Homework 3. Due May 7

CS180: Algorithms and Complexity
Spring 2018

GUIDELINES:

- Upload your assignments to Gradescope by 5:59 PM.
- Follow the instructions mentioned on the course webpage for uploading to Gradescope very carefully (including starting each problem on a new page and matching the pages with the assignments); this makes it easy and smooth for everyone. As the guidelines are simple enough, bad uploads will not be graded.
- You may use results proved in class without proofs as long as you state them clearly.
- Most importantly, make sure you adhere to the policies for academic honesty set out on the course webpage. The policies will be enforced strictly. Homework is a stepping stone for exams; keep in mind that reasonable partial credit will be awarded and trying the problems will help you a lot for the exams.

1. Give an algorithm based on BFS that given a graph $G = (V, E)$ (in adjacency list representation) checks whether or not G has a cycle. For full-credit, your algorithm should run in time $O(|V| + |E|)$. Prove that your algorithm works (you can use properties of BFS that we stated in class without further proving them). [.75 points]

   Solution There are several ways to solve the problem.

   First solution: One way is to use the following fact: any tree on k vertices has exactly $k - 1$ edges. By contrapositive, this implies that if a connected graph on k vertices has more than $k - 1$ edges then it is not a tree and hence must have a cycle.

   A tempting algorithm, is to just use the above observation directly: Count the number of edges in G and if this is more than $|V| - 1$, then G has a cycle. However, this is not correct because G may not be connected. We can fix this by using BFS to find the connected components and then apply this idea to the connected components.

   Recall that in class we used BFS to split a graph into connected components: Pick a vertex $s \in G$, and run BFS(s) and add all vertices reached by BFS(s) to the connected component containing s and we repeat this process with vertices not already reached.

   Concretely, here is an algorithm that given a graph $G = (V, E)$ returns lists $C_1, C_2, ...$, that contain the vertices in various connected components of G:

(a) Initialize an array `Discovered[ ]` with `Discovered`$[i]$ = `False` for all i.

(b) Set $i = 1$. Set $t = 1$.

(c) If `Discovered`$[i]$ is `False`:

i. Run $BFS(i)$ using the above defined `Discovered` array for marking discovered nodes (i.e., we use the same array for all the BFS's). While doing the $BFS(i)$ add the new vertices discovered to a new list C_t.

ii. Increment t.

iii. Set i to be the next index more than i that is still marked undiscovered.

Now that we have the vertices involved in the various connected components in $C_1, C_2, \ldots,$, we can do the following:

(a) For each C_t:

i. Count the number of edges in C_t. You can do this by looking at $(\sum_{u \in C_t} degree(u))/2$.

ii. If this number of edges is more than $|C_t| - 1$, then `Return` "G has a cycle".

Another approach for the problem is to do implement the BFS algorithm and look for cycles. Here I describe the algorithm assuming G is connected. If not, you have to use the decomposition approach above to first split the graph into connected components and then apply the algorithm on the pieces.

Intuitively, when running the BFS algorithm. Recall the BFS algorithm presented in class. There, when exploring the graph if we visit vertex that is already discovered, we don't do anything. However, conceivably this could be due to a cycle. The only issue is that you might visit the vertex again just because you are exploring its immediate neighbor. This can be fixed by keeping a 'parent' marker as below.

(a) Initialize Discovered[u] = false for all vertices u.

(b) Pick a vertex $s \in G$, and set Discovered[s] = true.

(c) Set $L[0] \leftarrow (s)$. Set $i \leftarrow 0$. Set parent[s] = s.

While $L[i]$ is not empty:

(a) Set $L[i+1] \leftarrow \emptyset$.

(b) For each vertex $u \in L[i]$:

i. For each neighbor v of u:

A. If Discovered[v] is true, then: Check if $parent[u] = v$. If yes, do nothing. If not, `Return` "G has a cycle".

B. Else: (a) Set Discovered[v] = true. (b) Add v to $L[i+1]$. (c) Set parent[v] = u.

(c) Increment i.

The analysis of run-time is the same as that of BFS (we just added a few lines each which take $O(1)$ time to the iterations). So the run-time will be $O(|V| + |E|)$.

For correctness, first observe that following the 'parent' links gives you the BFS tree associated with s. Now, consider an execution of step (A) of the algorithm. As v was already discovered,

it means that v belongs to one of the layers $L[0], \ldots, L[i]$. Further, as v is not a parent of u, it means that the edge $\{u, v\}$ is not part of the tree. Thus the graph G contains an edge that is not part of the BFS tree. This means that G has a cycle for adding an edge to any tree creates a cycle.

2. Problem 4.1 from here (Chapter 4 of [DPV]). For part (a), you don't have to show how the d-values are computed, just show the values after each iteration in a table with columns indicating the vertices and rows denoting the iterations. For part (b), it suffices to draw the final shortest path tree. [.75 points]

3. Consider the interval scheduling problem we studied in class: Given a sequence of requests with start and finish times $(s(i), t(i))$, $i = 1, \ldots, n$, find a set of non-conflicting jobs of maximum possible size. Show that the following algorithm solves the problem correctly (i.e., returns a set of non-conflicting jobs of maximum size).

   Latest Start Time (LST):

   (a) Set $R \leftarrow \{1, \ldots, n\}$, and $A \leftarrow \emptyset$.

   (b) While $R \neq \emptyset$:

      i. Pick request $i \in R$ with the latest start time.

      ii. Add i to A.

      iii. Remove all requests that conflict with i (including i) from R.

   (c) Return A.

   The goal of the problem is to give you some practice in analyzing a greedy algorithm and to better understand the analysis of EFF that we did in class. So for full-credit, you cannot assume the analysis of 'Earliest Finish First' (you can use the ideas though) and your answer must be comparable in detail to our analysis of EFF in class. [.75 points]

   (Hint: You can follow the same approach we used for analyzing Earliest Finish First in class.)

Solution This can be proved in a way analogous to what we did in class. In particular, you can compare LST to another potentially different optimal solution O and argue that LST 'starts sooner' (much like we showed EFF 'frees-up first' in class). Concretely, we first show the following Lemma by induction.

Lemma: Suppose that $A = \{i_1, i_2, ..., i_k\}$ where $s(i_1) > s(i_2) > ... > s(i_k)$, and an optimal solution is $O = \{j_1, j_2, ..., j_m\}$ where $s(j_1) > s(j_2) > ... > s(j_m)$, then $\forall l \leq k, s(i_l) \geq s(j_l)$.
Induction on l.
Base case: $l = 1$. True since $s(i_1)$ was the latest start time.
Induction step: Suppose claim is true for l, we want to show for $l + 1$.
We claim that when we picked i_{l+1}, j_{l+1} also belongs to the set R, since $s(i_l) \geq s(j_l) \geq f(j_{l+1})$ (j_{l+1} does not conflict with i_l). So $s(i_{l+1}) \geq s(j_{l+1})$.

Then we can show that Latest start time finds an optimal set of jobs.
Supose there exist $O = \{j_1, j_2, ..., j_m\}$ where $s(j_1) > s(j_2) > ... > s(j_k) > ... > s(j_m)$ and

$A = \{i_1, i_2, ..., i_k\}$ where $s(i_1) > s(i_2) > ... > s(i_k)$, such that $m > k$.
By the Lemma, $s(i_k) \geq s(j_k)$
$\implies$ j_{k+1} does not conflict with the jobs in A.
$\implies$ j_{k+1} is not removed.
$\implies$ The set R is not empty.

4* Given an undirected graph $G = (V, E)$, a subset of vertices $I \subseteq V$ is an independent set in G if no two vertices in I are adjacent to each other. Let $\alpha(G) = \max\{|I| : I \text{ an independent set in } G\}$. The goal of the following questions is to give an efficient algorithm for computing an independent set of maximum size in a tree. Recall that a *leaf* in a graph is a vertex of degree at most 1 and that every acyclic graph (graph without any cycles) has at least one leaf.

Let $T = (V, E)$ be an acyclic graph on n vertices.

(a) Prove that if u is a leaf in T, then there is a maximum-size independent set in T which contains u. That is, for every leaf u, there is an independent set I such that $u \in I$ and $|I| = \alpha(T)$. [.3 points]

(b) Give the graph T as input (in adjacency-list representation), give an algorithm to compute an independent-set of maximum size, $\alpha(T)$, in T. To get full credit your algorithm should run in time $O(|V| \cdot |E|)$ (or better) and you must prove correctness of your algorithm. You don't need to analyze the time-complexity of your algorithm and it is sufficient to solve this part assuming part (1) (if you want) even if you don't solve it. [.45 points]

(Hint: You can try a greedy approach where you add vertices one after the other based on property (1).)

Solution (a) . Let u be a leaf in T, and I be a maximum-size independent set.
If $u \in I$, then we are done. If $u \notin I$, then the only neighbor node v of u must be in I (otherwise, you can create a bigger independent set by adding u to I). Now, let J be the set obtained by removing v from I and adding u. Clearly, $|J| = |I|$ and J is also an independent set as the only new vertex we added was u but it has only adjacent vertex v which was removed. Therefore, J is a maximal-size independent set.

(b). Part (a) tells us that any leaf in an acyclic graph is *safe choice* for constructing a maximum-size independent set (MIS): there always exists a MIS that contains the leaf. We will design algorithm by exploiting this iteratively: We pick a leaf u in T and add it to S. We then delete the neighbor of u if one exists from T. And then repeat the process.

MIS(T):

(a) $S \leftarrow \emptyset$. Set $R = T$.

(b) While R is not empty:

i. Find a leaf u in R and add u to S.

ii. If u has no neighbors, then remove u from R.

iii. If u has a neighbor v, then delete u and v from R. (Delete here means you remove the corresponding adjacency lists of u and v and all mentions of v from the other lists.)

(c) Return S.

We next proof correctness. We will in fact show that the set S is a "safe choice" at every stage of the algorithm in that there always exists a MIS that contains S. We will do so by induction on the number of iterations (or equivalently on the size of S).

Base case: The statement holds for the first iteration by part (a).

Induction step: Assume claim holds after m iterations for some $m \geq 1$. We would like to show it for the next iteration. Let I be an MIS that contains the current S, let R' be the graph after m iterations and let u be the leaf in R' picked for adding to S.

Now, $I \setminus S$ must be a MIS in the graph R' (as I cannot contain any neighbors of vertices in S and if is not a MIS in R', then we can get a bigger MIS for T). Conversely, for any MIS J in R', $S \cup J$ is a MIS in T. Now, by part (a), there exists an MIS I' in R' that contains the leaf u. Now we are done as $S \cup I'$ is a MIS in T and it contains $S \cup \{u\}$ as required.

It now follows by induction that the set S the algorithm computes is always a safe choice.

Finally, to finish the argument first observe that by the above arguments there must be a MIS I that contains the final set S (when R becomes empty) the algorithm computes upon termination. Suppose for the sake of contradiction that $|I| > |S|$. However, in this case any of the vertices in $I \setminus S$ would never be deleted from R so R would not be empty leading to a contradiction. Thus S must be a MIS when the algorithm terminates.