CPS 430/590.06 **Prof. John Reif**
Design and Analysis of Algorithms **Fall 2013**
Homework 4 **Matt Dickenson**

Question 1

For this problem we seek a linear-time algorithm to find an odd-length cycle in a directed graph. Given a graph $G = \{V, E\}$ and a root $v \in V$, we perform a breadth-first search of G. As we do so, we store the vertices of G in a breadth-first tree T. We regard the root of T as the 0^{th} level, its children as the 1^{st} level, and so on. At even levels of T, we color the vertices white, and at odd levels we color the vertices black. Once we have completed the breadth-first search, we then check whether any adjacent vertices have the same color. If so, those vertices belong to an odd-length cycle.

The running time of this algorithm is the time it takes to conduct the breadth-first search, plus the time to go through the tree and check whether each node is adjacent to a node of the same color. Each vertex in the tree can store a reference to its immediate neighbors in the graph, for quick lookup. The running time for the breadth-first search is $O(|V| + |E|)$, and the running time to check each vertex of the tree for an adjacent vertex of the same color is $O(|V| + |E|)$ as well (because each vertex and its neigbors is checked). Thus, the total running time reduces to $O(|V| + |E|)$.

Because we label all vertices according to their distance from the root, then every vertex is either white or black and each edge connects vertices of the same or different colors. If no cycles exist or if the only cycles are of even-length, then every edge will connect two nodes of different colors. This is because the distance between the root (colored white) and any black node is an odd number of edges, and the distance between the root and any other white node is an even number of edges. If two adjacent vertices v_1 and v_2 are the same color, the length of the cycle *without* the edge connecting v_1 and v_2 must be even (because the length of the sides of the cycle are either both odd or both even, and so their sum must be even). Adding the edge connecting v_1 and v_2, the length of the cycle must be odd. Thus, the algorithm is correct.

Question 2

We are interesting in determining, for a directed graph $G = \{V, E\}$, whether or not there exists a vertex $v \in V$ from which all other vertices are reachable. Notice that for two vertices $u, v \in V$ and an the vertices reachable from each of them, denoted $r(u)$ and $r(v)$, if $u \in r(v)$ then $r(u) \in r(v)$. This suggests that a greedy algorithm will work, storing each $r(v)$ as a set. If we reach a vertex for which $r(v) = V$, then we return `true`, else we return `false`. The pseudo-code for this algorithm is similar to Floyd-Warshall, but we store reachability relations rather than the shortest path (we regard the vertices as integers from 1 to $n = |V|$:

```
for each vertex v in V:
  v.reachable = {v}
for each edge (u,v) in E:
  u.reachable = v.reachable

```

```
for each k in V:
  for each i in V:
    for each j in V:
      if k in i.reachable and j in k.reachable:
        i.reachable = union(i.reachable, j.reachable)
        if i.reachable == V:
          return true

return false
```

The running time for this algorithm is $O(|V|^3 + |E|)$.

Question 3

We wish to compute the number of different directed paths from vertex v to vertex w in a directed acyclic graph G. Because the graph is a DAG, we can compute a topological ordering starting at v. We do this by conducting a depth-first search and storing the topological order as the reverse of the order in which vertices are popped off the stack. Once we have the topological order, we then iterate over all vertices $i, v < i < w$, and see whether there is a path from v to w containing i. If there is, we add the path containing i to the existing list of paths. We return the length of the final list of paths.

The time to compute the topological ordering is linear in the number of vertices, $O(|V|)$, and the iteration over i is at most $O(|V|)$. Thus, the total running time is linear: $O(|V|)$.

Question 4

In this problem we wish to design a linear-time algorithm that determine, in a directed acyclic graph G, whether or not there is a directed path that touches every vertex exactly once (i.e. a Hamiltonian path). Notice that if such a path exists then there must be a unique topological sorting of G: the ordering is the same as the path.

To show that this is approach valid, assume that at least two topological orderings of G exist. If this is the case, then at some point in the path there must be two valid options for us to take, v_1 and v_2, and choosing once means that the other cannot be included in the path (otherwise we would have to sort one before the other). Thus, there can be no unique path that visits every vertex exactly once if there is more than one topological ordering.

To do this, we start with a topological sorting of G, which we can compute by conducting a depth-first search and setting the topological order as the reverse of the order in which vertices are popped off the stack. Once we have a single topological ordering, we need to determine whether a second valid ordering exists. To do so, we can iterate over the ordering, checking whether consecutive vertices have an edge between them. If they do not, we can try swapping them and seeing whether the ordering is valid. If it is, then the original ordering is not unique and there is no unique path.

The running time for this algorithm is the time it takes to generate the topological ordering plus the time to iterate over the initial ordering, which reduces to $O(|V| + |E|)$. Thus, we have found a linear time algorithm for determining whether the desired path exists.