Algorithm Theory - Theoretical exercise 5

Téo Bouvard

March 22, 2020

Problem 1

Since we deal with a directed acyclic graph, we can use the acyclic property to count the number of paths from a source node to a target node recursively. If use memoization, we can avoid solving overlapping subproblems. However, we can also solve this problem by using a bottom-up approach, using the same idea we used for computing the Fibonacci sequence in the previous assignment. We start at the target node with a number of paths equal to one, and add the node's counter to its parents while adding them to FIFO queue if they have not already been visited. If we do that until the queue is empty, we have the number of paths from every node in the graph to the target, while still having a linear complexity. An example implementation is given below. For simplicity, we assume that we have access to a FIFO data structure, and that the graph data structure can retrieve the predecessors of a node. Furthermore, we assume that nodes are objects with an *already_visited* attribute. If that was not the case, we could still maintain a separate list of already visited nodes.

```
N-PATHS(G, s, t)
    let n[0...length(G.V) − 1] be a zero-initialized array
    queue = FIFO()
    queue.enqueue(t)
    n[t] = 1
    while queue.empty() = FALSE
        current = queue.dequeue()
        pred = G.predecessors(current)
        for p ∈ pred
            if p.already_visited = FALSE
                queue.enqueue(p)
                p.already_visited = TRUE
            n[p] += n[current]
    return n[s]
```

This algorithm has complexity $\mathcal{O}(V + E)$ because it considers each node and its predecessors at most once. In fact it is nearly identical to a BFS, except that its goal is to count paths rather than searching for a particular node.

Problem 2

a)