FIBONACCI HEAPS

12 Introduction

Priority queues are a classic topic in theoretical computer science. The search for a fast priority queue implementation is motivated primarily by two network optimization algorithms: Shortest Path (SP) and Minimum Spanning Tree (MST), i.e., the connector problem. As we shall see, Fibonacci Heaps provide a fast and elegant solution.
The following 3-step procedure shows that both Dijkstra's SP-algorithm or Prim's MST-algorithm can be implemented using a priority queue:

1. Maintain a priority queue on the vertices $V(G)$.

2. Put s in the queue, where s is the start vertex (Shortest Path) or any vertex (MST). Give s a *key* of 0. Add all other vertices and set their key to infinity.

3. Repeatedly delete the minimum-key vertex v from the queue and mark it *scanned*. For each neighbor w of v do: If w is not scanned (so far), decrease its key to the minimum of the value calculated below and w's current key:

   - SP: key(v) + length(vw),
   - MST: weight(vw).

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 $O(\log n)$, and decrease-key $O(\log n)$, where n reflects the number of elements in the heap.
If a graph has n vertices and e edges, then running either Prim's or Dijkstra's algorithms will require $O(n \log n)$ time for inserts and deletes. However, in

the worst case, we will also perform e decrease-keys, because we may have to perform a key update every time we come across a new edge. This will take $\mathrm{O}(e \log n)$ time. Since the graph is connected, $e \geq n$, and the overall time bound is given by $\mathrm{O}(e \log n)$. As we shall see, Fibonacci heaps allow us to do much better.

13 Definition and Elementary Operations

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*, *link*, and *delete-min* (we have not covered why *link* is a useful operation yet, but this will become clear later on). 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.

Definition 13.1: 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 *degree* (also known as the order or *rank*) 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.

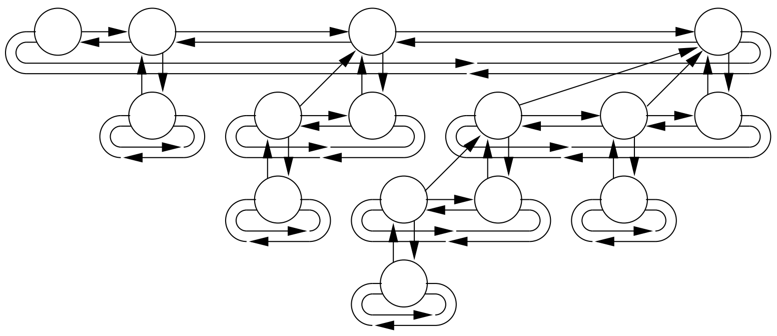

Fig. 9: A detailed view of a Fibonacci Heap. NULL pointers are omitted for clarity.

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). We have not described how the operations on Fibonacci heaps are implemented, and their implementation will add some additional properties to H. The following are some elementary operations used in maintaining Fibonacci heaps:

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.

Linking two trees x **and** y**:** Let x and y be the roots of the two trees we want to link; then if $key(x) \geq key(y)$, we make x the child of y; otherwise, we make y the child of x. We update the appropriate node's degrees and the appropriate child list; this takes $O(1)$ operations.

Cutting a node x**:** 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 degree of the parent of x is decremented, etc.). Again, this takes $O(1)$ operations. We assume that when we want to cut/find a node, we have a pointer *hanging* around that accesses it directly, so actually finding the node takes $O(1)$ time.

```
CLEANUP:
$newmin \leftarrow$ some root list node
for $i \leftarrow 0$ to $\lfloor \log n \rfloor$
    $B[i] \leftarrow$ NULL
for all nodes $v$ in the root list
    $parent(v) \leftarrow$ NULL
    unmark $v$
    if $key(newmin) > key(v)$
        $newmin \leftarrow v$
    LINKDUPES($v$)
```

```
LINKDUPES:
$w \leftarrow B[deg(v)]$
while $w \neq$ NULL
    $B[deg(v)] \leftarrow$ NULL
    if $key(w) \leq key(v)$
        swap $v$ and $w$
    remove $w$ from root list
    link $w$ to $v$
    $w \leftarrow B[deg(v)]$
$B[deg(v)] \leftarrow v$
```

Fig. 10: The CLEANUP algorithm executed after performing a *delete-min*

Marking a node x: 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 shall see later on that no marked node will lose a second child before it is cut itself.

13.1 The *delete-min* Operation

Deleting the minimum key node is a little more complicated. First, we remove the minimum key from the root list and splice its children into the root list. Except for updating the parent pointers, this takes $O(1)$ time. Then we scan through the root list to find the new smallest key and update the parent pointers of the new roots. This scan could take $O(n)$ time in the worst case. To bring down the *amortized* deletion time (see further on), we apply a CLEANUP algorithm, which links trees of equal degree until there is only one root node of any particular degree.
Let us describe the CLEANUP algorithm in more detail. This algorithm maintains a global array $B[1 \ldots \lfloor \log n \rfloor]$, where $B[i]$ is a pointer to some previously-visited root node of degree i, or NULL if there is no such previously-visited root node. Notice, the CLEANUP algorithm simultaneously resets the parent pointers of all the new roots and updates the pointer to the minimum key. The part of the algorithm that links possible nodes of equal degree is given in a separate subroutine LINKDUPES, see Figure 10. The subroutine

```
PROMOTE:
unmark $v$
if $parent(v) \neq$ NULL
    remove $v$ from $parent(v)$'s child list
    insert $v$ into the root list
    if $parent(v)$ is marked
        PROMOTE($parent(v)$)
    else
        mark $parent(v)$
```

Fig. 11: The PROMOTE algorithm

ensures that no earlier root node has the same degree as the current. By the possible swapping of the nodes v and w, we maintain the heap property. We shall analyze the efficiency of the *delete-min* operation further on. The fact that the array B needs at most $\lfloor \log n \rfloor$ entries is proven in Section 15, where we prove that the degree of any (root) node in an n-node Fibonacci heap is bounded by $\lfloor \log n \rfloor$.

13.2 The *decrease-key* Operation

If we also need the ability to delete an arbitrary node. The usual way to do this is to decrease the node's key to $-\infty$ and then use *delete-min*. We start by describing how to decrease the key of a node in a Fibonacci heap; the algorithm will take $O(\log n)$ time in the worst case, but the *amortized* time will be only $O(1)$. Our algorithm for decreasing the key at a node v follows two simple rules:

1. If $newkey(v) < key(parent(v))$, promote v up to the root list (this moves the whole subtree rooted at v).

2. As soon as two children of any node w have been promoted, immediately promote w.

In order to enforce the second rule, we now mark certain nodes in the Fibonacci heap. Specifically, a node is marked if exactly one of its children has been promoted. If some child of a marked node is promoted, we promote (and unmark) that node as well. Whenever we promote a marked node, we unmark it; this is the only way to unmark a node (if splicing nodes into the root list during a *delete-min* is not considered a promotion). A more formal description of the PROMOTE algorithm is given in Figure 11. This algorithm is executed if the new key of the node v is smaller than its parent's key.

14 Amortized Analysis

In an *amortized analysis*, time required to perform a sequence of data structure operations is averaged over all the operations performed. Amortized analysis can be used to show that the average cost of an operation is small, if one averages over a sequence of operations, even though a single operation might be expensive. Amortized analysis differs from average-case analysis in that probability is not involved; an amortized analysis guarantees that *average performance of each operation in the worst case.*

There are several techniques used to perform an amortized analysis, the method of amortized analysis used to analyze Fibonacci heaps is the potential method. When using this method we determine the the amortized cost of each operation and may overcharge operations early on to compensate for undercharges later. The potential method works as follows. We start with an initial data structure D_0 on which s operations are performed. For each $i = 1, \dots, s$, we let c_i be the actual cost of the i-th operation and D_i be the data structure that results after applying the i-th operation to the data structure D_{i-1}. A potential function Φ maps each data structure D_i to a real number $\Phi(D_i)$, which is the *potential* (energy) associated with the data structure D_i. The *amortized cost* $\hat{c}_i$ of the i-th operation with respect to the potential function Φ is defined by:

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}). \tag{1}$$

The amortized cost of each operation is thus its actual cost plus the increase in potential due to the operation. The total amortized costs of the s operations is

$$\sum_i \hat{c}_i = \sum_i c_i + \Phi_{D_s} - \Phi_{D_0}, \tag{2}$$

If we can prove that $\Phi_{D_s} \geq \Phi_{D_0}$, then we have shown that the amortized costs bound the real costs. Thus, we can analyze the amortized costs to obtain a bound on the actual costs. In practice, we do not know how many operations s might be performed. Therefore, if we require that $\Phi(D_i) \geq \Phi_{D_0}$ for all i, then we guarantee that we pay in advance. It is often convenient to define $\Phi(D_0) = 0$ and then to show that $\Phi(D_i) \geq 0$.

Intuitively, if the potential difference $\Phi(D_i) - \Phi(D_{i-1})$ of the i-th operation is positive, then the amortized cost $\hat{c}_i$ represents an overcharge to the i-th operation, and the potential of the data structure increases. If the potential difference is negative, then the amortized costs represents an undercharge and the actual cost of the operation is paid by a decrease in the potential.

14.1 Amortized Analysis of the *delete-min* and *decrease-key* Operation

Define $\Phi(H)$ as the number of root nodes $t(H)$ plus two times the number of marked nodes $m(H)$ in the Fibonacci heap H, i.e., $\Phi(H) = t(H) + 2m(H)$. We assume that a single unit of potential can pay for a constant amount of work, where the constant is sufficiently large to cover the cost of any of the specific constant-time pieces of work that we might encounter. Assume that a Fibonacci heap application begins with no heaps (this is the case for both the SP and MST algorithm). The initial potential, therefore, is 0, and obviously the potential is nonnegative at all subsequent times. Hence, the total amortized cost is thus an upper bound on the total actual cost for the sequence of operations (see Eq. (2)). We further assume that there is some upper bound $D(n)$ on the maximum degree of any node in an n-node Fibonacci heap. We derive this upper bound in Section 15.

The actual cost of a delete-min operation can be accounted for as follows. An $O(1)$ contribution comes from splicing the (at most $D(n)$) children of the minimum node in the root list (because setting the parent pointers to Null and unmarking the nodes is done by the Cleanup algorithm). The size of the root list upon calling the Cleanup algorithm is $D(n) + t(H) - 1$, since it consist of the original $t(H)$ root list nodes, minus the minimum node, plus the children of the extracted node. Meaning, at most $D(n) + t(H) - 1$ link operations are performed. Thus, the total amount of work performed is at most proportional to $D(n) + t(H)$, i.e., $O(D(n) + t(H))$. The potential before extracting the minimum node is $t(H) + 2m(H)$, and the potential afterward is at most $(D(n) + 1) + 2m(H)$, since at most $D(n) + 1$ roots remain and no nodes become marked during the operation. The amortized cost is thus at most

$$\begin{aligned}
&O(D(n)+t(H))+((D(n)+1)+2m(H))-(t(H)+2m(H)) \\
&\quad = \quad O(D(n))+O(t(H))-t(H) \\
&\quad = \quad O(D(n)),
\end{aligned}$$

since we can scale up the units of the potential to dominate the hidden constant in $O(t(H))$. Intuitively, the cost of performing the link operations is paid by the decrease in the potential due to reducing the number of nodes on the root list.

Let us now consider the *decrease-key* operation. Decreasing the key has an actual cost of $O(1)$. Suppose that c recursive invocations of the PROMOTE function are called. Each recursive call takes O(1) time except for the recursive calls, hence, the actual cost of *decrease-key* is O(c). Next, we compute the change in potential. Each recursive call, except for the last, cuts a marked node and unmarks the node. Afterward, there are $t(H)+c$ trees (the original $t(H)$, $c-1$ trees produced by the recursive calls, and the tree rooted at the node whose key was decreased). Whereas the maximum number of marked nodes equals $m(H)-c+2$ ($c-1$ were unmarked and the last call may have marked a node). The change in potential is therefore at most

$$((t(H)+c)+2(m(H)-c+2)-(t(H)+2m(H))=4-c. \tag{3}$$

Thus, the amortized cost of the *decrease-min* is at most

$$O(c)+4-c=O(1), \tag{4}$$

by scaling up the units of the potential to dominate the hidden constant in $O(c)$.

15 Bounding the Maximum Degree

In order to prove that the amortized time of the *delete-min* operation is $O(\log(n))$, we must show that $D(n)$ is bounded by $O(\log n)$. In this section we shall show that $D(n) \leq \lfloor \log_\phi n \rfloor$, where $\phi = (1+\sqrt{5})/2$ is the golden ratio.

LEMMA 15.1: Let x be any node in a Fibonacci heap, and suppose that $d(x) = k$, where $d(x)$ denotes the degree of x. Let $y_1, y_2, \ldots, y_k$ denote the children of x in the order in which they were linked to x, from the earliest to the latest. Then, $d(y_1) \geq 0$ and $d(y_i) \geq i-2$, for $i = 2, 3, \ldots, k$.

Proof: Obviously, $d(y_1) \geq 0$. For $i \geq 2$, we note that when y_i was linked to x, all of $y_1, y_2, \ldots, y_{i-1}$ were children of x, so we must have had $d(y_i) \geq i-1$, as y_i was only linked to x if $d(x) = d(y_i)$. Since then, node y_i has lost at most one child, otherwise y_i would have been cut. We may conclude $d(y_i) \geq i-2$. Q.E.D.

Let us now define the Fibonacci numbers F_k, for $k \geq 0$ as follows: $F_0 = 0$, $F_1 = 1$ and $F_k = F_{k-1} + F_{k-2}$, for $k \geq 2$. Then, we can easily show by induction on k that $F_{k+2} = 1 + \sum_{i=0}^{k} F_k$, for all $k \geq 0$ (simply apply induction on the term F_{k+1}). Moreover, $F_{k+2} \geq \phi^k$ (apply induction on both terms and use the fact that $(1+\phi) = \phi^2$).

THEOREM 15.1: Let x be any node in a Fibonacci heap, and let $k = d(x)$. Then, $size(x) \geq F_{k+2} \geq \phi^k$, where $\phi = (1+\sqrt{5})/2$.

Proof: Let s_k denote the minimum possible value of $size(z)$ over all nodes z such that $d(z) = k$. That is, s_k denotes the minimum number of nodes in a tree that is rooted by a degree k root node. Let $y_1, y_2, \ldots, y_k$ denote the children of x as in Lemma 15.1 in the order they were linked to x. To compute a lower bound on $size(x)$, we count one for x itself, one for the first child y_1 and then apply Lemma 15.1 for the remaining children. We have

$$size(x) \geq s_k \geq 2 + \sum_{i=2}^{k} s_{i-2}. \tag{5}$$

We now show by induction that $s_k \geq F_{k+2}$, for all $k \geq 0$. The cases for $k = 0$ and 1 are trivial. By induction, we have $s_i \geq F_{i+2}$ for $i = 0, \ldots, k-1$, therefore,

$$\begin{aligned} s_k &\geq 2 + \sum_{i=2}^{k} s_{i-2} \\ &\geq 1 + F_1 + \sum_{i=2}^{k} F_i = F_{k+2}. \end{aligned}$$

Thus, we have shown $size(x) \geq s_k \geq F_{k+2} \geq \phi^k$.

Q.E.D.

Corollary 15.1: The maximum degree $D(n)$ of any node in an n-node Fibonacci heap is bounded by $\lfloor \log_\phi n \rfloor$, meaning it is $O(\log n)$.

Proof: Let x be any node of an n-node Fibonacci heap and let k be its degree. By the previous theorem we have $n \geq size(x) \geq \phi^k$. Taking the base-$\phi$ logarithms yields $k \leq \log_\phi n$. Therefore, the maximum degree is $O(\log n)$.

Q.E.D.

As a results of this corollary, Prim's MST algorithm (or Dijkstra's SP) has an amortized cost of $O(n \log n + e)$, as a *decrease-key* operation is $O(1)$ and a *delete-min* operations is $O(\log n)$.