Algorithm

Homework 3

CHANG LIU

chang.liu@jhu.edu

Mar. 24, 2012

1

2

```
CheckCircle(G(V, E), degree[], list[])
{
    //degree[i] is the degree of node i
    //list[i] is a list of nodes who is connected with i
    calculate the degree of nodes
    insert nodes whose degree is less than 2 into queue;
    while (queue is not empty)
    {
        if (E >= V)
        {
            There is circle
        }
        node = queue.dequeue();
        if (degree[node] == 0)
        {
            V--;
        }
        else
        {
            E--;
            V--;
            degree[list[node]]--;
            delete node from list[list[node]];
            if (degree[list[node]] == 1)
            {
                queue.enqueue(list[list[node]]);
            }
        }
    }
    if (V != 0)
        There is circle
    There is no circle
}
```

3 Exercise 9.3-3

When use the modified deterministic PARTITION algorithm that takes an element to partition around as an input parameter, we can make Quicksort run in O(nlgn) time.

SELECT takes an array A, the bounds p and r of the subarray in A, and the rank I of an order statistic, and in time linear in the size of the subarray A[p..r] it returns the ith smallest element in A[p..r].

NEW-QUICKSORT(A, p, r)
If p<r
    i = $\lfloor (r-p+1)/2 \rfloor$
    x = SELECT(A, p, r, i)
    q = PARTITION(x)
    NEW-QUICKSORT(A, p, q-1)
    NEW-QUICKSORT(A, q+1, r)

For an e-element array, the largest subarray that NEW-QUICKSORT recurses on has n/2 elements. This situation occurs when n = r – p + 1 is even; then the sub array A[q+1 .. r] has n/2 elements, and the subarray A[p..q-1] has n/2 – 1 elements.

Because NEW-QUICKSORT always recurses on sub arrays that are at most half the size of the original array, the recurrence for the worst-case running time is

$$T(n) \le 2T(\frac{n}{2}) + \Theta(n) = O(n \lg n)$$

4 Exercise 9.3-8

FIND -MEDIAN (X , Y, n)

```
k = ⌈n/2⌉
if k ≤ 2 then
    Merge X and Y , and return the median of the merged array.
if n is odd then
    if X [k] == Y [k] then
        Return X [k]
    else if X [k] < Y [k] then
        Return FIND -MEDIAN(X [k..n], Y [1..k], k)
    else
        Return FIND -MEDIAN(X [1..k], Y [k..n], k)
else
    if X [k] == Y [k + 1] then
        Return X [k]
    else if X [k] < Y [k + 1] then
        Return FIND -MEDIAN(X [k + 1..n], Y [1..k], k)
    else
        Return FIND -MEDIAN(X [1..k], Y [k + 1..n], k)
```

This algorithm is similar to binary search. The time complexity is

$T(n) = T(n/2) + O(1) \Rightarrow T(n) = O(\lg n)$.

5 Problem 9.2

6 Exercise 22.2-4

The time for initialization is $O(n)$, where $n = |v|$. And as with the adjacency lists, the total time devoted to queue operations is $O(n)$. Also, within each queue operation, each location corresponding to the row in the adjacency matrix must be checked, and the time is $O(n)$. Thus the total running time is $O(n + n^2) \to O(n^2)$.

7 Exercise 22.2-5

In the correctness proof for the BFS, the algorithm itself doesn't assume that the adjacency lists are in any particular order, thus it is independent. In the Figure 22.3, if precedes x in Adj[w], we can get the breadth-first tree shown in the figure. But if x precedes t in Adj[w] and u precedes y in Adj[x], we can get edge(x, u) in the breadth-first tree.

9 Problem 22-1

a.1 Suppose (u, v) is a back edge or a forward edge in a BFS of an undirected graph. Then one of u and v, let's say u, is a proper ancestor of the other (v) in the breadth-first tree. Since we explore all edges of u before exploring any edges of any of u's descendants, we must explore the edge (u, v) at the time we explore u. But then (u, v) must be a tree edge.

a.2 In BFS, an edge (u, v) is a tree edge when we set $v.\pi = u$. But we only do so when we set v.d = u.d + 1. Since neither u.d nor v.d ever changes thereafter, we have v.d = u.d + 1 when BFS completes.

a.3 Consider a cross edge (u, v) where, without loss of generality, u is visited before v. At the time we visit u, vertex v must already be on the queue, for otherwise (u, v) would be a tree edge. Because v is on the queue, we have v.d <= u.d + 1 by Lemma 22.3 By Corollary 22.4, we have v.d >= u.d. Thus neither v.d = u.d or v.d = u.d + 1.

b.1 Suppose (u, v) is a forward edge. Then we would have explored it while visiting u, and it would have been a tree edge.

b.2 It is the same as for undirected graphs.

b.3 For any edge (u, v), whether or not it's a cross edge, we cannot have v.d > u.d +1, since we visit v at the latest when we explore edge (u, v). Thus, v.d <= u.d +1

b.4 Clearly, v.d >= 0 for all vertices v. For a back edge (u, v), v is an ancestor of u in the breadth-first tree, which means that v.d <= u.d.

10 BFS/DFS

DFS cannot be used to find a shortest path because it cannot always guarantee to find the shortest path.

11 Disjoint Sets

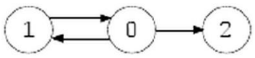
12 MST

Prim algorithm starts with one vertex of a graph as the tree and adds the smallest edge while the tree grows. Whereas Kruskal's algorithm starts with all of the vertices of a graph as the forest and adds the smallest edge that joins two trees in the forest.

These two algorithm focused on two different parts: Prim algorithm looks at only a few edges at a time while Kruskal's algorithm looks at all edges at one time. Thus we can apply 2 algorithms to different cases.

13 Exercise 22.5-3

No.

Consider 

For this algorithm, the first DFS will give a list 1,2, 0 for the 2nd DFS. All vertices will be incorrectly reported to be in the same SCC. Thus the algorithm cannot always produce the correct results.

14 Exercise 23.2-4

We know that Kruskal's algorithm takes O(V) time for initialization, O(ElgE) time to sort the edges, and O(Eα(V)) time for the disjoint-set operations, for a total running time of O(V + ElgE + Eα(V)) = O(ElgE).

If we knew that all of the edge weights in the graph were integers in the range from 1 to |V|, then we could sort the edges in O(V + E) time using counting sort. Since the graph is connected, V = O(E), and so the sorting time is reduced to O(E). This would yield a total running time of O(V + E + Eα(V)) = O(Eα(V)).

If the edge weights were integers in the range from 1 to W for some constant W , then we could again use counting sort to sort the edges more quickly. Sorting would take O(E + W) = O(E) time, since W is a constant. As in the first part, we get a total running time of O(Eα(V)).