CS350 Homework 3

Russell Miller

February 28, 2011

22.1-1

The out-degree of a vertex is just its adjacency list, so in $\Theta(V)$ time you can get the out-degree of every vertex. Finding the in-degree for every vertex would require looking for that vertex in every vertex's adjacency list. So the entire adjacency list must be visited, which would be $\Theta(V * E)$, where V is the total number of vertices, and E is the total number of edges.

22.1-3

```
Adjacency List
[(1,[2,3]),
 (2,[4,5]),
 (3,[6,7])]
```

```
Adjacency Matrix
```

	1	2	3	4	5	6	7
1	0	1	1	0	0	0	0
2	0	0	0	1	1	0	0
3	0	0	0	0	0	1	1
4	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0

22.1-3

```
TRANSPOSE(ajd_list A)
new adj_list B
for each v in A
  for each (v,x) in v
    B.add(x,v)
```

```
TRANSPOSE(adj_matrix A)
new ajd_matrix B[A.width][A.height]
for i in A.width
  for j in A.height
    B[j][i] = A[i][j]
```

The running time of the list version of the transpose is $O(|V|^2)$, where V is the number of vertices. In the best case, with a completely disconnected graph, it would only take $|V|$.

The running time of the matrix version is $\Theta(|V|^2)$, because it copies the entire matrix which is width and height V.

22.1-6

Finding a universal sink has 3 parts: identifying the correct vertex, verifying that its in-degree is $|V| - 1$, and verifying that its out-degree is 0.

In order to find a vertex with zero out-degree, the worst-case would be $|V|^2$, so immediately we know this is not where to start. We also know that determining a vertex's in-degree is $|V|^2$.

So as a shortcut, we'll start from an arbitrary row in the matrix, and when a 1 is found in column j, move to row i, where $i = j$. Then find the first 1, repeating this process until a row i is found with all zeros. This could potentially be the universal sink. The last step is to verify that in column j, where $j = i$, there are all 1s. While jumping from row to row, there were a few traversals within each row, for less than V rows. There was also the traversal of the sink row, which is length V. And lastly, there was the traversal of the V-length column. This is $3V$, which is $O(V)$.

This is clever, but it's not actually complete. This is because there are cases when this would be much worse than $O(V)$. Here is the matrix to show a counterexample. Imagine a graph where all vertices are connected to each other, AND the sink, and the algorithm traverses every edge before reaching the sink. It would take $O(V^2)$.

	1	2	3	4	5	6	7
1	0	1	0	0	0	0	1
2	0	0	1	0	0	0	1
3	0	0	0	1	0	0	1
4	0	0	0	0	1	0	1
5	0	0	0	0	0	1	1
6	0	0	0	0	0	0	1
7	0	0	0	0	0	0	0

22.2-8

```
MAZESOLVE(G)
new array visits[ G.edges.length ]
new array status[ G.vertices.length]
for each v in G.vertices
  status[v] = unvisited
endfor
new queue Q
Q.enqueue(G.vertices[0])
status[G.vertices[0]] = discovered
while Q.notempty
  u = Q.dequeue
  for each v in u.adj_list
    if status[v] == unvisited
      ++visits[(u,v)]
      status[v] = discovered
      Q.enqueue(v)
    endif
  endfor
  status[u] = visited
  for each v in u.adj_list
    ++visits[(u,v)]
  endfor
endwhile
```

This algorithm should be setting each value of the 'visits' array to 2. That means each edge has been visited twice. The way this can solve a maze is by traversing the graph, you can find your way back out by following the vertices you've marked twice. Those are the ones that are visited.

22.3-8

Using adjacency-list representation, here is an example graph:

```
{ A : (B,C),
  B : (),
  C : (B) }
```

The depth-first search would produce a time table such as this:

	A	B	C
d	1	2	4
v	6	3	5

This includes a path $C \to B$, where $d[B] < v[C]$.