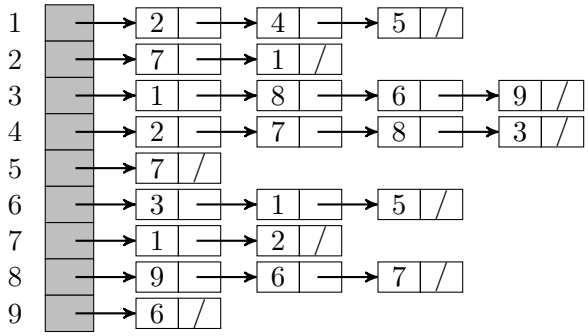
Exercise VIII, Algorithms I 2024-2025

These exercises are for your own benefit. Feel free to collaborate and share your answers with other students. There are many problems on this set, solve as many as you can and ask for help if you get stuck for too long. Problems marked * are more difficult but also more fun :).

Basic Graph Algorithms

1 What are the numbers of edges going into and out of vertex 7 in the graph represented by the following adjacency-list?



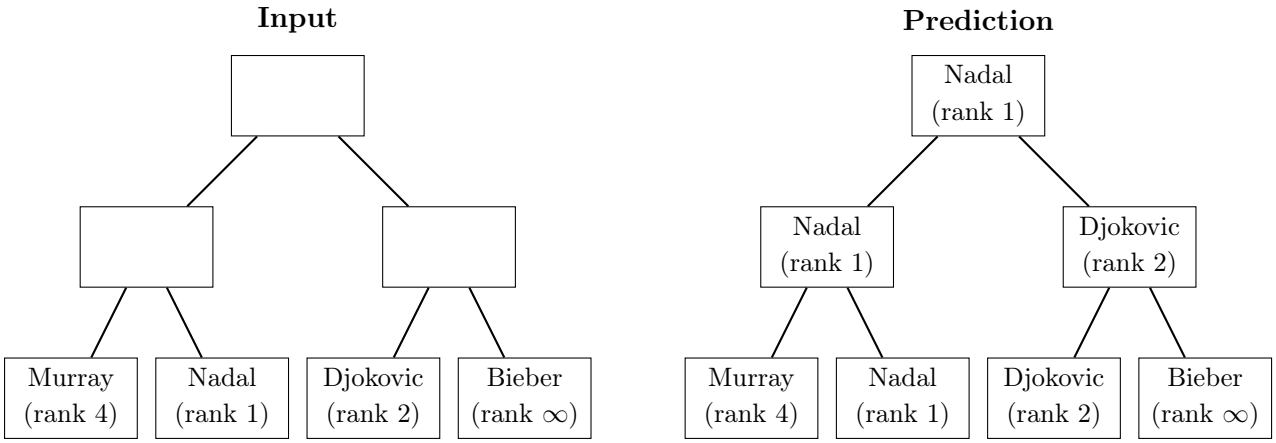
Solution: The number of edges going out of 7 is 2 because vertex 7 has an outgoing edge to vertex 1 and an outgoing edge to vertex 2. Its indegree is 4 since it has incoming edges from vertices $2, 4, 5$, and 8.

2 (Exercise 22.4-3) Give an algorithm that determines whether or not a given undirected graph $G = (V, E)$ contains a cycle. Your algorithm should run in $O(|V|)$ time, independent of $|E|$.

Solution: To detect a cycle in an undirected graph $G = (V, E)$ we slighly modify DFS as follows. IF DFS-VISIT(G, u) encounters a neighbor of u, call it v that is gray and v is not the parent of u, then v is an ancestor of vertex u in the DFS tree. This means that there is a path P_{vu} from vertex v to u that does not use the edge (u, v). Thus we could extend this path by edge (u, v) and obtain a cycle. If for any connected component of G we have the situation described above our modified DFS returns "cycle", otherwise it returns "no cycle". The running time of the algorithm is $O(|V|)$ and is independent of $|E|$, because if $|E| > |V| - 1$ then there is a cycle in the graph and this will be detected in $O(|V|)$ time by the above procedure.

3 *(old exam question)* **Australian Open.** The draw of Australian Open was recently announced. In tennis each match is between two players and the winner progresses to the next round. This naturally leads to a complete binary tree structure of the tournament. At the leaves, we have all the players in the tournament. At the next level, we have those that won their first match, and so on. In particularly, the root of the tree contains the winner of the tournament. We are interested in predicting the outcome of *every match* in Australian Open 2014. To do this we use the following simplifying assumption: a better ranked player always wins over a player with worse rank.

Consider the figure below for an example. We have four players entering the tournament of various rankings. The prediction tree then predicts the winner of each match. For example, as Rafael Nadal is currently ranked number one and Andy Murray is ranked number four, Rafael Nadal wins against Andy Murray. Similarly, we predict that Nadal wins against Djokovic in the final.



Design and analyze an efficient algorithm for the Australian Open prediction problem:

Input: The root of a complete binary tree (the draw) with n players as leaves. Each player/node has a *name* and a *ranking* that is initially empty for nodes that are not leaves. In addition, each node has pointers to its *left child*, its *right child* and *its parent*. Finally, no two players have the same ranking.

Output: A complete binary tree (the prediction) where each node contains the player (his name and rank) that has reached this stage assuming that better ranked players always win over worse ranked players.

Your algorithm should run in *linear time* in the number of players.

Solution: We shall do an algorithm that is similar to DFS and inspired by Divide-and-Conquer. When we visit a node/player p we first calculate his left subtree rooted at $p.left$ and then his right subtree rooted at $p.right$. After we have calculated both subtrees, we compare the two winning players of them and update p accordingly:

- If $p.left.rank < p.right.rank$ then (left player is winning) set $p.rank = p.left.rank$ and $p.name = p.left.name$
- Else (right player is winning) set $p.rank = p.right.rank$ and $p.name = p.right.name$

The pseudocode of the algorithm PREDICT-TOURNAMENT that takes the root of the tournament as input is as follows:

```
PREDICT-TOURNAMENT(p)
1. if p.left ≠ NIL and p.right ≠ NIL
2.     PREDICT-TOURNAMENT(p.left)
3.     PREDICT-TOURNAMENT(p.right)
4.     if p.left.rank < p.right.rank
5.         p.rank = p.left.rank
6.         p.name = p.left.name
7.     else
8.         p.rank = p.right.rank
9.         p.name = p.right.name
```

Runtime Analyzes: Note that the body of the algorithm takes $\Theta(1)$ time and is executed exactly once for each node in the complete binary tree. As the complete binary tree has n leaves, the total number of nodes it has is $\sum_{i=0}^{\log_2 n} n/2^i$ which is less than $2n$ and greater than n. Therefore the total running time of the algorithm is $\Theta(n)$.

4 (*, Exercise 22.2-7) There are two types of professional wrestlers: "babyfaces" ("good guys") and "heels" ("bad guys"). Between any pair of professional wrestlers, there may or may not be a rivalry. Suppose we have n professional wrestlers and we have a list of r pairs of wrestlers for which there are rivalries. Give an $O(n+r)$-time algorithm that determines whether it is possible to designate some of the wrestlers as babyfaces and the remainder as heels such that each rivalry is between a babyface and a heel. If it is possible to perform such a designation, your algorithm should produce it.

Solution: Consider the graph $G = (V, E)$ in which the vertices V correspond to the n professional wrestlers and the edges E correspond to the r pairs of rivalries. We note that the desired designation is possible if and only if the graph G is bipartite. Thus, our goal is to give an $O(n + |E|)$ algorithm to determine if G is bipartite.

To do this, we can use Breadth-First Search, which runs in $O(|V| + |E|)$ time. We modify the code BFS(G,s) to label each vertex an "odd" or "even". In the beginning of the code (say, before line 2), we label vertex s (the starting vertex) as "odd". In line 11, when we DEQUEUE(Q) an element u, we note if the label of u is "odd" or "even" and we set the current label to be the opposite. Then for each unlabeled vertex $v \in G.Adj[u]$, we assign this opposite label to vertex v.

We want to run BFS on each connected component of G. To this end, we modify it further as follows: it first sets $u.d = \infty$ for each vertex u, and then loops over every vertex s as the starting vertex; then, if $s.d = \infty$, it performs a BFS-search from s (it initializes the queue, etc.). This way, one BFS-search corresponds to one connected component of G.

After running BFS on each component of G, we can (in $O(|V| + |E|)$ time go through each vertex u and check if it is has the opposite label as each vertex in $G.Adj[u]$. If not, we have found an edge in which both endpoints have the same label, which means that G is not bipartite.

5 (Exercise 22.5-1) How can the number of strongly connected components of a graph change if a new edge is added?

Solution: The number of strongly connected components (SCCs) may remain the same or reduced to any number no less than 1, i.e., let m be the number of SCCs in the original graph, and m' be the number of SCCs of the new graph after adding the edge, then

$$m' \leq m \qquad \text{and} \qquad m' \geq 1.$$

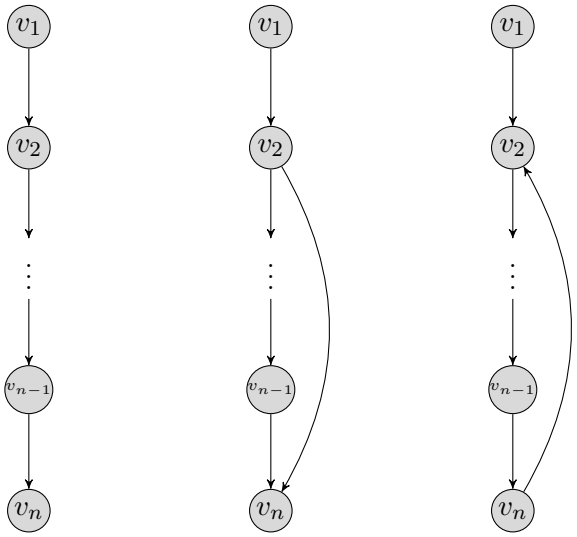


Figure 1. Examples for changing of the strongly connected components by adding an edge.

An explanatory example is shown in Figure 1. The left figure shows the original graph in which each node is an SCC, thus total n SCCs. If the new added edge is a self-loop of any node, or if the new added edge is pointing down, then the number of SCCs will not change. If the new added edge is pointing up, it forms an SCC, and it may reduce the number of SCC to any number between 1 and n.

6 (Exercise 22.5-3) Professor Bacon claims that the algorithm for strongly connected components would be simpler if it used the original (instead of the transpose) graph in the second depth-first search and scanned the vertices in order of *increasing* finishing times. Does this simpler algorithm always produce correct results?

Solution: Unfortunately, it does not work as can be seen by the graph consisting of three vertices A, B and C. The edges are (A, B) (B, A) and (A, C). So if we start a depth-first search from A, then go to B and finally visit C we will have the following discovery and finishing times:

$$\begin{aligned} A.d &= 1, \qquad A.f = 6 \\ B.d &= 2, \qquad B.f = 3 \\ C.d &= 4, \qquad C.f = 5 \end{aligned}$$

In other words, the finishing order is B, C, A. So Bacon will start his DFS from B and then discover the whole graph (B, A, C), since both A and C are reachable from B. Clearly this is incorrect, as there should be two strongly connected components: (B, A) and C.

For a longer discussion, see *http://mypathtothe4.blogspot.ch/2013/03/strongly-connected-components-225-3-clrs.html*

Flow Networks

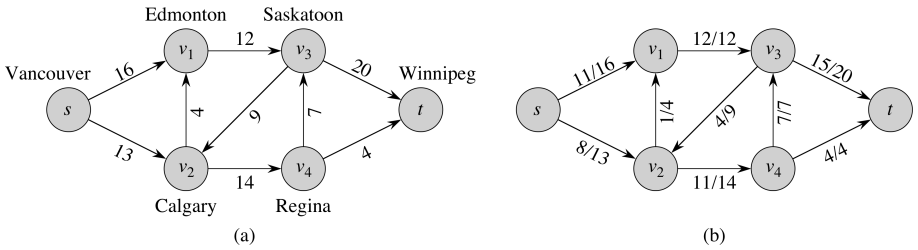


Figure 2. Example of a flow network (on the left) and of a flow (on the right).

7 (Exercise 26.2-2) In Figure 2(b) what is the flow across the cut $(\{s, v_2, v_4\}, \{v_1, v_3, t\})$? What is the capacity of this cut?

Solution: Deferred to problem set 9.