Project 2 Shortest Path Algorithm with Heaps

- Lin Juyi - 3180103721
- Lu Tian - 3180103740

1. Architecture of the Project

We implement this project in Java and use Maven for project management and building automation. As a result, the project is organized in a "Maven" way. The implementation of all the Heap is inside "/src/main/java/heap/" the Dijkstra and the performance test code is within the class "algorithm.Dijkstra". Also, we adopt the idea of Test Driven Development(TDD), and constructed a set of unit test in "src/test/java" before actual development procedure.

The test data is in "/dataset/". And the test result is in "/Results/". You can also run the main method in "algorithm.Dijkstra", relaunch the test and generate the result if you like.

2. Introduction of the 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. A Fibonacci heap in figure 1 is a collection of trees satisfying the heap property.

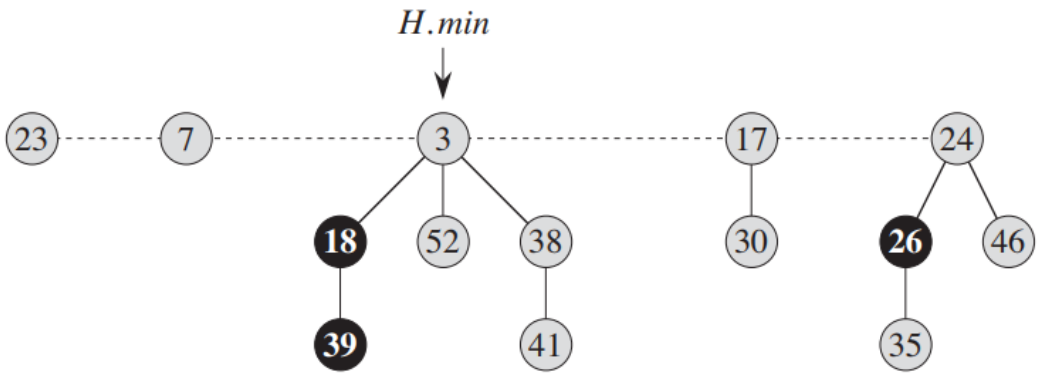


figure 1

In our implementation, the node of the tree is organized in a doubled linked list. Each node maintains the value, key and the degree of the current node. Also the node also store the pointer to its parent and leftmost child.

```
private class Node {

    public boolean mark;
    public int degree;
    public K key;
    public V value;
    public Node before, after, parent, children;
}
```

The FibonacciHeap keep track of node with the minimum key. Also, in order to support the DECREASE-KEY operation in $\Theta(1)$ time complexity, the HashMap of keyToNodeMap is introduced to accelerate the look up operation.

```
public class FibonacciHeap<K extends Comparable<K>, V> implements Heap<K, V> {
    // ......

    // ptr to the minNode is also the ptr to the rootNodeList
    private Node minNode;
    private int N;
    private Map<K, Node> keyToNodeMap;

    // .......
}
```

The push operation is rather simple and swift —— we simply insert the new node into the list of the root nodes and update the map. Which will take only constant time.

```
@Override
public void push(K key, V value) {
    Node theNode = new Node(key, value);
    keyToNodeMap.put(key, theNode);
    insertIntoRootList(theNode);
    N++;
}
```

The pop operation first insert the children of the minNode into the root node list and then remove the minNode. After that the delayed work of consolidating the trees in the root list finally occurs. The consolidate operation aims to make the trees within the root list only have unique degree(the number of children). The consolidate operation makes sure that the root list wouldn't expand too long. Intuitively thinking, if the root list is so long that it is degenerates to a linked list with the pointer to its minimum node. In that case, he pop operation will have to iterate through the list to find the minimum node after the pop, which is undesirable.

```
@Override
public V pop() {
    assert (minNode != null);

    Node ret = minNode;
    if (ret != null) {
        // 1. insert the child node into root node list
        Node theNode = minNode.children;
        while (theNode != null) {
            Node nextNode = theNode.after;
            removeFromDoubleLinkedList(theNode);
            insertAfter(minNode, theNode);
            theNode.parent = null;
            theNode = nextNode;
        }

        // 2. remove the current minNoe
        minNode = ret.after;
        detachNode(ret);
        keyToNodeMap.remove(ret.key);
        N--;
```

```
        // 3. consolidate
        consolidate();
    }
    return ret.value;
}

private void consolidate() {
    Map<Integer, Node> degreeToNode = new HashMap<>();
    Node currentNode = minNode;
    minNode = null;

    while (currentNode != null) {
        Node next = currentNode.after;
        removeFromDoubleLinkedList(currentNode);
        int degree = currentNode.degree;
        while (degreeToNode.get(degree) != null) {
            Node other = degreeToNode.get(degree);
            currentNode = mergeTwoNode(currentNode, other);
            degreeToNode.remove(degree);
            degree = currentNode.degree;
        }
        degreeToNode.put(degree, currentNode);
        currentNode = next;
    }

    for (Node node : degreeToNode.values()) {
        insertIntoRootList(node);
    }
}
```

The decreaseKey operation of the heap is not as complicated as the pop operation. We firstly look up the node in the map and then update its key. If the key is less than the key of its parent, we cut the node from its parent and insert it into the root node list.

In cascadeCut operation, why would we bother doing that? Why would we cut the parent from the grandparent node if the grandparent node has been marked?. The answer is that the cascade operation makes sure that the tree wouldn't have too much kid and become too flat in some layers. Otherwise, the flat tree will take much time to insert all these children of the flat layer of the tree if the parent node of the layer is popped. However, the cascade will cause the root list to be longer than a simple cut. I think it's kind of like a tradeoff between the length of the root node list and the flatness of the tree in the root node list.

```
@Override
public void decreaseKey(K before, K after) {
    assert(before.compareTo(after) > 0);
    assert (!keyToNodeMap.containsKey(after));

    // 1. update the map
    Node theNode = keyToNodeMap.get(before);
    theNode.key = after;
    keyToNodeMap.remove(before);
    keyToNodeMap.put(after, theNode);

    if (theNode.parent != null) {
        if (theNode.key.compareTo(theNode.parent.key) < 0) {
```

```
            Node parentNode = theNode.parent;
            cut(theNode, theNode.parent);
            cascadeCut(parentNode);
        }
    } else {
        // The node is within the root list
        removeFromDoubleLinkedList(theNode);
        insertIntoRootList(theNode);
    }
}

private void cut(Node theNode, Node parentNode) {
    parentNode.degree--;
    if (theNode == parentNode.children) {
        parentNode.children = theNode.after;
    } else {
        theNode.parent = null;
        theNode.before.after = theNode.after;
        if (theNode.after != null) {
            theNode.after.before = theNode.before;
        }
    }
    removeFromDoubleLinkedList(theNode);
    insertIntoRootList(theNode);
    theNode.mark = false;
}

private void cascadeCut(Node theNode) {
    Node parent = theNode.parent;
    if (parent != null) {
        if (!theNode.mark) {
            theNode.mark = true;
        } else {
            cut(theNode, parent);
            cascadeCut(parent);
        }
    }
}
```

3. Theoretical Comparison

Operations	Binary Heap	Fibonacci Heap	Binomial Heap
Make-Heap	O(1)	Θ(1)	O(1)
Insert	Θ(logn)	Θ(1)	O(logn)
Maximum (or Minimum)	Θ(1)	Θ(1)	O(logn)
Delete Max (or Delete Min)	Θ(logn)	O(log n)	Θ(logn)
Merge (Union)	Θ(n)	Θ(1)	O(logn)

4. Theoretical Analysis of the Dijkstra Algorithm

Bounds of the running time of Dijkstra's algorithm on a graph with edges **E** and vertices **V** can be expressed as a function of the number of edges **|E|** and the number of vertices **|V|**. The complexity is mainly determined by the data structure that is used to represent the priority queue of the vertex.

For any data structure, the running time is[1]:

$$\Theta(|E| * T_{\mathrm{dk}} + |V| * T_{\mathrm{pop}})$$

The dk and pop are the complexities of the decrease-key and pop operations of the implementation data structure beneath the priority queue which is already presented in the last section.

5. Test

the download links of these test data sets:

http://www.dis.uniroma1.it/challenge9/download.shtml

We download 4 states: CO ,DC,DE and TX.

state	nodes	edges
CO	448253	539295
DC	9559	14909
DE	49109	60512
TX	2073870	2584159

The format of the uncompressed file is following. It is a whitespace-separated list of numbers:

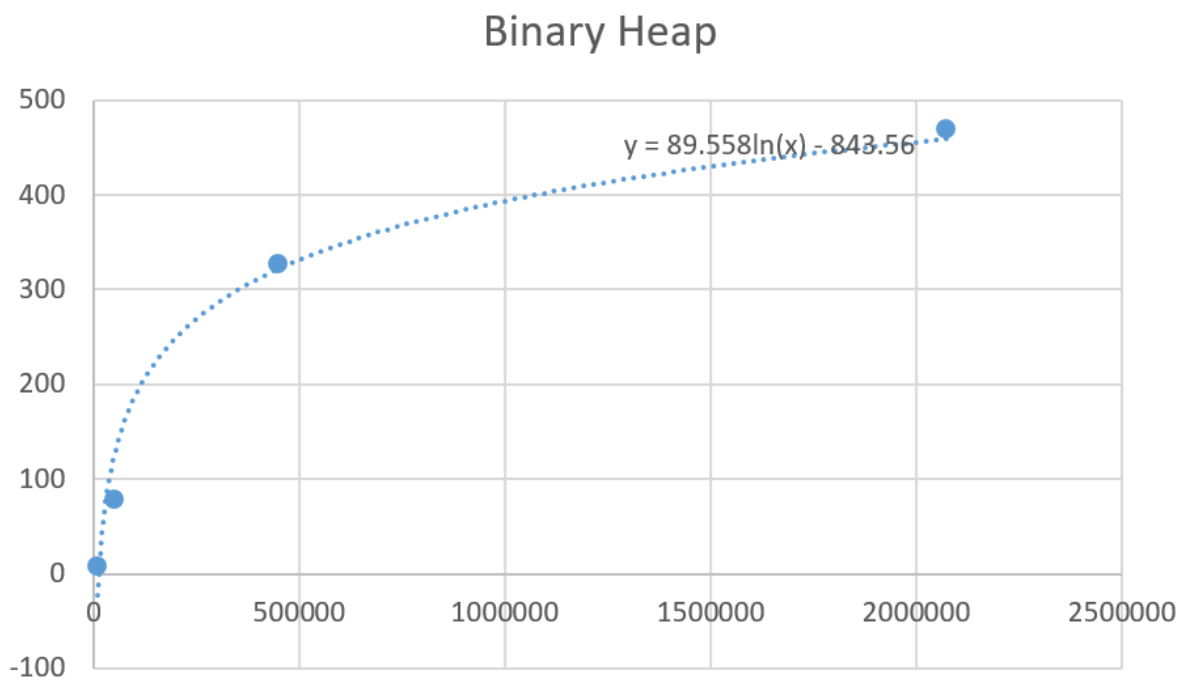
- Number of nodes
- Number of edges
- For each edge:
  id of the source node
  id of the target node
  travel time
  spatial distance in meters

Test strategy: For each graph, N = 10 nodes are selected and the shortest path from that node to all reachable nodes. Repeat 10 times and take the average.

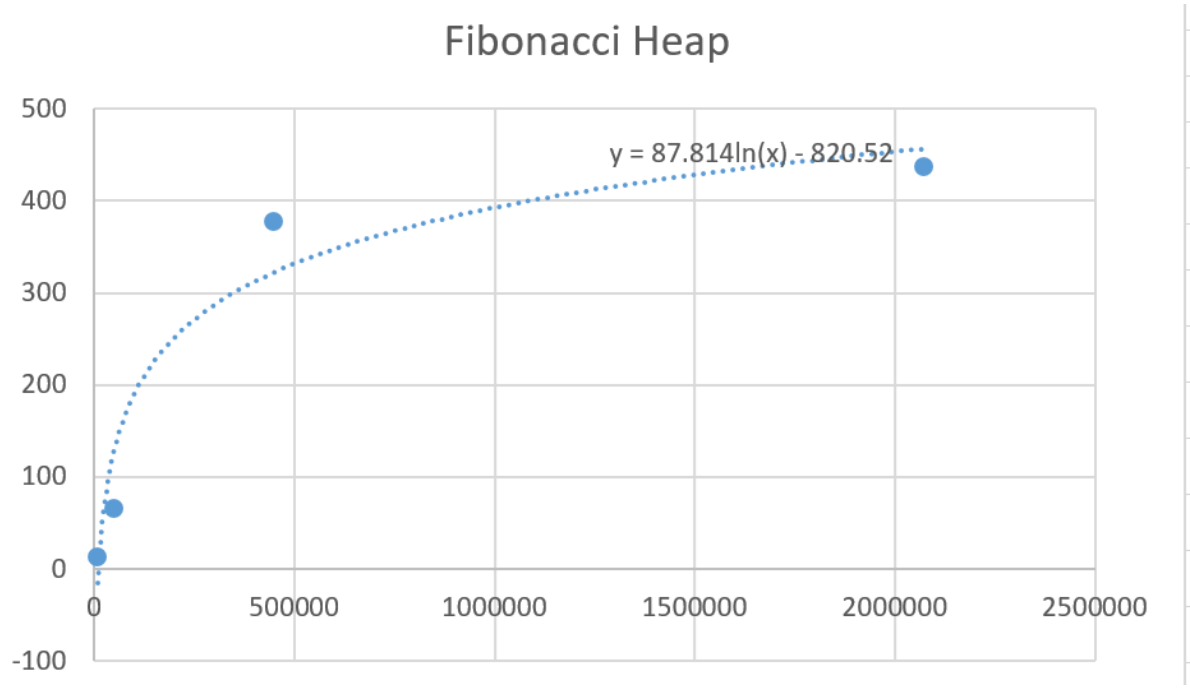
graph	implements	running times
DC	binary Fibonacci binomial	7.4 12.4 3.9
DE	binary Fibonacci binomial	78.2 65.5 47.5
CO	binary Fibonacci binomial	327.5 376.4 166.3
TX	binary Fibonacci binomial	468.9 437 308.9

We can find that the Fibonacci heap doesn't perform better than other heaps in our experiments. The binomial heap has the best performance of the three heaps.

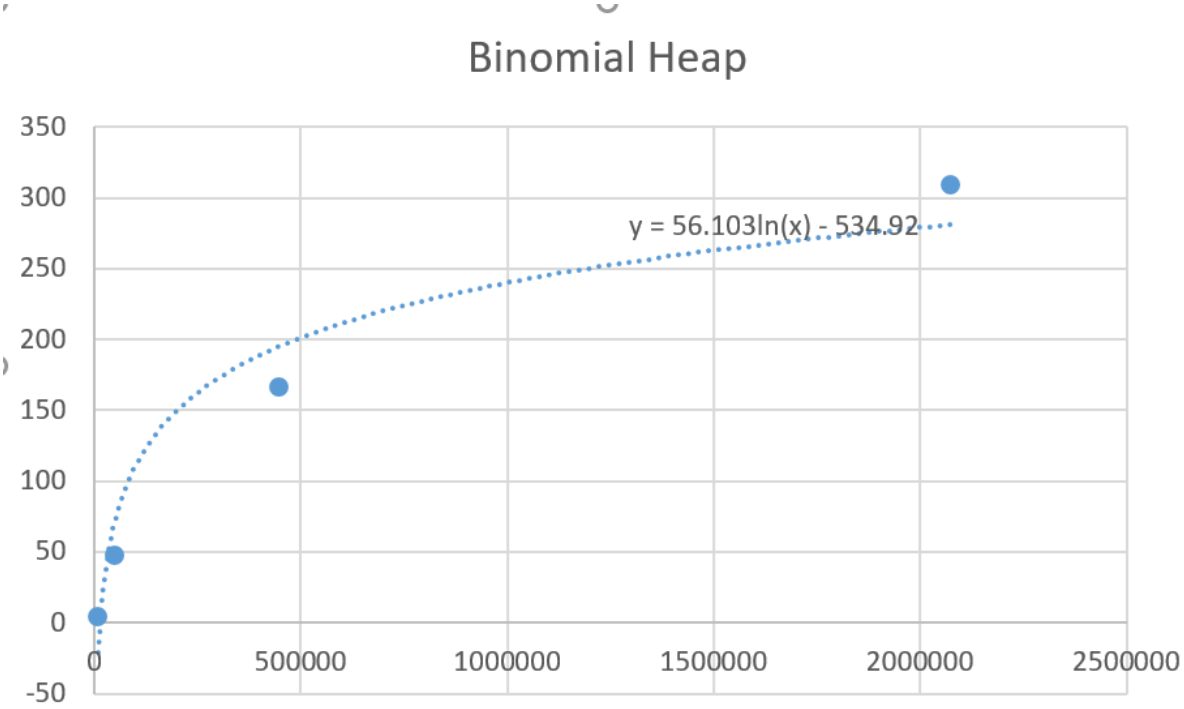
Binary Heap



Fibonacci Heap



Binomial Heap



Actually, other's experiments shows that Fibonacci heaps only outperformed binary heaps when the graph was incredibly large and dense. In 10000 vertices, 20000000 edges(much intense than our datasset) graph, Fibonacci heap has 11% speed up[2]. The constant term of the Fibonacci heap time complexity is very large, and the implementation is too complicated. Thus, it does not perform well in most sparse graphs.

6. Discussion & Conclusion

As the test shows, it seems like the fibonacci heap doesn't outperform the binary heap a lot, sometimes even worse than the binomial heap. But why? It is because of the way we implement the Dijkstra algorithm.

In our implementation, we don't push all the vertex into the PQ at the beginning. Rather, we only push the node into the PQ as its been visited. As a result, the size of the heap is not that big and the improvement by advanced algorithm is not prominent. On the contrary, the complex operation of the doubled linked list will takes more time than some naive approach. However, if the Dijkstra algorithm makes the heap with large scale at the beginning, the fibonacci heap will show its efficiency of amortized log(1) complexity DECREASE-KEY.

```
public static void search(List<Vertex> graph, Vertex source, Heap<Long, Vertex>
pq) {

    for (Vertex vertex : graph) {
            vertex.setVertexBefore(null);
            vertex.setDistanceFromSource(Long.MAX_VALUE);
    }

    Set<Vertex> vertexInHeap = new HashSet<>();

    source.setVertexBefore(null);
    source.setDistanceFromSource(0);
    pq.push(0L, source);
    vertexInHeap.add(source);

    while (!vertexInHeap.isEmpty()) {
        Vertex theVertex = pq.pop();
```

```
        vertexInHeap.remove(theVertex);
        for (Vertex neighbor : theVertex.getNeighbors()) {
            if (neighbor.getVertexBefore() == null) {
                // the vertex that is firstly visited will be added to the pq &
the set
                neighbor.setVertexBefore(theVertex);
                neighbor.setDistanceFromSource(theVertex.getDistanceFromSource()
+ theVertex.getDirectDistance(neighbor));
                vertexInHeap.add(neighbor);
                pq.push(neighbor.getDistanceFromSource(), neighbor);
            } else {
                if (!vertexInHeap.contains(neighbor)) {
                    // if the vertex is already visited and not inside the pq,
the shorted path to it has already been found
                    continue;
                }
                long alt = theVertex.getDistanceFromSource() +
theVertex.getDirectDistance(neighbor);
                if (alt < neighbor.getDistanceFromSource()) {
                    pq.decreaseKey(neighbor.getDistanceFromSource(), alt);
                    neighbor.setDistanceFromSource(alt);
                }
            }
        }
    }
}
```

The result of the project enlighten us that the more advanced approach might not always be the best approach, we should design and pick up the most suitable approach in practice.

7. References

1. Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001). "Section 24.3: Dijkstra's algorithm". Introduction to Algorithms (Second ed.). MIT Press and McGraw–Hill. pp. 595–601. ISBN 0-262-03293-7.
2. https://stackoverflow.com/questions/504823/has-anyone-actually-implemented-a-fibonacci-heap-efficiently