Algorithms and Datastructures (HS2024)
Week 9

Rui Zhang

November 16, 2024

Contents

1 Revision Theory

1.1 Graphs in Programming

When dealing with graphs in practice, we have multiple options of saving them in memory. One of them is an adjacency matrix and one of them is the adjacency list.

Adjacency Matrix: For a graph $G = (V, E)$ with $|V| = n$ we can choose refer to the vertices each by a unique number from $1, \ldots, n$. Then, we choose to save an $n \times n$ matrix A, in which $A_{u,v} = 1 \Leftrightarrow$ there exists an edge from u to v. Note that in undirected graphs, that means that A would be symmetric.

Adjacency List: For the same graph definition as above and the same names for the nodes, we choose to save an array Adj of linked lists. Then, $Adj[u]$ for $u \in V$ is a linked list containing all successors of u.

Comparison: There are some operations often considered in graph algorithms. Let us compare the asymptotic complexities of the two options for these operations:

Operation	Runtime Matrix	Runtime List
Check if $(u, v) \in E$ (or $\{u, v\} \in E$)	$O(1)$	$O(1 + deg_{out}(u))$
Iterate through all successors of u	$O(n)$	$O(1 + deg_{out}(u))$

Here, the reason for the +1 in the runtime for the lists is because the degree could be zero. In which case we would not want to have $O(0)$ though (note that this O-notation means literally zero operations, unlike $O(1)$, which means any constant number of operations - why?).

1.2 Depth First Search

We look at this algorithm in the context of directed graphs.

1.2.1 Intuitive Description

Depth-First-Search (DFS), is one of the most fundamental algorithms of graph theory. Think of it as an algorithm which traverses an entire connected component starting at some chosen vertex v in a fashion which goes "depth first", as the name suggests (exploring deep first, then reaching a dead-end and only then going back). Another strategy is "breadth first" (BFS, next week), which explores "layer by layer".

Starting at some vertex u, depth first search starts with the first successor v of u and explores v. Then, it repeats the same procedure with v and so on. At each point, DFS marks the explored vertex as "explored". Once DFS reaches a point where all successors are already marked, it backtracks to the last point where there was a successor that was not marked and continues from there. This algorithm can be realized most simply via recursion.

1.2.2 Runtime Analysis

Yes, dfs is recursive. Now, after so many exercises consisting of modelling the runtime of a recursive algorithm as a recursive runtime function, we are going to do an absolute 180 and think of the runtime of dfs from another perspective.

Since we mark each vertex after having explored them, we will never go though the successors of any vertex $v \in V$ more than just once. In other words, if apply dfs to all connected components of the graph, then we will have traversed every vertex **exactly once** and went through all successors of each vertex exactly once. That is a total runtime of

$$\sum_{v\in V} O(deg_{out}(v)+1) = O(\sum_{v\in V} deg_{out}(v) + \sum_{v\in V} 1) \quad (1)$$

$$= O(m+n) \quad (2)$$

if we store the graph in an adjacency list. Where the second equality follows from handshake lemma.

1.3 DFS-Tree and Tree, Back, Forward, Cross-Edges

To visualize how DFS operates, we may draw a DFS-tree. The root of this tree is the vertex at which we started DFS. The children of this root are all successors of the first vertex, which were not marked at the time of exploration. Based on such a tree, we also see from which vertices we reached which other vertices.

Notice: Not all edges were drawn into this tree. Take for example the case where the DFS algorithm skipped a successor v of a vertex u because v was already marked / visited. Then, there exists an edge (u, v) in the original graph, but not in the DFS-Tree.

The edges that are present in the DFS-Tree must be present in the original graph however, so we categorize all of the edges in the DFS-Tree as **"Tree Edges"**. For all the rest we have the following categories:

The edges in the original graph that go from a top layer to a lower layer in the same subtree in the DFS-Tree are called **forward-edges**.

The edges in the original graph that go from a bottom layer to an higher layer in same subtree of the DFS-Tree, are called **back-edges**.

The edges in the original graph that go from one subtree to another subtree in a DFS-Tree are called **cross-edges**.

Notice: As the existence of a back edge is equivalent to the existance of a directed cycle. Can you explain this on an intuivive level.

1.4 Topological Sorting

We have talked about sorting in arrays. Now, we will sort graphs (say whaatt??)

The graph is topologically sorted, if we have arranged the vertices of the graph from left to right such that the edges always go from left to right.

Intuitively, if we have a directed cycle in a graph, then topological sortings should be hard, since no matter how we position the nodes of the cycle, there will always be an edge going in the wrong direction. And yes, indeed it is impossible to have a topological sorting if we have a cycle (or equivalently, a back edge, as we have seen in the last section).

1.4.1 Pre, Post-Order

If we track each step of the DFS algorithm and assign each step a number representing time (and then increment the time), then we will be able to gain more insight into how DFS works and maybe also solve a problem along the way.

Each time we visit a vertex v, we have this loop where we go over all of its successors and recursively traverse them if not already marked. Before we go through this loop, we assign this vertex v a "**pre-number**". This number represents the first time-point at which we traversed v. After traversing all successors of v, we will return to v. Now, we assign to v a "**post-number**". It represents the last time-point we will ever come back to v (sadge).

Now, we may order the vertices by their post-orders, then we will have a backwards topological sorting as long as there exists no directed cycles / back-edges in the graph.

2 Exercise Sheet

2.1 Exercise Sheet 9 - Priority List

1. - 9.2) very similar to exam exercises
2. - 9.4) DP is always good
3. - 9.3) It is important to know the runtimes differences of the two datastructures for graphs
4. - 9.5) Is still a bonus exercise, although in my opinion not as exam relevant as other exercises (don't quote this)
5. - 9.1)

2.2 Exercise Sheet 7 - Feedback

- Be more precise in subproblem definition. For the subset sum with duplicates subproblem, many of you failed to mention that it is a subset sum **potentially with duplicates**. If you feel comfortable, I would even rather recommend a mathematical definition of the subproblem as in the solution.
- The dimensions of the DP-Table should also be more precise. If one axis starts at 0 and ends at n, then the size of that axis is $(n + 1)$.
- Be sure to mention that the calculation order must be ascending / descending. It is a better descriptor than "from left to right / from top to bottom / etc."

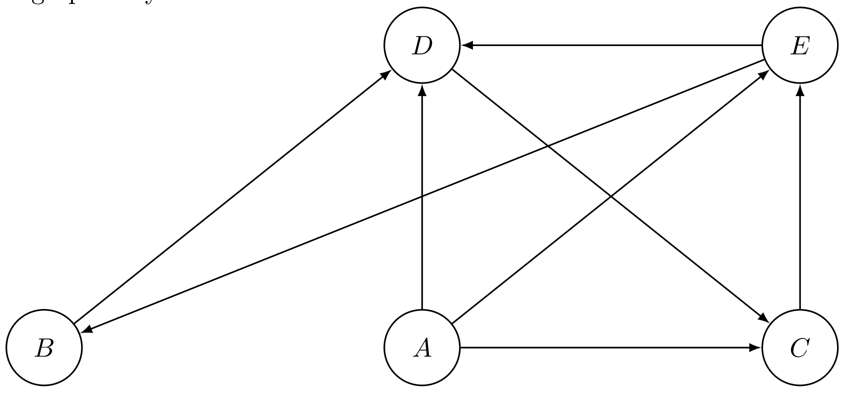
3 Supplementary Exercises

3.1 Depth First Search

Without looking at the lecture notes, write the pseudocode for DFS.

3.2 Depth First Search / Topological Sorting

For the following graph, perform depth first search starting at vertex A and note the pre- and post- numbers. While performing dfs, start with vertices with lexicographically lower names.



3.3 Cross Edges

Explain why cross edges can only go from a higher layer to a lower layer in the dfs-tree.

3.4 HS15 Exercise 1e)

`https://exams.vis.ethz.ch/exams/7l4qz2nu.pdf`

3.5 Breadth First Search

Think about the "layer by layer" description of the Breadth-First-Search as above. What could this mean in terms of an actual algorithm? How would you implement this algorithm in pseudocode.