CSC 225 – LAB 8 PART 1

CLARE

In this lab I examine 3 different algorithms to find the shortest paths between two vertices in a graph, as discussed in the textbook:

1. Ford Algorithm
2. Label Correcting Algorithm using queue
3. Label Correcting Algorithm using deque

I write a simple class Graph as following:

```
class graph{
private:
    int vertices; // # of vertices & edges
    map<pair<int, int>, int> edges; // pair<source,dest>, weight
public:
    graph(){
        vertices = 0;
    }
    bool readFile(string file);
    void showGraph();
    void friend printDist(vector<int> dist, vector<int> visits, int source);
    void FordAlgorithm(int source);
    void labelCorrectingAlgorithmQueue(int source);
    void labelCorrectingAlgorithmDeque(int source);
};
```

Each graph object will store the total number of vertices and a map of edges. The reason behind using map to store edges is that they will be stored in ascending order in terms of their key (which is the source and destination of each edge). A graph object reads a graph from a text document. The text document must be formatted as follow:

```
0 1 2 3 4 5 6 7 8
0 1 1
1 4 -5
2 3 1
2 6 1
2 7 1
3 0 2
3 4 4
3 8 1
4 5 4
6 3 -1
7 6 -1
8 5 1
```

The first row includes the list of vertices, with each vertex separated by a space. Each subsequent row contains 3 numbers that represent an edge. The first number represents the source vertex, the second represents the destination vertex, and the third is its weight. The vertices need not to be written in any order, as they will be stored ascendingly when inserted into the edges map anyway. Also, for simplicity, I name the vertices "0", "1", "2", …, "8" in place of "a", "b", "c", …, "i".

Here is the method that reads a text file:

```
bool graph::readFile(string file){
    string line;
    ifstream f(file);
    if (!f.good())
        return false;
    // READ FIRST LINE THAT CONTAINS LIST OF VERTICES
    getline(f, line);
    stringstream temp(line);
    string val;
    while (getline(temp, val, ' '))
        vertices++;
    // READ THE REST THAT CONTAINS LIST OF EDGES
    while (getline(f, line)){
        stringstream temp(line);
        string val;
        int curr[3], i = 0;
        while (getline(temp, val, ' ')){
            curr[i] = atof(val.c_str());
            i++;
        }
        i = 0;
        edges.insert(make_pair(make_pair(curr[0], curr[1]), curr[2]));
    }
    return true;
}
```

In main, I basically ask the user to input the file name, and then enter the source vertex (which should be from 0 to 8):

```
int main(void){
    int source;
    graph myGraph;
    string file;
    cout << "Enter data file (such as graph.txt): ";
    getline(cin, file);
    if (!myGraph.readFile(file)){
        cout << "Error reading file\n";
        return 1;
    }
    myGraph.showGraph();
    cout << "Enter a source node (0 to 8): ";
    cin >> source;
    if (0 <= source && source <= 8){
        cout << "\nFord Algorithm:\n";
        myGraph.FordAlgorithm(source);
        cout << "\nLabel Correcting Algorithm using queue:\n";
        myGraph.labelCorrectingAlgorithmQueue(source);
        cout << "\nLabel Correcting Algorithm using deque:\n";
        myGraph.labelCorrectingAlgorithmDeque(source);
    }
    else
        cout << "Invalid source" << endl;
    return 0;
}
```

FORD ALGORITHM:

```
void graph::FordAlgorithm(int source){
    vector<int> visits(vertices, 0);
    vector<int> dist(vertices, INT_MAX);
    dist[source] = 0;
    for (unsigned int i = 1; i < dist.size(); i++){
        for (map<pair<int, int>, int>::const_iterator j = edges.begin(); j != edges.end(); ++j){
            int u = j->first.first;
            int v = j->first.second;
            visits[v]++;
            int weight = j->second;
            if (dist[u] != INT_MAX && dist[u] + weight < dist[v]){
                dist[v] = dist[u] + weight;
            }
        }
    }
    printDist(dist, visits, source);
    return;
}
```

The method accepts an integer as the source vertex, and then find the shortest paths to every other vertices. For V vertices and E edges in the map, this algorithm always runs [V – 1] iterations, and in each iteration runs through every edge once. Thus it yields a complexity of O(|V|*|E|).

Note that for every algorithm, I use the vector visits to track how many times each edge is visited.

Here is the output using Ford Algorithm and setting 2 as the source node:

```
Ford Algorithm:
Shortest paths:
        2 -> 0 : 1
        2 -> 1 : 2
        2 -> 2 : 0
        2 -> 3 : -1
        2 -> 4 : -3
        2 -> 5 : 1
        2 -> 6 : 0
        2 -> 7 : 1
        2 -> 8 : 0
Number of visits per vertex:
        Vertex 0: 8 visit(s)
        Vertex 1: 8 visit(s)
        Vertex 2: 0 visit(s)
        Vertex 3: 16 visit(s)
        Vertex 4: 16 visit(s)
        Vertex 5: 16 visit(s)
        Vertex 6: 16 visit(s)
        Vertex 7: 8 visit(s)
        Vertex 8: 8 visit(s)
```

LABEL CORRECTING ALGORITHM USING QUEUE:

```
void graph::labelCorrectingAlgorithmQueue(int source){
    vector<int> visits(vertices, 0);
    vector<int> dist(vertices, INT_MAX);
    dist[source] = 0;
    queue<int> tobeChecked;
    tobeChecked.push(source);
    while (!tobeChecked.empty()){
        int v = tobeChecked.front();
        tobeChecked.pop();
        for (map<pair<int, int>, int>::const_iterator i = edges.begin(); i != edges.end(); ++i){
            if (i->first.first == v){
                int weight = i->second;
                int u = i->first.second;
                visits[u]++;
                if (dist[u] > dist[v] + weight){
                    dist[u] = dist[v] + weight;
                    tobeChecked.push(u);
                }
            }
            else if (i->first.first > v)
                break;
        }

    }
    printDist(dist, visits, source);
}
```

While the Ford Algorithm checks every edge every time, which is not necessary, the above Label Correcting Algorithm uses the tobeChecked queue as the order in which edges are visited. Every time a vertex is visited and dequeued, its adjacent vertices are enqueued if their current distances are updated. Here is the output using the above Label Correcting Algorithm, also setting vertex 2 as the source:

```
Label Correcting Algorithm using queue:
Shortest paths:
        2 -> 0 : 1
        2 -> 1 : 2
        2 -> 2 : 0
        2 -> 3 : -1
        2 -> 4 : -3
        2 -> 5 : 1
        2 -> 6 : 0
        2 -> 7 : 1
        2 -> 8 : 0
Number of visits per vertex:
        Vertex 0: 3 visit(s)
        Vertex 1: 3 visit(s)
        Vertex 2: 0 visit(s)
        Vertex 3: 3 visit(s)
        Vertex 4: 6 visit(s)
        Vertex 5: 8 visit(s)
        Vertex 6: 2 visit(s)
        Vertex 7: 1 visit(s)
        Vertex 8: 3 visit(s)
```

We see that the number of visits per edge decreases significantly when compared to the Ford Algorithm.

LABEL CORRECTING ALGORITHM USING DEQUE:

A disadvantage of using queue to store tobeChecked is that in a queue you can only access and insert at the front of the queue, and remove at the back of the queue. Using a deque would give more versatility, as elements in both the front and back can be accessed, inserted, and removed. More specifically, we traverse the queue from back to front, and if a vertex is added to tobeChecked for the first time, it would be inserted at the back of the queue, and otherwise inserted at the front of the queue. Here is the code:

```
void graph::labelCorrectingAlgorithmDeque(int source){
    vector<int> visits(vertices, 0);
    vector<int> dist(vertices, INT_MAX);
    dist[source] = 0;
    deque<int> tobeChecked;
    vector<bool> enqueue(vertices, false);
    tobeChecked.push_back(source);
    enqueue[source] = true;
    while (!tobeChecked.empty()){
        int v = tobeChecked.back();
        tobeChecked.pop_back();
        for (map<pair<int, int>, int>::const_iterator i = edges.begin(); i != edges.end(); ++i){
            if (i->first.first == v){
                int weight = i->second;
                int u = i->first.second;
                visits[u]++;
                if (dist[u] > dist[v] + weight){
                    dist[u] = dist[v] + weight;
                    if (enqueue[u] == false){
                        tobeChecked.push_back(u);
                        enqueue[u] = true;
                    }
                    else
                        tobeChecked.push_front(u);
                }
            }
            else if (i->first.first > v)
                break;
        }
    }
    printDist(dist, visits, source);
}
```

And here is the output, setting source vertex as 2:

When compared to using queue, we visit each vertex even less. The author states using deque on average performs at least 60% better than queue, but using deque has the worst case of visiting each edge exponential times.