Homework 4 (5pt.)

Submission instruction:

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

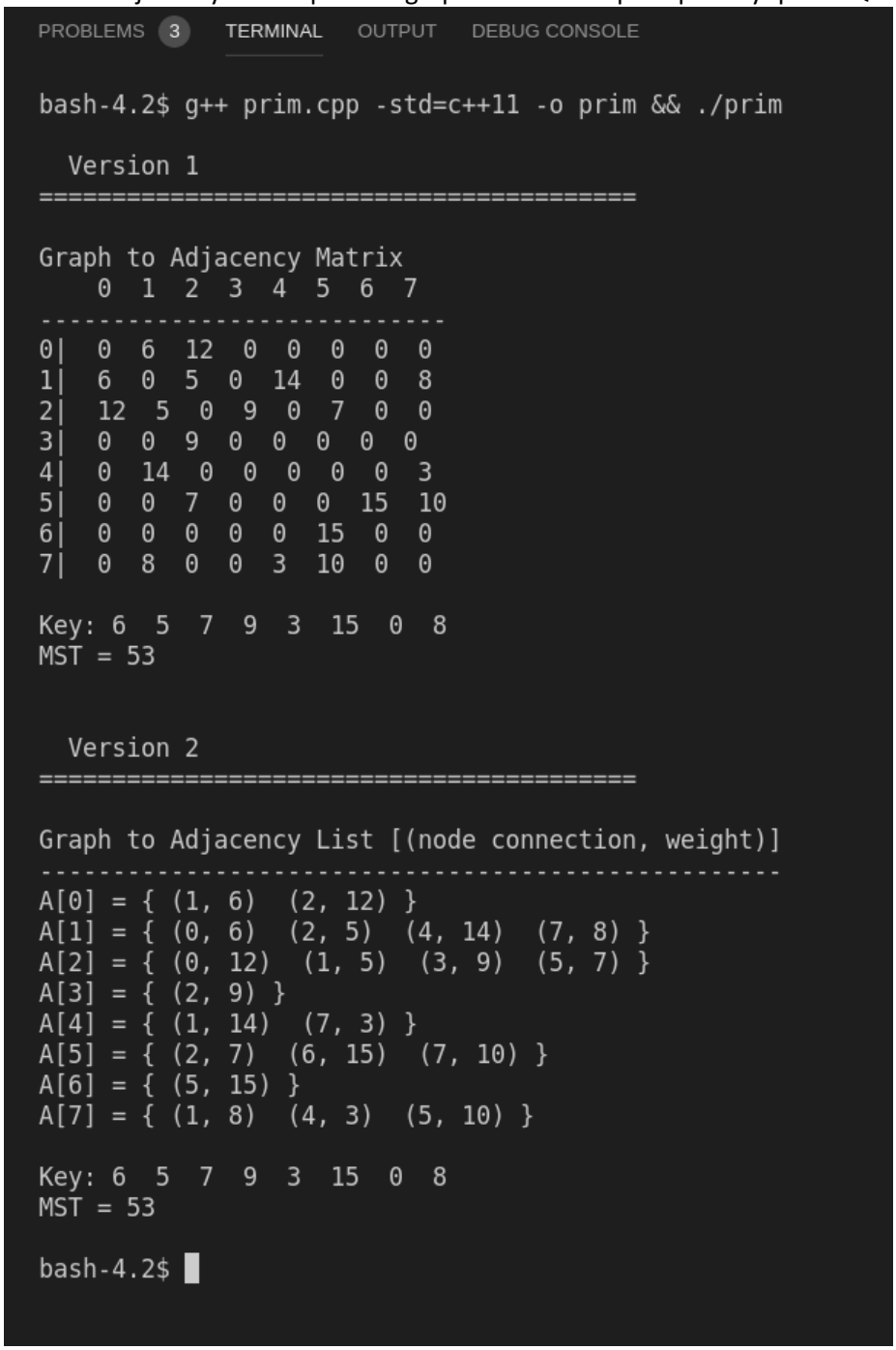
For coding problems, copy and paste your codes. Report your results.

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

Question 1 (5 pt.) MST: Write codes for Prim's algorithm.

Version 1: use adjacency matrix to present graph and use unsorted array for priority queue Q.

Version 2: use adjacency lists to present graph and use heap for priority queue Q.



```
1   //  Prim's Algorithm        //
2   //  19 November 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 <algorithm>
11  #include <iterator>
12
13  //
    Macros
    //
14  #define V 8
15
16  //              Add point from graph to each graph
    representation                                          //
17  void add_point(int matrix[V][V], std::vector<std::pair<int, int>> list[], int node, int
    connection, int weight){
18      //      Add Points to Matrix                //
19      matrix[node][connection] = weight;
20      matrix[connection][node] = weight;
21
22      //      Add Points to List                  //
23      list[node].push_back(std::make_pair(connection, weight));
24      list[connection].push_back(std::make_pair(node, weight));
25
26      return;
27  }
28
29  //              Find Minimum to
    Extract                                                                 //
30  int extract_min(std::vector<int> Q, std::vector<int> key, std::vector<int> trash){
31      //      Local Variables                     //
32      int prev, next, index, extract_min_index;
33
34      //      Initialize Variables                //
35      prev = INT_MAX;
36      extract_min_index = 0;
37
38      //      Iterate through list                //
39      for(index = 0; index < V; index++){
40          next = key[index];
41
42          //      Compare minimum with next value     //
43          if(next < prev){
44              if(std::find(std::begin(trash), std::end(trash), index) == std::end(trash)){
45                  prev = next;
46                  extract_min_index = index;
47              }
48          }
49      }
50
51      return extract_min_index;
52  }
53
54  //              Determines if a Value is contained within Priority
    Queue                                   //
55  bool locate(std::vector<std::pair<int, int>> Q, std::pair<int, int> i){
56      //      Compare node values of Q and Graph  //
57      for(auto j : Q){
58          if(j.first == i.first){
59              return true;
60          }
61      }
```

```
62
63      return false;
64  }
65
66  //              Determines Index of Node Value for
    Q                                                   //
67  int locate_index(std::vector<std::pair<int, int>> Q, std::pair<int, int> i){
68      //      Local Variables                         //
69      int index = 0;
70
71      //      Compares Vertices                       //
72      for(auto j : Q){
73          if(j.first == i.first){
74              return index;
75          }
76
77          index++;
78      }
79
80      return index;
81  }
82
83  //              Returns Left Node
    Value                                                           //
84  int left(int i){
85      return 2 * i + 1;
86  }
87
88  //              Returns Right Node
    Value                                                         //
89  int right(int i){
90      return 2 * i + 2;
91  }
92
93  //              Corrects Single
    Instance                                                        //
94  std::vector<std::pair<int, int>> heapify(std::vector<std::pair<int, int>> A, int i, int
    heapsize){
95      //      Local Variables                         //
96      int l = left(i);
97      int r = right(i);
98      int largest = i;
99
100     //      Determines Largest Value                //
101     if(l < heapsize && A[l].second > A[largest].second){
102         largest = l;
103     }
104     if(r < heapsize && A[r].second > A[largest].second){
105         largest = r;
106     }
107
108     //      Swap Values                             //
109     if(largest != i){
110         std::swap(A[i], A[largest]);
111         A = heapify(A, largest, heapsize);
112     }
113
114     return A;
115 }
116
117 //              Sorts
    Heap                                                                    //
118 std::vector<std::pair<int, int>> heapsort(std::vector<std::pair<int, int>> A, int n){
119     //      Builds Heap                             //
120     for(int i = n / 2 - 1; i >= 0; i--){
121         A = heapify(A, i, n);
122     }
123
124     //      Sorts Heap                              //
```

```
125         for(int i = n - 1; i >= 0; i--){
126             std::swap(A[0], A[i]);
127             A = heapify(A, 0, i);
128         }
129 
130         return A;
131     }
132 
133     //              Version 1:                                                                  //
134     // Utilizing adjacency matrix to present graph and unsorted array for priority queue      //
135     void prim_alg_v1(int graph[V][V]){
136         //      Local Variables                     //
137         int u, v, s, index;
138         std::vector<int> key;
139         std::vector<int> pi[V];
140         std::vector<int> Q;
141         std::vector<int> trash;
142         std::vector<int>::iterator it;
143 
144         //      Print Adjacency Matrix              //
145         std::cout << "Graph to Adjacency Matrix" << std::endl;
146         std::cout << "    ";
147         for(u = 0; u < V; u++) {
148             std::cout << u << "  ";
149         }
150         std::cout << std::endl;
151         std::cout << "---------------------------" << std::endl;
152         for(u = 0; u < V; u++) {
153             std::cout << u << "|  ";
154             for(v = 0; v < V; v++) {
155                 std::cout << graph[u][v] << "  ";
156             }
157             std::cout << std::endl;
158         }
159 
160         //      Initialize key and Q                //
161         for(u = 0; u < V; u++){
162             key.push_back(INT_MAX);
163             Q.push_back(u);
164         }
165 
166         //      Randomly Select Node                //
167         srand(time(NULL));
168         s = rand() % (Q.size() - 1) + 0;
169         key[s] = 0;
170 
171         //      While priority queue exists         //
172         while(Q.size() != 0){
173             //      Extract min from Q                  //
174             u = extract_min(Q, key, trash);
175             trash.push_back(u);
176 
177             it = std::find(Q.begin(), Q.end(), u);
178             index = std::distance(Q.begin(), it);
179             Q.erase(Q.begin() + index, Q.begin() + (index + 1));
180 
181             //      Compare and replace key values      //
182             for(v = 0; v < V; v++){
183                 if(graph[u][v] > 0 && std::find(std::begin(Q), std::end(Q), v) !=
                    std::end(Q) && graph[u][v] < key[v]){
184                     key[v] = graph[u][v];
185                     pi[v].push_back(u);
186                 }
187             }
188         }
189 
190         //      Calculate MST                       //
```

```
191         int summation = 0;
192         std::cout << std::endl << "Key: ";
193         for(auto i : key ){
194             std::cout << i << "  ";
195             summation += i;
196         }
197         std::cout << std::endl << "MST = " << summation << std::endl;
198 
199         return;
200     }
201 
202     //              Version
        2:                                                                  //
203     // Utilizng adjacency lists to present graph and heap for priority
        queue                                //
204     void prim_alg_v2(std::vector<std::pair<int, int>> graph[]){
205         //  Local Variables                         //
206         int u, v, s, first_key, first_node, Q_index;
207         std::vector<int> key;
208         std::vector<int> pi[V];
209         std::vector<std::pair<int, int>> Q;
210 
211         //  Print Adjacency List                    //
212         std::cout << "Graph to Adjacency List [(node connection, weight)]" << std::endl;
213         std::cout << "-------------------------------------------------" << std::endl;
214         for (u = 0; u < V; u++) {
215             std::cout << "A[" << u << "] = {";
216             for (v = 0; v < graph[u].size(); v++){
217                 std::cout << " (" << graph[u][v].first << ", " << graph[u][v].second << ")
                    ";
218             }
219             std::cout << "}" << std::endl;
220         }
221 
222         //      Initialize key and Q                //
223         for(u = 0; u < V; u++){
224             key.push_back(INT_MAX);
225             Q.push_back(std::make_pair(u, key[u]));
226         }
227 
228         //      Randomly Select Node                //
229         srand(time(NULL));
230         s = rand() % (Q.size() - 1) + 0;
231         key[s] = 0;
232         Q[s].second = 0;
233 
234         //      While priority queue exists         //
235         while( Q.size() != 0){
236             //      Sort Heap                               //
237             Q = heapsort(Q, Q.size());
238 
239             //      Extract min from Q                      //
240             first_node = Q[0].first;
241             first_key = Q[0].second;
242             Q.erase(Q.begin(), Q.begin() + 1);
243 
244             //      Compare and replace key values      //
245             for(auto i : graph[first_node]){
246                 if( locate(Q, i) && i.second < key[i.first]){
247                     key[i.first] = i.second;
248                     pi[i.first].push_back(i.first);
249 
250                     Q_index = locate_index(Q, i);
251                     Q[Q_index].second = i.second;
252                 }
253             }
254         }
255 
256         //      Calculate MST                       //
```

```
257         int summation = 0;
258         std::cout << std::endl << "Key: ";
259         for(auto i : key ){
260             std::cout << i << "  ";
261             summation += i;
262         }
263         std::cout << std::endl << "MST = " << summation << std::endl;
264 
265         return;
266     }
267 
268     //              Main
        function                                                                    //
269     int main(){
270         //  Local Variables                     //
271         int i, j;
272         int matrix[V][V];
273         std::vector<std::pair<int, int>> list[V];
274 
275         //  Initialize Matrix                   //
276         for(i = 0; i < V; i++){
277             for(j = 0; j < V; j++){
278                 matrix[i][j] = 0;
279             }
280         }
281 
282         //  Add Points to Graph                 //
283         add_point(matrix, list,  0, 1, 6);
284         add_point(matrix, list,  0, 2, 12);
285         add_point(matrix, list,  1, 2, 5);
286         add_point(matrix, list,  2, 3, 9);
287         add_point(matrix, list,  1, 4, 14);
288         add_point(matrix, list,  1, 7, 8);
289         add_point(matrix, list,  2, 5, 7);
290         add_point(matrix, list,  5, 6, 15);
291         add_point(matrix, list,  4, 7, 3);
292         add_point(matrix, list,  5, 7, 10);
293 
294         //  Version 1                           //
295         std::cout << std::endl;
296         std::cout << "  Version 1" << std::endl;
297         std::cout << "========================================" << std::endl;
298         std::cout << std::endl;
299         prim_alg_v1(matrix);
300 
301         //  Version 2                           //
302         std::cout << std::endl;
303         std::cout << std::endl;
304         std::cout << "  Version 2" << std::endl;
305         std::cout << "========================================" << std::endl;
306         std::cout << std::endl;
307         prim_alg_v2(list);
308 
309         std::cout << std::endl;
310 
311         return 0;
312     }
```