Analysis and Design of Algorithms

Design Document for
Lab Assignment #1

Investigation of the performance of Binary Heaps and Fibonacci Heaps in the context of finding Minimum Cost Spanning Trees

---

Description of Problem

Investigation of the performance of Binary Heaps and Fibonacci Heaps in the context of finding Minimum Cost Spanning Trees

Introduction

Heaps are widely used Data Structures which are used in many areas of Computer Science like Heap Sorting, Prim's Minimum Spanning Tree Algorithm, Dijktra's Shortest Path Algorithm and so on. Various designs of heaps exist which vary in design complexity and performance. Therefore the performance characteristics of a particular heap can greatly affect the running time characteristics of the application which use heaps. Some types of heaps are:

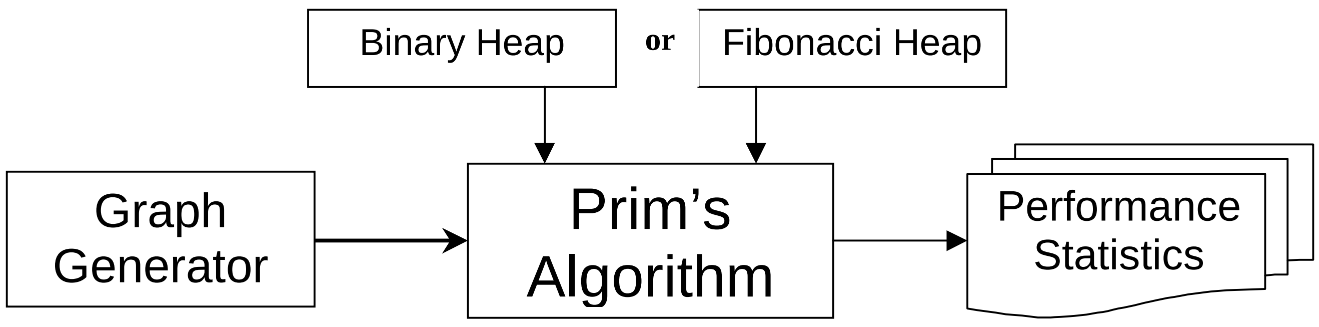
1. Binary Heap
2. Binomal Heap
3. Fibonacci Heap
4. 2-3 Heap

All the above heaps have different running times for operations and therefore their use depends on the particular application.
In this assignment, performance of Binary Heaps and Fibonacci Heaps will be investigated and compared in the context of finding Minimum Cost Spanning Tree of graphs of varying sizes.

Strategy

The Algorithm used for finding the Minimum Cost Spanning Tree will be Prim's Algorithm. Since Prim's Algorithm requires heaps for its implementation, both Binary heaps and Fibonacci Heaps will be used for implementing the Prim's Algorithm. After the algorithms have been implemented, graphs of various sizes will be generated randomly within the system and used for finding Minimum Cost Spanning Tree using both the above heaps and their performance compared as a function of graph size. It will be analysed, which heap performs better in which situations. Constants are expected to play a major role in the above performance analysis.

The various components in this assignment can be identified as:
1. Prim's Algorithm for finding Minimum Cost Spanning Trees
2. Binary Heap
3. Fibonacci Heap
4. Graph Generator



Implementation

The above algorithms will be implemented on JAVA Development Platform. The following sections discuss the implementation of the above components which will aid in their implementation in JAVA.

Binary Heaps

Heaps are primarily used to implement Priority Queues. It is an Abstract Data Type which offers methods that allow removal of the item with the maximum (or minimum) key value , insertion , deletion and other operations like decrease_key and arbitrary_delete.

Various implementations of heaps are possible each having its own design and performance characteristics.

A Binary Heap is a binary tree which has the following characteristics :

1. It is complete. Level I has 2I nodes except the last level which may have less than 2I nodes but there are no gaps between the leftmost leaf and the rightmost leaf.
2. It is usually implemented using arrays. Arrays offer an efficient and convenient implementation of Complete Binary Trees.
3. Each node in the heap , satisfies the heap condition , which states that the node's key is smaller than or equal to the keys of its children.

The operations which will be supported in the Binary Heap are :

1. Insertion
2. DeleteMin
3. ReturnMin
4. DecreaseKey

Representation

The array representation is useful because it is very easy ro get from a node to its fathers and sons. The father of the node in position *j* is in position *j div 2* , and , conversely , the two sons of the node in position *j* are in position *2j* and *2j+1*. This makes traversal of such a tree even easier than if the tree were implemented using a standard linked representation. The minimum key is always the first position of the array.

Time Complexity

All of the algorithms operate along some path from the root to the bottom of the heap. It is easy to see that, in a heap of N nodes , all paths have about log(n) nodes on them. The number of levels in a complete binary tree are log(n). Thus all the operations can be done in logarithmic time using heaps.

Algorithms

The heap algorithms all work by first making a simple structural modification which could violate the heap condition, then travelling through the heap modifying it to ensure that the heap condition is satisfied everywhere.

The nodes in the heap can be defined as follows :

Class : Node

```
// Class for Node
public class Node
{
        public int key;  // The key of the node
        // Also declare other auxiliary data associated with the node

        // Constructor
        public Node(int k)
        {
                key=k;
        }

        // Returns the key of this node
        public int returnkey()
        {
                return key;
        }
}
```

The above class written in JAVA will represent the nodes in the heap. An array can be declared of the above type to represent the heap.

Insertion

The code for insertion is as follows. It uses another function upheap to trickle the new element upwards until the heap property is satisfied.

N represents the current size of the heap.

The ***Insertion*** function can be defined as follows:

```
Insertion(Node node)
{
        N=N+1;
        heap[N]=node;
        upheap(N);
}
```

Function ***upheap*** is defined as:

```
upheap(k)
{
        Node node=heap[k];
        While (heap[k div 2]>=node.returnkey()) do
        Begin
                Heap[k]=heap[k div 2];
                K=k div 2;
        End;
        Heap[k]=node;
}
```

DeleteMin (Delete Minimum)

The ***DeleteMin*** function works by replacing the first node in the array by the last node (thus decreasing the size of the heap by 1 in the process) and then percolating the first node until the heap property is satisfied.

The ***DeleteMin*** function is defined as follows:

```
// returns and deletes the minimum node from the heap
```

```
DeleteMin()
{
    Node r=heap[1];
    heap[1]=heap[N];
    N=N-1;
    downheap(1);
    return r;
}
```

The ***downheap*** function is responsible for restoring the heap property after replacing the first element by the last element. It is defined as follows :

```
Downheap(k)
{
    Node node=heap[k];
    while(k <= N div 2) do
    begin
        j=k*2;
        if (j < N and heap[j].returnkey() > heap[j].returnkey()) then j=j+1;
        if(node.returnkey() <= heap[j].returnkey()) then exit loop;
        heap[j]=heap[k];
        k=j;
    end;
    heap[k]=node;
}
```

ReturnMin (Return Minimum key node)

The ***ReturnMin*** function is trivial to implement as it just returns the first element of the array.
It is defined as follows :

```
ReturnMin()
{
    return heap[1];
}
```

DecreaseKey

DecreaseKey functions by decreasing the key of a particular node and then percolating it upwards until the heap property is satisfied. It is defined as follows :

```
DecreaseKey(index, val)
{
    //
    heap[index].key=heap[index].key-val;
    upheap(index);
}
```

It calls the ***upheap*** function to percolate the concerned node upwards.

Fibonacci Heaps

Fibonacci Heaps is a more sophisticated implementation of heaps. They have some advantages, which greatly reduce their amortised operation cost. In fact all operations where deletion of an element is not involved, they run in O(1) amortised algorithm. However, due to the complicated nature of Fibonacci Heaps, various overheads in maintaining the structure are involved which increase the constant term in the order. Hence in practical terms, it is not very clear when Fibonacci heaps will outperform Binary heaps. The purpose of this investigation is to study and identify when this happens in the context of finding the Minimum Cost Spanning Tree.

The operations, which will be implemented, are:

1. Insertion
2. Union
3. ReturnMin
4. DeleteMin
5. DecreaseKey

Representation

Fibonacci Heap is essentially a collection of heap – ordered trees. We can store the collection by linking the roots with a doubly linked list. Each node has children which are further linked with a doubly linked list. The order of the trees in the root linked list is arbitrary.

Algorithms

The key idea in heap operations on Fibonacci heaps is to delay work as long as possible. Thus, there will be a trade off among various operations in terms of performance.

No attempt to link the trees in any fashion is made during insertion, melding. We simply add the node or tree in the doubly linked list. Thus, these operations result on O (1) time. However, during delete all the trees are combined in such a manner such that for a particular outdegree of the root, only one tree is present. This reduces the number of trees and by further analysis it can be shown that number of trees which result is of O(log n).

The total number of nodes in the heap and a pointer to the minimum node is also maintained.

Structure of Node

The nodes used in the Fibonacci Heaps are defined as follows :

```
class FBNode
{
        public Node node;              // This contains the actual information. For defn. see previous section.
        public FBNode next,prev;       // Pointer to next and previous nodes
        public int outdegree;          // Number of children
        public FBNode child,parent;    //  Pointers to first child and parent
        boolean marked;                // Whether this node is marked. Used in Cascading cuts
}
```

Insertion

Insertion simply involves creation of a new node with the required key and adding it to the root list. No attempt is made to combine the resulting list in any manner thus resulting in a constant time operation

Insertion function is defined as follows:

```
Insert(heap,key)
{
        Create new node and set its key;
        node.degree=0;
        node.parent=null;
        node.child=null;
        node.right=node;
        node.left=node;
        node.marked=false;

        Add the newly created node to the root list , adjusting its left and right pointer appropriately
        Increase the node counter of the heap by 1
        Update the minimum node pointer if new key is the new minimum.
}
```

Union of two heaps

To unite two Fibonacci Heaps , we simply need to combine one root list with the another , update node count and minimum node pointer appropriately. It is a constant time operation.

Union is defined as follows :

```
Union(heap1,heap2)
{
        make new empty heap;
        assign the min node pointer to the minimum of the two min pointers of the two heaps;
        combine the root lists of heap1 and heap2 and assign the resulting heap to heap;
        heap.count=heap1.count + heap2.count;
        return heap;
}
```

Find Minimum Node

It simply returns the minimum node pointer maintained with the heap. It is a constant time operation.

FindMin is defined as follows :

```
FindMin(heap)
{
        return heap.minnode; // Return pointer to the minimum node in the tree
}
```

Delete Minimum Node

The operation of deleting the minimum node is much more complicated than the previously defined operations. The task of combining all the trees in the root list to to reduce their number is also done in the ***DeleteMin*** operation.

Using amortised analysis, the running time of ***DeleteMin*** comes out be O(log n).

DeleteMin function is defined as follows:

```
// Deletes and returns minimum node from the heap. Returns null if tree is empty
DeleteMin(heap)
{
        if (heap.minnode is not null) // Check if heap is not empty
        {
                Add each child of the minimum node to the root list making its parent pointer null;
                remove heap.minnode from the root list of heap;
                Combine(heap);
                heap.count = heap.count –1;
        }

        return heap.minnode;
}
```

The ***DeleteMin*** operation calls the ***Combine*** operation, which combines the root list such that for a particular value of outdegree of the root, only one tree is present.

This is done by making an array of trees with the outdegree of the tree as its index. Obviously the array can store only one tree with a particular outdegree.
Now all the trees in the root list are fetched and added to this array according to their outdegree. If another tree is already present in the location , the two trees of equal outdegree are linked such that heap property is maintained resulting in a tree with outdegree increased by unity. This tree is again added to the next index location and the above process repeated until an empty index is found for the new tree resulting at each step.

The maximum resulting outdegree by following the above process can shown to be O(log n).

The ***Combine*** function is defined as:

```
Combine(heap)
{
        // maxd denotes the maximum resulting outdegree
        // array denotes the array used in combining the nodes in the root list
        for every node v in the root list of heap
        {
                d = v.outdegree;
                while ( array[d] is not empty)
                {
                        w=array[d];
                        link v and w maintaining the heap property and assign to v
                        array[d] = empty;
                        d = d + 1;
                }
                array [d] =  v;
        }
        //___Now rebuild the root list and update the min pointer in the combined list
        heap.rootlist  = empty;
        for all elements of the array
        {
                if( current element not empty)
                {
                        add to root list
                        if added node is minimum node till now then update heap.minnode pointer;
                }
        }
}
```

The ***Combine*** operation needs to link two nodes. The ***link*** function can be defined as follows :

```
link(node v1 , node v2 )
{
        make v2 a child of v1 and increment outdegree of v1 by unity
        unmark v2;
        return v1;
}
```

Decrease Key operation

In Fibonacci Heaps, the ***DecreaseKey*** operation is performed by simply cutting the node from its parent if the heap property is violated by the decreased node. However if cut trees arbitrarily, the exponential relation between the outdegree and the number of nodes may no longer be maintained. To avoid this, we follow a marking scheme, which ensures the exponential relationship. The marking scheme is shown in the algorithms below.

Using amortised analysis, the running time of ***DecreaseKey*** operation comes out to be O(1).

The ***DecreaseKey*** function can be defined as follows :

```
DecreaseKey(heap,node,amount)
{
        decrease node.key by amount;
        if node.parent is not null and node.parent.key > node.key
        {
                Cut node from node.parent;
                Perform Cascading – cut operation on node.parent;
        }
        Update heap.minnode pointer if decreased node in new minimum node;
}
```

The ***DecreaseKey*** operation performs two other operations namely :

1. Cut
2. Cascading Cut

The ***Cut*** function is defined as follows:

```
// Cuts v from its parent
Cut(heap , v)
{
        remove v from its parent’s child list;
        decrease outdegree of v.parent by unity;
        v.parent=null;
```

```
    v.marked=false;
}
```

The ***CascadingCut*** function is defined as follows:

```
CascadingCut(heap, v)
{
    if (v.parent is not null)
    {
        if(v.marked=false)
        {
            v.marked=true;
        }
        else
        {
            Cut(heap,v);
            CascadingCut(heap,v.parent)
        }
    }
}
```

Prim’s Algorithm

A ***minimum spanning tree*** of a weighted graph is a collection of edges that connects all the vertices such that the sum of the weights of the edges is at least as small as the sum of weights of any other collection of edges that connects all the vertices. The minimum spanning tree for a particular graph may not be unique.

The basic method to finding a ***Minimum Spanning Tree*** is based on a greed approach. From a particular vertex , the next vertex is so chosen so that it can be connected to the current tree using the edge of the lowest weight. Repeating this process until all the nodes are included yields the ***Minimum Spanning Tree***.

The ***Prim’s Algorithm*** is based on the above approach. It uses a Heap for finding the next vertex with the minimum edge weight which can be included in the Minimum spanning tree.

Prim’s Algorithm can be defined as :

```
Prim(graph,root)
{
        Create empty heap; // heap will contain all vertices not included in MST
        Add all the vertices of graph to heap with keys set as infinity;
        Set the key of root to zero;
        Set the parent of root to null;

        while heap is nonempty
        {
                extract vertex with minimum key and assign it to v;
                add v as a child of parent of v;
                for all vertices w adjacent to v
                {
                        if w belongs to heap and cost(w,v) < key(w)
                        {
                                set  parent of w equal to v;
                                set key(w) equal to cost(w,v);
                        }
                }
        }
}
```

The above function also calls :
1. cost(u,v) : This function returns the cost of the edge between vertex u and vertex v
2. key(v) : This function returns the key value associated with the vertex v in the heap

The performance of the Prim’s Algorithm depends on the performance of the heap used in the above algorithm. This is is the reason this application is chosen to compare the two types of heaps : Binary Heap and Fibonacci Heap.

We can run Prim’s Algorithm for various graphs using both types of heaps and compare their performance. By running this algorithm on graphs of varying complexity , we can identify under what conditions do Fibonacci Heaps perform better than Binary Heaps and how much improvement is there. Are Fibonacci Heaps practical to use in real applications , and if the answer is yes , then under what conditions. Such questions of practical importance can be answered by performing the above investigation.

Random Graph Generation

The graphs which will be given as input to the Prim's Algorithm will be generated randomly within the system itself.

Let us define the ***random*** function as follows :
random(n) returns a random integer varying from 1 to n.

For Random Graph Generation some parameters will be needed which will indicate the complexity, size etc. of the graph.

Parameters:
1. Number of vertices in the graph
2. Density of the graph (can be specified in terms of number of edges required)

Using the above parameters edges can be chosen at random between any two vertices and added to the graph structure.

One way is to add equal number of edges for each vertex. If total number of edges required are E and number of vertices are N , then edges per vertex will be E/N. In case E = N *N we will get a complete graph.

```
// N is the number of vertices and E is the number if edges
GenerateRandomGraph(N,E)
{
        // M is the number of edges required per vertex
        M = E / N;
        Create Empty Adjacency List for graph and assign it to G;

        int array[N];
        for i =  1 to N  do array[i]=i;

        for each vertex in G
        {
                //  Randomly arrange the array to get a random set of vertices
                for i= 1 to N
                {
                        r = random ( N );
                        swap array[i] with array[r];
                }
                Add first M elements of array to the edge list of current vertex in the adj. list;
        }
        return G;
}
```

The above algorithm can also be modified to add random no of edges to each vertex with some specified minimum number of edges for each vertex. One simply needs to change the line shown in **bold** to add the required number of edges for each vertex.

Using the above function , we can generate graphs of varying complexity say with N = 10 , 100 , 500, 1000 and trying out different values of E (for sparse and dense graphs). We can run the Prim's Algorithm for each of these graphs using Binary Heaps as well as Fibonacci Heaps and then compare their performance. The running times can be plotted vs graph complexity and one can figure out which heap is better in which situation.