6.854 Advanced Algorithms

Lecture 1: September 9, 1999 Lecturer: David Karger
Scribes: David G. Andersen, Ioana Dumitriu, John Dunagan, Akshay Patil (2003)

Fibonaccci Heaps

1.1 Motivation and Background

Priority queues are a classic topic in theoretical computer science. As we shall see, Fibonacci Heaps provide a fast and elegant solution. The search for a fast priority queue implementation is motivated primarily by two network optimization algorithms: Shortest Path and Minimum Spanning Tree (MST).

1.1.1 Shortest Path and Minimum Spanning Trees

Given a graph $G(V, E)$ with vertices V and edges E and a length function $l : E \rightarrow \Re^+$. We define the Shortest Path and MST problems to be, respectively:

shortest path. For a fixed source $s \in V$, find the shortest path to all vertices $v \in V$

minimum spanning tree (MST). Find the minimum length set of edges $F \subset E$ such that F connects all of V.

Note that the MST problem is the same as the Shortest Path problem, except that the source is not fixed. Unsurprisingly, these two problems are solved by very similar algorithms, Prim's for MST and Djikstra's for Shortest Path. The algorithm is:

1. Maintain a priority queue on the vertices
2. Put s in the queue, where s is the start vertex (Shortest Path) or any vertex (MST). Give s a key of 0.
3. Repeatedly delete the minimum-key vertex v from the queue and mark it "scanned"

   For each neighbor w of v:

   If w is not in the queue and not scanned, add it with key:

   - Shortest Path: $key(v) + length(v \rightarrow w)$
   - MST: $length(v \rightarrow w)$

   If, on the other hand, w is in the queue already, then decrease its key to the minimum of the value calculated above and w's current key.

1.1.2 Heaps

The classical answer to the problem of maintaining a priority queue on the vertices is to use a binary heap, often just called a heap. Heaps are commonly used because they have good bounds on the time required for the following operations:

insert	$O(\log n)$
delete-min	$O(\log n)$
decrease-key	$O(\log n)$

If a graph has n vertices and m edges, then running either Prim's or Djikstra's algorithms will require $O(n \log n)$ time for inserts and deletes. However, in the worst case, we will also perform m decrease-keys, because we may have to perform a key update every time we come across a new edge. This will take $O(m \log n)$ time. Since the graph is connected, $m \geq n$, and the overall time bound is given by $O(m \log n)$.

Since $m \geq n$, it would be nice to have cheaper key decreases. A simple way to do this is to use *d-heaps*.

1.1.3 d-Heaps

d-heaps make key reductions cheaper at the expense of more costly deletions. This trade off is accomplished by replacing the binary heap with a d-ary heap—the branching factor (the maximum number of children for any node) is changed from 2 to d. The depth of the tree then becomes $\log_d(n)$. However, delete-min operations must now traverse all of the children in a node, so their cost goes up to $d \log_d(n)$. Thus, the running time of the algorithm becomes $O(nd \log_d(n) + m \log_d(n))$. Choosing the optimal $d = m/n$ to balance the two terms, we obtain a total running time of $O(m \log_{m/n} n)$.

When $m = n^2$, this is $O(m)$, and when $m = n$, this is $O(n \log n)$. This seems pretty good, but it turns out we can do much better.

1.1.4 Amortized Analysis

Amortized analysis is a technique for bounding the running time of an algorithm. Often we analyse an algorithm by analyzing the individual operations that the algorithm performs and then multiplying the total number of operations by the time required to perform an operation. However, it is often the case that an algorithm will on occasion perform a very expensive operation, but most of the time the operations are cheap. Amortized analysis is the name given to the technique of analyzing not just the worst case running time of an operation but the average case running time of an operation. This will allow us to balance the expensive-but-rare operations against their cheap-and-frequent peers.

There are several methods for performing amortized analysis; for a good treatment, see *Introduction to Algorithms* by Cormen, Leiserson, and Rivest. The method of amortized analysis used to analyze Fibonacci heaps is the potential method:

- Measure some aspect of the data structure using a potential function. Often this aspect of the data structure corresponds to what we intuitively think of as the complexity of the data structure or the amount by which it is out of kilter or in a bad arrangement.

- If operations are only expensive when the data structure is complicated, and expensive operations can also clean up ("uncomplexify") the data structure, and it takes many cheap operations to noticeably increase the complexity of the data structure, then we can *amortize* the cost of the expensive operations over the cost of the many cheap operations to obtain a low average cost.

Therefore, to design an efficient algorithm, we want to force the user to perform many operations to make the data structure complicated, so that the work doing the expensive operation and cleaning up the data structure is amortized over those many operations.

We compute the *potential* of the data structure by using a potential function Φ that maps the data structure (DS) to a real number $\Phi(DS)$. Once we have defined Φ, we calculate the cost of the i^{th} operation by:

$$cost_{amortized}(operation_i) = cost_{actual}(operation_i) + \Phi(DS_i) - \Phi(DS_{i-1})$$

where DS_i refers to the state of the data structure after the i^{th} operation. The sum of the amortized costs is then

$$\sum cost_{actual}(operation_i) + \Phi_{final} - \Phi_{initial}$$

.

If we can prove that $\Phi_{final} \geq \Phi_{initial}$, then we've shown that the amortized costs bound the real costs, that is, $\sum cost_{amortized} \geq \sum cost_{actual}$. Then we can just analyze the amortized costs and show that this isn't too much, knowing that our analysis is useful. Most of the time it is obvious that $\Phi_{final} \geq \Phi_{initial}$ and the real work is in coming up with a good potential function.

1.2 Fibonacci Heaps

The Fibonacci heap data structure invented by Fredman and Tarjan in 1984 gives a very efficient implementation of the priority queues. Since the goal is to find a way to minimize the number of operations needed to compute the MST or SP, the kind of operations that we are interested in are *insert*, *decrease-key*, *merge*, and *delete-min*. (We haven't covered why *merge* is a useful operation yet, but it will become clear.) The method to achieve this minimization goal is laziness – **"do work only when you must, and then use it to simplify the structure as much as possible so that your future work is easy"**. This way, the user is forced to do many cheap operations in order to make the data structure complicated.

Fibonacci heaps make use of heap-ordered trees. A heap-ordered tree is one that maintains the *heap property*, that is, where $key(parent) \leq key(child)$ for all nodes in the tree.

A Fibonacci heap H is a collection of heap-ordered trees that have the following properties:

1. The roots of these trees are kept in a doubly-linked list (the "root list" of H);
2. The root of each tree contains the minimum element in that tree (this follows from being a heap-ordered tree);
3. We access the heap by a pointer to the tree root with the overall minimum key;
4. For each node x, we keep track of the *rank* (also known as the *order* or *degree*) of x, which is just the number of children x has; we also keep track of the *mark* of x, which is a Boolean value whose role will be explained later.

For each node, we have at most four pointers that respectively point to the node's parent, to one of its children, and to two of its siblings. The sibling pointers are arranged in a doubly-linked list (the "child list" of the parent node). Of course, we haven't described how the operations on Fibonacci heaps are implemented, and their implementation will add some additional properties to H. Here are some elementary operations used in maintaining Fibonacci heaps.

1.2.1 Inserting, merging, cutting, and marking.

Inserting a node x. We create a new tree containing only x and insert it into the root list of H; this is clearly an $O(1)$ operation.

Merging two trees. Let x and y be the roots of the two trees we want to merge; then if the key in x is no less than the key in y, we make x the child of y; otherwise, we make y the child of x. We update the appropriate node's rank and the appropriate child list; this takes $O(1)$ operations.

Cutting a node. If x is a root in H, we are done. If x is not a root in H, we remove x from the child list of its parent, and insert it into the root list of H, updating the appropriate variables (the rank of the parent of x is decremented, etc.). Again, this takes $O(1)$ operations. (We assume that when we want to find a node, we have a pointer hanging around that accesses it directly, so actually finding the node takes $O(1)$ time.)

Marking. We say that x is marked if its mark is set to "true", and that it is unmarked if its mark is set to "false". A root is always unmarked. We mark x if it is not a root and it loses a child (i.e., one of its children is cut and put into the root-list). We unmark x whenever it becomes a root. We will make sure later that no marked node loses another child before it itself is cut (and reverted thereby to unmarked status).

1.2.2 Decreasing keys and Deleting mins

At first, *decrease-key* does not appear to be any different than *merge* or *insert*; just find the node and cut it off from its parent, then insert the node into the root list with a new key. This requires removing it from its parent's child list, adding it to the root list, updating the parent's rank, and (if necessary) the pointer to the root of smallest key. This takes $O(1)$ operations.

The *delete-min* operation works in the same way as *decrease-key*: Our pointer into the Fibonacci heap is a pointer to the minimum keyed node, so we can find it in one step. We remove this root of smallest key, add its children to the root-list, and scan through the linked list of all the root nodes to find the new root of minimum key. Therefore, the cost of a *delete-min* operation is $O(\#$ of children $)$ of the root of minimum key plus $O(\#$ of root nodes); in order to make this sum as small as possible, we have to add a few bells and whistles to the data structure.

1.2.3 Population Control for Roots

We want to make sure that every node has a small number of children. This can be done by ensuring that the total number of descendants of any node is exponential in the number of its children. In the absence of any "cutting" operations on the nodes, one way to do this is by **only** merging trees that have the same number of children (i.e, the same rank). It is relatively easy to see that if we only merge trees that have the same rank, the total number of descendants (counting onself as a descendant) is always $(2^{\#\text{ of children}})$. The resulting structure is called a binomial tree because the number of descendants at distance k from the root in a tree of size n is exactly $\binom{n}{k}$. Binomial heaps preceded Fibonacci heaps and were part of the inspiration for them. We now present Fibonacci heaps in full detail.

1.2.4 Actual Algorithm for Fibonacci Heaps

- Maintain a list of heap-ordered trees.
- *insert*: add a degree 0 tree to the list.
- *delete-min*: We can find the node we wish to delete immediately since our handle to the entire data structure is a pointer to the root with minimum key. Remove the smallest root, and add its children to the list of roots. Scan the roots to find the next minimum. Then consolidate all the trees (merging trees of equal rank) until there is ≤ 1 of each rank. (Assuming that we have achieved the property that the number of descendants is exponential in the number of children for any node, as we did in the binomial trees, no node has rank $> c\log n$ for some constant c. Thus consolidation leaves us with $O(\log n)$ roots.) The consolidation is performed by allocating buckets of sizes up to the maximum possible rank for any root node, which we just showed to be $O(\log n)$. We put each node into the appropriate bucket, at cost $O(\log n) + O(\#$ of roots). Then we march through the buckets, starting at the smallest one, and consolidate everything possible. This again incures cost $O(\log n) + O(\#$ of roots).
- *decrease-key*: cut the node, change its key, and insert it into the root list as before, Additionally, if the parent of the node was unmarked, mark it. If the parent of the node was marked, cut it off also. Recursively do this until we get up to an unmarked node. Mark it.

1.2.5 Actual Analysis for Fibonacci Heaps

Define $\Phi(DS) = (k\cdot\, \#$ of roots in $DS + 2\cdot\#$ marked bits in $DS)$. Note that *insert* and *delete-min* do not ever cause nodes to be marked - we can analyze their behaviour without reference to marked and unmarked bits. The parameter k is a constant that we will conveniently specify later. We now analyze the costs of the operations in terms of their amortized costs (defined to be the real costs plus the changes in the potential function).

- *insert*: the amortized cost is O(1). O(1) actual work plus k * O(1) change in potential for adding a new root. O(1) + kO(1) = O(1) total amortized cost.
- *delete-min*: for every node that we put into the root list (the children of the node we have deleted), plus every node that is already in the root list, we do constant work putting that node into a bucket corresponding to its rank and constant work whenever we merge the node. Our real costs are putting the roots into buckets (O(#*roots*)), walking through the buckets (O($\log n$)), and doing the consolidating tree merges (O(#*roots*)). On the other hand, our change in potential is $k*(\log n - \#roots)$ (since there are at most $\log n$ roots after consolidation). Thus, total amortized cost is O(#*roots*) + O($\log n$) + $k * (\log n - \#roots)$ = O($\log n$).
- *decrease-key*: The real cost is O(1) for the cut, key decrease and re-insertion. This also increases the potential function by O(1) since we are adding a root to the root list, and maybe by another 2 since we may mark a node. The only problematic issue is the possibility of a "cascading cut" - a cascading cut is the name we give to a cut that causes the node above it to cut because it was already marked, which causes the ndoe above it be cut since it too was alrady marked, etc. This can increase the actual cost of the operation to (# of nodes already marked). Luckily, we can pay for this with the potential function! Every cost we incur from having to update pointers due to a marked node that was cut is offset by the decrease in the potential function when that previously marked node is now left unmarked in the root list. Thus the amortized cost for this operation is just O(1).

The only thing left to prove is that for every node in every tree in our Fibonacci heap, the number of descendants of that node is exponential in the number of children of that node, and that this is true even in the presence of the "weird" cut rule for marked bits. We must prove this in order to substantiate our earlier assertion that all nodes have degree $\leq \log n$.

1.2.6 The trees are big

Consider the children of some node x in the order in which they were added to x.

Lemma : The i^{th} child to be added to x has rank at least $i-2$.

Proof : Let y be the i^{th} child to be added to x. When it was added, y had at least $i-1$ children. This is true because we can currently see $i-1$ children that were added earlier, so they were there at the time of the y's addition. This means that y had at least $i-1$ children at the time of it's merger, because we only merge equal ranked nodes. Since a node could not lose more than one child without being cut itself, it must be that y has at least $i-2$ children ($i-1$ from when it was added, and no more than a potential 1 subsequently lost). ■

Note that if we had been working with a binomial tree, the appropriate lemma would have been $rank = i-1$ not $\geq i-2$.

Let S_k be the minimum number of descendants of a node with k children. We have $S_0 = 1, S_1 = 2$ and,

$$S_k \geq \sum_{i=0}^{k-2} S_i$$

This recurrence is solved by $S_k \geq F_{k+2}$, the $(k+2)^{th}$ Fibonacci number. Ask anyone on the street and that person will tell you that the Fibonacci numbers grow exponentially; we have proved $S_k \geq 1.5^k$, completing our analysis of Fibonacci heaps.

1.2.7 Utility

Only recently have problem sizes increased to the point where Fibonacci heaps are beginning to appear in practice. Further study of this issue might make an interesting term project; see David Karger if you're curious.

Fibonacci Heaps allow us to improve the running time in Prim's and Djikstra's algorithms. A more thorough analysis of this will be presented in the next class.