Data Structure and Algorithm

Homework #4 Solution

TA email: dsa1@csie.ntu.edu.tw

Problem 1. More on trees (20%)

If not specified, the following nodes of a tree are numbered from 0 to $N-1$, where N is the number of nodes in the tree.

1. Centroid (7%)

   (a) (3%)

```
let weights[] be an empty array of N elements

DFS_weight(node, prev) {
    weights[node] = 1
    for each child of node do
        if child != prev do
            weights[node] += DFS_weight(child)
        end
    end

    return weights[node]
}

DFS_weight(root, NULL)
```

   (b) (4%)

```
min_max_w = INF
centroid = NULL
let weights[] be an empty array of N elements

DFS_weight(node, prev) {
    max_w = 0
    weights[node] = 1
    for each child of node do
        if child != prev do
            w = DFS_weight(child)
            weights[node] += w
            max_w = max(w, max_w)
        end
    end

    max_w = max(N - weights[node], max_w)
    if max_w < min_max_w do
        min_max_w = max_w
```

```
19          centroid = node
20      end
21
22      return weights[node]
23  }
24
25  DFS_weight(root, NULL)
```

2. Diameter (7%)

   (a) (3%)

```
1   let heights[] be an empty array of N elements
2
3   DFS_height(node, prev) {
4       heights[node] = 1
5       for each child of node do
6           if child != prev do
7               h = DFS_height(child) + 1
8               heights[node] = max(h, heights[node])
9           end
10      end
11
12      return heights[node]
13  }
14
15  DFS_height(root, NULL)
```

   Please note that the algorithm above matches the example given in this problem. However, the 1 on line 4 should be replaced with 0 if the definition of "The height of a tree is the maximum distance from root to a leaf" is used. Sorry for the inconsistency. Both version will be accepted.

   (b) (4%)

```
1   diameter = 0
2
3   DFS_height(node) {
4       height = 0 // See the note below
5       max1 = 0
6       max2 = 0
7       for each child of node do
8           if child != prev do
9               h = DFS_height(child) + 1
10              if h > max1 do
11                  max2 = max1
12                  max1 = h
13              else if h > max2 do
14                  max2 = h
```

```
            end
            height = max(h, height)
        end
    end

    diameter = max(max1 + max2, diameter)
    return height
}

DFS_height(root, NULL)
```

Note that in this problem, we adopt a different definition of height from the one used in (a), "The height of a tree is the maximum distance from root to a leaf." However, both versions will be accepted. Sorry for the inconsistency.

3. Midpoint (6%)

   (a) (3%) The following is a stronger statement than the original problem: the farthest node from the root is always *one of the endpoint* of a diameter path. (For the original problem, the first two cases below can be ignored.)

   Consider a diameter path P with endpoints D_1, D_2, and any chosen root R. Let $d(\cdot, \cdot)$ be the distance function between two points. Suppose F is one of the farthest node from R. There are several cases to consider:

   - $R = D_1$.

     If $F \neq D_2$, then by definition of F, $d(R, F) \geq d(R, D_2)$.

     - If the equality holds, $d(D_1, D_2) = d(R, D_2) = d(R, F)$ and thus F is indeed an endpoint of another diameter path with endpoints R, F.
     - Otherwise, $d(R, F) > d(R, D_2) = d(D_1, D_2)$, a contradiction to the definition of D_1, D_2. Then $F = D_2$.

   - $R = D_2$.

     This is similar to the previous case.

   - $R \neq D_1$ and $R \neq D_2$.

     Firstly, D_1, D_2 should be the leaves of the tree rooted at R. Otherwise, without loss of generality suppose D_1 is not a leaf. Then for any child C of D_1, $d(C, D_2) = 1 + d(D_1, D_2) > d(D_1, D_2)$, a contradiction.

     Let A to be the lowest common ancestor (you can google it if needed) of D_1, D_2, which may or may not be R. Now we have two cases:

     - F is on the path between R, D_i for some $i \in \{1, 2\}$. If $F \neq D_i$, $d(R, D_i) > d(R, F)$ since D_i is the leaf, a contradiction. So $F = D_i$.

- F is NOT on the path between R, D_i for all $i \in \{1, 2\}$. We examine the two cases of $d(R, F) \geq d(R, D_2)$:
  - If the equality holds, with a similar argument above, F is indeed an endpoint of another diameter path with endpoints D_1, F.
  - Otherwise, $d(D_1, F) = d(D_1, R) + d(R, F) > d(D_1, R) + d(R, D_2) \geq d(D_1, A) + d(A, D_2) = d(D_1, D_2)$, a contradiction to the definition of D_1, D_2. So it is impossible for this case to happen.

We see that in any case, F is indeed an endpoint of some diameter path. Note that the property that trees are acyclic is implicitly applied several times.

(b) (3%)

```
int distance[N+100]; //N is number of nodes
int previous[N+100]; //It's a hint for (b)
void DFS(int u, int father, int dis):
    // section A
    previous[u] = father;

    distance[u] = dis;
    for each edge connected (u, v) :
        if v == father:
            continue
        DFS(v, u, dis+1);

int get_farthest():
    node = 1
    for i = 2 to N:
        if distance[i] > distance[node]:
            node = i
    return node

int solve():
    root = 1;
    DFS(root, -1, 0);
    a = get_farthest();
    DFS(a, -1, 0);
    b = get_farthest(); // distance[b] is the diameter
    // section B
    diameter = distance[b];
    for i = 1 to (diameter/2):
        b = previous[b];
    return b; //b is the midpoint
```

Problem 2. More Sorting (25% + 3%)

1. (6%)

   (a) (3%)

   1, 2, 3, 4, 5, 6, 7

   (b) (3%)

   1, 2, 3, 4, 5, 6, 7

2. (5%)

```
int c[k+1] = {0} //Cumulative Mass
preprocessing(input, n, k):
    for(i in 1 to n):
        c[input[i]] += 1
    for(i in 1 to k):
        c[i] += c[i-1]

query(a, b):
    if(a to b is a valid range): // a<=b && a>=0 && b<=k
        if(a==0):
            return c[b]
        else:
            return c[b] - c[a-1]
```

   The preprocessing time is the same as the running time of *CountingSort*, so the time complexity is $\Theta(n+k)$. And for any query, we can easily check if the range is valid, and return the answer by substracting two elements from the array c. So the time complexity is $O(1)$.

3. (8%)

   (a) (3%)

   RadixSort with LSD sorting :

initial	501	939	1137	2345	666	34	218
1st pass	501	34	2345	666	1137	218	939
2nd pass	501	218	34	1137	939	2345	666
3rd pass	34	1137	218	2345	501	666	939
final pass	34	218	501	666	939	1137	2345

   (b) (5%)

   The time complexity of *RadixSort* : $O((n+r)\log_r k)$

   The time complexity of *CountingSort* : $O(n+k)$

   In this problem, $n = 7$, $k = 2345$, and $r = 10$. Thus, the total computational cost of *CountingSort* is much higher than *RadixSort*.

4. (6%)

   The modified *LSD RadixSort* using *MergeSort* for each digit: $[20, 29, 36, 37, 50, 57, 59]$

   The modified *LSD RadixSort* using *HeapSort* for each digit: $[29, 20, 36, 37, 59, 57, 50]$

   The 2 results are different. Since *HeapSort* is not a stable sorting, it gets a wrong sorting result in the end. On the other hand, *MergeSort* is a stable sorting, thus the result is still correct.

5. (3%) Bonus

   BucketSort

   In this video, we easily put the numbers into ten buckets, and in each bucket we use another simple sorting algorithm to sort the numbers.

Problem 3. Disjoint Set (15%)

1. (5%)

   (1, 4, 6), (1, 4, 7), (2, 4, 6), (2, 4, 7), (3, 4, 6), (3, 4, 7), (1, 5, 6), (1, 5, 7), (2, 5, 6), (2, 5, 7), (3, 5, 6), (3, 5, 7)

2. (10%)

```
for (nodeA, nodeB, color) in E:
  if color is 'b':
    connect nodeA and nodeB

visit[N+1] = {False}
disjoint_set = []
for i in range(1, N+1):
  if visit[i] == False:
    l = list()
    l.append(i)
    for all i-connected node j:
        l.append(j)
        visit[j] = True
    disjoint_set.append(l)

ans = 0
for S0 in range(len(disjoint_set)-2):
  for S1 in range(S0+1, len(disjoint_set)-1):
    for S2 in range(S1+1, len(disjoint_set)):
      ans += len(disjoint_set[S0])*len(disjoint_set[S1])*len
          (disjoint_set[S2])
return ans
```

The problem can be regarded as several disjoint sets separated by red edges. Within each set, nodes are connected by black edges. First, we form the disjoint sets by given edges. If the edge is black, we connect two nodes. Otherwise(the edge is red), ignore the edge. Next, we choose three sets out of these disjoint sets, calculate the product of their sizes and add it to the answer. Repeat until all combinations are traversed. The summation of products is the answer.

Problem 4. Tiger, Chicken, Worm, Bat, Go! (20%)

There are several approaches to solve this problem. The first one is most interesting and inspiring. I strongly recommend you to read and learn something from it.

Solution 1

IMO, this is the most elegant solution which only needs pure disjoint set. But, only few people solved it with this idea.

Let's consider the easier version of this problem. Suppose the outcomes reported are only `Same`. Then, the information we can learn is and only is that some pair of robots are in the same type. It's a formal structure we can maintain with disjoint set. We can first construct N disjoint sets each consisting of one robot. Whenever we receive a result, we union the disjoint set of `x`-th robot and the one of `y`-th robot. And, actually, all the results will be **valid**. Then, for each query, if two chosen robots are in the same disjoint sets, we can confidently report that the outcome of them must be `Same`. Otherwise, we have no idea about the outcome of them.

To convince yourself that we have no idea about some pair of robots yet, we can view the `valid` results as a graph. If we are given a `valid` result, add an edge between `x`-th robot and `y`-th robot. Then, we can identify the outcome of some pair of robots only if we can find a series of result connecting them. i.e. we know the result of x-th and i_1-th, i_1-th and i_2-th, ..., and i_k-th and y-th. Then, we can tell the relation between x-th and y-th. That is, if we can find a path between x-th and y-th robot on the graph mentioned above, we can identify the outcome of them. In other words, they should be connected. Actually, maintaining disjoint set described in previous paragraph is actaully maintaining connectivity. Therefore, for the 'no idea' case, we can also prove that the outcome is actually unknown.

Let's focus on original problem now. We cannot maintain the disjoint set described above in original problem. Since there are some outcome as `Win`, `Lost`, `Nothing`, disjoint set can only maintain some equivalence relation. For example, for the method of easier version, two robots are in the same disjoint set if **they are the same type**. But, with exactly the same structure, when we view it as a graph, two robots are in the same disjoint set now if **they are connected**. That is, the meaning of one disjoint set can be given in any reasonable way. If we can give other variant reasonable meaning of disjoint set, we can make it more powerful.

Then, we can redefine an element of disjoint set as (i, x) indicating that i-th robot will always choose x(x is one of "Tiger", "Chicken", "Worm", or "Bat".). And define the meaning of disjoint set as: for each two elements in the same disjoint set, one of them will happen iff both of them happen. That is, for each disjoint set, either all of the elements happen or all of the elements not happen. Then, initially, we can create $4 \times N$ disjoint set each contains one of (1,"Tiger"),(1,"Chicken"),(1,"Worm"),(1,"Bat") (2,"Tiger"),(2,"Chicken"),...,(N,"Bat"). For

each result of outcome, suppose it's `Win`. We can first check whether (`x`, Tiger) and (`y`, Tiger) are in the same disjoint set, (`x`, Tiger) and (`y`, Worm) are in the same disjoint set, or (`x`, Tiger) and (`y`, Bat) are in the same disjoint set. If one of above happens, it's clearly an **invalid** result. Otherwise, we should union the disjoint sets of (`x`, Tiger) and (`y`, Chicken), the disjoint sets of (`x`, Chicken) and (`y`, Worm), the disjoint sets of (`x`, Worm) and (`y`, Bat), and the disjoint sets of (`x`, Bat) and (`y`, Tiger). Other outcome can be maintained in similar way. For each query, we should check which one of following happens: (`x`, Tiger) and (`y`, Tiger) are in the same disjoint set, (`x`, Tiger) and (`y`, Chicken) are in the same disjoint set, (`x`, Tiger) and (`y`, Worm) are in the same disjoint set, or (`x`, Tiger) and (`y`, Bat) are in the same disjoint set. If none of above happens, the result will be unknown. (We can still view it as a graph. With similar proof, you can convince yourself that it's an unknown situation).

The method described above takes time complexity of $O((N+Q)\alpha(N))$.

Solution 2

This one requires an idea of graph and maintains the structure with disjoint set.

As described above, we can view the relations as a graph. Suppose we view "Tiger" as 0, "Chicken" as 1, "Worm" as 2, while "Bat" as 3. If given `x` wins `y`, it equivalent to link a directed edge from `x` to `y` with a length of 1 while link a directed edge from `y` to `x` with a length of 3. From x, we can go through each directed edge and sum up the length. Then, we can know the **difference**(outcome) between `x` and others. For example, if `x` wins `y` while outcome of `y` and `z` are `Nothing`, we will link a directed edge from `x` to `y` with length of 1 and a directed edge from `y` to `z` with length of 2. From `x`, following the directed edge, we will meet `z` with total length of 3. Then, we know the **difference** between `x` and `z`. That is, if `x` is "Tiger", `z` will be "Bat". We can easily know the outcome of `x` and `z` now.

Actually, to maintain the structure, we can use disjoint set. Besides storing the parent of each node, we also store the difference with the parent(directed edge from itself and it's parent node). Even with this extra value, path compression can be dealt with. But, even without path compression, it's enough efficient with weighted union. Therefore, this method takes time complexity of $O((N+Q)\alpha(N))$.

Solution 3

This one only needs the concept of graph and seems to be easier to come up with.

Let's directly assume all of the robots are choosing "Tiger". With the graph idea mentioned above, for each result, if `x` and `y` are in the same connected component, we can directly check them. Otherwise, we should add an edge between them and make these two connected components into one. To merge two connected components, since we originally assume all the robots are choosing

"Tiger", the outcome of `x` and `y` may not agree with the result reported. Therefore, we should fix it. We can choose one of the connected component and reassign the one it chooses according to current result. For example, if we are given the outcome of `x`-th and `y`-th is `Win`. But, now `x` choosing "Tiger" and `y` also choosing "Tiger", we should fix `y` as choosing "Chicken". Keep all(some) of **valid** results can help us reassign the choosing item correctly. Then, for each query, we can directly answer the outcome just by comparing the chosen item. However, you should be careful when they are not in the same connected component. In this case, the outcome is unknown.

If for each merging, you always choose to fix the smaller connected component, the overall time complexity will be $O(N \lg N)$(You can try to prove it). And, fortunately, in general, if you randomly choose one of the connected component to fix, the time complexity is also $O(N \lg N)$. But, actually, in worst case, the time complexity will reach $O(N^2)$ if you always choose the bigger one to fix. Some of testcases actually contains these kind of results. If you have bad luck and choose the order I guess you will choose, you may fall into $O(N^2)$ situation and result in Time Limit Exceed.

Since there's an implementation of disjoint set takes time complexity of $O((N + Q) \lg N)$. It's not reasonable to let this solution exceed time limit.

Problem 5. Portal (20%)

Let's describe this problem in the language of Graph.

Given an unweighted directed graph $G = (V, E)$, two vertices $s, t \in V$ and a set of edges E'. For each edge $e \in E'$, you are required to find the length of the shortest path from s to t in the graph $G' = (V, E \cup e)$.

BFS (Breadth-First Search) can find the shortest path from a certain vertex for a given graph in $O(|V| + |E|)$. So directly use BFS $|E'|$ times can solve this problem with time complexity $O(|E'| \times (|V| + |E|))$, which is $O(q \times (n + m))$. This approach can get 40% points.

To get 100% points, we needs more observations. Let P be a shortest path from s to t in graph G', then exact one of the following statement is true.

1. Edge e is not used in path P.
2. Edge e is used in path P.

For case 1., P is also a shortest path from s to t in graph G, so only one BFS process is required for this case among every G'.

For case 2., Let $e = (u, v)$, it can be proved that this path P must be a shortest path from s to u + edge e + a shortest path from v to t. [1] One BFS process gives not only the shortest path from s to t but also from s to any vertex, so it's easy to find the length of the shortest path from s to u. Let G^T be the transpose graph of G [2], then a path from t to v on graph G^T is the reverse of a path from v to t on graph G. Again, a single BFS process can find the shortest path from t to any vertex on G^T, so it also finds the shotest path from any vertex to t on G. It's also easy to find the length of the shortest path from v to t with preprocess.

Here's the pseudo code of the solution to this problem:

```
input G, E
Gt := transpose graph of G

dis[] = use BFS to find the length of shortest path sourced at 1
    in graph G
dis_r[] = use BFS to find the length of shortest path sourced at
    n in graph Gt
(distance are set to Infinite if no path exists)

for e in E:
    (u, v) = e
    print(min(dis[n], dis[u] + 1 + dis_r[v]))
```

[1] The formal proof is omitted here. The main concept is to assume any part (s to u or v to t) of the path P is not the shortest and prove by contradiction.

[2] The graph that all edges are reversed.