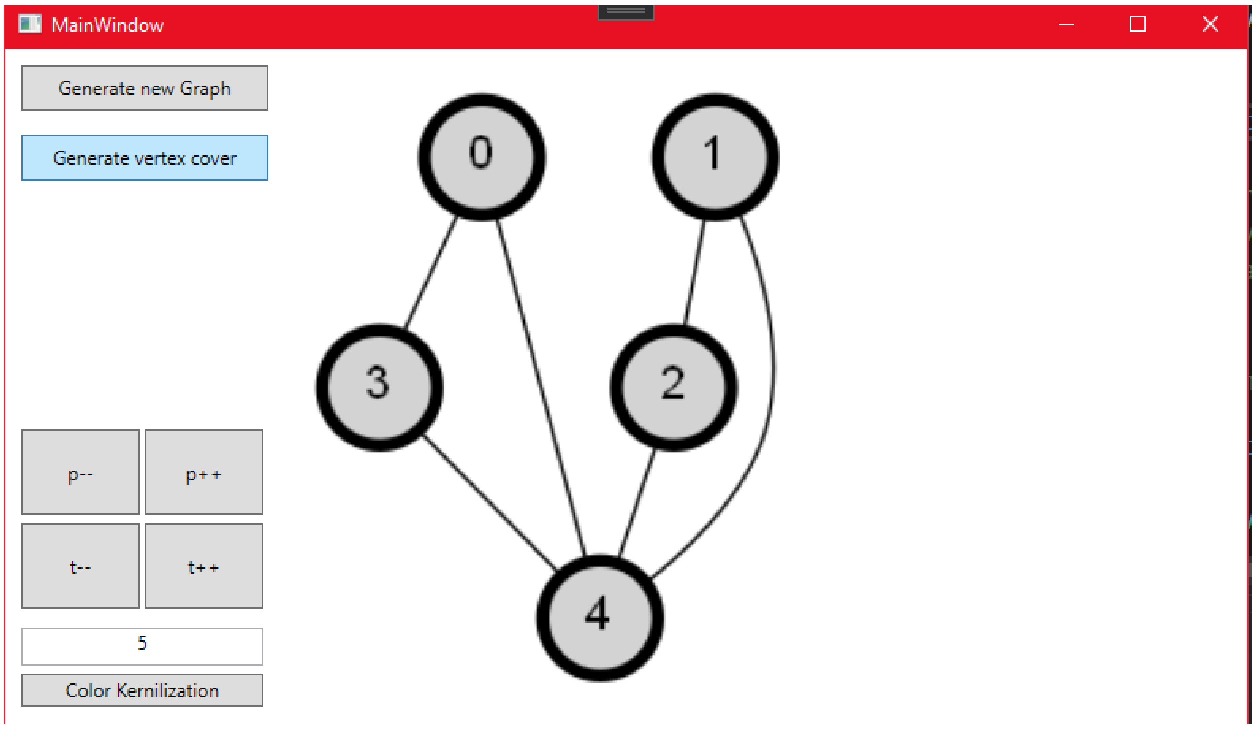
Week 3

UI with pendent and top buttons

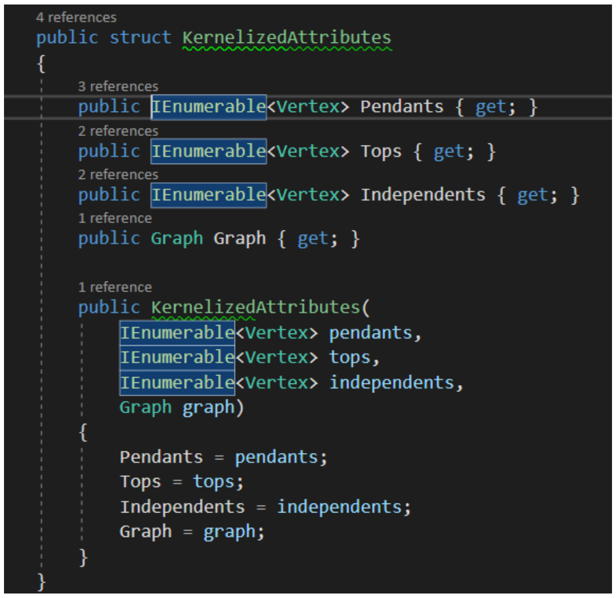
The first thing that we added was the new buttons that where required for the



We also added a separate button so that we can color the graph with the kernelization we perform.

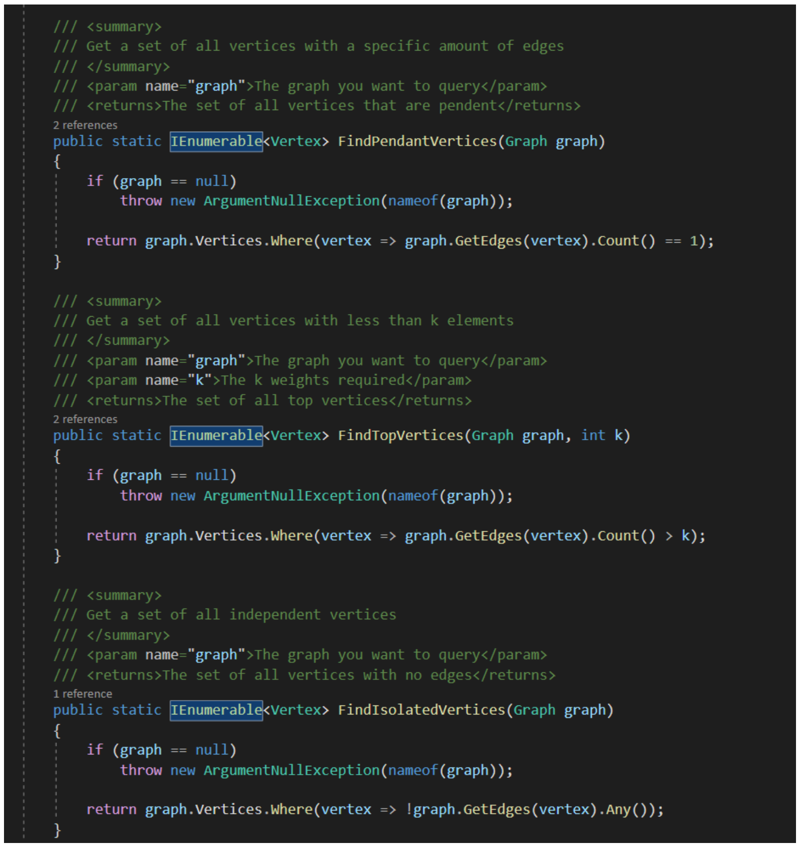
Finding Pendant and Top vertices

We created a utility that finds the pendant, top and independent vertices. For ease of use we also decided to create a basic data structure. The data structure holds all pendent, top and independent vertices:



```
4 references
public struct KernelizedAttributes
{
    3 references
    public IEnumerable<Vertex> Pendants { get; }
    2 references
    public IEnumerable<Vertex> Tops { get; }
    2 references
    public IEnumerable<Vertex> Independents { get; }
    1 reference
    public Graph Graph { get; }

    1 reference
    public KernelizedAttributes(
        IEnumerable<Vertex> pendants,
        IEnumerable<Vertex> tops,
        IEnumerable<Vertex> independents,
        Graph graph)
    {
        Pendants = pendants;
        Tops = tops;
        Independents = independents;
        Graph = graph;
    }
}
```

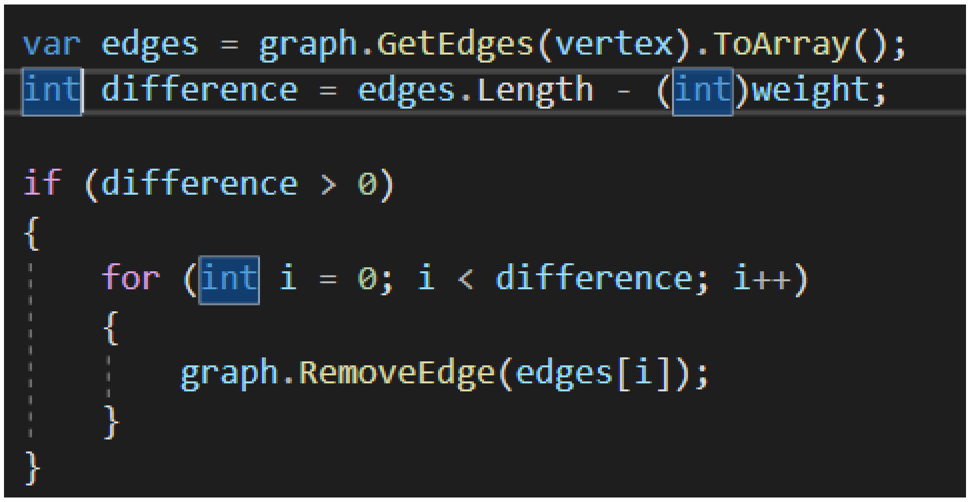
To find the values to put into this datatype we used these methods:



```
/// <summary>
/// Get a set of all vertices with a specific amount of edges
/// </summary>
/// <param name="graph">The graph you want to query</param>
/// <returns>The set of all vertices that are pendent</returns>
2 references
public static IEnumerable<Vertex> FindPendantVertices(Graph graph)
{
    if (graph == null)
        throw new ArgumentNullException(nameof(graph));

    return graph.Vertices.Where(vertex => graph.GetEdges(vertex).Count() == 1);
}

/// <summary>
/// Get a set of all vertices with less than k elements
/// </summary>
/// <param name="graph">The graph you want to query</param>
/// <param name="k">The k weights required</param>
/// <returns>The set of all top vertices</returns>
2 references
public static IEnumerable<Vertex> FindTopVertices(Graph graph, int k)
{
    if (graph == null)
        throw new ArgumentNullException(nameof(graph));

    return graph.Vertices.Where(vertex => graph.GetEdges(vertex).Count() > k);
}

/// <summary>
/// Get a set of all independent vertices
/// </summary>
/// <param name="graph">The graph you want to query</param>
/// <returns>The set of all vertices with no edges</returns>
1 reference
public static IEnumerable<Vertex> FindIsolatedVertices(Graph graph)
{
    if (graph == null)
        throw new ArgumentNullException(nameof(graph));

    return graph.Vertices.Where(vertex => !graph.GetEdges(vertex).Any());
}
```

Updating the amount of edges

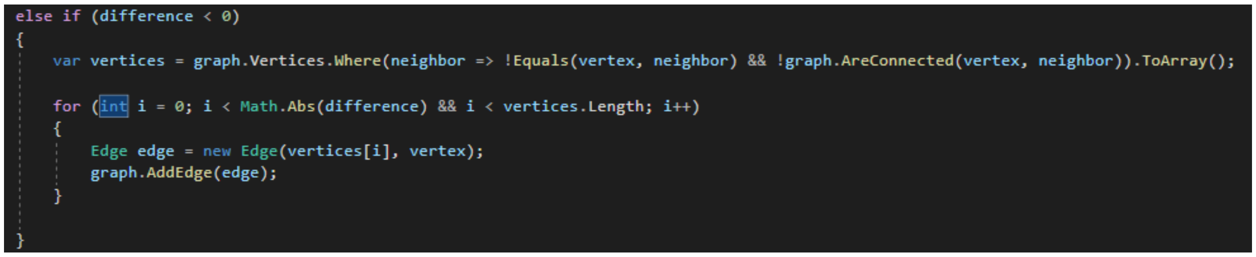
We created a method that removes or add edges so that a specific vertex will be of that degree. The first thing we do in this method is determine the distance from the requested degree this vertex has. We then have two paths. The first path is the for positive values. This means that we need to delete edges to get to the request vertex degree. That looks like this:



```
var edges = graph.GetEdges(vertex).ToArray();
int difference = edges.Length - (int)weight;

if (difference > 0)
{
    for (int i = 0; i < difference; i++)
    {
        graph.RemoveEdge(edges[i]);
    }
}
```

For the part where we have to add vertices that is the negative difference. We had to create a more complex query. This query gets all vertices that are not adjacent to the vertex we chance the edges for. We also did not want vertices that connect to themselves so we also excluded that vertex from the queried data.

We then add all the edges that we can so that we get the request vertex degree in the graph.



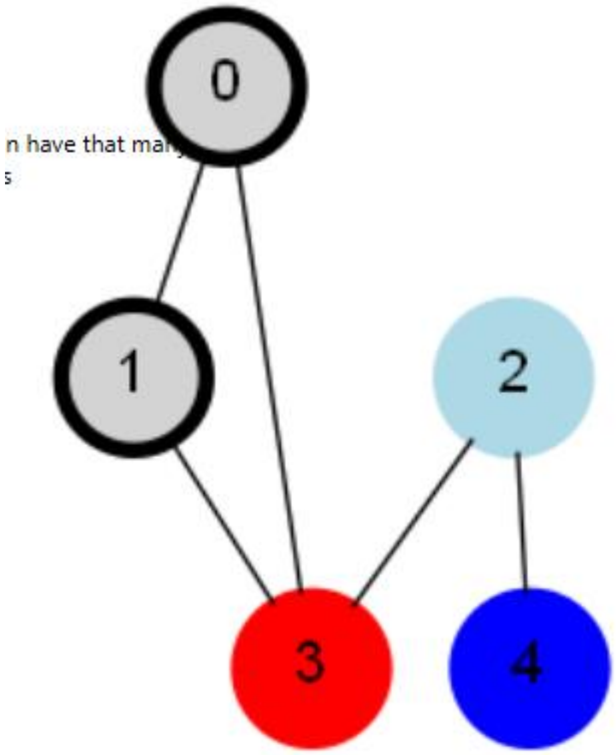
```
else if (difference < 0)
{
    var vertices = graph.Vertices.Where(neighbor => !Equals(vertex, neighbor) && !graph.AreConnected(vertex, neighbor)).ToArray();

    for (int i = 0; i < Math.Abs(difference) && i < vertices.Length; i++)
    {
        Edge edge = new Edge(vertices[i], vertex);
        graph.AddEdge(edge);
    }

}
```

Of course when the difference is 0 we do nothing.

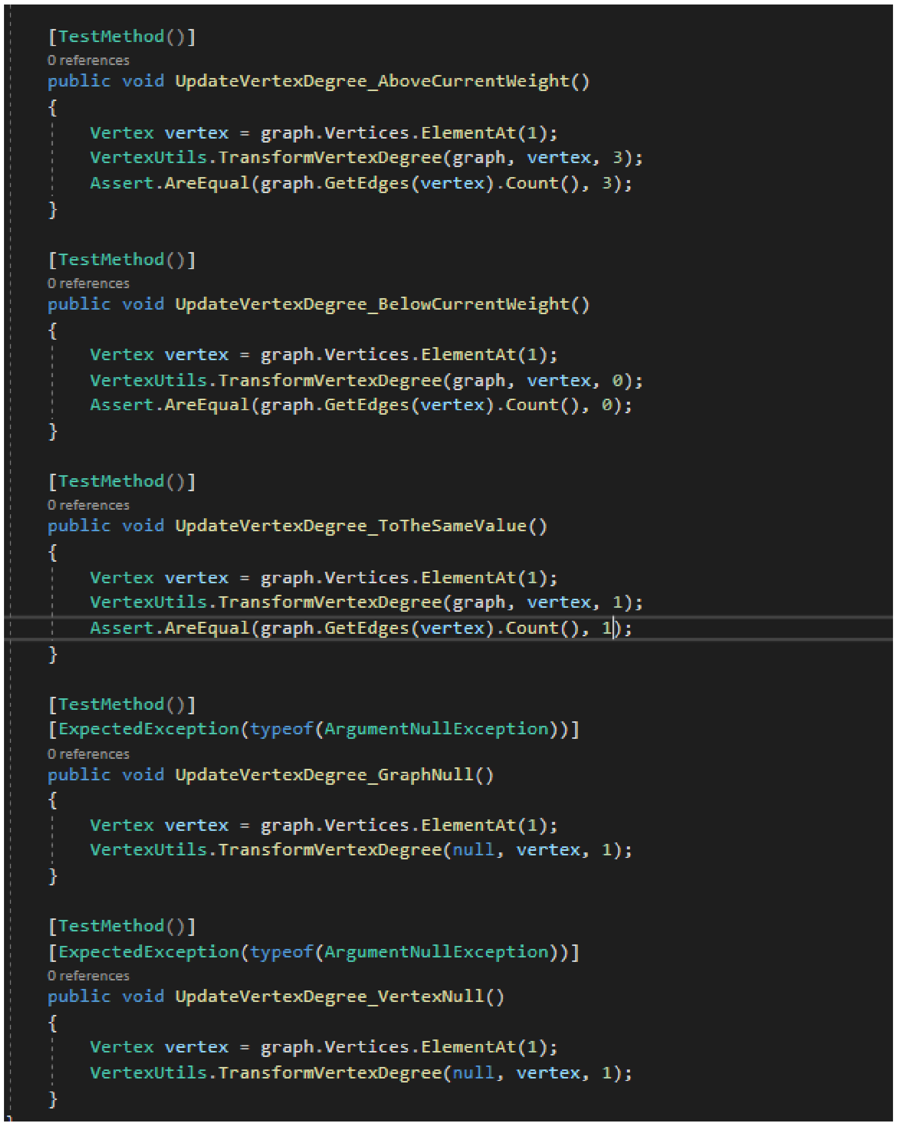
Coloring the Kernilization

For the coloring of the kernelization we decided on a few basic colors to represent the kernelization. We decided to color the top vertices red, the pendent vertices blue, the neigbouring vertices of the pendent vertices light blue, The independent vertex are colored green.

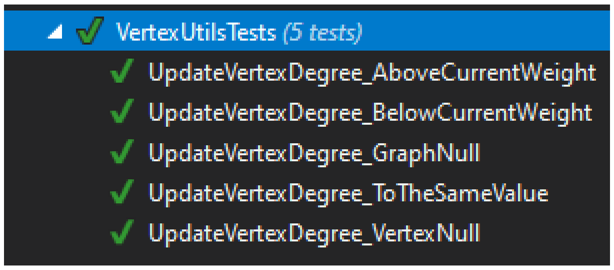
This is the end result after coloring in vertixes:



Testing



```
[TestMethod()]
0 references
public void UpdateVertexDegree_AboveCurrentWeight()
{
    Vertex vertex = graph.Vertices.ElementAt(1);
    VertexUtils.TransformVertexDegree(graph, vertex, 3);
    Assert.AreEqual(graph.GetEdges(vertex).Count(), 3);
}

[TestMethod()]
0 references
public void UpdateVertexDegree_BelowCurrentWeight()
{
    Vertex vertex = graph.Vertices.ElementAt(1);
    VertexUtils.TransformVertexDegree(graph, vertex, 0);
    Assert.AreEqual(graph.GetEdges(vertex).Count(), 0);
}

[TestMethod()]
0 references
public void UpdateVertexDegree_ToTheSameValue()
{
    Vertex vertex = graph.Vertices.ElementAt(1);
    VertexUtils.TransformVertexDegree(graph, vertex, 1);
    Assert.AreEqual(graph.GetEdges(vertex).Count(), 1);
}

[TestMethod()]
[ExpectedException(typeof(ArgumentNullException))]
0 references
public void UpdateVertexDegree_GraphNull()
{
    Vertex vertex = graph.Vertices.ElementAt(1);
    VertexUtils.TransformVertexDegree(null, vertex, 1);
}

[TestMethod()]
[ExpectedException(typeof(ArgumentNullException))]
0 references
public void UpdateVertexDegree_VertexNull()
{
    Vertex vertex = graph.Vertices.ElementAt(1);
    VertexUtils.TransformVertexDegree(null, vertex, 1);
}
```



```
◢ ✓ VertexUtilsTests (5 tests)
    ✓ UpdateVertexDegree_AboveCurrentWeight
    ✓ UpdateVertexDegree_BelowCurrentWeight
    ✓ UpdateVertexDegree_GraphNull
    ✓ UpdateVertexDegree_ToTheSameValue
    ✓ UpdateVertexDegree_VertexNull
```