CMPSC 465 — Data Structures & Algorithms
Fall 2024 — Mingfu Shao and Debarati Das

# Recitation 4

Sep 30, 2024

1. (Pouring Water)

   We have three containers whose sizes are 10 pints, 7 pints, and 4 pints, respectively. The 7-pint and 4-pint containers start out full of water, but the 10-pint container is initially empty. We are allowed one type of operation: pouring the contents of one container into another, stopping only when the source container is empty or the destination container is full. We want to know if there is a sequence of pourings that leaves exactly 2 pints in the 7- or 4-pint container.

   1. Model this as a graph problem: give a precise definition of the graph involved and state the specific question about this graph that needs to be answered.
   2. What algorithm should be applied to solve the problem?

   Solution:

   1. Let $G = (V, E)$ be our (directed) graph. We will model the set of nodes as triples of numbers $(a_0, a_1, a_2)$ where the following relationships hold: Let $S_0 = 10, S_1 = 7, S_2 = 4$ be the sizes of the corresponding containers. $a_i$ will correspond to the actual contents of the $i$-th container. It must hold $0 \leq a_i \leq S_i$ for $i = 0, 1, 2$ and at any given node $a_0 + a_1 + a_2 = 11$ (the total amount of water we started from). An edge between two nodes $(a_0, a_1, a_2)$ and $(b_0, b_1, b_2)$ exists if both the following are satisfied:
      - the two nodes differ in exactly two coordinates (and the third one is the same in both).
      - if $i, j$ are the coordinates they differ in, then either $a_i = 0$ or $a_j = 0$ or $a_i = S_i$ or $a_j = S_j$.

      The question that needs to be answered is whether there exists a path between the nodes $(0, 7, 4)$ and $(*, 2, *)$ or $(*, *, 2)$ where $*$ stands for any (allowed) value of the corresponding coordinate.
   2. We can apply DFS on this graph, starting from node $(0, 7, 4)$ with an extra line of code that halts and answers 'YES' if one of the desired nodes is reached and 'NO' if all the connected component of the starting node is exhausted and no desired vertex is reached.

2. (Uniqueness of Linearization)

   Let $G = (V, E)$ be a DAG. Design an $O(|V| + |E|)$ time algorithm to decide if there is only one possible linearization for $G$. Prove that your algorithm is correct.

   Solution. Let $n = |V|$. Let $X[1 \cdots n]$ be a linearization of DAG $G$ (recall that $X$ is an ordering of all vertices and we use $X[i]$ to represent the $i$-th vertex in $X$). Then $G$ has a unique linearization (i.e., $X$) if and only if there is a directed edge between each pair of consecutive vertices in $X$, i.e., $(X[i], X[i+1]) \in E$ for any $1 \leq i \leq n-1$. Let's prove this statement. We first prove that if $(X[i], X[i+1]) \in E$, $1 \leq i \leq n-1$, then $X$ is the unique linearization of $G$. By definition of linearization, the existence of edge $(X[i], X[i+1])$ forces that vertex $X[i]$ is before vertex $X[i+1]$ in any linearization, $1 \leq i \leq n-1$. Hence, there is only one possible linearization that satisfies all these ordering constraints, which is $X$. Now we prove the other side: suppose that there exists $(X[k], X[k+1]) \notin E$ for some $1 \leq k \leq n-1$, then the linearization of $G$ is not unique. This is because we can construct another linearization of $G$ by swapping $X[k+1]$ and $X[k]$ in $X$ to obtain $X'$, i.e., $X' = (X[1], X[2], \cdots, X[k-1], X[k+1], X[k], X[k+2], \cdots, X[n])$. The reason why $X'$ is a linearization of $G$ is implied by the fact that $X$ is a linearization of $G$. Specifically, let $(u, v) \in E$; then $u$ is before $v$ in $X'$, as $u = X[k]$ and $v = X[k+1]$ cannot be true at the same time, and that $u$ is before $v$ in $X$.

The above statement immediately suggests an algorithm to decide whether a DAG has a unique linearization: We can use DFS to get one linearization, which takes $O(|E| + |V|)$ time. We then check every consecutive vertices $u$ and $v$ in the linearization whether $(u, v) \in E$. This again can be done in $O(|E| + |V|)$ time, as we only need to examine the adjacent edges for each vertex once (suppose that we use the adjacency list representation). The correctness of this algorithm is guaranteed by the correctness of above statement.