Advanced Data Structures Project
COP5536 FALL 2013

Implementation and Performance comparison of
Prim's algorithm using an array and a Fibonacci heap

Name : Kumar Sadhu
UFID : 2949-3662
Email ID: kumarsadhu@ufl.edu

<u>Contents:</u>

<u>Objective:</u>

The objective of the project is to implement Prim's minimum spanning tree algorithm using array data structure and a Fibonacci heap data structure and compare the performance between the two schemes.

The program should support two modes.
1.<u>User mode</u>
In the user mode an input file is read from disk and a graph is generated with edges and weights present in the file.
The way to run with different schemes is as follows

```
$mst -s <input>
```

for simple scheme, i.e using an array and,

```
$mst -f <input>
```

for Fibonacci scheme, i.e using a Fibonacci scheme.
The data in input file is as follows.
First line: n m
where n is number of vertices and m is number of edges in the graph
Next m lines are as follows
i j w
Where an edge (i,j) in the graph has weight w.

2.<u>Random mode</u>
In this mode, two parameters are given as input to the program as follows

```
$mst -r n d
```

where -r implies it is to be run in random mode and n represents number of vertices in the graph and d represents percentage of density of edges in the graph.

<u>Output</u>:
For each mode, output minimum spanning tree cost and the set of edges that comprise of minimum spanning tree.

Implementation: Associated File and Structures

I have chosen to implement the project in c++ with a mix of classic c structures. The implementation for the project is done in the file **mst.cpp** which is a single file comprising of multiple data structures to facilitate the successful execution of the program. Here is a list of the associated data structures and function prototypes used for completing this project.

Heap structures and functions:

```
typedef struct f_heap_node{
	int degree;	// Number of children of this node
	bool c_cut; // Whether this node is marked
	struct f_heap_node* next;   // Next and previous elements in the list
	struct f_heap_node* prev;
	struct f_heap_node* parent; // Parent of this node, if any.
	struct f_heap_node* child;  // Child node, if any.
	int vertex;	// Element being stored here
	int weight; // Its priority
}fnode;

typedef struct f_heap{
	fnode* minNode;
	int nodes;
}fheap;

fheap* makeheap(fheap* &root);
fnode* mergeroots(fnode* &n1,fnode* &n2);
fnode* finsert(fheap* &h,fnode* &node,int vertex,int weight);
fheap* mergeheaps(fheap* h1,fheap* h2);

void cut(fheap* &h ,fnode* &chld ,fnode* &prnt );
void cascade(fheap* &h,fnode* &y);

void decrease_weight(fheap* &h,fnode* &n,int amount);
void consolidate(fheap* &h);
```

```
fnode* removeMin(fheap* &h);
```

<u>Graph structures and Functions:</u>

```
struct AdjListNode{
	int dest;// The destination vertex j in the edge(i,j)
	int weight;// weight of edge (i,j)
	struct AdjListNode* next;// pointer to next node
};
struct AdjList{
	struct AdjListNode *head;  // pointer to head node of list
};
struct Graph{
	int V; // total vertices in the graph
	struct AdjList* array; // array of adjacency lists
};

struct AdjListNode* newAdjListNode(int dest);
struct Graph* createGraph(int V);
void addEdge(struct Graph* graph, int src, int dest, int weight);
struct Graph* generateRandomGraph(int vertices, int density);
```

<u>DFS functions:</u>

```
void dfs(struct Graph* g, int v, bool marked[]);
bool checkConnected(struct Graph* g);
```

<u>Prim's implementations:</u>

clock_t **prims_fscheme**(struct Graph* g);
clock_t **prims_simple**(struct Graph* g);

<u>Compiler Description:</u>

I have compiled the program using To run the program with gcc version 4.6.3

(Ubuntu/Linaro 4.6.3-1ubuntu5)
To compile and run the program follow the steps mentioned below.

Compile: Enter the following command from the directory where mst.cpp is present on a Unix machine.

```
Shell]$ g++ mst.cpp
```

This generates a file called **a.out** in the current directory which is the executable program to run the program.

Run: Enter the following in the shell prompt to run the program
If you want to run in **user input mode**, if input.txt is the file which contains input graph data, place it in current directory. Then run the following command.
If you input file has spaces in the name escape them with a '\'. So “inp 1” would be given as “inp\ 1”.

For simple scheme
```
Shell]$ ./a.out -s input.txt
```
For Fibonacci heap scheme
```
Shell]$ ./a.out -f input.txt
```

To run in **random mode**, issue the following command
```
Shell]$ ./a.out -r <number of vertices> <density percentage>
```

For both the modes, in the output you will see the minimum spanning tree cost and the edges which are included in the minimum spanning tree.

Program Structure and Overview:

Description:
The main method reads in the input arguments from command line and based on the

input parameter mode, appropriate modules are called for execution in that mode.

If its a **<u>user mode</u>**,

The file is read line by line and a graph is constructed with
```
struct Graph* createGraph(int V);
```
function call which returns the graph.
Then line by line each edge is added to the graph with the function
```
void addEdge(struct Graph* graph, int src, int dest, int weight);
```

Once the graph is constructed depending on the user input, a check is made to make sure if its a connected graph. If its not, a message is displayed. If its a valid graph, one of the following are the two methods are called. They compute the minimum spanning tree of the graph g and return the time elapsed for the computation.
The cost of the tree and the edges are stored in a list and are then output at the end of the function.

clock_t **prims_fscheme**(struct Graph* g);
clock_t **prims_simple**(struct Graph* g);

If its a **<u>random mode,</u>**
The vertices and density percentage parameters are read and passed to the following function to generate a random graph.

struct Graph* generateRandomGraph(int vertices, int density);

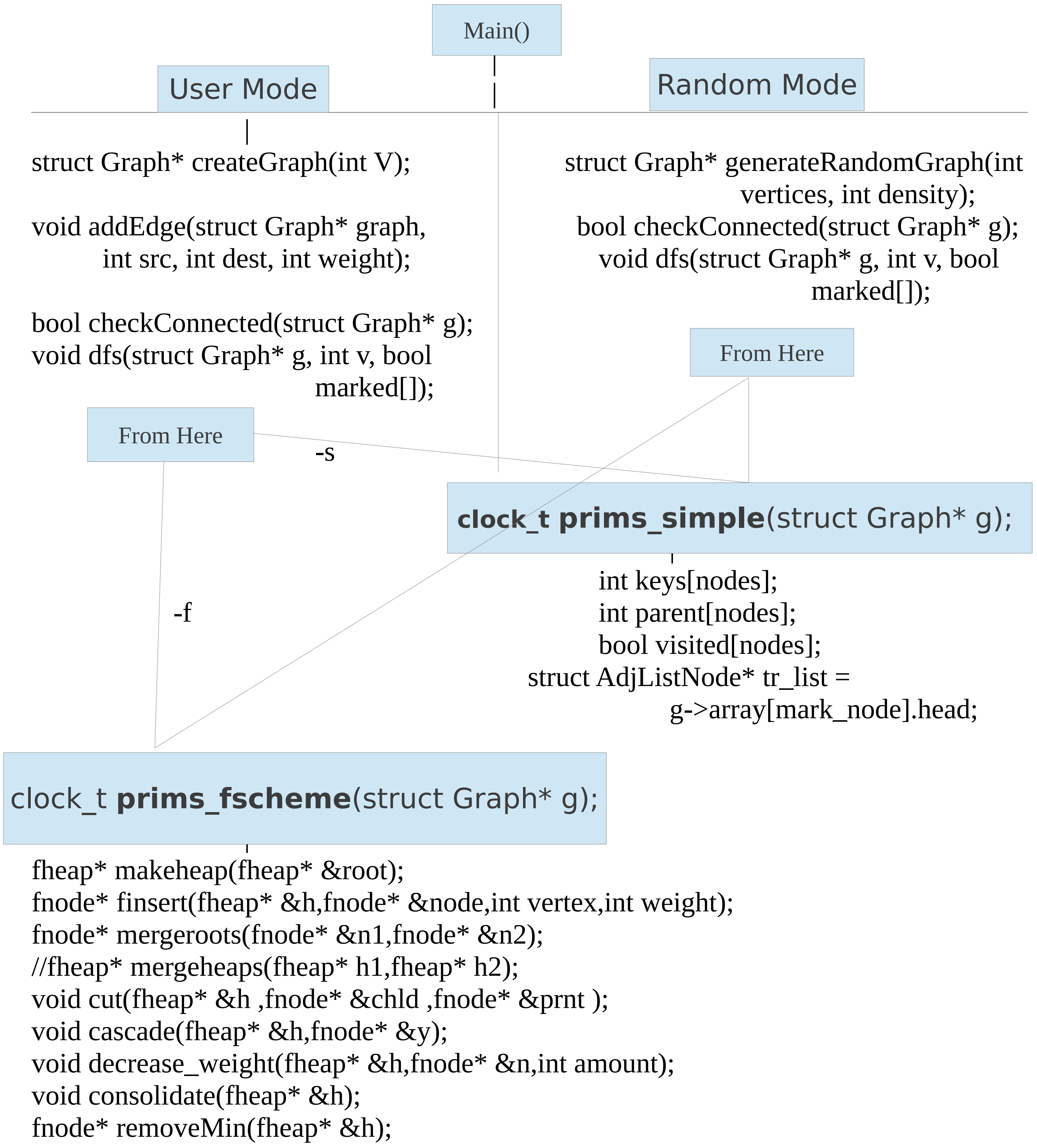
Once a graph is generated, it is checked for connectivity with the following method.

bool checkConnected(struct Graph* g);

which in turn calls the dfs function
void dfs(struct Graph* g, int v, bool marked[]);

A random graph is generated until it is a connected graph.
Once a connected graph is generated then both the schemes are called, then prims algorithm is called with both the scheme methods highlighted above.

Program structure:


Main()
User Mode
Random Mode
From Here
From Here
-s
-f
clock_t prims_simple(struct Graph* g);
clock_t prims_fscheme(struct Graph* g);

```
struct Graph* createGraph(int V);

void addEdge(struct Graph* graph,
        int src, int dest, int weight);

bool checkConnected(struct Graph* g);
void dfs(struct Graph* g, int v, bool
                          marked[]);
```

```
struct Graph* generateRandomGraph(int
              vertices, int density);
bool checkConnected(struct Graph* g);
void dfs(struct Graph* g, int v, bool
                       marked[]);
```

```
int keys[nodes];
int parent[nodes];
bool visited[nodes];
struct AdjListNode* tr_list =
            g->array[mark_node].head;
```

```
fheap* makeheap(fheap* &root);
fnode* finsert(fheap* &h,fnode* &node,int vertex,int weight);
fnode* mergeroots(fnode* &n1,fnode* &n2);
//fheap* mergeheaps(fheap* h1,fheap* h2);
void cut(fheap* &h ,fnode* &chld ,fnode* &prnt );
void cascade(fheap* &h,fnode* &y);
void decrease_weight(fheap* &h,fnode* &n,int amount);
void consolidate(fheap* &h);
fnode* removeMin(fheap* &h);
```

Performance Analysis:

The program has been run multiple time with various input parameters in random mode to observe the performance of prim's algorithm implementation in both the schemes.

Expectation: The asymptotic analysis of Prim's algorithm with both the data structures is as follows.
Array: O(n2)
Fibonacci heap: O(nlogn+e)

As the removeMin and decreaseKey operations of fibonacci heap are having amortized cost of O(logn) and O(1) respectively, it is initially expected that fibonacci heap must outperform the simple array implementation because extractMin and update are linear in array scheme. As we can see for a sparse graph, number of edges are in linear order so, run time with Fibonacci heap would be O(nlogn). For array scheme it would be O(n2). For a dense graph when e is of the order of O(n2), the cost of both would be O(nlogn+n2) and O(n2) for heap and array respectively.

However, its not guaranteed that Fibonacci scheme will always outperform array scheme. Some of the reasons for it could be.
1.Hidden cost of large constants
2.Extra overhead of maintaining nodes and their data.
3.Heap initialization would cost lot of memory operations as new nodes are to be created and initialized. For large n value it could be considerable time.

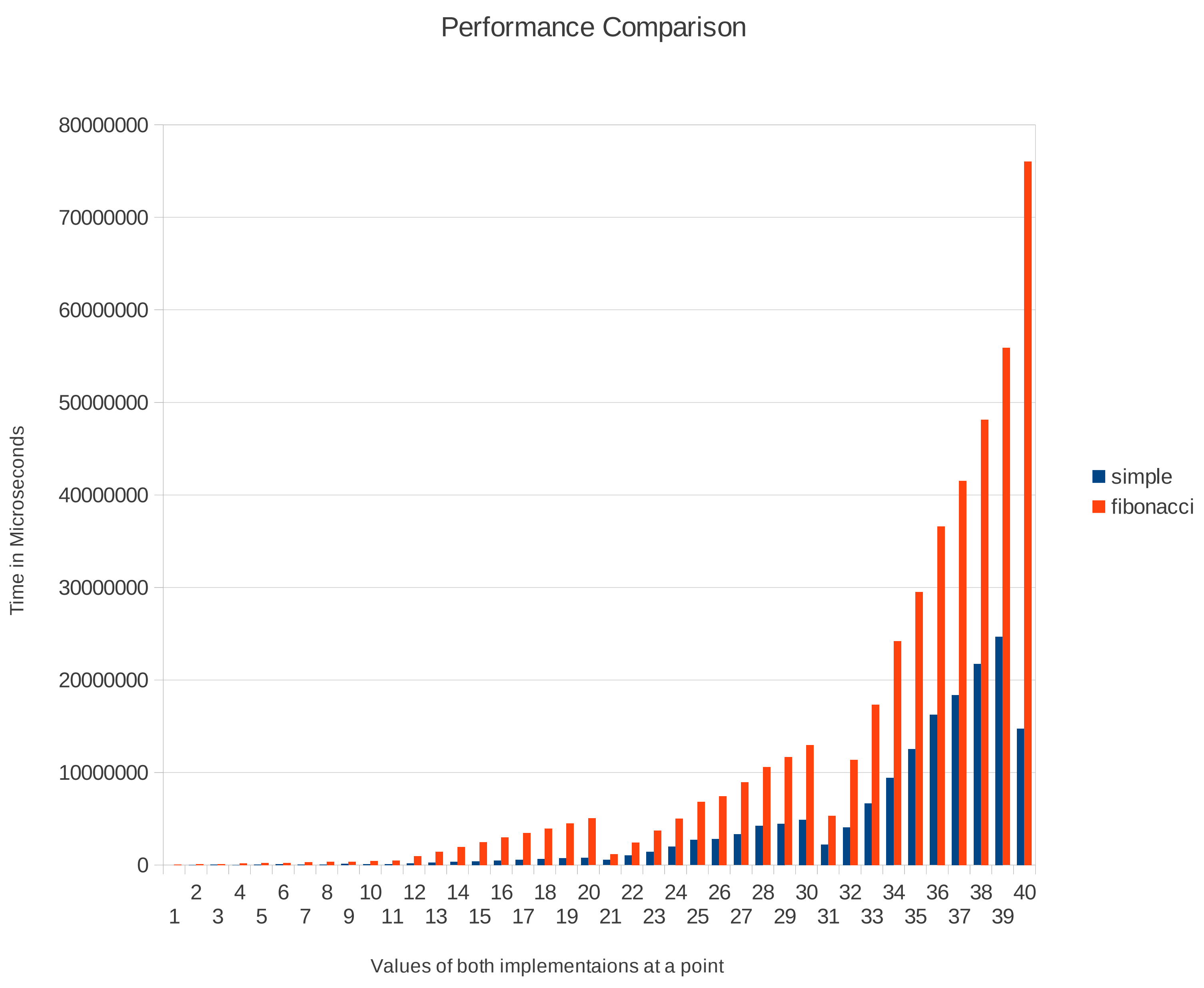
Observations:
My observation is that fibonacci scheme is asymptotically same as array scheme when density is small but when density is high, array scheme gave results faster compared to Fibonacci scheme. This behavior could be explained by the overhead of heap operations on small number of vertices.

The original expectation that Fibonacci heaps outperform array implementation for Prim's or other greedy algorithms will hold when we run with millions of nodes, but for small values of n, that is in few thousands Fibonacci heap implementation is not recommended.

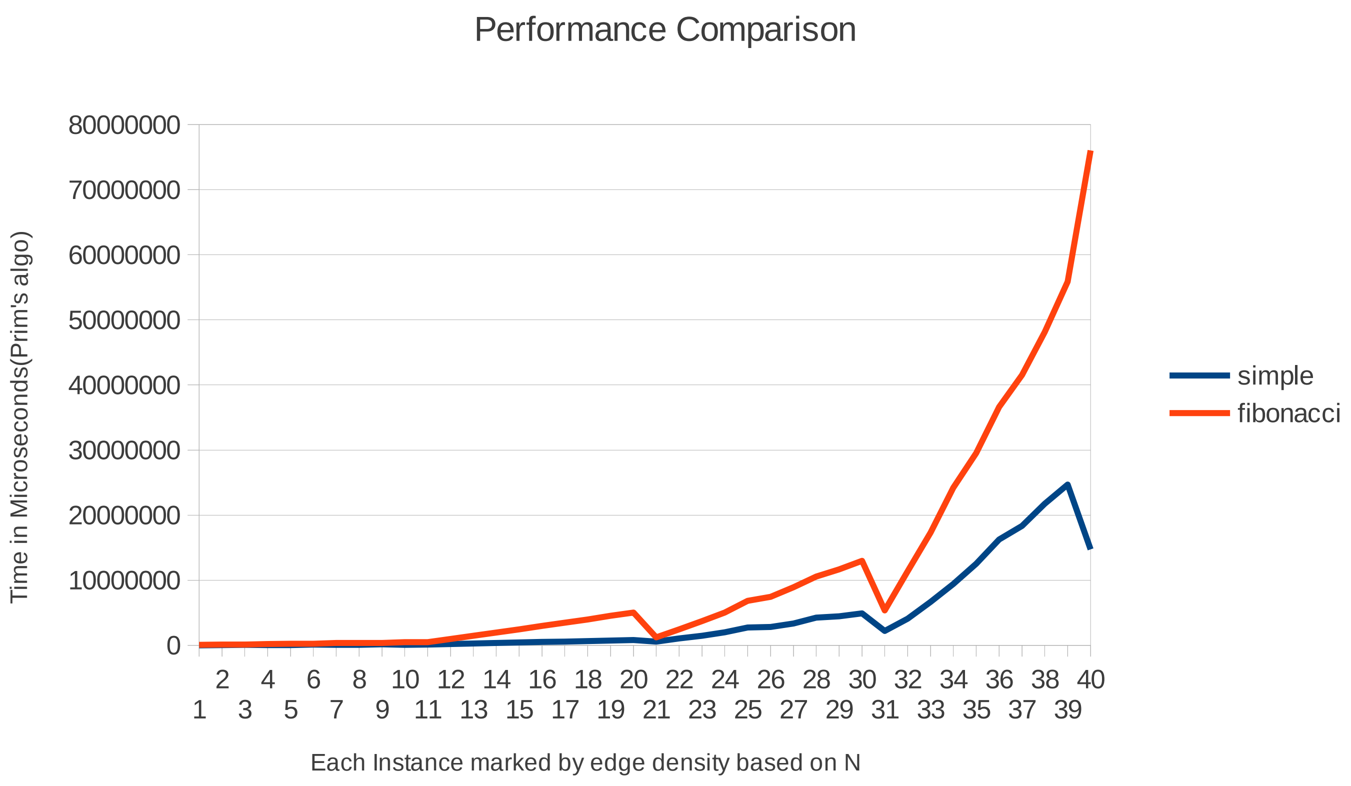
Below is the plot of my recorded times for implementations using simple array and Fibonacci schemes.

The x-axis shows the 40 points that I have run the program in random mode with following inputs
1000 nodes from 10% density to 100% density of edges.-------First 10 points
3000 nodes from 10% density to 100% density of edges.-------Second 10 points
5000 nodes from 10% density to 100% density of edges.--------Third 10 points
10000 nodes from 10% density to 100% density of edges.-------Fourth 10 points

I ran these tests on my laptop with Ubuntu 12.04 OS with core i5 processor
Intel(R) Core(TM) i5-3210M CPU @ 2.50GHz with 4GB RAM.



The same values with a line plot are is as shown below.



The x-axis values are 40 instances described above. The y axis is the time in Microseconds recorded for run time of Prim's algorithm.

Conclusion:

As we can see that for each of the case(value of n, that is number of nodes in the graph) as the density increases, the time gap between the run times also increases. This behavior is expected as mentioned in the expectation section. As the analysis is done for values of n=1000,3000,5000,10000 nodes, the performance of Fibonacci heap looks like a overkill. As the computational resources are not available to test the program for very large values of n like 1 million or more it is not possible to conclude that Fibonacci scheme should always outperform simple array scheme. But with the obtained results it can be said that Fibonacci scheme is not very bad compared to simple array scheme in any scenario asymptotically.

Fibonacci scheme would perform better than array scheme when the removeMin and decreaseKey operations are frequent and number of nodes are very large.

Run times in actual values for the 40 instances.

simple	fibonacci
0	50000
10000	100000
50000	120000
30000	190000
40000	240000
90000	240000
60000	330000
70000	370000
140000	350000
80000	470000
110000	480000
190000	960000
270000	1450000
350000	1960000
420000	2460000
500000	2960000
580000	3460000
660000	3960000
730000	4520000
810000	5040000
560000	1200000
1030000	2430000
1460000	3700000
1990000	5010000
2710000	6820000
2830000	7450000
3330000	8930000
4230000	10560000
4470000	11650000
4900000	12980000
2210000	5340000
4080000	11380000
6650000	17320000
9420000	24190000
12540000	29530000
16240000	36610000
18350000	41540000
21730000	48120000
24700000	55880000
14750000	76030000