

Algorithms 2 - Week 5

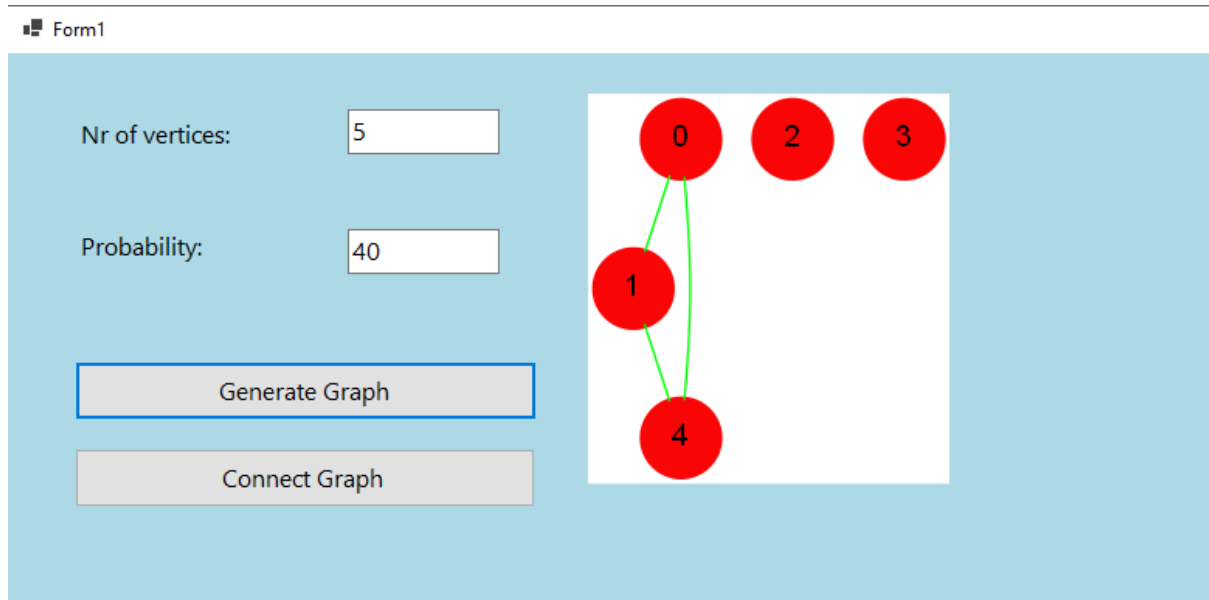
Group 2

Assignment Description

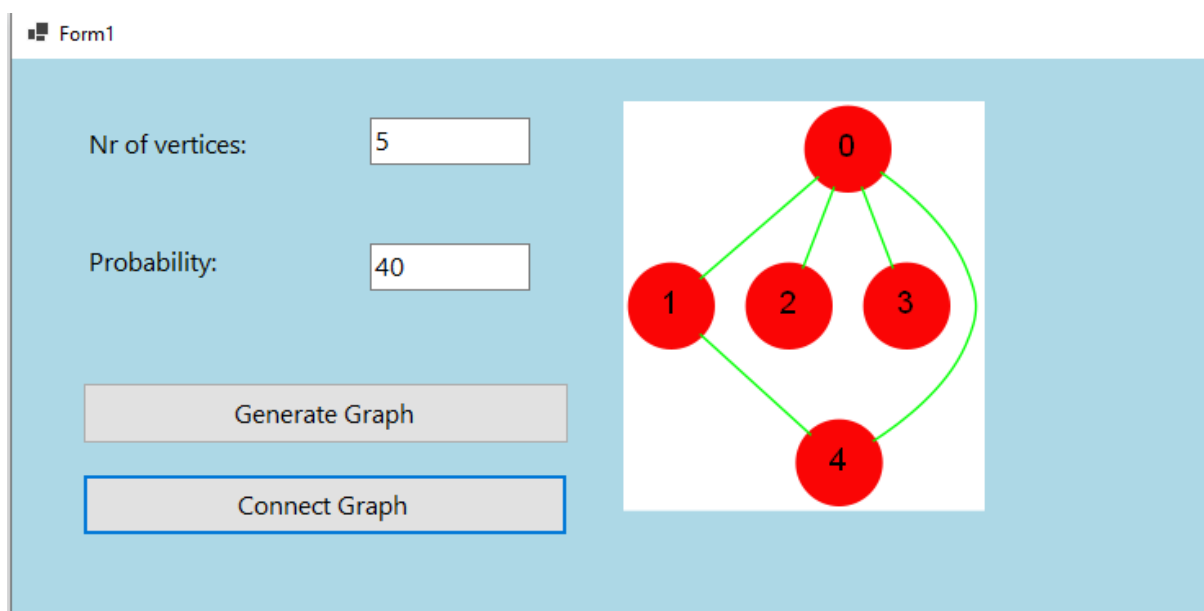
C# Windows forms application Vertex Cover. The application has two text boxes. One for the number of vertices in the graph, and the other for the probability of connection between the vertices. The button "Generate Graph" takes the input from the two text boxes and sends them to the Graph class. The Graph class generates a list of lists of integers. Each vertex has a list for storing possible connections with other vertices. After that the ConnectionProbabilities method is called and randomly connects/builds edges between vertices. The ConnectionProbabilities works according to the probability from the initial textbox input. The script for the .dot file is generated in class Graph in the ToString method. The generated text is built like the one in the LabManual provided to us. We have initially generated a dot file in which we override the content each time a new graph is generated. Initially, our filestream writes the blueprint of an unconnected graph, in order to generate the initial picture. Furthermore, we then have a button that calls a method from our Graph class in order to make the vertices connected. We then re-use the already generated file and override it. After that, we re-generate the picture using Graphviz and the dot file and represent the connected graph.

Example graphs

We have a graph with 5 vertices and a probability of 40 of the vertices being connected.



After clicking on the button to connect, we can see the picture regenerates and represents the same graph but now including connected graphs.



Week 2

Assignment description

Brute Force Search:

In a text box, the user can indicate the requested size k of the vertex cover. For each possible vertex cover assignment, determine if it is a valid vertex cover (and stop if it is found).

Solution:

We first modify our Graph class to be suited for the progress bar and the validation of the vertices:

```
public bool validVertex {  
    get; set;  
} = true;  
public int progress {  
    get; set;  
} = 0;
```

Within our graph class, we have the function that will help us validate whether it is a valid vertex cover.

```
for (int i = 0; i < cover.Length; i++)  
{  
    if (cover[i] == true)  
    {  
        count++;  
        progress++;  
    }  
}
```

The above code snippet helps us increase the progression bar based on how many times a vertex cover is true.

We then have a boolean that tells us whether we've gone through it all and set it initially to true. Furthermore, we go through the adjacent list and check how many vertices we have (which helps us set the parameter for the 2nd for loop) We then check whether the cover is false. In the case that we go through the validation process, we simply set the boolean to true and validate the vertex.

```
bool IsReached = true;
if (count == k)
{
    for (int i = 0; i < cover.Length; i++)
    {
        for (int j = 0; j < adjList[i].Count;
j++)
        {
            if ((cover[i] == false) &&
(cover[adjList[i][j]] == false))
            {
                IsReached = false;
                return false;
            }
        }
    }
}
else
{
    IsReached = false;
    return false;
}
```

To specify whether the vertex cover is successful or not, we have the following miscellaneous code to control a label (ui to show the user whether it is true or not) and the progress bar.

```
        UsedVertices usedV = new UsedVertices();
        graph.validVertex = true;
        bool algResult =
usedV.ValidationOfVertexRec(graph, new
bool[graph.getVertices()], graph.getVertices(), 0,
Convert.ToInt32(txtBSizeK.Text));
        if (algResult)
        {
            lblResult.Text = "TRUE";
            lblResult.ForeColor = Color.Yellow;
            if (graph.progress <= 100)
            {
                progressBarGraph.Value = graph.progress;
                graph.progress = 0;
            }
            else
            {
                progressBarGraph.Value = 100;
                graph.progress = 0;
            }
        }
        else
        {
            progressBarGraph.Value = 0;
            graph.progress = 0;

            lblResult.Text = "FALSE";
            lblResult.ForeColor = Color.Red;
        }
    }
```

Output & UI:

Form1

Nr of vertices:

Probability:

k

TRUE

A progress bar showing approximately 20% completion, with a green segment on the left and a grey segment on the right.

Week 3

Assignment description:

pendants & tops

The Pruning and Kernelization algorithm are based on pendent and tops vertices (a tops vertex: vertex with $> k$ edges).

Add (four) buttons to increase & decrease the number of pendants & tops. E.g. button `p--` selects an arbitrary non-pendant vertex and makes it a pendant by removing arbitrary edges; button `t++` selects an arbitrary non-tops vertex and makes it a top by adding arbitrary edges.

prepare for kernelization

Perform a kernelization by:

- finding isolated vertices
- finding pendant vertices (and their adjacent vertices)
- finding tops vertices

Those should be indicated in the graph picture with proper coloring of the vertices and/or edges.

- -

We added four buttons - for increasing and decreasing the number of pendants and tops. With every increase/ decrease of the pendants and tops the colours of the different vertices and edges between them are changed and the graph is visualised all over again. The .dot file is rewritten and re-executed.

Incrementing / decrementing pendants

“Pendant”: Let G be a graph, a vertex v of G is called a pendant vertex if and only if v has degree 1. In other words, pendant vertices are the vertices that have degree 1, also called pendant vertex.

Incrementing pendants

In order to do this, we iterate through our matrix and check whether there is an edge. If we do indeed find one, we remove it and add a pendant

```
public void PendantIncrement()
{
    for (int i = 0; i < vertices; i++)
    {
        if (adjList[i].Count > 1)
        {
            for (int j = 0; j < vertices; j++)
            {
                if (adjList[j].Count > 1)
                {
                    remove_Edge(i, j);
                }
            }
            if (!Pendants.Contains(i))
            {
                Pendants.Add(i);
            }
            break;
        }
    }
}
```

Decrementing pendants

As *PendantIncrement()*, this process is the same, but we now instead add a connection (edge) between two vertices and remove a pendant in its place

```
public void PendantDecrement()
{
    for (int i = 0; i < vertices; i++)
    {
        if (adjList[i].Count == 1)
        {
            for (int j = 0; j < vertices; j++)
            {
                if (adjList[j].Count == 1)
                {
                    AddConnection(i, j);
                }
            }
            if (Pendants.Contains(i))
            {
                Pendants.Remove(i);
            }
            break;
        }
    }
}
```

Incrementing / decrementing tops

As the name suggests, with the following functions we remove/add top vertices. A top vertex is a vertex with more than k edges. We check if each one of the vertices has less than k edges and if so we make a connection between this and another one like this vertex. And we add the new vertex with more than k edges to the list of Tops.

```
public void TopIncrement(int k)
{
    for (int i = 0; i < vertices; i++)
    {
        if (adjList[i].Count <= k)
        {
            for (int j = 0; j < vertices; j++)
            {
                if (adjList[i].Count <= k)
                {
                    AddConnection(i, j);
                }
            }
            if (!Tops.Contains(i))
            {
                Tops.Add(i);
            }
            break;
        }
    }
}
```

We execute TopDecrement the same way as TopIncrement, but instead of checking whether a vertex has less than or equal to k edges, we check if a vertex has more than k edges. We then remove edges from such like vertices one by one and remove them from the Tops list.

```

public void TopDecrement(int k)
{
    for (int i = 0; i < vertices; i++)
    {
        if (adjList[i].Count > k)
        {
            for (int j = 0; j < vertices; j++)
            {
                if (adjList[j].Count > k)
                {
                    RemoveEdge(i, j);
                }
            }
            if (Tops.Contains(i))
            {
                Tops.Remove(i);
            }
            break;
        }
    }
}

```

In order to find pendants and outline them, we have a function that helps us with the process. We initially clear the list of our pendants and then iterate through our matrix (graph) in order to find and add all of the newly added/removed pendants.

```

private void FindPendants()
{
    Pendants.Clear();

    for (int i = 0; i < vertices; i++)
    {
        if (adjList[i].Count == 1)
        {

```

```

        if (!Pendants.Contains(i))
        {
            bool ok = true;
            foreach (int pendant in Pendants)
            {
                if (adjList[pendant].Contains(i))
                {
                    ok = false;
                }
            }
            if (ok)
            {
                Pendants.Add(i);
            }
        }
    }
}

```

Similarly, we have a function to discover newly added/removed tops in the following function

```

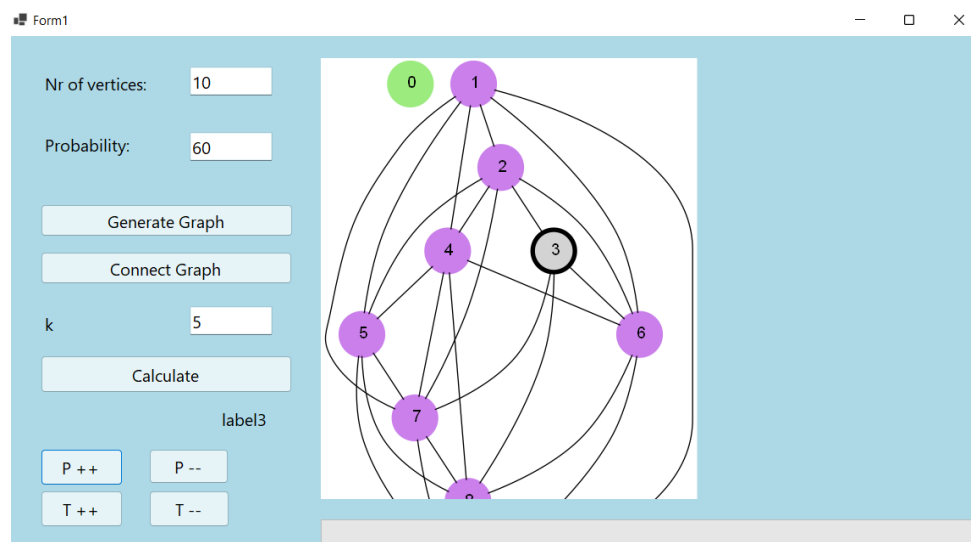
private void FindTops(int k)
{
    Tops.Clear();
    for (int i = 0; i < vertices; i++)
    {
        if (adjList[i].Count > k)
        {
            if (!Tops.Contains(i))
            {
                Tops.Add(i);
            }
        }
    }
}

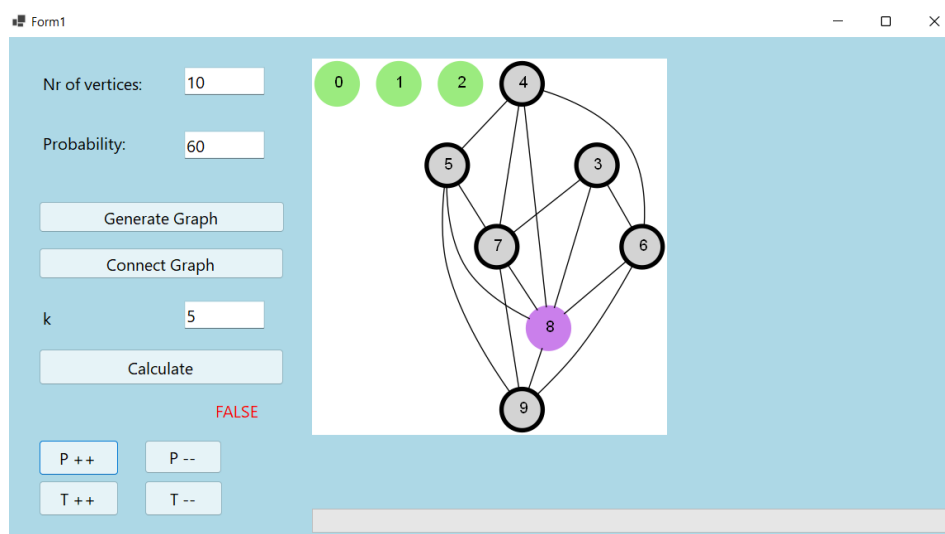
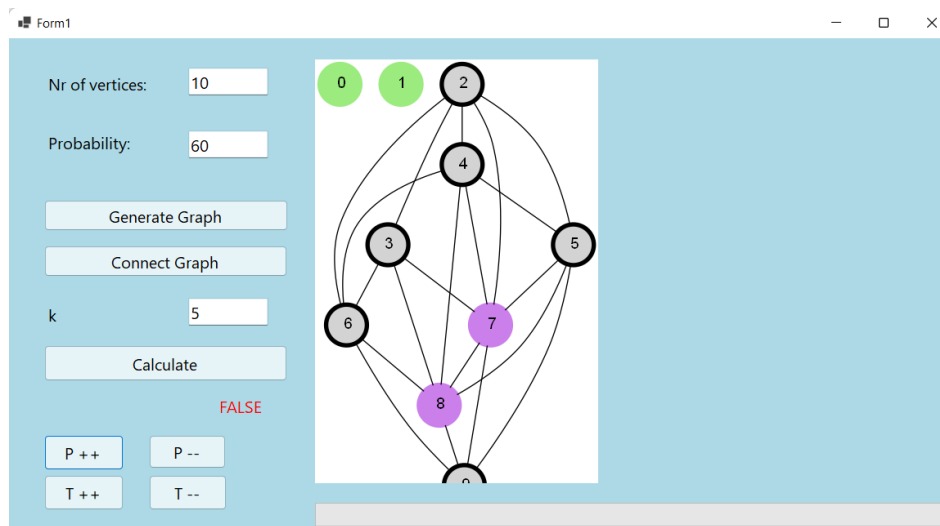
```

Lastly, in order to outline lone vertices (isolated), we initialize a list of IsolatedVertices, which upon calling the function, is also cleared. We then iterate through our graph and see whether it contains any isolated vertices.

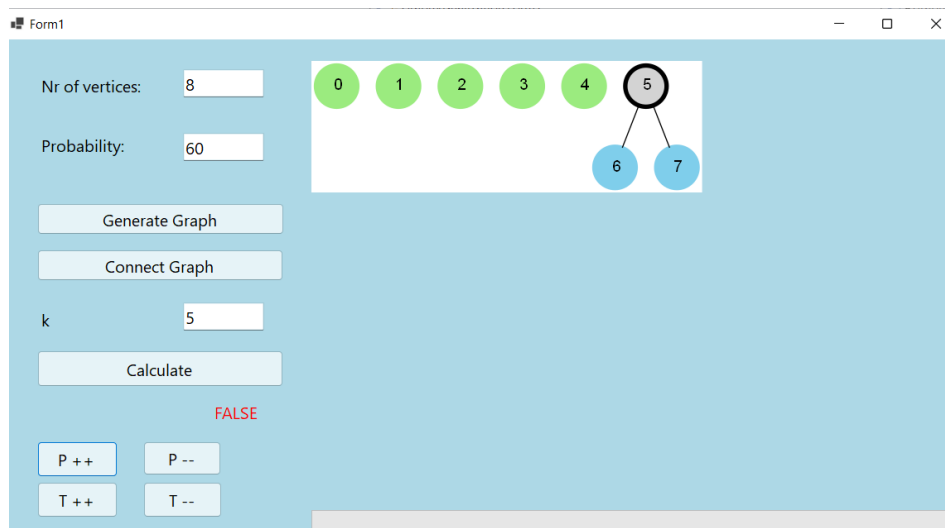
```
private void findIsolated()
{
    IsolatedVertices.Clear();
    for (int i = 0; i < vertices; i++)
    {
        if (adjList[i].Count == 0)
        {
            if (!IsolatedVertices.Contains(i))
            {
                IsolatedVertices.Add(i);
            }
        }
    }
}
```

Pendant Increment:





Pendant Increment:



Pendant decrement:

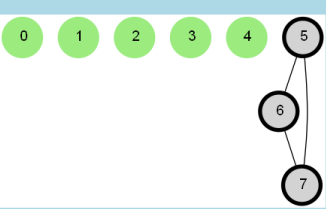
Form1

Nr of vertices:

Probability:

k

FALSE



The graph visualization shows 8 vertices labeled 0 through 7. Vertices 0, 1, 2, 3, and 4 are green circles. Vertices 5, 6, and 7 are black circles. The edges form a path: 0-5, 5-6, and 6-7. Vertices 1, 2, 3, and 4 are isolated.

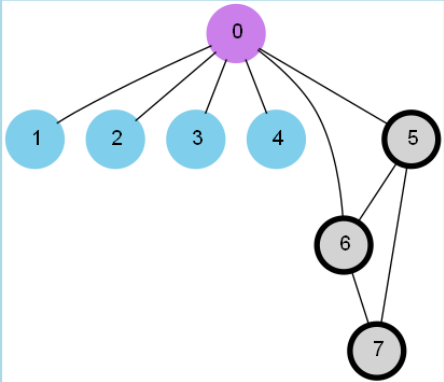
Tops increment:

Nr of vertices:

Probability:

k

FALSE



The graph visualization shows 8 vertices labeled 0 through 7. Vertex 0 is a purple circle. Vertices 1, 2, 3, and 4 are blue circles. Vertices 5, 6, and 7 are black circles. The edges form a star graph with vertex 0 at the center connected to vertices 1, 2, 3, 4, and 5. Additionally, there is a path: 5-6-7.

Week 4

Description:

enhanced brute force

Based on the preparation steps made in week 3 and reduction rules for kernelization for vertex cover problem as described in [Kernelization - Wikipedia](#), certain branches of the brute force can be excluded (because we know that certain vertices are in (c.q. out) the vertex cover). Improve the brute force search such that the not relevant branches are excluded.

If everything works out well, you will see large jumps in the progress bar.

For this week's assignment we had to enhance the "brute search force" assignment from week 3. Having found the pendant vertices, top vertices and the isolated vertices, we can remove all non-relevant vertices and edges from the graph. The output of this week's improvement displays the graph and the non-relevant vertices - all in different colours. The progress bar's functionality has also improved - now showing the "connectivity" of the graph.

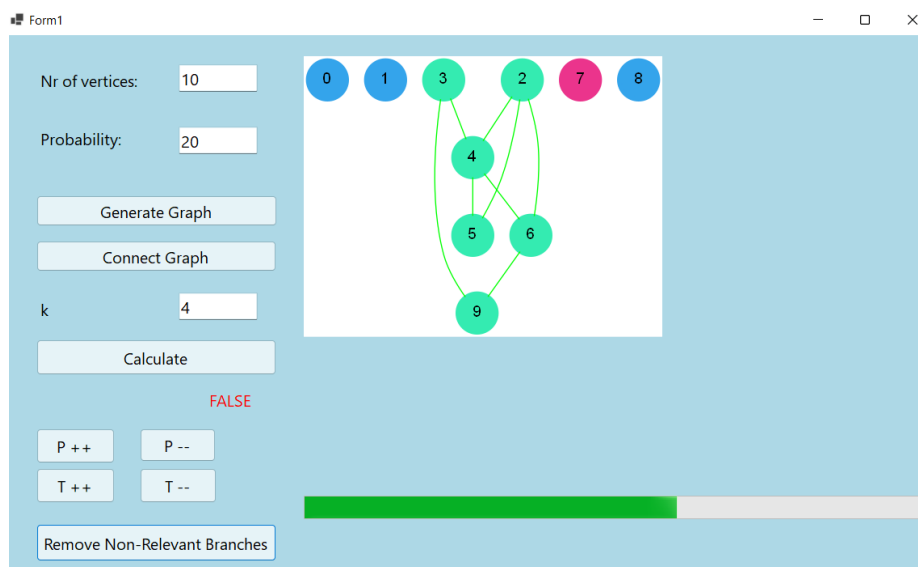
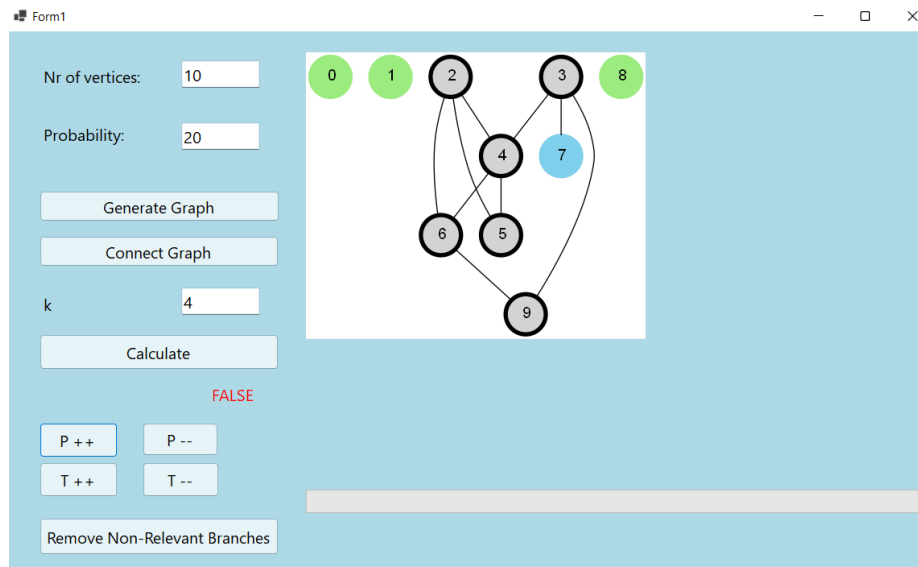
Removing Non-Relevant Vertices

Having already created lists for storing the pendant, top and isolated vertices, it is rather easy to distinguish them from the graph, thus creating a new graph within the graph without removing anything. Still the graph and the non-relevant vertices cannot get mixed up as they are displayed in different colours. The non-relevant vertices do not share edges with the relevant ones.

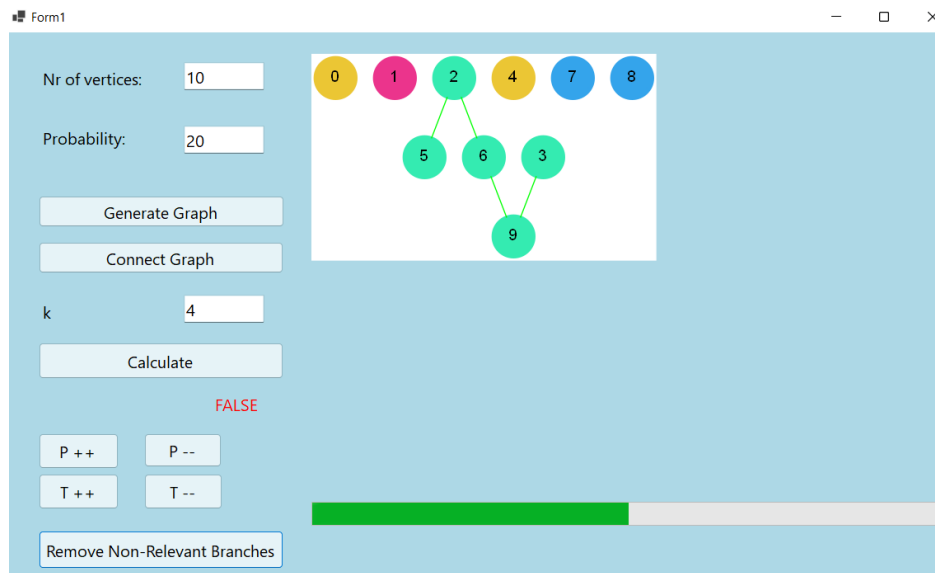
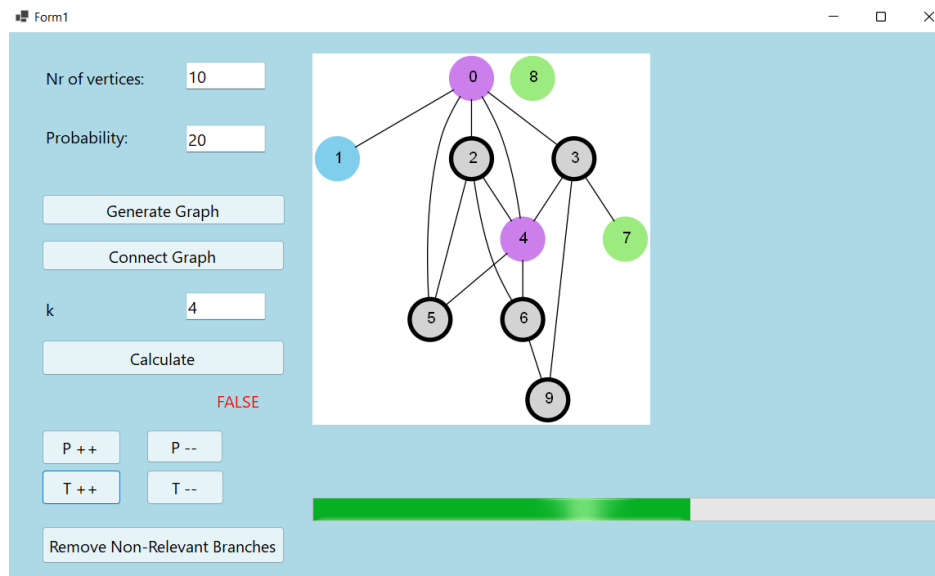
Colouring

In order to handle the colouring of the vertices/edges, a function that takes in an index is used to help ease the process. We check whether any of our lists (of Pendants, tops, isolated/lone vertices and the graph itself) contain an element at the selected index. Based on that, we give each instance a different colour to help with distinction between each other.

Output:



Here, we can see that after removing the non-relevant vertices, the pendants were coloured in pink, the isolated vertices were coloured in blue and the rest of the graph stays connected and its coloured in green.



Here, we can see that after removing the non-relevant vertices, the pendants were coloured in pink, the isolated vertices were coloured in blue, the tops were coloured in yellow and the rest of the graph stays connected and its coloured in green.

Week 5

Assignment description:

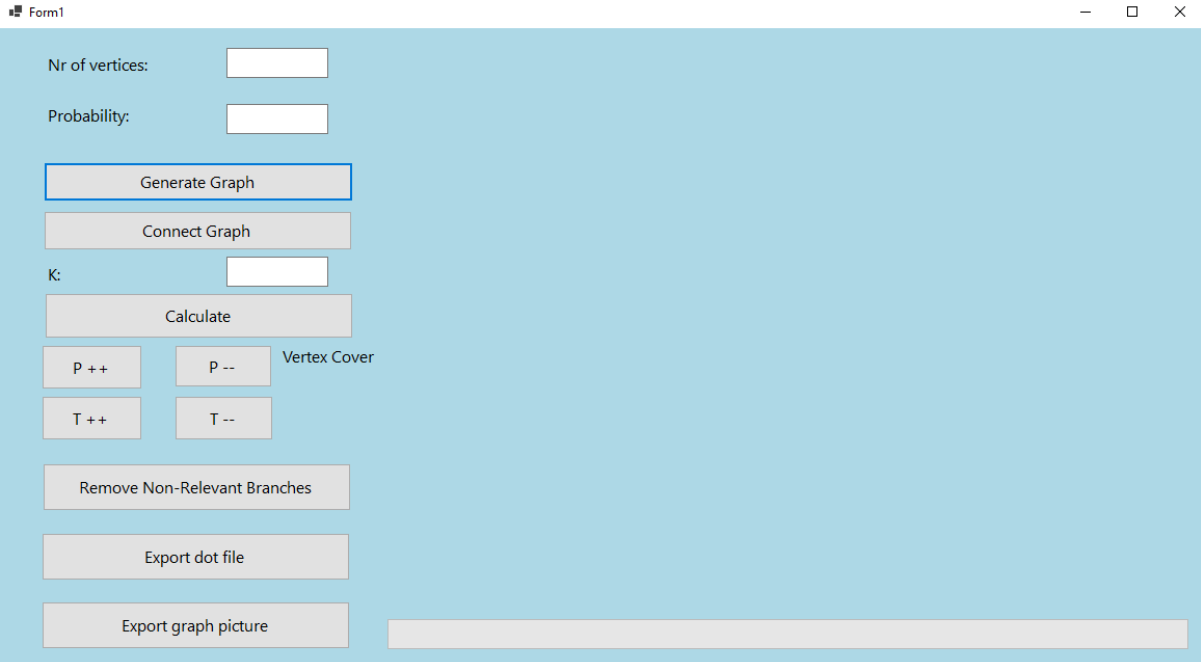
further ideas

recursive kernelization, smart search tree, export and import graphs (to exchange with your fellow students)

For this week we were left to further develop the recursiveness of our kernelization function. We also included an export function to the application. Now both the .dot file can be exported into a word file and the .png file is exported as a picture.

Furthermore, in order to make sharing/viewing the information about a given graph, we added two extra buttons that allow the user to export the graph as a png, as well as directly opening the dot file. The purpose of this is for users to be able to easily share graph pictures /information.

The latest interface can be seen below



The screenshot shows a software interface titled "Form1" with a light blue background. It contains several input fields and buttons arranged vertically on the left side. The inputs are: "Nr of vertices:" with a text box, "Probability:" with a text box, "K:" with a text box, and "Vertex Cover" with two buttons labeled "P ++" and "P --". Below these are buttons for "Generate Graph", "Connect Graph", "Calculate", "Remove Non-Relevant Branches", "Export dot file", and "Export graph picture". At the bottom right, there is a long, empty rectangular box, likely a placeholder for a graph visualization or output.