Zijian Gao
26319560
4/7/2016
CS311 HW4

22.3.)

a.

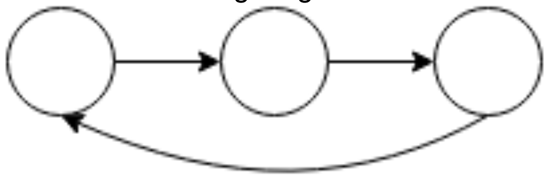
For some strongly connected, directed graph G=<V,E>, it has an Euler tour if and only if in-degree(v) = out-degree(v) for each vertex v of V.

Prove by cases:

Case if G=<V,E> has an Euler Tour, then in-degree(v) = out-degree(v) for each vertex v of V:
Prove by contrapositive, let G' be some graph for which it is not the case that in-degree(v')=out-degree(v') for each vertex v' of V'. This means that it is possible for there to exist more than one vertex with an out-degree of 0, which means it would be impossible to traverse an edge out of those vertices and there would exist vertices that are unreachable by only traversing one edge without backtracking and there would not be an Euler Tour

Case if in-degree(v) = out-degree(v) for each vertex v of V, G=<V,E> has an Euler Tour.
Prove by induction:
IH: for i = in-degree(v) = out-degree(v) for each vertex v of V for graph G=<V,E>, the graph has an Euler Tour

BC: i = 1, since the graph is strongly connected, each node is connected to another node. Since nodes can only have one edge going in and one edge coming out of them, no vertex can be connected to more than two vertices. The most elementary example of this is the following diagram:



All graphs with in-degree(v)=1=out-degree(v) for each vertex will follow some similar permutation/extension of this example. There is trivially an euler path here.

IS: For the same graph with i+1=in-degree=out-degree, remove the extra inbound and outbound edges to reduce the graph to an equivalent graph with i=in-degree=out-degree, this can be done recursively until the base case is reached, and there exists a trivial Euler path.

Conclusion: since both sides of the bi-conditional have been proved, I conclude that for some strongly connected, directed graph G=<V,E>, it has an Euler tour if and only if in-degree(v) = out-degree(v) for each vertex v of V.

b.

```
FIND_EULER_TOUR(G, V, E):

eulertour = Tree()
vertex_queue = Queue()
starting_node = choose_random_vertex(V)
first_run = true
vertex_queue.enqueue(starting_node)
while vertex_queue.length != 0
  v = vertex_queue.dequeue()
  cycle = Tree()
  while v.out-degree > 0
    v.out-degree -= 1
    u = some unvisited vertex in adj[V]
    u.visited = visited
    cycle.addNode(v)
    if v.out-degree > 0
      vertex_queue.enqueue(v) //enqueue if not done exploring out-edges
    v = u //move onto adjacent node to find cycle
  if first_run
    eulertour = cycle
    first_run = false
 else eulertour = UNION(tour,cycle) //use UNION to join root of cycle with the root of tour
return eulertour
```

Proof of correctness:
This algorithm starts at some arbitrary vertex and visits one outbound edge. It will then traverse any reachable vertex via those subsequent outbound edges until it exhausts all outbound edges and produces a cycle (assuming the input graph does contain an Euler Tour). The first cycle that is discovered does not need to be joined into the tour via union trivially. By the fact that for every cycle that is discovered, those outbound edges are removed and vertices are marked as visited, all cycles that are discovered are disjoint from other cycles and so must be joined via the UNION function onto the resulting Euler Tour. Since the cycles are edge-disjoint and every cycle is guaranteed to be visited by the loop invariant, the algorithm is correct.

Runtime:
The while loop visits each out-edge exactly once, with each iteration of the inner loop (while v.out-degree > 0) removing one out-edge until every edge has been visited and every vertex has an in and out degree of 0, and the inner loop contains only constant-time operations.
Since this is the case I conclude that the runtime is $O(|E|)$.