Design and Analysis of Algorithms (Sheet #3)

Marius Gavrilescu

1. Remove the given edge, then run a depth first search from one of the nodes on this edge. A cycle containing the edge exists if and only if the other node is reached by the search.

2. Very easy for undirected graphs: run a DFS coloring nodes alternatingly red and blue, if we find an edge connecting two nodes of the same color then there exists an odd-length cycle.

   The same algorithm will work on a strongly connected component, treating all edges as being undirected. Suppose we find two nodes of the same color. Then there is an (undirected) cycle of odd length.

   Let $[v_{n+1} = v_0, v_1, \ldots, v_n]$ be the cycle so that there is an edge from v_i to v_{i+1} or from v_{i+1} to v_i. If all edges are from v_i to v_{i+1}, we found an odd cycle. Otherwise take an i such that there is an edge from v_{i+1} to v_i. Since we are working on a strongly connected component, there has to be a path from v_i to v_{i+1}. If the length of this path is even, then we add the edge from v_{i+1} to v_i to it and we get an odd cycle. Otherwise, we can replace the edge from v_{i+1} to v_i with this path in the original cycle, thus "reversing" it while preserving the parity of the cycle. Since we can reverse any edge, the original (undirected) odd cycle can be turned into a directed odd cycle.

   To find an odd cycle in the whole graph, we can just apply the algorithm above on each of its SCCs.

4. Do a modified breadth-first search which keeps a list of back pointers to all nodes from which the current node can be reached in the minimum amount. Suppose $pi[n, i]$ is the ith back pointer for node n and $pino[n]$ is the number of back pointers for node n. Then:

```
VAR mem : ARRAY NMAX OF INTEGER;
PROCEDURE nr(n : INTEGER);
VAR i : INTEGER;
BEGIN
  IF mem[n] = 0 THEN
    mem[n] := 1;
    FOR i := 0 TO pino[n] - 1 DO
      mem[n] := mem[n] * nr(pi[n,i])
    END
  END;
  RETURN mem[n]
END
```

   Finally `nr(v)` is the answer.

5. The given complexity suggests that we should run Dijkstra's algorithm on the graph $|V|$ times. A very simple solution is to simply run Dijkstra's algorithm from every node to create a table $d[a, b]$ which is the length of the shortest path from node a to node b, or $|V|$ if there is no path. Then:

```
m := 2 * V;
FOR i := 0 TO V - 2 DO
  FOR j := i + 1 TO V - 1 DO
    IF d[i,j] + d[j,i] < m THEN m := d[i,j] + d[j,i] END
  END
END
IF m = 2 * V THEN Out.Line("Acyclic graph")
             ELSE Out.Int(m, 0) END;
Out.Ln
```

7. (a) Say the edge we added connects nodes A and B. Since T (without e) is a spanning tree containing A and B, there is a unique path from A to B that does not contain e. By adding e to this path we get a unique cycle.

   (b) Since the cycle is unique, if we remove one of the edges the resulting graph is acyclic. Suppose now the new graph does not span the tree. This means that there exist two nodes C and D between which there only exist paths that contain e (between A and B). But in any of these paths we can replace edge e by the other path from A to B (the rest of the cycle), which contradicts the assumption that our graph does not span the tree.

   So the resulting graph is a spanning tree.

8. (a) Suppose there exist two different minimal spanning trees.

   Take the smallest edge e in the symmetric difference on the trees. Add it to the other tree. In this new graph, we know we have a cycle. If the largest edge in this cycle is e then we have a cycle in the first tree (false). So there is an edge larger than e in this cycle, which we can remove and get a smaller spanning tree. This contradicts our assumption that there are two different minimal spanning trees.

   (b) Multiply the edge weights of the graph by -1 then run any minimum spanning tree algorithm.

9. (a) Lift them in increasing order of t_i.

   (b) Take any other order. There will be some $i < j$ such that $t_i > t_j$ and i is lifted before j. Then by swapping them in the order every diver lifted after i up to and including j will spend $t_j - t_i$ less underwater.

   (c) i. Take any safe order. If the order is $d_1, \ldots, d_n$ we're done. Otherwise take the minimum i such that d_i is not the ith diver and let d_j be the ith diver. We have $a_j > a_i$.
   We will swap d_i and d_j. Obviously, this does not affect divers lifted before d_j or after d_i in the original order. Now, d_j will be lifted at the same time d_i was lifted before (so earlier than a_i, so earlier than a_j) and d_i is lifted earlier than before. So the only divers we need to check are those between d_i and d_j in our order.
   Let d_k be such a diver. Obviously $a_k > a_i$. Since in the original order they were lifted before d_i, and d_i was lifted before a_i we know that d_k was lifted before $a_i - t_i$. Then, in our new order d_k is guaranteed to be lifted before a_i (because we added d_i before d_k, ignoring the deletion of d_j). But $a_i < a_k$, so diver d_k is safe.
   Therefore if we have a safe order with the first t divers being $d_1, \ldots, d_t$ we can find a safe order with the first $t + 1$ divers being $d_1, \ldots, d_{t+1}$. By induction we can find an order with the first n divers being $d_1, \ldots, d_n$, which is exactly the order we are looking for.

   ii. We know that if there is a safe order then lifting the divers in increasing order of a_i is safe. So we can simply check if the increasing order of a_i is safe. If yes, then we have a safe order. Otherwise, there is no safe order.