use -1 and 1 to represent the two symbols and create a polynomial where the coefficient of x^i is the number corresponding to s_i.

The inscription on the ring can be encoded as a length $n+m$ sequence of 1 and -1's as well (replicate the first m elements at the end of the sequence to account for cyclic rotations). We create a polynomial representing the ring as $R(x) = \sum [[r_i = a]]x^{m+n-i} - \sum [[r_i = b]]x^{m+n-i}$, where r_i is the ith character on the ring (pick a starting point arbitrarily). Note that $R(x)$ is reversed in order to facilitate convolution via polynomial multiplication.

Use FFT to multiply the two polynomials. The coefficient of x^i of the product will correspond to a shift of i. If everything matches, the coefficient at the matching offset will have value m indicating that every character in s was matched.

Total time: $O((n+m)\log(n+m))$

(b) [10 points] Balbo inspects the second ring and tries to find a second length-m word s in the inscription, but the ring has several unreadable characters, so he concludes they don't matter. Therefore, an unreadable character on the ring can match *any* one character from s. Develop a $O(|\Sigma|(n+m)\log(n+m))$ algorithm to perform the matching.

Solution:

Here we need new ideas for two reasons: 1) we now have an alphabet with more than two symbols, and (2) we need to deal with the wild-cards. The fact that $|\Sigma|$ appears in the target for the running time gives us some hints.

For each letter $\sigma \in \Sigma$, we find whether the ring and word match when considering only the letter σ. For this we use a technique similar to (a). For the word s, we map all occurrences of σ to 1 and all other characters to 0 and get the corresponding polynomial $S(x)$. We do similarly to the ring, except that we also map all wild-cards to 1 and get a polynomial $R(x)$. When we multiply $S(x) * R(x)$, the coefficient at a given position will equal the number of matches of σ. We say that a position is *good* for σ if the coefficient equals the number of occurrences of σ in s (note that it cannot be greater). If a position is good for σ, then all occurrences of σ in s were properly matched by either a σ in r or by a wild-card.

We do this for every character σ and we output those positions that were good simultaneously for all characters in the alphabet. In those positions, the ring matches s on every $\sigma \in \Sigma$.

The running time is clearly $O(|\Sigma|(n+m)\log(n+m))$ as the time needed to process each character in Σ is $O((n+m)\log(n+m))$.

(c) [5 points] Balbo looks at the third ring and is relieved that all of the characters are readable! Unfortunately, the length-m word s he wants to match to the ring has some unreadable characters. As before, an unreadable character in s can match *any* one character from the ring. Modify your algorithm from part (b) to solve this under the same runtime constraints.

Solution:

This looks almost the same as part (b) but in reverse. However, there is a small problem that needs to be dealt with.

In the solution for part (b), to verify whether a position was good for σ, we needed to compare the coefficient of the resulting polynomial to the number of occurrences of σ in the word. In part (b), this was easy as the word s (the one without the wild-cards) was fixed. Therefore, the number of occurrences of every σ was also easily precomputed.

This is not so easy when the wild-cards appear in the word s. Now, the number of occurrences of σ changes in every possible window of length m in the ring. Luckily, this count can be precomputed for every position in time $O(|\Sigma|(n+m))$. We just make a scan of the ring, keeping track of the number of occurrences in a window of size m. Every time we move the window forward, we increment the count for the character that we add, and decrement the count for the character that we drop.

Once these counts are precomputed, the solution is identical to part (b).

Problem 5-4. Feedback Form [10 points] Please fill out a feedback form about this problem set at

`https://forms.gle/sB9PwZcWybKS1PHm6`.

It should not take more than a few minutes and will greatly help us improve teaching and material for future semesters!