```
class SampleVertex
{
	// Main list of vertices
	string Text;
	// Indicates if vertex is visited during search
	bool Active;
	// List of all Vertices that this Vertex points to
	List<SampleVertex> AdjecencyList;
}
```

Each vertex in graph is an instance of SampleVertex Class. The class Construcor only takes in a string as a name, other properties are altered by functions. In the main Class we keep the following:

```
// Main list of vertices
List<SampleVertex> verticesList = new List<SampleVertex>();
// Name of graph
string graphName = "";
// Keeps track of all graph names from DataBase
ObservableCollection<string> graphNames = new ObservableCollection<string>();
// Flag that tells you if DFS search is running
bool dfsOn = false;
// Flag that tells you if BFS search is running
bool bfsOn = false;
```

Vertices are added by assigning them to edges, that way no idle vertices are created. Function for adding new Vertices is the following one:

```
private void AddEdge(SampleVertex from, SampleVertex to)
        {
             // Keeps track if the 'from' Vertex exists
            bool fromFlag = true;
             // Keeps track if the 'to' Vertex exists
            bool toFlag = true;
             // Keeps track if the new Edge exists
            bool edgeFlag = true;

             // Checks if a vertex already exists in our List
             // if it does the 'from'/'to' vertex is equated to it
            foreach (SampleVertex v in verticesList)
            {
                if (v.Text == from.Text)
                {
                    from = v;
                    fromFlag = false;
                }

                if (v.Text == to.Text)
                {
                    to = v;
                    toFlag = false;
                }
```

```
    }


     // In case that the passed in 'from' vertex is a new one
     // it is added to our List of Vertices
     if (fromFlag)
    {
        verticesList.Add(from);
    }

     // In case that the passed in 'to' vertex is a new one
     // it is added to our List of Vertices
    if (toFlag)
    {
        verticesList.Add(to);
    }

     // Checks if the new Edge already exists
     // by going through all the Vertices in our List
     // and checking if there is a combination of that Vertex
     // and a Vertex in its adjecencyList
    foreach (SampleVertex v in verticesList)
    {
        foreach (SampleVertex adj in v.adjacencyList)
        {
            if (v.Text == from.Text && adj.Text == to.Text)
            {
                edgeFlag = false;
            }
        }
    }

     // In case of the new Edge being actually new
    if (edgeFlag)
    {
        if (!from.adjacencyList.Contains(to))
        {
            from.adjacencyList.Add(to);
        }
         // In case of it being a bidirectional graph
         // a Vertex is added 'to' -> 'from'
        if (cbBidirectional.IsChecked == true)
        {
            if (!to.adjacencyList.Contains(from))
            {
                to.adjacencyList.Add(from);
            }
        }
    }
}
```

The process of deleting an edge is similar to adding it, only somewhat in reverse:

```
private void DeleteEdge(SampleVertex from, SampleVertex to)
        {
             // Passes through each Vertex in our List,
             // as well as all of the Vertices adjecent to it,
             // and then checks if there is a combination of Vertices
             // that fits the Edge.
             // In which case the adjecent vertex is removed from the List
             // of ajdecencies of the main Vertex.
             // In case of a vertex being idle,
             // i.e. it not pointing to any other Vertex or being pointed at
            foreach (SampleVertex v in verticesList)
            {
                foreach (SampleVertex adj in v.adjacencyList)
                {
                    if (from == v && to == adj)
                    {
                        v.adjacencyList.Remove(adj);
                        // The Function CheckIfVertexIsUsed
                        // checks if the passed in vertex has
                        // an empty adjecency List and if no other
                        // Vertices have it in their lists
                        if (!CheckIfVertexIsUsed(to))
                        {
                            verticesList.Remove(to);
                            if (verticesList.Count == 1)
                            {
                                verticesList.Remove(from);
                            }
                        }

                        if (!CheckIfVertexIsUsed(from))
                        {
                            verticesList.Remove(from);
                            if (verticesList.Count == 1)
                            {
                                verticesList.Remove(to);
                            }
                        }
                        return;
                    }
                }
            }
        }
```

There are two most simple graph search algorithms:

DFS stands for Depth First Search is an edge-based technique. It uses the Stack data structure and performs two stages, first visited vertices are pushed into the stack, and second if there are no vertices then visited vertices are popped.

```
private async Task DFSAsync(SampleVertex vertex)
        {
             // the passed in node is the root node
             // of the Depth First Search Algorithm
            dfsOn = true;
            List<SampleVertex> stack = vertex.adjacencyList.ToList();
            stack.Reverse();
            stack.Add(vertex);

            while (stack.Count > 0)
            {
                List<SampleVertex> help = new List<SampleVertex>();
                SampleVertex temp = stack[stack.Count - 1];

                temp.Active = true;
                stack.Remove(temp);

                foreach (SampleVertex v in temp.adjacencyList.ToList())
                {
                    if (!v.Active && !stack.Contains(v))
                    {
                        help.Add(v);
                    }
                }

                if (help.Count != 0)
                {
                    help.Reverse();
                    stack.AddRange(help);
                    help.Clear();
                }
                 // This function checks if all the vertices have been activated
                if (AllActive())
                {
                    break;
                }
            }

            dfsOn = false;

            return;
        }
```

BFS stands for Breadth First Search is a vertex-based technique for finding the shortest path in the graph. It uses a Queue data structure that follows first in first out. In BFS, one vertex is selected at a time when it is visited and marked then its adjacent are visited and stored in the queue. It is slower than DFS.

```
private async Task BFSAsync(SampleVertex vertex)
        {

            bfsOn = true;
            List<SampleVertex> queue = vertex.adjacencyList.ToList();
            queue.Reverse();
            queue.Add(vertex);

            while (queue.Count > 0)
            {

                List<SampleVertex> help = new List<SampleVertex>();
                SampleVertex temp = queue[queue.Count - 1];
                temp.Active = true;

                queue.Remove(temp);
                queue.Reverse();

                foreach (SampleVertex v in temp.adjacencyList.ToList())
                {
                    if (!v.Active && !queue.Contains(v))
                    {
                        help.Add(v);
                    }
                }

                if (help.Count != 0)
                {
                    queue.AddRange(help);
                    help.Clear();
                }

                queue.Reverse();

                if (AllActive())
                {
                    break;
                }
            }

            bfsOn = false;

            return;
        }
```