CS261P Project 2
Performance Analysis of Binary Heap and Fibonacci Heap in Selection Problem

Yifeng Zhang
21406209
yifengz7@uci.edu

Chen Cui
47764330
ccui@uci.edu

December 1, 2019

1 Introduction

In this project we are to solve the selection problem using binary heap and Fibonacci heap, and compare their performance basing on different test cases.

We denote the selection problem as follows. Given an non-negative array of integers with length n, find the k largest elements in the array (the elements can be in any order).

The intuitive method to complete the task would require sorting. We can do this simply by sorting the array in descending order, and output the first k elements in the array. However, sorting the whole array takes a lot of space (memory), and we would like to use heaps instead of sorting to minimize the memory usage. For that reason, we used min-heaps instead of max-heaps. See following sections for detailed explanation.

2 Binary Heap

A binary heap is defined as a binary tree with additional constraints:

1. **Shape property:** a binary heap is a complete binary tree; that is, all levels of the tree, except possibly the last one (deepest) are fully filled, and, if the last level of the tree is not complete, the nodes of that level are filled from left to right.

2. **Heap property:** the key stored in each node is either greater than or equal to or less than or equal to the keys in the node's children, according to some total order.

Heaps where the parent key is less than or equal to are called min-heaps. We implemented efficient (logarithmic time) algorithms for inserting an element, and removing the smallest from the min-heap.

3 Fibonacci Heap

Fibonacci heap is a data structure for priority queue operations, consisting of a collection of heap-ordered trees. It has a better amortized running time than many other priority queue data structures including the binary heap and binomial heap.

For the Fibonacci heap, the find-minimum operation takes constant amortized time. The insert and decrease key operations also work in constant amortized time. Deleting an element works in logarithm amortized time. This means that starting from an empty data structure, any sequence of a insert and decrease key operations and b delete operations would take $O(a + b \log n)$ worst case time, where n is the maximum heap size. In a binary or binomial heap such a sequence of operations would take $O((a + b) \log n)$

time. A Fibonacci heap is thus better than a binary or binomial heap when b is smaller than a by a non-constant factor. It is also possible to merge two Fibonacci heaps in constant amortized time, improving on the logarithmic merge time of a binomial heap, and improving on binary heaps which cannot handle merges efficiently.

4 Explanation about Min-heap

In this project, our implementation of binary heap and Fibonacci heap are min-heaps. As described above, this is for the sake of minimizing memory usage.

We start with an empty heap, and simply add the first k elements in the array to the heap. For the $k+1^{th}$ and following elements, compare each of them with the minimum element in the heap, if the element in the array is larger, extract the minimum element from the heap, and insert the element in the array into the heap. Repeat the process until we are done with the last element in the array.

At this time, the heap contains the k largest elements in the array.

5 Code

```
class FibheapNode:
    def __init__(self, key):
        self.key = key
        self.degree = 0
        self.p = 0
        self.child = 0
        self.left = 0
        self.right = 0
        self.mark = False

    def MakeChild(self, cn):
        cn.p = self
        self.degree = self.degree + 1
        if self.child == 0:
            self.child = cn
            cn.left = cn
            cn.right = cn
        else:
            c = self.child
            cn.right = c
            cn.left = c.left
            cn.left.right = cn

    def search_child(self, key):
        w = self
        v = self
        while True:
            if w.key == key:
                return w
            if w.child != 0:
                res = w.child.search_child(key)
            if res != 0:
                return res
            w = w.right
            if w == v:
                return 0

    def printNode(self):
        if self.child != 0:
            c = self.child
            cl = c.left
            while 1:
                print("%d␣has␣child␣%d" % (self.key, c.key))
                c.printNode()
                if cl == c:
```

```
                    break
                c = c.right


class FibHeap:
    def __init__(self, n, minx):
        self.n = n
        self.min = minx

    def insert(self, x):
        if self.min == 0:
            self.min = x
            x.left = x
            x.right = x
        else:
            x.right = self.min
            x.left = self.min.left
            x.left.right = x
            self.min.left = x
            if x.key < self.min.key:
                self.min = x
        self.n += 1

    def find_min(self):
        return self.min

    def Link(self, y, x):
        y.right.left = y.left
        y.left.right = y.right
        y.p = x
        x.degree = x.degree + 1
        y.mark = False
        if x.child == 0:
            x.child = y
            y.left = y
            y.right = y
        else:
            c = x.child
            y.right = c
            y.left = c.left
            y.left.right = y
            c.left = y

    def delete_min(self):
        i = self.min
        if i != 0:
            if i.child != 0:
                j = i.child
                while j.p != 0:
                    a = j.left
                    j.p = 0
                    j.left = 0
                    j.right = 0
                    j.right = self.min
                    j.left = self.min.left
                    j.left.right = j
                    self.min.left = j
                    j = a
            i.right.left = i.left
            i.left.right = i.right
            if i == i.right:
                self.min = 0
            else:
                self.min = i.right
                self.Consolidate()
            self.n -= 1
        return i
```

```
def Consolidate(self):
    A = []
    for i in range(0, int(math.log(self.n, 2) + 1)):
        A.append(0)
    w = self.min
    t = w.left
    while 1:
        temp = w.right
        x = w
        d = x.degree
        while A[d] != 0:
            # print(A[d])
            y = A[d]
            if x.key > y.key:
                v = x
                x = y
                y = v
            self.Link(y, x)
            A[d] = 0
            d = d + 1
        A[d] = x
        if w == t:
            break
        w = temp

    self.min = 0
    for i in range(0, int(math.log(self.n, 2) + 1)):
        if A[i] != 0:
            if self.min == 0:
                self.min = A[i]
                A[i].left = A[i]
                A[i].right = A[i]
            else:
                A[i].right = self.min
                A[i].left = self.min.left
                A[i].left.right = A[i]
                self.min.left = A[i]
                if A[i].key < self.min.key:
                    self.min = A[i]

def find_node(self, key):
    w = self.min
    res = 0
    if w.key > key:
        return 0
    else:
        cr = w
        while 1:
            if cr.key == key:
                return cr
            else:
                if cr.child != 0:
                    res = cr.child.search_child(key)
                if res != 0:
                    return res
                cr = cr.right
                if cr == w:
                    return 0

def PrintHeap(self):
    root = self.min
    c = root
    print("min␣node␣is␣%d" % c.key)
    c.printNode()
    c = c.right
    while c != root:
        print("root␣number␣is␣%d" % c.key)
        c.printNode()
```

```
            c = c.right

class BinaryHeap:
    def __init__(self):
        self.__heap = []
        self.__last_index = -1

    def push(self, value):
        self.__last_index += 1
        if self.__last_index < len(self.__heap):
            self.__heap[self.__last_index] = value
        else:
            self.__heap.append(value)
        self.__siftup(self.__last_index)

    def pop(self):
        if self.__last_index == -1:
            raise IndexError('pop␣from␣empty␣heap')

        min_value = self.__heap[0]

        self.__heap[0] = self.__heap[self.__last_index]
        self.__last_index -= 1
        self.__siftdown(0)

        return min_value

    @property
    def min(self):
        return self.__heap[0]

    def __siftup(self, index):
        while index > 0:
            parent_index, parent_value = self.__get_parent(index)

            if parent_value <= self.__heap[index]:
                break

            self.__heap[parent_index], self.__heap[index] =\
                self.__heap[index], self.__heap[parent_index]

            index = parent_index

    def __siftdown(self, index):
        while True:
            index_value = self.__heap[index]

            left_child_index, left_child_value = self.__get_left_child(index, index_value)
            right_child_index, right_child_value = self.__get_right_child(index, index_value
                )

            if index_value <= left_child_value and index_value <= right_child_value:
                break

            if left_child_value < right_child_value:
                new_index = left_child_index
            else:
                new_index = right_child_index

            self.__heap[new_index], self.__heap[index] =\
                self.__heap[index], self.__heap[new_index]

            index = new_index

    def __get_parent(self, index):
        if index == 0:
            return None, None
```

```
        parent_index = (index - 1) // 2

        return parent_index, self.__heap[parent_index]

    def __get_left_child(self, index, default_value):
        left_child_index = 2 * index + 1

        if left_child_index > self.__last_index:
            return None, default_value

        return left_child_index, self.__heap[left_child_index]

    def __get_right_child(self, index, default_value):
        right_child_index = 2 * index + 2

        if right_child_index > self.__last_index:
            return None, default_value

        return right_child_index, self.__heap[right_child_index]

    def __len__(self):
        return self.__last_index + 1

    def __str__(self):
        return str(self.__heap)

    def __repr__(self):
        return str(self)
```

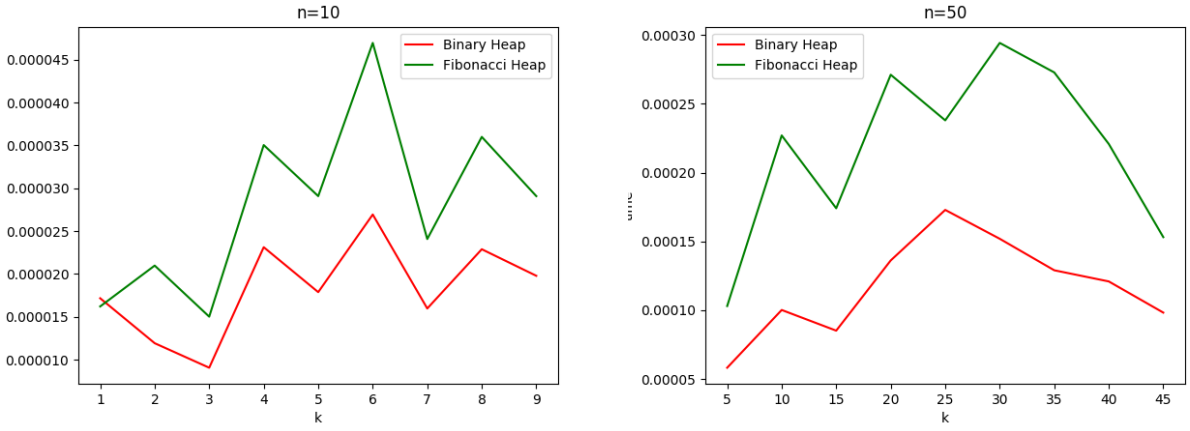
6 Metrics and Evaluation

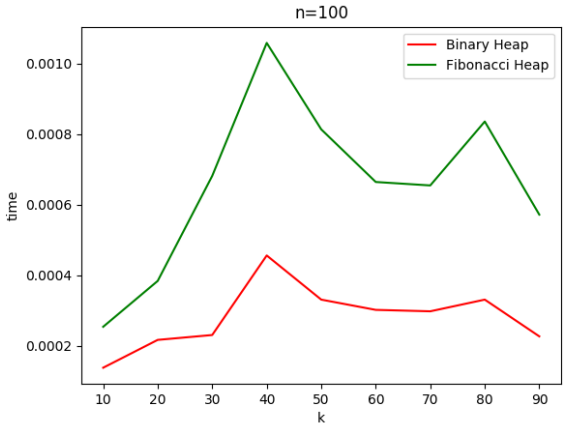
After testing the correctness of our implementation, we evaluated the selection algorithm two heaps. There are many factors to be considered.

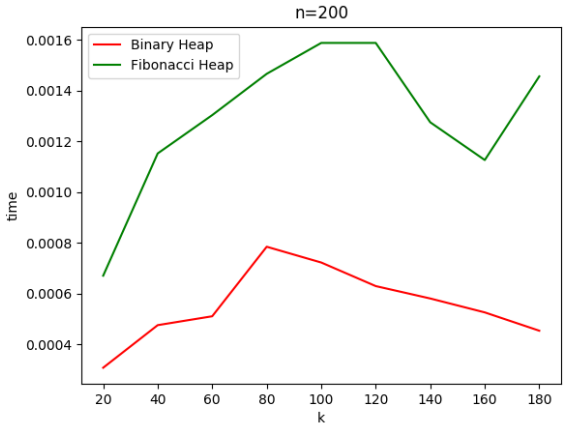
Here we defined the actual running time for the algorithm to be the time elapsed between the start of building the empty heap, and the end We used the actual running time to determine the performance of each algorithm.

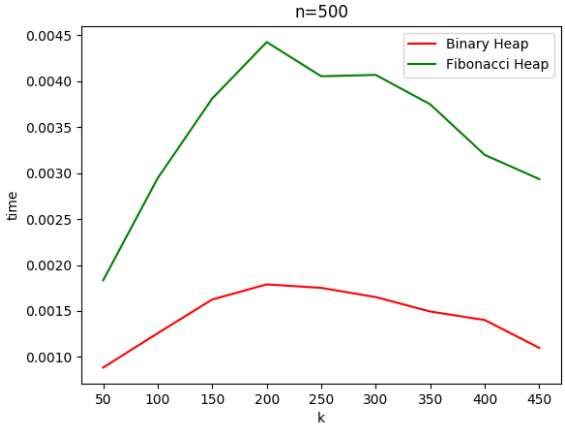
We tested the algorithms using different n and k, and plotted one figure for each n, in which the x-axis is k, and the y-axis is the running time of an algorithm.

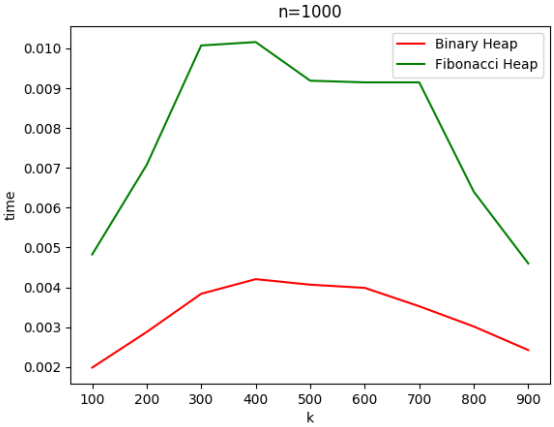
7 Figures

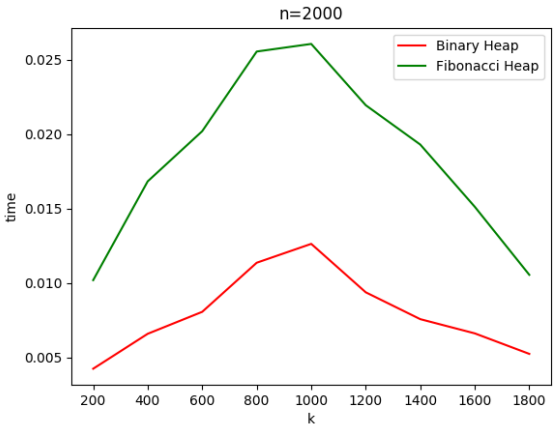


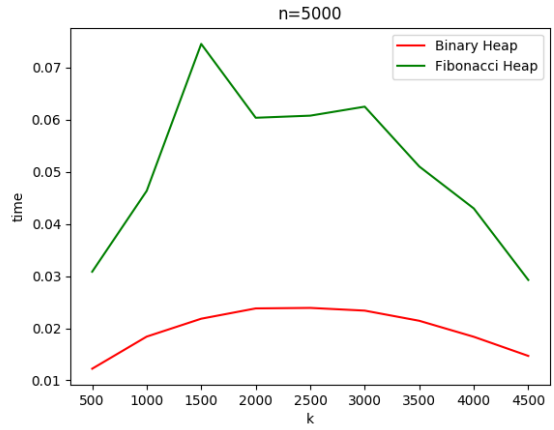

n=100
Binary Heap
Fibonacci Heap
time
k
0.0010
0.0008
0.0006
0.0004
0.0002
10 20 30 40 50 60 70 80 90

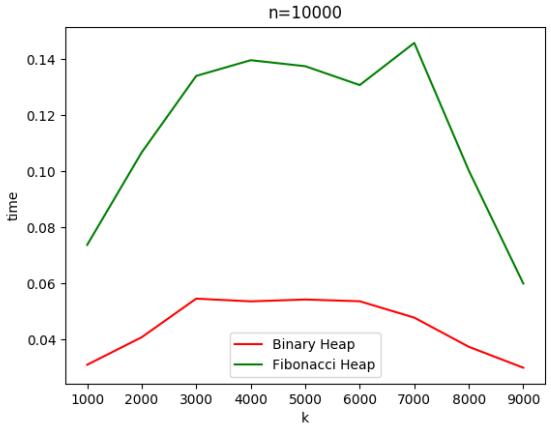

n=200
Binary Heap
Fibonacci Heap
time
k
0.0016
0.0014
0.0012
0.0010
0.0008
0.0006
0.0004
20 40 60 80 100 120 140 160 180

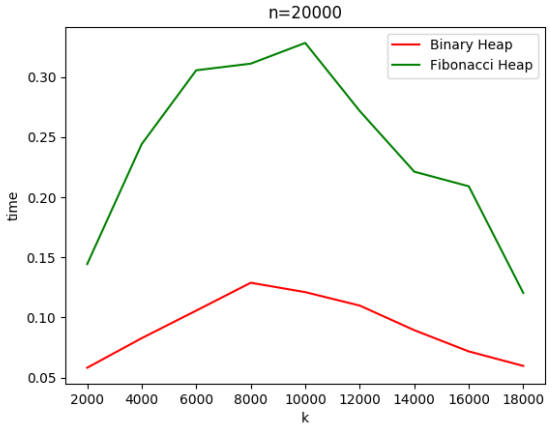

n=500
Binary Heap
Fibonacci Heap
time
k
0.0045
0.0040
0.0035
0.0030
0.0025
0.0020
0.0015
0.0010
50 100 150 200 250 300 350 400 450

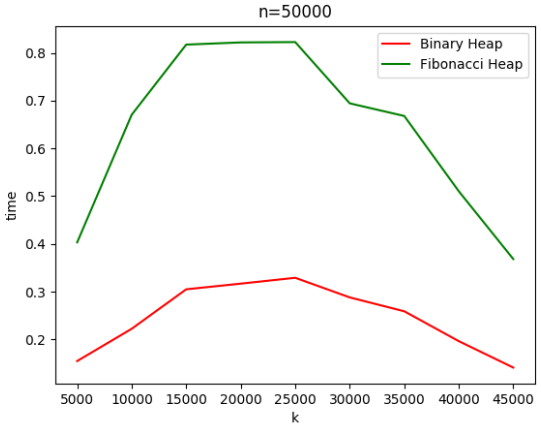

n=1000
Binary Heap
Fibonacci Heap
time
k
0.010
0.009
0.008
0.007
0.006
0.005
0.004
0.003
0.002
100 200 300 400 500 600 700 800 900

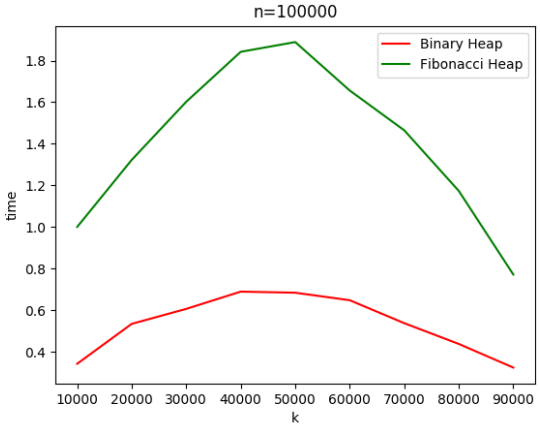

n=2000
Binary Heap
Fibonacci Heap
time
k
0.025
0.020
0.015
0.010
0.005
200 400 600 800 1000 1200 1400 1600 1800

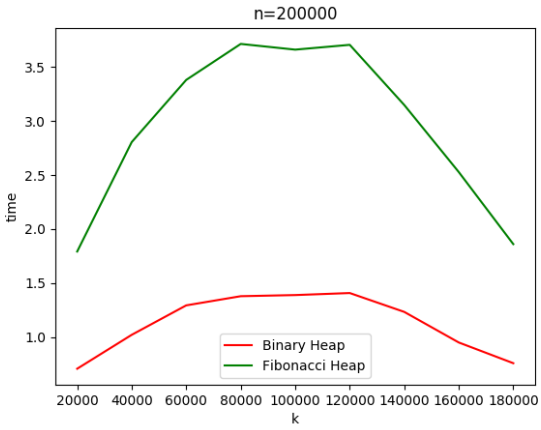

n=5000
Binary Heap
Fibonacci Heap
time
k
0.07
0.06
0.05
0.04
0.03
0.02
0.01
500 1000 1500 2000 2500 3000 3500 4000 4500

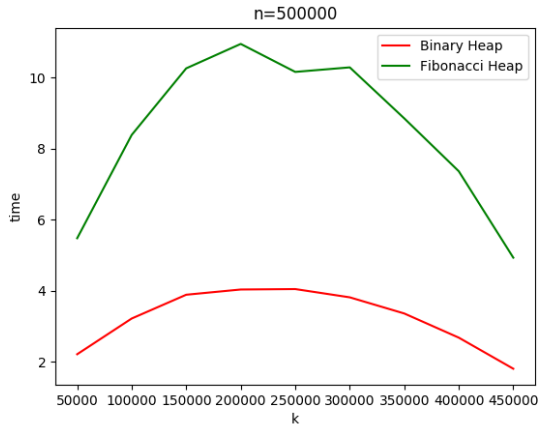

n=10000
time
k
Binary Heap
Fibonacci Heap

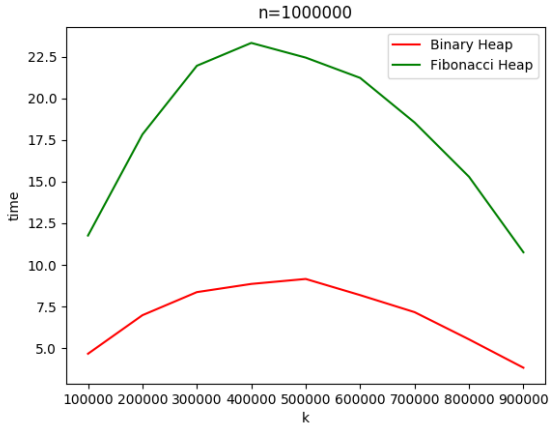

n=20000
time
k
Binary Heap
Fibonacci Heap


n=50000
time
k
Binary Heap
Fibonacci Heap


n=100000
time
k
Binary Heap
Fibonacci Heap


n=200000
time
k
Binary Heap
Fibonacci Heap


n=500000
time
k
Binary Heap
Fibonacci Heap



8 Analysis

1. We observed that when k is near a half of n, the actual running time for the both heap algorithms would reach the maximum (with n fixed).

   The explanation for this observation is pretty straightforward. Note that for both heaps, building the heap of size k takes $O(k)$ time, and substituting the rest $n-k$ elements takes $O((n-k)\log k)$ time. So the total time complexity would be $O(k+(n-k)\log k)$. For $f(k) = k+(n-k)\log k$, its derivative $f'(k) = 1 + \frac{n}{k} - (\log k + 1) = \frac{n}{k} - \log k$. Draw the graph of $f'(k)$ for arbitrary $n > 0$ and $n > k > 0$, we can observe that $f'(k)$ is positive when $k < x$, where x is the solution for k to $k \log k = n$; when $k > x$, $f'(k)$ is always negative.

2. We observed that in the selection problem, Fibonacci heaps are much less efficient than binary heaps. This can be caused by the fact that Fibonacci heaps are much more complex than binary heaps, and thus need more inner operations (such as combining two trees and marking deleted nodes) than binary heaps, resulting in longer running time.