18.415/6.854 Advanced Algorithms September 3rd, 2008

Fibonacci heaps

Lecturer: Michel X. Goemans

1 Introduction

Today we will describe Fibonacci heaps, a data structure that provides a very efficient implementation of a priority queue. By priority queue we mean a data structure that stores a set S of elements where with each element s we associate a key $k(s)$ being "priority" of that element. Now, we want the queue to handle three operations on set S:

- INSERT Adding a new element s' with a key $k(s')$ to S
- EXTRACT-MIN Returning an element s^* of S having minimal key and removing s^* from S
- DECREASE-KEY Replacing the value of a key of some element s by a new, smaller value.

The motivation behind the search for fast implementation of priority queues can be observed on the example of two classical graph problems: Single-source Shortest Paths and Minimum Spanning Tree.

1.1 Single-source Shortest Paths problem

We are given a directed graph $G = (V, E)$, some vertex $s \in V$ and a length function $l : E \to \mathbb{R}_+$ on the arcs. Observe that we impose that the lengths are nonnegative. Now, for each vertex $v \in V$ we want to compute the length $d_s(v)$ of the shortest path from s to v.

The classical solution for this problem is Dijkstra's algorithm. The algorithm is:

1. Maintain a priority queue containing some subset S of vertices of G with keys $k(v)$. Initially, $S = V$, $k(s) = 0$ and $k(v) = +\infty$.
2. As long as S is nonempty:
   - Extract a vertex u from S with minimum key. Output $k(u)$ as the value of $d_s(u)$.
   - For each out-neighbor $v \in S$ of u, we update (i.e. possibly decrease) the key $k(v)$ of v to be $\min\{k(v), k(u) + l((u, v))\}$.

In this algorithm, $k(v)$ represents the length of the shortest path from s to v using only intermediate vertices not in S, and represents $d_s(v)$ when extracted. The algorithm can be adapted to output the shortest paths.

1.2 Minimum Spanning Tree problem

Given an undirected graph $G = (V, E)$ and a weight function $w : E \to \mathbb{R}$ on edges, we would like a spanning tree of G of minimal weight. Surprisingly, one of the classical solutions of this problem - Prim's algorithm - is very similar to the approach of Dijkstra's algorithm for Single-source Shortest Path problem. The algorithm is as follows:

1. Maintain a priority queue containing some subset S of vertices of G with keys $k(v)$ and a tree T spanning $V \setminus S$. Initially, $T = \emptyset$, $S = V$, $k(s) = 0$ for some arbitrary vertex s and $k(v) = +\infty$ for $v \neq s$.

2. As long as S is nonempty:

   - Extract a vertex u from S with minimum key. If $u \neq s$ (first iteration), add to T the corresponding edge (i.e. the minimum-weight edge connecting u to T of weight $k(u)$).
   - For each neighbor $v \notin S$ of u, we update (i.e. possibly decrease) $k(v)$ to be $\min\{k(v), w((u,v))\}$.

1.3 Number of priority queue operations

We will not prove the correctness of the algorithms. However, for the sake of the running time analysis that we will do later, we notice that in both cases the algorithm uses $|V|$ insert operations, $|V|$ extract-min operations and, since each edge can enforce at most one decrease-key operation, at most $|E|$ decrease-key operations.

2 Binary heaps

The classical implementation[1] of priority queues are binary heaps. A binary heap T is a binary tree whose nodes correspond to elements of the set S and has two properties:

- it is almost complete i.e. if T has depth h then it has exactly 2^i vertices on depth i if $i < h$ and the last level is filled from the left.
- *heap-ordering*: the key of every child is not smaller than the key of its parent.

Keeping this properties in mind it is relatively easy (see [CLRS]) to develop procedures for inserting, extracting the minimal element and decreasing the key that execute any of these operations in $O(\log n)$ time where n is the number of items in the priority queue. Therefore, since the number of elements is at most $|V|$ in our applications, we obtain the total running time of both algorithms to be $O((|V|+|E|)\log|V|)$. Obviously, $|E| \geq |V|$ in case of connected graphs and therefore the running time is dominated by the $O(|E|\log|V|)$ term corresponding to decrease-key operations. The question is: can we do better ?

3 d-ary heaps

One of the ideas to get a better running time is increasing the arity of the tree that we are using. If we use a d-ary tree instead of a binary one then we reduce the depth of our tree and thus our inserts and bottlenecking decrease-key operations execute in $O(\log_d |S|)$ time. On the other hand, the execution of extract-min operation requires $O(d \log_d |S|)$ time. So, by choosing the best possible $d = \lceil |E|/|V| \rceil$ we get the total running time of our algorithms to be $O((|E|+d|V|)\log_d |V|) = O(|E| log_{\lceil |E|/|V| \rceil} |V|)$, which is a significant improvement for dense graphs. However, it turns out that we can do even better. Namely, we can implement priority queue in such a way that from the point of view of running time analysis of our algorithms the cost of DECREASE-KEY will be constant and costs of INSERT and EXTRACT-MIN will be logarithmic. This leads to the essentially optimal $O(|E| + |V|\log|V|)$ running time of both algorithms and for some present-day applications (think graphs with billions of edges) this improvement can make a huge difference.

[1] A comprehensive coverage of binary heaps (as well as Fibonacci heaps) can be found in [CLRS].

4 Fibonacci heaps

The Fibonacci heaps were proposed by Fredman and Tarjan in 1984 giving a very efficient implementation of the priority queues. The main motto of this construction is laziness - "we do work only when we must, and then use it to simplify the structure as much as possible so that the future work is easy". This way, we enforce that any sequence of operations has to contain a lot of cheap ones before we need to do something computationally expensive - the formalization of this intuition will be given later.

4.1 Construction

A Fibonacci heap consists of a collection of heap-ordered trees (of variable arity) with following properties:

1. nodes of the trees correspond to elements being stored in the queue,
2. roots of heap-ordered trees are arranged in a doubly-linked list,
3. we keep a pointer to the root of a tree that corresponds to the element with minimum key (note that heap-ordering of the trees implies that such minimum element has to be a root of some tree),
4. for each node we keep track of its rank (degree), i.e. the number of its children, as well as whether it is marked (the purpose of marking will be defined later on),
5. *size requirement*: if a node u has rank k then the subtree rooted at u has at least F_{k+2} nodes, where F_i is the i-th Fibonacci number, i.e. $F_0 = 0$, $F_1 = 1$ and $F_i = F_{i-1} + F_{i-2}$ for $i \geq 2$.

We proceed now to describing how do we perform priority queue operation on our Fibonacci heap.

4.1.1 INSERT

Inserting is very simple. We just add the new element s as a new heap-ordered tree to our collection and check whether $k(s)$ is smaller that the current minimum for the queue—if so then we change the pointer to the minimum accordingly (see Figure 1).

4.1.2 DECREASE-KEY

When we decrease the key of an element s, if the heap-ordering is still satisfied then we do not need to do anything else. Otherwise, we just cut s out of the tree in which it resides and put it as a root of a new tree in our collection (note that all the descendants of s are now in this new tree as well). We compare the new key of s and the previously minimum key and change the pointer accordingly (see Figure 1).

This way we end up with something that looks like a desired Fibonacci heap. However, the problem with simply cutting each such s is that, when we perform in this manner many DECREASE-KEY operations, we may end up violating the size requirement that we wanted to preserve. Therefore, to alleviate this issue we introduce an additional rule that when we cut s we check whether its parent is marked. If so then we cut the parent as well (and we unmark it). Otherwise, we just mark the parent. Note that we do this cutting recursively, so if the parent of marked parent of s is also marked then we cut it as well, and so on. Obviously, if we cut a root we are not doing anything, and so it is useless to mark a root. This (potentially cascading) cutting procedure therefore always ends.

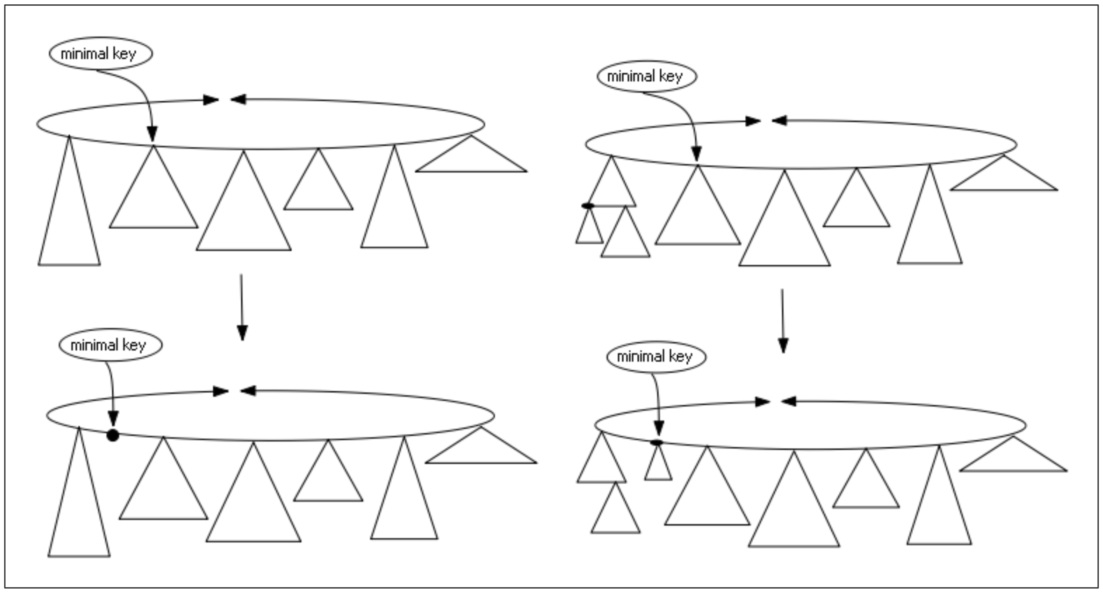

Figure 1: Illustration of: (left side) inserting a new element to the Fibonacci heap; (right side) cutting a vertex in the first step of DECREASE-KEY operation. In both examples we assumed that the newly created root has smaller key than the keys of all the other elements.

4.1.3 EXTRACT-MIN

Finally, we can describe extracting the minimum element s^*. We start with removing s^* (recall that we stored the pointer to it) and putting all the children of s^* as roots of new trees in our collection. Next, we scan the entire list of roots in our collection to find the new minimum element and we set the relevant pointer accordingly.

In principle at this point we could be done, because we obtain once again a valid Fibonacci heap. However, it is not hard to see that so far executing of any of our queue operations makes the list longer and longer. So, going through the whole list of roots during EXTRACT-MIN can be very expensive computationally. Therefore, in the spirit of laziness, if we have to do this work anyway then we can use this opportunity to do some cleaning as well, and avoid in this way the necessity of doing the whole work again when doing the next EXTRACT-MIN . What we do is, as long as there are two trees whose roots have the same rank, say k, we merge these trees to obtain one tree of rank $k + 1$. Merging consist of just comparing the keys of the roots and setting the root of the tree with larger key as a new child of the other root (see Figure 2). Note that since merging can introduce a second tree of rank $k + 1$ in the collection, one root can take part in many merges.

4.2 Running-time Analysis

Now we want to analyze the worst-case performance of the described Fibonacci heap data structure.

4.2.1 A worst-case example

Let's imagine the following scenario: We do n consecutive INSERT operations into the Fibonacci heap such that it is a circular linked list containing all elements as singleton heaps. If we perform an EXTRACT-MIN operation on this Fibonacci heap, this operation will have to go through the entire list to determine the new minimum. This takes $O(n)$ time — an unbearable performance for just one operation.

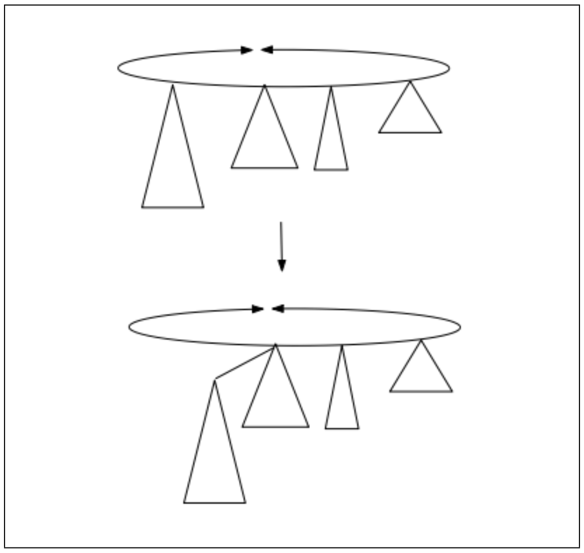

Figure 2: Illustration of merging of two trees of the same rank.

4.2.2 Are Fibonacci heaps useless?

Does this mean that Fibonacci heaps are inefficient? No! Intuitively such heavy operations can occur only very rarely and make no big contribution to the overall running time of an algorithm using the heap. Being not able to give worst-case performance guarantees for each individual operation, we want to consider a sequence of operations and give a proof that, for any such sequence, the total running time is small, in the sense that this running time can be apportioned between the individual operations so that each has a small contribution. This type of analysis is called *amortized analysis* [CLRS]. More precisely, if we have ℓ different types of operations and we claim that the *amortized* running time of an operation of type j is at most t_j, this means that for any sequence of operations composed of k_j operations of type j for all $j = 1, \cdots, \ell$ (with operations of different types interlaced in any way), the total running time is upperbounded by $\sum_j k_j t_j$.

4.2.3 Excursion: Amortized Analysis via the Potential Method

The most common way to perform amortized analysis is using the potential method. The idea of the potential method allows cheap operations to save up time for the use of heavy operations. This functions like a bank account with time deposited in it. The potential function Φ represents the balance in the account. Initially, the balance is zero, and remains nonnegative during the whole sequence. Now operations are performed having costs (i.e. running times) of $c_1, c_2, c_3, ..., c_k$. Every operation is allowed to either pay more than its actual cost c_i thereby increasing its amortized cost, placing the credit/savings in the bank account thus increasing the balance Φ, or pay less than the actual cost by withdrawing the difference from Φ. This gives the amortized cost.

Often one can think of the potential function as a measurement of the complexity of the data structure or configuration within an algorithm. In this case cheap operations are allowed to increase the internal complexity, while operations which simplify or clean up the data are allowed to take more time.

Making this formal, a potential function, Φ, maps a configuration D_i of an evolving algorithm or data structure D into a nonnegative number. The start configuration is normalized to have the value 0: $\Phi_0 = \Phi(D_0) = 0$. Consider a sequence of operations $o_1, o_2, o_3, ..., o_k$ and let D_i be the configuration of the data structure after performing the ith operation. We impose that the potential function remains nonnegative throughout:

$$\forall t : \Phi_t = \Phi(D_t) \geq 0.$$

If operation o_i has cost (running time) c_i then its amortized cost is defined by:

$$a_i = c_i + \Delta\Phi_i = c_i + \Phi_i - \Phi_{i-1}.$$

Given this, it is easy to see that the sum of the amortized costs upperbounds the original total cost:

$$\sum_{i=1}^{k} a_i = \sum_{i=1}^{k} (c_i + \Phi_i - \Phi_{i-1}) = \sum_{i=1}^{k} c_i + \Phi_k - \Phi_0 \geq \sum_{i=1}^{k} c_i.$$

Thus amortized analysis provides an upper bound on the worst-case cost of any sequence of operations.

The difficulty in performing amortized analysis is in choosing the right potential function.

4.2.4 Fibonacci heaps obey the size requirement

The first important observation regarding the heap-ordered trees in a Fibonacci heap is that the restriction to cut off at most one child prevents cutting down the nice binomial tree like structure built up through the combination steps. This guarantees that the size requirement we want to have is preserved.

Lemma 1 *Consider a node x with rank (number of children) d. Let $y_1, y_2, ..., y_d$ be those children in the order they were added to the tree. Then every child y_i has rank at least $i-2$.*

Proof: When y_i was added to x, at least the $i-1$ children y_1 to y_{i-1} were present. Since only roots of the same rank get combined, y_i had at least $i-1$ children at this time. At most, one of these children could have been cut away since otherwise y_i would have qualified for a cascading cut. Thus y_i has at least $i-2$ children. □

A simple counting argument given in the next lemma reveals that the number of nodes in a subtree rooted at a node of rank d is at least F_{d+2}. This exponential growth upperbounds the heap degrees to be logarithmic.

Lemma 2 *Let $N(d)$ be the smallest possible number of nodes in a subtree rooted at a node of rank d. Then $N(d) \geq F_{d+2}$. Thus, the rank of any node in a Fibonacci heap with n elements is $O(\log n)$.*

Proof: For N, it holds that $N(0) = 1$, $N(1) = 2$ and we have the recurrence relation:

$$N(d) \geq 2 + \sum_{i=2}^{d} N(i-2)$$

because of Lemma 1 (counting one for the root, one for the first child y_1 and $N(i-2)$ for each remaining child y_i). Proceeding by induction on d (thus assuming that $N(j) \geq F_{j+2}$ for $j < d$), we get that

$$N(d) \geq 2 + \sum_{i=2}^{d} F_i = 1 + \sum_{i=0}^{d} F_i.$$

The right-hand-side is F_{d+2}; this can be shown again by induction on d: $1 + \sum_{i=0}^{d} F_i = F_{d+1} + F_d = F_{d+2}$. Thus we have shown the first part of the lemma that $N(d) \geq F_{d+2}$.

Using the closed-form expression for the Fibonacci numbers, we get that

$$N(d) \geq F_{d+2} = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^{d+2} - \left(\frac{1-\sqrt{5}}{2} \right)^{d+2} \right) \geq 1.61^d.$$

Since $N(d) \leq n$, all ranks of nodes in the heap are at most $log_{1.61}\ n$. □

4.2.5 Amortized Analysis of the Fibonacci heap operations

Each individual adding, combining and cutting step takes only $O(1)$ time. Thus the only two critical situations occur when we have to search through many roots for finding the minimum and when we have a long chain of cascading cuts. The length of a cascading cut corresponds to the number of nodes being unmarked. With this intuition, we choose the potential function to be

$$\Phi_t = r_t + 2m_t$$

where r_t is the number of roots and m_t the number of marked nodes at time t. The reason for the factor of 2 will become clear in the analysis. Here is the amortized analysis of each operation.

- INSERT

  Inserting a new root in the list takes $c_t = O(1)$ time and increases the number of roots $r_t = r_{t-1} + 1$ by one. Thus the amortized cost for an INSERT operation is also constant:

$$a_t = c_t + (r_t - r_{t-1}) + 2(m_t - m_{t-1}) = O(1) + 1 + 0 = O(1).$$

- EXTRACT-MIN

  During an EXTRACT-MIN operation, we start with r_t roots, cut away the minimum root (say of rank d) leaving $r_{t-1} + d - 1$ roots in the list. These get combined to r_t roots, having different ranks. Since, by Lemma , the maximum possible rank is $O(\log n)$, there are in the end only $r_t = O(\log n)$ roots left. Since the cut and each of the combining steps takes $O(1)$ time and eliminates one root the actual time spend on an EXTRACT-MIN operation is at most $c_t = r_{t-1}+d-1$ units (where the 'unit' may need to be redefined to take into account constants). Putting this together the amortized cost for an EXTRACT-MIN operation is logarithmic:

$$a_t = c_t + (r_t - r_{t-1}) + 2(m_t - m_{t-1}) = (r_{t-1} + d - 1) + (r_t - r_{t-1}) + 0 = r_t + d - 1 = O(\log n).$$

- DECREASE-KEY

  Let's assume that during a DECREASE-KEY operation we do k cuts, $k \geq 1$. Each (but the first) cut unmarks a node and each cut introduces a new root. Thus the increase in the number of roots, $r_t - r_{t-1}$, is equal to the number k of cuts performed. The decrease $m_{t-1} - m_t$ of marked nodes is either $k-1$ or k (depending on whether the node itself was marked); thus, in any case, $m_{t-1} - m_t \geq k-1$. The key decreasing, cutting and reinserting takes $1+k$ units of time (redefining the unit, if needed), and thus its amortized cost is:

$$a_t = c_t + (r_t - r_{t-1}) + 2(m_t - m_{t-1}) \leq 1 + k + k - 2(k-1) = O(1).$$

  This last relation justifies the constant 2 in the definition of the potential function.

Summarizing, in a Fibonacci heap, every INSERT and DECREASE-KEY takes $O(1)$ amortized time, and every EXTRACT-MIN takes $O(\log n)$ amortized time.

4.2.6 Using Fibonacci heaps to speed up Prim's and Dijkstra's algorithm

Using Fibonacci heaps in the two algorithms mentioned in the introduction leads to improved running times of $O(|E| + |V| \log |V|)$.

References

[CLRS] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Cliff Stein. *Introduction to Algorithms (Second Edition)*. MIT Press and McGraw-Hill.