University of Freiburg
Dept. of Computer Science
Prof. Dr. F. Kuhn
Marc Fuchs

Algorithm Theory
Sample Solution Exercise Sheet 4

Due: Friday, 14th of November 2025, 10:00 am

Exercise 1: USB drive *(7 Points)*

You have a special USB drive with a total capacity of T gigabytes (GB), where $1 \leq T$. Your goal is to fill the drive with as much data as possible **without (ever) exceeding** its capacity T.
You have access to an unlimited supply of two types of data files:

- **Text Packages (Type A):** Each is exactly A GB in size.
- **Video Packages (Type B):** Each is exactly B GB in size.

A, B and T are integers, and we know that $1 \leq A < B \leq T$. You can add these files one by one, starting from an empty drive (0 GB used). You also have a one-time-use **compression algorithm** that can be applied **at most once** at any point[1]. When you run it, it instantly compresses all data currently on the drive, reducing the total space used to **half** of its current amount, **rounded down** (e.g., $\lfloor \text{space\_used}/2 \rfloor$). After compressing, you can continue to add more files to the drive.
Your task is to determine the **maximum amount of space** you can fill on the drive without ever exceeding the T GB capacity. Your algorithm to solve this must run in $O(T)$ time. *(7 Points)*

Hint: Before you solve this problem, think of how you would use *dynamic programming* to solve the easier problem where you do not have a one-time-use compression algorithm[2], i.e., you should be able to tell for all integers $1 \leq x \leq T$ if you can exactly fill space x using the two package types. You can now extend this algorithm to solve the original problem.

Sample Solution

Let's first solve the easier problem where no compression is used with a dynamic programming approach. We define $PRE(x) \in \{0, 1\}$ to be true (1) if we can somehow add the packages such that exactly x GB of our USB store are used. If there is no way to combine type A and B packages to use exactly x GB we instead denote the output as false (0), precisely, $PRE(x) = 0$. For convenience, we define $PRE(0) = 1$ (as if we do not take any package,0 GB are used).
We will now discuss how to compute $PRE(x)$ for all $1 \leq x \leq T$ in time $O(T)$. More precisely, we will show that for a fixed integer x in this interval $PRE(x)$ can be computed in constant time if for all $1 \leq y < x$ the value of $PRE(y)$ is already known. The idea is as simple as follows: If we can exactly fill y GB of space and either $y + A = x$ or $y + B = x$, we can also fill exactly x GB of our USB drive. Otherwise, we can not.

$$PRE(x) = \begin{cases} 1 & \text{if } x = 0 \\ 1 & \text{if } PRE(x - A) = 1 \\ 1 & \text{if } PRE(x - B) = 1 \\ 0 & \text{else} \end{cases}$$

[1] To be clear, you can compress at any point where the currently stored data takes $0 \leq x \leq T$ GB.

[2] Note that this simpler problem can actually be solved without DP in time $O(T/B)$. However, it is not so clear if this solution can be extended to solve the original problem. Hence, try to construct an $O(T)$ time DP algorithm!

Note that we can now compute all values in increasing order $PRE(0), PRE(1), PRE(2), ..., PRE(T)$, and thus need at most time $O(T)$ until all these values are computed. The answer to the problem is now the highest x such that $PRE(x) = 1$. Since all these values are already computed (and say stored in an array), we can find this value in at most $O(T)$ rounds.
We will now solve the problem with compression. For that we first compute $PRE(x)$ as before. When this is done we compute $POST(x) \in \{0, 1\}$ as the function that states if we can reach space x using the compression. Note that $POST(x) = 1$ is true if some $POST(y)$ is true with $y+A = x$ or $y+B = x$ (symmetrical to PRE), but $POST(x)$ is also true if $PRE(2x)$ is true (and due to the rounding also $PRE(2x + 1)$), and we then use compression. This is formalized as follows:

$$POST(x) = \begin{cases} 0 & \textit{if } x = 0 \\ 1 & \textit{if } POST(x - A) = 1 \\ 1 & \textit{if } POST(x - B) = 1 \\ 1 & \textit{if } PRE(2 \cdot x) = 1 \\ 1 & \textit{if } PRE(2 \cdot x + 1) = 1 \\ 0 & \textit{else} \end{cases}$$

We now again compute all values step-by-step in increasing order, and since PRE is already computed, each new $POST$ value can be computed in time $O(1)$ by checking these 4 values given in above recursion. Thus, also all $POST$ can be computed in time $O(T)$. The output is now the highest x for that either $PRE(x) = 1$ or $POST(x) = 1$.

Exercise 2: Weighted Independent Set on Trees *(6 Points)*

Let $G = (V, E)$ be a graph with **node** weights $\in \mathbb{N}$, denoted by $w(v)$ for node $v \in V$. We call a subset of the nodes $I \subseteq V$ a maximal weight independent set if for all nodes $u, v \in I$ with $u \neq v$, we have $\{u, v\} \notin E$ (in other words, no two nodes in I are neighbors in G) and furthermore the sum over the weights in I, denoted as $w(I)$,

$$w(I) := \sum_{v \in I} w(v)$$

is as large as possible among all such independent sets.

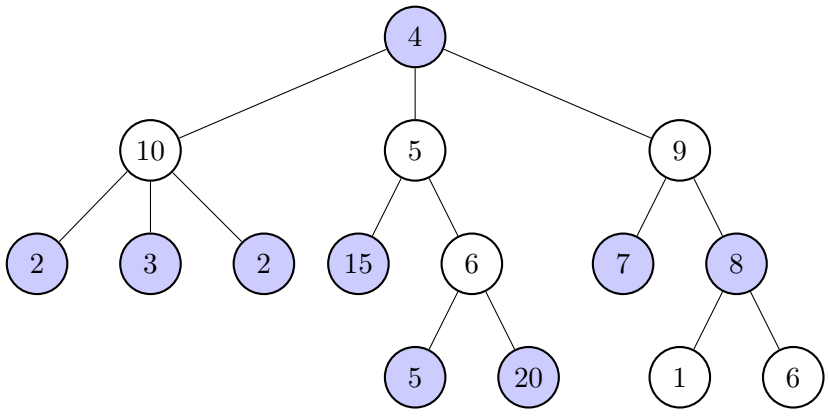

Figure 1: Example tree with a maximum weight independent set marked in blue.

For this task, assume you are given a (node weighted) tree T of n nodes and you want to compute the maximum weight independent set of T. For simplicity, you can assume that the tree is rooted, i.e., there is a root node r and each node v has exactly one parent node (except for the root r) and all other neighbors of v are its children. Try to come up with a *dynamic programming* solution that runs in time $O(n)$. Argue that your running time is as stated. *(5 Points)*

Hint: For each node v consider the tree T_v, that is the subtree of T rooted at v. What is the maximum weight independent set of T_v if you already know the maximum weight independent sets of T_u for all children u of v?

Sample Solution

We define $OPT(v, b)$ for $v \in V$ and $b \in \{0, 1\}$ to be the value if the maximum weight independent set I of T_v in the case that for $b = 0$, v is not in I and for $b = 1$ we have v is in I. Furthermore, let $C(v)$ be the set of children of v in T. Note that, if v is a leaf of T, $OPT(v, 0) = 0$ and $OPT(v, 1) = w(v)$. If v is not a leaf, in the case $OPT(v, 1)$ case we want each node in $C(v)$ to be **not** in the solution (as this would contradict the independence property), while for $OPT(v, 0)$ we do not really care if the children of v are in the independent set or not, we just want the weight to be maximized. This observation can now be put together as follows:

$$OPT(v,1) = \begin{cases} w(v) & \textit{if } v \textit{ is a leaf} \\ w(v) + \sum_{u \in C(v)} OPT(u,0) & \textit{else} \end{cases}$$

$$OPT(v,0) = \begin{cases} 0 & \textit{if } v \textit{ is a leaf} \\ \sum_{u \in C(v)} \max\{OPT(u,0), OPT(u,1)\} & \textit{else} \end{cases}$$

Our algorithm is as simple as follows: First compute both OPT values for each leaf node, then compute both OPT values for the layer above the leafs and so on until you reach the root r. The output is then $\max\{OPT(r, 0), OPT(r, 1)\}$.

Runtime For a leaf node, the computation of both OPT values is constant. For a non-leaf node v the computation time is $O(|C(v)|)$ as we have to add all the childrens values together. Thus, for all the leaf nodes the computation takes time at most $O(n)$. Note that each edge in the tree corresponds to exactly one child. Therefore, for all non-leaf nodes, the overall computation time is $O(m)$ (where m is the number of edges). Since trees have exactly $m = n-1$ edges, the overall runtime is $O(m+n) = O(n)$ to compute all OPT values.

Exercise 3: Longest Walk *(7 Points)*

Suppose you are given a graph $G = (V, E)$, where each node $v \in V$ is labeled with an elevation value $h(v) \in \mathbb{N}$. You can safely traverse an edge $(u, v) \in E$ from node u to node v if and only if the absolute difference between the elevation values of u and v is at most δ, that is, $|h(u) - h(v)| \leq \delta$, where $\delta > 0$ is an integer parameter.

Our goal now is to find a longest possible walk in the graph such that the walk starts at some node $s \in V$ and ends at another node $t \in V$. The walk should consist of two segments: an uphill segment, where the elevation must strictly increase in each step, followed by a downhill segment, where the elevation must strictly decrease in each step.

Note that a walk in a graph is path on which nodes are allowed to repeat. However, in the uphill segment, the elevation values are strictly increasing and in the downhill segment, the elevation values are strictly decreasing. Each node can therefore appear at most once in the uphill segment and at most once in the downhill segment (but can appear in both segments).

Your input consists of the graph $G = (V, E)$, the elevation function $h : V \to \mathbb{N}$, the starting node s, the target node t, and the parameter $\delta > 0$. Your task is to find a longest possible walk as described above or determine that no such walk exists.

Give an algorithm that solves the problem in time $O(nm)$, where as usual $n = |V|$ and $m = |E|$. Argue correctness and run time.

Hint: It makes sense to first solve the following problem: For every $v \in V$, find the longest path from s to v on which the elevation values strictly increase or determine that no such path from s to v exists.

Sample Solution

Rough idea and correctness: We first solve the simpler version of the problem where given a fixed starting node s and a target node t as input and the goal is to find the longest such walk starting from this s and ending in this t. Initially, we can get rid of any edge that doesn't satisfy the property $|h(u) - h(v)| \leq \delta$ and now we work on our new graph G' (which could be disconnected). Next, we make sure via e.g. BFS that s and t belong to the same connected component, otherwise we know no such walk from s to t exists.
Now, let $uphill(s, v)$ be the value of the longest path value from s to v on which the elevation values strictly increase in G' or determine that no such path from s to v exists. Similarly let $downhill(v, t)$ be the value of the longest path value from v to t on which the elevation values strictly decreases or determine that no such path from v to t exists.
Afterwards, we compute for each $v \in V$, the values $uphill(s, v)$ and $downhill(v, t)$. Thus the final solution should be $\max_{v \in V}\{uphill(s, v) + downhill(v, t)\}$. Hence we only need to solve the hint to solve our problem i.e. for all $v \in V$ compute all $uphill(s, v)$ values and in an analogous way we can find all the $downhill(v, t)$ values.
Thus, let's answer the hint. First we orient the graph we are working on such that for every edge (u, v) we orient u into v iff $h(u) < h(v)$. Now, we notice that our graph is a DAG. And for such graphs we can find a topological order in $O(m + n)$, thus we can enumerate our vertices $\{v_1, v_2, ..., v_k\}$ where $k \leq n$ according to the topological order i.e. for every edge (v_i, v_j) we have that $i < j$.
Now our problem can be reformulated to finding the longest directed path from s to any v if it exists. To eliminate the nodes that don't have a directed path from s to them, we can e.g. in $O(m + n)$ run a BFS algorithm over the oriented graph starting from s (i.e. the BFS proceeds only over outgoing edges i.e. the fist level is s and the second contains all out neighbors of s and so on..), and at the end we know that all nodes v that this BFS didn't visit don't have a directed path from s to them, hence we can output that no such s to v directed path exists and only consider the nodes that are reachable from s in the following. Let $N^-(v)$ be the set containing all incoming neighbors of v.
We notice the following recursion $uphill(s, v) = 1 + max_{u \in N^-(v)} uphill(s, u)$.
And for the base case, we have $uphill(s, s) = 0$.
Hence, if we use this recursion to compute the uphill value from s to the final node that is reachable from s according our topological order, we would have also computed all $uphill(s, v)$ values for any v by doing so and answered the hint.
To see this notice that the nodes v_i on any directed path must have their corresponding indices in an increasing order i.e. for any subpath $v_i \to v_j \to v_k$ we should have that $i < j < k$. So we will notice that each $uphill(s, v_i)$ will only use precomputed (memorized) values i.e. only values of the form $uphill(s, v_j)$ for $j < i$ to compute its own uphill value.

Finally, if we want to solve the more general version of the problem where we don't have a fixed s and t rather we ask to find the longest such walk starting from any s and ending in any t, then we first notice that the ascending and descending segments will have to be the same length i.e. an optimal solution would be to find the longest directed path in the graph and use the same path for the longest walk i.e. we go up and down on that same path. Hence, it would be enough to check for every node s what is the longest directed path from this s to any other node and for that we need to compute $uphill(s, v)$ for all v and we can do so using the simpler version of the problem. Eventually, we compute the $max_{s \in V}\big(max_{v \in V} uphill(s, v)\big)$ for the final solution (we can multiply it by 2 if we want to consider the length of the longest walk).

Running time:
For the simpler version, we have at most n subproblems corresponding to the values of the $uphill(s, v)$ for all v reachable from s via a directed path. And if we give a dp algorithm based on our recursion above, then the computation of each such subproblem corresponding to finding $uphill(s, v)$ takes at most the $degree(v)$ much time (i.e. ofcourse and as always without not counting the cost of recursive calls), thus all in all the running time for solving the hint will be $O(\sum_{v \in V} degree(v)) = O(m)$. Analogously, we spend $O(m)$ for the downhill value and to find $\max_{v \in V}\{uphill(s, v) + downhill(v, t)\}$ we

have an additive $O(n)$ in the running time. Moreover, for the BFS algorithm and topological sorting we will spend another additive $O(m+n)$. Therefore, the overall running time will be $O(m+n)$.

As for the general version, we will have to invoke the simpler version solution n times, which will give us a total running time of $O(n(m+n))$ which is in $O(nm)$.