# Lab Assignment 5 (L5)

**Done By**

**Name** – Md Junaid Mahmood

**Enrolment Number** – 19116040

**Department** – Computer Science & Engineering

**Year** – IInd Year

# Problem Statement 1

**Write a C++ program to implement a graph using adjacency list (linked list) without using STL. Perform following operations on the graph after creating the graph based on the edge list given in the input file.**

1. **BFS traversal**
2. **DFS traversal**
3. **Cycle finding in the graph**
4. **Calculate diameter of the graph**

First of all, as specified in the question adjacency list was used for storing the graph. The adjacency list was implemented using linked list data structure. Details of data structure and algorithm for other functionalities are given below –

1. To count the number of nodes in the graph, list was traversed once and total number of unique element was counted.

   Thus, time complexity is O(n) where n is the number of edges and space complexity is O(k) where k is a constant.

2. To implement the breadth first search in the graph, queue data structure (using dynamic array) was used. Apart from this, some variables and one array (whose size is allocated

dynamically using number of nodes in the graph) were used. The purpose of the array is to store whether a node has been traversed or not.

Pseudo code for the breadth first search algorithm is –

i. Start from node 1
ii. front = 0 and rear = 0 // They are variables for queue
iii. queue[front] = node 1
iv. front = front + 1
v. while(real < front) {
vi. add all the node adjacent to node queue[rear] which are not yet traversed to the queue at the front
vii. rear = rear + 1
viii. } // end of the loop
ix. Print the output of the queue

Thus, time complexity is O(v + e) and space required is O(v). Here, v is number of vertices and e is number of edges.

3. To implement the depth first search in the graph, stack data structure (using dynamic array) was used. Apart from this, some variables and two arrays (whose size is allocated dynamically using number of nodes in the graph) were used. The purpose of first array is to store whether a node has been

traversed or not and that of second array is to store the final output of the depth first search.

Pseudo code for the breadth first search algorithm is –

i. Start from node 1
ii. top = 0 // it is variable for stack
iii. stack[top] = node 1
iv. top = top + 1
v. while(top > 0) {
vi. check if there is a node adjacent to node at queue[top - 1] which is not yet traversed
vii. if yes add it to the stack and continue
viii. if not pop the top element from the stack and decrease the value of top by 1
ix. } // end of the loop
x. Print the output of the stack

Thus, time complexity is O(v + e) and space required is O(v). Here, v is number of vertices and e is number of edges.

4. To check the cycle depth first search algorithm was used with some modification.

Pseudo code for the cycle check algorithm is –

i. Start the algorithm for Depth First Search algorithm from node 1
ii. Before pushing a node check whether the new coming node has been already traversed or not
iii. If yes check whether the traversed node is adjacent to the node which is at the top of the stack
iv. If yes then continue with the algorithm
v. Else cycle is found and end the program

Since the algorithm is basically using depth first search, hence, time complexity is O(v + e) and space required is O(v). Here, v is number of vertices and e is number of edges.

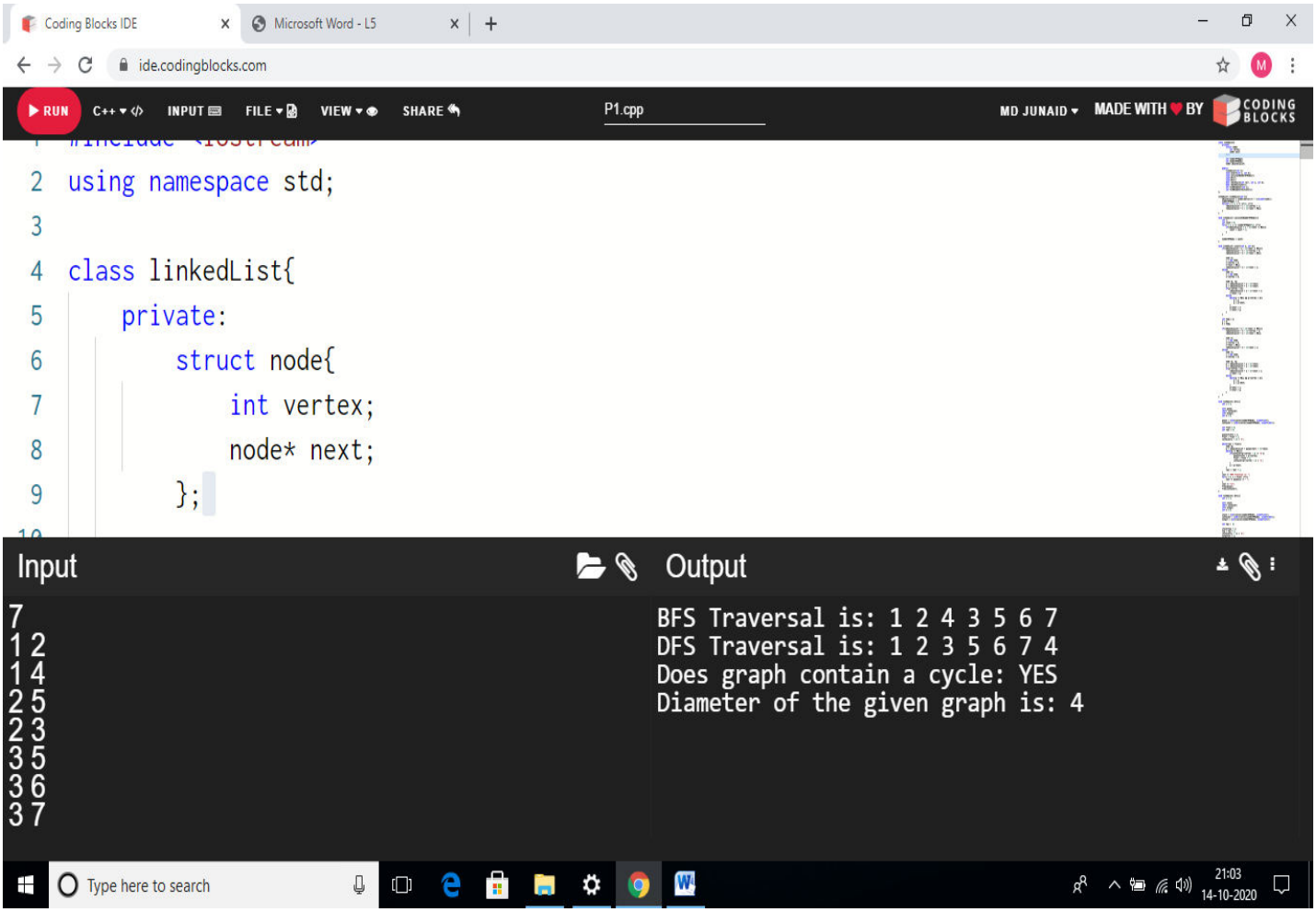
5. To find the diameter of the graph, breadth first search algorithm was used with a slight modification.

Pseudo code for the diameter finding algorithm is –

i. n = 1 // n is a variable of type int
ii. diameter = 0
iii. while(n <= total nodes) {
iv. do the breadth first search starting from node n
v. store the length of path (iteration at which particular node came) to come to a particular node starting from node n
vi. find the maximum path among all the paths

vii. if( maximum path > diameter ){
viii. diameter = maximum path
ix. } // end of the loop
x. n = n + 1
xi. } // end of the loop
xii. print diameter

Since the algorithm is basically using breadth first search from each node one at a time, hence, time complexity is O(v * e) and space required is O(v). Here, v is number of vertices and e is number of edges.




Following screenshot was taken in Coding Blocks IDE due to better view for showing input and output. The code ran perfectly well on Codechef IDE as well.

# Problem Statement 2

**Given a set of nodes connected to each other in the form of a weighted undirected graph G, find the minimum spanning tree (MST). A spanning tree T of an undirected graph G is a subgraph that is a tree which includes all of the vertices of G, with minimum possible number of edges. G may have more than one spanning trees. The weight of a spanning tree is the sum of weights given to each edge of the spanning tree. A minimum spanning tree (MST) is a spanning tree whose weight is less than or equal to that of every other spanning tree.**

**For given input graph (given as a text file having the format as shown in the example below), implement Kruskal's algorithm in C++ program using UNION-FIND data structures (without using STL) and show all the edges of the MST as output in both the command line and in the "dot file", where DOT is a graph description language. Also, print the total edge weight of the MST. For more details follow this link https://www.graphviz.org/doc/info/lang.html. Further use the "dot file" file to visualize the output graph in .pdf or .png file using Graphviz.**

**Use the online Graphviz for visualization of DOT file: https://edotor.net/**

First of all, linked list data structure was used for storing the information about the graph. New nodes in the list were added in the sorted manner based on the weight of the edge. Details of data structure and algorithm for other functionalities are given below –

1. To count the number of nodes in the graph, list was traversed once and total number of unique element was counted.

   Thus, time complexity is O(n) where n is the number of edges and space complexity is O(k) where k is a constant.

2. For implementing Kruskal's algorithm, union-find data structure was used. Union-find data structure was implemented using an array with the following convention
   a) 0 means particular node is not yet traversed
   b) A negative number means that a particular node is the root element. Root element means that it is the topmost element (that is, it does not have any parent further). Magnitude of the number shows its number of child nodes.
   c) A positive number shows that the given node is the child node. The number represents the immediate parent of the node.
   d) In this convention array index starts from 1

Pseudo code for the Kruskal's algorithm is –

```
i.    node k // k is of type node of a linked list
ii.   k = head
iii.  int arr[] // for maintaining union-find data structure
iv.   /* initially array arr does not contain any non-zero value
      which signifies that the set is empty. This set is basically
      the set of edges of the minimum spanning tree */
v.    while(k != null) {
vi.       /* check whether given edge can be added into the
          set or not */
vii.      if(find(k, arr)) {
viii.           /* add the given edge to the set */
ix.             union(k, arr)
x.        }
xi.       k = k->next
xii.  } // end of the loop
```

Pseudo code for the find(k, arr) functionality is –

```
i.    // here we have node k and array arr as arguments
ii.   v1 = k->vertex1
iii.  v2 = k->vertex2
iv.   r1 = root_Element(v1, arr)
v.    r2 = root_Element(v2, arr)
vi.   if ( r1 == r2 )
```

```
vii.        return False
viii.   Else
ix.         return True
```

Pseudo code for the root_Element(v, arr) functionality is –

```
i.   // here we have vertex v and array arr as arguments
ii.  If ( arr[v] <= 0 )
iii.        return v
iv.  Else
v.          int a = arr[v]
vi.         int b = root_Element(a, arr)
vii.        return b
```

Pseudo code for the union(k, arr) functionality is –

```
i.   // here we have node k and array arr as arguments
ii.  v1 = k->vertex1
iii. v2 = k->vertex2
iv.  r1 = root_Element(v1, arr)
v.   r2 = root_Element(v2, arr)
vi.  if( r1 == 0 and r2 == 0 ) {
vii.       arr[v1] = -2
viii.      arr[v2] = v1
ix.  }else if(r1 == 0) {
```

```
x.          arr[v1] = r2
xi.         arr[r2] = arr[r2] – 1;
xii.    }else if(r2 == 0) {
xiii.       arr[v2] = r1
xiv.        arr[r1] = arr[r1] - 1
xv.     }else{
xvi.        If(r1 < r2){
xvii.               arr[v2] = r1
xviii.              arr[r1] = arr[r1] + r2
xix.        }else {
xx.                 arr[v1] = r2
xxi.                arr[r2] = arr[r2] + r1
xxii.       }
xxiii.  }
```

Now time complexity of each union operation, find operation and root_Element is O(v), where v is the total number of vertex in the graph.

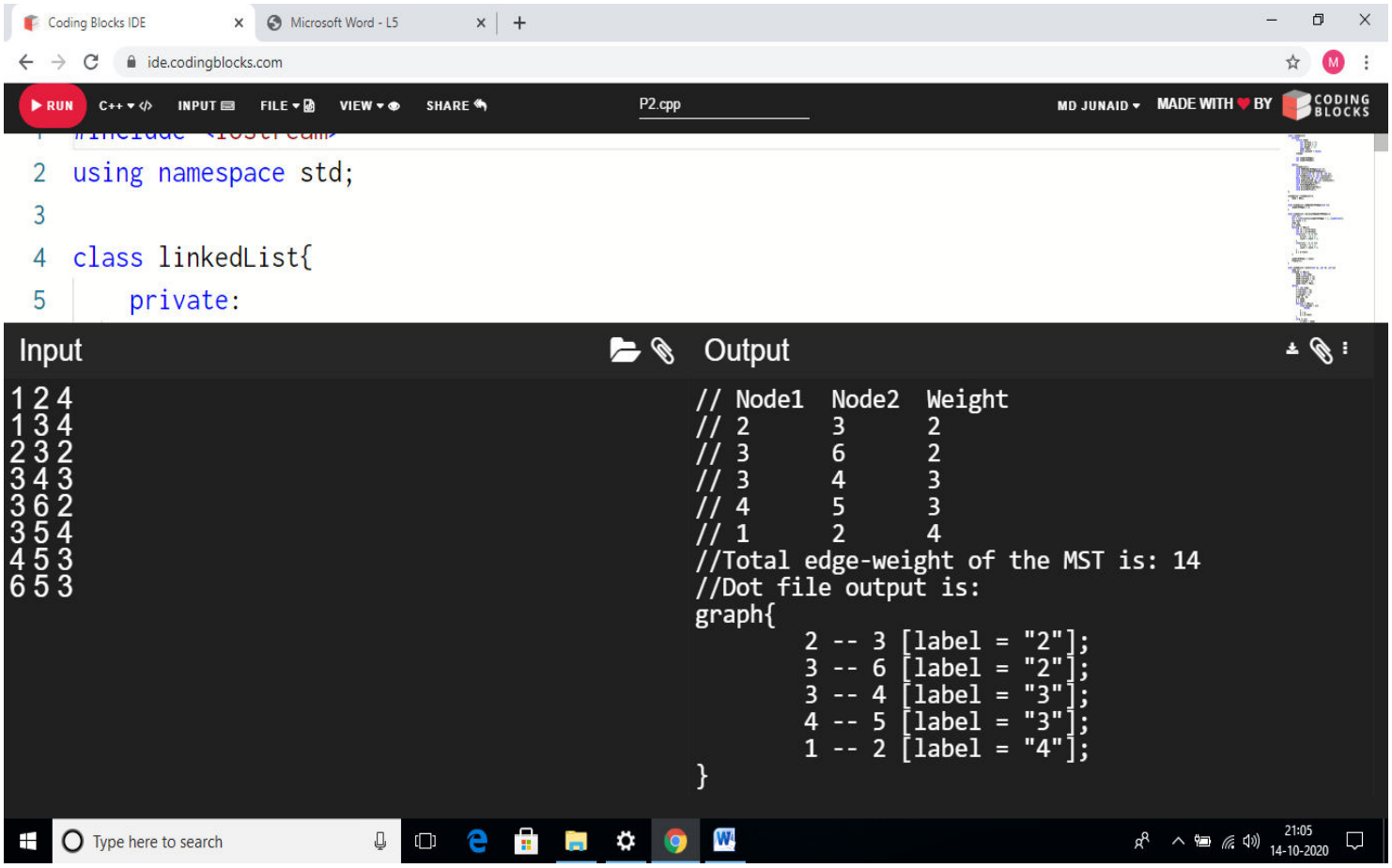
Now in Kruskal's algorithm we are doing above operations at most e times, where e is the total number of edges. Thus, time complexity for Kruskal's algorithm is O(v * e), where v is total number of vertices and e is the total number of edges. However, above complexity is for the worst case. For an

average case, the above mentioned complexity gets reduced to O(log e * e).

Now space complexity for above problem is O(v), where v is the total number of vertices of the graph.

3. To print the edges in the minimum spanning tree and the dot file, we just have to traverse through the linked list once. In the linked list, the information about which edge is a part of the minimum spanning tree is stored.

   Thus, time complexity is O(n) where n is the number of edges and space complexity is O(k) where k is a constant.




Following screenshot was taken in Coding Blocks IDE due to better view for showing input and output. The code ran perfectly well on Codechef IDE as well.