Week 4

Vertex Cover

Group 4

Mohammed Aljader & Andreea Șindrilaru

Table of contents

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.

Week 2

week 2

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).

It can be implemented in a recursive method. A global sketch:

```
bool Validate(Graph g, bool[] cover, int n, int i, int k)
{
      if (i == n)
      {
            return (g.Validate(cover, n, k));
      }
      else
      {
            cover[i] = false;
            Validate(g, cover, n, i+1, k);
            cover[i] = true;
            Validate(g, cover, n, i+1, k);
      }
}
```

Week 3

week 3

For this part you need to do some pre-research about the kernelization techniques and in particular kernelization in vertex cover problem.

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.

Week 4

week 4

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.

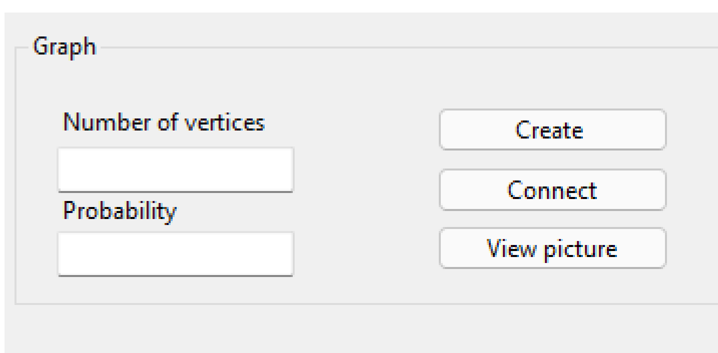
If everything works out well, you will see large jumps in the progress bar.

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);
            }
        }
```

Week 2

In the second week, we had to implement a functionality to let the user indicate a specific size k of the vertex cover. First of all, we have to check for each possible vertex cover if it's valid and return “True”, if not return “False”.

In order to solve the current assignment, we made some functions that try recursively all the possible combinations until either one good combination that matches the requirements is found or exhaust all options, suggesting that there is no solution for the given input.

```
// function Validate in Graph class that checks if we can cover the graph,
// with the given size of vertex cover.
// cover is the list of the vertices that can cover the graph
// n is the number of the current vertices
// k is the given size of the vertex cover.
public bool Validate(bool[] cover, int n, int k)
        {
            if (!IsOkVertex)
            {
                return true;
            }
```

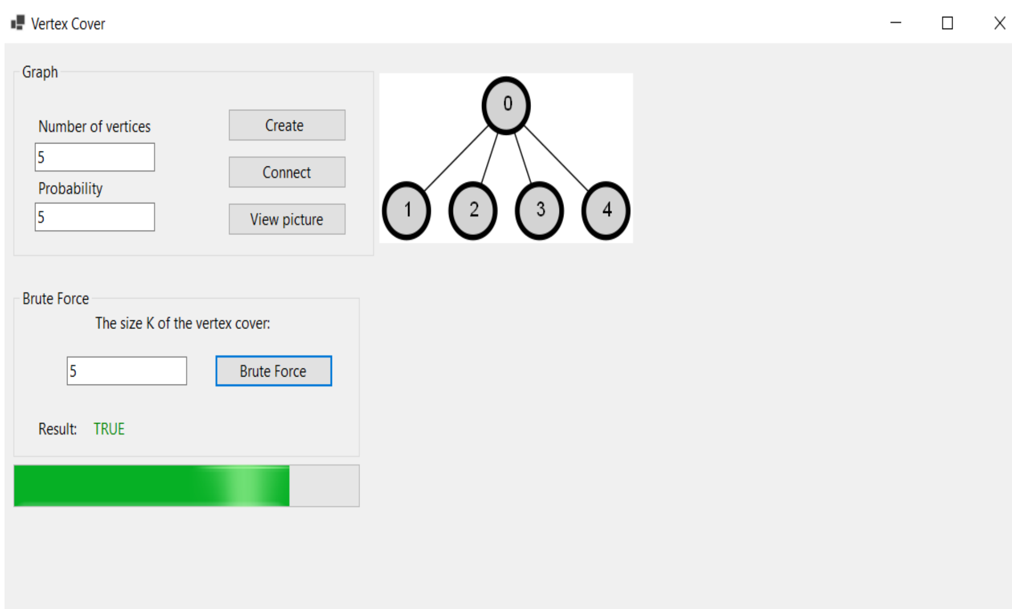
```
            int count = 0;

            // we iterate through each vertex found in the cover list
            for (int i = 0; i < cover.Length; i++)
            {
                // if there is any cover that is true, then
                if(cover[i] == true)
                {
                    // increment the count
                    count++;
                    progress++; //used for the progress bar
                }
            }

            // we check if the count is equal to the given size
            bool IsReached = true;
            if(count == k)
            {
                // then we go through all the cover and adjacent lists
                for (int i = 0; i < cover.Length; i++)
                {
                    for (int j = 0; j < adjacent_list[i].Count; j++)
                    {

                        // we go through all the edges and check if all
                        // the edges are covered
                        if ((cover[i] == false) &&
(cover[adjacent_list[i][j]] == false))
                        {
                            IsReached = false;
                            return false;
                        }
                    }
                }
            }
            // if the count is not equal to the input size return false
            else
            {
                IsReached = false;
                return false;
            }
```

```
            if (IsReached)
            {
                IsOkVertex = false;
            }

            return IsReached;
        }
    }
```

```
// This is the main validate function (as described in the LabManual) which
// is used in the form to
// validate the vertex cover of size k.
 public bool Validate(Graph g, bool[] cover, int n, int i, int k)
        {
            if (k > n)
                return false;
            if (!g.IsOkVertex)
            {
                return true;
            }

            if (i == n)
            {
                return g.Validate(cover, n, k);
            }
            else
            {
```

```
                cover[i] = false;
                Validate(g, cover, n, i + 1, k);
                if (!g.IsOkVertex)
                {
                    return true;
                }

                cover[i] = true;
                Validate(g, cover, n, i + 1, k);
                if (!g.IsOkVertex)
                {
                    return true;
                }
            }
            return false;
        }
```

Output week 2



Week 3

In the third week we have been tasked to implement kernelization for the vertex cover. Before we implemented the process, we had to do some small research about different kernelization techniques.

Since vertex cover is known as an NP-hard problem, we can use the following reduction rules to kernelize it.

1. If **k** (set of vertices that includes an endpoint of every edge in the graph, if there exists such a vertex, or failure exception if there is no such vertex) > 0 and a vertex **v**, whose degree is greater than k, then remove **v** from the graph and decrease **k** by one.
2. If **v** is an isolated vertex, remove it.
3. If more than $\mathbf{k}^2$ edges remain in the graph, and neither of the previous rules can be applied, then the graph cannot contain a vertex cover of size k.

Besides these reduction rules, we also need to add tops and pendants. A top vertex is a vertex with more than k edges, while a pendant is a vertex which has only 1 neighbor.

Below, the actual implementation can be found alongside comments that explain what each part does.

```
public void kernelization(int k)
       {
           findPendants();
           findTops(k);
           findIsolated();
       }

       // a function to find the pendants of the graph
       private void findPendants()
       {
           Pendants.Clear()

           // iterate through the vertices of the graph
           for (int i = 0; i < Vertices; i++)
           {
               // checks if the vertex is leaf and only has one adjacent
vertex
               if (adjacent_list[i].Count == 1)
               {
```

```
                    // make sure that Pendants list does not contain the
current vertex
                    if (!Pendants.Contains(i))
                    {
                        bool ok = true;
                        // iterate through all pendants elements to make
sure that the adjacents of the elements
                        // does not contain the current vertex
                        foreach (int pendant in Pendants)
                        {
                            if (adjacent_list[pendant].Contains(i))
                            {
                                ok = false;
                            }
                        }
                        //if we found the pendant we add it to the pendants
list
                        if (ok)
                        {
                            Pendants.Add(i);
                        }
                    }
                }
            }
        }

        private void findTops(int k)
        {
            Tops.Clear();

            // iterate through the vertices of the graph
            for (int i = 0; i < Vertices; i++)
            {
                // checks if the current vertex has more adjacent than the
(k edges)
                if (adjacent_list[i].Count > k)
                {
                    // adds the current vertex to the Tops vertices list
                    if (!Tops.Contains(i))
                    {
                        Tops.Add(i);
                    }
                }
```

```
            }
        }

        private void findIsolated()
        {
            IsolatedVertices.Clear();

            // iterate through the vertices of the graph
            for (int i = 0; i < Vertices; i++)
            {
                // checks if the current vertex is not connected to any
vertices in the graph
                if (adjacent_list[i].Count == 0)
                {
                    // adds the current vertex to the Isolated Vertices
list
                    if (!IsolatedVertices.Contains(i))
                    {
                        IsolatedVertices.Add(i);
                    }
                }
            }
        }
```

```
 public void TopIncrement(int k)
        {
            // iterate through the vertices of the graph
            for (int i = 0; i < Vertices; i++)
            {
                // checks if the current vertex is less or equal than (k
edges)
                if (adjacent_list[i].Count <= k)
                {
                    // iterate through the rest of vertices of the graph
                    for (int j = 0; j < Vertices; j++)
                    {
                        // checks if the j vertex is less or equal than (k
```

```
edges)
                        if (adjacent_list[j].Count <= k)
                        {
                            // we add new edge between i and j
                            add_edge(i, j);
                        }
                    }
                    // we add the current vertex to the Tops list
                    if (!Tops.Contains(i))
                    {
                        Tops.Add(i);
                    }
                    // out of for-loop to avoid looping forever
                    break;
                }
            }
        }

        public void TopDecrement(int k)
        {
            // iterate through the vertices of the graph
            for (int i = 0; i < Vertices; i++)
            {
                // checks if the current vertex is greater than (k edges)
                if (adjacent_list[i].Count > k)
                {
                    for (int j = 0; j < Vertices; j++)
                    {
                        if (adjacent_list[j].Count > k)
                        {
                            // we remove the edge between i and j
                            remove_Edge(i, j);
                        }
                    }
                    // we remove the current vertex from the Tops list
                    if (Tops.Contains(i))
                    {
                        Tops.Remove(i);
                    }
                    // out of for-loop to avoid looping forever
                    break;
                }
            }
```

```
    }

    public void PendantIncrement()
    {
        // iterate through the vertices of the graph
        for (int i = 0; i < Vertices; i++)
        {
            // checks if the current vertex is less than 1
            if (adjacent_list[i].Count > 1)
            {
                for (int j = 0; j < Vertices; j++)
                {
                    if (adjacent_list[j].Count > 1)
                    {
                        // we remove the edge between i and j
                        remove_Edge(i, j);
                    }
                }
                // we add the current vertex to the Pendants list
                if (!Pendants.Contains(i))
                {
                    Pendants.Add(i);
                }
                // out of for-loop to avoid looping forever
                break;
            }
        }
    }

    public void PendantDecrement()
    {
        // iterate through the vertices of the graph
        for (int i = 0; i < Vertices; i++)
        {
            // checks if the current vertex is equal to 1
            if (adjacent_list[i].Count == 1)
            {
                for (int j = 0; j < Vertices; j++)
                {
                    if (adjacent_list[j].Count == 1)
                    {
                        // we add new edge between i and j
                        add_edge(i, j);
```

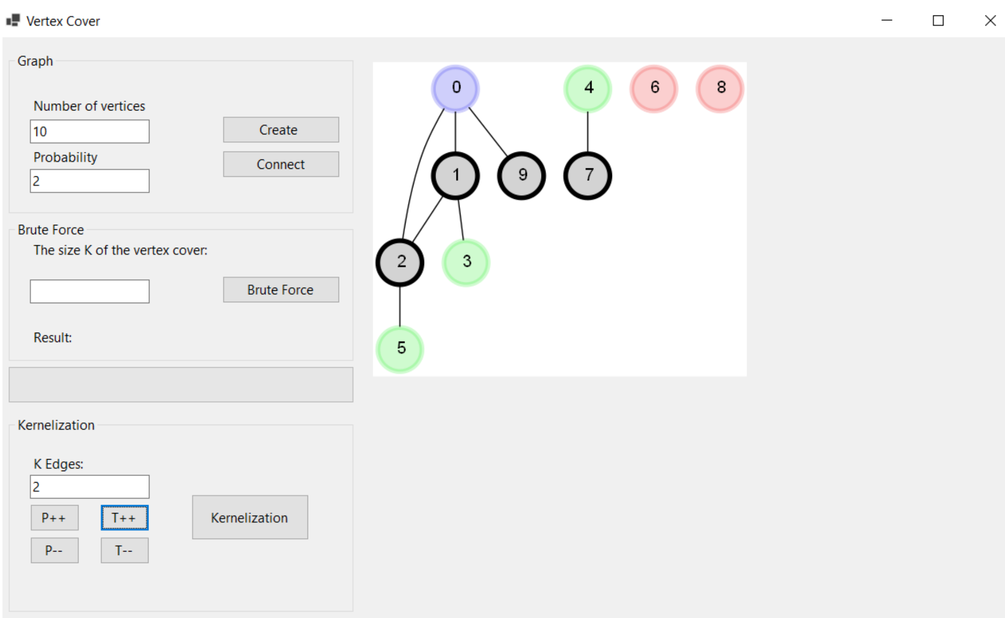
```
                    }
                }
                // we remove the current vertex from the Pendants list
                if (Pendants.Contains(i))
                {
                    Pendants.Remove(i);
                }
                // out of for-loop to avoid looping forever
                break;
            }
        }

    }
```

```
    // a function to update the color of a vertex
        public void updateColorGraphToFile()
        {
            FileStream fileStream = null;
            StreamWriter streamWriter = null;

            try
            {
                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++)
                {
                    if (Tops.Contains(i))
                        streamWriter.WriteLine("node" + i + "[ label =\" "
+ i + "\" color=\"#4040f040\"]");
                    else if (Pendants.Contains(i))
                        streamWriter.WriteLine("node" + i + "[ label =\" "
+ i + "\" color=\"#40f04040\"]");
                    else if (IsolatedVertices.Contains(i))
```

```
                        streamWriter.WriteLine("node" + i + "[ label =\" "
+ i + "\" color=\"#f0404040\"]");
                    else
                        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("}");

            }
            catch (IOException)
            {
                throw;
            }
            catch (Exception e)
            {
                Console.WriteLine(e.Message);
            }
            finally
            {
                if (streamWriter != null)
                {
                    streamWriter.Close();
                }
            }

        }

        // a function to remove an edge between two vertices
        public bool remove_Edge(int src, int dest)
        {
```

```
            if (adjacent_list[src].Contains(dest) &&
adjacent_list[dest].Contains(src))
            {
                adjacent_list[src].Remove(dest);
                adjacent_list[dest].Remove(src);
                return true;
            }
            return false;
        }
```

Output week 3:



Week 4

In the fourth week, after watching the Udacity videos, we have been tasked to improve the brute force method done in week 2. Now that last week we have implemented the basics for the kernalization process, we can use it to improve the brute force search.

First of all, we need to keep track of every pendant, top and isolated vertex. Therefore, we can create an initial vertex cover that will already have some vertices set to true, namely the tops and the neighbours of pendants).

The main difference between “normal” brute force and “enhanced” brute force is that, in the “enhanced” version, we can remove branches from the tree automatically by using pendants, tops and isolated vertices.

This comes in handy, as for example, in case the vertex cover is larger than the maximum nodes, we can return “False” immediately. Otherwise, we look for pendants, tops and isolated vertices. For example, if we have 0/1 then we have either a pendant or an isolated vertex, so we can automatically mark it as false and resume the algorithm. If we have a vertex which is 2, then we know for sure that it is part of the vertex cover and we don’t have to check the false branches of the tree anymore.

Below the actual implementation, for this week, can be found.

```
// function used to keep track of the pendants and tops
private int[] preprocess()
        {
            graph.kernelization(Convert.ToInt32(txt_bruteForce.Text));
            int[] cover = new int[graph.Vertices];

            foreach (int top in graph.Tops)
            {
                cover[top] = 2;
            }

            foreach (int pendant in graph.Pendants)
            {
                cover[graph.get_adjacent_list()[pendant][0]] = 2;
            }

            return cover;
        }
```

```
// same logic as in the initial brute force only that now we use pendants,
// tops and isolated vertices and we are not checking every vertex as we
// did before, which in the end results in better performance.
public bool enhanced_brute_force(Graph g, int[] cover, int n, int i, int k)
        {
            if (k > n)
            {
                return false;
            }

            if (!g.IsOkVertex)
            {
                return true;
            }

            if (i == n)
            {
                return g.ValidateSmart(cover, n, k);
            }

            else
            {
                if (cover[i] == 0 || cover[i] == 1)
                {
                    if (g.Pendants.Contains(i) ||
g.IsolatedVertices.Contains(i))
                    {
                        cover[i] = 0;
                        enhanced_brute_force(g, cover, n, i + 1, k);
                        if (!g.IsOkVertex)
                        {
                            return true;
                        }
                    }
                    else
                    {
                        cover[i] = 0;
                        enhanced_brute_force(g, cover, n, i + 1, k);
                        if (!g.IsOkVertex)
                        {
                            return true;
                        }
```

```
                        cover[i] = 1;
                        enhanced_brute_force(g, cover, n, i + 1, k);
                        if (!g.IsOkVertex)
                        {
                            return true;
                        }
                    }
                }
                else if (cover[i] == 2)
                {
                    enhanced_brute_force(g, cover, n, i + 1, k);
                    if (!g.IsOkVertex)
                    {
                        return true;
                    }
                }
            }
            return false;
        }
```

Output week 4: