Albert-Ludwigs-Universität, Institut für Informatik
Prof. Dr. Fabian Kuhn
M. Ahmadi, P. Schneider, J. Uitto

December 8, 2017

Algorithm Theory - Winter Term 2017/2018
Exercise Sheet 3

Hand in by Thursday 10:15, November 30, 2017

Exercise 1: Dynamic Programming - MaxIS (5+5 Points)

Let $G = (V, E)$ be a graph. A set of nodes $I \subseteq V$ is called *independent* if for all $u, v \in I$ it holds that $\{u, v\} \notin E$. In other words, *no* two nodes in the independent set I are adjacent. A *Maximum* Independent Set (MaxIS) is an independent set with maximum cardinality (number of nodes in I).

(a) Devise an *efficient*[1] algorithm that uses the principle of dynamic programming and finds a maximum independent set in a rooted tree[2].

(b) Prove that your algorithm is correct, i.e. returns a maximum independent set and prove that it has the claimed runtime.

Sample Solution

(a) Consider an arbitrary tree $T = (V, E)$ rooted at t. We can base our decision whether a node should be in the MaxIS on the size of the MaxIS of its subtrees. Let $s(v)$ be the size of the maximum independent set of the subtree rooted at v. Let $C(v)$ be the set of children of node $v \in V$. If we know the sizes of the MaxIS of v's children and grandchildren we can compute $s(v)$ recursively as follows:

$$s(v) = \max \left\{ \sum_{u \in C(v)} s(u), \;\; 1 + \sum_{u \in C(v)} \sum_{w \in C(u)} s(w) \right\}$$

Algorithm 1 is a recursive algorithm calculating the MaxIS of a tree. It is computing $s(v)$ for all the nodes in V, and based on the these calculated values determines the MaxIS. For any $v \in V$, as soon as $s(v)$ is calculated, it is stored in a memo to avoid the unnecessary recalculations.

(b) **Correctness:** The resulting set is `maxIS`. We have to show two properties: That `maxIS` is independent and has maximum cardinality among all independent sets.

Claim: *For any subtree $T' = (V', E')$, $V' \cap$ `maxIS` is a maximum independent set of T' of size $s(v)$ (v is the root of T'), where `maxIS` is the node set calculated by Algorithm 1.*

Proof. We show this with induction on the structure of the tree. **Induction base:** For a leaf v we have $g = 1$ and $c = 0$ (note that we define a sum over an empty set as 0) which is therefore added to `maxIS`. Then `maxIS` is independent and has maximum cardinality $s(v) = 1$.

Induction hypothesis: Assume the claim is true for all subtrees *attached to* $v \in V$. Let $T' = (V', E')$ be the subtree rooted at v.

[1] An algorithm is efficient if it has a runtime of $\mathcal{O}(p(n))$, where $p(n)$ is a polynomial and n the size of the input.
[2] The input graph is connected, has no cycles, has a dedicated root node, and each node knows its parent and children.

Algorithm 1: `TreeMaxIS`(v, T) (We consider integer array `memo`, set `maxIS` to be global)

if `memo`$[v] \neq \perp$ **then** `// If` $s(v)$ `is already calculated`
  return `memo`$[v]$

$c \leftarrow \sum_{u \in C(v)}$`TreeMaxIS`$(u, T)$ `// Size of MaxIS without` v
$g \leftarrow 1 + \sum_{u \in C(v)} \sum_{w \in C(u)}$`TreeMaxIS`$(w, T)$ `// Size of MaxIS with` v

`// Base cases:` v `has no (grand-)children covered by defining` $\sum_{\emptyset}(\cdot) := 0$

if $g > c$ **then**
  add v to `maxIS`

`memo`$[v] \leftarrow \max\{c, g\}$

return `memo`$[v]$

Induction step: First we show that $V' \cap$ `maxIS` is independent in T'. Due to the induction hypothesis we have independence if $v \notin$ `maxIS`. So let us consider $v \in$ `maxIS`. If v joined `maxIS` we have $g > c$. Now the independence of $V' \cap$ `maxIS` is only violated if at least one child of v is in `maxIS`. For a contradiction assume this is the case. A child of v joined `maxIS` only if we obtain a bigger MaxIS by adding it instead of its children (v's grandchildren), i.e., condition $s(u) > \sum_{w \in C(u)} s(w)$ applies for at least one child u of v. Thus

$$\sum_{u \in C(v)} s(u) > \sum_{u \in C(v)} \sum_{w \in C(u)} s(w)$$
$$\iff \quad c = \sum_{u \in C(v)} s(u) \geq \sum_{u \in C(v)} \sum_{w \in C(u)} s(w) + 1 = g$$

a contradiction to $g > c$.

Moreover, since we pick as $s(v)$ the maximum of $\sum_{u \in C(v)} s(u)$ and $1 + \sum_{u \in C(v)} \sum_{w \in C(u)} s(w)$ we ensure that $s(v)$ is the size of a maximum independent set (induction hypothesis for children and grandchildren applies, thus we calculated $s(u), s(w)$ in the formula correctly). Since we select `maxIS` in accordance with the choice we make to calculate $s(v)$, we have $s(v) = |V' \cap$ `maxIS`$|$, thus `maxIS` is maximum in T'. □

Runtime: First consider the number of calls of `TreeMaxIS`. `TreeMaxIS`(v) is called only by v's parent and grandparent. Moreover, `TreeMaxIS`(v) is called *at most once* by its parent and grandparent, because after the first call the size of their MaxIS is memorized and they will never make recursive calls again. Thus `TreeMaxIS` is called at most $2|V| = \mathcal{O}(|V|)$ times in total.

Besides recursion, the costliest operations that remain in `TreeMaxIS` are the summations done in `TreeMaxIS`. So each node occurs in a sum at most twice, once due to its parent and once due to its grandparent. Therefore the overall number of summations we do in the calls of `TreeMaxIS` is $2|V| = \mathcal{O}(|V|)$. This gives us a linear runtime in the input size.

Exercise 2: Worst Case Analysis - Fibonacci Heaps (3+7 Points)

Fibonacci heaps are efficient in an *amortized* sense. However, the time to execute a *single* operation can be quite large. Show that in the worst case, (a) the `delete-min` operation and (b) the `decrease-key` operation can require time $\Omega(n)$ *for an arbitrary* n.

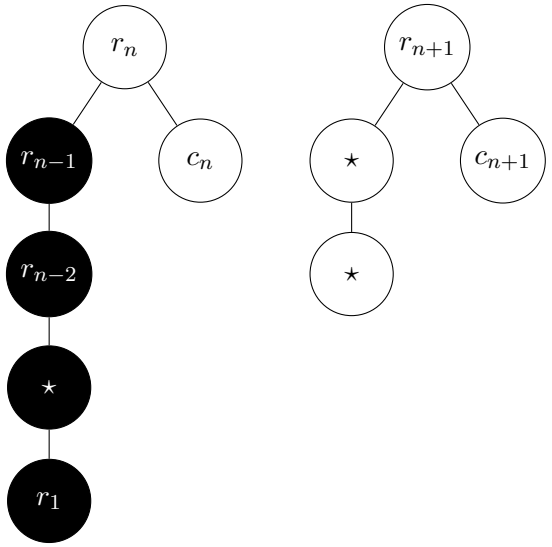
Hint: Describe a valid scenario where the `delete-min` *or* `decrease-key` *operation respectively requires at least linear time in* n *for an arbitrary* n.

Sample Solution

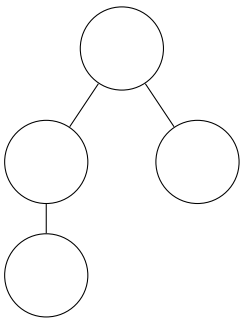
(a) **A costly *delete-min***: First n elements are added to the heap, which causes them all to be roots in the root list. Deleting the minimum causes a *consolidate* call, which combines the remaining $n-1$ elements, which need at least $n-2$ *merge* operations, i.e., it costs $\Omega(n)$ time.

(b) **A costly *decrease-key*:** We construct a degenerated tree T_n with the root r_n, that has two children r_{n-1} and c_n. Node c_n is unmarked and r_{n-1} is marked. The latter has a single child r_{n-2} that is also marked and has a single child r_{n-3} and so on, until leaf r_1 (which may be marked or unmarked). In other words, T_n consists of a line of marked nodes, plus the root and one further unmarked child of the root. Let k_n be the key of the root r_n.

For the purpose of an inductive construction procedure, we show how to create a T_{n+1} from a T_n. First we add add another 5 nodes to the heap and delete the minimum of them, causing a *consolidate*. In more detail let us add a node r_{n+1} with key $k_{n+1} \in (0, k_n)$, a node with key 0 and three nodes with keys $k' > k_{n+1}$. When we delete the minimum, first both pairs of singletons are combined to two trees of rank 1, which are combined again to one binomial tree of rank 2, with the node r_{n+1} as the root and we name its childless child c_{n+1} (confer the picture for the current state).



Since also T_n has rank 2 we now combine it with the new tree and r_{n+1} becomes the new root. We now decrease the key of c_n to 0 as well as the keys of the two ($\star$)-nodes and delete the minimum after each such operation. Note that a root node will not get marked or cut! Decreasing the key of c_n, however, will now mark its parent r_n, as it is not a root anymore. Thus the remaining heap is of exactly the same shape as T_n, except that its depth did increase by one: a T_{n+1}.



Using the same technique as before we obtain the tree above. We cut off the lowest leaf and now have a T_1. Together with the construction procedure above the rest follows via induction. Obviously, a *decrease-key* operation on r_1 will cause a cascade of $\Omega(n)$ cuts if applied to a heap consisting of such a T_n.

Exercise 3: Amortized Analysis - Counting (4+6 Points)

Consider non-negative integers in their canonical bit-representation. Similarly to the lecture, we execute n increment operations (add 1) starting from 0. For the analysis, we add a little twist. Flipping the i^{th} bit b_i[3] now has a cost 2^i.

(a) Show that the amortized cost is super-constant (i.e. in $\omega(1)$).

(b) Show that the amortized cost is $\mathcal{O}(\log n)$.

Sample Solution

(a) When counting up to $n-1$, bit i is flipped at least 2^{k-i} times. E.g., the least significant bit is flipped n times incurring a cost of n in total (not amortized). Assume now that the amortized cost is at most c for some constant c (i.e., $\mathcal{O}(1)$). Let n be a positive integer such that $\log n =: k > c$ for some constant $k \in \mathbb{N}_0$. Now, we get that the total cost of counting up to n is at least

$$2^k \cdot 2^0 + 2^{k-1} \cdot 2^1 + 2^{k-2} \cdot 2^2 + \ldots + 2^0 \cdot 2^k = k \cdot 2^k = n \log n > c \cdot n$$

implying that the amortized cost (i.e. divided by number of increment operations n) is bigger than c, a contradiction.

(b) The solution is a modification of the accounting argument from the lecture. *First*, we observe that in each increment operation there occurs at most one bit-flip from 0 to 1 (since carry-overs occur only when bits are flipped from 1 to 0). *Second*, we observe that when bit b_i is flipped from **0 to 1 and back to 0 again**, the lesser significant bit b_{i-k} was flipped from **0 to 1** for 2^k times before (e.g. apply an inductive argument). What we do is this:

- Whenever a bit is flipped **from 0 to 1**, we pay in total $2\lceil \log n \rceil$ to the bank spread over separate accounts (where $\lceil \log n \rceil$ is the number of bits that are needed to count to $n-1$). Specifically, we pay 2 coins on each account a_i for $i \in \{1, \ldots, \lceil \log n \rceil\}$.
- Moreover, when bit b_i is flipped **in either direction** we subtract 2^i from account a_i to pay for that flip.

Based on our observations, we show that every account a_i stays balanced. By the second observation, after we flipped b_i from 0 to 1 back to 0, each of the lower bits b_{i-k} has been paying 2 coins to a_i for 2^k times. This means that together with the two coins that b_i itself was paying on its own account, we have that after we flipped b_i from 0 to 1 back to 0 we already paid

$$2 \cdot \sum_{i=0}^{k} 2^i = 2 \cdot 2^{k+1} - 1 \geq 2 \cdot 2^k$$

to a_i which suffices to pay for both flips of b_i from 0 to 1 back to 0. Since there is only one flip from 0 to 1 per increment operation, we are paying only $2\lceil \log n \rceil \in \mathcal{O}(\log n)$ to the bank per increment, thus proving the claim.

Exercise 4: Amortized Analysis - Multisets (10 Points)

Let S be a multiset of integers (a set which allows duplicate values). We would like to change S as follows. Either we insert a new element into S, or we delete the $\lceil |S|/2 \rceil$ largest elements from S (in case of removing some but not all the duplicates of the same element, we break the tie arbitrarily). Design a data structure that supports two operations `Insert`(S, x) and `DeleteLargerHalf`(S) such that any sequence of m operations starting from an empty multiset takes $O(m)$ time. Prove the correctness and amortized runtime of your implementation of `Insert`(S, x) and `DeleteLargerHalf`(S).

Hint: There is an algorithm to find the median of a multiset of n integers in $O(n)$ time. You can use it as a blackbox.

[3] Bit b_i is the bit in the i^{th} position ascending from the least significant bit.

Sample Solution

Regarding the data structure, we consider a doubly linked list. Let us now study the actual cost of the two operations. Adding an element into a doubly linked list has a constant cost by just adding the element to the head of the list. For removing the larger half of k elements, we can calculate the mean in time $O(k)$, and then for removing the the largest $\lceil k/2 \rceil$ elements, we scan the doubly linked list twice as follows. In the first scan, we remove all the elements smaller than the mean. In the second scan, we remove the elements which are equal to the mean as long as we removed less than $\lceil k/2 \rceil$ elements. Note that removing an element from a doubly linked costs constant by changing a constant number of pointers.
Hence, the actual cost for the `Insert(S,x)` operation is constant and for the `DeleteLargerHalf(S)` operation is linear in the size of S. Now let us show that the amortized cost of each of these operations in a sequence of operations starting at an empty doubly linked list is constant. Let $\Phi = 2N$ be the potential function when N is the current number of elements in the doubly linked list. Clearly the potential function cannot be negative as the size of the data structure is always non-negative.
For the analysis, we normalize the cost of `Insert(S,x)` and `DeleteLargerHalf(S)` to 1 and $|S|$ respectively (c.f. lecture). Fix an arbitrary sequence of operations. Then let N_{i-1} denote the number of elements in S before the i^{th} operation and N_i the number of elements after the i^{th} operation [4]. In the following a_i and t_i are amortized and actual costs, respectively, for operation i. For the i^{th} operation we consider the cases that it is an `Insert` or a `DeleteLargerHalf` operation separately:

If the i^{th} operation is `Insert(S,x)`, then:

$$
\begin{aligned}
N_i &= N_{i-1} + 1. \\
t_i &= 1 \text{ (the actual cost of operation } \texttt{Insert(S,x)}\text{)}. \\
a_i &= t_i + \Phi_i - \Phi_{i-1} = 1 + 2N_{i-1} + 2 - 2N_{i-1} = 3 \in \mathcal{O}(1).
\end{aligned}
$$

If the i^{th} operation is `DeleteLargerHalf(S)`, then:

$$
\begin{aligned}
N_i &= N_{i-1} - \lceil N_{i-1}/2 \rceil. \\
t_i &= N_{i-1} \text{ (the actual cost of operation } \texttt{DeleteLargerHalf(S)}\text{) as given in the exercise.} \\
a_i &= t_i + \Phi_i - \Phi_{i-1} = N_{i-1} + (2N_{i-1} - 2\lceil N_{i-1}/2 \rceil) - 2N_{i-1} = N_{i-1} - 2\lceil N_{i-1}/2 \rceil \leq 0 \in \mathcal{O}(1).
\end{aligned}
$$

Hence, the amortized costs for both operations `Insert(S,x)` and `DeleteLargerHalf(S)` are constant.

[4] N_i depends on the type of operation that is performed in step i. Thus N_i differs in the two considered cases