COMP4500/7500 Advanced Algorithms & Data Structures
Sample Solution to Tutorial Exercise 4 (2014/2)[*]

School of Information Technology and Electrical Engineering, University of Queensland

August 27, 2014

1. (See CLRS Exercise 22.1-6, p593 [3rd], p530 [2nd], CLR Exercise 23.1-6, p468 [1st])
When an adjacency-matrix representation is used, most graph algorithms require time $\Omega(|V|^2)$, but there are some exceptions. Show how to determine whether a directed graph contains a ***universal sink*** — a vertex with in-degree $|V| - 1$ and out-degree 0 — in time $\Theta(|V|)$, given an adjacency-matrix representation for G.

   In the adjacency-matrix representation of G, an entry $G.edge(u, v)$ being TRUE corresponds to an edge from u to v. A vertex v is a universal sink if row v is all FALSE (out-degree 0) and column v is all TRUE except for the entry in row v. There can be at most one universal sink in a graph.

```
FIND-SINK($G$)
if $G.V == \emptyset$
    return NULL // empty graph has no sink
$S = G.V$
$sink$ = REMOVE($S$)   // The initial candidate for a sink is the first vertex
// In the following predicate, S is the set of vertices yet to be traversed.
// Invariant: $\forall w \in G.V - (S \cup \{sink\}) \bullet \neg(w \text{ sink of } G)$
while $S \neq \emptyset$
    $v$ = REMOVE($S$)   // v iterates through the remaining vertices
    if $G.edge(sink, v)$
        // $(sink, v)$ an edge implies $sink$ is not a sink, but maybe $v$ is,
        // so make $v$ the new candidate to be a sink.
        $sink = v$
    else // $(sink, v)$ not an edge implies $v$ not a sink, but $sink$ may still be a sink,
        // so leave $sink$ as the candidate.
// $\forall w \in G - \{sink\} \bullet \neg(w \text{ sink of } G)$
// The only possible sink is $sink$ but we need to check whether it is.
for each vertex $w \in G$
    // $sink$ cannot be a universal sink if either:
    //   there is an edge with source $sink$ (and destination $w$) or
    //   $w$ is a vertex other than $sink$ and there is no edge from $w$ to $sink$
    if $G.edge(sink, w) \vee (w \neq s \wedge \neg G.edge(w, sink))$
        return NULL
// $\forall w \in G \bullet \neg G.edge(sink, w) \wedge (w == sink \vee G.edge(w, sink))$
return $sink$   // $sink$ is a universal sink of $G$
```

Figure 1: Algorithm to find a universal sink of a graph

Sample solution. The algorithm in Figure 1 maintains $sink$ as a candidate sink vertex. It relies on two properties:

- if there is an edge (u, v), then u cannot be a sink (but v may be a sink), and
- if there is no edge (u, v), for u and v distinct, then v cannot be a sink (but u may be a sink).

The algorithm maintains the following loop invariant. It states that of all the vertices not in S, only $sink$ could possibly be a sink.

$$\forall w \in G.V - (S \cup \{sink\}) \bullet \neg(w \text{ sink of } G)$$

[*]Copyright © reserved August 27, 2014

Both loops sequence through all the vertices, and for each loop every iteration is constant time, so both loops are $\Theta(|V|)$. Hence, FINDSINK is $\Theta(|V|)$.

(Aside: A more careful analysis reveals that this algorithm can be written to examine at most $n-1$ matrix entries in the first loop and at most $2n-2$ in the second loop. An optimal algorithm exists which utilizes tournaments and examines at most $3n - \lceil \lg n \rceil - 3$ entries.)

2. (CLRS Exercise 22.2-4; CLR Exercise 23.2-4)
Argue that in a breadth-first search, the value of $v.d$ assigned to a vertex v is independent of the order in which the vertices in each adjacency list are given.

   Sample solution. The only time that a value is assigned to $v.d$ is the first time that vertex v is discovered in an adjacency list for some vertex u being processed. The value assigned to $v.d$ is independent of the order of the elements in the adjacency list because all the vertices in the adjacency list that are modified (i.e., haven't been visited before) are assigned the same value, namely $u.d + 1$. Note that all unvisited vertices in the adjacency list are assigned a distance before their adjacency lists are examined.

3. (CLRS Exercise 22.4-3; CLR Exercise 23.4-3)
Give an algorithm that determines whether or not a given ***undirected*** graph $G = (V, E)$ contains a cycle. Your algorithm should run in $O(|V|)$ time, independent of $|E|$.

   Sample solution. An undirected graph is acyclic (i.e., a forest) if and only if a depth-first search finds no back edges. (IF: no back edges means only tree edges, which means we have a forest, which is acyclic. ONLY IF: A back edge builds a cycle.)

   So we can run depth-first search: if we find a back edge we have a cycle and stop. But depth-first search is $O(|V| + |E|)$. We observe that the time for cycle discovery is $O(|V|)$ by noting that we must find a back edge not later than seeing $|V|$ distinct edges, since in any acyclic undirected forest, $|E| \leq |V| - 1$. (How do you prove that?)

4. (CLRS Exercise 22.3-11; CLR Exercise 23.3-9)
Show that a depth-first search of an undirected graph G can be used to identify the connected components of G, and that the depth-first forest contains as many trees as G has connected components. More precisely, show how to modify depth-first search so that each vertex v is assigned an integer label $v.comp$ between 1 and k, where k is the number of connected components of G, such that $u.comp = v.comp$ if and only if u and v are in the same connected component.

   Sample solution. We rely on the fact that for an **un**directed graph, a depth-first search initiated at any vertex reaches all vertices connected to that vertex. Thus in the main loop of DFS(G) (below) each unvisited (WHITE) vertex corresponds to a new component. For each new component we increment the variable *component* and this value is assigned to all vertices reached by a depth-first search from an unvisited vertex.

```
DFS-COMPONENT($G$)
1  for each vertex $u \in G.V$
2      $u.color$ = WHITE
3      $u.\pi$ = NIL
4  $time = 0$
5  $component = 0$ // new counter for components
6  for each vertex $u \in G.V$
7      if $u.color$ == WHITE
8          $component = component + 1$ // increment component (new component)
9          DFS-VISIT-COMPONENT($G, u, component$) // component is extra argument
```

```
DFS-VISIT-COMPONENT($G, u, component$)
1  $time = time + 1$
2  $u.d = time$
3  $u.color =$ GREY
4  $u.comp = component$ // label vertex with component no
5  for each vertex $v \in G.Adj[u]$
6      if $v.color$ == WHITE
7          $v.\pi = u$
8          DFS-VISIT-COMPONENT($G, v, component$) // pass unchanged component counter
9  $u.color =$ BLACK
10 $time = time + 1$
11 $u.f = time$
```

5. (CLRS Exercise 22.2-7; CLR Exercise 23.2-7)

The *diameter* of a tree $T = (V, E)$ is given by

$$\max_{u,v \in V} \delta(u, v),$$

that is, the diameter is the largest of all shortest-path distances in the tree. Give an efficient algorithm to compute the diameter of a tree, and analyse the running time of your algorithm. Assume that the tree is represented as an undirected graph and that you are given an algorithm that performs a breadth-first search to find the distance from a single node r to all other nodes.

Sample solution. We run breadth-first search with the source being any node r in the tree. This gives us the distance from r to all other nodes in the tree. From these we pick a node s at maximal distance from r. We run breadth-first search again with node s as the source. The diameter of the tree is the furthest distance of a node from s. The analysis is easy.

Why does this work?

Two useful properties of trees are that the shortest path between any two nodes is unique, and the end points of a maximal-length path are leaves of the tree (otherwise there would be a longer path). As we are only concerned with shortest paths, we just call them paths in the following discussion, and use the notation u–v for the path between u and v.

The first execution of breadth-first search establishes that node s is of maximal distance from r:

$$\forall v : V \bullet \delta(r, s) \geq \delta(r, v).$$

We need to show that given any maximal length path, say u–v, there is a path from s to some node in the tree of the same length. Consider the paths from u, v and s to r, and let

- x be the first common point on u–r and v–r (x may equal r),
- y be the first common point on s–r and u–r, and
- z be the first common point on s–r and v–r.

Either y and z are on the path u–v or both are on x–r.

(a) If y is on x–r, then it follows that $z = y$. Without loss of generality, assume $\delta(v, x) \leq \delta(u, x)$, otherwise swap u and v in the following argument. Hence $\delta(u, v) \leq 2 \cdot \delta(u, x)$. Because s is maximal distance from r, we also know $\delta(s, r) \geq \delta(u, r)$, but y is the first common point on the paths s–r and u–r, so $\delta(s, y) \geq \delta(u, y)$. Therefore,

$\delta(s, u) = \delta(s, x) + \delta(x, u)$	x on path s–u
$\geq \delta(s, y) + \delta(x, u)$	y precedes x on path s–u
$\geq \delta(u, y) + \delta(x, u)$	$\delta(s, y) \geq \delta(u, y)$
$\geq \delta(u, x) + \delta(x, u)$	$\delta(u, y) \geq \delta(u, x)$
$= 2 \cdot \delta(u, x)$	
$\geq \delta(u, v)$	$\delta(u, x) \geq \delta(x, v)$

(b) If y is on u–v then so is z. We consider two subcases: either y is closer to r, or z is closer to r (in both cases we allow $y = z$).

i. If y is closer to r, then $y = x$. We can show
$\delta(s,u) \geq \delta(u,v)$
$\iff \delta(s,z) + \delta(z,u) \geq \delta(v,z) + \delta(z,u)$
$\iff \delta(s,z) \geq \delta(v,z)$
$\iff \delta(s,z) + \delta(z,r) \geq \delta(v,z) + \delta(z,r)$
$\iff \delta(s,r) \geq \delta(v,r)$
The last inequality holds because s is of maximal distance from r.

ii. If z is closer to r, then $z = x$. We can show
$\delta(s,v) \geq \delta(u,v)$
$\iff \delta(s,y) + \delta(y,v) \geq \delta(u,y) + \delta(y,v)$
$\iff \delta(s,y) \geq \delta(u,y)$
$\iff \delta(s,y) + \delta(y,r) \geq \delta(u,y) + \delta(y,r)$
$\iff \delta(s,r) \geq \delta(u,r)$
The last inequality holds because s is at a maximal distance from r.