Algorithms and Data Structures (WS15/16)

Example Solutions for Unit 20

Problem 1

For strongly connected components:

We show the claim by using basically the same arguments as the one stated in the lecture for proving correctness of the algorithm for identifying the 2-connected components.

- a strongly connected component of a graph $G = (V, E)$ is a set of vertices $S \subseteq V$ such that for every $u, v \in S$ there is a directed path from u to v and a directed path from v to u in G
- we keep in mind the lemma from the lecture: two nodes $u, v \in V$ are in the same strongly connected component if and only if there is a closed directed sequence (cycle with repetition of vertices) of edges that contains both u and v
- stack S_v is used for vertices in open components and stack C for open components (each represented by its first vertex)
- we call a marked node open if there has been no backtracking over it, all other nodes are closed
- for every strongly connected component, there is a unique first node, and a component is open/closed if the first node is open/closed
- we prove the claim by induction over the number of calls of traverse and backtrack:

  After the first t calls of traverse and backtrack, let G_t be the subgraph induced by marked nodes. Denote open components, their node sets, and their first nodes by $G_t^{(i)}$, $V_t^{(i)}$, and $v_t^{(i)}$, for $1 \leq i \leq k_t$, in the order in which $v_t^{(i)}$ were marked. We use the following invariants:

  - The nodes of a closed component point to their first node.
  - On C we have $v_t^{(1)} \prec \ldots \prec v_t^{(k_t)}$ in this order.
  - On S_V we have the nodes from $V_t^{(1)}, \ldots, V_t^{(k_t)}$ in this order.

  All conditions are true in the beginning. Assume they hold until $t-1$ and consider step t in two cases:

  - Case1: traverse(v, e, w):
    * If e is a tree edge, then w represents a new open component containing only w. Obviously, the invariants hold in this case.
    * If e is a back edge, then it closes a directed cycle by traversing tree edges up to w. All the nodes in this directed cycle belong to the same strongly connected component (keep in mind the lemma). Since all nodes n with $w \prec n$ are removed from C, the invariants will also hold here.

    - If e is a cross edge with $w \in S_V$ (the end node is on the stack that contains the vertices of the currently open components), we also found a directed cycle (it is important that $w \in S_V$ because otherwise we would have a directed path from v a **closed** component, which in turn means that there is no path from this component to v, therefore there is no directed cycle) and all the nodes in the cycle belong to the same strongly connected component. Again, since all nodes n with $w \prec n$ are removed from C, the invariants hold.
    - If e is a forward edge, the algorithm ignores it. This is justified because a forward edge is just a "shortcut" in an already discovered directed path, it gives no new information on the possible existence of directed cycles.
  - Case2: backtrack(w, e, v):
    - Node w becomes closed. For C, w is the first node of the open component with the highest index if and only if it is on the top of C (from the invariant). Forward edges cannot make a strongly connected component grow, as already mentioned. Cross edges that we may find later also cannot make this strongly connected component grow (because this means, since we didn't already find that cross edge before coming to the backtrack part, that we have a "one-way" directed path, therefore no cycle), we can safely close the current component. For S_V containing the nodes of the component, the repeat-loop deletes the correct nodes from the stack. Hence, keeping the invariants satisfied.

For 2-edge connected components in undirected graphs:

Using similar argument to the one stated in the lecture for proving the correctness of the algorithm for identifying the 2-connected components:

- As a reminder, stack S_v is used for vertices in open components and stack C for open components (each represented by its first vertex).
- Using induction over the calls of traverse and backtrack. We call a marked node open if there has been no backtracking over it, all other nodes are closed. For every 2-edge connected component, there is a unique first node, and a component is open/closed if the first node is open/closed.
- After the first t calls of traverse and backtrack, let G_t be the subgraph induced by marked nodes. Denote open components, their node sets, and their first nodes by $G_t^{(i)}$, $V_t^{(i)}$, and $v_t^{(i)}$, for $1 \leq i \leq k_t$, in the order in which $v_t^{(i)}$ were marked. We use the following invariants:
  - The nodes of a closed component point to their first node.
  - On C we have $v_t^{(1)} \prec \ldots \prec v_t^{(k_t)}$ in this order.
  - On S_V we have the nodes from $V_t^{(1)}, \ldots, V_t^{(k_t)}$ in this order.

All conditions are true in the beginning. Assume they hold until $t-1$ and consider step t in two cases:

- Case1: traverse(v, e, w):
  - If e is a tree edge, then w represents a new open component containing only w. Obviously, the invariants hold in this case.
  - If e is a back edge, then it closes a cycle by traversing tree edges up to w. All the nodes in this cycle belong to the same 2-edge connected component. Since all nodes n with $w \prec n$ are removed from C, the invariants will also hold here.
- Case2: backtrack(w, e, v):
  - Node w becomes closed. For C, w is the first node of the open component with the highest index if and only if it is on the top of C (from invariant). As there are no cross or forward edges in undirected DFS, the component can't grow and is therefore closed. For S_v containing the nodes of the component, the repeat-loop deletes the correct nodes from the stack. Hence, keeping the invariants satisfied.

Problem 2

Point a)

We will proceed by induction on the number of nodes n. This proof and the algorithm on 2b) will both assign numbers sequentially, which means π is a permutation of $\{1, \ldots, n\}$.

Base case: For $n = 1$, we just set $\pi(v) = 1$ for the only vertex.

To get the induction step $(n - 1 \to n)$, we will simply pick one vertex v with no incoming arcs. Such a vertex must exist since the G is a DAG.

Now, we define G' as the graph induced by deleting v from G. Since G' is a DAG with $n - 1$ nodes, we now apply the induction hypothesis, and therefore we get some $\pi : V \setminus \{v\} \to \mathbb{N}$. We extend this function to V by setting

$$\pi(v) = 1 + \max_{u \in V \setminus \{v\}} \pi(u)$$

We remark that by induction hypothesis all the values are in the range $\{1, \ldots, n-1\}$ for G'. Therefore, $\pi(v) = n$.

Now, we simply need to check that the constraint $\pi(u) > \pi(w)$ for $(u, w) \in E$. There are two types of edges in E:

- If $u = v$, then by definition of $\pi(v)$, $\pi(v) > \pi(w)$
- Otherwise, by induction hypothesis, since $u, w \in V \setminus \{v\}$, then $\pi(u) > \pi(w)$.

```
1  Procedure: topological_order(G)
2  Let S be a stack, L be a (linked) list, P an array of (parent) nodes;
3  C := 1;
4  while ∃ v unvisited do
5      Push(S, v);
6      while S is not empty do
7          u ← Top(S);
8          if u is unvisited then
9              mark u as visited;
10             foreach (u, w) ∈ E do
11                 if w is unvisited then
12                     P[w] ← u;
13                     Push(S, w);
14                 else
15                     if π(w) is undefined then
16                         // That means that w is on the stack, and we have a cycle
17                         while u ≠ w do
18                             InsertHead(L, u);
19                             u ← P[u];
20                         end
21                         return L;
22                     end
23                 end
24             end
25         else
26             Pop(S);
27             if π(u) is undefined then
28                 π(u) ← C;
29                 C ← C + 1;
30             end
31         end
32     end
33 end
34 return π
```

```
1  Procedure: topological_order(G)
2  Set C := 1;
3  Use DFS framework with:
4  Procedure traverse(v, e, w)
5      if w is visited and π(w) is undefined then
6          // That means that w is on the stack, and we have a cycle
7          while u ≠ w do
8              InsertHead(L, u);
9              u ← incoming[u];
10         end
11         // The following return statement should stop the algorithm completely, since it
           returns a cycle.
12         return L;
13     end
14 Procedure backtrack(w, p, t)
15     if π(w) is undefined then
16         π(w) ← C;
17         C ← C + 1;
18     end
```

Point b)

We can see the algorithm for this exercise in Figures 34 and DFS Algorithm for Topological Order (using the framework). Both versions are equivalent.

We remark that the `foreach` cycle on line 10 only runs once for each vertex, since the vertex is marked as visited immediately before, and this code only runs if the vertex is unvisited. Therefore, each edge is only considered once. Moreover, the number of `Push` operations is at most $m + n$, since one such operation is done for each vertex, on line 5, and for each edge, on line 13.

We conclude that the code inside the cycle on line 6 runs at most $n + (m + n)$ times, with at most n of these times into the first branch (line 8) and $m + n$ times into the second branch (line 25). We already saw that the multiple executions of the `foreach` cycle on line 10 take $O(m)$ time. The second branch (line 25) runs in constant time, so over the $m + n$ times it is run it takes $O(m + n)$ time. Therefore, the terminates in $O(m + n)$ time.

One other important point is that $\pi(v)$ is defined for all $v \in V$, since every unvisited vertex is eventually added to the stack S, and the only `Pop` operation is followed by setting $\pi(v)$ if it is undefined.

We will now argue that the mapping π produced by the algorithm is correct. (that is, it satisfies $\pi(u) - \pi(v)$, for $(u, v) \in E$). Due to the way DFS works, the algorithm tries to traverse as much as possible, backtracking only if it cannot advance anymore. In particular, the backtrack operation for a vertex v is only executed after all its children are visited. Therefore, for a child

u of v, the backtracking operation for u runs first than the backtracking operation for u, and thus $\pi(u)$ is set before $\pi(v)$. Since π is set using the global counter C, that increases each time some π is set from some node, then it must be that $\pi(v) > \pi(u)$.

Finally, let us look at the case in which the graph is not a DAG. If at some point one of the children w of a node u is already visited, but $\pi(w)$ is not yet defined, this means that w was already inserted in the stack, but backtrack for it has not been executed yet, and therefore the node is still on the stack. We conclude that the edge (u, w) is a backedge and that there is a cycle consisting of the path from w to u and the edge (u, w).

Problem 3

a) **Show that G has an Euler tour $\leftrightarrow$ in-degree(v) = out-degree(v) for each vertex $v \in V$**

"$\rightarrow$"

We will call a cycle simple if it visits each vertex no more than once, and complex if can visit a vertex more than once. We know that each vertex in a simple cycle in-degree and out-degree one, and any complex cycles can be expressed as a union of simple cycles. This implies that any vertex in a complex cycle (and in particular an Euler tour) has in-degree equal to its out-degree. Thus, if a graph has an Euler tour than all of its vertices have equal in- and out-degrees.

"$\leftarrow$"

Suppose we have a connected graph for which the in-degree and out-degree of all vertices are equal. Let C be the longest complex cycle within G. If C is not an Euler tour, then there is a vertex v of G touched by C such that not all edges in and out v of are exhausted by C. We may construct a cycle C' in $G - C$ starting and ending at v by performing a walk in $G - C$. (The reason is that $G - C$ also has a property that in-degrees and out-degrees are equal.) this simply means that the complex cycle that starts at v goes along the edges of C' (returning to v) and then goes along the edges of C is a longer complex cycle than C. This contradicts our choice of C as the longest complex cycle which means that C must have been an Euler tour.

b) **Describe an $O(m)$-time algorithm to find an Euler tour of G if one exists.**

ALGORITHM

Given a starting vertex , the v_0 algorithm will first find a cycle C starting and ending at v_0 such that C contains all edges going into and out of v_0. This can be performed by a walk in the graph. As we discover vertices in cycle C, we will create a linked list which contains vertices in order and such that the list begins and ends in vertex v_0. We set the current pointer to the head of the list. We now traverse the list by moving our pointer "current" to successive vertices

until we and a vertex which has an outgoing edge which has not been discovered. (If we reach the end of the list, then we have already found the Euler tour). Suppose we find the vertex, v_i, that has an undiscovered outgoing edge. We then take a walk beginning and ending at v_i such that all undiscovered edges containing v_i are contained in the walk. We insert our new linked list into old linked list in place of v_i and more "current" to the new neighbor pointed to the first node containing v_i. We continue this process until we search the final node of the linked list, and the list will then contains an Euler tour.

Running Time: The algorithm traverse each edge at most twice, first in a walk and second while traversing the list to find vertices with outgoing edges. Therefore, the total running time of the algorithm is $O(|E|)$.

Reference: http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/GraphAlgor/eulerTour.h