Using Data Structures to Design Clever Algorithms
202
Lecture 7

Lecture given by Russell Impagliazzo, scribed by Sashka Davis

November 15, 17, 2005

1 Fitting data structures to algorithms: general comments

Another way to cross the boundary of methodical algorithms and design clever algorithms is to design data structures that are tuned to the needs of your algorithms. We will illustrate this process of designing clever algorithm by using Dijkstra's algorithms for single source shortest paths in graphs with non-negative weights on edges.

Our general methodology is: 1) Identify the operations needed by the algorithm: view the algorithm as a series of abstract operations performed on some 'dynamic data structure'; 2) Abstract away the operations that must be provided by your data structure and needed by the algorithm, from the underling implementation model. Here we define a class of algorithms, which are parametrized by different implementation of the 'dynamic data structure'. 3) Analyze the algorithm using this abstract notion of a data structure: look at the life of the algorithm and how often different operations are performed. 4) Design a data structure such that the operations that are frequent have efficient asymptotic complexity. This is the step where we find from the class of algorithms the best one by plugging in the "best" data structure for our particular algorithm. 5) Be clever and use advanced analysis techniques like amortized analysis to show that the algorithm using your new data structure has (somewhat) surprising running time.

2 Dijkstra's Algorithm

Given a graph $G = (V, E)$ and a source $s \in V$, the single source shortest path in graphs with negative weights problem is to find the shortest path from s to $v \in V - \{s\}$, provided that all edges have non-negative weights $E : V \mapsto \Re^+$.

Dijkstra solves the problem by maintaining two sets of nodes S and T. S are the nodes, whose shortest path is fund, and T are those "frontier" nodes that are adjacent to nodes in S. At each iteration of the algorithm nodes in T, that are adjacent to nodes in S have their shortest path estimate re-evaluated. The node with the minimum shortest path estimate, say that is u is removed from T, and is moved to S; and the neighbors of u have their shortest path estimates adjusted. Pseudo code follows.

```
$S$ is the nodes with computed distance from $s$.
Initially we have only $S = \{s\}$.
$c(v) = \min_{u \in S; N_{in}(v)}\{D[u] + w(u, v)\}$
Repeat $n - 1$ times
```

```
$v \leftarrow \text{argmin}_{v' \in T} c(v')$
$D[v] \leftarrow c(v)$
$S \leftarrow S \cup \{v\}; T \leftarrow T - \{v\};$
For each $v' \in adj(v)$
    $c(v' = min\{c(v'), D[v] + w(v, v')\}$
```

2.1 Analysis and parameterization

The set T is dynamic and each element has a name and a key, which is the shortest path estimate c. The set of operations that we need to perform on T is exactly the operations provided by the priority queue data structure.

- FIND_MIN - finds the element with minimum key.
- DELETE_MIN - deletes the element with the minimum key.
- DECREASE_KEY_BY_NAME - given a name of an element, finds the element and adjust its key.

2.2 Parameterized Analysis of Dijkstra

Each element is removed from T exactly once, hence we use $O(|V|)$ FIND_MIN and $O(|V|)$ DELETE_MIN operations. DECREASE_KEY_BY_NAME is used in total $O(|E|)$ times. Let T_F, T_{DEL} and T_{DEC} be the time complexity of the FIND_MIN, DELETE_MIN, and DECREASE_BY_NAME operations, respectively. Then the time complexity of Dijkstra is:

$$T(|V|, |E|) = O(|V|T_F + |V|T_{DEL} + |E|T_{DEC})$$

Note that the instances are graphs and unless there are many isolated vertices (in which case the problem is not interesting) then $O(|E|)$ is at least $O(|V|)$.

2.3 Array, Binary Heap, d-Heap

Here we look at the time complexity of Dijkstra's algorithm using a few example implementations of priority queues.

- Array implementation of PQ

  PQ(Array):

Operation	Cost	Times Used	Total Cost
FIND_MIN	$O(\|V\|)$	$(\|V\|)$	$O(\|V\|^2)$
DELETE_MIN	$O(\|1\|)$	$(\|V\|)$	$O(\|V\|)$
DEC_BY_NAME	$O(\|1\|)$	$(\|E\|)$	$O(\|E\|)$

  If we use an array implementation of a priority queue then Dijkstra's algorithm run in

  $$T(|E|, |V|) = O(|V|^2) + O(|E|).$$

  Note that this simple implementation is as good as you can get for dense graphs where $|E| \in O(|V|^2)$. We would be done if all graphs were dense, but that is not the case and in fact many real life graphs are sparse. So we continue our quest to find an a priority queue data structure that is best tuned for the requirements of Dijkstra's algorithm.

- Binary Heap

  A simple (and efficient[1]) implementation of a priority queue, uses an array implementation but represents a rooted balanced binary tree where each node except the root and the leaves have a parent and satisfy the following *heap property*: $key(p(x)) \leq key(x)$, for min heap, otherwise the inequality is reversed. This structure supports, insert, delete, delete by position, delete min, and decrease key by position operations in logarithmic time (the depth of the tree), and find min is constant time.

  Note that the binary heap is not sufficient to implement all operations needed, you need to augment it by an array where each name points to the position in the heap where the element is located. However, this augmentation does not worsen the asymptotic complexity of the operations. We refer to this augmentation: $ARRAY + BINARY\_HEAP = NAME\_HEAP$.

  PQ(Name_Heap):

Operation	Cost	Times Used	Total Cost
FIND_MIN	$O(1)$	$(\lvert V\rvert)$	$O(\lvert V\rvert)$
DELETE_MIN	$O(\log\lvert V\rvert)$	$(\lvert V\rvert)$	$O(\lvert V\rvert \cdot \log\lvert V\rvert)$
DEC_BY_NAME	$O(\log\lvert V\rvert)$	$(\lvert E\rvert)$	$O(\lvert E\rvert \cdot \log(\lvert V\rvert)$

  If we use a binary heap as a priority queue then Dijkstra's algorithm runs in $O(|V| \cdot \log|V|) + O(|E| \cdot \log|V|)$, which is dominated by the second term, hence

  $$T(|E|, |V|) = O(|E| \cdot \log|V|).$$

- d-heap

  Can we tune PQ by increasing the top fan-in of tree representing the priority queue? The answer is yes. A d-heap is a priority queue data structure (satisfies the heap property) and each internal node, including the root node has d children.

  The depth of such a tree is $log_d n = \log n/\log d$. To perform an decrease key takes time proportional to the depth, since nodes only need to be compared and possibly swapped with the parents, but the time to delete the minimum node is the depth times d, since we need to promote the minimum weight child of each deleted or promoted node.

  PQ(d-Heap):

Operation	Cost	Times Used	Total Cost
FIND_MIN	$O(1)$	$(\lvert V\rvert)$	$O(\lvert V\rvert)$
DELETE_MIN	$O(d\frac{\log\lvert V\rvert}{\log d})$	$(\lvert V\rvert)$	$O(d \cdot \lvert V\rvert \cdot \frac{\log\lvert V\rvert}{\log d})$
DEC_BY_NAME	$O(\frac{\log\lvert V\rvert}{\log d})$	$(\lvert E\rvert)$	$O(\lvert E\rvert \cdot \frac{\log(\lvert V\rvert}{\log d})$

  The running time in this case is $O(d \cdot |V| \cdot \frac{\log|V|}{\log d}) + O(|E| \cdot \frac{\log(|V|}{\log d})$.

  Here we already have a parameter that we can use to additionally to tune the heap to get optimal performance. We want to get the best d such that the running time is minimized. This happens when neither term dominates the other, hence they are both equal:

  $$d \cdot |V| \cdot \frac{\log|V|}{\log d} = |E| \cdot \frac{\log(|V|}{\log d}$$

  Hence the value of the best $d = \frac{|E|}{|V|}$ and the worst case running time becomes

  $$T(|E|, |V|) = O(|E| \frac{\log|V|}{\log|E| - \log|V|}).$$

---

[1] No address arithmetic is required. Addresses of children of say element x, can be computed by multiplying the address of x by 2, which is just a shift.

Note that when the graph is moderately dense, $|E| > |V|^{1+\alpha}$ for any fixed constant α, this is $O(|E|)$, which seems optimal. On the other hand, if $|E| \in O(|V|)$, this is $O(|V| \log |V|)$ just as for a binary heap.

3 Amortized Analysis

Amortized analysis is a technique for showing tighter bounds on the worst case running time of an algorithm/data structure. This is accomplished by analyzing a series of operations as a whole and amortizing the total cost over the operations analyzed. Intuitively, there will be advantage to use amortized analysis when the actual cost of different operations is not the same. Sometimes an operation will be very expensive but other times it will be very cheap. In such cases worst case time analysis (where we take the worst case over all runs and do not amortize) is very pessimistic and the asymptotic time complexity you obtain is an overestimate of the real cost. The toy example from class is the binary counter. You have n-bit counter and the operation performed on the counter is INCREMENT (add 1 to the counter). You want to know what is the complexity of INCREMENT. You shouldn't be surprised that takes constant time to INCREMENT it. Why? Intuitively the least significant bit is flipped every time, the second to least significant bit is flipped every other time, and so on, to flip the most significant bit you need 2^{n-1} prior INCREMENTs. That is the intuition. (Similar example is a stack with a MULTIPOP operation. MULTIPOP clears the whole stack, although it sounds that MULTIPOP is expensive, and clearly could take O(n) operations, but could it take O(N) time twice in a row? No. You need at least O(n) PUSH operations before you get enough elements to do expensive MULTIPOP).

3.1 Potential Function Method

There are three techniques to perform amortized analysis, the aggregate method, the accounting method, but we will consider the most general one - the potential function method. The potential function, Φ, maps the state S of your data structure D, at any time t to the real numbers, so $\Phi : S_i \mapsto \mathbb{R}^+$. The art is to select a potential function such that the analysis of your data structure is tight (usually this is not a trivial matter). *It is important that* Φ *is non-negative* and you will see why shortly.

The framework is as follows. Suppose after a series of n operations $op_1, ..., op_n$ your data structure goes from state $S_1, S_2, ..., S_n$, respectively. Let $\Phi : S_i \mapsto \mathbb{R}^+$ be a potential function. Let the the actual cost of operation op_i be denoted by $Cost(op_i)$. Then define the *amortized cost* by: $AmCost(op_i) = Cost(op_i) + \Phi(S_i) - \Phi(S_{i-1})$. The goal is to bound Total Cost of all operations, which is the actual cost incurred by the algorithm, $TotalCost = \sum_{i=1}^{n} Cost(op_i)$.

$$\sum_{i=1}^{n} AmCost(op_i) = \sum_{i=1}^{n} Cost(op_i) - \Phi(S_0) + \Phi(S_n)$$

The sum of the amortized costs is the sum of the actual cost and a telescoping sum of the potential functions, of which only values of the potential function at time 0 and n remain, everything else cancels out.

$$TotalCost = \sum_{i=1}^{n} Cost(op_i) = \sum_{i=1}^{n} AmCost(op_i) + \Phi(S_0) - \Phi(S_n)$$

But remember that Φ is a non-negative function, and usually $\Phi(S_0) = 0$, hence $\Phi(S_0) - \Phi(S_n)$

is bounded by the maximum value Φ can get. Then we have

$$TotalCost = \sum_{i=1}^{n} Cost(op_i) = \sum_{i=1}^{n} AmCost(op_i) - \Phi(S_n)$$

And finally we bound the actual cost by the sum of the amortized costs of the operations

$$TotalCost = \sum_{i=1}^{n} Cost(op_i) \leq \sum_{i=1}^{n} AmCost(op_i)$$

Now suppose that you show that for all i $AmCost(op_i) \in O(f(n))$ and there are m operations, then the total cost of your algorithm will be

$$TotalCost \leq \sum_{i=1}^{m} AmCost(op_i) \in mO(f(n))$$

In the toy example above the potential function of the counter at time t is $\Phi_t(counter) =$ (the number of bits that are 1 in the counter), $\Phi_t = (\#1s)$. Suppose at time t the counter is incremented and as a result c consecutive bits are flipped from 1 to 0, and the $c+1$-st bit is flipped from 0 to 1. The actual cost of the operation is $c+1$. Say the potential function at time $t-1$ has value K then: $AmCost = Cost + \Phi_t - \Phi_{t-1} = c+1+K-c+1-K = 2 = O(1)$. Note that, if we don't start with an empty counter, then we still get the bound $TotalCost \leq TotalAmortizedCosts + \Phi(S_0)$. If the counter had D bits, and we perform m increments, this is $O(m + D)$. If we start with a counter full of 1's we could get the additional D right off, but this would clear the counter.

In the second toy example, we could look at a stack that has two operations $Push$ and $Pop(k)$ where $Pop(k)$ deletes the top k elements from the stack. We could use $\Phi(S)$ to be the size of the stack. Then a $Push$ operation has amortized cost 2, one for the operation, and the other for increasing Φ by 1. On the other hand, a $Pop(k)$ operation has cost $min(k, |S|)$, and either decreases $|S|$ by k or clears the stack if $k > |S|$. Thus, the amortized cost of the Pop operation is 0, since its actual cost is equal to the decrease in Φ. (In other words, by paying 2 for each Push, we have already paid for the consequent Pop whenever it happens.)

4 Fibonacci Heaps

The Fibonacci heap (F-heap is also used) data structure was designed by M. L. Fredman and R. E. Tarjan [1] with the purpose to show that Dijkstra can run in $O(|V|\log|V| + |E|)$. Fredman and Tarjan extended the binomial heap data structure created by Vullemin. And in fact if Dijkstra didn't need decrease key operation to run in $O(1)$ amortized cost, then the binomial heap would do just as fine as you will see later that the difference between binomial heap and Fibonacci heap is in implementation of decrease key operation.

4.1 Binomial heaps

The binomial heap is a collection of binomial trees. Each tree is a heap, in the sense that each element of the tree satisfies the heap property ($x.key \geq p(x).key$, if the heap is a min heap). A binomial tree of top fan-out k, denoted by B_k, has 2^k elements. A binomial tree B_{k+1} is created from two binomial trees B_k by making the tree with smaller root the root of the new tree and making the other tree a child. Note that the top fan-out of the new tree is $k+1$ and the number

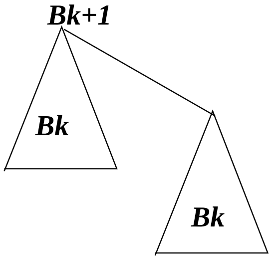

Figure 1: Binomial tree of $B_{k+1} = B_k + B_k$

of nodes is $2^k + 2^k = 2^{k+1}$. A few things to notice: 1) You make a tree of fan-out $k+1$ by merging (combining) two trees of the same fan-out k. 2) The number of comparisons required to merge two trees is constant since we only need to compare the roots and chose the one with smaller key as the root of the new tree (however, other pointers need to be adjusted to add the node to a list of siblings, and we also need to increment the degree counter).

The number of elements at depth i is $\binom{k}{i}$, hence its name[2]. In particular, it follows that $k \leq \log n$ for any binomial heap. We will ensure that we have at most one heap of any degree in our set at any time, by merging heaps of the same degree.

Binomial heaps can handle *Insert* and *DeleteMin* operations. Suppose a binomial heap contains the binomial trees $B_0, B_1, \ldots, B_k$, with B_i a degree i binomial heap, and no heap of degree $k+1$. Then adding a new element (which we represent as a one element heap B_0') will cause B_0 and B_0' to merge and create a new B_1' which will be merged with B_1 and so on. The resulting heap contains one binomial tree B_{k+1}. In the process we used $k+1$ comparisons and $O(k+1)$ pointer changes. But now this new heap has all $0, 1, 2, ..., K$ slots empty. The analysis of the binomial heap is exactly like the analysis of the toy example - binary counter. In fact, we can generalize the Insert operation to InsertTree, which inserts a binomial heap into the set, merging until the one heap of any degree invariant is restored.

Then to delete the minimum, we find the minimum root of and InsertTree each of its children. This takes amortized $O(k) = O(\log n)$ because the number of nodes we need to compare and the number of children are both bounded by the maximum possible degree.

4.2 Structure and analysis of Fibonacci heaps

Back to Fibonacci heaps. We will not give a complete description of a Fibonacci just yet, but will develop their structure as we discuss the operations and their amortized analysis. We need a more relaxed structure than the binomial heaps. Binomial heaps are nice, have clear structure, however maintaining invariants that each tree is a binomial tree, and the heap has at most one tree of given fan-out is costly. Namely operations like decrease key cannot be done efficiently (constant amortized cost, which is what we are after). So Fibonacci heaps behave as binomial heaps when you insert elements, or when were to use a combination of insert and delete min. This structure is not maintained during decrease key.

The structure of a Fibonacci heap (incomplete since a few important invariants are missing) is:

---

[2] You can show that by induction, using the recursive definition of the tree. Suppose the property holds for binomial tree of fan-out k, then the number of elements at depth i is $\binom{k}{i} + \binom{k}{i-1} = \binom{k+1}{i}$. You use induction to prove many properties of recursively defined structures like Binomial and Fibonacci heaps.

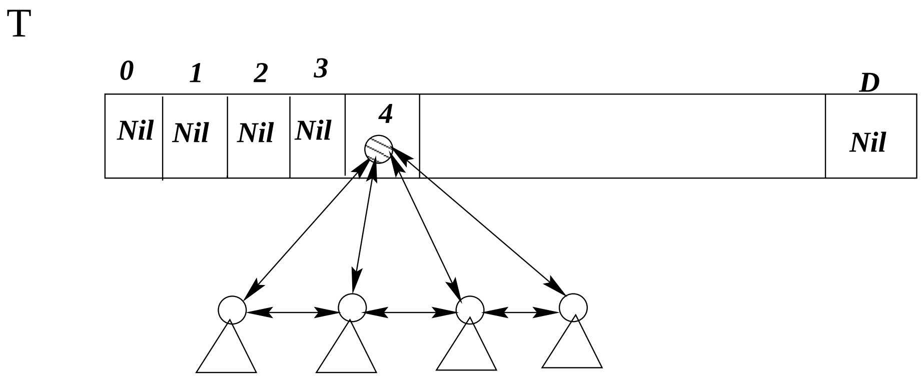

Figure 2: A Fibonacci heap T, in which only a root tree of degree 4 is present.

1. A Fibonacci heap, like a binomial heap, is a set of trees with the heap property.
2. Each tree has a distinct degree (fan-out of the root node of the tree), hence there is at most one tree of degree k.
3. The different trees of the heap involve disjoint elements.
4. The heaps are unordered, hence the minimum element of the heap can be found by probing the roots of all its trees[3].
5. For each element of the heap we maintain an accurate count of the fan-out of the heap rooted at that element, also referred to as degree.

Each node has a bunch of pointers, a **key** field, a **degree** field, and later we will add a **mark** field. In this representation, we assume that the maximum degree of a root tree of the heap is D. Later we will derive a bound of the size of each tree of degree k and then we will upper bound D (as a function on the total number of elements n stored in the heap.).

4.3 INSERT and DELETE MIN Operations

Let T be a Fibonacci heap, whose maximum degree is D. INSERT and DELETE MIN must preserve the invariant that in a Fibonacci heap there is at most one tree of degree k, for any $k \leq D$. We consider a generalized INSERT procedure where we insert not a single element but a whole tree r, whose degree is smaller than D.

```
INSERT TREE(r)
$d \leftarrow r.degree$;
if $T[d] = Nil$ then $T[d] = r$; HALT.
$s \leftarrow T[d]$;
$T[d] \leftarrow Nil$;
if $r.key < s.key$
then
```

---

[3] Alternatively, you can use a pointer to the minimum root of a root tree, and update in during each operation. However for our purpose does not change the asymptotic complexity of Dijkstra, since the number of FIND MINs is the number of DELETE MINs. But if you are using many FIND MIN operations then it would be wise to modify the structure to support $O(1)$ FIND MIN.

```
    $p(s) \leftarrow r$;
    attach $s$ to the end of $r$'s child list.
    $r.degree++$;
    $t \leftarrow r$;
else
    Do the same with $s$ and $r$ reversed...
INSERT TREE (t);
```

Define a potential function $\Phi : T \mapsto \mathbb{R}^+$ that maps the state of our Fibonacci heap T to the positive reals. This function will have two components, reflect the status of the data structure after INSERT TREE operation and the other after DECREASE KEY operation. We write $\Phi = \Phi_{INSERT} + \Phi_{DECREASE}$. We will arrange things so that Insert and DeleteMin operations do not change $\Phi_{Decrease}$ so that we can analyze these operations separately (and use them as subroutines in DeleteMin).

We consider the Φ_{INSERT} first. Note the similarity between the increment operation of a binary counter and the cost of insert tree routine. The actual cost $Cost_{INSERT} = c($#of consecutive full positions in T start
Let

$$\Phi_{INSERT} = c(\#\text{number of root trees in T}).$$

Let k be the number of consecutive full positions of T starting at position d. And let the number of trees before INSERT TREE operation be $N + k$. Then the change of the potential function is $\Delta\Phi = \Phi_{i+1} - \Phi_i = c(M + 1) - c(T + k) = -c.k$ Then

$$AmCost(insert\_tree)_{i+1} = Cost(insert\_tree)_{i+1} + \Delta\Phi == c.k - c.k + 1 = O(1)$$

To implement DELETE MIN remember that the rooted trees in the array T, representing the heap, are unordered, hence we need to search sequentially through T to find the minimum key element. Then we delete x, and using sequence of INSERT TREE operations we re-insert each child subtree of x to T.

```
DELETE MIN(T)
    Scan array T, and keep track of MIN=d, where d is index in T.
    Delete root of tree of degree d;
    Do $d$ INSERT TREE operations for each child.
```

The complexity of $DELETE\_MIN$ is $O(D)$, where D is the size of the array, which stores the rooted trees of the Fibonacci heap.

Intuitively, referring back to the binomial heap data structure, a binomial heap on n will contain at most $\log n$ binomial trees. The number of binomial trees is the size of our array T, hence $D \in O(\log n)$, so we have shown that the complexity of $DELETE\_MIN$ is $O(\log n)$. However, this is not an exact estimate, since the structure of a Fibonacci heap is different than that of a binomial heap. This bound on D although good up to a constant (as we will see $D \in O(\log_\phi n)$ is correct if the only operations performed on the heap were INSERT TREE and DELETE MIN. The precise proof that $D \in O(\log n)$) is postponed to Section 4.5.

4.4 DECREASE KEY operation

Here we use the MARK field. For an element x, $x.mark = T$, if some child of the tree rooted at x has been deleted. And it is set to true when no child has been deleted. (However, we reset it to false when x is made the root of a tree at the top level, in our set.)

DECREASE KEY (x, new key) essentially performs the following:

1. Change the key of x to the new key value.
2. Check whether the heap property is not violated, that is $x.key > p(x).key$ still holds, or the parent of x $p(x) = Nil$ (implying that the tree is a root tree of the heap), and if one of the conditions is true then HALT.
3. Otherwise remove x from the child list of $p(x)$ and decrement the degree of $p(x)$ by 1. Remember the original parent of x: $y \leftarrow p(x)$.
4. INSERT TREE (x).
5. Clear the mark of x: $x.mark \leftarrow FALSE$.
6. If the original mark of the parent of x was $TRUE$ then clear it $y.mark \leftarrow FALSE$ and then recursively execute lines 3-6.
7. Otherwise, $y.mark \leftarrow TRUE$.

Observations: Couple of things to note, first we always keep accurate count of the degree of each node. Second, if a node has a mark set to TRUE, then if a second child is attempted to be removed, then the tree is cut and INSERT TREE is called for that tree. Hence only one child can be removed before a subtree is re-inserted to T.

Analysis: Clearly DECREASE KEY(x) procedure follows the parent links in the tree that contain the node x towards the root of the tree until it either sees a node whose mark is clean (FALSE) and sets the mark to TRUE, or reaches a root node. Hence a candidate of the potential function is the number of marked nodes. So we set the second component of Φ to be the number of marked nodes in the Fibonacci heap.

$$\Phi_{DECREASE} = c'(\#\text{marked nodes})$$

hence the potential of a Fibonacci heap at time i is proportional to the number of root trees (non-nill slots in T) and the number of marked nodes (with mark set to true)

$$\Phi = \Phi_{INSERT} + \Phi_{DECREASE} = c(\#trees) + c'(\#\text{of marked nodes}).$$

Then the change of the potential function during DECREASE KEY is measured by the length of the path from x to the first node whose mark was set to TRUE. DECREASE KEY will clear all the marks along the path, except the last one will be set. In addition for each recursive execution of lines 3-6 we have a new INSERT-TREE procedure. Suppose DECREASE KEY(x) traverses chain of l marked nodes starting at x, hence the actual cost is $O(l)$. Now the change in the potential function is

$$\Delta\Phi_{DECREASE} = cl + c'(-l + 2)$$

Now for proper choice of the constants c, c' we can make

$$AmCost_{DECREASE} = Cost_{DECREASE} + \Delta\Phi_{DECREASE} = O(l) - l = O(1)$$

4.5 Maximum degree, size and other structural properties of Fibonacci heaps

Each node of a Fibonacci heap satisfies the following invariant:

Claim 1 *(Fibonacci property) Let x be any node in a Fibonacci tree, let $x.degree = d$ be the number of children of the node, and let $c_1, c_2, \ldots, c_d$ be the d children of x in the order of which they were added as children of x. Then $c_i.degree \geq i - 2$, for all $i = 1, \ldots, d$.*

Proof: At the time c_i was merged with x then the degree of x was at least $i - 1$, since $c_1, ..c_{i-1}$ were already children. But the INSERT TREE procedure merges trees with the same degree, hence $c_i.degree = x.degree \geq i - 1$ (at that time). But the DECREASE KEY procedure removes at most one child of any node, when a second child is removed then the tree is made a root tree and INSERT TREE is called. Therefore the degree could have gone down by at most 1. Therefore, the current degree of c_i has been decremented at most once, so now $c_i.degree \geq i - 2$, for all $i = 1, \ldots, d$. ▮

Now we want to estimate the size of the Fibonacci tree (not heap, just a root tree of a Fibonacci heap) whose degree is k. If no DECREASE KEY operations are performed then the Fibonacci heap is essentially a binomial heap. But DECREASE KEY operation makes the tree sparser. We want to lower bound the size of a Fibonacci tree. We look at the minimum possible size of the Fibonacci of say degree k.

Claim 2 *Let s be a Fibonacci tree of degree k then the number of nodes in the tree is $size(s_k) = size(s) \geq F_{k+2}$.*

Proof:

We count one for the node s itself and then we assume each of the children of s has minimum size, i.e., in what follows we assume $size_{(}s_m)$ is the size of the smallest fibonacci tree of degree m.

$$size(s) \geq 1 + \sum_{i=1}^{k} size(c_i.degree)$$

But the trees satisfy the Fibonacci invariant above,

$$size(s) \geq 2 + \sum_{i=2}^{k} size(i-2)$$

Now we use induction on d to complete the claim. Suppose for all $m = 1, k-2$ $size(s_m) \geq F_{m+2}$ then

$$size(s) \geq 2 + \sum_{i=2}^{k} size(i-2) = 1 + \sum_{i=1}^{k} F_i = F_{k+2}$$

To see that just unwind the sum and add $F_0 = 0$ to the sum. ▮

The bottom line is that $size(s_k) \geq F_{k+2} \geq (\frac{1+\sqrt{5}}{2})^k$. You can derive that by solving the characteristic equation for the Fibonacci sequence $x^2 - x - 1$, finding the roots of the equation, and deriving a closed form for the Fibonacci sequence.

Corollary: The maximum degree of any tree of a Fibonacci heap on n nodes is $\log_\phi n \in O(\log n)$.

5 Conclusion

To summarize our work. We considered the Single source shortest path problem in graphs with no negative weights on edges. Then we decided to examine a specific algorithm that solves a problem, with the goal in mind to achieve the best possible running time. In the process we parameterized Dijkstra by defining abstract operations that a data structure must satisfy and then we designed a data structure that is tuned to the algorithm. The result is that we have an optimal algorithm for the case $E \in \Omega(V \log V)$.

Now to put things in perspective, regarding the single source shortest path problem. If the graph is very dense $E \in O(V^2)$ then a trivial array implementation of a priority queue is as good as anything else. If the graph is sparser $E \in o(V \log V)$ then there are better algorithms then Dijkstra: Ahuja, Mehlhorn, Orlin, and Tarjan obtained $O(E + V \log W)$, where W is the maximum weight of a node, Thorup obtained $O(E \lg \lg V)$ algorithm for the directed case and $O(E + V)$ for the undirected case.

I don't meen to be pessimistic but although amazing the Fibonacci heap data structure, has too complex implementation, and hence find little to no practical applications. However the potential function method is a very powerful weapon to achieve tight analysis of algorithms.

References

[1] Michael L. Fredman and Robert Endre Tarjan, *Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms* Journal of ACM, 34(3), 1987.

[2] Thomas Cormen and Charles Leiserson and Ronald Rivest and Clifford Stein, *Introduction to Algorithms* MIT Press, 2001.