CSEP521: Algorithms October 28, 2022

Homework 2

Anup Rao Due: November 6, 2022

Read the fine print[1]. Each problem is worth 10 points:

1. Give an algorithm to detect whether a given undirected graph has a cycle. If the graph has a cycle, your algorithm should output the cycle. The algorithm should run in time $O(m+n)$, where m is the number of edges and n is the number of vertices. HINT: Argue that if there is a cycle in a particular connected component, then some edge of that cycle must not be a part of the shortest path tree. Show that the breadth first search algorithm can be modified to detect any edge that is not part of the shortest path tree and so find a cycle.

   Solution. We will modify the BFS to obtain the algorithm to find a cycle:

   (a) For $i = 1, \ldots, n$ do the following steps.

   (b) If the vertex i has been discovered, do nothing. Otherwise, run the BFS algorithm from class to construct the shortest path tree for the connected component of i. During this execution if an edge (u, v) is explored that is *not added* to the shortest path tree, then compute the least common ancestor z of u, v in the tree. Output the path from z to v, followed by u and the path from u to z. These vertices form a cycle.

   (c) If no cycle is discovered above, output that there is no cycle.

   Proof of correctness: If the algorithm discovers an edge u, v above, then u, v combined with the two paths in the shortest path tree will certainly be a cycle of the graph. Moreover, if the graph contains a cycle, then some edge (u, v) of the cycle cannot belong to the shortest path tree, since the tree is acyclic. This edge (u, v) will be discovered when u is explored.

   Runtime analysis: Since the algorithm is a modification of BFS and BFS takes the runtime $O(m+n)$, the runtime for this algorithm would is also the same, as we have an additional n operations at most to output the cycle.

2. You are given two sorted lists of integers of length m and n. Give an $O(\log m + \log n)$ time algorithm for computing the k'th smallest integer in the union of the lists.

   Solution The pseudocode is given below.

[1] In solving the problem sets, you are allowed to collaborate with fellow students taking the class, but **each submission can have at most one author**. If you do collaborate in any way, you must acknowledge, for each problem, the people you worked with on that problem. The problems have been carefully chosen for their pedagogical value, and hence might be similar to those given in past offerings of this course at UW, or similar to other courses at other schools. Using any pre-existing solutions from these sources, for from the web, constitutes a violation of the academic integrity you are expected to exemplify, and is strictly prohibited. Most of the problems only require one or two key ideas for their solution. It will help you a lot to spell out these main ideas so that you can get most of the credit for a problem even if you err on the finer details. Please justify all answers. Some other guidelines for writing good solutions are here: `http://www.cs.washington.edu/education/courses/cse421/08wi/guidelines.pdf`.

Input: Two sorted list A of size m and B of size n
Result: k smallest number in the union of the lists
if $k > m + n$ **then**
  return null;
end
if $k \leq 1$ **then**
  return min(A[0], B[0])
end
if $m/2 + n/2 > k$ **then**
  if *A[m/2] > B[n/2]* **then**
    k cannot appear in A[m/2+1] to A[m];
    Update list A = A[0:m/2];
  end
  else
    k cannot appear in B[n/2+1] to B[n];
    Update list B = B[0:n/2];
  end
end
else
  if *A[m/2] > B[n/2]* **then**
    B[0] to B[n/2] smaller than the k smallest number;
    Update list B = B[n/2:n];
    Update k = k - n/2;
  end
  else
    A[0] to A[m/2] smaller than the k smallest number;
    Update list A = A[m/2:m];
    Update k = k - m/2;
  end
end
Recursion the above steps. Return hasCycle

Runtime Analysis: In each recursive step the algorithm removes half of the either list, so the size of the A reduce to $m/2$ or siz of B reduce to $n/2$, $T(m) = T(m/2)+1$ or $T(n) = T(n/2)+1$, A list costs $O(\log m)$ time and B list costs $O(\log n)$ time, so total run time is $O(\log m + \log n)$.

Proof of Correctness: For number k, there are only two cases: either k is less than $m/2 + n/2$ or at least $m/2 + n/2$. When k is less than $m/2 + n/2$, we compare the median of two list, if $A[m/2] > B[n/2]$, we know $A[m/2]$ is larger than $m/2 + n/2$ elements in the union set, so all the elements larger than $A[m/2]$ are larger than k smallest element, so it is safe to delete all the elements from $A[m/2]$ to $A[m]$. Same situation can be applied on $B[n/2] > A[m/2]$

When k is at least $m/2+n/2$, we also compare the median of two list. If $A[m/2] < B[n/2]$, we know $A[m/2]$ is at most to $m/2+n/2$ elements in the union set. Hence, all the elements smaller

than $A[m/2]$ are less than k smallest element. We can delete all the elements from $A[0]$ to $A[m/2-1]$ and update the size of k to $k-m/2$. Same reasoning holds when $B[n/2] < A[m/2]$.

Each step we safely shrink the list size. When $k = 1$, we can get the smallest element by simply compare the smallest elements in two lists.

3. Here's a problem that occurs in automatic program analysis. For a set of variables $x_1, \ldots, x_n$, you are given some equality constraints, of the form $x_i = x_j$ and some disequality constraints, of the form $x_i \neq x_j$. Is it possible to satisfy all of them? For instance, the constraints $x_1 = x_2, x_2 = x_3, x_3 = x_4, x_1 \neq x_4$ cannot be satisfied. Give an efficient algorithm that takes m constraints over n variables, and outputs whether they can be satisfied or not in time $O(m \log n)$. HINT: Use the union-find data structure.

Solution. The pseudocode is given below.

Input: m constraints, n variables
Result: True if all constraints can be satisfied
Make a Union-Find data structure U for n variables, each of them is in its own set;
for *all equality constraints* $a = b$ **do**
  $U.union(a, b)$;
end
for *all unequality constraints* $a! = b$ **do**
  if $U.find(a) == U.find(b)$ **then**
    return $false$;
  end
end
return $true$

Runtime Analysis: This algorithm loops over all m constraints, and each operation takes $O(\log n)$. Thus the time complexity is $O(m \log n)$.

Proof of Correctness: Equality has transitive property that if $a = b$ and $b = c$, then $a = c$. So it's correct to put equal variables into the same set (by Union operation) when processing through all equality constraints first. After processing all equality constraints, we can check if these sets can satisfy all inequality constraints by the idea that if $x \neq y$, x and y cannot be in the same set. And we can use Find operation to figure that out.

4. You are given a graph G with n vertices and m edges, and a minimum spanning tree T of the graph. Suppose one of the edge weights $w(e)$ of the graph is updated. Give an algorithm that runs in time $O(m)$ to test if T still remains the minimum spanning tree of the graph. You may assume that all edge weights are distinct both before and after the update. HINT: If $e \in T$, consider the cut obtained by deleting e from T. If $e \notin T$, consider the cycle formed by adding e to T.

Solution. The pseudocode is given below.

```
Input: Graph G(V, E), MST T, edge e ∈ E, weight w' that e will be updated to
Output: Whether or not T remains the MST of G after weight(e) = w'
if e = {u, v} ∈ T then
    Delete e from T;
    Use depth first search in T from u to all reachable vertices and mark each
     vertex as being in S;
    for all edges e' in E do
        if e crosses the cut S and weight(e') < w' then
            return False;
        end
    end
end
else
    Compute the path in T from u to v;
    if weight(the heaviest edge in the path) > w' then
        return False;
    end
end
return True;
```

Runtime Analysis: All graph traversals take $O(n)$ time, and looping over all edges takes $O(m)$. So the total time complexity is $O(n+m)$ which is $O(m)$ because n is not greater than m.

Proof of Correctness: There are two cases in this problem, and the algorithm solves each one separately.

First, when e is in T, consider the cut S obtained after removing e. By the cut property, the edge that connects the two cuts to make the MST needs to have the lightest weight among all other crossing edges. Thus, if $weight(e)$ is not the smallest, T is no longer the MST.

Conversely, if $weight(e)$ is still the lightest weight edge, then we claim that T must remain an MST. To see this, suppose T' is an MST in the new graph of lighter weight. Then without loss of generality T' must contain e, for if T does not contain e, we can exchange e for an edge of T' (one that belongs to the cycle formed by adding e to T' to obtain another MST). Then we see that since T' and T both contain e, T' must have had lighter weight than T even before e's weight was updated, which is a contradiction.

The second case is when $e \notin T$. For T to still be the MST, e needs to be a heaviest edge in the cycle forming by adding e to T (by the cycle property, the heaviest edge in any cycle is always not in the MST).

Conversely, if e is a heaviest weight edge in such a cycle, then T must remain an MST. To see this, suppose T' is an MST. We can assume that T' does not contain e, for if it does, we can exchange e for an excluded edge from the cycle (which must be of the same weight as e, since e is of heaviest weight). Since both T and T' do not contain e, if T' has smaller weight than T after the update, it must have had smaller weight before the update, which is

a contradiction.