University of Freiburg
Dept. of Computer Science
Prof. Dr. F. Kuhn
S. Faour, M. Fuchs, A. Mályusz

Algorithm Theory
Sample Solution Exercise Sheet 4

Due: Friday, 15th of November, 2024, 10:00 am

Remark: You are required to use the principle of *dynamic programming* in all of your algorithms. It is preferable if you write the algorithms in pseudocode.

Exercise 1: Paper Submissions *(6 Points)*

Professor Kuhn has to finish writing n different research papers p_i and would like to submit each one of them to a conference. However each paper p_i takes t_i time to be done and submitted by deadline d_i i.e. each deadline is according to the conference that the Professor wants to submit that paper in. Note that when we say by deadline d_i, we mean that at the latest the paper submission should be at time d_i. Moreover, writing any paper is available to be scheduled starting at time s.
Now, the task of writing the full paper p_i and then submitting it needs to be assigned a period from $s_i \geq s$ to $f_i = s_i + t_i$, and doing the same task for different papers should be assigned nonoverlapping intervals. Such an assignment of times will be called a *schedule*.
We consider the case in which writing each paper must either be finished and submitted by its deadline or not at all. We'll say that a subset P of the papers is *schedulable* if there is a schedule where the Professor is able to finish writing each paper in P and submit each of them by its deadline. Your problem is to select a schedulable subset of papers of maximum possible size and give a schedule for this subset that allows each paper to be fully written and submitted by its deadline.

Assume that all deadlines d_i and required times t_i are integers and $d_i \geq t_i$. Give an algorithm to find an optimal solution. Your algorithm should run in time polynomial in the number of papers that needs to be written and submitted n, and the maximum deadline $D = \max_i d_i$. Argue correctness and running time.
Hint: Prove that there is an optimal solution P (i.e., a schedulable set of maximum size) in which the papers in P are scheduled in increasing order of their deadlines.

Sample Solution

(a) Using a simple exchange argument, we can see that if at any point in an optimal schedule some paper p_i was scheduled just before some other paper p_j such that $d_j < d_i$, then one can build a new optimal schedule with only p_i and p_j swapped and nothing else changed i.e. all papers will still finish by their deadlines.

(b) **Algorithm idea and correctness:** Assume the papers are given by increasing order of deadlines and we will assume w.l.o.g that they are numbered this way i.e. we assume that this is the corresponding deadlines for our input set of papers $p_1, p_2, ..., p_n$ are $d_1 \leq d_2 \leq ... \leq d_n = D$, respectively. We will define two different parameters for our recursive function. For $0 \leq d \leq D$ and $j = 1, \ldots, n$, let $OPT(D, n)$ denote the maximum schedulable subset of papers from the set $\{p_1, \ldots, p_n\}$ that can be satisfied by the deadline D i.e. each $p_i \in \{p_1, \ldots, p_n\}$ will be submitted by time $min\{d_i, D\}$ (so after time D we cannot submit any paper anymore even if the deadline d_i of the paper is $d_i > D$).

We now have two cases: either the last paper p_n will be chosen in the schedule and submitted or not. If p_n is not chosen, then the problem reduces to the subproblem using only the first $n-1$ papers. On the other hand, if paper p_n is chosen, then by part (a) we know that we may assume that paper p_n will be scheduled last. In order to make sure we submit p_n by its deadline $min\{d_n, D\}$, all other papers p_i chosen for our schedule should be also done by time $min\{d_i, D - t_n\}$.

Hence we have only the following cases:

- If paper n is not chosen in the optimal solution $OPT(D, n)$, then $OPT(D, n) = OPT(D, n-1)$ (or better $OPT(D, n) = OPT(min\{D, d_{n-1}\}, n-1)$)
- If paper n is chosen in the optimal solution $OPT(D, m)$ and this can only happen iff $t_n \leq D$, then $OPT(D, n) = OPT(D - t_n, n-1) + 1$ (or better $OPT(D, n) = OPT(min\{D - t_n, d_{n-1}\}, n-1) + 1)$

For the base case we have for $0 \leq d \leq D$, $OPT(d, 0) = 0$, hence taking the maximum of both cases above gives us the final solution.

Based on that, we propose the following top down and bottom down dp algorithm using memoization:

Algorithm 1 `Schedule`(D, n) ▷ `memo` is a 2-dimensional array initially empty

if $n = 0$ **then return** 0 ▷ base case
if `memo`$[D, n] \neq$ `Null` **then return** `memo`$[D, n]$ ▷ decision was made before
`memo`$[D, n] \leftarrow \max \begin{cases} \texttt{Schedule}(min\{D, d_{n-1}\}, n-1) \\ \texttt{Schedule}\ (min\{D - t_n, d_{n-1}\}, n-1) + 1 \quad \text{if } t_n \leq D \end{cases}$ ▷ Memoization
return `memo`$[D, n]$

And the following is a bottom top algorithm (a way to build up values for the subproblems):

Algorithm 2 Schedule(D, n)

```
Array S[0...D, 0...n]
for d = 0, ..., D do
    S[d, 0] = 0
for m = 1, ..., n do
    for d = 0, ..., D do
        if t_m > d then
            S[d, m] = S[min{d, d_{m-1}}, m - 1]
        else
            if S[min{d, d_{m-1}}, m - 1] > S[min{d - t_m, d_{m-1}}, m - 1] + 1 then
                S[d, m] = S[min{d, d_{m-1}}, m - 1]
            else
                S[d, m] = S[min{d - t_m, d_{m-1}}, m - 1] + 1
return S[D, n]
```

Running time: we have $(D+1) \times (n+1)$ entries in our 2D array to fill, i.e. $(D+1) \times (n+1)$ subproblems (notice that in the iterative procedure how we are filling our matrix column by column). Moreover, when filling the matrix iteratively (in the bottom top approach) or due to memoization (in the top bottom approach), we compute each entry value `memo`$[i, j]$ for $i, j \in \{0, 1, ..., n\} \times \{0, 1, ..., D\}$ once. Also, each computation of `memo`$[i, j]$ costs $\mathcal{O}(1)$ in the current step (i.e. not counting the cost of recursive calls, thus the only thing we are doing is determining the maximum between two values). Therefore, the total running time is $\mathcal{O}(n \cdot D)$.

Exercise 2: Generalized Max. Product Subarray *(7 Points)*

Given an integer array A of length n, and an integer m (where $m \leq n$). The *goal* is to find the maximum product that can be obtained by selecting exactly m non-overlapping contiguous subarrays. Note that each subarray should be non-empty. The following is an example:
Let $A = [1, -2, 3, -4, 5, -6, 0, 7, 8, 9, 0]$ and $m = 3$.
The maximum product can be achieved by choosing the subarrays $[3, -4, 5, -6], [7]$, and $[8, 9]$ with a total product of $3 \times -4 \times 5 \times -6 \times (7) \times (8 \times 9) = 181440$.

Give a polynomial time algorithm that achieves our goal. Argue correctness and run time.

Sample Solution

Rough idea and correctness: Notice that for $m = 1$ it is the original maximum product subarray problem discussed as an extra bonus question in our very first tutorial session, so this question is a generalization of it.
Now let $max(i, k)$ be the maximum product that is obtained by selecting exactly k nonoverlapping contiguous subarrays from the subarray $[A[0], ..., A[i]]$. Similarly, we define $min(i, k)$ to be the minimum product that is obtained by selecting exactly k nonoverlapping contiguous subarrays from the subarray $[A[0], ..., A[i]]$.
Now we have only the following cases for both $max(i, k)$ and $min(i, k)$ (which we keep track of simultaneously):

- If $A[i]$ was not part of the optimal solution, then
  $max(i, k) = max(i - 1, k)$ and
  $min(i, k) = min(i - 1, k)$
- If $A[i]$ is part of the optimal solution and is considered an array on its own, then
  $max(i, k) = max\{A[i].max(i - 1, k - 1), A[i].min(i - 1, k - 1)\}$ and
  $min(i, k) = min\{A[i].max(i - 1, k - 1), A[i].min(i - 1, k - 1)\}$
- If $A[i]$ is part of the optimal solution and belongs to the final subarray (ie the kth one) that will also include at least $A[i - 1]$ in it, then
  $max(i, k) = max_{1 \leq r \leq i-k+1}\{A[i]....A[i-r]max(i-r-1, k-1), A[i]....A[i-r]min(i-r-1, k-1)\}$
  and $min(i, k) = min_{1 \leq r \leq i-k+1}\{A[i]....A[i-r]max(i-r-1, k-1), A[i]....A[i-r]min(i-r-1, k-1)\}$

For the base case, we can learn for all $i = 0, ..., n - 1$, the values $max(i, 1)$ and $min(i, 1)$ by solving the original problem i.e. for $m = 1$ in $O(n)$ (we did this in our first tutorial session as mentioned above) and for $i = 1, ..., n - 1$, we set $max(i, i + 1) = min(i, i + 1) = A[0]...A[i]$.

Now for the final value of $max(i, k)$ (and $min(i, k)$), we take the maximum (and minimum resp.) amongst the $max(i, k)$ (and $min(i, k)$ resp.) values of the above three cases.
Note that for each $k = 1, ..., m$, we will only compute the values $max(i, k)$ (and $min(i, k)$) for $i = k - 1, ..., n - 1$, as the other values can't exist.

Running time: Notice that in the dp algorithm based on our recursion above, we would have to fill the upper triangular matrix of both the two 2D arrays simultaneously (one array for the max and another for the min). Hence we have $O(mn)$ entries, i.e. we have $O(mn)$ many subproblems to compute. Moreover, when filling the matrix iteratively (in the bottom top approach) or due to memoization (in the top bottom approach), we compute each entry value $max(i, k)$ and $min(i, k)$ once. Also, the computation of $max(i, k)$ and $min(i, k)$ each costs $\mathcal{O}(n)$ in the current step (i.e. not counting the cost of recursive calls, because the only thing we are doing is determining the maximum (min resp.) between three cases: for the first two cases above we spend $O(1)$ time, however for the final case, which is the most expensive step, we would have to check the max (min resp) of $O(n)$ values in the worst case). Therefore, the total running time is $\mathcal{O}(n^2 m)$.

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. Note however that 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. A node can however appear in the uphill and in the downhill segment.

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 eg 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 refomulated 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 eg 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 recusion $uphill(s, v) = 1 + max_{u \in N^-(v)} uphill(s, u)$.
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 ie 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 (memoized) 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 desceding 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 ie 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 anf 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 take the 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 ie 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 $O(nm)$.