Date: 17/Sep/2015

COL702: Advanced Data Structures
Programming Assignment – 1

Submitted By: Rajesh Kedia (2014CSZ8383)

Design Document

Section-1: Introduction

Problem Statement: Implement the below algorithms for Minimum Spanning Tree (MST) and compare their performance for various kinds of graphs:

1. Prim's algorithm using binary heaps
2. Fredman Tarjan (FT) algorithm using Fibonacci heaps

Graph Specification:
The graph to be provided as an input will be subjected to following constraints:

1. It will be a connected graph.
2. It will be a simple graph.
3. It will follow a format as per the below:

   The graph is given as a text file to the program. The format of the text file should be as follows:

   1) First node id's will be given on every line. Each line will have only one id.
   2) The end of node id's will be marked by a line containing #.
   3) Next, each line sholud specify an edge as follows:
      id1<space>id2<space><weight>

Expectation:

1. Fredman Tarjan (FT) algorithm using Fibonacci heap has a lower asymptotic execution time compared to Prim's algorithm using Binary heaps. But the constant factor in the asymptotic bounds are very high for Fredman Tarjan algorithm compared to Prim's algorithm.
2. Hence the expectation is that initially for smaller graphs, the FT algorithm will show larger execution time, but eventually will cross-over the execution time for Prim's algorithm for larger size of graph.

Section-2: Design Overview

The system is implemented using C++ as the programming language. Classes are used to define heaps and graphs as objects.

Design Overview for Prim's algorithm:
Below are the classes defined for the purpose of the above problem. The details about parameters and functions for each of the class is given in later sections of this chapter.

1. adjList: Implements an adjacency list for representing edges in a graph.
2. Graph: Implements a graph using adjacency list representation.
3. heapNode: class to represent each node of a binary heap.
4. minHeap : Implements a binary min heap using array of pointers.

In addition to the above classes, below functions are defined to implement the Prim's algorithm:

1. void createHeapFromGraph(Graph G, minHeap* heap): Takes a graph G as input and

generates an initial binary heap from it.

2. int prim(Graph G, Graph* mst): The function which actually implements the Prim's algorithm. It takes Graph G as input, converts to heap using createHeapFromGraph and uses the heap to implement the Prim's algorithm. The final MST is output in the mst instance while the cost of MST is returned as int.

Design Overview for Fredman Tarjan (FT)'s algorithm:

The system is implemented using C++ as the programming language. Classes are used to define heaps and graphs as objects. At very limited places, STLs are used from C++.

Below are the classes defined for the purpose of the above problem. The details about parameters and functions for each of the class is given in later sections of this chapter.

1. adjList: Implements an adjacency list for representing edges in a graph.
2. Graph: Implements a graph using adjacency list representation.
3. FibheapNode: class to represent each node of a Fibonacci heap.
4. Fibheap: Implements a fibonacci heap using FibheapNode as the base element.
5. edge: Represents an edge with source, destination and weight information to be used in the FT MST algorithm.
6. tree: Represents a single tree of the FT MST algorithm.
7. treelist: Uses tree as the base and implements a list of tree to be used in FT MST algorithm.

In addition to the above classes, functions are defined to implement the FT algorithm:

1. void createTreeListFromGraph(Graph *G, treelist *TL): Function to create a treelist from graph G. The function takes Graph G as input and outputs the treelist in TL.
2. int fredTarPass(treelist *TL, Graph *mst, int heapSize): The single pass of FT algorithm. It takes treelist TL and heapSize as input, uses them to compute the pass and updates the MST grown till now in mst.
3. void radixsort(std::vector<edge> edges, std::vector<edge> *sorted_edges, int count_tree): The function implements radixsort needed in the FT algorithm to remove duplicate edges after each pass. It takes an array of edges and then outputs the final list of edges in sorted_edges.
4. bool condense(treelist *TL): The function implements the condensing step as defined in the FT algorithm. It takes treelist TL as the input argument and updates it. It returns true or false to indicate whether further iterations are to be terminated or not.
5. int fredTarMST(Graph *G, Graph *mst): The actual FT algorithm implementation. It takes graph G as input and outputs the MST in graph mst. Internally, it calls the above functions to implement its functionality.

Section-3: Implementation Details:

A. Definition and function prototype of classes used for Prim's algorithm:

1. adjListNode

Variables:

public:

```
int dest;           //destination field
int weight;         //weight of the edge from the source to the destination
adjListNode* next;  //pointer to next adjacency list node
```

2. adjList:

public:
adjListNode* start; *//points to first member in the list connected to the vertex.*

3. Graph:
Variables:
public:
int num; *//number of elements in the graph*
adjList *array; *//array of adjacency list.*

Functions:
Graph(int n); *//constructor function. allocates space in memory for a graph of size n*
void init_array(void); *//member function to initialize the graph of size num with num vertices and no edges*
void addEdge(int src, int dest, int weight); *//add an edge of weight 'weight' between src and dest nodes*
void displayGraph(void); *//displays the graph as adjacency list.*

4. heapNode:
Variables:
public:
int vertex; *//vertex ID of the node.*
int source; *//ID of node from which the minimum distance exists.*
int key; *//actual key value of the node.*

5. minHeap
Variables/Parameters:
private:
int curSize; *//current number of elements in heap*
int maxSize; *//max. supported number of elements in heap*
int *pos; *//array to maintain the position of a vertex in heap.*
heapNode **array; *//array of pointers to heapnodes*

Functions:
public:
minHeap(int max); *//constructor function to allocate memory space*
void deleteMin(void); *//deletes the root (minimum) element from heap*
void display(void); *//displays the heap as a tree*
void insertNode(int vertex, int key); *//insert a node into the heap*
void buildHeap(); *//builds a heap if elements are stored in the array*
void decreaseKey(int vertex, int key, int source); *//decrease key of a vertex*
bool isEmpty(void); *//check if the heap is empty or not*
heapNode* peekMin(void); *//look into the parameters of the minimum element in the heap*
bool isInHeap(int node); *//check if a node is present in heap or not.*
private:
void moveup(int index); *//move upwards to maintain the heap property*
void movedown(int index); *//move downwards to maintain the heap property*
void swap (int index1, int index2); *//swap two nodes in the heap*
int left(int index); *//find the index of left child of a node*
int right(int index); *//find the index of right child of a node*
int parent(int index); *//find the index of parent of a node*

B. Details of the function implementation for the various classes in Prim's algorithm:

1. Graph:

Sl. No.	Function Name	Input Arguments	Output values	Function Description
1	init_array	None	None	member function to initialize the graph of size num with num vertices and no edges.
2	Graph	1. int n: corresponding to number of nodes in the graph	None	constructor function. allocates space in memory for a graph of size n and also initialize it as an empty graph without any edges.
3	addEdge	1. int src: source node connecting to the edge 2. int dest: destination node connecting to the edge 3. int weight: weight of edge between src and dest	None	adds an edge between src and dest nodes. Edge weight is weight. Since we are dealing with undirected graphs, we add 2 edges. One between src and dest other between dest and src. 2 nodes of adjacency list are allocated from heap and linked appropriately.
4	displayGraph	None	None	displays the graph as an adjacency list

2. minHeap:

Sl. No.	Function Name	Input Arguments	Output values	Function Description
1	minHeap (constructor)	int max	None	constructor function to initialize the heap to support upto max number of elements.
2	display	None	None	prints the elements of the heap for each level, from left to right.
3	left	int index	int	returns the index of the left child of a given node index.
4	right	int index	int	returns the index of the right child of a given node index.
5	parent	int index	int	returns the index of the parent of a given node index.
6	swap	int index1, int index2	None	swaps the nodes with indices index1 and index2. The swap process also involves updating the pos array accordingly.
7	moveup	int index	None	moves an element up in the heap until it finds a parent which has a key smaller than the given node.

8	movedown	int index	None	moves an element down in the heap until it finds that both children are larger than the given element.
9	deleteMin	None	None	deletes the minimum element (root of the heap) and re-adjusts remaining nodes so that the heap retains its properties.
10	insertNode	1. int vertex - the vertex number of the node to be added 2. int key -key value associated with the vertex	None	inserts a new element with id as "vertex" and key as "key" into the heap.
11	peekMin	None	heapNode	returns the handle to the node having the minimum key value (i.e. the root node of the heap). This function doesn't delete the minimum element.
12	isInHeap	int node	bool	checks if a given node is present in the heap or not. This function works correctly assuming that all inserts are done first followed by the delete operations, as is the case with the Prim MST algorithm.
13	isEmpty	None	bool	Check if the heap is empty or has atleast one element
14	decreaseKey	1. int vertex - id of the vertex whose key is to be reduced. 2. int key - new key to be used for the vertex 3. int source - stores the source node from which there is an edge to the node "vertex"	None	checks if the new key is smaller than existing key. If it is smaller, then it updates the key of node vertex and readjusts the nodes to maintain the heap property. If key is larger or equal, this function doesn't do anything

C. Implementation of Prim's algorithm using above classes:

The toplevel flow for the implementation of Prim's algorithm using the above classes is as follows:

1. Read graph input from file and store as graph in Graph class instance.
2. Call the prim() function to process graph and create MST.
3. Display the mincost to the user and finish the computation.

Sl. No.	Function Name	Input Arguments	Output values	Function Description
1	prim	1. Graph G -> The	int -> the	the function computes the mst for a given

		graph for which the minimum spanning tree is to be calculated 2. Graph* mst -> Points to an instance of a graph in which the mst is grown	cost of the mst grown is returned	graph by using binary minheaps using prims algo
2	createHeapFromGraph	1. Graph G -> The graph for which the minimum spanning tree is to be calculated 2. minheap* heap -> Points to an instance of the binary heap	none. The generated minHeap is returned via pointer	the function creates a binary minheap from a graph
3	main	int argc, char * argv: to parse command line arguments to read the file name	int : returns the status (default 0)	1. The main function parses the file containing the graph information. 2. Then it stores the information in a graph (instance of class Graph). 3. Then it invokes the MST algorithm (prim) and collects its output 4. It also instruments the code to compute the execution time 5. Finally it displays the cost of the MST

D. Definition and function prototype of classes used for FT's algorithm:
1. Graph: This is the same implementation as used in the Prim's algorithm.

2. FibheapNode:
Variables:
public:
```
        int key;
        int vertex;
        FibheapNode *parent;
        FibheapNode *child;
        FibheapNode *left;
        FibheapNode *right;
        bool marked;
        int degree;
```
Functions:
public:
```
        FibheapNode();
```

3. Fibheap:

Variables:
private:
int curSize; //current number of elements in heap
int maxSize; //max. supported number of elements in heap
std::map<int, FibheapNode *> pos; //hash map array to maintain the position of a vertex in heap.
FibheapNode *firstRoot; //pointer to first root of the fib heap
FibheapNode *minNode; //pointer to minimum element of the fib heap
Functions:
public:
Fibheap(int max); //constructor function, allocates space for max elements
~Fibheap(void); //destructor function, releases space
void deleteMin(void); //deletes the minimum element from heap
void display(void); //displays the heap as a tree
void insertNode(int vertex, int key); //insert a node into the heap
void decreaseKey(int vertex, int key); //decrease key of a vertex
void deleteNode(int vertex); //delete an arbitrary node in the heap
bool isEmpty(void); //check if the heap is empty or not
FibheapNode* peekMin(void); //look into the values of the minimum element
bool isInHeap(int node); //check if a node is present in heap or not.
void cutNode(FibheapNode *, FibheapNode *); //cut a node from its parent
void cascadeCut(FibheapNode *); //cascading cut operation to maintain the shape of the tree
FibheapNode* linkNodes(FibheapNode *, FibheapNode *);//make one node as child of other
bool hasChild(FibheapNode *); //check if child exists for a node
bool hasParent(FibheapNode *); //check if parent exists for a node
bool isMarked(FibheapNode *); //check if the node is marked
void consolidate(void); //consolidates heaps of same size and links then iteratively
bool hasSpace(void); //check if any space is available or heap is full
private:
FibheapNode* getFirstChild(FibheapNode *); //find the first child of a node
FibheapNode* getRightSibling(FibheapNode *); //find the right sibling of a node
FibheapNode* getLeftSibling(FibheapNode *); //find the left sibling of a node
FibheapNode* getParent(FibheapNode *); //find the parent of a node
void addtoList(FibheapNode *list, FibheapNode *node); //add a node to an existing list of nodes
FibheapNode* deleteFromList(FibheapNode *list, FibheapNode *node); //delete a node from existing list

4. egde:
Variables:
public:
int src; //source node of the edge (the current tree number in the iteration)
int dst; //destination node of the edge (the current tree number in the iteration)
int actsrc; //actual source node number (as per the original graph)
int actdst; //actual destination node number (as per the original graph)

```
        int weight;             //weight of the edge
```

5. tree:

Variables:

public:

```
        bool marked;            //indicates whether the tree is marked
        int tree_num;           // indicates the number assigned to the tree
        int key;                //current key for the tree
        bool inHeap;            //flag to indicate whether the tree is present in heap or not
        int src;                //source node from which the current distance is minimum
        int dst;
        int weight;             //weight is the current distance from src
        std::list<edge> E;      //a list of edges incident on the tree. each of them is an instance of
class edge
```

6. treelist:

Variables:

public:

```
        int num_edges;              //number of edges present in total
        std::vector<tree> T;        //an array of trees
        treelist();                 //constructor function to initialize num_edges to 0
```

E. Details of the function implementation for the various classes in FT's algorithm:

1. Fibheap

Sl. No.	Function Name	Input Arguments	Output values	Function Description
1	Fibheap (constructor)	int max	None	constructor function to initialize the heap to support upto max number of elements.
2	~Fibheap (destructor)	None	None	destructor function to free up space once out of scope
3	insertNode	1. int vertex - the vertex number of the node to be added 2. int key -key value associated with the vertex	None	inserts a new element with id as "vertex" and key as "key"into the heap.
4	consolidate	None	None	consolidate step as defined in the cormen book. This step actually joins the trees of similar degree and makes one as the child of other. As a process, some of the trees which are part of root becomes child of other trees and root list is reduced.
5	isMarked	FibheapNode *n -	bool	This function returns a boolean flag indicating

		pointer to a fibonacci heap node		whether the node is marked or not. 0: not marked, 1: marked
6	display	None	None	Function to display the entries in the fibonacci heap
7	peekMin	None	FibheapNode *	returns a pointer to the Fibonacci heapnode containing the minimum element
8	isInHeap	int node - node id for which we need to check if it is present in heap	bool	returns a boolean value to indicate whether the node is present in heap or not 0: not in heap, 1: is in heap
9	cutNode	1. FibheapNode *x: the node whose link from its parent is to be cut 2. FibheapNode *y: parent node of x	void	the function cuts the node x from the child list of y, adds x to the rootlist It also clears the marked flag of x algorithm referenced from cormen book
10	decreaseKey	1. int vertex: vertex ID for which the key is to be reduced 2. int key: new key to be assigned to the vertex	void	the function decreases the key of node with vertex ID to the new key provided the new key is lesser than existing key. The algorithm then cuts the node if the heap property is violated and then adds the tree to the root list. Then the algorithm also manages the tree shape by iteratively calling cut and cascade cut operations algorithm referenced from cormen book
11	cascadeCut	1. FibheapNode *y: the node from which the cascade cut is to be started	void	this function performs the cascading cut as defined for the fibonacci heap the process involves checking if the node is a root, then don't proceed further. It marks the node if it is not marked. If it is already marked, it cuts it from its parent and adds the parent to the root list. Does this iteratively. algorithm referenced from cormen book
12	hasSpace	None	Bool	Returns the boolean flag indicating whether the heap has any space or is it full. 0: No space, 1: space avaliable
13	deleteMin	None	None	the below function deletes the minimum element from the heap It firstly checks if the node has any child. If yes, it adds the child list to the root list. It then deletes the node from the root list. Afterwards, it calls the consolidate function to readjust the heap algorithm referenced from cormen book

14	deleteNode	int vertex: vertex ID of the node to be deleted	None	arbitrary node delete function. Deletes any node with vertex specified as the ID algorithm referenced from cormen book
15	linkNodes	1. FibheapNode *n1, FibheapNode *n2	FibheapNode *	links nodes n1 and n2. The one with the smaller key will be made the parent, other will be made its child. The function returns the ID of the node which is made the parent. It also updates the degree and number of children for the parent.
16	hasChild	1. FibheapNode *n	bool	The function checks if the node n has any child or not and returns the boolean flag for the same. 0: no child, 1: has child
17	addtoList	1. FibheapNode *list 2. FibheapNode *node	None	The function adds a Fibheapnode node to a list pointed to by list It takes care of appropriately adjusting the left and right pointers and parent pointer as well.
18	deleteFromList	1. FibheapNode *list 2. FibheapNode *node	Fibheapnode *	The function deletes a FibheapNode node from the list pointed to by list and returns the updated list It takes care of appropriately adjusting the left and right pointers and parent pointer and degree as well.
19	hasParent	1. FibheapNode *n	bool	The function checks if the parent exists for a given node n. return value: 0: parent not there, node is the root. 1: parent present

F. Implementation of FT's algorithm using above classes:

Sl. No.	Function Name	Input Arguments	Output values	Function Description
1	isInHeap	1.int tree: tree number for which to check 2. treelist *TL: treelist instance in which tree is present	bool	This function checks if a tree named 'tree' is present in treelist *TL It returns a boolean flag to indicate whether it is present or not in the heap 0: not present, 1: present
2	createTreeListFromGraph	1. Graph *G: graph G from which the treelist is to be created 2. treelist *TL: an	None	This function creates a treelist from an input graph. The graph is represented as an adjacency list.The function reads the various edges from graph and then creates a list of trees (each tree is a singleton node) with

		instance of treelist which is updated by the function		associated edges from each of them.Time complexity of this function is O(log E), E being number of edges in the tree
3	fredTarPass	1. treelist *TL - instance of the treelist on which to operate the algorithm 2. Graph *mst - the instance of the mst graph, where final MST is being built 3. int heapsize - the max. allowed heapsize for this iteration	int: the sum of weights of the edges added to MST in this Pass	This function implements a single pass of the Fredman Tarjan algorithm. The function uses a Fibonacci heap of size heapSize for storing the edges to find the minimum The MST is grown as a forest and an instance *mst of the Graph is used to store it. The function does the following steps iteratively (until all trees are marked): 1. Add any tree from the treelist to the heap (if it is not marked) with weight as 0. Set it to current tree 2. Extract the minimum node (minNode) from heap and mark it. If it is not the same as current tree, add to MST 3. If the minNode is already marked, then stop growing the current tree and start from next one 4. For all edges incident on minNode: a. if the neighbor is already in heap, and if current key is greater than edge weight, decrease key to edge weight b. if neighbor not in the heap and heap has space, add it to the heap with key as edge weight c. else stop the current growing tree 5. For trees present in the heap, set their key back to INFINITY. goto step-1
4	radixsort	1.std::vector<edge > edges: This is the array of edges which need to be sorted to remove duplicate ones 2.std::vector<edge > *sorted_edges: An instance of array of edges where the sorted edges are returned to the caller 3. int count_tree: count of number of trees present	None	The function sorts the array of edges based on their destination tree number. The array is already sorted by the source number based on the way this structure is handled in the caller function. For each source, the function uses an intermediate array arr to store the edges based on destination number and keeps on updating it if finds edges to same destination and uses weight as deciding parameter. After this, the edges are added back to sorted_edges array based on the tree numbers.
5	condense	1. treelist *TL: treelist for the	bool: indicates	The function performs the condensing step as defined in the Fredman & Tarjan algorithm

		current iteration	whether the MST iteration needs to be terminated. This is determined by checking if the number of edges remaining is 0	from the original paper. 1. It firstly parses the treelist to prepare an array of edges. Only edges which are going from one tree to another are kept, within same tree are dropped. 2. It also updates the src and dst tree number in the edges. 3. Then it calls the radixsort function to sort the edges and remove duplicate ones. 4. Finally it updates the treelist based on the new tree numbers and the remaining edges.
6	fredTarMST	1. Graph *G : original graph on which the MST is to be computed 2. Grpah *mst: the instance of graph on which the MST will be grown as forest	int : returns the cost of the MST for the graph	the function is the top function for the Fredman & Tarjan MST algorithm It does the below steps: 1. Create a treelist by parsing the graph 2. invoke a single pass of the Fredman Tarjan algo. (fredTarPass function) 3. Condense the treelist after the pass is over 4. If number of trees remaining is 1, then quit.
7	main	int argc, char * argv: to parse command line arguments to read the file name	returns the status (default 0)	1. The main function parses the file containing the graph information. 2. Then it stores the information in a graph (instance of class Graph). 3. Then it invokes the MST algorithm (fredTarMST) and collects its output 4. It also instruments the code to compute the execution time 5. Finally it displays the cost of the MST

Section-4: Verification and test data generation:

Verification Strategy:

Following verification strategy was used for verifying the Prim's and FT's algorithm:

1. Unit level verification was done during development of each of the function in the classes and for the complete class using some random operations (omitting details here).
2. Once Prim's algorithm was implemented, the verification of Prim's algorithm was done using some random graphs to debug basic issues. Once resolved and Prim's algorithm providing correct results for some of the smaller graphs, larger graphs were tried out.
3. Results for Prim's algorithm were taken as golden and was used to compare the results with the FT's algorithm.

Graph generation strategy:

FT's algorithm can converge the computation of MST using smaller heap size compared to Prim's. But it is dependent on the ratio of number of edges and number of nodes in the input graph. The

graph generation program supports 4 different modes:

1. only_adjacent: This is the simplest case where all nodes are connected only to 2 nodes in a circular fashion. FT's algorithm will converge with a heap size of 2 in this case.
2. low_fanout: The graph has low fanout for each of the node (max.4 fanout). Such graphs are excellent targets for using FT's algorithm and should get converged in a smaller heap size.
3. fully_connected: The generated graph is fully connected graph. In such cases, the FT's algorithm ends up using heap size equal to the number of nodes.
4. High_fanout: The graph is not fully connected, but some of the nodes have high fanout. Here, many of the nodes will be able to become part of MST with lower heap size, but the algorithm convergence will require a larger heap size.

Flow for graph generation:

1. In each of the case, a random number is generated within the range provided as an input and a graph of that size of vertices is created.
2. The edge weights are randomly assigned.
3. The patterns in which edges are generated is dependent on the mode selected. e.g. for the adjacent connected nodes, node (i) is connected to node (i+1) for all i within the generated random number.
4. For the low fanout and high fanout nodes cases, a first part ensures that all vertices are connected by atleast one node. Then randomly it generates a source & destination pair and selects it as an edge in the graph.

Time Computation method:

1. The execution time computation involves starting the clock timer before calling the MST algorithm and ending it just after the function call returns. The time for reading the file and storing it internally is not accounted for.
2. To avoid any discrepancy, all the debug related print statements within the function are removed.
3. There is a 2-3% variation seen by running the test for the same graph again and again. To account for this variability in the OS environment, an averaging is done by running the test for 5 times with the same graph.

All the instances of testing were automated using a bash shell script which generates out a csv file for analysis.

References:

1. Michael L. Fredman , Robert Endre Tarjan, Fibonacci heaps and their uses in improved network optimization algorithms, Journal of the ACM (JACM), v.34 n.3, p.596-615, July 1987
2. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. (MIT Press, Cambridge, MA, 2009)
3. Taken help from google search regarding some of the syntax and compile errors.