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
Gleb Novikov, Tommaso d'Orsi, Ulysse Schaller, Rajai Nasser

29. November 2021

Algorithms & Data Structures Exercise sheet 10 HS 21

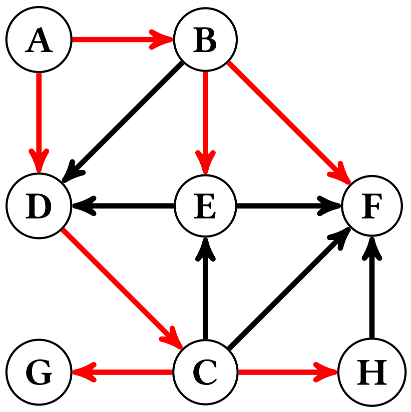
Exercise Class (Room & TA):	
Submitted by:	
Peer Feedback by:	
Points:	

Submission: On Monday, 6 December 2021, hand in your solution to your TA *before* the exercise class starts. Exercises that are marked by * are challenge exercises. They do not count towards bonus points.

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

Execute a breadth-first search (Breitensuche) on the following (directed) graph starting from vertex A. Use the algorithm presented in the lecture.

When processing the neighbors of a vertex, process them in alphabetical order.


$$\begin{tikzcd}
A \arrow[r] \arrow[d] & B \arrow[dl] \arrow[d] \arrow[dr] & \\
D \arrow[dr] & E \arrow[l] \arrow[r] & F \\
G & C \arrow[l] \arrow[u] \arrow[ur] \arrow[r] & H \arrow[u]
\end{tikzcd}$$

a) Provide the BFS order of visit of the nodes.

Solution: The BFS order of visit is: A, B, D, E, F, C, G, H

b) Provide the enter and leave time of each node.

Solution:

- A: enter=0, leave=1
- B: enter=2, leave=4
- D: enter=3, leave=7
- E: enter=5, leave=9

- F: enter=6, leave=10
- C: enter=8, leave=11
- G: enter=12, leave=14
- H: enter=13, leave=15

c) Indicate the shortest-path-tree that is obtained by BFS.

Solution: The BFS shortest-path-tree is indicated in red.

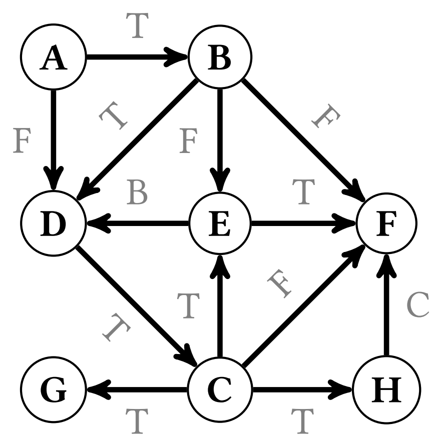
d) Determine the distance from A to every node in the graph.

Solution:

- The distance from A to A is 0.
- The distance from A to B is 1.
- The distance from A to D is 1.
- The distance from A to E is 2.
- The distance from A to F is 2.
- The distance from A to C is 2.
- The distance from A to G is 3.
- The distance from A to H is 3.

Exercise 10.2 *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, give its *pre-* and *post*-number.

Solution: A (1,16), B (2,15), D (3,14), C (4,13), E (5,8), F (6,7), G (9,10), H(11,12)

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

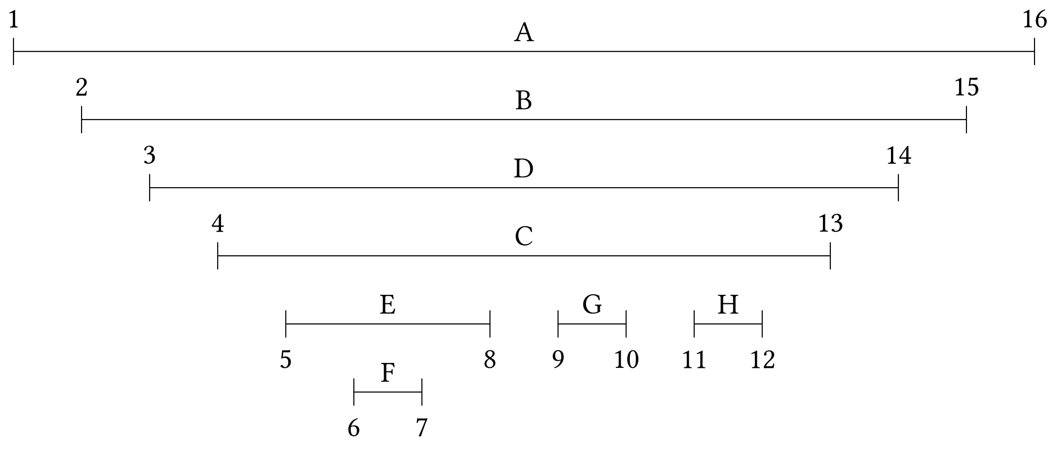
d) Mark every forward edge (Vorwärtskante) with an "F", every backward edge (Rückwärtskante) with an "B", and every cross edge (Querkante) with a "C".

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 and C $\to$ E. Together they form the directed cycle (D $\to$ C $\to$ 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 with the edge from E to D removed. 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?

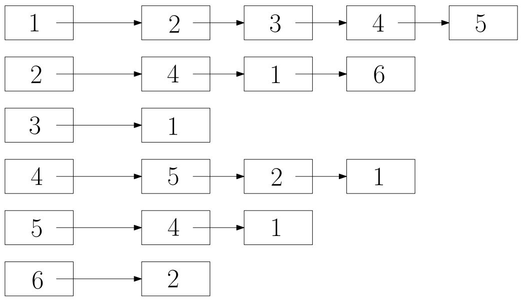
Solution: The execution of depth-first search doesn't change. The topological sorting is: A, B, D, C, H, G, E, F. The pre-ordering is A, B, D, C, E, F, G, H.; it does not give a topological ordering, since there is an edge (H, F) in the graph.

Exercise 10.3 *Data structures for graphs.*

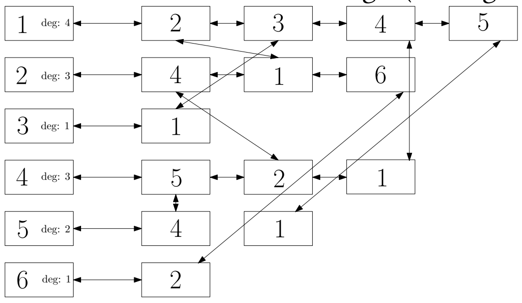
Consider three types of data structures for storing a graph G with n vertices and m edges:

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).

(i) 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.

(ii) 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.

(iii) 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.

(iv) 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.

(v) 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).

(vi) 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.

(vii) 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 10.4 *Maze solver.*

You are given a maze that is described by a $n \times n$ grid of blocked and unblocked cells (see Figure 1). 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.*

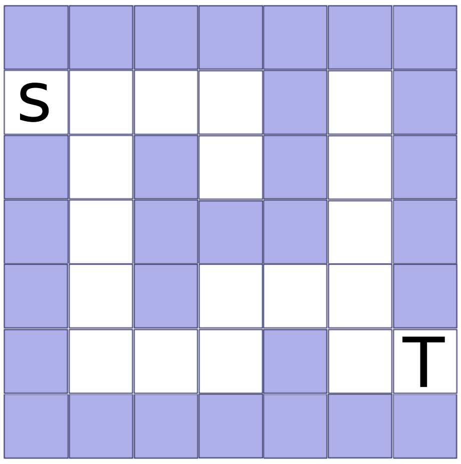


Figure 1: An example of 7×7 maze in which purple fields are blocked, white fields can be traversed (are unblocked). The start field is marked with 'S' and the target field with a 'T'.

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.

Exercise 10.5 *Driving on highways* **(1 point)**.

In order to encourage the use of train for long-distance traveling, the Swiss government has decided to make all the m highways between the n major cities of Switzerland one-way only. In other words, for any two of these major cities C_1 and C_2, if there is a highway connecting them it is either from C_1 to C_2 or from C_2 to C_1, but not both. The government claims that it is however still possible to drive from any major city to any other major city using highways only, despite these one-way restrictions.

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 major cities in Switzerland (which is of size $|V| = n$), and there is a directed edge from $u \in V$ to $v \in V$ if and only if there is a highway going from city u to city v. The corresponding graph problem is to determine whether for any two vertices $u, v \in V$, there is a (directed) path from u to v in $G = (V, E)$.

b) Describe an algorithm that verifies the correctness of the claim in time $O(n + m)$.

Hint: *You can make use of an algorithm from the lecture. However, you might need to modify the graph described in part (a) and run the algorithm on some modified graph.*

Solution:

The algorithm is the following. Let $v_0 \in V$ be any vertex in the graph. We first run DFS starting from vertex v_0 on the graph G described in part (a), and denote by V_0 the set of vertices that were visited by the DFS. Then, we define a new graph $G' = (V, E')$ with the same vertices, and whose edges are given by the reversed edges of G, i.e. $E' = \{(v, u) \in V^2 : (u, v) \in E\}$. We run DFS again starting from vertex v_0 on the graph G', and denote by V_0' the set of vertices that were visited by the DFS. The algorithm outputs that the claim is correct if $V_0 = V = V_0'$, and that the claim is false otherwise. Note that the first DFS takes time $O(|V| + |E|)$, and the second one $O(|V| + |E'|)$, and since $|V| = n$ and $|E'| = |E| = m$ the total runtime is indeed $O(n + m)$.

For completeness, let us now explain why this is a correct algorithm (you did not have to do it since we had not explicitly asked for it in the task). We have to show the equivalence of the following two statements:

(1) For any two vertices $u, v \in V$, there is a (directed) path from u to v in $G = (V, E)$.

(2) $V_0 = V$ and $V_0' = V$.

(1) $\Longrightarrow$ *(2):*

Note that V_0 is the set of vertices v for which there is a directed path from v_0 to v in G. By (1), there is a path from v_0 to v for all vertices $v \in V$, and thus $V_0 = V$. On the other hand, V_0' is the set of all vertices v for which there is a directed path from v to v_0 in G. Indeed, if $v \in V_0'$, this means that in the graph G' with reversed edges there is a path from v_0 to v, which corresponds to a path from v to v_0 in the original graph G. Again by (1), we conclude that $V_0' = V$.

(2) $\Longrightarrow$ *(1):*

Let $u, v \in V$. Since $V_0' = V$ and we have seen that V_0' is the set of vertices from which we can reach v_0 in G, we know there is a directed path π_u from u to v_0 in G. Since $V_0 = V$ and V_0 is the set of vertices that we can reach from v_0 in G, we know there is a directed path π_v from v_0 to v in G. Concatenating the paths π_u and π_v, we obtain a directed path from u to v in G.