Nama : Andyan Yogawardhana

NIM : 21/482180/PA/21030

Kelas : KOMB1

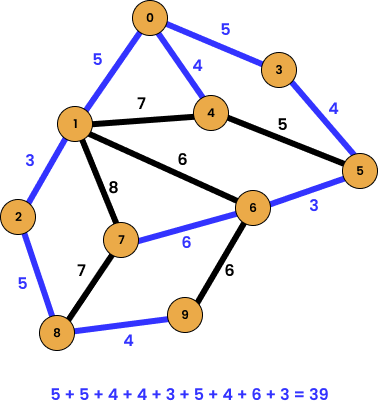
Tugas 8 – Prim’s Algorithm

Source Code

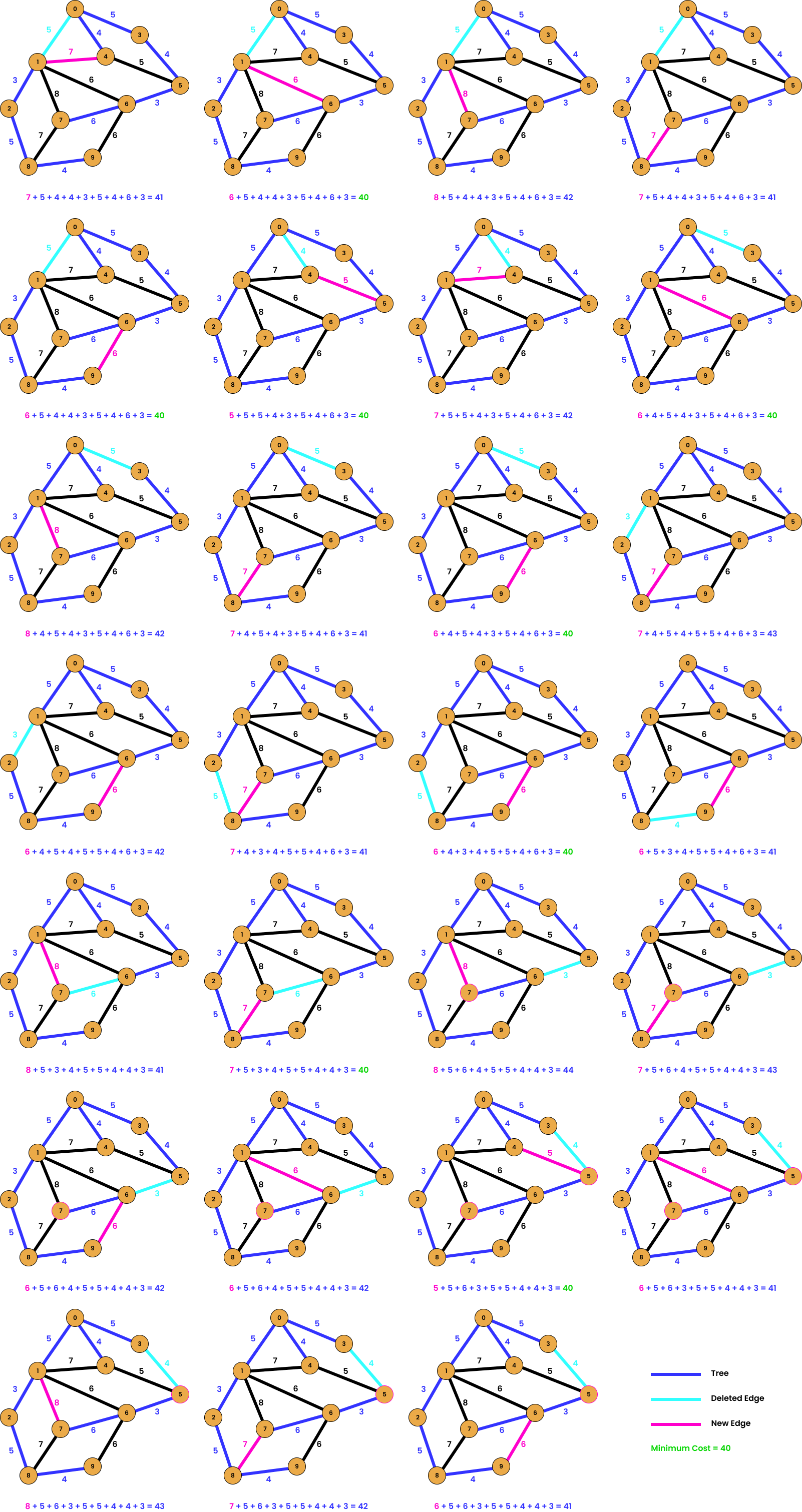
1. import java.util.Stack;
2. public class Prim {
3. public static void main(String[] args) {
4. *// create graph (graph 8.2.2)*
5. MST t = new MST();
6. int graph[][] = new int[][]{{0,5,0,5,4,0,0,0,0,0},
7. {5,0,3,0,7,0,6,8,0,0},
8. {0,3,0,0,0,0,0,0,5,0},
9. {5,0,0,0,0,4,0,0,0,0},
10. {4,7,0,0,0,5,0,0,0,0},
11. {0,0,0,4,5,0,3,0,0,0},
12. {0,6,0,0,0,3,0,6,0,6},
13. {0,8,0,0,0,0,6,0,7,0},
14. {0,0,5,0,0,0,0,7,0,4},
15. {0,0,0,0,0,0,6,0,4,0}};
16. *// print the solution*
17. t.primMST(graph);
18. t.spanTree(graph);
19. }
20. }
21. class MST {
22. *// number of vertices in the graph*
23. private static final int V = 10;
24. *// total weight on a mst*
25. private int totalWeight = 0;
26. *// array to store constructed MST*
27. int[] parent = new int[V];
28. *// two dimensional array to store adjacency value*
29. private boolean[][] isConnected = new boolean[V][V];
31. *// MST class constructor*
32. public MST() {
33. for(int i = 0; i < V; i++) {
34. for(int j = 0; j < V; j++) {
35. *// set the adjacency of i-th and j-th vertices to false*
36. isConnected[i][j] = false;
37. }
38. }
39. }
40. *// a utility function to find the vertex with minimum key value, from the set of vertices not yet included in MST*
41. public int minKey(int[] key, boolean[] mstSet) {
42. *// initialize min value*
43. int min = Integer.MAX\_VALUE, minIndex = -1;
44. for(int v = 0; v < V; v++) {
45. if(mstSet[v] == false && key[v] < min) {
46. min = key[v];
47. minIndex = v;
48. }
49. }
50. return minIndex;
51. }
52. *// a utility function to print the constructed MST stored in parent[]*
53. private void printMST(int[] parent, int[][] graph) {
54. System.out.println("---- Graph 8.2.2 ----\n");
55. System.out.println("Edge \tWeight");
56. for(int i = 1; i < V; i++) {
57. System.out.println(parent[i] + " - " + i + "\t  " + graph[i][parent[i]]);
58. }
59. System.out.println("\nTotal MST Cost = " + totalWeight);
60. }
62. *// a function to construct and print MST for a graph represented using adjacency matrix representation*
63. public void primMST(int[][] graph) {
64. *// key values used to pick minimum weight edge in cut*
65. int[] key = new int[V];
66. *// to represent set of vertices included in MST*
67. boolean[] mstSet = new boolean[V];
68. *// initialize all keys as infinite*
69. for(int i = 0; i < V; i++) {
70. key[i] = Integer.MAX\_VALUE;
71. mstSet[i] = false;
72. }
73. *// always include first 1st vertex in mst*
74. key[0] = 0; *// make key 0 so that this vertex is picked as first vertex*
75. parent[0] = -1; *// first node is always root of MST*
76. *// the MST will have V vertices*
77. for(int count = 0; count < V; count++) {
78. *// pick the minimum key vertex from the set of vertices not yet included in MST*
79. int u = minKey(key, mstSet);
80. *// add the picked vertex to the MST set*
81. mstSet[u] = true;
82. *// update key value and parent index of the adjacent vertices of the picked vertex*
83. *// consider only those vertices which are not yet included in MST*
84. for(int v = 0; v < V; v++) {
85. *// graph[u][v] is nonzero only for adjacent vertices of m*
86. *// mstSet[v] is false for vertices not yet included in MST*
87. *// update the key only if graph[u][v] is smaller than key[v]*
88. if(graph[u][v] != 0 && mstSet[v] == false && graph[u][v] < key[v]) {
89. parent[v] = u;
90. key[v] = graph[u][v];
91. }
92. }
93. }
94. *// count the total weight using loop*
95. for(int i = 1; i < V; i++) {
96. totalWeight += graph[i][parent[i]];
97. }
98. *// print the constructed MST*
99. printMST(parent, graph);
100. }
101. *// a function to count the possible spanning trees and the minimum tree costs among all of them*
102. public void spanTree(int[][] graph) {
103. *// to store number of possible trees*
104. int trees = 0;
105. *// to store cost of minimum tree and set its initial value to infinity*
106. int cost = Integer.MAX\_VALUE;
107. *// to store temporary value of cost that will be used to compare the minimal cost from trees*
108. int costTemp = 0;
109. *// connect the i-th vertex to its parent*
110. for(int i = 1; i < V; i++) {
111. isConnected[i][parent[i]] = true;
112. isConnected[parent[i]][i] = true;
113. }
115. for(int j = 1; j < V; j++) {
116. *// set the j-th vertex and its parent adjacency to false*
117. isConnected[j][parent[j]] = false;
118. isConnected[parent[j]][j] = false;
119. for(int k = 0; k < V; k++) {
120. for(int l = k + 1; l < V; l++) {
121. *// skip the iteration if the index of k equals to the value of current vertex*
122. if(k == parent[j] && k == j) {
123. continue;
124. }
125. *// check if k-th and l-th vertices are not connected but it has an edge value*
126. if(isConnected[k][l] == false && graph[k][l] != 0) {
127. *// set the adjacency of those vertices to true (connected)*
128. isConnected[k][l] = true;
129. isConnected[l][k] = true;
130. *// check if the graph is a valid tree or not*
131. if(isValidTree(isConnected)) {
132. for(int m = 0; m < V; m++) {
133. for(int n = m + 1; n < V; n++) {
134. if(isConnected[m][n] == true) {
135. *// add the temporary cost by the value of m-n edge value*
136. costTemp += graph[m][n];
137. }
138. }
139. }
140. *// check if the temporary cost value is between the total weight and current cost*
141. if(costTemp > totalWeight && costTemp < cost) {
142. *// set the cost to current temporary cost*
143. cost = costTemp;
144. }
145. *// check if the temporary cost value is more than the total weight*
146. if(costTemp > totalWeight) {
147. *// add the total possible spanning tree*
148. trees++;
149. }
150. *// reset the temporary cost*
151. costTemp = 0;
152. }
153. *// reset the vertex adjacency*
154. isConnected[k][l] = false;
155. isConnected[l][k] = false;
156. }
157. }
158. }
159. *// reset the vertex and its parent adjacency*
160. isConnected[j][parent[j]] = true;
161. isConnected[parent[j]][j] = true;
162. }
163. *// print the result*
164. System.out.println("Spanning Trees = " + trees);
165. System.out.println("Minimum Cost = " + cost);
166. }
167. *// function to check whether the graph is a valid tree or not (using dfs method)*
168. private boolean isValidTree(boolean[][] graph) {
169. boolean visit[] = new boolean[V];
170. Stack<Integer> s = new Stack<Integer>();
171. int start = 0;
172. *// variable to count the total number of visited vertices*
173. int vertices = 1;
174. visit[start] = true;
175. s.push(start);
176. while(!s.isEmpty()){
177. start = s.pop();
178. for(int i = 1; i  < V; i++){
179. if(graph[start][i] && !visit[i]){
180. visit[i] = true;
181. s.push(i);
182. *// increase the vertices count after a vertex is being visited*
183. vertices++;
184. }
185. }
186. }
187. *// if the total counted vertices equals to the graph's initial vertices, then it is a valid tree*
188. if(vertices == V) {
189. return true;
190. }
191. *// else, it is not a valid tree*
192. return false;
193. }
194. }

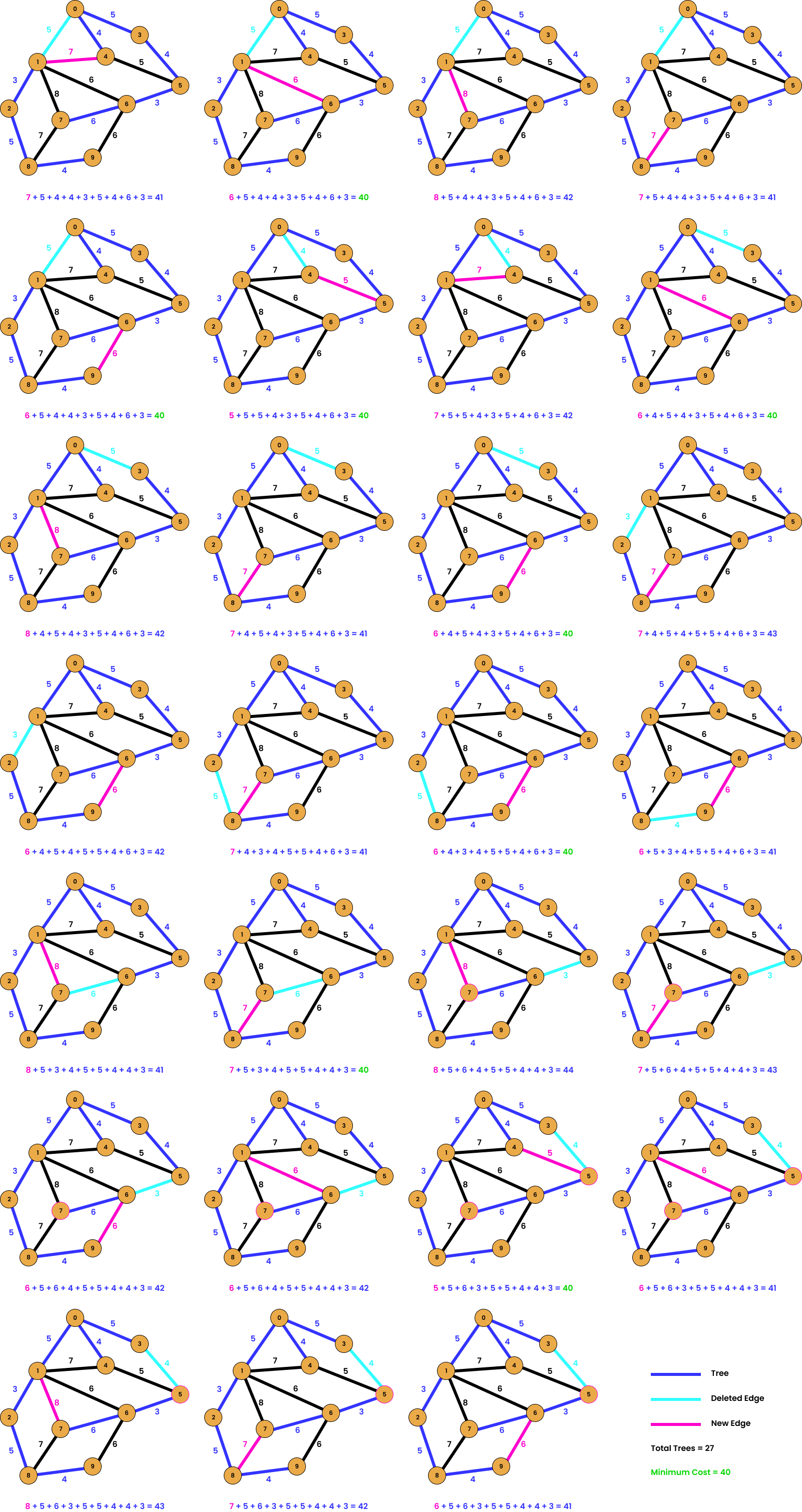
Manual Calculation

1. MST cost



2. Spanning Tree





Output Terminal

