Homework 5 (5pt.)

Submission instruction:

Submit one single pdf file for this homework including both coding problems and analysis problems.

For coding problems, copy and paste your codes. Report your results.

For analysis problems, either type or hand-write and scan.

Question 1 (3 pt.) Write codes for Dijkstra's algorithm using unsorted array for priority Q.

```
Active Edges
-----------
{0: 0, 1, 10}
{1: 0, 2, 5}
{2: 1, 2, 2}
{3: 1, 3, 1}
{4: 2, 3, 9}
{5: 2, 1, 3}
{6: 2, 4, 2}
{7: 3, 4, 4}
{8: 4, 3, 6}
{9: 4, 0, 7}


Dijsktra's Algorithm Results
----------------------------
Node    Parent  Distance
0       0       0
1       2       8
2       0       5
3       1       9
4       2       7
bash-4.2$
```

Question 2 (2 pt.) Write codes for Bellman-Ford algorithm.

```
Active Edges
-----------
{0: 0, 1, 6}
{1: 0, 2, 7}
{2: 4, 0, 2}
{3: 1, 2, 8}
{4: 1, 3, 5}
{5: 3, 1, -2}
{6: 1, 4, -4}
{7: 2, 3, -3}
{8: 2, 4, 9}
{9: 4, 3, 7}


Bellman-Ford Results
--------------------
Node    Parent  Dist
0               0
1       3       2
2       0       7
3       2       4
4       1       -2
bash-4.2$
```

```cpp
//  Dijkstra's Algorithm        //
//  8 December 2019             //
//  Author: Anna DeVries        //

/*      Libraries                       */
#include <iostream>
#include <stdlib.h>
#include <vector>
#include <bits/stdc++.h>
#include <limits>
#include <stdio.h>

/*      Struct objects                  */
//  Edge object
struct Edge {
    int src;
    int dst;
    int weight;

    Edge(int s, int d, int w) : src(s), dst(d), weight(w) {};
};

//  Graph object
struct Graph {
    int V;
    int E;
    std::vector<Edge> edges;

    Graph(int v) : V(v) {
        E = 0;
        edges = std::vector<Edge>();
    }
};

/*      Graph functions                 */
//  Add edge to graph
void addEdge(Graph* graph, int src, int dst, int weight){
    graph->edges.push_back(Edge(src, dst, weight));
    graph->E++;
}

/*      Utility functions               */
//  Print edges
void printAdjList(Graph graph){
    std::cout << "Active Edges" << std::endl;
    std::cout << "------------" << std::endl;
    //  Print each node and its destinations
    for(int i = 0; i < graph.E; i++){
        std::cout << "{" << i << ": " << graph.edges[i].src;
        std::cout << ", " << graph.edges[i].dst << ", ";
        std::cout << graph.edges[i].weight << "}" << std::endl;
    }
    std::cout << std::endl;
}

//  Extracts minumum element from queue
int extract_min(std::vector<int> d, std::vector<bool> q){
    //  Local variables
    int min = INT_MAX;
    int index = 0;

    //  Finds first unvisted element
    for(int i = 0; i < q.size(); i++){
        if (!q[i]){
            index = i;
            min = d[i];
            break;
        }
    }
```

```cpp

    //  Finds min element
    for(int i = 0; i < d.size(); i++){
        if(d[i] < min && !q[i]){
            index = i;
            min = d[i];
        }
    }

    //  Returns min element's node
    return index;
}

/*      Dijkstra's Algorithm        */
void dijkstra(Graph graph, int startPoint){
    //  Local variables
    std::vector<int> d;
    std::vector<int> parents;

    //  Start INITIALIZE-SINGLE-SOURCE
    for (int i = 0; i < graph.V; i++) {
        d.push_back(INT_MAX);
        parents.push_back(-1);
    }

    d[startPoint] = 0;
    parents[startPoint] = 0;
    //  End INITIALIZE-SINGLE-SOURCE

    //  Initialize priority Q
    std::vector<bool> Q;
    for (int i = 0; i < graph.V; i++) {
        Q.push_back(false);
    }

    for(int i = 0; i < graph.V; i++){
        int u = extract_min(d, Q);
        Q[u] = true; // Extract min element from Q

        for(auto edge : graph.edges){
            //  Start RELAX FUNCTION
            if(edge.src == u){
                if(!Q[edge.dst] && d[edge.dst] > d[u]+edge.weight){
                    d[edge.dst] = d[u] + edge.weight;
                    parents[edge.dst] = u;
                }
            }
            //  End RELAX FUNCTION
        }
    }

    //  Print results
    printf("\nDijsktra's Algorithm Results\n");
    printf("---------------------------\n");
    std::cout << "Node\tParent\tDistance" << std::endl;
    for (int i = 0; i < graph.V; i++) {
        if (parents[i] < 0) {
            std::cout << i << "\t\t" << d[i] << std::endl;
        }
        else {
            printf("%d\t%d\t%d\n", i, parents[i], d[i]);
        }
    }
}

/*      Main Function       */
int main(){
    //  Local variables
    int src = 0;
```

```
    int V = 5;
    Graph graph = Graph(V);

    //  Add edges to graph
    addEdge(&graph, 0, 1, 10);
    addEdge(&graph, 0, 2, 5);
    addEdge(&graph, 1, 2, 2);
    addEdge(&graph, 1, 3, 1);
    addEdge(&graph, 2, 3, 9);
    addEdge(&graph, 2, 1, 3);
    addEdge(&graph, 2, 4, 2);
    addEdge(&graph, 3, 4, 4);
    addEdge(&graph, 4, 3, 6);
    addEdge(&graph, 4, 0, 7);

    //  Print edges of graph
    printAdjList(graph);

    //  Perform shortest path algorithm
    dijkstra(graph, src);
}
```

```cpp
//  Bellman-Ford's Algorithm    //
//  8 December 2019             //
//  Author: Anna DeVries        //

/*      Libraries                        */
#include <iostream>
#include <stdlib.h>
#include <vector>
#include <bits/stdc++.h>
#include <limits>

/*      Globals                          */
std::vector<int> parent;

/*      Typedefs                         */
typedef struct Edge_* Edge;
typedef struct Graph_* Graph;

/*      Struct objects                   */
// Edge object
struct Edge_{
    int src;
    int dst;
    int weight;
};

// Graph object
struct Graph_{
    int V;
    int E;
    Edge edge;
};

/*      Graph functions                  */
// Creates the graph
Graph createGraph(int V, int E){
    // Local Variables
    Graph graph;
    int i;

    // Initialize graph
    graph = (Graph) malloc(sizeof(struct Graph_));
    (*graph).V = V;
    (*graph).E = E;
    (*graph).edge = (Edge) malloc(sizeof(struct Edge_) * E);

    // Return function
    return graph;
}

// Add point to graph
void addEdge(Graph graph, int i, int src, int dst, int weight){
    // Create node for directed graph
    (*graph).edge[i].src = src;
    (*graph).edge[i].dst = dst;
    (*graph).edge[i].weight = weight;
}

// Destroy graph
Graph destroy(Graph graph){
    // Free heap
    free((*graph).edge);
    free(graph);

    // Return empty graph
    graph = NULL;
    return graph;
}

```

```
/*      Utility functions                   */
//  Prints graph as an adjacency list
void printAdjList(Graph graph){
    //  Local variables
    int i;

    std::cout << "Active Edges" << std::endl;
    std::cout << "------------" << std::endl;

    //  Print each node and its destinations
    for(i = 0; i < (*graph).E; i++){
        std::cout << "{" << i << ": " << (*graph).edge[i].src;
        std::cout << ", " << (*graph).edge[i].dst << ", ";
        std::cout << (*graph).edge[i].weight << "}" << std::endl;
    }

    std::cout << std::endl;
}

//  Prints results
void print_result(std::vector<int> d, int V){
    //  Local variables
    int i;

    std::cout << std::endl;
    std::cout << "Bellman-Ford Results\n" << "--------------------" << std::endl;
    std::cout << "Node\tParent\tDist" << std::endl;

    for(i = 0; i < V; i++){
        if(parent[i] < 0){
            std::cout << i << "\t" << "\t" << d[i] << std::endl;
        }
        else{
            std::cout << i << "\t" << parent[i] << "\t" << d[i] << std::endl;
        }
    }
}

//  Initialize distances
std::vector<int> initialize_single_source(Graph graph, int src, std::vector<int> d){
    //  Local variables
    int i;

    //  Set distances to infinity
    for(i = 0; i < (*graph).V; i++){
        d.push_back(std::numeric_limits<int>::max());
        parent.push_back(-1);
    }

    //  Set src at 0
    d[src] = 0;

    //  Return array
    return d;
}

//  Relax all edges
std::vector<int> relax(std::vector<int> d, int u, int v, int weight){
    //  Determine if/how to relax distance
    if(d[u] != std::numeric_limits<int>::max() && d[u] + weight < d[v]){
        d[v] = d[u] + weight;
        parent[v] = u;
    }

    //  Return array
    return d;
}

/*      Bellman-Ford Algorithm              */
```

```
bool bellman_ford(Graph graph, int src){
    //  Local variables
    int i, j, u, v;
    std::vector<int> d;

    // Initialize distances
    d = initialize_single_source(graph, src, d);

    // Relax edges
    for(i = 1; i < (*graph).V - 1; i++){
        for(j = 0; j < (*graph).E; j++){
            u = (*graph).edge[j].src;
            v = (*graph).edge[j].dst;

            d = relax(d, u, v, (*graph).edge[j].weight);
        }
    }

    //  Check for negative-weight cycles
    for(i = 0; i < (*graph).V; i++){
        u = (*graph).edge[i].src;
        v = (*graph).edge[i].dst;

        if(d[u] != std::numeric_limits<int>::max() && d[u] + (*graph).edge[i].weight <
        d[v]){
            return false;
        }
    }

    //  Print result
    print_result(d, (*graph).V);

    //  Return true
    return true;
}

/*      Main                                */
int main(){
    //  Local variables
    int src = 0;
    int V = 5;
    int E = 10;

    //  Create graph
    Graph graph = createGraph(V, E);

    //  Add edges and nodes to graph
    //  Example from book
    addEdge(graph, 0, 0, 1, 6);
    addEdge(graph, 1, 0, 2, 7);
    addEdge(graph, 2, 4, 0, 2);
    addEdge(graph, 3, 1, 2, 8);
    addEdge(graph, 4, 1, 3, 5);
    addEdge(graph, 5, 3, 1, -2);
    addEdge(graph, 6, 1, 4, -4);
    addEdge(graph, 7, 2, 3, -3);
    addEdge(graph, 8, 2, 4, 9);
    addEdge(graph, 9, 4, 3, 7);

    //  Print graph as an adjacency list
    printAdjList(graph);

    //  Solve single-source shortest-paths problems
    bellman_ford(graph, src);

    //  Free memory
    destroy(graph);

    return 0;
```

```
}
```