Design Document

Harman Kumar
2013CS10224

September 17, 2015

1 Overall Design

The whole project is divided into the following parts:

1. Implementing the data structures required by the algorithms, namely binary heap, union find and fibonacci heap.
2. Using the above mentioned data structures, implementing the prims MST algorithm and the fredman-tarjan MST algorithm.
3. Generating random graphs on which the the running times of the above mentioned MST algorithms would be tested.
4. A script for the automated testing of the algorithms on the generated graphs and timing them.

2 Random Graph Generation

The random graph generator was implemented in C++.

Some features of the random graph generator are as follows:

1. The generated graph is guaranteed to be connected. This is done by generating a tree and adding edges to it, until a graph of the desired properties is not formed.
2. The number of vertices and desired number of edges were taken as user parameters and the graph is generated accordingly.
3. The approach of adding vertices to the tree is as follows:

   Pick two random vertices:

   (a) If they are connected, do nothing.

(b) If not, add an edge with a randomly generated weight between them.

Do the above till either the desired number of edges is reached **OR** there is a timeout condition.
After the graph has been generated, print it out in a file.

3 Prims Algorithm and The Binary Heap

- The implementation of the binary heap provided by the boost heap library is used.
- The graph is read from a text file and stored in the form of an adjacency list.
- Prims algorithm is run on the graph and the MST is computed.
- The output of the program is the time taken to compute the MST of the input graph.

The input graph is read from the file **RandomGraph.txt**.

The MST, it's weight and the time taken are written in the file **prim_timing.txt**

For testing purposes, only the time was written, the MST and its weight could be given as output by uncommenting that part of code.

4 Fredman-Tarjan Algorithm and The Fibonacci Heap

In order to implement the Fredman-Tarjan algorithm, the fibonacci heap data structure was implemented.
Features of the Fibonacci Heap:

- The fibonacci heap implemented takes in a string and integer as key-value pair.
- The heap provides only the decrease key operation for modifying nodes, increase key is not implemented, since the algorithm requires only the former.

Following is the structure of a node of the fibonacci heap:

```
1 template <class V> struct node
2 {
3
```

```
public:
  // Data members.

  node<V>* prev;
  node<V>* next;
  node<V>* child;
  node<V>* parent;
  V value;
  int degree;
  bool marked;

  // Get functions.

  node<V>* getPrev() {return prev;}
  node<V>* getNext() {return next;}
  node<V>* getChild() {return child;}
  node<V>* getParent() {return parent;}
  V getValue() {return value;}
  bool isMarked() {return marked;}
  bool hasChildren() {return child;}
  bool hasParent() {return parent;}
};
```

Following is the structure of the fibonacci heap (only user accesible functions are mentioned here):

```
template <class V> class FibonacciHeap {
protected:
  node<V>* heap;  //Pointer to the min node.

public:

  FibonacciHeap() // Constructor
  {
    heap = NULL;
  }

  node<V>* emplace(V value) // Inserts a node in the heap with
    value V.
  {

  }

  void merge_lazy(FibonacciHeap& other) // Links the two heaps
    together and updates the min pointer.
  {

  }

  bool isEmpty() // Tells whether the heap is empty.
  {

  }

  V top() // Returns the value of the smallest element of the heap.
  {
```

```
29
30    }
31
32    V pop() // Removes the smallest element of the heap
33    {
34
35    }
36
37    void decreaseKey(node<V>* n, V value) // Decreases the value of
        the node pointed by n to the new value.
38    {
39
40    }
```

In order to maintain disjoint sets of vertices, the Union-Find data structure is implemented. The **Union by rank** and **Path compression** heuristics were added in order to make the it efficient.

```
1  class UF
2  {
3      int cnt;  //count of the number of disjoint sets.
4      unordered_map<string, int> sz;  // Size of a set.
5      unordered_map<string, string> mapping;  // For parent relation.
6
7  public:
8      UF(set<string> str) // Create an union find data structure with
        N isolated sets.
9
10     ~UF() // Destructor.
11
12     string find(string p) // Return the id of component
       corresponding to object p.
13
14     void merge(string x, string y)  // Replace sets containing x
       and y with their union.
15
16     bool connected(string x, string y)   // Are objects x and y in
       the same set?
17
18     int count()   // Return the number of disjoint sets.
19
20 };
```

The graph is read from a text file and stored in the form of an adjacency list. The Fredman-Tarjan algorithm is applied on the graph and the weight of the MST is given as output.

The input graph is read from the file **RandomGraph.txt**.

The MST, it's weight and the time taken are written in the file **tarjan_timing.txt**

For testing purposes, only the time was written, the MST and its weight could be given as output by uncommenting that part of code.

5 Automated Testing

This is done by a python script (Automated_Tester.py). This script Generates a random graph, and runs the two algorithms on the graph. The time taken by the algorithms in written in a text file, which is later read and recorded.

The timing of the algorithms was done inside the C++ code itself by using the c++11 <chrono> library. The recorded time was in milliseconds in order to get a accurate execution time. The time was that taken by the algorithm to compute the MST of the graph, exclusive of any time spent on file I/O etc.

The following was done in order to get a better idea of the running times:

1. While compiling the programs, the **-o0** flag was used in order to turn off all compiler optimizations.
2. Disk caching was disabled, so that every memory access takes the same amount of time.
3. All the other processes running were turned off so that all the resources are available for testing.

The tests were carried out for the following categories of graphs:

1. Very sparse (near tree) graphs.
2. sparse graphs.
3. dense graphs.

6 Acknowledgements

- Boost C++ libraries, for the boost::heap library, and a few other libraries. (used in the implementation of prims algorithm).
- Introduction to Algorithms by CLRS, for the pseudocode for fibonacci heap, that was translated into C++ code.
- g++ documentation and windows caching documentation for turning off all levels of compiler optimizations and disk caching.