Detailed implementation and Performance Comparison between Array and Fibonacci Heap in implementing Prim's Algorithm

Pengjie Zhang
University of Florida
Department of Computer & Information Science & Engineering
Gainesville, FL 32611, USA
pengjie@cise.ufl.edu

October 25, 2013

Contents

1 File Structure and Method to Compile 1

2 Sytem Flow Chart 1

3 Class Structure and Definitions 2
- 3.1 Fibonacci heap . . . 2
- 3.2 Graph Representation . . . 5
- 3.3 All Other Classes . . . 6

4 Expected Performance 10

5 Experimental Results 10

1 File Structure and Method to Compile

All my source files are in the directory "Zhang", the structure of my source files is as follows:

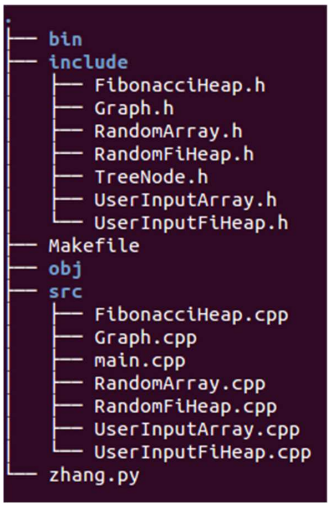

Figure 1: File Structure

My program only have two main directories: "include" and "src". "include" directory contains all the .h header files and "src" contains all the .cpp source files. In order to executing my program, you should first type "make" in shell, then there will be a executable file named "mst" come into being in the "bin" directory (directory "obj" will store all object files). In face with having "mst" generated, you have several choices to run "mst":

1. "./bin/mst -r n d"
2. "./bin/mst -f file-name"
3. "./bin/mst -s file-name"
4. "python zhang.py"

I have run all these commands in our "thunder.cise.ufl.edu" server and get correct responses, so you should execute them without any difficulty (there is a zhang.log in my directory, that's the result by typing "python zhang.py >zhang.log").

2 Sytem Flow Chart

This program will handle two cases, each case has two subcases:

1. random mode:
   (a) simple scheme (use array as the underlying data structure, denoted as "RandomArray")
   (b) fibonacci heap scheme (use Fibonacci heap as the underlying data structure, denoted as "RandomFiHeap")
2. user input mode:

(a) simple scheme (use array as the underlying data structure, denoted as "InputArray")

(b) fibonacci heap scheme (use Fibonacci heap as the underlying data structure, denoted as "Input-FiHeap")

The flow chart of this program is as the following figure:

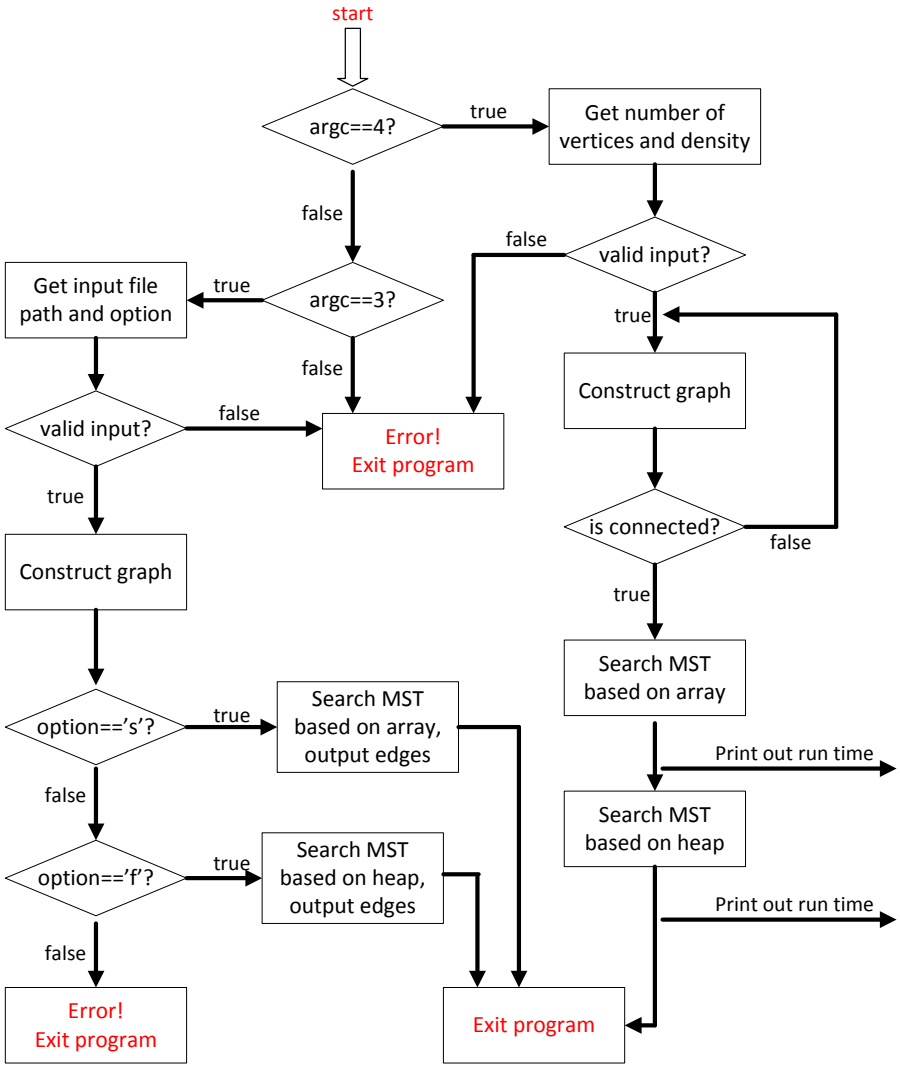

Figure 2: flow chart

3 Class Structure and Definitions

In this program, I devise several classes, each class will achieve a specific task.

3.1 Fibonacci heap

For Fibonacci heap, there are two classes, one is "class TreeNode". "TreeNode" has all the data fields that taught in class plus two extra fields that help utilize them in Prim’s algorithm. Remember, each "TreeNode" is related with a "vertex" in "Minimum Spanning Tree":

1. "int data;" -> record the minimum length from known vertices to this vertex;
2. "int identity;" -> record the index of this vertex (all vertices are numbered from "0" to "n-1");
3. "int neighbor;" -> the index of one of the known vertices who has the minimum length to this vertex;

```
#ifndef _TREE_NODE_H_
#define _TREE_NODE_H_

#include <stdlib.h>

//for this class, there is no need to provide another "TreeNode.cpp" file
// since all functions are defined inside class
class TreeNode{
                friend class FibonacciHeap;
        public:
                TreeNode(){
                        identity = -1;
                        neighbor = -1;
                        degree = 0;
                        child = NULL;
                        leftSibling = rightSibling = this;
                        parent = NULL;
                        childCut = false;
                }
                //constructor
                TreeNode(int _data, int _identity):data(_data), identity(_identity){
                        degree = 0;
                        child = NULL;
                        leftSibling = rightSibling = this;
                        parent = NULL;
                        childCut = false;
                }
                //overloaded constructor
                TreeNode(int _data, int _identity, int _neighbor):data(_data), identity(
                    _identity), neighbor(_neighbor){
                        degree = 0;
                        child = NULL;
                        leftSibling = rightSibling = this;
                        parent = NULL;
                        childCut = false;
                }
        public:
                ///getter for "data".
                int getData(){
                        return data;
                }
                ///setter for "data".
                void setData(int _newData){
                        data = _newData;
                }
                int getIdentity(){
                        return identity;
                }
                void setIdentity(int _identity){
                        identity = _identity;
                }
                int getNeighbor(){
                        return neighbor;
```

```
                }
                void setNeighbor(int _neighbor){
                        neighbor = _neighbor;
                }

        private:
                //these two member variables are not must for Fibonacci heap
                //but in order to use Fibonacci heap as the underlying data structure
                //for Prim's algorithm, we need these two variables
                int identity; ///the index of this vertex
                int neighbor; ///the index of the vertex that connnect this vertex in MST

        private:
                int degree;
                TreeNode* child;
                TreeNode* leftSibling;
                TreeNode* rightSibling;
                TreeNode* parent;
                bool childCut;
                int data;
};

#endif
```

And our "class FibonacciHeap" utilize the "class TreeNode", provides all necessary functions:

1. insert. (only insert a single node)
2. meld. (maybe a single node or another Fibonacci heap)
3. getMin.
4. removeMin. (only the root)
5. remove. (maybe the root or any other node)
6. decreaseKey.
7. pairwise combine.

```
#ifndef _FIBONACCI_HEAP_H_
#define _FIBONACCI_HEAP_H_

#include "TreeNode.h"

//as we know, fibonacci heap is similar with fibonacci heap, the action fibonacci provide is
//(1) Insert (either insert a single node or a min tree)
//(2) RemoveMin
//(3) Meld (meld with another fibonacci heap, since a fibonacci heap is a collection
//    of min trees, so a fibonacci heap can be a single min tree or, even a single node,
//    so "Meld" is much similar with "Insert")
//(4) Remove (this function can also perform "removeMin")
//(5) DecreaseKey
// For our implementation, we would provide "pairwiseCombine"

class FibonacciHeap{
        public:
                //default constructor
```

```
19                 FibonacciHeap(){
20                         minimumPointer = NULL;
21                 }
22                 //another overloaded constructor
23                 FibonacciHeap(TreeNode* _minimumPointer):minimumPointer(_minimumPointer){}
24
25         public:
26                 //anotherTree could be only one single node tree, or be a "min tree",
27                 ///after insert, we will call "pariwiseCombine(...)"
28                 void insert(TreeNode* anotherTree);
29                 ///anotherHeap is another fibonacci heap, we will change "anotherHeap", so
30                 ///we use reference. after meld, we will call "pairwiseCombine(...)"
31                 void meld(FibonacciHeap& anotherHeap);
32                 ///this function is added by myself to utilize fibonacci heap
33                 TreeNode* getMin();
34                 ///we must call "pairwiseCombine(...)" inside this "removeMin(...)"
35                 ///in order to achieve O(logn) amortized time complexity
36                 bool removeMin();
37                 ///this arbitrary remove accept an argument which is pointing to a existed
38                 /// node in the heap
39                 bool remove(TreeNode*& nodeInTheHeap);
40                 ///"nodeInTheHeap" is a pointer which points to a current node in the heap,
41                 ///"_newKey" is the key we are going to assign to it
42                 bool decreaseKey(TreeNode* nodeInTheHeap, int _newKey);
43
44         private:
45                 ///define pairwiseCombine as private function since this function only
46                 ///used by other functions users shouldn't know this function is working.
47                 ///This function would modify "minimumPointer" and "topLevelDegree"
48                 /// in order to do pairwise combine, we need a "tree table" just as the
49                 ///slides indicate, we will use vector<TreeNode*> table as the table
50                 void pairwiseCombine(TreeNode *temporayPointer);
51         private:
52                 TreeNode* minimumPointer;
53 };
54
55 #endif
```

3.2 Graph Representation

Just as the project specification describes, we need to use "adjacency lists" to represent the *undirected graph* (if use "adjacency matrix", the time to search the matrix will dominate the time used by Fibonacci heap, thus give us illusion that Fibonacci is not as good as expected). My "class Graph" is acted as a "undirected graph" which has some functions:

```
1  #ifndef _GRAPH_H_
2  #define _GRAPH_H_
3
4  #include <vector>
5
6  class Graph{
7          public :
8                  ///default constructor
9                  Graph(){
10                         numberOfVertices = 0;
11                         numberOfEdges = 0;
12                         connected = false;
```

```
                }
                ///copy constructor
                Graph(const Graph &g){
                        numberOfVertices = g.numberOfVertices;
                        numberOfEdges = g.numberOfEdges;
                        connected = g.connected;
                        udGraph = g.udGraph;
                }
                ///destructor
                ~Graph(){}

        public:
                ///whether the undirected graph is connected or not
                bool isConnected();
                ///use dfs to check whether the undirected graph is connected or not
                bool checkConnectedness();
                ///build a undirected graph which has specified number of vertices and edges
                bool buildRandomGraph(int vertices, int edges);
                ///build a undirected graph from a formatted file
                bool buildGraphFromFile(char *filePath);
                ///return neighbours of a vertex whose index is "index"
                std::vector<std::pair<int,int> > getNeighbors(int index);
        public:
                ///getter
                int getNumOfVertices();
                ///setter
                void setNumOfVertices(int _vertices);
                ///getter
                int getNumOfEdges();
                ///setter
                void setNumOfEdges(int _edges);

        private:
                ///used by checkConnectedness() to decide whether is connected or not
                void dfs(int index, std::vector<bool> &con, int &count);

        private:
                //the underlying graph representation
                std::vector<std::vector<std::pair<int, int> > > udGraph;
                bool connected;
                int numberOfVertices;
                int numberOfEdges;
};

#endif
```

3.3 All Other Classes

There are another four classes which are acting as "wraper":

1. class RandomFiHeap

```
#ifndef _RANDOM_FI_HEAP_H_
#define _RANDOM_FI_HEAP_H_
#include "FibonacciHeap.h"
#include "Graph.h"

```

```
class RandomFiHeap{
        public:
                RandomFiHeap(){}
                ~RandomFiHeap(){}
        public:
                ///return the total length of the minimum spanning tree
                int searchMst(Graph &g);
};

#endif
```

2. class RandomArray

```
#ifndef _RANDOM_ARRAY_H_
#define _RANDOM_ARRAY_H_
#include "Graph.h"

class RandomArray{
        public:
                RandomArray(){}
                ~RandomArray(){}
        public:
                ///return the total length of the minimum spanning tree
                int searchMst(Graph &g);
};
#endif
```

3. class UserInputFiHeap

```
#ifndef _USER_INPUT_FI_HEAP_H_
#define _USER_INPUT_FI_HEAP_H_
#include "Graph.h"
#include "FibonacciHeap.h"

class UserInputFiHeap{
        public:
                UserInputFiHeap(){}
                ~UserInputFiHeap(){}
        public:
                ///return the total length of the minimum spanning tree
                int searchMst(Graph &g);
};

#endif
```

4. class UserInputArray

```
#ifndef _USER_INPUT_ARRAY_H_
#define _USER_INPUT_ARRAY_H_
#include "Graph.h"

class UserInputArray{
        public:
                UserInputArray(){}
                ~UserInputArray(){}
        public:
```

```
10              ///return the total length of the minimum spanning tree
11              int searchMst(Graph &g);
12 };
13 #endif
```

You can find that all above classes only have one useful function, and the signature is the same. I will show the detailes about how they works:

1. class RandomFiHeap

```
1  #include <vector>
2  #include <queue>
3  #include "RandomFiHeap.h"
4  using namespace std;
5
6  int RandomFiHeap::searchMst(Graph &g)
7  {
8          int vertices = g.getNumOfVertices();
9          int edges = g.getNumOfEdges();
10
11         vector<bool> visited(vertices, false);
12         vector<TreeNode*> help(vertices);
13
14         FibonacciHeap fheap;
15         vector<pair<int, int> > neighbors = g.getNeighbors(0);
16         for(int i = 0; i < neighbors.size(); i++)
17         {
18                 int identity = neighbors[i].first;
19                 int data = neighbors[i].second;
20                 TreeNode *tmp = new TreeNode(data, identity);
21                 help[identity] = tmp;
22                 fheap.insert(tmp);
23         }
24
25         visited[0] = true;
26         int count = 1;
27         int totalLength = 0;
28         while(count < vertices)
29         {
30                 TreeNode *mini = fheap.getMin();
31                 int length = mini->getData();
32                 int identity = mini->getIdentity();
33                 totalLength += length;
34                 visited[identity] = true;
35                 count++;
36
37                 fheap.removeMin();
38
39                 neighbors = g.getNeighbors(identity);
40                 for(int i = 0; i < neighbors.size(); i++)
41                 {
42                         int v = neighbors[i].first;
43                         int cost = neighbors[i].second;
44                         if(visited[v] == false)
45                         {
46                                 if(help[v] == NULL)
47                                 {
48                                         TreeNode *tmp = new TreeNode(cost, v);
```

```
                                        help[v] = tmp;
                                        fheap.insert(tmp);
                                }
                                else if(help[v]->getData() > cost)
                                        fheap.decreaseKey(help[v], cost);
                        }
                }
        }
        return totalLength;
}
```

2. class RandomArray

```
#include <vector>
#include "RandomArray.h"
#include <limits.h>
using namespace std;

int RandomArray::searchMst(Graph &g)
{
        int vertices = g.getNumOfVertices();
        int edges = g.getNumOfEdges();
        vector<bool> visited(vertices, false);
        vector<int> distance(vertices, INT_MAX);

        vector<pair<int, int> > neighbors = g.getNeighbors(0);
        for(int i = 0; i < neighbors.size(); i++)
        {
                int v = neighbors[i].first;
                int cost = neighbors[i].second;
                distance[v] = cost;
        }

        visited[0] = true;
        int count = 1;
        int totalLength = 0;

        while(count < vertices)
        {
                int minimumLength = INT_MAX;
                int index = -1;

                for(int i = 0; i < vertices; i++)
                {
                        if(visited[i] == false && distance[i] < minimumLength)
                        {
                                minimumLength = distance[i];
                                index = i;
                        }
                }

                count++;
                totalLength += minimumLength;

                neighbors = g.getNeighbors(index);
                for(int i = 0; i < neighbors.size(); i++)
                {
                        int v = neighbors[i].first;
```

```
46                          int cost = neighbors[i].second;
47                          if(visited[v] == false && distance[v] > cost)
48                                  distance[v] = cost;
49                  }
50          }
51
52          return totalLength;
53  }
```

4 Expected Performance

Though fibonacci heap's actual cost is high, its amortized cost will be much better if we execute cascading cut and pairwise combine, the resulted time complexity for prim's algorithm could be $O(nlogn + e)$. On the other hand, simple scheme can use $O(n^2)$ time complexity. When the graph is dense, they will have the same performance, but when the graph is sparse, then fibonacci heap should work much better than an array.

	actual	amortized
Insert	O(1)	O(1)
Remove min (or max)	O(n)	O(log n)
Meld	O(1)	O(1)
Remove	O(n)	O(log n)
Decrease key (or increase)	O(n)	O(1)

Fibonacci heap

- Array
  O(n^2) overall complexity
- Min heap
  O(nlogn + elogn) overall complexity
- Fibonacci heap
  O(nlogn+e) overall complexity

5 Experimental Results

I measure my program's performance only in random mode, generate 10 connected undirected graphs with different edge densities (10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, 90%, 100%) for each of the cases n = 1000, 3000, and 5000. In order to get reliable result, I run my program 5 times for each case specified and average them by dividing the number of runs. The follwing figures show the result:

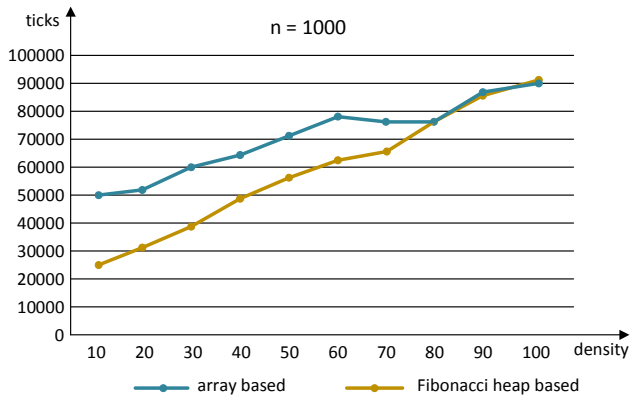

Figure 3: when n equals to 1000

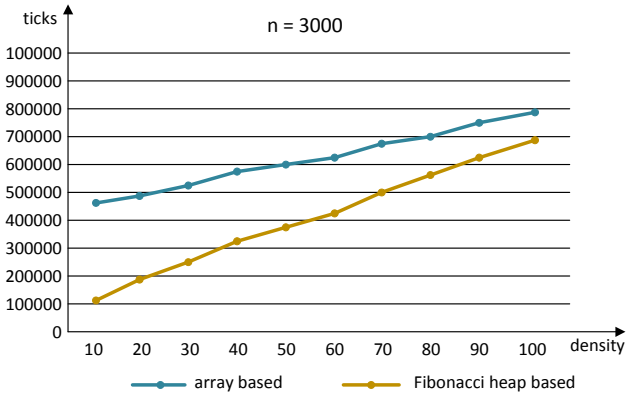

Figure 4: when n equals to 3000

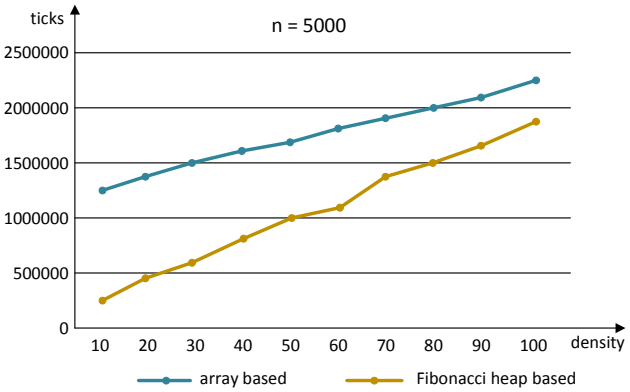

Figure 5: when n equals to 5000

Just as you have seen, generally the fi heap scheme is much better than simple scheme. When the graph is sparse, the fi heap scheme is much better than simple scheme; when the graph is dense, fi heap scheme is again better than simple scheme even though the performance is a littler closer.