Submission for Algorithm Theory Exercise 4

Students: Anton Merlin Geburek (5348694), Matthias Zumkeller (5115157), Kai Malaka (5518467), Manuel Di Biase (5515822)

November 15, 2024

1 Paper Submissions

Algorithm for paper submission

```
• sort P with increasing deadlines
• initiate memorization dictionary:
    – memo = {}
• define Opt(P, i, D) with set of papers P, index i and max Deadline D
    – handle base case, no papers or time left
        ∗ if i < 0 or D == 0: return 0, [ ]
    – check if result is already computed
        ∗ if (i, D) in memo: return memo[(i, D)]
    – get $t_i, d_i$ from $P[i]$
    – case 1 exclude paper i
        ∗ exclude paper, subset = Opt(P, i-1, D)
    – case 2 include paper i, if enough time is left and the deadline will be met
        ∗ include paper = 0, subset include = [ ]
        ∗ if $D \geq t_i$ and $D \leq d_i$
            · include paper, subset include = $Opt(P, i-1, D-t_i)$
            · include paper += 1
            · subset include = subset include + P[i]
    – choose option with max amount of papers
        ∗ result = max((exclude paper, subset exclude), (include paper, subset include))
    – safe result in memo
        ∗ memo[(i, T)] = result
    – return result
```

Correctness

1. The algorithm evaluates each paper $P[i]$ under two cases:

- excluding the current paper $P[i]$
- including the current paper $P[i]$ if time allows

It chooses the option that maximizes the number of papers, ensuring an optimal decision at each step

2. by storing solutions for each paper, the algorithm avoids redundant calculations

Time Complexity

1. Sorting: sorting papers by deadlines takes $O(logn)$ where n is the number of papers
2. Dynamic Programming: the Opt function explores each combination of papers and possible time schedules up to the maximum deadline D, this leads to $O(n * D)$ states stored in memo w
3. Overall Time Complexity: total complexiti is $O(n * logn + n * D)$ where $O(n * log * n)$ is sorting and $O(n * D)$ is the dynamic programming

2 Generalized Max. Product Subarray

An algorithm that solves this problem applying dynamic programming may look as follows. First a matrix M with size $n \times m$ gets initialized with attributes $min_p, max_p, minSets, maxSets$ for each cell, that represent the minimum product, maximum product and their corresponding sets of subarrays for the given cell. For $0 \leq i \leq m$ all entrys $M[i : i].minSets = M[i : i].maxSets = \bigcup_{0 \leq j \leq i} \{\{A[j]\}\}$ get initialized and the corresponding min and max products get set to the product of the sets (e.g. $\prod_{e \in M[i:i].minSets} e$). Additionally all cells $M[i : j]$ with $j < i$ can be initialized with some dummy value *null* since there cannot exist j disjoint sets of i values in this case. After the initialization the remaining cells can iteratively be calculated by reusing previously calculated results. In the following we present pseudocode describing our algorithm. Note that we define functions $max(), min()$ such that they ignore inputs that evaluate to *null* or out of bound indexed inputs of M.

Algorithm 1 Algorithm that calculates m maximum subarrays for an input array A

```
function MaxProductSubarray($A, m$)
    $M \leftarrow new\ Array(n, m)$
    for $i = 1 \dots m$ do
        $M[i:i].minSets \leftarrow \bigcup_{0 \le j \le i} \{\{A[j]\}\}$
        $M[i:i].maxSets \leftarrow \bigcup_{0 \le j \le i} \{\{A[j]\}\}$
        $M[i:i].min_p \leftarrow \prod_{e \in A[0:i]} e$
        $M[i:i].max_p \leftarrow \prod_{e \in A[0:i]} e$
    end for        ▷ Assume that the dummy value $null$ is initialized as described above
    for $i = 1 \dots n$ do
        for $j = 1 \dots m$ do
            if $i == j$ or $M[i:j] == null$ then $continue$
            end if
            $M[i:j].max_p \leftarrow max(M[i-1:j].max_p * A[i], M[i-1:j].min_p * A[i], M[i-1:j-1].max_p * A[i], M[i-1:j-1].min_p * A[i], M[i-1:j].max_p, A[i])$

            $M[i:j].min_p \leftarrow min(M[i-1:j].max_p * A[i], M[i-1:j].min_p * A[i], M[i-1:j-1].max_p * A[i], M[i-1:j-1].min_p * A[i], M[i-1:j].min_p, A[i])$        ▷ Assume that all $M[i-1:j]$ values get excluded if the last element the last subarray is not $A[i-1]$

            if $M[i-1:j]$ is in max then        ▷ The max contains a product from $M[i-1:j]$
                $M[i:j].maxSets \leftarrow (M[i-1:j].maxSets \setminus M[i-1:j].maxSets[j]) \cup (M[i-1:j].maxSets[j] \cup \{A[i]\})$        ▷ Add the number to the last list of $M[i-1:j].maxSets$
            end if
            if $M[i-1:j-1]$ is in max then
                $M[i:j].maxSets \leftarrow M[i-1:j-1].maxSets \cup \{\{A[i]\}\})$        ▷ Add the number as single set to $M[i-1:j-1].maxSets$
            end if        ▷ Do the same update of indices for the minimum sets
        end for
    end for
    return $M[n:m].maxSets$
end function
```

Complexity:
The algorithm runs two nested for-loops and in the second for loop there are multiplications with $\le n$ operations. Therefore the complexity is $T(n) \le n * m * n \le n * n * n \in \mathcal{O}(n^3)$ and thus polynomial.

3 Longest Walk

An algorithm solving this in $\mathcal{O}(nm)$ could look like this:

- Initialize a list $list_s$ of all vertices with an empty $parent_s$-entry and a $pathlength_s$-entry of 0
- Initialize a queue with only entry s.
- Pop the next queue element (s), do an exhaustive search across all edges e_{su} to neighbouring vertices u, where $0 > h(u) - h(s) \& |h(u) - h(s)| \le \delta$, record them with s as $parent_s(u) = s$ and $pathlength_s(u) = pathlength_s(s) + 1$ and add them to the queue. (This is *the* Dynamic Programming step, as we reuse a previously computed sub-solution and maximize over two options.)
- Repeat until the queue is empty and when encountering a new vertex u coming from vertex v, check whether the $pathlength_s(u) < pathlength_s(v) + 1$, only then do $pathlength_s(u) = pathlength_s(v) + 1$ and $parent_s(u) = v$.

- Repeat the entire process with a new list $list_t$, now starting with t as first queue-entry.
- Ignoring the vertices with $pathlength = 0$, we receive a list $list_s$ of all vertices reachable from s via strictly uphill movements and the same for t with $list_t$.
- If there is no vertex appearing in both lists with $pathlength \neq 0$ and $s \neq t$, there is no „uphill-downhill-walk" from s to t.
- For all vertices appearing in both lists with $pathlength \neq 0$, choose the vertex v with $max(pathlength_s(v) + pathlength_t(v))$. The walk up from s to v and down from v to t then is the longest possible „uphill-downhill-walk".

The runtime of each exhaustive search comes out to $\mathcal{O}(nm)$, as we have to at worst visit each edge of each vertex. The maximization only needs $\mathcal{O}(n)$.
Correctness:
Assume, there is a „peak-vertex" w, such that $pathlength_s(w) + pathlength_t(w) > max(pathlength_s(v) + pathlength_t(v))$. Then for the s- or the t-ascend-list, there has to be an edge e_xy connecting the path from w to the rest of the list, possibly directly to s (or t), with $0 > h(x_{from\,w}) - h(y_{to\,s}) \& |h(x_{from\,w}) - h(y_{to\,s})| \leq \delta$. But this edge e_xy would have been discovered and traversed during the ascend itself. The path to w was therefore part of the maximization and cannot be a better solution.