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

Algorithm Theory
Sample Solution Exercise Sheet 5

Due: Monday, 22th of November 2024, 10:00 am

Assumption: You may assume that calculations with real numbers can be performed with arbitrary precision in constant time.

Exercise 1: DP *(10 Points)*

Given a tree $G = (V, E)$, an integer $d \in \mathbb{N}$, and a weight function $w : V \to \mathbb{R}_+$, your task is to select a subset $S \subset V$ of the nodes that maximizes the sum $\sum_{u \in S} w(u)$ while ensuring that the subgraph induced by S, denoted $G[S]$, has a maximum degree of at most d.

(a) Solve the problem if $d = 1$ and $w \equiv 1$ in $O(n \log n)$. *(2 Points)*

(b) Now solve the general problem in $O(d \cdot n \log n)$ *(5 Points)*

(c) Now solve the general problem in $O(n \log n)$ *(3 Points)*

Sample Solution

(a) This is similar to the *maximum matching* problem on a tree, which can be solved optimally by a DP solution.

Define:

- dp$[u][0]$: The maximum weight of the subtree of u if u is not in S.
- dp$[u][1]$: The maximum weight of the subtree of u if u is in S but no child of u is in S.
- dp$[u][2]$: The maximum weight of the subtree of u if u is in S and a child is in S as well.

For each node u:

- If u is in S, exactly one child is allowed to be in S as well (but is not allowed to have a child itself in S)
- If u is not in S, all its children are allowed to be in S, but do not need to.

For the base cases, we can define dp$[u][0] = 0$, dp$[u][1] = 1$, dp$[u][2] = 0$ for every leaf node u. From here we recursively go up the tree and compute the values as follows, let u be a non-leaf node, where these 3 values are already computed for all its children:

$$\text{dp}[u][0] = \sum_{\text{child v of u}} \max_{b \in \{0,1,2\}} \text{dp}[v][b]$$

$$\text{dp}[u][1] = 1 + \sum_{\text{child v of u}} \text{dp}[v][0]$$

$$\text{dp}[u][2] = 1 + \max_{\text{child w of u}} \left(\text{dp}[w][1] + \sum_{\text{child v of u} \setminus \text{w}} \text{dp}[v][0] \right)$$

This leads to a solution with a complexity of $O(n)$ due to the tree structure where we have exactly $n-1$ edges and these three computation take time $O(deg(v))$ per node.

(b) **General Solution with Complexity** $O(d \cdot n \log n)$

For the general case where w is not necessarily constant and d can be greater than 1:

- We use a dynamic programming approach with sorting to manage the constraints efficiently.

Define:

- $\mathrm{dp}[u][k][0]$: The maximum weight sum obtainable from the subtree rooted at u, where exactly k children of u are in S and u itself is not in S. This is for any $0 \leq k \leq d$.
- $\mathrm{dp}[u][> d][0]$: The maximum weight sum obtainable from the subtree rooted at u, where more than d children of u are in S and u itself is not in S.
- $\mathrm{dp}[u][k][1]$: The maximum weight sum obtainable from the subtree rooted at u, where exactly k children of u are in S. u itself is in S. This is for any $0 \leq k \leq d$.

Note that there is no such thing as a $\mathrm{dp}[u][> k][1]$ case. What are the base cases? Again, for any leaf node u we define the base cases to be as follows:

$$\begin{aligned} \forall 0 \leq k : \mathrm{dp}[u][k][0] &= 0 \\ \mathrm{dp}[u][0][1] &= w(u) \\ \forall 1 \leq k : \mathrm{dp}[u][k][1] &= 0 \end{aligned}$$

Now lets look on the non-base cases u. For definition, let's define $B_k(u)$ as the set that contains all k-sized subsets of children of u. And let $C(u)$ be all the children of u. Then for any $0 \leq k \leq d$:

$$\mathrm{dp}[u][k][0] = \max_{B \in B_k(u)} \left(\sum_{v \in B} \left(\max_{0 \leq k' \leq d} \mathrm{dp}[v][k'][1] \right) + \sum_{v \in C(u) \setminus B} \left(\max_{0 \leq k'} \mathrm{dp}[v][k'][0] \right) \right)$$

For that one special case where more than d children are in the optimal solution

$$\mathrm{dp}[u][> d][0] = \sum_{v \in C(u)} \max_{0 \leq k'} \max_{b \in \{0,1\}} \mathrm{dp}[v][k'][b]$$

and if u is in S then:

$$\mathrm{dp}[u][k][1] = w(u) + \max_{B \in B_k(u)} \left(\sum_{v \in B} \left(\max_{0 \leq k' \leq d-1} \mathrm{dp}[v][k'][1] \right) + \sum_{v \in C(u) \setminus B} \left(\max_{0 \leq k'} \mathrm{dp}[v][k'][0] \right) \right)$$

How to implement it efficiently? First note that all these max values like $\max_{0 \leq k' \leq d-1} \mathrm{dp}[v][k'][1]$ are fixed (and already computed) values and hence we know them. To find the best B in the first equation, we compute $\max_{0 \leq k' \leq d} \mathrm{dp}[v][k'][1] - \max_{0 \leq k' \leq d} \mathrm{dp}[v][k'][0]$ for each child v and sort regarding this values. Hence, the best k best values here always give us the best k-elementing subset of the childrens. Hence, for a given u we can compute all the $\mathrm{dp}[u][k][0]$ in time $O(d \cdot deg(v) \cdot \log(deg(v)))$. The same is also true for all the $\mathrm{dp}[u][k][1]$ values of u. Thus, the overall time is as follows:

$$\sum_{v \in V} O(d \cdot deg(v) \cdot \log(deg(v))) \leq O(d \cdot \log n \cdot \sum_{v \in V} deg(v)) = O(d \cdot n \cdot \log n)$$

(c) **Optimized General Solution with Complexity $O(n \log n)$**

Optimize the DP by realizing that we do not need a solution of dp$[u][k][0]$ for every k, as stated above, it is enough if we store the best values smaller d, exactly d and greater d. The same is true for dp$[u][k][1]$. Hence, each node has to compute only $O(1)$ values (i.e., for node u it would be dp$[u][< d][0]$, dp$[u][= d][0]$, dp$[u][> d][0]$, dp$[u][< d][1]$, dp$[u][= d][1]$) instead of the $O(d)$ many as before. By the same analysis as above, we can spare the factor $O(d)$.

Exercise 2: Making binary search dynamic *(10 Points)*

Binary search of a sorted array takes logarithmic search time, but the time to insert a new element is linear in the size of the array. You can improve the time for insertion by keeping several sorted arrays. Specifically, suppose that you wish to support SEARCH and INSERT on a set of n elements. Let $k = \lceil \log_2(n+1) \rceil$, and let the binary representation of n be $\langle n_{k-1}, n_{k-2}, \ldots, n_0 \rangle$. Maintain k sorted arrays $A_0, A_1, \ldots, A_{k-1}$, where for $i = 0, 1, \ldots, k-1$, the length of array A_i is 2^i. Each array is either full or empty, depending on whether $n_i = 1$ or $n_i = 0$, respectively. The total number of elements held in all k arrays is therefore $\sum_{i=0}^{k-1} n_i \cdot 2^i = n$. Although each individual array is sorted, elements in different arrays bear no particular relationship to each other.

(a) Describe how to perform the SEARCH operation for this data structure. Analyze its worst-case running time.

(b) Describe how to perform the INSERT operation. Analyze its worst-case and amortized running times, assuming that the only operations are INSERT and SEARCH.

(c) Describe how to implement DELETE. Analyze its worst-case and amortized running times, assuming that there can be DELETE, INSERT, and SEARCH operations.

Sample Solution

(a) Using binary search, we can search for an element in $O(\log n)$ time within an array. As we have k arrays altogether, the runtime of search is $O(k \cdot \log n) = O(\log^2 n)$.

(b) INSERTION of an element x works as follows: We search for the first array that is empty, say array A_m, then we merge the arrays $A_0, ..., A_{m-1}$ into array A_m (of size 2^m) and add the element x into A_m as well. Then we empty the arrays $A_0, ..., A_{m-1}$. Note that this ensures that A_m is full, while all the previous ones are empty after the operation. The question is how to exactly merge these arrays efficiently. One possibility is the following: We put x into a new Array of size 1 and merge it together with A_0 into a new array of size 2. This new array is now merged with A_2 and so on. Hence, we always merge two arrays of the same size.

- **Worst-Case:** For merging two sorted arrays, we will use the merge step as we know from MergeSort. Hence, merging arrays of size B_1 and B_2 will take time $O(B_1 + B_2)$. Thus, merging all the $A_0, ..., A_{m-1}$ arrays takes the following time:

$$\sum_{i=0}^{m-1} O(2^i) = O(2^m)$$

  Since the worst case is when $m = k = \lceil \log_2(n+1) \rceil$, the worst case time is $O(n)$.

- **Amortized:** Here we show that the amortized cost of an insert operation is in $O(\log n)$. First, say we use the aggregation method. We know from above that merging two layers i and j where $i > j$ takes time $O(i)$. For simplicity let's normalize this costs to costs exactly i. Note that an empty A_0 will be filled by the next insert, an empty A_1 will be filled by 2 inserts, A_3 will be filled by 4 inserts, and so on. Note that this means, that we have merge costs of 2 when we merge A_1. This we have in $n/2$ operations. In $n/4$ of our insert

operations we have to pay costs of 4 as we fill A_2. This proceeds and hence, we pay the following overall costs

$$n + 2 \cdot \frac{n}{2} + 4 \cdot \frac{n}{4} + 8 \cdot \frac{n}{8} + ... + (k-1)\frac{n}{k-1} = k \cdot n$$

Thus, the amortized cost is $\frac{k \cdot n}{n} = k = O(\log n)$.

Alternative with Potential Function Method: We define the potential of the r-th insert as follows:

$$\phi_r := \sum_{A_i \text{is full}} (k-i) \cdot 2^i$$

Note that this means $\phi_0 = 0$ (before the first element got inserted). Before computing the potential difference, note that for $m \geq 1$ the following is always true (can be shown by induction).

$$\sum_{i=0}^{m-1} i \cdot 2^i \leq (m-1) \cdot 2^m$$

We now want to show that for any r, that $a_r = t_r + \phi_r - \phi r - 1$ can be bounded by $O(k)$. We do some case distinction. First case is that A_0 is empty before the insert. Not that this implies that $\phi_r = \phi_{r-1} + k$. Further, by our worst-case observation we have (normalized) costs of $2^0 = 1$. Thus,

$$a_r = 1 + k.$$

In the second case A_0 is full. Let A_m be the insert that is filled with the next element, i.e., $A_0, A_1, ..., A_{m-1}$ are full before the insert. By that, we have

$$\begin{aligned}
\phi_{r-1} &= \sum_{i=0}^{m-1} \left((k-i) \cdot 2^i\right) + \sum_{A_i \text{is full and } i>m} (k-i) \cdot 2^i \\
&= k \cdot \sum_{i=0}^{m-1} 2^i - \sum_{i=0}^{m-1} i \cdot 2^i + \sum_{A_i \text{is full and } i>m} (k-i) \cdot 2^i \\
&\geq k \cdot (2^m - 1) - (m-1) \cdot 2^m + \sum_{A_i \text{is full and } i>m} (k-i) \cdot 2^i \\
&= (k-m) \cdot 2^m - k + 2^m + \sum_{A_i \text{is full and } i>m} (k-i) \cdot 2^i.
\end{aligned}$$

Further, we have

$$\phi_{r-1} = (k-m) \cdot 2^m + \sum_{A_i \text{is full and } i>m} (k-i) \cdot 2^i$$

Like before, we have $t_r = 2^m$. We are now ready to complete our statement:

$$\begin{aligned}
a_r &= t_r + \phi_r - \phi_{r-1} \\
&\leq 2^m + (k-m) \cdot 2^m - ((k-m) \cdot 2^m - k + 2^m) \\
&= 2^m + (k-m) \cdot 2^m - (k-m) \cdot 2^m + k - 2^m \\
&= k
\end{aligned}$$

So we have seen that in both cases $a_r = O(k) = O(\log n)$.

(c) Assume we want to delete an element x. First, we search x in $O(\log n)$ time (as described in the first task) and let A_m be the array containing x. Let i be the smallest index where A_i is full. If $i = m$, we delete x and distribute the remaining elements into $A_0, A_1, ..., A_{i-1}$ (as they are empty). If $m > i$, we delete x in A_m and replace it with an arbitrary element from A_i. Note that the replacement could take $O(n)$ time in worst-case, as we may have to shift the element to a position that makes the array sorted. Thus, the worst case here is $O(n)$.

Exercise 3: Problem for the exercise session *(0 Points)*

Your plan to implement a `Stack` with the classical operations `push`, `pop` and `peek`. As underlying data structure you use a dynamic array that will grow its size whenever 'many' elements are stored and on the other hand also shrinks its size when only a view elements remain in the array. In the following let n_i be the number of elements stored in the array and let s_i be the size of the array after the i-th operation.

- Before you **push** a new element x to the array, you check if $n_{i-1} + 1 < 80\% \cdot s_{i-1}$. If this is the case then you simply add x. We say for simplicity, that this can be done in 1 time unit. If on the other hand $n_{i-1} + 1 \geq 80\% \cdot s_{i-1}$, you set up a new (empty) array of size $s_i := 2s_{i-1}$ and copy all elements (and x) into the new one. We assume this can be done in s_{i-1} time units[1].
- To **pop** an element from the array, you first check if $n_{i-1} - 1 > 20\% \cdot s_{i-1}$. If this is the case then pop x within 1 time unit. If the table size is small, say $s_{i-1} \leq 8$, you also just pop x. But, if $n_{i-1} - 1 \leq 20\% \cdot s_{i-1}$ and $s_{i-1} > 8$, create a new (empty) array of size $s_i := s_{i-1}/2$ and copy all values except x into this new array. By assumption, this step takes s_i time units.
- The **peek** operation returns the last inserted element in 1 time unit. Note that state of the array does not change, i.e., $n_{i-1} = n_i$ and $s_{i-1} = s_i$.

Initially, the array is of size $s_0 = 8$. Assume that this initial step can also be done in 1 time unit. Note that by this initial size and the definition of the pop method we have $s_i \geq 8$ for all $i \geq 0$. Also note that after every operation that resized the array at least one element can be pushed or popped until a further resize is required.

a) Let i be a push operation that resized the array. Show that the following holds.

$$0.4 \cdot s_i \leq n_i < 0.55 \cdot s_i$$

Further, show that if i is a pop operation that resized the array, the following holds.

$$0.25 \cdot s_i < n_i \leq 0.4 \cdot s_i$$

b) Use the `Accounting Method` from the lecture to show that the **amortized running times** of push, pop and peek are $O(1)$, i.e., state by how much you additionally charge these three operation and show that the costs you spare on 'the bank' are enough to pay for the costly operations.
Hint: Use the previous subtask, even if you didn't manage to show them.

c) Show the same statement as in the previous task, but use the `Potential Function Method` this time, i.e., find a potential function $\phi(n_i, s_i)$ and show that this function is sufficient to achieve constant amortized time for the supported operations.
Hint: There is not just one but infinitely many potential functions that work here. However, you may want to use a function of the form $c_0 \cdot |n_i - c_1 \cdot s_i|$ for some properly chosen constants $c_0 > 0$ and $c_1 > 0$.

[1] For a simpler calculation we use normalized time units, such that all the operations that would take $O(1)$ time will take at most 1 time unit and operations that would take $O(s_{i-1})$ time will take at most s_i time units.

Sample Solution

a) **Push:** It is clear that for the previous state $n_{i-1} < 0.8s_{i-1}$ is true (otherwise there would have been a resize before) and by definition also $n_{i-1} + 1 \geq 0.8s_i$ holds. Since $n_i = n_{i-1} + 1$ and $s_i = 2s_{i-1}$ we directly get $0.4 \cdot s_i \leq n_i < 0.4 \cdot s_i + 1 \leq 0.525 \cdot s_i$ (because $s_i \geq 8$). This implies the statement.
Pop: For similar reasons as before, we have $0.2 \cdot s_{i-1} < n_{i-1} \leq 0.2 \cdot s_{i-1} + 1$. Substituting $n_i = n_{i-1} - 1$ and $s_i = s_{i-1}/2$ we get $0.4 \cdot s_i < n_i + 1 \leq 0.4 \cdot s_i + 1$. By subtracting all sides by 1 and use that $1 \leq s_i/8$ we get $0.275 \cdot s_i < n_i \leq 0.4 \cdot s_i$. This implies the statement.

b) We charge all 3 operations by 25 'dollars'. Every push, pop or peek operation costs one actual dollar (not counting resizing) and puts the remaining 24 in the bank to pay for resizing. Now, let us estimate how much money is **at least** in the bank before the next resizing takes place. Observe that we can ignore peek operations in the following, since they just increase our bank account (by 24 per operation) and do not change the state of the array. Let us for now assume that the last operation (say operation i) did a resizing and currently there is no money at our account. If this last resize operation was 'push', then we have $0.4 \cdot s_i \leq n_i < 0.55 \cdot s_i$. Thus, the next costly operation can not happen before $(0.8 - 0.55)s_i = 0.25s_i$ push or $(0.4 - 0.2) \cdot s_i = 0.2s_i$ pop operations. Before we proceed, let us see how it looks if operation i was a pop operation. By the statement of the previous task we have the next costly operation not before $(0.8 - 0.4) \cdot s_i = 0.4s_i$ push or $(0.25 - 0.2) \cdot s_i = 0.05 \cdot s_i$ pop operations. By this analysis, the *worst-case* (i.e., the shortest chain of operation until the next resize) is a costly pop operation that is followed by more than $0.05 \cdot s_i$ additional deletings. Since we charge both operation with the same amortized cost, we clearly have more 'money' on our account in the other cases. Also observe that if we alternate between pushing and deleting over and over, the hash table is never resized, so we save up a lot of money in the bank. For that, assume operation i is a pop operation and from here on at least $0.05 \cdot s_i$ further pop operations follow until the next resize comes. The money on the bank after these many operations is at least $(25 - 1) \cdot 0.05 \cdot s_i = 1.2 \cdot s_i$. Because the costly operation costs $1 \cdot s_i$, we can afford it and thus, the bank account never drops below zero. Therefore, the amortized cost of pop is $O(1)$. This, by above's analysis, also implies amortized costs of $O(1)$ for push (as well as for peek).

c) We define our potential function by $\phi(n_i, s_i) := c_0 \cdot |n_i - c_1 \cdot s_i|$ and start by guessing some constant $c_1 \in [0, 1]$. For some intuition: We want our potential to be large before a resizing operation and small (close to zero) after a resizing operation. To make sure that the potential before resizing is not 0, we have to choose $c_1 \neq 0.8$ in the push case and $c_1 \neq 0.2$ in the pop case. So let us 'guess' $c_1 = 0.4$ and show later that it works. Now we go through all operations and proof that by choosing a large enough $c_0 > 0$, all amortized costs are in $O(1)$. Note that we are going into 5 cases now, that we call `peek`, `cheap push` (push without resizing), `costly push` (includes resizing), `cheap pop` (deleting without resizing) and `costly pop` (includes resizing). Like in the lecture, we notate the actual cost of operation i by t_i and its amortized costs by $a_i := t_i + \phi(n_i, s_i) - \phi(n_{i-1}, s_{i-1})$.

Peek Here we have $s_i = s_{i-1}$ and $n_i = n_{i-1}$ and thus,

$$\begin{aligned} a_i &= 1 + c_0 \cdot |n_i - 0.4s_i| - c_0 \cdot |n_{i-1} - 0.4s_{i-1}| \\ &= 1 + c_0 \cdot |n_i - 0.4s_i| - c_0 \cdot |n_i - 0.4s_i| \\ &= 1 \end{aligned}$$

Cheap Push Here we have $n_i = 1 + n_{i-1}$ and $s_i = s_{i-1}$

$$\begin{aligned} a_i &= 1 + c_0 \cdot |n_i - 0.4s_i| - c_0 \cdot |n_{i-1} - 0.4s_{i-1}| \\ &= 1 + c_0 \cdot |n_i - 0.4s_i| - c_0 \cdot |n_i - 1 - 0.4s_i| \\ &\leq 1 + c_0 \cdot |n_i - 0.4s_i| - c_0 \cdot (|n_i - 0.4s_i| - 1) \\ &\leq 1 + c_0 \end{aligned}$$

Note that this implies that $a_i \in O(1)$ if c_0 is a fixed constant.

Costly Push Here we have $t_i = s_{i-1}$, $s_i = 2s_{i-1}$, $n_i = n_{i-1} + 1$ and by the first subtask $0.55s_i > n_i \geq 0.4s_i$

$$\begin{aligned} a_i &= s_{i-1} + c_0 \cdot |n_i - 0.4s_i| - c_0 \cdot |n_{i-1} - 0.4s_{i-1}| \\ &= \frac{s_i}{2} + c_0 \cdot |n_i - 0.4s_i| - c_0 \cdot (|n_i - 0.2s_i| - 1) \\ &\leq \frac{s_i}{2} + c_0 \cdot (n_i - 0.4s_i) - c_0 \cdot (n_i - 0.2s_i) + c_0 \\ &\leq \frac{s_i}{2} - \frac{1}{5} \cdot c_0 \cdot s_i + c_0 \\ &\leq c_0 \end{aligned}$$

That the last step only follows if we choose $c_0 \geq \frac{5}{2}$.

Cheap Pop Here we have $n_i = -1 + n_{i-1}$ and $s_i = s_{i-1}$

$$\begin{aligned} a_i &= 1 + c_0 \cdot |n_i - 0.4s_i| - c_0 \cdot |n_{i-1} - 0.4s_{i-1}| \\ &= 1 + c_0 \cdot |n_i - 0.4s_i| - c_0 \cdot |n_i + 1 - 0.4s_i| \\ &\leq 1 + c_0 \cdot |n_i - 0.4s_i| - c_0 \cdot |n_i - 0.4s_i| + c_0 \\ &\leq 1 + c_0 \end{aligned}$$

Costly Pop Here we have $t_i = s_{i-1}$, $s_i = s_{i-1}/2$, $n_i = n_{i-1} - 1$ and by the first subtask $n_i > 0.25s_i$ and $n_i \leq 0.4s_i$.

$$\begin{aligned} a_i &= s_{i-1} + c_0 \cdot |n_i - 0.4s_i| - c_0 \cdot |n_{i-1} - 0.4s_{i-1}| \\ &= 2s_i + c_0 \cdot |n_i - 0.4s_i| - c_0 \cdot |n_i + 1 - 0.8s_i| \\ &\leq 2s_i + c_0 \cdot (0.4s_i - n_i) - c_0 \cdot (0.8s_i - n_i) + c_0 \\ &= 2s_i - \frac{2}{5} \cdot c_0 \cdot s_i + c_0 \\ &\leq c_0 \end{aligned}$$

That the last step only follows if we choose $c_0 \geq 5$.

Final Statement It is clear by the previous calculations that if we choose $c_0 := 5$ the amortized costs for all 3 operations are at most $a_i \leq 1 + c_0 = 6$ and therefore constant. The potential function used is

$$\phi(n_i, s_i) := 5 \cdot |n_i - \frac{2}{5} \cdot s_i| = |2 \cdot s_i - 5 \cdot n_i| \geq 0$$

Remark: We have that $\phi(n_0, s_0) = 5 \cdot |0 - \frac{2}{5} \cdot 8| = 16$ and hence $\sum_i t_i \leq 16 + \sum_i a_i$. This does not completely match with the definition of amortized costs, however we can fix this problem by choosing the potential function $\phi'(n_i, s_i) := 5 \cdot |n_i - \frac{2}{5} \cdot s_i - \frac{16}{5}|$. Here we have $\phi'(n_0, s_0) = 0$. Further we have that for all i, $|\phi'(n_i, s_i) - \phi(n_i, s_i)| \leq 16$ holds, and thus we can simply adjust the 5 cases and show that for each operation it follows $a_i \leq t_i + \phi(n_i, s_i) - \phi(n_{i-1}, s_{i-1}) + 2 \cdot 16 \leq 6 + 32 = 38$.