COP 5536 fall '13

ASSIGNMENT ONE

First Name:	Sethuraman
Last Name:	Sundararaman
UFID:	11321142
UF Email:	susundar@cise.ufl.edu

Project Overview

To compare the performance of prim's algorithm using simple scheme with prim's algorithm using fibonacci heap.

Code Specification:

Language used: C++
Compiler used: g++ 4.8 2011 standards
IDE used: Netbeans 7.0.1

Compile Instruction:

>>Cd to <source directory>
>>make

Execution Instructions:

For file input mode using Simple Scheme :
>>./MST –s <filename>.txt
For file input mode using Fibonacci Scheme :
>>./MST –f <filename>.txt
For random input mode:
>> ./MST –r <number of nodes> <graph density>

Function Prototypes:

1. **char **v –** arguments passed to the program during run time
   v[1][1] – consists of information for mode
   v[2] – pass the no. of vertices if random otherwise filename for the input mode
   v[3] - density of graph in random mode
2. **class graph**
   struct Edge **edges- Linked list of edges for each of the vertices
   int *degree- holds the degree for each vertex
   int numVertices- Number of vertices
   unsigned long long int numEdges - Number of edges

fibheap *fib- pointer to the fibonacci heap

bfs()- performs a breadth first traversal to check if the graph is connected by trying to visit all the nodes.

prims(int start,mode)- generates minimum weight spanning tree by using primitive data structures ie., arrays.

1. Prims using simple scheme
2. Initialize a tree with a the start vertex, passed to the function
3. Grow the tree by one edge at a time: Of all the edges that connect the tree to vertices not yet in the tree, find the minimum-weight edge, and transfer it to the tree.
4. Repeat step 2 until all vertices are added to the tree.

insert_edge(int x, int y, int wei, int run): Insert edges source, destination, weight. Inserts twice for both source and destination vertices

fibPrims: Prims using fibonacci heap

1. Inserts all the vertices into the fibonacci heap with infinite key [1002]
2. Decreses key of start vertex to 0.
3. Removes Min [and add that to the MST] and decrease the key of its neighbors [if not in MST] with its corresponding distance
4. Repeats the step 3 for n-1 time

randomConstruct: Random Construction : If the density is not 100 random edges are inserted into the graph, otherwise all the edges with random weight is inserted into the graph.

Construct: Used to construct graph from an input file, passed as an argument

3. **class fibheap**

fibHpNode *heapmin- pointer to the min element in the Fibonacci heap**;**

fibHpNode **ref- maintain references to each and every node in the heap to enable decrease key of a particular node

void insert(fibHpNode *hnode) : Inserting into a Fibonacci heap. Called only by the fibonacci heap during initialization.

void hinsert(fibHpNode *hnode) : Inserting into a Fibonacci heap. Called only by the fibonacci heap during cascade cut procedure

int* hdeletemin() : Deletes the Minimum Element from Fibonacci Heap and returns the key and the element index.

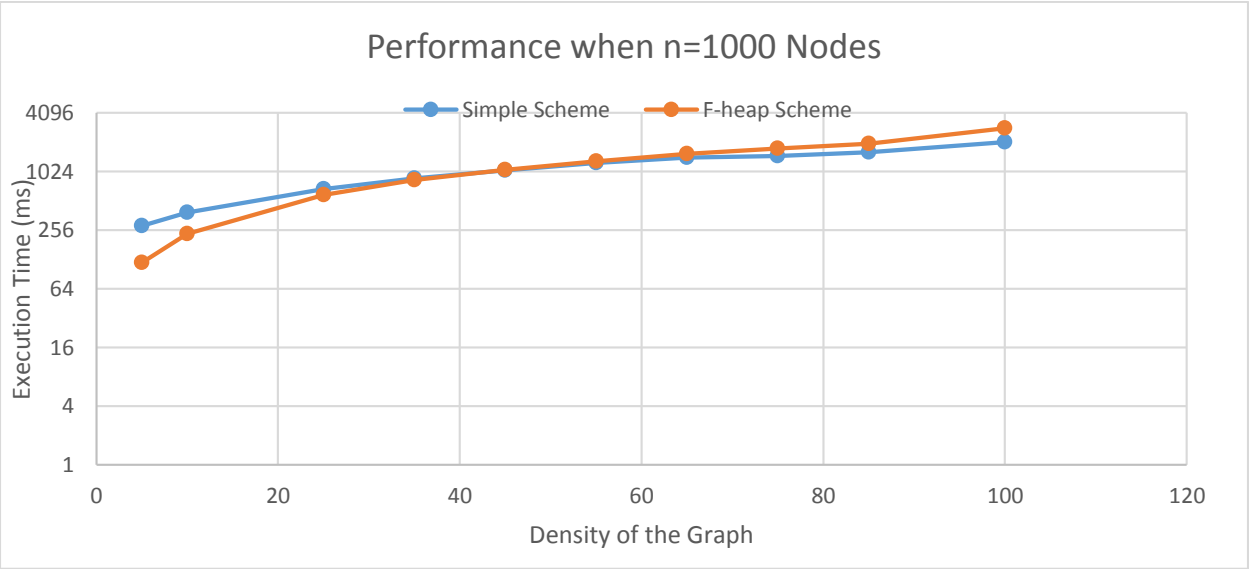
void consolidate(): Consolidate function to perform Pairwise combination

int decrease_key(int w, int e): Decrease Key function for Fibonacci Heap which decreases the key to “e” of “w th” vertex. The decrease_Key operation is performed by simply cutting the node from heap and then reinserting it incase if the operation violates the properties of the Fibonacci heap.

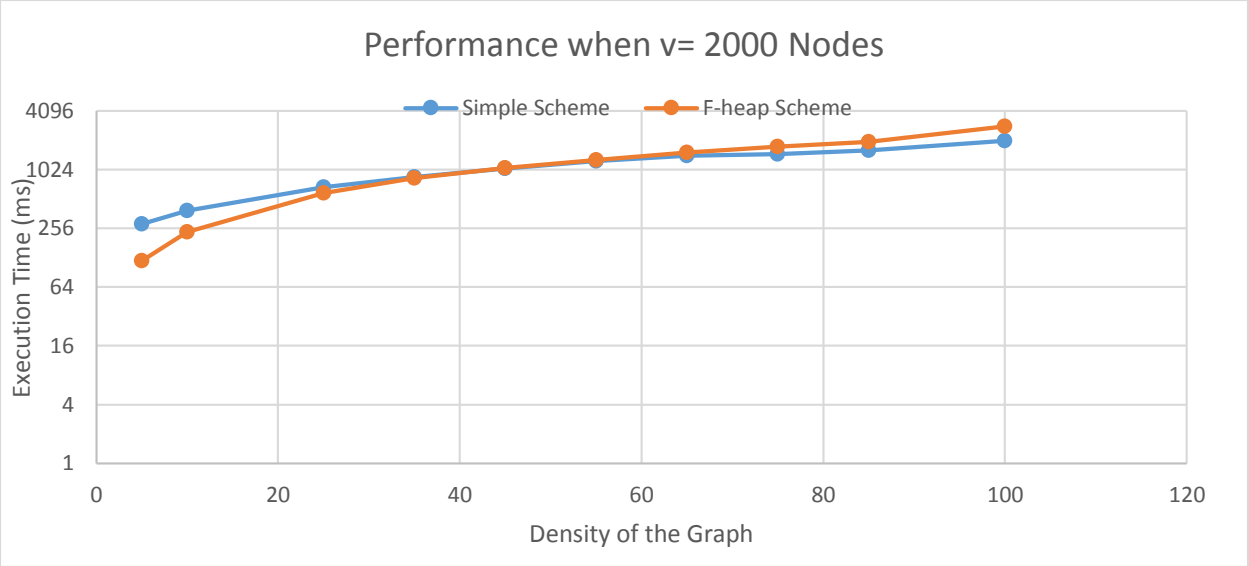
void cut(fibHpNode *om) : Cut operation to remove the node from the fibonacci heap. Used to ensure Fibonacci heap properties after decrease key operation.

void remove(int x) : Remove function based on index value

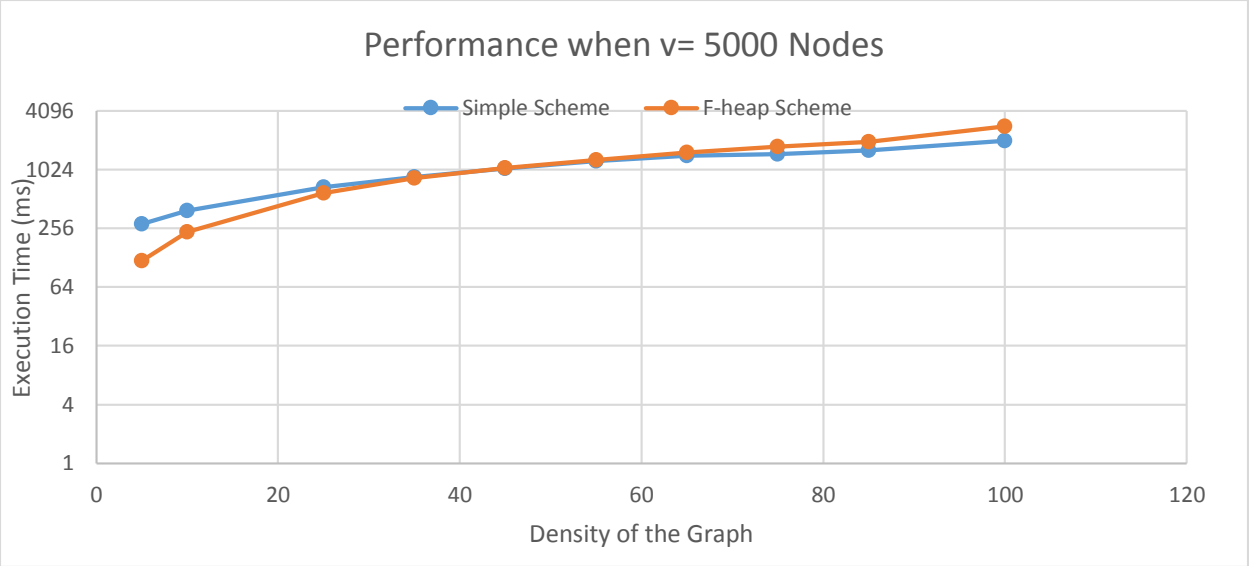
Result comparison



Density	Simple Scheme (in ms)	F-heap Scheme (in ms)
5	8	3.3
10	19	5.66
25	25	21
35	33	29
45	38	37
55	48	45
65	51	52
75	55	59
85	57	61
100	85	84



Density	Simple Scheme (time in ms)	F-heap Scheme (time in ms)
5	100	42
10	137	82
25	237	197
35	310	276
45	371	357
55	424	427
65	474	503
75	506	571
85	626	645
100	785	882



Density	Simple Scheme (time in ms)	F-heap Scheme (time in ms)
5	284	119
10	389	235
25	677	588
35	863	840
45	1051	1064
55	1250	1295
65	1420	1540
75	1480	1760
85	1610	1970
100	2031.6	2831

Result Analysis:

Simple scheme: it takes ∑deg(v) to analyze all the neighbors of every vertex. It boils down to O(2*E) where E is the number of edges. It takes O(N) to extract min at each iteration which sums up to $O(N^2)$. So the total time complexity is given by $O(E + N^2)$.

We can also observe that each analysis operation of a neighbor involves more overhead than extracting the min operation as the former makes use of linked list whereas the latter uses sequential arrays. As the density approaches 100 the complexity tends to $O(N^2)$ on the analyzing a neighbor of a vertex.

Fibonacci scheme:

The theoretical analysis suggests that the Fibonacci scheme takes O(E + VlogV). But the implementation imbibes an I/O overhead like a traditional tree using a linked structure. It takes O(E) time to decrease key of all E edges (same as analyzing neighbors of a simple scheme) and O(logV) to extract each minimum element[V times this operation gives O(V*logV)]

Apparently it looks like Fibonacci heap must outperform simple scheme. But experimental analysis does not show that as Fibonacci heap is heavily affected by the overhead of linked structures. But when the density of the graph is very low Fibonacci scheme really out performs simple scheme.

Even Fibonacci scheme tends to $O(N^2)$ when the density approaches 100%.

My observation: The use of Fibonacci scheme is highly recommended when the density of the graph is low.