Week 1

Vertex Cover

Group 4

Mohammed Aljader & Andreea Șindrilaru

Table of contents

Problem description

Introduction

This document describes the programming assignment for ALG2. Do the exercise as if you are a researcher who is investigating a new algorithm: add your own features to see the characteristics of the algorithm and how it operates. And make it possible to interfere with the algorithm. Suggestions are made in the text below, but your imaginations will be much richer than the teacher's...

Description

Implement the Vertex Cover with Kernelization, Pruning, Search Tree Optimization and Brute Force. The text below is a guideline; you don't have to implement everything literally.

Week 1

week 1

generate graph

With two text boxes the user selects the number of vertices `n` in the graph and a probability `p`. Create the `nxn` adjacency matrix where each edge is added to the graph with probability of `p`.

Show the graph in a picture box.

connect

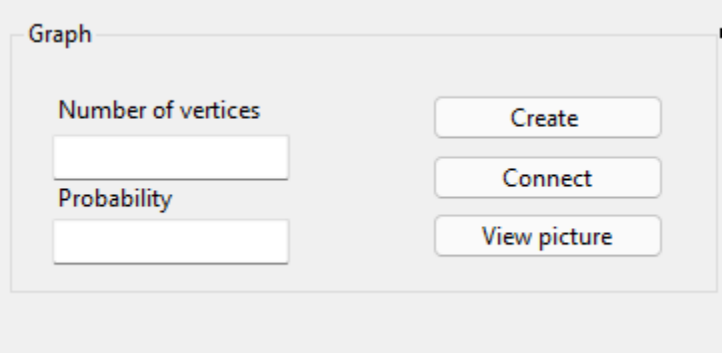
The graph may be not connected. Add a button to make it connected. In such a case: find two disconnected subgraphs, select an arbitrary vertex in each of them and add an edge between those two vertices.

Implementation

Week 1

During the first week, we had to make the implementation to generate a graph and connect it.

First, we initialized a form with 2 textboxes, 1 textbox where the user can mention the number of vertices the graph should contain, and another textbox for the probability. This probability will be used in adding edges.



What is interesting about this assignment is that we have to use a tool to display the graph as a picture, in a picture box. After installing “GraphViz”, we had to write the graph into a “graph.dot” file according to the following structure :

```
graph my_graph {
   node [ fontname = Arial, style="filled,setlinewidth(4)",
shape=circle ]
    node0 [ label = "A" color="#4040f040" fillcolor="#40f04040" ]
    node3 [ label = "D" color="#40f04040" ]
    node4 [ label = "E" color="#f0404040" ]
    node5 [ label = "F" color="#4040f040" ]

    node6 [ label = "G" ]
    node8 [ label = "I" ]
    node9 [ label = "J" ]
    node10 [ label = "K" ]
    node11 [ label = "L" ]
    node0 -- node4 [ style=dashed ]
    node0 -- node5 [ color="#40404040", style=dashed ]
    node0 -- node6 [ color=red, style=dashed ]
    node0 -- node10 [ style=dashed ]
    node3 -- node11 [ color=yellow ]
    node5 -- node6 [ color="#40404040", style=dashed ]
    node5 -- node8 [ color="#40404040", style=dashed ]
    node5 -- node9 [ color="#40404040", style=dashed ]
    node6 -- node10
    node6 -- node11
    node8 -- node9 [ color=green ]
    node8 -- node10
}
```

In order to do that, we created a method to write the graph to a .dot file.

```
fileStream = new FileStream("graph.dot", FileMode.Open, FileAccess.Write);
                streamWriter = new StreamWriter(fileStream);

                streamWriter.WriteLine("graph my_graph { node[fontname =
Arial, style = \"filled,setlinewidth(4)\",shape = circle]");

                for (int i = 0; i < adjacent_list.Count(); i++)
                {
                    streamWriter.WriteLine("node" + i + "[ label =\" " + i
+ "\"]");
                }

                for (int i = 0; i < adjacent_list.Count(); i++)
                {
                    for (int j = 0; j < adjacent_list[i].Count(); j++)
                    {
                        if (adjacent_list[i][j] >= i)
                        {
                            streamWriter.WriteLine("node" + i + "--" +
"node" + adjacent_list[i][j]);
                        }
                    }
                }
                streamWriter.WriteLine("}");
```

Moving on to the interesting part, that being the generation of the graph and connecting it.

After doing some research, we found out that in terms of time complexity, making an adjacency list would be more efficient than making an adjacency matrix. Therefore, we have decided to use an adjacency list instead of an adjacency matrix in order to achieve better results.

In order to initialize the graph we have created a class ***Graph.*** In this class we made the needed function to create vertices of the graph and add edges to connect them. First, we create **AdjacentList** and **Vertices** properties and we made create_graph function in order to add the vertices to the **AdjacentList.**

```
 public void create_graph (int v)
        {
```

```
            this.vertices = v;
            adjacent_list = new List<List<int>>();

            for (int i = 0; i < v; i++)
            {
                adjacent_list.Add(new List<int>());
            }
        }
```

Then we made **add_adge** function to connect the source vertex with the destination one and add them both to the **AdjacentList**.

```
 public bool add_edge(int source, int destination)
        {
            if (adjacent_list[source].Contains(destination) || source ==
destination)
            {
                return false;
            }
            adjacent_list[source].Add(destination);
            adjacent_list[destination].Add(source);
            return true;
        }
```

Afterwards, in order to add edges between the vertices, we are tasked to use probabilities. Therefore, our function takes in a parameter to take the probability into account, in such a way that we iterate through the vertices and “make the probability” of adding an edge between any two vertices.

```
 public void add_edges_on_probability(int prob)
        {
            for (int i = 0; i< vertices; i++)
            {
                for(int j = i + 1; j < vertices; j++)
                {
                    int probability = random.Next(1, 101);
                       if (probability <= prob)
                       {
```

```
                        add_edge(i, j);
                    }
                }
            }
        }
```

In order to see what vertices need to be connected, we have made the function seen below. First, we check if there is any arbitrary vertex. If there's no such vertex, then we add the vertex "v" to a list ("verticesToConnect"). Afterwards, we mark the current node as being visited and iterate through the adjacent vertices of "v". If there is a vertex which is not visited, then we call the function recursively.

```
public void reachable_vertices(int v, bool[] visited, bool
arbitraty_vertex)
        {
            if (!arbitraty_vertex)
            {
                verticiesToConnect.Add(v);
                arbitraty_vertex= true;
            }

            // Mark the current node as being visited.
            visited[v] = true;
            // Recur for all the vertices adjacent to this vertex

            foreach (int x in adjacent_list[v])
            {
                if (!visited[x])
                {
                    reachable_vertices(x, visited, arbitraty_vertex);
                }
            }
        }
```

With the help of this function, we first mark all the vertices not being visited so that we can then iterate through all of them and check if they are not visited so that we can find all reachable vertices from vertex v.

```
public void components()
        {
            // Mark all the vertices as being not visited
            bool[] visited = new bool[vertices];
            for (int v = 0; v < vertices; v++)
            {
                if (!visited[v])
                {
                    // Print all reachable vertices from v.
                    reachable_vertices(v, visited, false);
                }
            }
        }
```

Finally, to connect the vertices, we iterate through the list of verticesToConnect and add edges.

```
public void connect()
        {
            foreach (int vertex in verticiesToConnect)
            {
                add_edge(verticiesToConnect[0], vertex);
            }
        }
```