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

January 12, 2018

Algorithm Theory - Winter Term 2017/2018
Exercise Sheet 4

Hand in by Thursday 10:15, December 14, 2017

Exercise 1: Data Structures - Heapsort (2+2+2+2+2 Points)

We can employ a binary min-heap to sort a sequence of n distinct keys in an ascending order as follows. First, we insert all keys one by one with the `Insert` operation. Then, we obtain the sorted sequence by removing all n keys with `Delete-Min`.

(a) Show that `Heapsort` takes $\Theta(n \log n)$ if the input sequence is sorted *descending*.

(b) Show that `Heapsort` takes $\Theta(n \log n)$ if the input sequence is sorted *ascending*.

Assume that you have two binary min-heaps that only allow manipulation via the operations `Get-Min`, `Insert`, `Delete-Min`, `Decrease-Key`, `Get-Size` and `Is-Empty` and may have duplicate keys.

(c) Give a protocol that merges two heaps of size n and m, where $m < n$, in time $\mathcal{O}(m \log n)$ and prove the runtime.

(d) For arbitrary m and n, where $m < n$, give two binary min-heaps of size m and n where the `Merge` protocol takes $\Omega(m \log n)$.

(e) For arbitrary m and n, where $m < n$, give two binary min-heaps of size m and n where the `Merge` protocol takes $\mathcal{O}(m)$.

Sample Solution

From the lecture we know that the run-time of `Insert` and `Delete-Min` is upper bounded by $\mathcal{O}(\log n)$ (in a tree of size n). This gives an upper bound of $\mathcal{O}(n \log n)$ for `Heapsort`.

(a) Consider the input-sequence $(n, n–1, \ldots, 1)$. It remains to be shown that the run-time of `Heapsort` with this input sequence is lower bounded by $\Omega(n \log n)$.

Inserting these keys into the heap from left to right always violates the min-heap rule and the new key has to be moved to the root of the heap since it is the smallest at the time it is inserted.

Moving an element from a leaf to the root takes $\Omega(\log(\text{treesize}))$ many operations, since the new element swaps positions with its parent as long as it is smaller. A binary heap of size n has $\Omega(n/2)$ leaves and $\Omega(n/2)$ non-leaf nodes.

Since the last $\lfloor \frac{n}{2} \rfloor$ keys are inserted into a tree of size at least $\Omega(n/2)$, moving those elements to the root takes at $\Omega(\frac{n}{2} \log(\frac{n}{2})) = \Omega(n \log n)$ time, and we have our claim.

(b) Consider the input-sequence $(1, \ldots, n-1, n)$. We show that the run-time of `Heapsort` with this input sequence is lower bounded by $\Omega(n \log n)$.

Since the input sequence is already sorted, we never have to fix the heap when we insert the keys (in fact we get a run-time of $\mathcal{O}(n)$ to insert all elements).

Due to the given input sequence the $\lceil \frac{n}{2} \rceil$ leaves of the complete binary tree[1] contain the biggest keys of the input sequence. This structure makes `Delete-Min` costly.

To delete an element we remove the root, return it, then we cut the *'last'* element[2] and transfer it to the root instead. Then we fix the heap by *'sifting'* the root *'down'* as long as it violates the heap property.

Starting from the heap containing the complete sequence $(1, \ldots, n-1, n)$, the first $\lceil \frac{n}{2} \rceil$ `Delete-Min` operations move one of the $\lceil \frac{n}{2} \rceil$ biggest keys from a leaf to the root.

In order to repair the tree after each deletion the new root has to be *'sifted down'* to a leaf again by swapping it $\Omega(\log \frac{n}{2})$ times with the smallest of its current children. Therefore the first $\lceil \frac{n}{2} \rceil$ deletions take $\Omega(\lceil \frac{n}{2} \rceil \log \frac{n}{2}) = \Omega(n \log n)$ time.

(c) The following procedure merges two binary heaps.

Algorithm 1 `Merge`(H_1, H_2) ▷ *wlog we assume* $|H_1| > |H_2|$ *(check is easy with* `Get-Size`*)*

```
while not H2.Is-Empty do
    H1.Insert(H2.Delete-Min)        ▷ insert all elements from H2 into H1
return H1
```

Let $n := |H_1|$ and $m := |H_2|$. We do m deletions in H_2 for at most $\mathcal{O}(m \log m) \subseteq \mathcal{O}(m \log n)$ time. We insert the m deleted values into H_1, which takes $\mathcal{O}(m \log(m+n)) \subseteq \mathcal{O}(m \log(2n)) = \mathcal{O}(m \log n)$.

Remark: A min-heap can be implemented such that `Get-Size` *and* `Is-Empty` *take* $\mathcal{O}(1)$ *time.*

(d) Let H_1, H_2 be the heaps to be merged and let $n := |H_1|$, $m := |H_2|$. We give a worst case example for arbitrary $m < n$ to show the worst case lower bound.

Let the smaller heap H_2 contain small keys $(1, \ldots, m)$ and let the larger heap H_1 contain larger keys $(m+1, \ldots, n+m+1)$. Observe that this structure makes `Insert` costly. In specific inserting all keys of in H_2 into H_1 costs $\Omega(m \log(n))$ (cf. (a)).

(e) Presume that all the keys in the smaller heap are equal but larger than all the keys in the larger heap. We perform the merge with m `Delete-Min` operations on the smaller heap and m `Insert` operations on the larger. Notice that deleting from the smaller heap is $\mathcal{O}(1)$ since we don't have to fix the heap after deleting an element since all are equal. Furthermore inserting into the larger heap is in $\mathcal{O}(1)$, since we do not have to fix the heap. This leads into a runtime of $O(m)$.

Exercise 2: Union-Find (5+5 Points)

(a) In the lecture the union-by-size heuristic was introduced to guarantee shallow trees when implementing a Union-Find data structure. Another heuristic that can be used for `union(x,y)` is the union-by-rank heuristic. For the heuristic, the rank of a tree is defined as follows:

(1) The rank $r(T)$ of a tree T consisting of only one node is 0.

(2) When joining trees T_1 and T_2 by attaching the root of tree T_2 as a new child of the root of tree T_1, the rank of the new combined tree T is defined as $r(T) := \max\{r(T_1), r(T_2) + 1\}$.

[1] In a complete bin. tree all layers except the last have max. number of nodes. Nodes in the last layer are aligned left.
[2] by order of going from top layer to bottom layer of the tree and from left to right within layers.

When applying the union-by-rank heuristic, whenever combining two trees into one tree (as the result of a union operation), we attach the tree of smaller rank to the tree of larger rank (if both trees have the same rank, it does not matter which tree is attached to the other tree). Provide pseudo-code for the `union(x,y)` operation when using the union-by-rank heuristic.

Show that when implementing a Union-Find data structure by using disjoint-set forests with the union-by-rank heuristic, the height of each tree is at most $\mathcal{O}(\log n)$.

(b) Demonstrate that the above analysis is tight by giving an example execution (of merging n elements in that data structure) that creates a tree of height $\Theta(\log n)$. Can you even get a tree of height $\lfloor \log_2 n \rfloor$?

Sample Solution

(a)

Algorithm 2 `union(x,y)` ▷ *Assert that* `find(x)` $\neq$ `find(y)`

```
r ← find(x), p ← find(y)                    ▷ Get representatives of x, y
if r.rank ≥ p.rank then
    p.parent ← r
    r.rank ← max{r.rank, p.rank+1}
    return r
else
    r.parent ← p
    p.rank ← max{p.rank, r.rank+1}
    return p
```

Remark: As in the lecture n is the number of `make_set` operations, i.e. the number of 'nodes'.

The claim that any given tree in the forest has rank at most $\mathcal{O}(\log n)$, is obviously true if we can show that a tree with k nodes has rank at most $\lfloor \log_2 k \rfloor$. We show this via induction on k.

Induction Base: For $k = 1$ we have a single root node with rank $0 = \lfloor \log_2 1 \rfloor$.

Induction Hypothesis: Presume the claim is true for all trees with number of nodes $< k$.

Induction Step: Now let T be a tree with k nodes. The tree T was created with a `union`(T_1, T_2) operation where the trees T_1, T_2 have $k_1, k_2 < k$ many nodes. W.l.o.g. let $k_1 \geq k_2$.

If $r(T_1) > r(T_2)$ then $r(T) = r(T_1) \leq \lfloor \log_2 k_1 \rfloor \leq \lfloor \log_2 k \rfloor$.

If $r(T_1) < r(T_2)$ then $r(T) = r(T_2) \overset{IH}{\leq} \lfloor \log_2 k_2 \rfloor \leq \lfloor \log_2 k \rfloor$.

If $r(T_1) = r(T_2)$ then

$$r(T) = r(T_1)+1 = r(T_2)+1 \overset{IH}{\leq} \lfloor \log_2 k_2 \rfloor + 1 \leq \lfloor \log_2 2k_2 \rfloor \overset{k=k_1+k_2 \geq 2k_2}{\leq} \lfloor \log_2 k \rfloor.$$

Remark: Another possible solution is to argue that the rank equals the height of the tree and use the fact that 'most of' the inner nodes have at least two children to upper bound the height.

(b) We construct a tree from n single node trees (created with `make_set`) that has height (rank) $\lfloor \log_2 n \rfloor$ using the `union` operation. First we do the construction under the assumption $n = 2^k$ for some $k \in \mathbb{N}_0$ and then we show how to construct a tree of height $\lfloor \log_2 n \rfloor$ for arbitrary n.

For $n = 2^k$ we do the following iteratively until just one tree is left: We merge all remaining roots from the previous iteration (initially single-node trees) *pairwise* with `union`. Note that we call union only on the roots, so no path compression takes place! Since n is a power of two, with each iteration the number of trees in the Union-Find structure halves hence we have k iterations. In each iteration the trees are symmetric (same number of nodes and same topology). Therefore the rank of the trees increases by one in each iteration. Thus the rank of the remaining tree after k iterations is $k = \log_2 n$.

For arbitrary n let $k \in \mathbb{N}_0$ such that $2^k \leq n < 2^{k+1}$ (the biggest power of two smaller equal n). Now we do the same procedure as above for only 2^k of the n nodes to obtain a single tree with

2^k nodes that has rank k. Then we use the `union` operation to merge this tree directly with the $n - 2^k$ nodes left over (if any). The rank of these single nodes is smaller than the rank k of the tree we constructed before (unless $n = 1$ in which case we don't have leftover nodes). Thus the rank of the bigger tree remains the same when we merge it with the single nodes, i.e. it still has rank k. Since $k = \log_2 2^k \leq \log_2 n < \log_2 2^{k+1} = k + 1$ we have $\lfloor \log_2 n \rfloor = k$.

We have shown that there exist trees for the Union-Find data structure (by rank variant) with rank $\lfloor \log_2 n \rfloor$. Additionally we saw in part (a) that any given tree of our Union-Find structure has rank smaller equal $\lfloor \log_2 n \rfloor$. Hence the worst case rank is exactly $\lfloor \log_2 n \rfloor = \Theta(\log n)$.

Remark: For the question whether you can give a tree of height $\lfloor \log_2 n \rfloor$, 'yes' sufficed as answer.

Exercise 3: Max Flow (6 Points)

You are given a (connected) directed graph $G = (V, E)$, with positive integer capacities on each edge, a designated source $s \in V$, and a designated sink $t \in V$. Additionally you are given a current maximum $s - t$ flow $f : E \to \mathbb{N}$.
Now suppose we increase the capacity of one specific edge $e_0 \in E$ by one unit. Show how to find a maximum flow in the resulting graph with the manipulated capacity in time $\mathcal{O}(|E|)$.

Sample Solution

We construct the residual graph of the manipulated graph in time $\mathcal{O}(|E|)$. Then we search the residual graph for a path from s to t with a simple *Breadth/Depth First Search* in $\mathcal{O}(|E|)$ where we consider only edges with residual capacity > 0. If there is no augmenting path, the previous flow is still maximal. If we find an augmenting path from s to t, the flow can be increased by one along this path.

Exercise 4: Network Flow (4+4+6 Points)

Professor Adam has two children who, unfortunately, dislike each other. The problem is so severe that not only do they refuse to walk to school together, but in fact each one refuses to walk on any street that the other child has used. The children have no problem with their paths crossing at street intersections. Fortunately both the professor's house and *the only school* in town are on intersections, but beyond that he is not sure if it is going to be possible to send both of his children to the only school in town. The professor has a map of his town given as a graph $G = (V, E)$, where E are the streets and V are the intersections.

(a) Show how to formulate the problem of determining whether both his children can go to the school in town as a maximum flow problem.

(b) What algorithm from the lecture would you use to solve the problem? Assuming that for a street network $G = (V, E)$, we have $|E| \leq 3|V|$, what is the asymptotic running time of your algorithm as a function of the number of nodes $n = |V|$?

(c) Now, assume that the two children start disliking each other even more and they now only accept to go to the school if their walks also avoid crossing some of the intersections. Assume that we are given a subset $U \subseteq V$ of intersections and we now need to find two paths from the professor's home to the school such that every intersection in U and every street is part of at most one of the two paths. You can of course assume that the professor's home and the school are not in U.

How can you now reduce the problem to a maximum flow problem?

Sample Solution

(a) Given the city map, we construct a graph where each node represents an intersection, and node a and b are connected to each other if and only if there is street between the corresponding

intersections to nodes a and b. Let the home be in the intersection corresponding to node s, and the school be in the intersection corresponding to node t. Direct all the adjacent edges of s as outgoing edges. Direct all the adjacent edges of t as incoming edges. Moreover, replace all other edges in the graph by two directed edges in opposite directions. Label all the edges with 1 as their edge capacity.

If the maximum flow in this graph from s to t is at least 2, then there exist two "street disjoint" paths from s to t. The reason is that the flow has two paths ("streams") of size 1 which are edge disjoint. If it so happens that one path $(s, \ldots, x, u, v, y \ldots, t)$ uses and edge (u, v) and the other path $(s, \ldots, p, v, u, q \ldots, t)$ uses the edge (v, u) in the opposite direction we can always omit both edges (u, v),(v, u) by simply redirecting the paths: $(s, \ldots, p, y \ldots, t)$, $(s, \ldots, x, q \ldots, t)$. Doing this for each instance where an edge is used in both directions, we obtain two "street disjoint" paths and hence the children can go to school as desired.

(b) To find out whether the maximum flow is at least than 2 or not we only need to run at most two iterations of the Ford-Flukerson Algorithm. It takes $2 \cdot \mathcal{O}(|E|)$ rounds which is $\mathcal{O}(n)$.

(c) We replace each node $v \in U$ by two nodes v_{in} and v_{out}. Then we connect all the incoming edges of v to v_{in}, and we connect v_{out} to each of the nodes that v has an outgoing edge to. We then connect v_{in} and v_{out} with a directed edge from v_{in} to v_{out} with capacity 1. We run the algorithm again to find out whether the maximum flow is at least 2. If it is, then it means there are two paths from s to t such that these path are edge disjoint and also node disjoint with respect to the nodes in U, otherwise not.