S. Schwoon / D. Suwimonteerabuth Winter Term 2008/09

Fundamental Algorithms
Solution Keys 8

1. Assume that nodes are uniquely numbered with 1 to n, where n is the number of nodes. We first give our data structure for representing adjacency lists. Let `Entry` be a record consisting of three items: (i) `node`, an integer; (ii) `edge`, a pointer to `Edge` (the data structure for edges in the lecture); (iii) `next`, a pointer to `Entry`. The item `node` is the node that `edge` points to. An adjacency list is a sequence of `Entry`, connected in a row by using `next`. Therefore, a graph can be represented by an array of length n, whose elements are pointers to adjacency lists. Let us denote the array by `adj` from now on.

   We now propose an algorithm that finds a Euler circle in a graph. The algorithm assumes that the graph is connected and the Euler circle exists. The idea of the algorithm is to traverse the adjacency lists starting from node 1, and find a sequence of edges which leads to node 1 again. The output is a list of `Entry`. Then, we look at each node in the list and repeat the step above by finding loops starting from that node and avoiding the edges that are already visited. The new loops (which is again a list of `Entry`) are inserted into the original list.

   For this, we need to extend the record `Edge` (from the lecture) to include an extra item: `visited`, a Boolean variable. It is initially set to false for each edge, and will be set to true in the course of the algorithm to denote that the edge is already taken into consideration and should be ignored in the future. As a consequence, we need a procedure to find the next entry in an adjacency list that corresponds to an unvisited edge.

```
Procedure nextEntry
Input: a node $i$
Output: the next entry in adj[$i$] that corresponds to an unvisited edge

$t$ := adj[$i$];
while $t \neq$ NIL and $t\rightarrow$.edge$\rightarrow$.visited do
    $t$ := $t\rightarrow$.next;
od
if $t \neq$ NIL then adj[$i$] := $t\rightarrow$.next;
return $t$;
```

   Let x := `new Entry`(a, b, c) be a shorthand for

   $$x := \texttt{new Entry};\ x\rightarrow.\texttt{node} := a;\ x\rightarrow.\texttt{edge} := b;\ x\rightarrow.\texttt{next} := c;\ .$$

   Given a node s and an entry t, the following procedure traverses adjacency lists from entry t, and construct a (new) list of entries where the last entry corresponds to node s. Notice that edges are immediately marked as visited at the beginning of the procedure, and therefore will be ignored (by `nextEntry`) in the future.

Procedure `findPath`
Input: a node s, an entry t
Output: a list of entries, starting at t and ending at an entry with node s

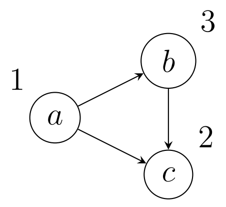
```
$t$→.edge→.visited := true;
if $t$→.node = $s$ then  return new Entry($s$, $t$→.edge, NIL);
return new Entry($t$→.node, $t$→.edge, findPath($s$, nextEntry($t$→.node)));
```

If l_1 and l_2 are lists of entries, we denote in the following by `insert`(l_1, l_2) a procedure that inserts list l_2 into list l_1 at the position immediately after the first entry. We can now construct the main procedure.

Procedure `main`
Input: an array of adjacency lists `adj`
Output: a list of entries representing an Euler circle

```
list := new Entry(1, adj[1]→.edge, findPath(1, nextEntry(adj[1]→.node)));
itr := list→.next;
while itr ≠ NIL do
    while $t$ := nextEntry(itr→.node) ≠ NIL do
        newlist := findPath($t$→.node, nextEntry($t$→.node));
        insert(itr, newlist);
    od
    itr := itr→.next;
od
return list;
```

2. The reverse of a pre-order numbering can violate the partial order! Consider as an example the following graph with a possible pre-order numbering next to the nodes:



After reversing, we obtain the following order: b, c, a; which violates the fact that $c \prec b$, i.e. c is reachable from b.

3. See exercise sheet 9.