Zijian Gao
26319560
2/29/2016
CS311 HW2

22.1-3)

```
Transpose_ADJ_List(adj[1..|V|]):
Let adjT[1..|V|] be an array of linked lists representing the transposed adjacency list,
assuming standard linked list operations are available
for (i from 1 to |V|):
   if (adj[i].size > 0)
      for (j from 1 to adj[i].size)
         adjT[adj[i].get(j)].add(i)
return adjT
```

Proof of correctness:
This algorithm iterates over every vertex in adj, compiling a new list that reverses every edge. Vertices that do not have any edges directed out of them can be disregarded as there are no edges to reverse. Each element e_j of the list at vertex i is used as the vertex key for the transposed list, and vertex i is added to that list so where originally, e_j was an element of the list for vertex i, instead vertex i will be an element of the transposed list for vertex e_j.

Runtime:
Let k be the length of the longest list in adj[].
This algorithm takes $\theta(|V|\bullet k(k+1))$ time to run if the size of adj[i] > 0 because the nested for loop on j iterates on each item in the list adj[i], using a linked list get operation which is $\theta(k)$ because at the best and worst case, the algorithm will need to perform a get on the last item in the list at index j and an add operation, which takes $\theta(1)$ trivially for a total of $\theta(k(k+1))$ time as the loop iterates a maximum of k times. The outer loop on i will run the inner loop a maximum of |V| times, the amount of vertices in the graph and also the amount of lists in adj for a total of $T() = \theta(|V|\bullet k(k+1))$ time.

This is efficient because every edge must be reversed, and for that to be possible, each vertex that is a member of some vertex v's list must be visited by the loop since it is known that the total added size of the adjacency list for some directed graph is |E|. One inefficiency is the k^2 time on the inner loop that reverses the edges, and this can be addressed by using an iterator implementation optimization that can access each next vertex in the list for some vertex in constant time, reducing the runtime to $\theta(|V|\bullet(k+1))$

```
Transpose_ADJ_Matrix(adj[|V|][|V|]):
Let adjT[|V|][|V|] be a new matrix representing the transposed matrix.
for (i from 1 to |V|) // i represents the "from" vertex in adj
   for( j from 1 to |V|) // j represents to "to" vertex in adj
      adjT[j][i] = adj[i][j]
return adjT
```

Proof of Correctness:

Similar to the adjacency list, this algorithm iterates over each possible edge, which is |V| x |V|. The value in the transposed adjT[j][i] is set to the value of adj[i][j], which would be 1 if j is reachable from i and 0 otherwise. So, adjT[j][i] is the reversed value, meaning an edge from j to i. This is guaranteed to terminate when the end of the adjacency matrix adj[][] is reached.

Runtime:

The runtime is $\theta(|V|^2)$ because for each "from" vertex (of which there are |V| total) in the matrix, there are |V| total "to" vertices, and each value must be copied in inverse index location at adjT. The algorithm has no prior knowledge of which edges exist within the matrix, so this exhaustive process is necessary and no more efficient algorithm exists.

22.3-2)

Vertex	Start time	Finish Time
q	1	16
r	17	20
s	2	7
t	8	15
u	18	19
v	3	6
w	4	5
x	9	12
y	13	14
z	10	11

Edge classification
q->s tree
q->w forward
q->t tree
s->v tree
v->w tree
w->s back
t->x tree
x->z tree
z->x back
x->y tree
y->q back
r->u tree
r->y cross
u->y cross