Exercise Sheet 4

Issue Date: November 14th, 2023
Due Date: November 20th, 2023 – 10:00 a.m.
$\sum$ 10 Points

University of Konstanz
Dr. Barbara Pampel
Sabrina Jaeger-Honz

Konzepte der Informatik INF-11700
Winter 2023/2024

Graph traversal, Iteration, Recursion

Exercise 1: Graphs (*5 points*)

The choice of the next unmarked edge to traverse in the depth-first search (Algorithmus *Tiefensuche*) („**if** es ex. unmarkierte Kante $\{v, w\} \in E$“) and breadth-first search (Algorithmus *Breitensuche*) is not deterministic („ **for** unmarkierte Kanten $\{v, w\} \in E$ “).
The DFS or BFS makes no assumption about whether adjacent vertices of a vertex are returned ordered or not. Hence, the neighborhood of a vertex might be unordered. This non-determinism makes the whole algorithm non-deterministic.

For this exercise this behavior is undesired, which is why we need a deterministic choice for the next edge to traverse coming from a vertex v. In order to check if an unmarked edge $\{v, w\}$ exists, we have to retrieve the corresponding w. Such a w is returned by the procedure *next*(v).

A deterministic (though not efficient) selection of the next neighbor to explore is shown in Algorithm 1. It returns the w with the lowest index that is reachable over an unmarked edge $\{v, w\}$ from the vertex v; if there is no such w, then a dummy $v_{+\infty}$ is returned which signals that there does not exist any unmarked edge $\{v, w\}$.

Algorithm 1 *next*(v)

```
Input: vertex $v$
Result: vertex $w$ to visit, $v_{+\infty}$ if no such vertex remains
procedure next(v)
    $w = v_{+\infty}$        ▷ initialize placeholder s.t. any other vertex index will be lower
    for $u \in N_G(v)$ do        ▷ iterate over all vertices adjacent to v (neighborhood $N_G(v)$)
        ▷ index( ) retrieves index values in subscripts
        if $\{v, u\}$ not yet marked $\wedge$ $index(u) < index(w)$
        then
            $w = u$
    return $w$
```

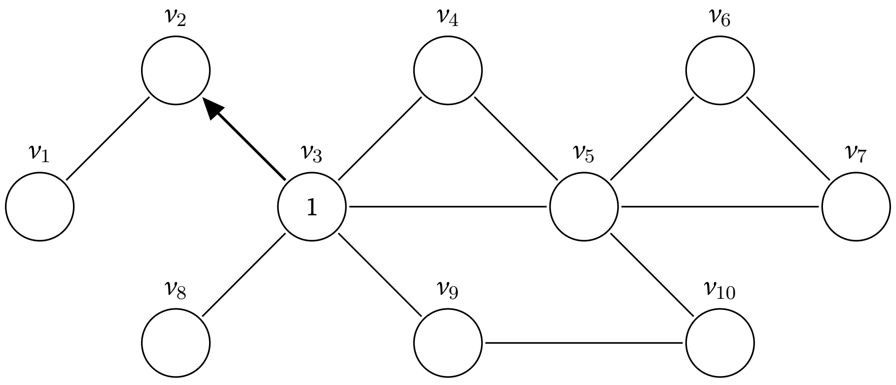
Using *next*(v), the DFS algorithm from the lecture can then be modified:

```
if $(w = next(v)) \neq v_{+\infty}$
then ▷ we assign w and afterwards check it for $v_{+\infty}$
```

a) (3 points) Execute the **modified depth-first search** (using the *next* procedure) in the undirected graph $G_2 = (V, E)$ depicted below, where $s = v_3$ (starting at v_3).

Show corresponding DFS numbers on vertices. Mark edge directions representing the *parent* relationship using an arrow to the child (these are *tree edges*) and edges that lead to already marked (numbered) vertices with a dashed line (these are *back edges*).

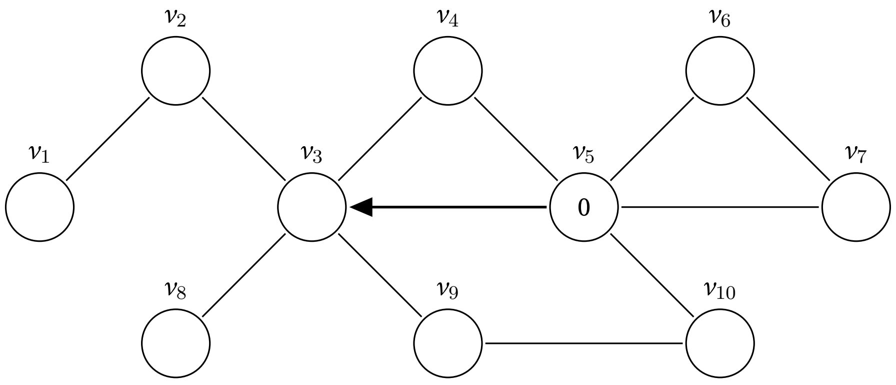
Since the starting vertex is v_3, it is assigned the DFS number 1. According to *next*( ), the first edge to traverse is $\{v_3, v_2\}$.



b) (2 points) Execute the **modified breadth-first search** (using the *next* procedure) in the undirected graph $G_2 = (V, E)$ depicted below, where $s = v_5$ (starting at v_5). Identical to the modification made for the DFS, at each vertex v, choose the next edge $e = \{v, w\}$ to traverse, such that e is unmarked and w has the lowest index (written in subscript) of all vertices connected (*i.e.*, adjacent) to v by an unmarked edge.

Show corresponding BFS numbers on vertices. Mark edge directions representing the *parent* relationship using an arrow to the child (these are *tree edges*) and edges that lead to already marked (numbered) vertices with a dashed line (these are *back edges*).

Since the starting vertex is v_5, it is assigned the BFS number 0.



Exercise 2: Iteration (*5 points*)

a) (2 points) Sort the array $A := [I, N, S, E, R, T]$ lexicographically using *SelectionSort*. Visualize the full array **at the end** of every run of the **outer** for-loop (analogous to the lecture slides).

b) (1 point) Please provide an array (maximum length = 4) to show that *SelectionSort* is an unstable sorting algorithm.

c) (2 points) In the lecture, InsertionSort was introduced on lists. The following algorithm is more suitable for arrays, because it swaps two items instead of moving whole sequences and while the algorithm from the lecture searches for the right place for inserting the item from left to right (along the linking-direction of the list), the algorithms for arrays works form right to left in the inner loop.

Algorithm 2 InsertionSort

```
Input: array A containing comparable items
Result: non-descendingly sorted array A
for $i = 2, \ldots, n$ do
    $j \leftarrow i$
    while $j > 1 \land A[j] < A[j-1]$ do
        swap $A[j]$ and $A[j-1]$
        $j \leftarrow j - 1$
```

Sort the array $A := [S, E, L, E, C, T]$ lexicographically using the variant of *InsertionSort* for arrays. Visualize the full array **at the end** of every run of the **inner** while-loop.