BLG372E

Analysis of Algorithms

2013 Spring

Report of Project 2

Date of Submission : 17.04 2013

Student Name: Volkan İlbeyli

Student Number : 040100118

Instructor : Hazım Ekenel

Introduction

Given three data sets, each containing a list of cities and flight paths between cities along with sources, destinations and costs, we are asked to implement Kruskal's algorithm and find the minimum spanning tree using a disjoint-set data structure.

Data Structures

Data structures used in the implementation is as follows:

- ### Edge

```
class Edge {
public:
    //constructors & destructors
    Edge();
    Edge(const Edge& orig);
    virtual ~Edge();
    Edge(unsigned const int, const int, const int);

    //getters & setters
    unsigned int GetCost() const {return cost;}
    unsigned int GetCity1() const {return city1;}
    unsigned int GetCity2() const {return city2;}

    //member functions
    bool operator<(const Edge*) const;
private:
    unsigned int cost;  //cost of the edge
    unsigned int city1, city2;   //cities that the edge connect
};
```

An `Edge` class is created to store flight path information. The class has three unsigned integers, each containing source city and destination city as indexes and the cost. Since the flight path information is kept as an `Edge*` vector, the < operator was overrode to compare the cost of the object itself and another objects cost which is passed to the `operator<` as an `Edge*`. In the main file, a functor (**func**tion compara**tor** `struct`) is implemented to compare `Edge*` pointers, as shown below:

```
//comparator functor for Edge* list
struct CompareCost{ //since two pointers cannot be compared in a member function
   bool operator () (const Edge* e1, const Edge* e2){
      return e1->operator <(e2);   //see how member operator< is invoked
   }
};
```

- ### STL Containers

Several STL containers were used. Some of them are as follows, along with `typedef`'s:

```
typedef vector<Edge*> vE;
typedef vector<Edge*>::iterator vEit;
typedef list<Edge*> lE;
typedef map<string, int> msi;
typedef list< set<int> > lsi;
typedef set<int> si;
```

In the main function, there are multiple lists and vectors.

```
vE edges;       //vector<Edge*>
lE edge_list;   //list<Edge*>
msi cities;     //map<int, string>
lE MST;     //minimum spanning tree
```

edges: A vector of Edge* that contains all the flight paths in the provided files.
edge_list: A list contains `Edge*` pointers which are randomly chosen from the `edges` vector.
cities: A map that contains `string` and `int` pairs. This data structure is sort of a translator, used when printing the `Edge*` vectors or `Edge*` lists for translating the `int` contained in object into the city name, associated with its index.
MST: The minimum spanning tree, list of `Edge*` pointers returned by the Kruskal's algorithm.

The set container is used in the algorithm as the local variable which contained the city indexes to implement the Kruskal's algorithm efficiently, by uniting the sets when an edge that has it's endpoints in different sets is added to the tree.

```
lE tree;    //tree, edge list
lsi city_sets; //list of set of ints
```

Algorithm

Kruskal's algorithm is one of the algorithms used for finding the minimum spanning tree in a weighed graph. The algorithm first sorts the edges by comparison based on the edge weight, i.e. cost. Then initializes an empty list, which will contain the edges that form the minimum spanning tree. After that, it makes each of the nodes that are provided as argument to the function a singleton set. Then comes the main loop which iterates through the sorted edges and adds the edges to the tree list if the edges do not create a cycle until all the nodes form a connected graph, a tree. The condition of creating cycles is tested by checking whether the endpoints of the edge are in the same set or not. If they are in the same set, adding the edge will create a cycle, therefore when this situation occurs, the algorithm continues with the next edge.

Pseudo-code

The algorithm in the main program is the C++ implementation of the following pseudo-code:

```
Kruskal(G, c) {
   Sort edges weights so that c1 ≤ c2 ≤ ... ≤ cm.
   T ← ϕ
   foreach (u ∈ V) make a set containing singleton u
   for i = 1 to m
      (u,v) = ei
      if (u and v are in different sets) {
         T ← T ∪ {ei}
         merge the sets containing u and v
      }
   return T
}
```

However, to implement the `if (u and v are in different sets)` block, a few iterations were necessary. The pseudo-code of the iterations are explained below. It's also sufficient to read the

comments in the `kruskal()` function. Besides those, since the provided edges are randomly chosen from the edge vector, they might not maintain a connected graph. Therefore the Kruskal's algorithm also checks at the end of the iterations if the resultant graph (tree) is connected or not.

```
Kruskal(G, c) {
   Sort edges weights so that c1 ≤ c2 ≤ ... ≤ cm.        O(mlogm)
   T ← ϕ                                                 O(1)
   foreach (u ∈ V)                                       O(n)
       make a set containing singleton                   |  O(1)
       add them in list L                                L  O(1)
   for i = 1 to m                                        O(m)
       (u,v) = ei                                        |  O(1)
       Iterate L to find set of u                        |  O(2n) in total
       When set of u is found                            .
           Iterate through the set of u                  .
           if(v is not in the set of u)                  .
               Iterate through L to find the set of v
               When set of v is found
                   T ← T ∪ {ei}
                   merge the sets containing u and v
           else
               break and continue edge iteration
   if(there are more than 1 set in the list L)
       display "an MST is not found!"
   else
       display "a minimum spanning tree is found"
   return T
}
```

Complexity Analysis

The iterations through the set list L and sets themselves has, in total, $O(n)$ running time when summed since when a set is merged with another, the size of the list shrinks by and the populated set's size grows by the merged set's size, thus $O(n)$. If we are to consider the search for the second city, the complexity becomes $O(2n)$, which is of order $O(n)$. The whole algorithm has $O(mlogm)$ complexity due to the sorting of edges has the highest order of growth.

The `main()` Function

The main function can be outlined as below.

```
main(){
       program parameter error check;
       read the files and initialize containers with data;
       print initial data (all edges and city map);
       scan input value for K;

       // TEST 1
       ask user if she wants to see the steps of the algorithm;
       do{
              randomly pick K edges;
              handle random pick errors;
              print kruskal parameters info;
              invoke kruskal();
       }while(kruskal doesn't return a connected graph(tree));

       print the minimum spanning tree;
       calculate the running time of the algorihtm;
       output test 1 data to file;
       print test 1 data to screen;


       // TEST 2
       do the same procedure with K = K*2;

       // TEST 3
       do the same procedure with K = K*3;

       return;
   }
```

The test results will both be in the output file and on the screen. Note that basic step count will only appear on the screen but not in the file.

There are some functions for "pretty printing". Normally, a function will print a list or a vector of Edge* pointers with the integer values of the cities which they connect. There exists an overridden function with an additional parameter, city map container. When that function is called, the list or vector is printed by the city names along with costs. The main function currently prints without city names, but if one desires to see city names instead of just numbers, she may invoke the "pretty printing" functions by providing the city map container to the function as the last parameter. Function prototypes will make themselves clearer now:

```
//function prototypes
bool read_input(string, string, msi&, vE&);
void print_vector(vE&, msi&);
lE randEdge(vE, unsigned const int);
int print_list(lE&, msi&);
int print_list(lE&);
string find_key(msi&, int);
void print_map(msi&);
void print_sets(lsi&);
void print_set(si&);
lE kruskal(lE&, msi&);
lE kruskal(lE&, msi&, bool);
double kruskal_time(lE&, msi&);
```

Besides those “pretty printing” functions, there appears to be 3 `kruskal()` functions. The 2nd `kruskal()` with three parameters contains extra lines printing the algorithm computations step by step. That function is called when the user types ‘Y’ or ‘y’ to the question asked by `ask user if she wants to see the steps of the algorithm;`. It can be considered good practice to call that function on small inputs to see how the implemented Kruskal algorithm works.

The final function simply runs the Kruskal algorithm many times with the same input to calculate the running time of the algorithm. That function exists due to the nature of the main function since it would be impossible to achieve the “nice program flow” otherwise.

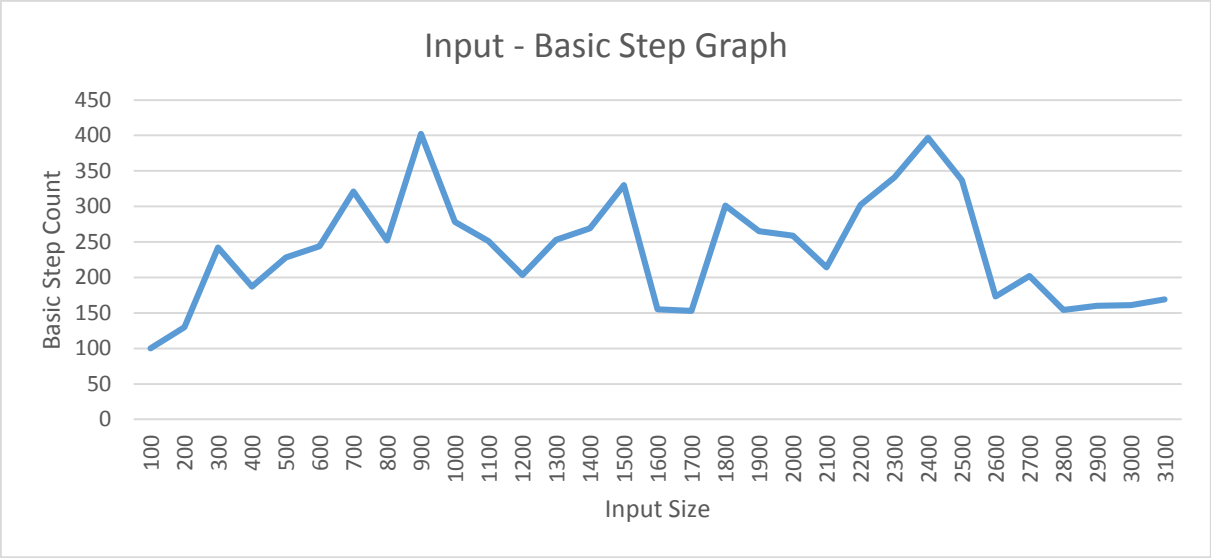
The program is self-explanatory enough with help of the comments and user friendly program flow. Therefore if more details are sought, one may simply read the contents of the “main.cpp” file. The key parts are covered above.

Output Interpretation

Edge count	Basic step	Running Time (ms)
100	100	1168
200	130	1124
300	242	1193
400	187	1142
500	228	1133
600	244	1122
700	321	1148
800	252	1098
900	402	1121
1000	278	1183
1100	251	1095
1200	203	1176
1300	253	1123
1400	269	1177
1500	330	1138
1600	155	1189
1700	153	1118
1800	301	1218
1900	265	1124
2000	259	1128
2100	214	1111
2200	302	1113
2300	341	1132
2400	397	1159
2500	337	1123
2600	173	1202
2700	202	1155
2800	154	1168
2900	160	1159
3000	161	1119
3100	169	1144

The test are done on the largest input and the output data is given. Interpreting the data, we can see that running time is irrelevant. This is due to there is no standard and generally-correct way to measure running time. Also it should be noted that running time will change from machine to machine.

Considering that the tests are done only once when creating the table, the values might be misleading since they are not mean values but are instantaneous values. We can see a trend in basic step to rise and then fall as the edge count grows. It was observed that the basic step count gets fixed as the maximum number of edges are reached, i.e. when K > total edge count.



Here you see the graph of the table. As expected, we first see a rise in the basic step count, and then it starts to fall as the input size grows, finally reaching its final value of 169 steps.

Compilation & Running

To compile the program, all three files, main.cpp – Edge.h – Edge.cpp should be in the same directory along with the input files.

Compile: `g++ main.cpp Edge.cpp`
Run: `./a.out Cities_10.txt Flights_10.txt result10.txt`

It should be noted that even though many error handling is done, not all error mechanisms are implemented. When user is asked to give an input, say the int value K, the user should enter an integer. Otherwise the program will go into infinite loop. One other not-implemented-error-checking is the input files. Do not provide different input files of different magnitude and do not cause a mismatch between the cities that are read from file and cities that edges connect.