Soft Heaps And Minimum Spanning Trees

Indranil Banerjee

George Mason University

ibanerje@gmu.edu

October 27, 2016

Quick Refresher: Heaps

A (min)-Heap is a data structure which stores a set of keys (with an underlying total order) on which following queries are supported:

1. Creat: Creates a (possibly empty) heap.
2. Insert(x): Inserts the key x to the heap.
3. Delete(x): Deletes the key x from the heap.
4. FindMin: Finds a key with the minimum value.
5. DecreseKey(x, y): Decreases the value of the key x to y.

and possibly,
Meld: Given two non-empty heaps H_1 and H_2, destructively merges them to produce H whose keys are union of the keys in H_1 and H_2

Quick Refresher: Heaps and Priority Queues

Sometime the FindMin and Delete is combined to a single operation called DeleteMin.

One of the most common method of implementing a priority queue is by using a heap.

- A min-heap can be used to implement a min-priority queue where the keys are popped in the increasing order of their priority.

In what to follow we shall only work with mergable-heap operations and ignore DECRESEKEY and DELETE.

Heaps and Priority Queues

Complexity: Lower bound

- Given a set of n elements, if we first make n-insertions and then n consecutive DeleteMin calls the extrcated sequence will be sorted.
- However, sorting n keys takes $\Omega(n \log n)$ comparisons.
- Hence, a sequence of n arbitrary operations on a heap requires $\Omega(n \log n)$ comparisons.

Binary Heap

A tree whose nodes contain keys, is said to be (min)-heap ordered if every parent's key is no more than the minimum key among its children

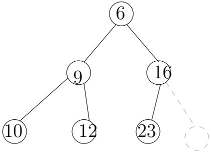
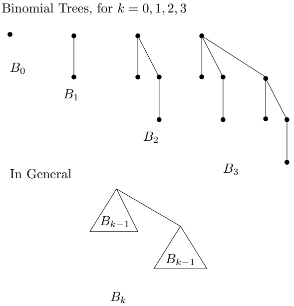


Figure: A binary heap with 6 nodes

Main problem: Melding takes $O(n)$ time.

Binomial Tree

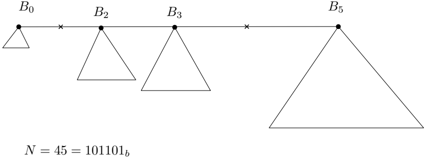
First we need to define a binomial tree:



- B_k, a tree with rank k has 2^k nodes
- Number of nodes at the i-level of B_k is $\binom{k}{i}$

Binomial Heap

A binomial heap consists of a list of heap ordered binomial trees. Using a list of trees help as achieve fast melds

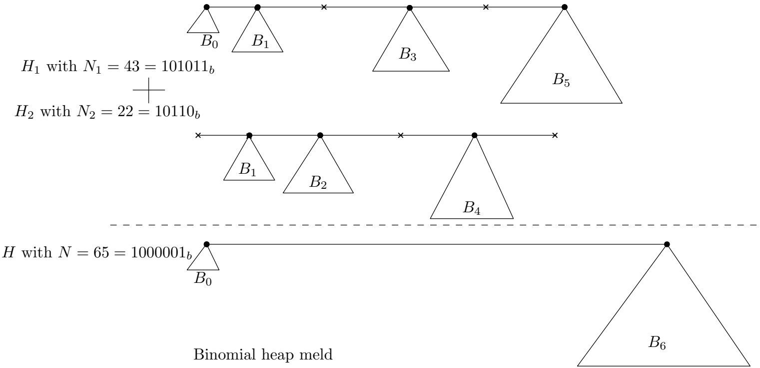


A binomial heap with N= 45 keys

Properties:

- DeleteMin takes $O(\log n)$ time, rest can be done in $\bar{O}(1)$ ($\bar{O}(.)$ = amortized time)
- Which means, Melds can also be done in $\bar{O}(1)$ time

Binomial Heap: Melds (H_1, H_2)



Binomial heap meld

Motivation

- Recall: For a classical heap, there is some sequence of $O(n)$ operations, that takes $\Omega(n \log n)$ total time to execute.
- The main motivation for soft heaps is to overcome this lower bound.
- Idea: what if we do not need to be correct all the time.
- For example, if we are allowed to err every time then clearly every heap operation can be performed in $O(1)$ time.

Motivation

- Q. Suppose we are allowed to err ϵ fraction of the time, then what is the best we could do?

Motivation

- Q. Suppose we are allowed to err ϵ fraction of the time, then what is the best we could do?
- A. $\Omega(n \log \frac{1}{\epsilon})$.

As we shall soon see, a soft-Heap achieves this bound.

- Idea: instead of maintaining exact keys, we allow for some keys to become corrupted
- thease corrupted keys are grouped and we only maintain an upper bound on the group maxima

Motivation

- Q. Suppose we are allowed to err ϵ fraction of the time, then what is the best we could do?
- A. $\Omega(n \log \frac{1}{\epsilon})$.

As we shall soon see, a soft-Heap achieves this bound.

Definition (Soft-Heap)

For any $\epsilon \in (0, \frac{1}{2}]$, in a soft-Heap a mixed sequence of operations including n-inserts can be performed in $\bar{O}(1)$ time, except for inserts which takes $O(\log \frac{1}{\epsilon})$ time. Additionally, the data-structure does not contain more than ϵn corrupted keys at any time.

Optimallity

Let H be a soft-heap and X a set of n unordered keys.

Definition (ϵ-near sorted)

We call a sequence S of keys ϵ-near sorted if the rank of any key in S is no more than $\epsilon\ n$ way from its actual rank in X.

Example: let $X = 5, 3, 2, 9, 13, 4$ and $\epsilon = \frac{1}{3}$ then

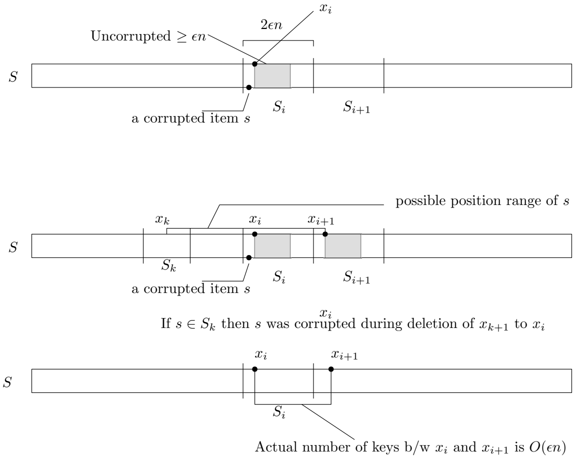
$S = (3, 2, 5, 4, 13, 9)$ is $\frac{1}{3}$-near-sorted.

Optimallity

We can use a soft-heap ϵ-near sort a set of keys as follows:

1. Insert the n items successively to build the heap.
2. Use DeleteMin n times and let S be the sequence of items popped
3. Consider set of keys S_i popped during i-th phase
4. where a phase is a block of $2\epsilon n$ DeleteMin operations

Optimallity



From S we can easily create a $O(\epsilon)$-near-sorted sequence.

Optimallity

Given $X = x_1, \ldots, x_n$ the #of ϵ near sorted permutation $C(n, \epsilon) =$

$$\binom{n}{\underbrace{\epsilon n, \epsilon n, \ldots, \epsilon n}_{\frac{1}{\epsilon} \text{ terms}}}$$

Hence, ϵ near sorting requires at least $\log C(n, \epsilon) = \Omega(n \log \frac{1}{\epsilon})$ comparisons

Construction

Soft-heap was introduced by Chazelle in 2000. Main differences with a binomial heap:

1. Binomial trees (called soft-queues) in the list may be partial
2. Some nodes in a soft-queue may contain more than one key, we call such nodes corrupted
3. Each node in addition maintains a super-key which is an upper bound on the keys present at the node
4. A soft-queue is (min)-heap ordered w.r.t these super-keys

Construction

The root of each soft-queue Q_k contains a pointer to the soft-queue Q_j ($j \geq k$) with the minimum super-key. (suffix-min-list)

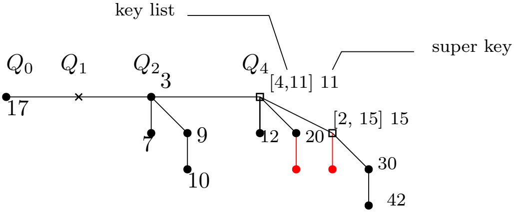
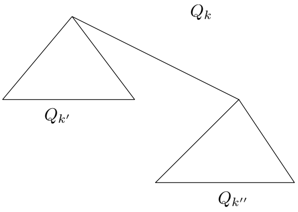


Figure: A soft heap with missing nodes, shown in red

Invariants

A rank of a node in Q_k is its corresponding rank in B_k
A soft-heap maintains the following invariants:

1. # of children at the root of Q_k is $\geq \lfloor k/2 \rfloor$
2. No node of rank below $r(\epsilon)$ is corrupted
3. See figure
4. No more than ϵn keys are corrupted at any given time, if the heap size is n



$k' = k'' \leq k - 1$

Heap operations

1. Insert, meld* works just like binomial heap
2. All the magic happens during the DeleteMin operations:
    - We look at the suffix-min pointer at Q_0, which points to the soft-queue with minimum super-key , say Q_k
    - However, the key list at the root of Q_k may be empty
    - In order to fix this, key(s) are moved up from the nodes below
    - This is accomplished using sift()

*except that we need to update the suffix-min-list

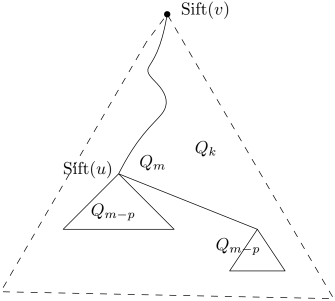
Heap operations: DeleteMin

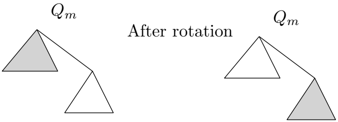
For now assume sift() works and refills the root as expected. We can now proceed with DeleteMin.

1. If the item-list in root is not empty then we return a key from it and we are done.
2. Otherwise, we have to use sift to refill the item list.
3. First we check if the rank invariant at the root still holds (# children $\geq \lfloor k/2 \rfloor$)
4. If not, the root is dismantled (we can do this since its item list is empty)
5. And all of its children re-melded back in to the heap
6. If the rank invariant holds we call sift(Q_k) to refill the root

Heap operations: Sift

The operation sift is what makes a soft-heap different from a regular heap.

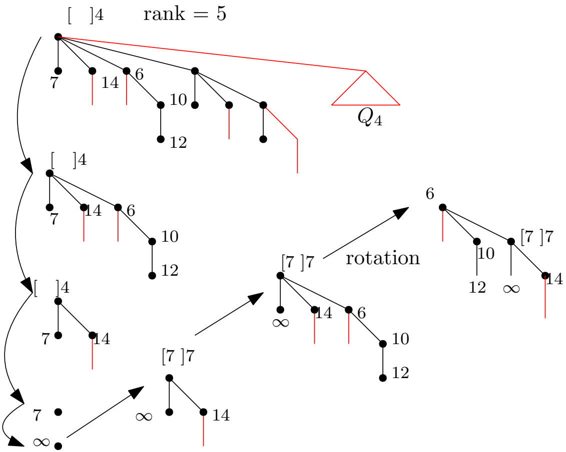




We sift again at Q_m if:

1. $m > r(\epsilon)$ and
2. Either m is odd or $p > 1$

Heap operations: Sift



After sift we clean up the nodes whose super-keys were set to ∞

Heap operations: Sift

Key observations:

1. If sift was never called twice during recursion, no branching will occur
2. In which case item lists will not merge and there will be no corrupted keys
3. Sift is only called twice for nodes with rank $> r(\epsilon)$
4. this ensures corruption occurs only higher up in the tree
5. Condition (2) makes branching somewhat balanced

Analysis

Set $r(\epsilon) = 2 + 2\lceil \log \frac{1}{\epsilon} \rceil$
Some lemmas:

Lemma

For node v with rank k, size of its item-list $\leq \max(1, 2^{\lceil k/2 \rceil - r(\epsilon)/2})$

Use induction on the depth of a recursion tree of a call to sift().

Lemma

Total number of corrupted items $\leq n/2^{r(\epsilon)-3}$

Use the previous lemma and sum over all the item lists of rank above $r(\epsilon)$.

Analysis

We only need to consider meld and sift. **Meld:**

- Meld takes constant amortized time, except in this case we have to update the suffix-min list
- This takes at most minimum of the rank of the two heaps
- A heap is built up using successive melds
- This we can model as a binary tree M
- An internal node z represents melding of two heaps
- Hence $cost(z) = 1 + \log \min(N(x), N(y))$
- summing this over all nodes gives $cost(M) = O(n)$

Analysis

We only need to consider meld and sift. **Sift:**

- First observe that, if a key becomes corrupted then it can never become uncorrupted again
- Hence calling sift strictly decreases the non-empty item lists in the heap (if branching occurs)
- Hence there can be at most $n - 1$ branching calls to sift
- By the branching condition, a branching call cannot occur at "depth" below $r(\epsilon)$
- Hence there can be at most $O(r(\epsilon)n)$ total calls to sift

Lastly, updating the suffix-min list during DeleteMin can be charged against the root dismantling, again due to the rank invariant.

An Application : MSF

The problem: Given a edge weighted graph G with n-vertices and m edges find a spanning forest F with the minimum total weight.

Solving MST is equivalent to solving MSF. **Lower Bound:**

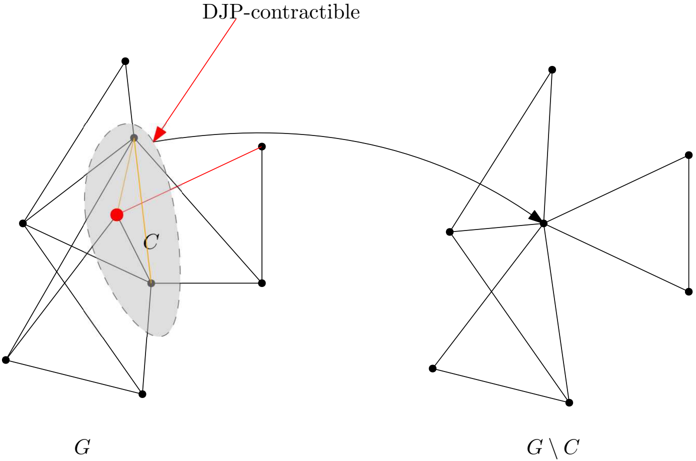
- The trivial lower bound is $\Omega(m)$.
- It is an **open problem** to determine the decision theoretic complexity of MSF (denote as $\mathcal{T}_{m,n}$)

Upper Bound:

- $O(m \log n)$, Dijkstra, Jarnik & Prim algorithms : grows a tree or a forest of trees
- $O(m \log n)$ Boruvka , uses minimum-weight matchings

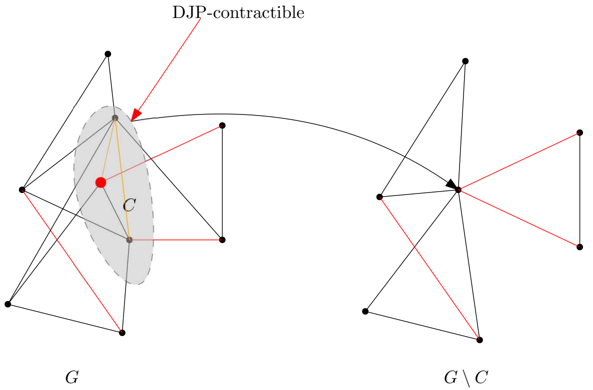
MST: Preliminary

A subgraph is DJP-contractible if a DJP tree grown inside C spans it.



MST: Preliminary

M be the set of corrupted red-edges, $M_C = C \cap M$ and G_M new graph with corrupted edges in M



Then, $MSF(G) \subset MSF(C) \cup MSF(G \setminus C - M_C) \cup M_c$.

MST: An Optimal algorithm [Pettie, 2000]

We can generalize this. Let $C_1, \ldots, C_k$ are all DJP-contractible and
Let $G' = G \setminus \bigcup_j C_j - \bigcup_j M_{c_j}$. Then

$MSF(G) \subset \bigcup_j MSF(C_j) \cup MSF(G') \cup \bigcup_j M_{c_j}$.
This yields the following strategy:

1. Solve MST for the DJP-contractible subgraphs using optimal number of comparisons (F_i's)
2. Solve MST in G' using the **dense case algorithm**(DCA) ($F_{G'}$)
3. Apply two steps of the Boruvka's algorithm on $G'' = \bigcup_i F_i \cup F_{G'} \cup M$
4. Recursively solve the reduced graph G'''.

MST: An Optimal algorithm

1. The algorithm first finds the DJP-contractible subgraphs
2. This is done by growing subgraphs using a min-edge weight priority queue, implemented using a **soft-heap**
3. Algorithm makes sure that if some C_i's from clusters, then these cluster sizes are large enough
4. For each subgraph C_i its MSF is calculated using some **optimal decision tree** for the $MSF(C_i)$
5. If $|C_i| = O(\log\log\log n)$ we can pre-compute all such ODTs in $o(n)$ time.

MST: An Optimal algorithm

1. Then a DCA is used to compute the MSF of G'
2. the dense case algorithm runs in linear time on a graph with $m/n = \Omega(\log\log\log n)$
3. Since each C_i clusters are $\Omega(\log\log\log n)$ the graph G' has $O(n/\log\log\log n)$ vertices.
4. Hence the DCA algorithm will run in $O(n+m)$ time in G'
5. Finally we need to compute the MSF of $\bigcup F_i \cup F_{G'} \cup M$
6. The two Baruvka step reduces the number of vertices to $\leq n/4$
7. Choosing $\epsilon = 1/8$ ensures that M is $\leq m/4$.
8. These give us the following recurrence:

$$T(n,m) \leq \sum \mathcal{T}(C_i) + T(n/4, m/2) + cm$$

References

Chazelle, B. (2000)
The soft heap: an approximate priority queue with optimal error rate.
Journal of the ACM (JACM), 47(6), 1012-1027.

Pettie, S., & Ramachandran, V. (2000)
An optimal minimum spanning tree algorithm
In International Colloquium on Automata, Languages, and Programming (pp. 49-60).