UM-SJTU Joint Institute
Introduction to Algorithms
VE477 Lab4 Report

Name: Taoyue Xia, ID:518370910087

Date: 2021/11/10

1. Implementation of Fibonacci Heap

The class of Fibonacci Heap is shown below:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.parent = None
        self.child = None
        self.left = None
        self.right = None
        self.degree = 0
        self.mark = False

    # remove a child of the node
    def remove_child(self, node):
        if self.child is None:
            raise ValueError("No children")
        if self.degree == 1:
            self.child = None
        else:
            if self.child is node:
                self.child = node.right
            node.left.right = node.right
            node.right.left = node.left
        self.degree -= 1

    # add a child to the node
    def add_child(self, node):
        if self.child is None:
            self.child = node
            node.right = node.left = node
        else:
            self.child.right.left = node
            node.right = self.child.right
            node.left = self.child
            self.child.right = node
        node.parent = self
        self.degree += 1
        node.mark = False


class FibonacciHeap:
    def __init__(self):
        self.make_heap()
    # initialize the heap
    def make_heap(self):
        self.min_node = None
        self.total_nodes = 0
        self.root_num = 0

    # return the node with minimum data
    def minimum(self):
        return self.min_node

    # add a new node to the root list, helper function of insert
```

```python
    def merge_to_list(self, node):
        if self.min_node is None:
            node.left = node.right = node
        else:
            node.right = self.min_node.right
            node.left = self.min_node
            self.min_node.right.left = node
            self.min_node.right = node
        self.root_num += 1

    # insert a new node
    def insert(self, data):
        n = Node(data)
        n.left = n.right = n
        if self.min_node is None:
            self.min_node = n
        else:
            self.merge_to_list(n)

        if self.min_node is None or self.min_node.data > n.data:
            self.min_node = n
        self.total_nodes += 1

    # combine two fibonacci heaps
    def union(self, fib):
        h = FibonacciHeap()

        if self.min_node is None:
            h.min_node = fib.min_node
        else:
            h.min_node = self.min_node
            h.min_node.right.left = fib.min_node.left
            fib.min_node.left.right = h.min_node.right
            h.min_node.right = fib.min_node
            fib.min_node.left = h.min_node

        if fib.min_node.data < self.min_node.data:
            h.min_node = fib.min_node
        h.total_nodes = self.total_nodes + fib.total_nodes
        h.root_num = self.root_num + fib.root_num
        return h

    # cut off a child, helper in decrease key
    def cut(self, child, parent):
        if child.mark:
            child.mark = False
        parent.remove_child(child)
        child.parent = None
        self.merge_to_list(child)

    # cut until find an unmarked node or reach root
    def cascading_cut(self, node):
        x = node.parent
        if x:
            if not node.mark:
                node.mark = True
            else:
                self.cut(node, x)
```

```
                self.cascading_cut(x)

    # decrease a node's key, and decide whether it should be put into root list
    def decrease_key(self, node, data):
        if data > node.data:
            raise ValueError("new data is larger than current data")
        node.key = data
        x = node.parent
        if x and node.data < x.data:
            self.cut(node, x)
            self.cascading_cut(x)
        if node.data < self.min_node.data:
            self.min_node = node

    # remove a node from the root list, used in extract_min
    def remove_root_node(self, node):
        node.right.left = node.left
        node.left.right = node.right
        self.root_num -= 1

    # link one node with its children to another
    def link(self, n1, n2):
        self.remove_root_node(n1)
        n2.add_child(n1)

    def consolidate(self):
        x = [None] * self.total_nodes
        m = self.min_node
        num = self.root_num
        for i in range(num):
            temp = m
            m = m.right
            deg = temp.degree
            while x[deg] is not None:
                temp1 = x[deg]
                if temp.data > temp1.data:
                    self.link(temp, temp1)
                else:
                    self.link(temp1, temp)
                x[deg] = None
                deg += 1
            x[deg] = temp if temp.data < temp1.data else temp1
        self.min_node = None
        for i in range(len(x)):
            if x[i]:
                if self.min_node is None:
                    self.min_node = x[i]
                else:
                    if x[i].data < self.min_node.data:
                        self.min_node = x[i]

    # extract the minimum node from the root list, and transform
    def extract_min(self):
        m = self.min_node
        if m is not None:
            c = m.child
            if c is not None:
```

```python
                for i in range(m.degree):
                    r = c.right
                    self.merge_to_list(c)
                    c.parent = None
                    c = r
            self.remove_root_node(m)
            if m == m.right:
                self.min_node = None
            else:
                self.min_node = m.right
                self.consolidate()
            self.total_nodes -= 1
        return m

    # delete a node from the heap
    def delete(self, node):
        self.decrease_key(node, -float("inf"))
        self.extract_min()

```

2. Time complexity of each operation

Operation	Time Complexity
MakeHeap	$\Theta(1)$
Minimum	$\Theta(1)$
Union	$\Theta(1)$
Delete	$\Theta(\log n)$
Insert	$\Theta(1)$
ExtractMin	$\Theta(\log n)$
DecreaseKey	$\Theta(1)$

3. Comparison with min-heap

Operation	Min-heap	Fibonacci Heap
MakeHeap	$\Theta(1)$	$\Theta(1)$
Minimum	$\Theta(1)$	$\Theta(1)$
Union	$\Theta(n)$	$\Theta(1)$
Delete	$\Theta(\log n)$	$\Theta(\log n)$
Insert	$\mathcal{O}(\log n)$	$\Theta(1)$
ExtractMin	$\Theta(\log n)$	$\Theta(\log n)$
DecreaseKey	$\mathcal{O}(\log n)$	$\Theta(1)$

- Advantage:

  We can obviously see from the above table that for the operations `Union`, `Insert` and `DecreaseKey`, Fibonacci heap will run faster than min-heap.

- Disadvantage:

  Fibonacci heap is harder to implement.

4. Preference of Fibonacci Heap

If some algorithms need more operations of `Union`, `Insert` and `DecreaseKey`, fibonacci heap is preferred.

For **Dijkstra Algorithm**, the operation `DecreaseKey` is used in every loop. Therefore, if using fibonacci heap, the time complexity will be reduced from $\mathcal{O}((V+E)\log V)$ to $\mathcal{O}(V\log V+E)$.