MATH 8650 Project Report
Fibonacci Heaps

Aishwarya Gujarathi
Madhumita Krishnan

December 8, 2017

1 Introduction

Priority queues are a classic topic in theoretical computer science.Fast algorithms for network optimizations problems such as finding single-source shortest path or computing minimum spanning trees(MST) of a graph require an efficient implementation of priority queue data structure. Fibonacci heaps provide an elegant solution for such an implementation as they are designed using **amortized analysis**. It is a method of analyzing the costs associated with a data structure that averages the worst operations out over time. Thus, even though a single operation might be expensive, if one averages over a sequence of operations, the average cost of that operation is small. This is particularly important when one of the operations performed on the data structure is very costly compared to the others.

Fibonacci heaps are **collection of min-heap-ordered-trees, which support mergeable-heap operations INSERT, FIND-MIN, EXTRACT-MIN, MERGE, along with operations DECREASE-KEY and DELETE. All the operations except for EXTRACT-MIN and DELETE run in O(1) amortized time**. Theoretically, Fibonacci heaps are desirable when the number of EXTRACT-MIN and DELETE operations is small relative to other operations performed. The network optimizations algorithms for graphs mentioned above may require DECREASE-KEY operation once per edge. For dense graphs, which has many edges, the O(1) amortized time of each DECREASE-KEY adds up to a big improvement over $\Theta(logn)$ worst-case time of binary or binomial heaps, where n is the number of nodes in the priority queue. The time complexities of Dijkstra's algorithm(shortest path) for a graph of m edges and n vertices, would drop from O(m log n) to O(m + n log n), which is asymptotically faster than before. These bounds are obtained by using "lazy" implementation, i.e. 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.

Since Fibonacci heaps are primarily used to minimize the number of operations needed to compute the MST or shortest path, the kind of operations that we are interested in are INSERT, DECREASE-KEY, LINK and EXTRACT-MIN. For this project, these operations are implemented in python.

2 Solution

2.1 Structure of Fibonacci heaps

Fibonacci heap H is a collection of min-heap-ordered trees having the properties given below:

1. The trees follow **minimum-heap** property so that the priority of the parent is always grater than the priority of children, i.e. the weight or key associated with the parent node is always less than associated with the children.
2. The trees are rooted but unordered.
3. The root of each tree thus contains minimum node of the tree.
4. The roots of all the trees are linked together into a circular, doubly linked list called **root list**.
5. The heap structure maintains a pointer **min[H]** to the overall minimum node which is essentially one of the roots of the trees. This pointer is used to access the heap H.
6. The children in the individual trees are also connected with their siblings using circular, doubly linked list called as **child list**.
7. n[H] maintains the total number of nodes in the Fibonacci heap.
8. A node x contains following to support the circular, doubly linked list:

   (a) A pointer to its parent, p[x].
   (b) pointer to any of its children, child[x]
   (c) A pointer to its left and right siblings, left[x] and right[x].
   (d) Degree/rank = number of children, degree[x].
   (e) Boolean value field to indicate whether it is marked or not, mark[x].
   (f) key[x] which is the priority associated with the node.

The node will be marked whenever we cut one of its child during EXTRACT-MIN operations. We will use these notations to describe various operations in section 2.2.

2.1.1 Potential function

For amortized analysis of Fibonacci heap operations, we consider a concept called as potential function. The equation of potention function of Fibonacci heap H is given as:

$$\Phi(H) = t(H) + 2m(H) \tag{1}$$

where, t(H) is the number of trees in the root list of H and m (H) is the number of marked nodes in H.

The potential of a set of Fibonacci heaps is the sum of the potentials of its constituent Fibonacci heaps. We shall assume that a 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. We assume that a Fibonacci heap application begins with no heaps. The initial potential, therefore, is 0, and by equation 1 , the potential is nonnegative at all subsequent times. The upper bound on the total amortized cost is an upper bound on the total actual cost for the sequence of operations.

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, . . . , 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 Di1. 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 $\hat{c}$ of the i^{th} operation with respect to the potential function Φ is defined by:

$$\hat{c} = c_i + \Phi(D_i) + \Phi(D_{i-1}) \tag{2}$$

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:

$$\sum_i \hat{c} = \sum_i c_i + \Phi(D_s) + \Phi(D_0) \tag{3}$$

2.2 Amortized analysis of Operations on Fibonacci heaps

Operations INSERT, EXTRACT-MIN, DECREASE-KEY are of primary importance for efficient implementation of Fibonacci heaps in fast algorithms for graph problems. INSERT AND EXTRACT_MIN are **mergeable-heap operations**. If only these are performed on the Fibonnaci heap, then the heap is a collection of **unordered binomial trees**. An unordered binomial tree is like a binomial tree, defined recursively. The unordered binomial tree U_0 consists of a single node, and an unordered binomial tree U_k consists of two unordered binomial trees U_{k-1} for which the root of one is made into any child of the root of the other. One of the properties of unordered binomial tree is that:
For the unordered binomial tree U_k, the root has degree k, which is greater than that of any other node. The children of the root are roots of subtrees U_0, U_1, . . . , U_{k-1} in some order. Thus, if an n-node Fibonacci in heap is a collection of unordered binomial trees, then

$$D(n) = logn \tag{4}$$

where **D(n)** is a **known upper bound** on the maximum degree of any code in an n-node Fibonacci heap.

For getting a good time bound using amortized analysis, the mergeable operations are implemented lazily. This means that we maintain the binomial tree structure(which makes number of trees in Fibonacci heaps small), which is useful for improving performance of EXTRACT-MIN, only when EXTRACT-MIN is called. If we go on maintaining the tree structure at every INSERT or MERGE operation, then it can take upto $\Omega(logn)$ time. Thus, work is delayed creating a performance trade-off among implementation of various operations. This reducing number of trees operation is known as consolidation, and is performed only when is really needed to extract the minimum element.

2.2.1 Inserting a node

Inserting a node x into Fibonacci heap H, just adds a node x to its root list and adds this node in a single-node tree. The node has to be filled with key[x].

To determine the amortized cost of INSERT, let H be the input Fibonacci heap and H' be the resulting Fibonacci heap. Then, t(H') = t(H) + 1 and m(H') = m(H), and the increase in potential is:

$$((t(H)+1)+2m(H))-(t(H)+2m(H))=1 \tag{5}$$

The actual cost of inserting is O(1), hence the **amortized cost of INSERT** according to equation 2 is:

$$O(1)+1=\mathbf{O(1)} \tag{6}$$

2.2.2 Extracting the minimum node

Extracting minimum node, along with deleting the minimum node, performs the delayed operation of maintaing the structure of the tree/ reducing number of trees, for ease of finding the new minimum node, i.e. it performs consolidation of trees in the root list of Fibonacci heap H.

To determine the amortized cost of EXTRACT-MIN, let H denote the Fibonacci heap just prior to the EXTRACT-MIN operation. The actual cost of extracting the minimum node can be accounted for as follows. An O(D(n)) contribution comes from there being at most D(n) children of the minimum node that are being processed.
The size of the root list upon calling CONSOLIDATE is at most D(n) + t(H) - 1, since it consists of the original t(H) root-list nodes, minus the extracted root node, plus the children of the extracted node, which number at most D(n). Every time through the while loop of lines 6-12 in CONSOLIDATE is called, one of the roots is linked to another, and thus the total amount of work performed in the for loop is at most proportional to D(n) + t(H). Thus, the total actual work is 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:

$$\begin{aligned}O(D(n)+t(H))+((D(n)+1)+2m(H))-(t(H)+2m(H)) &= O(D(n))+O(t(H))-t(H)\\ &= O(D(n)) \qquad (7)\end{aligned}$$

We have scaled up the unit of potential to dominate the constant hidden in t(H). Using equation 4, the **amortized cost of EXTRACT-MIN** becomes **O(log n)**.

2.2.3 Decreasing Key

This operation is replaces the key of node x with value k such that k< key[x] in the Fibonacci heap H.
DECREASE-KEY works as follows. Firstly, the key to be decreased to i.e. y is checked if it really less than key[x] as seen in lines 1-2. Then the new decreased key value is assigned

to key[x], if heap-order is not violated. Otherwise, CUT tree rooted at x and meld into root list. Unmark if already marked. Mark p[x] if it is unmarked(hasn't lost a child yet).

Now, there could be two situations: p[x] = y might have lost x as its only child or x could have been the second child it lost. Losing of second child could create a problem where the tree rooted in y number of nodes not exponential in degree[y]. Hence, we use CASCADING-CUT on y to CUT the y and meld it to the root list, unmarking it, and marking the parent of y, until you find a parent which is not marked or is a root of one of the trees in the root list of H. In simple words if heap order is violated while decreasing the key:

1. Decrease key of x.
2. Cut tree rooted at x, meld into root list, and unmark.
3. If parent p of x is unmarked (hasn't yet lost a child), mark it;
4. Otherwise, cut p, meld into root list, and unmark

Do steps 3 and 4 recursively for all ancestors that lose a second child.
To determine the amortized cost of DECREASE-KEY, lets consider the actual cost. The DECREASE-KEY procedure takes O(1) time, plus the time to perform the cascading cuts. Suppose that CASCADING-CUT is recursively called c times from a given invocation of DECREASE-KEY. Each call of CASCADING-CUT takes O(1) time exclusive of recursive calls. Thus, the actual cost of DECREASE-KEY, including all recursive calls, is O(c).

For change in potential,let H denote the Fibonacci heap just prior to the DECREASE-KEY operation. Each recursive call of CASCADING-CUT, except for the last one, cuts a marked node and clears the mark bit. Afterward, there are t(H) + c trees (the original t(H) trees, c- 1 trees produced by cascading cuts, and the tree rooted at x) and at most m(H) - c + 2 marked nodes (c - 1 were unmarked by cascading cuts and the last call of CASCADING-CUT may have marked a node). The change in potential is therefore at most:

$$((t(H) + c) + 2(m(H) - c + 2)) - (t(H) + 2m(H)) = 4 - c \tag{8}$$

Thus, the **amortized cost of DECREASE-KEY** is actual cost plus change in potential given as:

$$O(c) + 4 - c = \mathbf{O(1)} \tag{9}$$

We scale up the units of potential to dominate the constant hidden in C(c).

3 Results

Fibonacci heaps have better amortized running time as compared to binary heaps or binomial heaps.It performs better with the operations INSERT and DECREASE-KEY. Thus, we check our Fibonacci heap implementation for these operations.
In-order to cross validate the costs of our implementation, we timed to function calls for n/2 INSERT on Fibonacci heap H and plotted it for size[H] = n being 10000, 20000, 40000, 80000, 160000 and 320000. The plot shown in figure 1 was obtained.

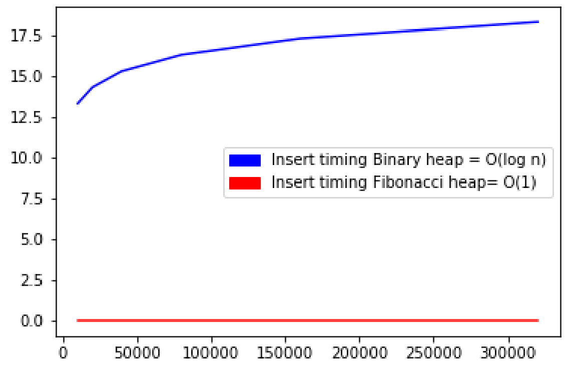

Figure 1: Comparing INSERT for binary and Fibonacci heap implementations

The timings obtained were 4.19, 8.47, 16.9, 33.5, 66.8, 133 in nano seconds. While plotting, the timings obtained for the inserts were divided by the size n, so that it can be compared with O(log n), which is cost of INSERT called on binary heap.
For checking timings for DECREASE-KEY, we first inserted the n items, extracted the minimum, so that binomial trees are created and then decreased the key of n/2 element to 0. The timings obtained were 531, 516, 532, 519, 519, 522 in nano seconds. While plotting, the timings were scaled by 100, as even though the cost is O(1), the constant is high as compared to the timings for O(log n), which is the cost of DECREASE-KEY for binary heap. Figure 2 shows the plot for this comparison.

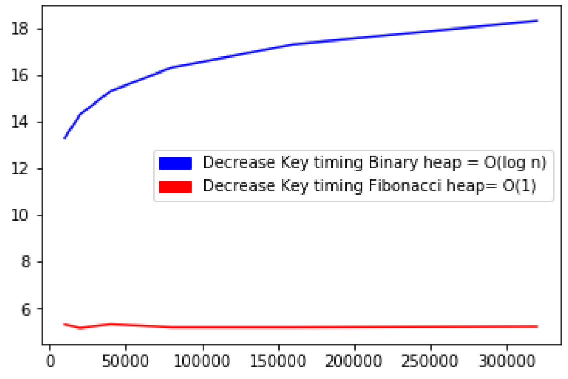

Figure 2: Comparing DECREASE-KEY for binary and Fibonacci heap implementations

4 Conclusion

Fibonacci heaps are desirable when number of EXTRACT-MIN and DELETE operations is small relative to the number of other operations performed as these cost O(log n), where n is number of elements in the heap.
For e.g. Graph algorithms such as computing single-source shortest paths(Dijkstra) or computing MST(Prims Algorithm) may call Decrease-Key once per edge. For dense graphs, edges ¿ vertices, O(1) amortized time of each Decrease-Key call adds up to a big improvement over O(log n) worst-case time of binary or binomial heaps.
Practically, high constant factors(for O(1)) and programming complexity(no. of pointers) makes it less desirable. Implementations such as Pairing heap perform better.
Use of python simplifies the implementation of circular doubly linked list as compared to languages like C and C++ as use of pointers is easy. Also, Fibonacci heap implementation comprises of an elements node which is itself has different structure than the Fibonacci heap structure and hence, OOP concept of inheritance is useful in this case. We can easily inherit structure of the node from Node class for use in structure of the heap in Fibonacci heap class.

5 References

1. Introduction to Algorithms by Charles E. Leiserson, Clifford Stein, Ronald Rivest and Thomas H. Cormen

2. https://www.cs.princeton.edu/courses/archive/spring13/cos423/lectures/FibonacciHeaps-2x2.pdf