CSCI 511 Analysis of Algorithms II **Winter 2013 - Johnson**

David Palzer
Homework 2

1. (Exercise 22.1-7 from the text) The *incidence matrix* of a directed graph, G = (V,E), with no self-loops is a $|V|x|E|$ matrix B = $[b_{ij}]$ such that

$$b_{ij} = \begin{cases} -1, \text{ if edge j leaves vertice i} \\ 1, \text{ if edge j enters vertice i} \\ 0, \text{ otherwise} \end{cases}$$

Let B^T denote the transpose of B. That is, $[B^T]_{ij} = b_{ji}$. Let C = $[c_{ij}] = BB^T$. What is the meaning of each c_{ij}, in terms of egdes in the original G.

Solution: Given a matrix B and its tranpose B^T, and C = BB^T, we know that C is symmetrical as it is known that a matrix times its tranpose is symmetrical.

$$B = \begin{vmatrix} -1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & -1 & 0 & 1 \\ 0 & 0 & 0 & 0 & -1 & -1 \end{vmatrix}, B^T = \begin{vmatrix} -1 & 1 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 \\ 0 & 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 & -1 \\ 0 & 0 & 0 & 1 & -1 \end{vmatrix},$$

$$C = BB^T = \begin{vmatrix} 2 & -1 & -1 & 0 & 0 \\ -1 & 3 & 0 & -1 & -1 \\ -1 & 0 & 2 & -1 & 0 \\ 0 & -1 & -1 & 3 & -1 \\ 0 & -1 & 0 & -1 & 2 \end{vmatrix}$$

Looking at C we notice that the elements C_{ii} are equal to the number of edges that interact with vertice i. $C_{ij}, j \neq i$ is negative one if there is an interaction by an edge between vertices i and j in either direction.

2. Let A = $[a_{ij}]$ be the adjacency matric of a directed graph G= (V,E) with no self-loops. Construct and algorithm with asymptotic time complexity $\Theta(|V|)$ that finds a vertex with the in-degree $|V| - 1$ and out-degree zero, if such a vertex exists. If the vertex does not exist, the algorithm should output a statement to that effect.

Solution: Let A be an adjacency matrix:

```
int guess = 0;
for (i = 1; i < |A|; i++)
    if A[guess][i] == 1
        guess = i;
for (i = 0; i < |A|; i++)
```

```
    if i != guess && (A[guess][i] == 1||A[i][guess] == 0)
        print "There is not sink state for this graph"
        return false;
print "Vertice [guess] is the sink state for this graph"
return true;
```

Let us use a loop invariant to prove the correctness of this algorithm.

Statement: We shall look at the first for loop. The variable, guess, holds our current guess for the sink state of the graph based on elimination of states through the loop.

Initialization: Before the first iteration we can see that no states have been eliminated and zero is our current guess.

Maintenance: At the beginning of an arbitrary iteration of the loop we compare A[guess][i] to one. From here we can figure out a couple things, either that the state i is not a possible sink state or that the guess state is not a possible sink state. If A[guess][i] is equal to one then we know that the current guess is not valid as it has an edge leaving it going to i meaning that it has an out-degree larger than zero. If this is true then we set guess equal to i. If A[guess][i] is not equal to one then that means that state i does not have a potential in-degree of $|A|$ - 1 so is not a valid option for a sink state and that our guess state still has a potential out-degree of zero. This means that for each loop we eliminate one state, either the current i or the current guess, and that at the end of the loop, guess is a valid candidate for the sink state.

Termination: After the loop we know that we have the only valid candidate for the sink state of the graph as we have looped $|A|$ - 1 times and eliminated one state each time resulting in one state left.

After the termination of first loop we still have to check to make sure that the guess state actually has an in-degree of $|A|$ - 1 and out-degree of zero. This is because of how the first loop looks at the states. The loop does not look at any spots in the matrix below the diagonal nor necessarily all of the states above the diagonal for our guess. The second for loop of the algorithm simply looks at each value in the column and row of our guess to make sure that it does not have any outgoing edges and that it has an incoming edge from every other vertice. This final test tells us whether or not our guess is a sink state, or if it isnt, that the graph does not have a sink state.

Let us now look at the run time of this algorithm. Looking at the first loop we see that it runs from one to $|A|$ - 1 with $\Theta(1)$ operations inside. That is $\Theta(|V|)$ no matter the input as the input has no effect on the run time. Next we look at the second loop. This loop runs from zero to $|A|$ with $\Theta(1)$ operations inside. That is $O(|V|)$ for the worst case where we have to look at each element and $\Omega(1)$ in the best running time where the first element we see eliminates our final guess. This loop then ends up being within $\Theta(|V|)$ time. By combining

these two loops we see that the run time is $\Theta(|V| + |V|) = \Theta(2|V|) = \Theta(|V|)$.

Based on the above, we can see that the algorithm will either give us the sink of the graph or tell us that the graph does not have a sink in $\Theta(|V|)$ time.

3. (Variation on Exercise 22.2-8 of the test) Consider a tree rooted at vertex r as an undirected graph G = (V,E). For any two vertices u,v $\in$ V, let $\Sigma(u, v)$ denote the shortest-path distance from u to v. Define

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

   Starting with an adjaceny list representation of G as a directed graph, construct a $\Theta(|V|)$ algorithm that computes D. Note that the input adjaceny list will only contain edges from a vertex to its children in the tree; it will not contain the reverse edges, although these reverse edges are used in computing the $\delta(u, v)$ values.

   Solution: First, we know that trees have $|V|$ - 1 edges. Here we can then assume that any operation that takes $\Theta(|E|)$ time is the same as an operation that takes $\Theta(|V|)$ time, that is $\Theta(|E|) = \Theta(|V|)$.

   Lemma 1: We know that the deepest leaf in the tree must be one of the end point of the diameter of the tree.
   Lemma 1 PF: Let there be three vertices a,b and, c, where a and b are leaves in the tree and c is the deepest leaf in the tree.

   Lemma 1, case 1: Let us assume that a, b, and, c all have the same first common parent, where the first common parent is the first parent vertice that any pair of child vertice both have in common. The distance to this common parent from a is distance x, from b is distance y and from c is distance z. We assume that the diameter of the graph is $\delta(a, b)$. This is the same numerical value as x + y and must be larger than any other shortest path. Since c is the deepest leaf in the tree we also know that z is greater than both x or y and so z + x and z + y are then both greater than x + y implying that both $\delta(c, b)$ and $\delta(c, b)$ are greather than $\delta(a, b)$. This is a contradiction since no shortest path can be longer than the diameter meaning that the diameter must include the deepest leaf in the tree.

   Lemma 1, case 2: Let us assume that (a,c) have a first common parent before either (a,b) or (b,c). The distance to this common parent from a is distance x, from b is distance y and from c is distance z. We assume that the diameter of the graph is $\delta(a, b)$. This is the same numerical value as x + y and must be larger than any other shortest path. Since c is the deepest leaf in the tree and further from the common parent than a we know that z is greater than x. This subsequently means that z + y is greater than x + y and implies that $\delta(c, b) > \delta(a, b)$. This is a contradiction since no shortest path can be longer than the diameter meaning that the diameter must include the deepest leaf in the tree.

   Lemma 1, case 3: Repeat case 2 but with (b,c) having a common parent instead

   Lemma 1, case 4: Let us assume that (a,b) have a first common parent before either (a,c)

or (b,c). The distance to this common parent from a is distance x, from b is distance y and from c is distance z. We assume that the diameter of the graph is $\delta(a, b)$. This is the same numerical value as x + y and must be larger than any other shortest path. Since c is the deepest leaf in the tree and further from the common parent than a or b we know that z is greater than both x or y. This results in z + x and z + y also being greather than x + y which implies that $\delta(c, a)$ and $\delta(c, b)$ are greater than $\delta(a, b)$. This is a contradiction since no shortest path can be longer than the diameter meaning that the diameter must include the deepest leaf in the tree.

Algorithm: Breadth First Search the tree starting at the root, r, and return v,$\max_{r,v \in V} \delta(r, v)$. BFS, as proved in class, is an $\Theta(V + E)$ operation, but E = V, so it is $\Theta(V + V) = \Theta(2V) = \Theta(V)$ and within our time constraints. We know that the vertice that was returned must be an ending point the diameter based on lemma 1.
Now we create an undirected adjacency list from the list we currently have. This involves going through the current list and for every edge in the list adding the opposite one into the new list. Once that has occured we now go though and copy over all of the existing edges in the current list to the new list. The new list now contains edges in both directions. This is an $\Theta(2E) = \Theta(2V) = \Theta(V)$ operation and within our time constraints.
We can now BFS (again $\Theta(V)$) on the undirected graph from the deepest leaf we found in our first BFS. This allows us to find the longest shortest path to another leaf in the tree from our starting point. The consequent max shortest distance and vertice that are returned will be the end point of the diameter path of the graph and the numerical value of that diameter.

The run time of the above algorithm is BFS + Adj. List Construction + BFS = $\Theta(V) + \Theta(V) + \Theta(V) = \Theta(V)$.

Based on the above we can see that the alogrithm will return the start and end points of the diameter of the graph and also the length of that diameter within $\Theta(V)$ time.