

## 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}

Dijkstra'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      0
1       3      2
2       0      7
3       2      4
4       1     -2
bash-4.2$
```

```

1 // Dijkstra's Algorithm //
2 // 8 December 2019 //
3 // Author: Anna DeVries //
4
5 /* Libraries */
6 #include <iostream>
7 #include <stdlib.h>
8 #include <vector>
9 #include <bits/stdc++.h>
10 #include <limits>
11 #include <stdio.h>
12
13 /* Struct objects */
14 // Edge object
15 struct Edge {
16     int src;
17     int dst;
18     int weight;
19
20     Edge(int s, int d, int w) : src(s), dst(d), weight(w) {};
21 };
22
23 // Graph object
24 struct Graph {
25     int V;
26     int E;
27     std::vector<Edge> edges;
28
29     Graph(int v) : V(v) {
30         E = 0;
31         edges = std::vector<Edge>();
32     }
33 };
34
35 /* Graph functions */
36 // Add edge to graph
37 void addEdge(Graph* graph, int src, int dst, int weight){
38     graph->edges.push_back(Edge(src, dst, weight));
39     graph->E++;
40 }
41
42 /* Utility functions */
43 // Print edges
44 void printAdjList(Graph graph){
45     std::cout << "Active Edges" << std::endl;
46     std::cout << "-----" << std::endl;
47     // Print each node and its destinations
48     for(int i = 0; i < graph.E; i++){
49         std::cout << "{" << i << ": " << graph.edges[i].src;
50         std::cout << ", " << graph.edges[i].dst << ", ";
51         std::cout << graph.edges[i].weight << "}" << std::endl;
52     }
53     std::cout << std::endl;
54 }
55
56 // Extracts minimum element from queue
57 int extract_min(std::vector<int> d, std::vector<bool> q){
58     // Local variables
59     int min = INT_MAX;
60     int index = 0;
61
62     // Finds first unvisited element
63     for(int i = 0; i < q.size(); i++){
64         if (!q[i]){
65             index = i;
66             min = d[i];
67             break;
68         }
69     }

```

```

70
71 // Finds min element
72 for(int i = 0; i < d.size(); i++){
73     if(d[i] < min && !q[i]){
74         index = i;
75         min = d[i];
76     }
77 }
78
79 // Returns min element's node
80 return index;
81 }
82
83 /*      Dijkstra's Algorithm      */
84 void dijkstra(Graph graph, int startPoint){
85     // Local variables
86     std::vector<int> d;
87     std::vector<int> parents;
88
89     // Start INITIALIZE-SINGLE-SOURCE
90     for (int i = 0; i < graph.V; i++) {
91         d.push_back(INT_MAX);
92         parents.push_back(-1);
93     }
94
95     d[startPoint] = 0;
96     parents[startPoint] = 0;
97     // End INITIALIZE-SINGLE-SOURCE
98
99     // Initialize priority Q
100    std::vector<bool> Q;
101    for (int i = 0; i < graph.V; i++) {
102        Q.push_back(false);
103    }
104
105    for(int i = 0; i < graph.V; i++){
106        int u = extract_min(d, Q);
107        Q[u] = true; // Extract min element from Q
108
109        for(auto edge : graph.edges){
110            // Start RELAX FUNCTION
111            if(edge.src == u){
112                if(!Q[edge.dst] && d[edge.dst] > d[u]+edge.weight){
113                    d[edge.dst] = d[u] + edge.weight;
114                    parents[edge.dst] = u;
115                }
116            }
117            // End RELAX FUNCTION
118        }
119    }
120
121    // Print results
122    printf("\nDijkstra's Algorithm Results\n");
123    printf("-----\n");
124    std::cout << "Node\tParent\tDistance" << std::endl;
125    for (int i = 0; i < graph.V; i++) {
126        if (parents[i] < 0) {
127            std::cout << i << "\t\t" << d[i] << std::endl;
128        }
129        else {
130            printf("%d\t%d\t%d\n", i, parents[i], d[i]);
131        }
132    }
133 }
134
135 /*      Main Function      */
136 int main(){
137     // Local variables
138     int src = 0;

```

```
139     int V = 5;
140     Graph graph = Graph(V);
141
142     // Add edges to graph
143     addEdge(&graph, 0, 1, 10);
144     addEdge(&graph, 0, 2, 5);
145     addEdge(&graph, 1, 2, 2);
146     addEdge(&graph, 1, 3, 1);
147     addEdge(&graph, 2, 3, 9);
148     addEdge(&graph, 2, 1, 3);
149     addEdge(&graph, 2, 4, 2);
150     addEdge(&graph, 3, 4, 4);
151     addEdge(&graph, 4, 3, 6);
152     addEdge(&graph, 4, 0, 7);
153
154     // Print edges of graph
155     printAdjList(graph);
156
157     // Perform shortest path algorithm
158     dijkstra(graph, src);
159 }
160
161
```

```

1 // Bellman-Ford's Algorithm //
2 // 8 December 2019 //
3 // Author: Anna DeVries //
4
5 /* Libraries */
6 #include <iostream>
7 #include <stdlib.h>
8 #include <vector>
9 #include <bits/stdc++.h>
10 #include <limits>
11
12 /* Globals */
13 std::vector<int> parent;
14
15 /* Typedefs */
16 typedef struct Edge_ * Edge;
17 typedef struct Graph_ * Graph;
18
19 /* Struct objects */
20 // Edge object
21 struct Edge_{
22     int src;
23     int dst;
24     int weight;
25 };
26
27 // Graph object
28 struct Graph_{
29     int V;
30     int E;
31     Edge edge;
32 };
33
34 /* Graph functions */
35 // Creates the graph
36 Graph createGraph(int V, int E){
37     // Local Variables
38     Graph graph;
39     int i;
40
41     // Initialize graph
42     graph = (Graph) malloc(sizeof(struct Graph_));
43     (*graph).V = V;
44     (*graph).E = E;
45     (*graph).edge = (Edge) malloc(sizeof(struct Edge_) * E);
46
47     // Return function
48     return graph;
49 }
50
51 // Add point to graph
52 void addEdge(Graph graph, int i, int src, int dst, int weight){
53     // Create node for directed graph
54     (*graph).edge[i].src = src;
55     (*graph).edge[i].dst = dst;
56     (*graph).edge[i].weight = weight;
57 }
58
59 // Destroy graph
60 Graph destroy(Graph graph){
61     // Free heap
62     free((*graph).edge);
63     free(graph);
64
65     // Return empty graph
66     graph = NULL;
67     return graph;
68 }
69

```

```

70  /*      Utility functions      */
71  // Prints graph as an adjacency list
72  void printAdjList(Graph graph){
73      // Local variables
74      int i;
75
76      std::cout << "Active Edges" << std::endl;
77      std::cout << "-----" << std::endl;
78
79      // Print each node and its destinations
80      for(i = 0; i < (*graph).E; i++){
81          std::cout << "{" << i << ": " << (*graph).edge[i].src;
82          std::cout << ", " << (*graph).edge[i].dst << ", ";
83          std::cout << (*graph).edge[i].weight << "}" << std::endl;
84      }
85
86      std::cout << std::endl;
87  }
88
89  // Prints results
90  void print_result(std::vector<int> d, int V){
91      // Local variables
92      int i;
93
94      std::cout << std::endl;
95      std::cout << "Bellman-Ford Results\n" << "-----" << std::endl;
96      std::cout << "Node\tParent\tDist" << std::endl;
97
98      for(i = 0; i < V; i++){
99          if(parent[i] < 0){
100              std::cout << i << "\t" << "\t" << d[i] << std::endl;
101          }
102          else{
103              std::cout << i << "\t" << parent[i] << "\t" << d[i] << std::endl;
104          }
105      }
106  }
107
108  // Initialize distances
109  std::vector<int> initialize_single_source(Graph graph, int src, std::vector<int> d){
110      // Local variables
111      int i;
112
113      // Set distances to infinity
114      for(i = 0; i < (*graph).V; i++){
115          d.push_back(std::numeric_limits<int>::max());
116          parent.push_back(-1);
117      }
118
119      // Set src at 0
120      d[src] = 0;
121
122      // Return array
123      return d;
124  }
125
126  // Relax all edges
127  std::vector<int> relax(std::vector<int> d, int u, int v, int weight){
128      // Determine if/how to relax distance
129      if(d[u] != std::numeric_limits<int>::max() && d[u] + weight < d[v]){
130          d[v] = d[u] + weight;
131          parent[v] = u;
132      }
133
134      // Return array
135      return d;
136  }
137
138  /*      Bellman-Ford Algorithm      */

```

```

139 bool bellman_ford(Graph graph, int src){
140     // Local variables
141     int i, j, u, v;
142     std::vector<int> d;
143
144     // Initialize distances
145     d = initialize_single_source(graph, src, d);
146
147     // Relax edges
148     for(i = 1; i < (*graph).V - 1; i++){
149         for(j = 0; j < (*graph).E; j++){
150             u = (*graph).edge[j].src;
151             v = (*graph).edge[j].dst;
152
153             d = relax(d, u, v, (*graph).edge[j].weight);
154         }
155     }
156
157     // Check for negative-weight cycles
158     for(i = 0; i < (*graph).V; i++){
159         u = (*graph).edge[i].src;
160         v = (*graph).edge[i].dst;
161
162         if(d[u] != std::numeric_limits<int>::max() && d[u] + (*graph).edge[i].weight <
163             d[v]){
164             return false;
165         }
166     }
167
168     // Print result
169     print_result(d, (*graph).V);
170
171     // Return true
172     return true;
173 }
174
175 /*      Main      */
176 int main(){
177     // Local variables
178     int src = 0;
179     int V = 5;
180     int E = 10;
181
182     // Create graph
183     Graph graph = createGraph(V, E);
184
185     // Add edges and nodes to graph
186     // Example from book
187     addEdge(graph, 0, 0, 1, 6);
188     addEdge(graph, 1, 0, 2, 7);
189     addEdge(graph, 2, 4, 0, 2);
190     addEdge(graph, 3, 1, 2, 8);
191     addEdge(graph, 4, 1, 3, 5);
192     addEdge(graph, 5, 3, 1, -2);
193     addEdge(graph, 6, 1, 4, -4);
194     addEdge(graph, 7, 2, 3, -3);
195     addEdge(graph, 8, 2, 4, 9);
196     addEdge(graph, 9, 4, 3, 7);
197
198     // Print graph as an adjacency list
199     printAdjList(graph);
200
201     // Solve single-source shortest-paths problems
202     bellman_ford(graph, src);
203
204     // Free memory
205     destroy(graph);
206
207     return 0;

```

