Week 4

Preprocessing

We already had create a utility class that allowed us to find the top, independent and pendent vertices. We used this class to preprocess a graph. We did this by modifying the graph itself. So we updated our graph to add the functionality to remove and add vertices.

This is how we implement the preprocessing:

```
PreProcessedGraphAttributes attributes = graphPreProcessor.GetProcessedGraph(graph);
VertexCover.AddRange(attributes.IncludedVertices);
coveredGraph = attributes.ProcessedGraph;
vertexCoverSize -= attributes.IncludedVertices.Count();

foreach (Vertex discardedVertex in attributes.DiscardedVertices)
{
    if (coveredGraph.Vertices.Count() >= vertexCoverSize)
    {
        break;
    }

    VertexCover.Add(discardedVertex);
    vertexCoverSize--;
}
```

Because we want to have k vertices. We also need to add back certain points otherwise we could never have a full vertex cover. Which has all vertices. We do this by adding vertices until the vertex cover is possible.

We then use the same vertex cover algorithm as before. There is one improvement. We could still look for top vertices each step. This would increase the speed of the algorithm even more.

Testing

```
[TestMethod()]
0 references
public void PreprocessesGraphAndReturnsPreProcessedGraphAttributes()
{
    Graph graphWithTwoPendants = new Graph(matrixWithTwoPendants);
    PreProcessedGraphAttributes graphAttributes = graphPreprocessor.GetProcessedGraph(graphWithTwoPendants);

    Assert.AreEqual(1, graphAttributes.IncludedVertices.Count());
    Assert.AreEqual(2, graphAttributes.IncludedVertices.ElementAt(0).ID);

    Assert.AreEqual(2, graphAttributes.ProcessedGraph.Vertices.Count());
    Assert.AreEqual(0, graphAttributes.ProcessedGraph.Vertices.ElementAt(0).ID);
    Assert.AreEqual(1, graphAttributes.ProcessedGraph.Vertices.ElementAt(1).ID);
    Assert.AreEqual(1, graphAttributes.ProcessedGraph.Edges.Count());
    Assert.AreEqual(graphAttributes.ProcessedGraph.Vertices.ElementAt(0), graphAttributes.ProcessedGraph.Edges.ElementAt(0).StartVertex);
    Assert.AreEqual(graphAttributes.ProcessedGraph.Vertices.ElementAt(1), graphAttributes.ProcessedGraph.Edges.ElementAt(0).EndVertex);
}

[TestMethod()]
0 references
public void PreprocessesGraphAndReturnsSameGraph()
{
    Graph graphWithNoPendants = new Graph(matrixWithNoPendants);
    PreProcessedGraphAttributes graphAttributes = graphPreprocessor.GetProcessedGraph(graphWithNoPendants);

    Assert.AreEqual(graphAttributes.IncludedVertices.Count(), 0);

    CollectionAssert.AreEqual(graphAttributes.ProcessedGraph.Vertices.ToList(), graphWithNoPendants.Vertices.ToList());
    CollectionAssert.AreEqual(graphAttributes.ProcessedGraph.Edges.ToList(), graphWithNoPendants.Edges.ToList());
}

[TestMethod()]
0 references
public void PreprocessesGraphAndReturnsAllVertices()
{
    Graph graphWithThreePendants = new Graph(matrixWithThreePendants);
    PreProcessedGraphAttributes graphAttributes = graphPreprocessor.GetProcessedGraph(graphWithThreePendants);

    Assert.AreEqual(graphAttributes.IncludedVertices.Count(), 3);
    Assert.AreEqual(graphAttributes.IncludedVertices.ElementAt(0).ID, 1);
    Assert.AreEqual(graphAttributes.IncludedVertices.ElementAt(1).ID, 2);
    Assert.AreEqual(graphAttributes.IncludedVertices.ElementAt(2).ID, 4);

    Assert.AreEqual(graphAttributes.ProcessedGraph.Vertices.Count, 0);
    Assert.AreEqual(graphAttributes.ProcessedGraph.Edges.Count, 0);
}
```