Eidgenössische Technische Hochschule Zürich
Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Federal Institute of Technology at Zurich

Departement of Computer Science
Markus Püschel, David Steurer
François Hublet, Goran Zuzic, Tommaso d'Orsi, Jingqiu Ding

28 November 2022

Algorithms & Data Structures Exercise sheet 10 HS 22

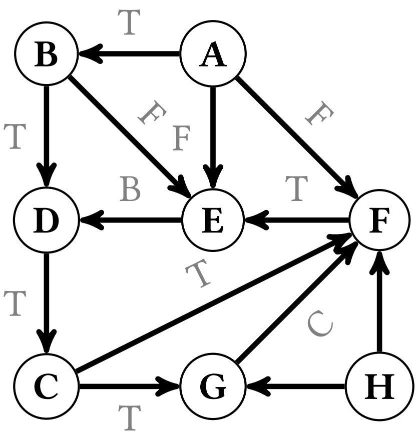
The solutions for this sheet are submitted at the beginning of the exercise class on 5 December 2022.

Exercises that are marked by * are "challenge exercises". They do not count towards bonus points.

You can use results from previous parts without solving those parts.

Exercise 10.1 *Depth-First Search* **(1 point)**.

Execute a depth-first search (*Tiefensuche*) on the following graph starting from vertex A. Use the algorithm presented in the lecture. When processing the neighbors of a vertex, process them in alphabetical order.



(a) Mark the edges that belong to the depth-first tree (*Tiefensuchbaum*) with a "T" (for tree edge).

(b) For each vertex in the depth-first tree, give its *pre*- and *post*-number.

Solution:

A(1,14) B(2,13) C(4,11) D(3,12) E(6,7) F(5,8) G(9,10).

(c) Give the vertex ordering that results from sorting the vertices by pre-number. Give the vertex ordering that results from sorting the vertices by post-number.

Solution:

Pre-ordering: A, B, D, C, F, E, G.

Post-ordering: E, F, G, C, D, B, A.

(d) Mark every forward edge (*Vorwärtskante*) with an "F", every backward edge (*Rückwärtskante*) with an "B", and every cross edge (*Querkante*) with a "C".

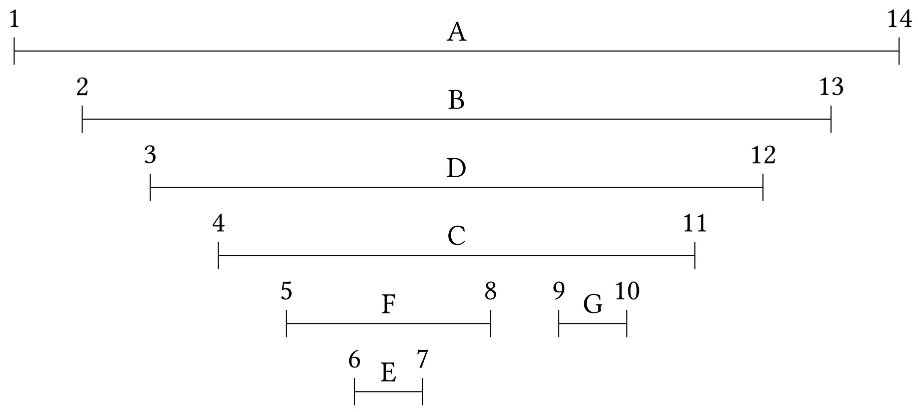
(e) Does the above graph have a topological ordering? How can we use the above execution of depth-first search to find a directed cycle?

Solution:

The decreasing order of the post-numbers gives a topological ordering whenever the graph is acyclic. This is the case if and only if there are no back edges. If there is a back edge, then together with the tree edges between its end points it forms a directed cycle. In our graph, the only back edge is E $\to$ D, and the tree edges from D to E are D $\to$ C, C $\to$ F, and F $\to$ E. Together they form the directed cycle (D $\to$ C $\to$ F $\to$ E $\to$ D).

(f) Draw a scale from 1 to 16, and mark for every vertex v the interval I_v from pre-number to post-number of v. What does it mean if $I_u \subset I_v$ for two different vertices u and v?

Solution:



If $I_u \subset I_v$ for two different vertices u and v, then u is visited during the call of DFS-Visit(v).

(g) Consider the graph above where the edge from E to D is removed and an edge from A to H is added. How does the execution of depth-first search change? Which topological sorting does the depth-first search give? If you sort the vertices by *pre-number*, does this give a topological sorting?

Solution:

The execution of depth-first search only changes in the last step, where H is visited (from A).

The topological sorting (reversed post-ordering) is: A, H, B, D, C, G, F, E.

The pre-ordering is A, B, D, C, F, E, G, H; it does not give a topological ordering, since there is an edge (G, F) in the graph.

Exercise 10.2 *Longest path in DAGs* **(1 point)**.

Given a directed graph $G = (V, E)$ without directed cycles (i.e., a DAG), the goal is to find the number of edges on the **longest path** in G.

Describe a dynamic-programming algorithm that, given G, returns the length of the longest path in G in $O(|V| + |E|)$ time. You can assume that $V = \{1, 2, \dots, n\}$, and that the graph is provided to you as a pair (n, Adj) of an integer $n = |V|$ and an adjacency list Adj. Your algorithm can access $Adj[u]$, which is a list of vertices to which u has a direct edge, in constant time. Formally, $Adj[u] := \{v \in V \mid (u, v) \in E\}$.

Example: $n = 5$

$$\begin{tikzcd}
1 \arrow[r, red] \arrow[d] & 2 \arrow[ddl] \arrow[dddl, red] \\
3 \arrow[d] & \\
4 & \\
5 \arrow[u, red] &
\end{tikzcd}$$

Output: 3

(the path is highlighted in red.)

In your solution, address the following aspects:

1. *Dimensions of the DP table*: What are the dimensions of the DP table?
2. *Definition of the DP table*: What is the meaning of each entry?
3. *Computation of an entry*: How can an entry be computed from the values of other entries? Specify the base cases, i.e., the entries that do not depend on others.
4. *Calculation order*: In which order can entries be computed so that values needed for each entry have been determined in previous steps?
5. *Extracting the solution*: How can the solution be extracted once the table has been filled?
6. *Running time*: What is the running time of your solution?

Solution:

For convenience, in the solution we will use the standard notation $n := |V|, m := |E|$.

1. *Dimensions of the DP table*: $DP[1 \ldots n]$, one for each node.
2. *Definition of the DP table*: $DP[u]$ is the length of the longest path starting at u.
3. *Computation of an entry*: If u has no outgoing neighbor, we set $DP[u] = 0$. Otherwise, $DP[u] = \max_{v \in Adj[u]} (1 + DP[v])$.
4. *Calculation order*: We compute the entries in reverse topological order (i.e., the first node is one with no outgoing edges and the last is one with no incoming edges). **Remark.** This is the crucial part that differentiates this dynamic program from all prior ones.
5. *Extracting the solution*: We return $\max_{u \in V} DP[u]$.
6. *Running time*: We compute the entry for each node, contributing $O(n)$ to the runtime. Furthermore, each node u processes all outgoing edges from v, but every edge gets processed only once. Hence, this contributes $O(|E|)$ to the runtime. In total, we get $O(|V| + |E|)$.

Exercise 10.3 *Subtree sum* **(1 point)**.

Definition 1. Given a directed graph $\vec{G} = (V, \vec{E})$ we define its **undirected version** as $\overleftrightarrow{G} = (V, \overleftrightarrow{E})$ with each directed edge $u \to v$ being transformed to an undirected $u \leftrightarrow v$. Formally, $\overleftrightarrow{E} := (\{u, v\} \mid (u, v) \in \vec{E}\}$.

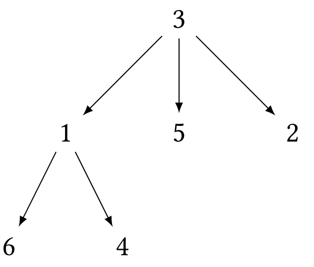
Definition 2. A directed graph $G = (V, E)$ is a **tree rooted at** $r \in V$ if G's undirected version $\overleftrightarrow{G}$ is an undirected tree (see Exercise 9.5 for a definition) and every node is reachable from r via a directed path.

Write the pseudocode of an algorithm that, given a rooted tree $G = (V, E)$, computes, for each vertex v, the total number of vertices that are reachable from v (via directed paths). The algorithm should have a runtime of $O(|V| + |E|)$. You an assume $V = \{1, 2, \dots, n\}$. The graph will be given to the algorithm as access to n, the root $r \in V$, and an adjacency list. Namely, the algorithm can access $Adj[u]$, which is a list of vertices to which u has a direct edge. Formally, $Adj[u] := \{v \in V \mid (u, v) \in E\}$.

Explain in a few sentences why your algorithm achieves the desired runtime.

Hint: *If needed, you can use the fact that "in G, there is a unique path from the root to each vertex" without proof.*

Example:
$n = 6$
$r = 3$



Output: [3, 1, 6, 1, 1, 1].

Solution:

Algorithm 1

```
1:  Input: integers $n, r$. Adjacency list $Adj[1 \dots n]$.
2:
3:  Let $reachable[1 \dots n]$ be a global variable, initialized to 0.
4:
5:  function $walk(u)$                          ▷ Calculate the number of reachable vertices from $u$.
6:      for each $v$ in $Adj[u]$ do                                  ▷ Iterate over all children $v$.
7:          $walk(v)$
8:          $reachable[u] \leftarrow reachable[u] + reachable[v]$
9:      $reachable[u] \leftarrow reachable[u] + 1$                          ▷ $u$ can reach itself.
10:
11: $walk(r)$                                                    ▷ Start walking from the root.
12: Print($reachable$)
```

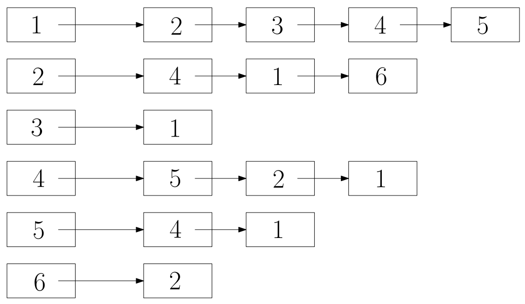
There is a unique path from the root to each vertex (easy to prove). Therefore, the function will be called on each vertex at most once (contributing $O(|V|)$). Each vertex processes all outgoing edges, hence each edge will be processed at most once (contributing $O(|E|)$). In total, we get runtime $O(|V| + |E|)$.

Exercise 10.4 *Data structures for graphs.*

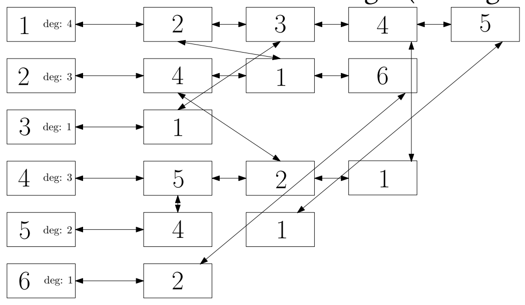
Consider three types of data structures for storing a graph G with n vertices and m edges:

a) Adjacency matrix.

b) Adjacency lists:



c) Adjacency lists, and additionally we store the degree of each node, and there are pointers between the two occurences of each edge. (An edge appears in the adjacency list of each endpoint).



For each of the above data structures, what is the required memory (in Θ-Notation)?

Solution:

$\Theta(n^2)$ for adjacency matrix, $\Theta(n + m)$ for adjacency list and improved adjacency list.

Which runtime (worst case, in Θ-Notation) do we have for the following queries? Give your answer depending on n, m, and/or $\deg(u)$ and $\deg(v)$ (if applicable).

(a) Input: A vertex $v \in V$. Find $\deg(v)$.

Solution:

$\Theta(n)$ in adjacency matrix, $\Theta(1 + \deg(v))$ in adjacency list, $\Theta(1)$ in improved adjacency list.

(b) Input: A vertex $v \in V$. Find a neighbour of v (if a neighbour exists).

Solution:

$\Theta(n)$ in adjacency matrix, $\Theta(1)$ in adjacency list and in improved adjacency list.

(c) Input: Two vertices $u, v \in V$. Decide whether u and v are adjacent.

Solution:

$\Theta(1)$ in adjacency matrix, $\Theta(1+\min\{\deg(v), \deg(u)\})$ in adjacency list and in improved adjacency list.

(d) Input: Two adjacent vertices $u, v \in V$. Delete the edge $e = \{u, v\}$ from the graph.

Solution:

$\Theta(1)$ in adjacency matrix, $\Theta(\deg(v) + \deg(u))$ in adjacency list and $\Theta(\min\{\deg(v), \deg(u)\})$ in improved adjacency list.

(e) Input: A vertex $u \in V$. Find a neighbor $v \in V$ of u and delete the edge $\{u, v\}$ from the graph.

Solution:

$\Theta(n)$ in the adjacency matrix ($\Theta(n)$ for finding a neighbor and $\Theta(1)$ for the edge deletion).

$\Theta(1+\max_{w:\{u,w\}\in E} \deg(w))$ for the adjacency list ($\Theta(1)$ for finding a neighbor and $\Theta(\max_{w:\{u,w\}\in E} \deg(w))$ for the edge deletion).

$\Theta(1)$ for the improved adjacency list ($\Theta(1)$ for finding a neighbor and $\Theta(1)$ for the edge deletion).

(f) Input: Two vertices $u, v \in V$ with $u \neq v$. Insert an edge $\{u, v\}$ into the graph if it does not exist yet. Otherwise do nothing.

Solution:

$\Theta(1)$ in adjacency matrix, $\Theta(1+\min\{\deg(v), \deg(u)\})$ in adjacency list and in improved adjacency list.

(g) Input: A vertex $v \in V$. Delete v and all incident edges from the graph.

Solution:

$\Theta(n^2)$ in adjacency matrix, $\Theta(n + m)$ in adjacency list and $\Theta(n)$ in improved adjacency list.

For the last two queries, describe your algorithm.

Solution:

Query (vi): We check whether the edge $\{u, v\}$ does not exist. In adjacency matrix this information is directly stored in the u-v-entry. For adjacency lists we iterate over the neighbours of u and the neighbours of v in parallel and stop either when one of the lists is traversed or when we find v among the neighbours of u or when we find u among the neighbours of v. If we didn't find this edge, we add it: in the adjacency matrix we just fill two entries with ones, in the adjacency lists we add nodes to two lists that correspond to u and v. In the improved adjacency lists, we also need to set pointers between those two nodes, and we need to increase the degree for u and v by one.

Query (vii): In the adjacency matrix we copy the complete matrix, but leave out the row and column that correspond to v. This takes time $\Theta(n^2)$. There is an alternative solution if we are allowed to *rename* vertices: In this case we can just rename the vertex n as v, and copy the n-th row and column into the v-th row and column. Then the $(n-1) \times (n-1)$ submatrix of the first $n-1$ rows and columns will be the new adjacancy matrix. Then the runtime is $\Theta(n)$. Whether it is allowed to rename vertices depends on the context. For example, this is not possible if other programs use the same graph.

In the adjacency lists we remove v from every list of neighbours of every vertex (it takes time $\Theta(n+m)$) and then we remove a list that corresponds to v from the array of lists (it takes time $\Theta(n)$). In the improved adjacency lists we iterate over the neighbours of v and for every neighbour u we remove v from the list of neighbours of u (notice that for each u we can do it in $\Theta(1)$ since we have a pointer between two occurences of $\{u, v\}$) and decrease $\deg(u)$ by one. Then we remove the list that corresponds to v from the array of lists (it takes time $\Theta(n)$).

Exercise 10.5 *Maze solver.*

You are given a maze that is described by a $n \times n$ grid of blocked and unblocked cells (see Figure **??**). There is one start cell marked with 'S' and one target cell marked with 'T'. Starting from the start cell your algorithm may traverse the maze by moving from unblocked fields to adjacent unblocked fields. The goal of this exercise is to devise an algorithm that given a maze returns the best solution (traversal from 'S' to 'T') of the maze. The best solution is the one that requires the least moves between adjacent fields.

Hint: *You may assume that there always exists at least one unblocked path from 'S' to 'T' in a maze.*

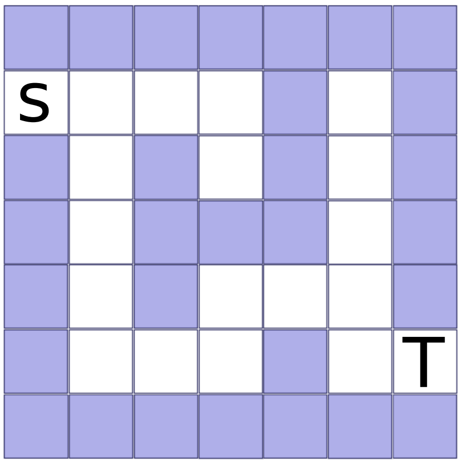


Figure 1: An example of 7×7 maze in which purple fields are blocked, white fields can be traversed (are unblocked). The start field is marked with 'S' and the target field with a 'T'.

(a) Model the problem as a graph problem. Describe the set of vertices V and the set of edges E in words. Reformulate the problem description as a graph problem on the resulting graph.

Solution:

V is the set of unblocked fields, and there is an edge between v_i and v_j if and only if v_i and v_j are adjacent unblocked fields. The corresponding graph problem is to find a shortest path between vertices 'S' and 'T' in $G = (V, E)$.

(b) Choose a data structure to represent your maze-graphs and use an algorithm discussed in the lecture to solve the problem.

Solution:

The data structure is adjacency list, the algorithm is BFS starting from 'S'. Once we know all the distances from 'S', we append vertices to a sequence starting from 'T' using the following rule: if the last appended vertex is v, we append some neighbour u of v such that $d_G(\text{'S'}, v) = d_G(\text{'S'}, u) + 1$. We stop after appending 'S'. Then we return a reverse sequence.

Hint: *If there are multiple solutions of the same quality, return any one of them.*

(c) Determine the running time and memory requirements of your algorithm in terms of n in Θ notation.

Solution:

Adjacency list requires $\Theta(|V|+|E|)$ memory, where V is a number of vertices and $|E|$ is a number of edges in the graph. BFS requires $\Theta(|V| + |E|)$ time and appending procedure also requires $\Theta(|V| + |E|)$ time, so the total running time is $\Theta(|V| + |E|)$. Since each vertex has degree at most 4, $|E| = O(|V|)$, so the running time and memory are $\Theta(|V|)$ which is $\Theta(n^2)$ in the worst case.