Homework 3 Solutions
Fundamental Algorithms, Fall 2005, Professor Yap

Due: Thu Nov 10, in class.
SOLUTION PREPARED BY Instructor and T.A.s

INSTRUCTIONS:

- Please read questions carefully. When in doubt, please ask.

Question 1

(30 Points)
(a) Prove that if a bigraph has no odd cycles, then it is bipartite.
(b) Prove that if a connected bigraph has an odd cycle, then BFS search from any source vertex will detect a level edge.
(c) Write the pseudo code for bipartite test algorithm outlined in the text. This algorithm is to return YES or NO only. You only need to program the shell routines. NOTE: Please be concise! All pseudo code must be reduced to $O(1)$ time operations or operations on standard data structures such as queues.
(d) Modify the algorithm in (c) so that in case of YES, it returns a Boolean array $B[1..n]$ such that $V_0 = \{i \in V : B[i] = \mathsf{false}\}$ and $V_1 = \{i \in V : B[i] = \mathsf{true}\}$ is a witness to the bipartiteness of G. In the case of NO, it returns an odd cycle.

ANSWER:

(a) We use the BFS Driver to call $BFS(v_i)$ repeatedly at unseen v_i's. This gives a set of BFS trees. As there are no odd cycles, there are no level edges. Thus every edge is either a tree edge or a cross edge. Hence, if we assign the nodes in even levels to a set U_0, and the nodes in odd levels to the set U_1, we find that all edges are between U_0 and U_1. Hence the graph is bipartite.

(b) Suppose G has an odd cycle $C = [u_0 - u_1 - \cdots - u_{2k}]$ for some $k \geq 1$. If there is any edge $(u_i - u_j)$ where $|i - j| > 1$, we say the cycle C is **reducible**; otherwise it is **irreducible**. If C is reducible, there is a smaller odd cycle containing the edge $(u_i - u_j)$, comprising only of the vertices in C. It is clear that if there is an odd cycle, then there exists an irreducible odd cycle. Hence assume C is irreducible. Wlog, let u_0 be the vertex in C which is seen first from some BFS search. Then u_1, u_{2k} would be nodes in the next level of the BFS tree; u_2, u_{2k-1} would be nodes in the next level, and so on. The last pair of nodes would u_k, u_{k+1}. Of course, $(u_k - u_{k+1})$ represents a level edge in the BFS tree.

(c) We use boolean array B[1...n], and BFS Algorithm.

```
PREVISIT(v,u):
    If (v is seen and B[u]=B[v]) then Throw Exception.
```

```
VISIT(v,u):
    B[v] ← ! B[u]
```

(d) We use another array Parent[1...n] to keep track of the parent node, and give a subroutine for computing the odd cycle:

```
VISIT(v,u):
    B[v] ← ! B[u]
    Parent[v] ← u
```

```
PREVISIT(v,u):
    If (v is seen and B[u]=B[v])
        Return OddCycle(u, v, Parent)
```

where

```
OddCycle
    Input: vertices u, v in a graph G, and array Parent[1..n]
    Output: a vector V containing an odd cycle
    vector V
    V.push-front(u)
    V.push-back(v)
    While (Parent[u] != Parent[v]) do
        u ← Parent[u]
        v ← Parent[v]
        V.push-front(u)
        V.push-back(v)
    V.push-front(Parent[u])
```

Question 2

(15 Points)
Explore the relationship between the traversals of binary trees and DFS.
(a) Why are there not two versions of DFS, corresponding to pre- and postorder tree traversal? What about inorder traversal?
(b) Give the analogue of DFS for binary trees. As usual, you must provide place holders for shell routines. Further assume that the DFS returns some values which is processed at the appropriate place. Be brief.
ANSWER:

(a) In DFS, we have folded pre- and postorder traversal into one program by defining the shells PREVISIT, VISIT, POSTVISIT. Note that VISIT in preorder is like PREVISIT, and VISIT in postorder is like POSTVISIT of DFS. What we call VISIT in DFS is not necessary for tree traversals because the binary tree structure a special kind of digraph. There is no “inorder” analogue for DFS since the concept depends on the “binary” nature of binary trees.

(b) The DFS version of binary tree traversal is basically the merger of preorder and postorder traversal. The program DFS(u) where u is a root of a binary tree goes as follows:

```
DFS(u):
    where u is a node of a binary tree
    PREVISIT(u);
    L← DFS(u.Left);
    R← DFS(u.Right);
    Return (POSTVISIT(u,L,R))
```

Question 3

(25 Points)
A vertex u is called a **bottleneck** if for every other vertex $v \in V$, either there is a path from v to u, or there is a path from u to v.
(a) Give an algorithm to determine if a DAG has a bottleneck.
(b) Prove the correctness of your algorithm.
(c) Analyze the complexity of your algorithm (it should be no more than $O(n^4)$ but hopefully faster).

ANSWER:

(a,b) If u is a bottleneck, the number of descendants of u in G and G^{REV} must be $n+1$, where n is the number of vertices. (Note that a vertex is its own descendent.) We modify DFS(u) to return the number of u’s descendants. We use a local variable ’count’ to sum up the number of VISITed node.

```
DFS(G,u)
        where u is a vertex of graph G
    color u as seen
    count ← 1
    for each v adjacent to u do
        if v is unseen then
            count += DFS(v)
    Return count.
```

Note that the above DFS algorithm can correctly compute the number of descendants only for the root node u. So we modify DRIVER program to call DFS for every vertex.

> DFS DRIVER (G)
>   Compute G^{REV}
>   Assume vertex set is $\{1, \ldots, n\}$.
>   For $i = 1, \ldots, n$
>   Color all vertices of G as unseen
>   Color all vertices of G^{REV} as unseen
>   If $(DGS(G, i) + DFS(G^{REV}, i) = n + 1)$, Return ("Bottleneck i")
>   Return("No bottleneck")

(c) We can compute the reverse graph G^{REV} in time $\mathcal{O}(m + n)$ where m is the number of edges. We call DFS at most $2n$ times, and each DFS is $\mathcal{O}(m + n)$. Therefore, the overall complexity is $\mathcal{O}(n \times (m + n))$. This is $\mathcal{O}(n^3)$.

Question 4

(10 Points)
Please read the revised notes for computing strong components of a digraph. Give a counter example to our original strong component algorithm (find a small an example as possible – we deduct points for unnecessarily large graphs).

ANSWER: $G = (V, E), V = \{a, b, c\}, E = \{(a, b), (b, a), (b, c)\}$. Error occurs when the permutation is $\{b, c, a\}$ or $\{a, c, b\}$.