Algorithm 1 Breadth-first Search with Coloring for clarity. This implementation uses an adjacency list for the graph data structure.

```
function BFS(G(V, E), sourceNode)                ▷ G has vertices V, edges E
    Tree ← ∅
    Dist ← ∅
    Color ← ∅
    Parent ← ∅
    Init(Tree, Dist, Parent, Color)

    Q ← ∅
    Color[sourceNode] ← Grey
    Parent[sourceNode] ← NIL
    Dist[sourceNode] ← 0

    Enq(Q, sourceNode)
    while NotEmpty(Q) do
        u ← Deq(Q)
        for v in Adj(u) do
            if Color[v] equals White then
                Visit(v)
                Color[v] ← Grey
                Parent[v] ← u
                Dist[v] ← Dist[u] + 1
                Tree ← Tree ⋃ {(u, v)}
                Enq(Q, v)
            else
                <Do processing logic here if desired>
            end if
        end for
        Color[u] ← Black
    end while

    return Tree
end function
```

Algorithm 2 Depth-first Search using color for clarity.

```
function DFS(G(V, E), sourceNode)
    Start ← ∅
    Finish ← ∅
    Parent ← ∅
    Color ← ∅
    DfsInit(G(V, E), Start, Finish, Parent, Color)

    Start[sourceNode] ← 0
    Parent[sourceNode] ← Nil
    time ← 0
    for  vertex in V do
        if Color[vertex] equals White then
            DfsVisit(vertex, time, Start, Finish, Parent, Color)
        end if
    end for
end function
```

Algorithm 3 The visit function associated with DFS.

```
function DfsVisit(u, time, Start, Finish, Parent, Color)
    Color[u] ← Grey
    time ← time + 1
    Start[u] ← time
    for v in Adj(u) do
        if Color[v] equals White then
            Parent[v] ← u
            DfsVisit(v, time, Start, Finish, Parent, Color)
        end if
    end for
    Color[u] ← Black
    Finish[u] ← time ← time + 1
end function
```

Algorithm 4 DFS Initialization function.

```
function DfsInit(G(V, E), Start, Finish, Parent, Color)
    for vertex in V do
        Color[vertex] ← White
        Start[vertex] ← −1
        Finish[vertex] ← −1
        Parent[vertex] ← Nil
    end for
end function
```

Algorithm 5 Kruskal's Algorithm for locating the Minimum Spanning Tree.

```
function GetMST(G(V, E))                          ▷ Kruskal's Algorithm
    Label ← ∅
    Labelize(Label, V)
    A ← ∅
    Sort(E, order = nondecreasing)
    for (u, v) in E do
        if not IntroducesCycle(A, u, v) then
            A ← A ⋃{edge}
            MergeLabels(u, v)   ▷ Must merge trees, not just ind. vertices
        end if
    end for
    return G(V, A)
end function
```

Algorithm 6 Function that determines if a given edge added to a set of edges will produce a cycle.

```
function IntroducesCycle(Label, u, v)
    if Label[u] equals Label[v] then
        return True
    else
        return False
    end if
end function
```

Algorithm 7 Prim's Algorithm for finding the Minimum Spanning Tree.

```
function FindMST(G(V, E))
end function
```