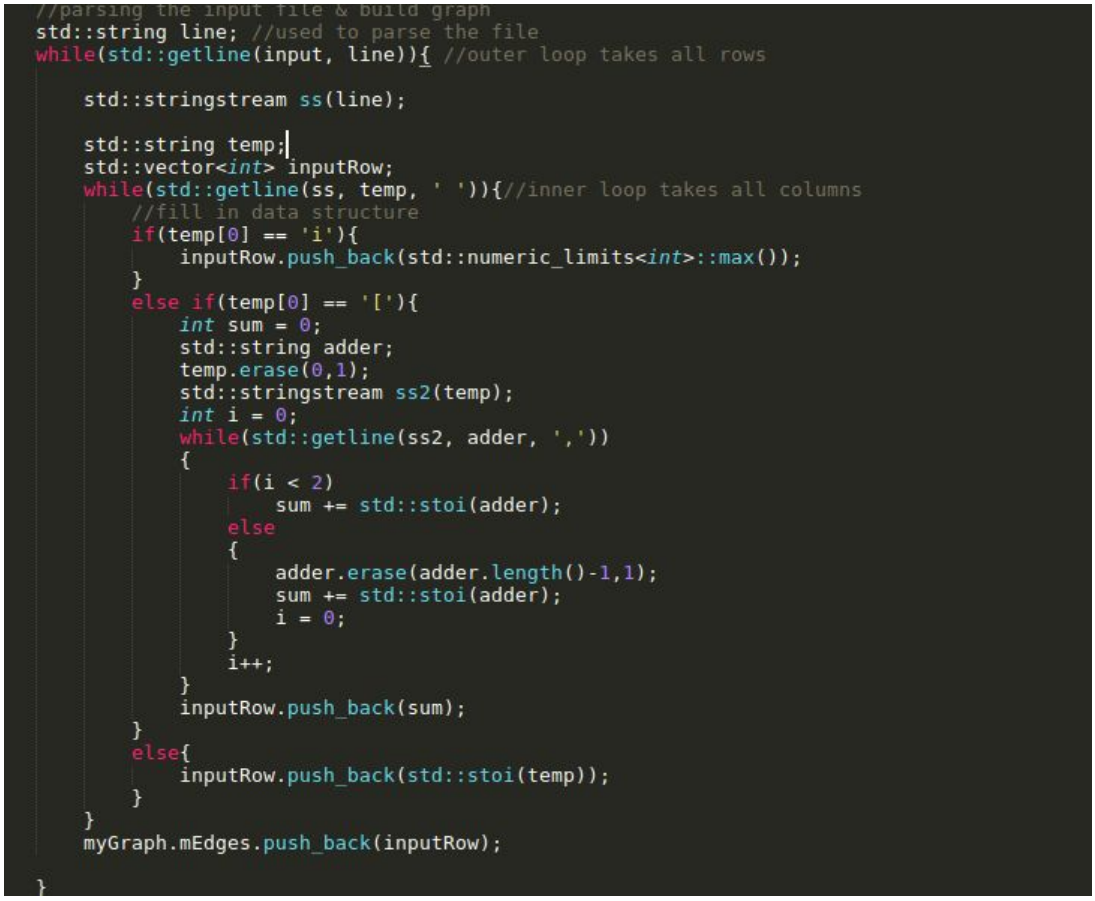
An Pham
9477698118

Lab 1 Bellman-Ford

Objective:
Design a Bellman-Ford Algorithm that would find the shortest path between the source know and all other n-1 nodes, detect a negative cycle in the graph or nodes that do not have a path from the source, and output these values, path, and iteration of the algorithm.

Code Overview:
The C++ program, “bellman.cpp” can be broken down into 3 parts:

1. Read in the input graph from a text file.



```
//parsing the input file & build graph
std::string line; //used to parse the file
while(std::getline(input, line)){ //outer loop takes all rows

    std::stringstream ss(line);

    std::string temp;
    std::vector<int> inputRow;
    while(std::getline(ss, temp, ' ')){//inner loop takes all columns
        //fill in data structure
        if(temp[0] == 'i'){
            inputRow.push_back(std::numeric_limits<int>::max());
        }
        else if(temp[0] == '['){
            int sum = 0;
            std::string adder;
            temp.erase(0,1);
            std::stringstream ss2(temp);
            int i = 0;
            while(std::getline(ss2, adder, ','))
            {
                if(i < 2)
                    sum += std::stoi(adder);
                else
                {
                    adder.erase(adder.length()-1,1);
                    sum += std::stoi(adder);
                    i = 0;
                }
                i++;
            }
            inputRow.push_back(sum);
        }
        else{
            inputRow.push_back(std::stoi(temp));
        }
    }
    myGraph.mEdges.push_back(inputRow);

}
```

   a. Outer while loop goes through the entire text file row by row
   b. Inner while loop goes through the columns of each row
   c. The If/Else ladder determines the different types of input.
      i. Case: INF - set edge to infinity

ii. Case: [- sum up the edge costs
iii. Case: Integer - just

2. Bellman-Ford.
    a. Bellman Ford can be broken down into three portions:
        i. Initialize
        ii. Bottom-Up Dynamic Programming Approach
        iii. Checking for Negative Cycles
    b. Initialization is straight forward. Set the D(x) for all x vector to infinity except for the source node, which is 0 for our lab.
    c. The Bottom-Up Dynamic Approach uses nested for-loops to build the array from the base case. Details are in the lab instructions; however, one thing to remember is that integer can happen when representing infinity as the maximum integer.

```
if((graph.mEdges[j][k] != 0) && (graph.mEdges[j][k] != std::numeric_limits<int>::max()) ){ //c
    if(graph.distances[j] != std::numeric_limits<int>::max()) //prevent int overflow
    {
        if(graph.distances[j] + graph.mEdges[j][k] < graph.distances[k])
        {
            graph.path[k] = j;
            graph.distances[k] = graph.distances[j] + graph.mEdges[j][k];
            graph.term = true;
        }
    }

}
```

    The first if statement ignores edges in the AdjMatrix that are 0 or infinity, while the second if statement prevents overflow from happening if graph.distances[j] + graph.mEdges[j][k] would cause the number to be a negative.
    d. Checking for Negative Cycles works by seeing if we ran the algorithm one more time, it would produce a change in the d(x) vector. If so then we have a negative cycle.

3. Output:
    a. The output can be separated into three portions:
        i. Shortest Distance Values
        ii. Paths
        iii. Iterations

```
//output
if(!myGraph.neg)
{
    //distances
    for(int i=0; i<myGraph.distances.size(); i++){
        results << myGraph.distances[i];
        if(i < myGraph.distances.size()-1)
        {
            results << ",";
        }
        else
            results << std::endl;
    }

    //paths
    for(int i=0; i<myGraph.path.size(); i++){

        int current = i;
        std::vector<int> way; //empty list
        while(current != -1){
           way.push_back(current);
           current = myGraph.path[current];
        }

        for(int j=way.size()-1; j>=0; j--){
            results << way[j];
            if(j != 0)
            {
                results << "->";
            }
            else
                results << std::endl;
        }
    }

    //Iterations
    results << "Iterations:" << myGraph.iterations << std::endl;

}
```

Key Aspects and Implementations:

1. Graph Representation:

Data for the graph is stored in a the “Struct Graph.” Here we represent the graph using an AdjMatrix that holds the edge weights between graphs. In addition, the graph has valuable

member data that is use to output the results.

```
//Graph Structure AdjMatrix
struct Graph
{
    //contains the edge weights of the nodes
    //size of the graph is the #of nodes
    std::vector<std::vector<int>> mEdges;

    //output data
    int iterations;
    std::map<int,int> distances;
    std::map<int,int> path; //only 1 predecessor
    bool term; //used for early termination
    bool neg;
    int negPred; //used to track a negative cycle
};
```

2. Representing infinity:
Bellman Ford's algorithm requires us to initialize the d(x) for all x to infinity. To do this we use #include <limits> //std::numeric_limits<int>::max();

Conclusion:
Overall there are many ways to program this lab, ranging from different ways to represent graphs or different ways to implement the Bellman-Ford algorithm. Future improvement for this lab would be to improve the algorithm with parallelization. Because we run the overall algorithm n-1 times, we can have the nodes run in parallel and use a common-shared resource, such as the boolean that indicates if any nodes has updated the graph.