CPSC-320 Intermediate Algorithm Design and Analysis Winter 2000

Solutions to Homework # 3

1. (b) To test if a graph is bipartite, we can try to construct a pair of sets V_1, V_2, such that all edges of the graph connect an edge from V_1 to V_2. To record for each node whether it is in V_1, V_2, or not yet assigned, we use an array A of size n, with one entry per node. Initially each entry is 0, indicating that the node is not yet assigned to either V_1 or V_2. An entry of 1 or 2 indicates when a node is in V_1 or V_2, respectively.

   To start our construction, pick any node, say v, and put it in V_1 (that is, set $A[v] = 1$). All of v's neighbors must be in V_2; put them there. All neighbors of nodes in V_2 must be in V_1, and so on. Using either a depth first or breadth first approach, we can visit the nodes of the graph, starting at v. When visiting node i, if its parent in the search was put in V_1 then put i in V_2 and vice versa. If it ever occurs that two nodes assigned to the same set are connected by an edge, we conclude that the graph is not bipartite.

   This idea works for a connected graph. Note that if the graph is not connected, we need to check separately that all connected components of the graph are bipartite.

   Here is the same algorithm, expressed in pseudocode:

```
BIPARTITE($G$)
begin
    //using DFS, use a 1-d array A to keep track of which
    //group a node is in.

    initialize array $A[\,]$ to 0 values.
    output SEARCH($v, 1$), where $v$ is some
    starting node
end

SEARCH($v, group\_number$)
begin
    $vertex\_group[v] \leftarrow group\_number$
    for each $w$ adjacent to $v$ do
        if ($vertex\_group[w] = 0$) then
            if group_number = 1 then SEARCH($w, 2$)
            else SEARCH($w, 1$)
        else if ($vertex\_group[w] = group\_number$) then
            return FAIL.
    return SUCCESS.
end
```

   This is a slight variant of DFS, which we proved in class was $O(n + m)$, which is linear time.

   (c) The correctness of the algorithm can be argued as follows.

First, suppose that the input graph is not bipartite. Then for **any** two sets V_1 and V_2, with $V = V_1 \cup V_2$, there will be at least one edge connecting two nodes in V_1 or two nodes in V_2. Let $V_1 = \{v \,|\, vertex\_group[v] = 1\}$ and $V_2 = \{v \,|\, vertex\_group[v] = 2\}$. Since there must be an edge connecting two nodes in V_1 or in V_2, two adjacent nodes must be assigned the same group number. The algorithm returns FAIL on the second call to the algorithm to these nodes.

Now suppose that the input graph G is bipartite. Assume that the algorithm returns FAIL. Then there must be adjacent nodes u and v which were assigned the same group_number. In order for two nodes to be assigned the same group_number, they must be connected by some path of even length, since the group number assignment alternates along each chain of recursive calls. Hence the graph G must contain a cycle of odd length (since u and v are connected by a path of even length **and** are adjacent). However, a bipartite graph cannot contain a cycle of odd length, so we have a contradiction. Thus, the assumption that the algorithm returns FAIL is false, i.e. the algorithm must return SUCCESS.

Pitfalls: it is important to notice that V_1 and V_2 are not part of the input to the problem. Rather, the problem is: given a graph G, determine if *there exists* a way to divide the nodes of G up into two sets, V_1 and V_2 so that all edges of the graph cross between V_1 and V_2.

The data structure needed here to store the group (V_1 or V_2) to which each node belongs is very simple - an array with one entry per node.

2. The algorithm described is incorrect. Following is an example graph on which it fails. Can you see why?

```
      1     2
  f <-- e <--- d
  ^    ^^      ^
  |   / |      |
10|  /1 |3     |-7
  | /   |      |
  |/    |      |
  a --> b ---> c
     4     1
```

3. The following code is to be run on both machines. Array A is on machine P. Array B is on machine Q.

```
Spot(A,n)
{
   Sort(A)
   Output(Nth(A,n,n,0)
}
Nth(A,n,Tot_N,Num_Dropped)
{
   mid <- middle element of A
   Send(A[mid],Q)              /* Send median element */
   Send(n,Q);                  /* Send # of elements in array to Q.*/
   Receive(Mid2,Q)             /* Receive median element from Q. */
   Receive(n2,Q);              /* Receive # of elements in B */
   if (n1 == 1) || (n2 == 1)
     goto BaseCase
   else
     Num_Dropped += mid;
     Nth(A[mid..n],n-Mid -1, Tot_N, Num_Dropped);

BaseCase:
/* The base case is kind of tricky.  I didn't grade if the student had
   all the cases right.  What is known is that the regions in A and B
   must contain the median.  So a special case could be to transfer
   a constant number of elements from A to B along with the associated
   ranks.  And then the median can be calculated.
*/
```

Proof of Correctness

At each iteration, elements from each array are eliminated that CAN NOT be the median. This means that those portions of the arrays that remain always contain the median element. This is true after each call. Therefore, at the time of the base case, we are guarenteed to have the median element in one of the regions.

Message Analysis

At each step in the process, the algorithm reduces the size of the problem. by one half. The base is some constant size of the problem.

T(n) = T(n/2) + O(1)

T(n) = O(ln n)

Alternate method 1

Some students chose to copy all the data from one machine to the other and then sort the data on the machine that contained all the data.

Correctness

All the data appears on one machine, so correctness only depends on being able to select the nth largest item.

Message Analysis

The number of messages passed is O(n).

Alternate Method 2

This method is similar to the original method. Instead of halving both arrays at each iteration, this method only halves one array. Assume this array is A.

In a constant number of messages check that A[1] and A[n] are not the nth element.

```
Get_Nth(first, last, size of array)
{
 while (size of array > constant number of elements)
 {
   Send the middle element to the other machine;
   Other machine returns the rank of the sent element wrt to array B's
      elements
   If middle + returned rank == n
     done
   else
      if middle + returned rank > n // Eliminates the top half of array
                                    // since none of those elements could
                                    // possibly be the nth element.
         Get_Nth(first, mid - 1 , mid - first)
      else
         Get_Nth(mid+1, last, last - mid) // The bottom half get eliminated.
 }

//This is the base case.

   Send the array elements that are still candidates, say A[i]..A[j] plus
   A[i-1] and A[j+1] to the other machine.  Along with the associated ranks.

}

On the OTHER machine:

   With the possible candidates from A and the associated ranks, determine
   if any of these observations are the nth observation. If none are the nth
   observation, then the nth observation must be in B.

//  This serves as a proof of correctness also.  The following analysis
//  will fail if the possible candidate portion of the array contains the
//  nth element, because the A[j+1]st element doesn't exist.  To get around
//  this, work with the jth element instead of the j+1st element.

   We have the following

       B[s] <= A[i-1] <= B[s+1]    s + i - 1 < n
```

```
    B[t] <= A[j+1] <= B[t+1]    t + j + 1 > n

This implies that all B[e] <= B[s] can not be the nth element.  We also
know that B[e] >= B[t+1] can not be the nth element.  So we are left
with a region of B that contains the possible candidates for the
nth element.

So we have the following possible candidate lists.

    A[i] ... A[j]

    B[s+1] ... B[t]

Now just search both lists for the nth candidate
```

Message Analysis

To determine if either A[1] or A[n](and in the process if either B[1] or B[n]) is the nth observation O(1) messages are required.

To reduce the problem to a base-case size:

T(n) = T(n/2) + O(1)

To communicate the base-case candidates from one machine to the other : O(1)

To communicate the result back to the other machine : O(1)

Overall running time is O(ln n)

4. A straightforward solution to this problem is simply to do a dfs or bfs from each node in turn. On the search from node i, see if all nodes of the graph are reached. If so, there is an arborescence rooted at node i. Since each search takes linear time and in the worst case n searches are done, this algorithm takes time $O(n(n+m)$.

   There is a way to do better, however! Suppose that a dfs is done starting at some node s. Consider the resulting dfs tree, call it T. Clearly, all nodes in T are reachable from s. But all nodes in the tree may be reachable from other nodes in the tree, too. For example, if a node x in the tree happens to have an edge back to the root s (this edge is NOT in the dfs tree but IS in the input graph), then all nodes in the tree are also reachable from x.

   Let's call a node x in the dfs tree T rooted at s *representative* of T if all nodes in T are reachable from x. It is not too hard to see how one could mark all of the representative nodes in the tree in time linear in the size of the tree. When creating the tree in the first place, one can mark all of the nodes in the tree that have an edge back to the root s. In another pass, one can mark all nodes on any path of the tree from the root to a marked node. (One can combine these two passes with a little care).

Why do we care about representative nodes? We know that ALL of the nodes in tree T, and nothing else, can be reached from the representative nodes. If T has fewer than n nodes, then it is necessary that there is some node in the remaining graph that has an edge to a representative node, in order that the graph has an arborescence. Therefore, once the dfs tree rooted at s is computed, it is useful to mark all nodes of T that are *not* representative as "processed," mark all other nodes of T as "representative," and also retain a count of how many nodes are in T.

We can use this information to advantage when starting a dfs from an unexplored node not in T (i.e. a node that is not processed or not representative). Suppose that the next dfs starts at node s'. We would like to know now many nodes are reachable from s'. During this dfs from s', there is no need to explore from nodes that are marked as "processed." Suppose that at some point in the dfs, a representative node N is visited, where N represents tree T. Then, we can simply mark N and all nodes that represent T as "processed" (they are no longer considered "representative"). Also, add the size of T to the total number of nodes reachable from s'. (At this point, the nodes in tree T are all marked as "processed.")

Here is a summary. Initially all nodes are marked as "unexplored." At all times, a node is marked as exactly one of "unexplored," "visited," "representative," or "processed." Once it is "processed" it is not considered further, and for this reason the whole algorithm takes linear time.

```
   for i from 1 to n do
      if i is unexplored then
          if count-reachable(i) = n-1 then return ''arborescence!''
          else
             for each ''visited'' node v (i.e. node in the dfs tree rooted at i) do
                if v is representative of the tree rooted at i then mark as such
                else mark as ''processed''

count-reachable(i)
   count <-- 1

   for each node v adjacent to i do
      if v is marked as ''unexplored'' then
         mark v as ''visited''
         add to dfs tree rooted at i
         count <-- count + count-reachable(v)
      elseif v is marked as processed then
         do nothing (because it was previously counted as part of another tree)
      elseif v is marked as representative of some tree T then
         mark v and all other nodes that represent T as ''processed''
         count <-- count + size(T)

    return count
```