Zhejiang University

College of Computer Science and Technology

Strip Packing

Project6 Report by Group1

Author:
Guo Jiahao
Wu Yihang
Sun Xinjie

Supervisor:
Mao Yuchen

Abstract:

An Assignment submitted for the ZJU:

21120491 Advanced Data Structure and Algorithm Analysis

May 23, 2022

1 Introduction of the Project

In this project, we need to solve the strip packing problem. The strip packing problem is a 2-dimensional geometric minimization problem. Given a set of axis-aligned rectangles and a strip of bounded width and infinite height, determine an overlapping-free packing of the rectangles into the strip minimizing its height. We may assume that the width of any rectangle is no more than the width of the bin, and we are not allowed to rotate the rectangles, and that the rectangles should not overlap.

Note that we need to implement at least two polynomial-time approximation algorithms for this problem. We must generate test cases of different sizes with different distributions of widths and heights, compare and analysis the solution quality and the running time of the two algorithms on these test cases.

2 Introduction of the Algorithms

2.1 Next Fit

We first discuss the simplest method to solve this problem which called next fit algorithm. In this algorithm, We place the items one by one. When the bottom edge of the next item cannot be placed at the same height as the previous item, we cover the "cover" of the previous layer, that is, place the bottom edge of the next item on the height of the top of the highest item on the current layer. Pseudo code is shown below:

Algorithm 1: Next Fit

Input: width w and height h of input items

let y = *0*;
let x = *0*;
let h = *0*;
for *Rectangle R = (w, h)*in the sequence **do**
    Try to fit the rectangle onto the current open shelf;
    if It does not fit **then**
        close the current shelf and open a new one;
    end
end

2.2 Implementation of Operations

2.2.1 Insertion

Insertion for Fibonacci heap is a lazy operation. The following procedure inserts node x into Fibonacci heap H:

First we initialize some attributes of node x. Next, we test if Fibonacci heap H is empty. If it is, we make x be the only node in Hs root list and set $H.min$ to point to x. Otherwise, we insert x into Hs root list and update $H.min$ if necessary. Finally, we increment $H.n$.

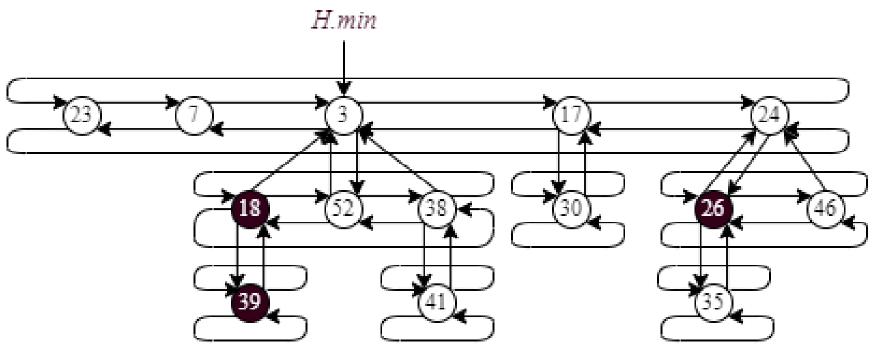

Figure 1: An example of a Fibonacci heap

Algorithm 2: Insertion for Fibonacci heap

Input: Fibonacci heap H, node x

```
x.degree = 0;
x.p = x.child = NULL;
x.mark = false;
if H.min == NULL then
    create a root list for H containing just x;
    H.min = x;
else
    insert x into H's root list;
    if x.key < H.min.key then
        H.min = x;
    end
end
H.n = H.n + 1
```

2.2.2 Make heap

For the making heap operation of Fibonacci heap, we just need to insert n keys continuously.

2.2.3 FindMin

Because we maintain a pointer to the minimum node of the heap, so we only need to return *H.min.data*.

2.2.4 DeleteMin

DeleteMin is much more complicated than insertion. It works by first making a root out of each of the minimum nodes children and removing the minimum node from the root list. It then consolidate the root list by linking roots of equal degree until at most one root remains of each degree. The pseudocode of Algorithm 2 extracts the minimum node.

The procedure consolidate uses an auxiliary array A, whose size can be limited by the bounding the maximum degree

$$D(n) \leq \lfloor \log_{\phi} n \rfloor, \quad \phi = (1+\sqrt{5})/2$$

, to keep track of roots according to their degrees. If $A[i] = y$, then y is currently a root with $y.degree = i$.

In detail, let's see the pseudocode of Algorithm 3, the consolidate procedure works as follows. We first allocate array A, then we link roots together, w may be linked to some other node and no longer be a root. Nevertheless, w is always in a tree rooted at some node x, which may or may not be w itself. Because we want at most one root with each degree, we look in the array A to see whether it contains a root y with the same degree as x. If it does, then we link the roots x and y but guaranteeing that x remains a root after linking. That is, we link y to x after first exchanging the pointers to the two roots if ys key is smaller than xs key. After we link y to x, the degree of x has increased by 1, and so we continue this process, linking x and another root whose degree equals xs new degree, until no other root that we have processed has the same degree as x. We then set the appropriate entry of A to point to x, so that as we process roots later on, we have recorded that x is the unique root of its degree that we have already processed. When this for loop terminates, at most one root of each degree will remain, and the array A will point to each remaining root.

Algorithm 3: DeleteMin for Fibonacci heap

```
Input: Fibonacci heap H
z = H.min;
if z != NULL then
    for each child x of z do
        add x to the root list of H;
        x.p = NULL
    end
    remove z from the root list of H;
    if z == z.right then
        H.min = NULL;
    else H.min = z.right
        consolidate(H);
    end
    H.n = H.n - 1
end
```

2.2.5 Merge

We don't need this procedure in this project, so I just describe this procedure without pseudocode. To merge two Fibonacci heaps H_1 and H_2, we first concatenate the root lists of H_1 and H_2 into a new root list H. Then we set the minimum node of H and new $H.n$.

2.2.6 DecreseKey

DecreseKey operation is crucial for the optimization for Dijkstra's algorithm because it has only $O(1)$ amortized time. To implement the operation, we first judge if the min-heap order has not been violated, if it is, we don't need to adjust the position of the node. If min-heap order has been violated, many changes may occur. We start by cutting procedure, which cuts the link between x and its parent y, making x a root.

Algorithm 4: Consolidate for Fibonacci heap

```
Input: Fibonacci heap H, node x
let A be a new array;
for i = 0 to size of A do
    A[i] = NULL;
end
for each node w in the root list of H do
    x = w;
    d = x.degree;
    while A[d] != NULL do
        y = A[d];
        if x.key > y.key then
            exchange x with y;
        end
        remove y from the root list of H;
        make y a child of x, incrementing x.degree;
        y.mark = false;
        A[d] = NULL;
        d = d + 1;
    end
    A[d] = x;
end
H.min = NULL;
for i = 0 to size of A do
    if A[d] != NULL then
        if H.min == NULL then
            create a root list for H containing just A[i];
            H.min = A[i];
        else
            insert A[i] into H's root list;
            if A[i].key < H.min.key then
                H.min = A[i];
            end
        end
    end
end
```

We use the mark attributes to obtain the desired time bounds. As soon as the second child has been lost, we cut x from its parent, making it a new root, and sometimes we need cascading-cut operation. Once all the cascading cuts have occurred, the procedure can finish up by updating *H.min* if necessary. The only node whose key changed was the node x whose key decreased. Thus, the new minimum node is either the original minimum node or node x.

Algorithm 5: DecreseKey for Fibonacci heap

```
Input: Fibonacci heap H, node x, new key value k
if k > x.key then
    error"new key is greater than current key";
end
x.key = k;
y = x.p;
if y!= NULL and x.key < y.key then
    remove x from the child list of y, decrementing y.degree;
    add x to the root list of H;
    x.p = NULL;
    x.mark = false;
    cascading_cut(H, y);
end
if x.key < H.min.key then
    H.min = x;
end
```

Algorithm 6: Cascade cut for Fibonacci heap

```
Input: Fibonacci heap H, node y
z = y.p;
if z!= NULL then
    if y.mark == false then
        y.mark = true;
    else
        remove y from the child list of z, decrementing z.degree;
        add y to the root list of H;
        y.p = NULL;
        y.mark = false;
        cascading_cut(H, z);
    end
end
```

2.2.7 Delete

For deletion, we just need to use decrease key function to decrease the data stored in the node we want to delete to a small value that normal node won't store.

3 Algorithm Analysis

3.1 Heap Operations

In this section, we define n as the number of elements in priority queue. And the result in the table represent the worst time, **except the column for Fibonacci heap, which represent amortized time**.

Operation	**Binary heap**	**Leftist heap**	**Binomial heap**	**Fibonacci heap**
Make heap	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Find min	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Insert	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(1)$
Delete min	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
Merge	$O(n)$	$O(\log n)$	$O(\log n)$	$O(1)$
Delete	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
Decrese key	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(1)$

Table 1: Running times of all operations of different heaps

3.2 optimization for Dijkstra's Algorithm

We have learned that theoretically, the total running time of Dijkstra's algorithm without priority queue is $O(|E|+|V|^2)$. So if the graph is dense, with $|E| = \Theta(|V|^2)$, the algorithm without priority queue is good enough.

But in practice, also in this project, the grapg is always sparse, which means $|E| = \Theta(|V|)$, the algorithm is too slow. So we consider the heap operation DeleteMin, whose running time is $O(n\log n)$ in all the heaps we use in the project, to find and delete the minimum vertex. And we use DecreseKey, or in practice insertion, whose running time is $O(n\log n)$ for binary heap, leftist heap, binomial heap, and $O(1)$ for Fibonacci heap.

The improtant observation is that the running time of Dijkstra algorithm is dominated by $|E|$ DecreseKey/Insertion operation and $|V|$ DeleteMin operations. So theoretically, for binary heap, leftist heap and binomial heap, the time bound for the optimized Dijkstra algorithm is $O(|E|\log|V|+|V|\log|V|) = O(|E|\log|V|)$, and for Fibonacci heap, the time bound will be decreased to $O(|E|+|V|\log|V|)$.

4 Experiment and Result

5 Discussion and Conclusion