Team:

Andreea Matei

Florin Deleanu

Programming Assignment

Data structures & algorithms

Fontys
University of Applied sciences

Table of contents:

1. OVERVIEW

This section provides an extremely useful insight into the overall organization of the code as well as introduces the strategy chosen for the algorithm behind.

TOPICS:

1.1 Code Structure

The architecture of the whole code is based as much as possible on object-oriented principles, thus trying to reach the goal of a piece of software easy to integrate, maintain and especially to extend.
The logic behind the game is in the Algorithm class, which manages and performs the algorithms on some given graphs, in other words, “pulling the strings of the show”. This class contains the “methods” needed mostly for the implementation of the famous problem: “Vertex Cover”. Besides other additional information, the methods receive as parameters the graph G on which we want to perform the operations. In turn, the Graph class contains all the necessary elements that will prove themselves to be of great help later when initializing the graph for the algorithms and not only. The Graph class also includes some other additional functions that are called when running an algorithm from the Algorithm class.
A better overview of the whole structure can be seen below in Fig. 1.1.

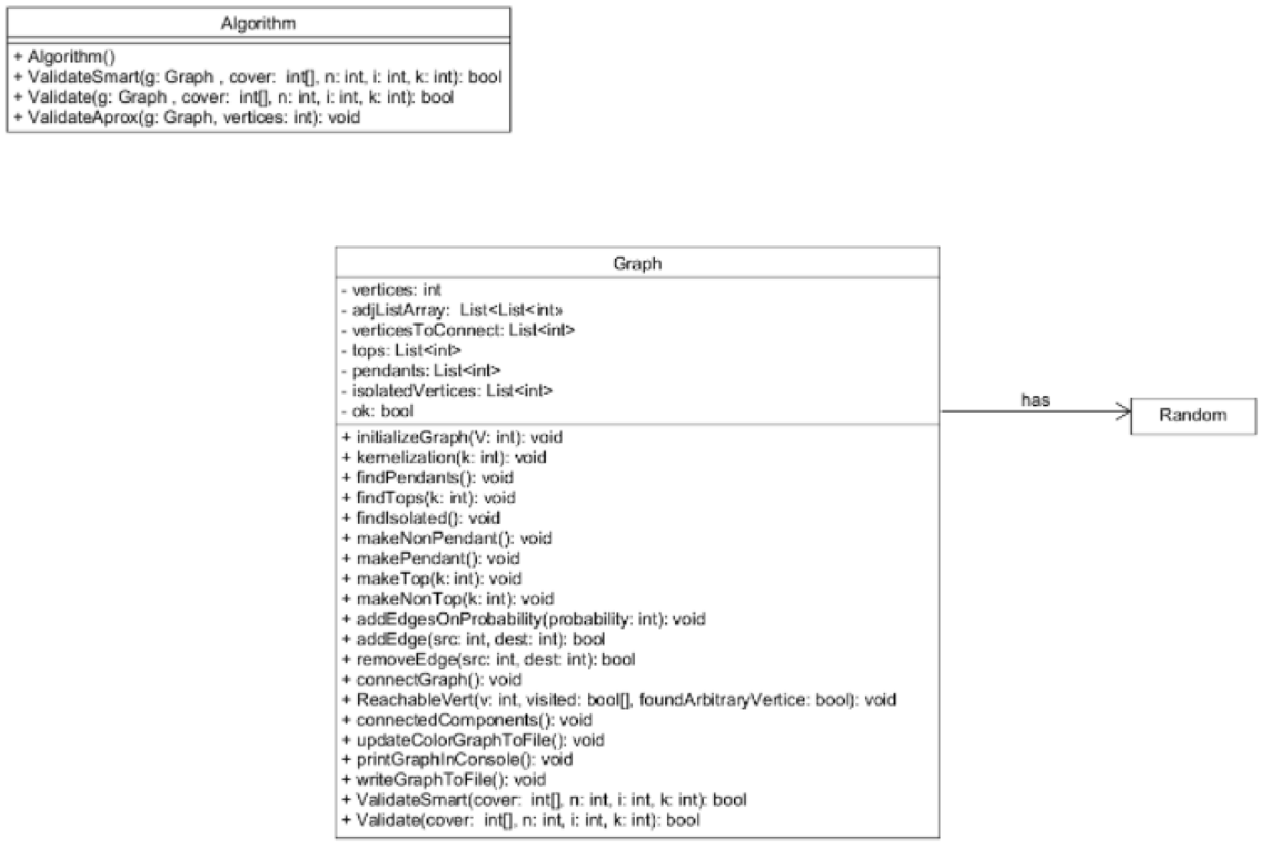
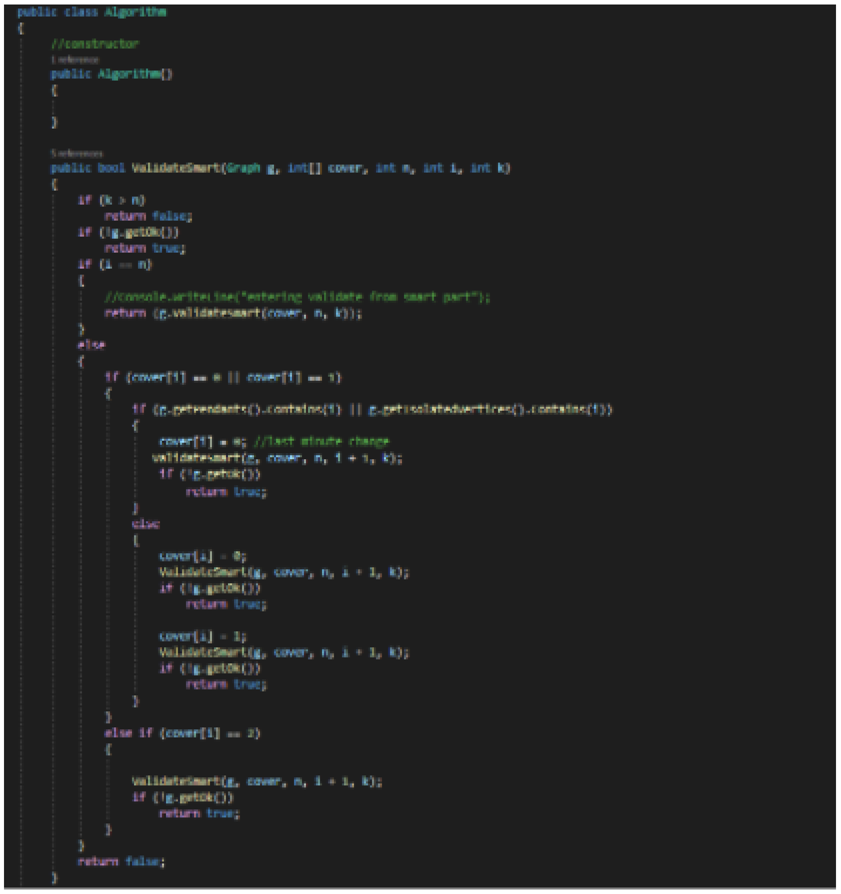

Algorithm
+ Algorithm()
+ ValidateSmart(g: Graph , cover: int[], n: int, i: int, k: int): bool
+ Validate(g: Graph , cover: int[], n: int, i: int, k: int): bool
+ ValidateAprox(g: Graph, vertices: int): void
Graph
- vertices: int
- adjListArray: List<List<int>>
- verticesToConnect: List<int>
- tops: List<int>
- pendants: List<int>
- isolatedVertices: List<int>
- ok: bool
+ initializeGraph(V: int): void
+ kernelization(k: int): void
+ findPendants(): void
+ findTops(k: int): void
+ findIsolated(): void
+ makeNonPendant(): void
+ makePendant(): void
+ makeTop(k: int): void
+ makeNonTop(k: int): void
+ addEdgesOnProbability(probability: int): void
+ addEdge(src: int, dest: int): bool
+ removeEdge(src: int, dest: int): bool
+ connectGraph(): void
+ ReachableVert(v: int, visited: bool[], foundArbitraryVertice: bool): void
+ connectedComponents(): void
+ updateColorGraphToFile(): void
+ printGraphInConsole(): void
+ writeGraphToFile(): void
+ ValidateSmart(cover: int[], n: int, i: int, k: int): bool
+ Validate(cover: int[], n: int, i: int, k: int): bool
has
Random

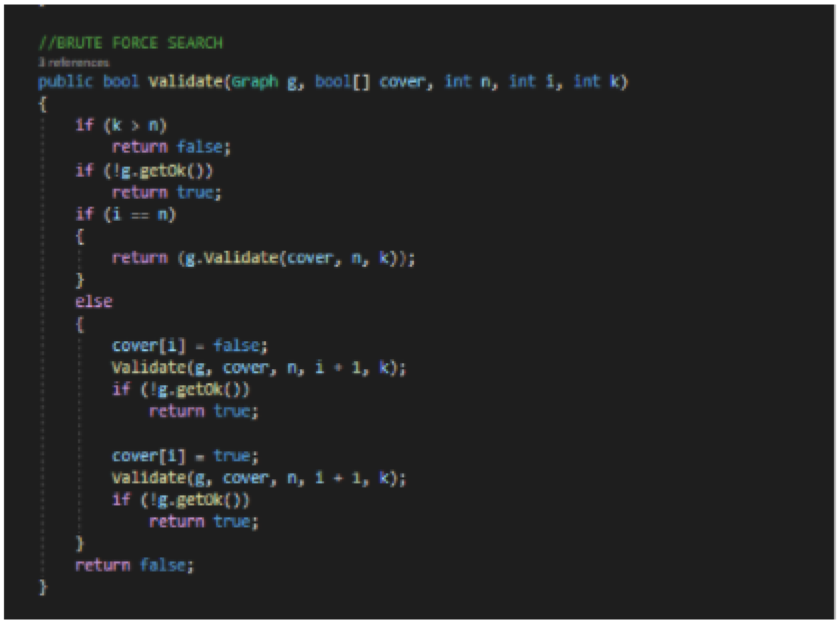
Fig. 1.1

The Algorithm class can be easier seen below:



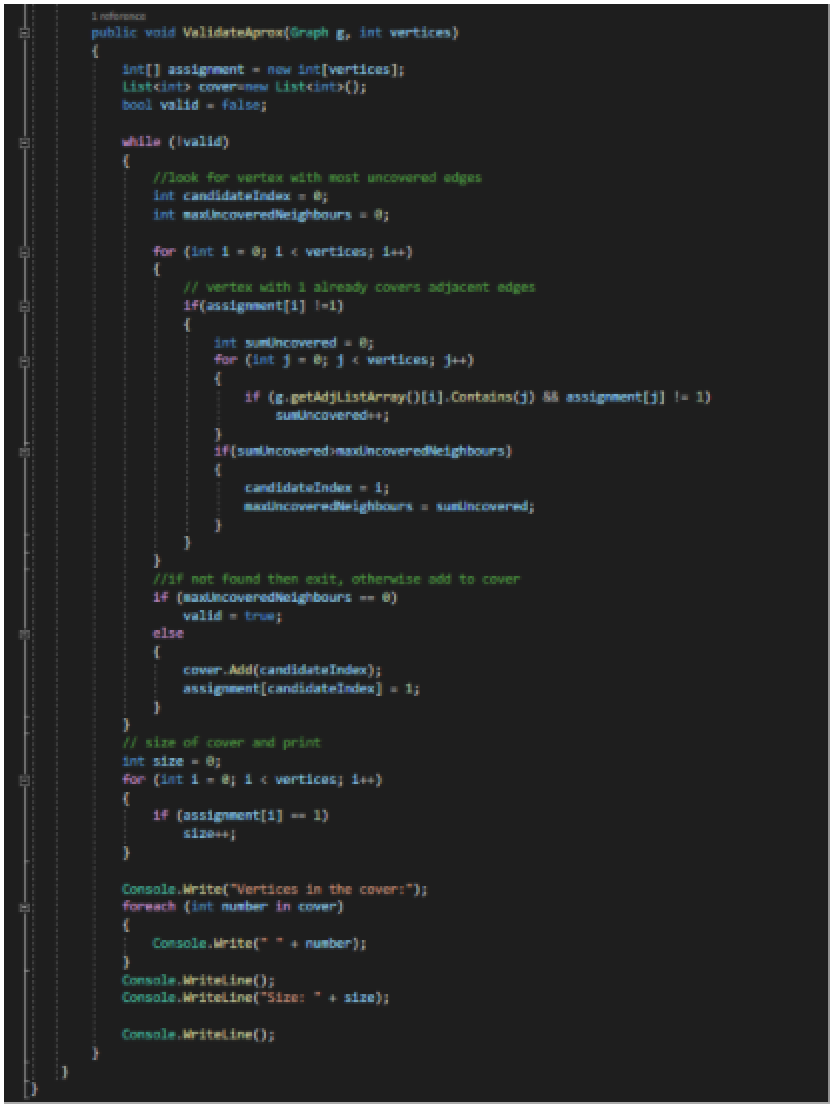
```
public class Algorithm
{
    //constructor
    1 reference
    public Algorithm()
    {

    }

    5 references
    public bool ValidateSmart(Graph g, int[] cover, int n, int i, int k)
    {
        if (k > n)
            return false;
        if (!g.getOk())
            return true;
        if (i == n)
        {
            //console.writeLine("entering validate from smart part");
            return (g.validatesmart(cover, n, k));
        }
        else
        {
            if (cover[i] == 0 || cover[i] == 1)
            {
                if (g.getPendants().contains(i) || g.getIsolatedVertices().contains(i))
                {
                    cover[i] = 0; //last minute change
                   validatesmart(g, cover, n, i + 1, k);
                    if (!g.getOk())
                        return true;
                }
                else
                {
                    cover[i] = 0;
                    ValidateSmart(g, cover, n, i + 1, k);
                    if (!g.getOk())
                        return true;

                    cover[i] = 1;
                    ValidateSmart(g, cover, n, i + 1, k);
                    if (!g.getOk())
                        return true;
                }
            }
            else if (cover[i] == 2)
            {

                ValidateSmart(g, cover, n, i + 1, k);
                if (!g.getOk())
                    return true;
            }
        }
        return false;
    }
```

Fig. 1.2



```
//BRUTE FORCE SEARCH
3 references
public bool Validate(Graph g, bool[] cover, int n, int i, int k)
{
    if (k > n)
        return false;
    if (!g.getOk())
        return true;
    if (i == n)
    {
        return (g.Validate(cover, n, k));
    }
    else
    {
        cover[i] = false;
        Validate(g, cover, n, i + 1, k);
        if (!g.getOk())
            return true;

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

Fig. 1.3



```
        1 reference
        public void ValidateAprox(Graph g, int vertices)
        {
            int[] assignment = new int[vertices];
            List<int> cover=new List<int>();
            bool valid = false;

            while (!valid)
            {
                //look for vertex with most uncovered edges
                int candidateIndex = 0;
                int maxUncoveredNeighbours = 0;

                for (int i = 0; i < vertices; i++)
                {
                    // vertex with 1 already covers adjacent edges
                    if(assignment[i] !=1)
                    {
                        int sumUncovered = 0;
                        for (int j = 0; j < vertices; j++)
                        {
                            if (g.getAdjListArray()[i].Contains(j) && assignment[j] != 1)
                                sumUncovered++;
                        }
                        if(sumUncovered>maxUncoveredNeighbours)
                        {
                            candidateIndex = i;
                            maxUncoveredNeighbours = sumUncovered;
                        }
                    }
                }
                //if not found then exit, otherwise add to cover
                if (maxUncoveredNeighbours == 0)
                    valid = true;
                else
                {
                    cover.Add(candidateIndex);
                    assignment[candidateIndex] = 1;
                }
            }
            // size of cover and print
            int size = 0;
            for (int i = 0; i < vertices; i++)
            {
                if (assignment[i] == 1)
                    size++;
            }

            Console.Write("Vertices in the cover:");
            foreach (int number in cover)
            {
                Console.Write(" " + number);
            }
            Console.WriteLine();
            Console.WriteLine("Size: " + size);

            Console.WriteLine();
        }
    }
}
```

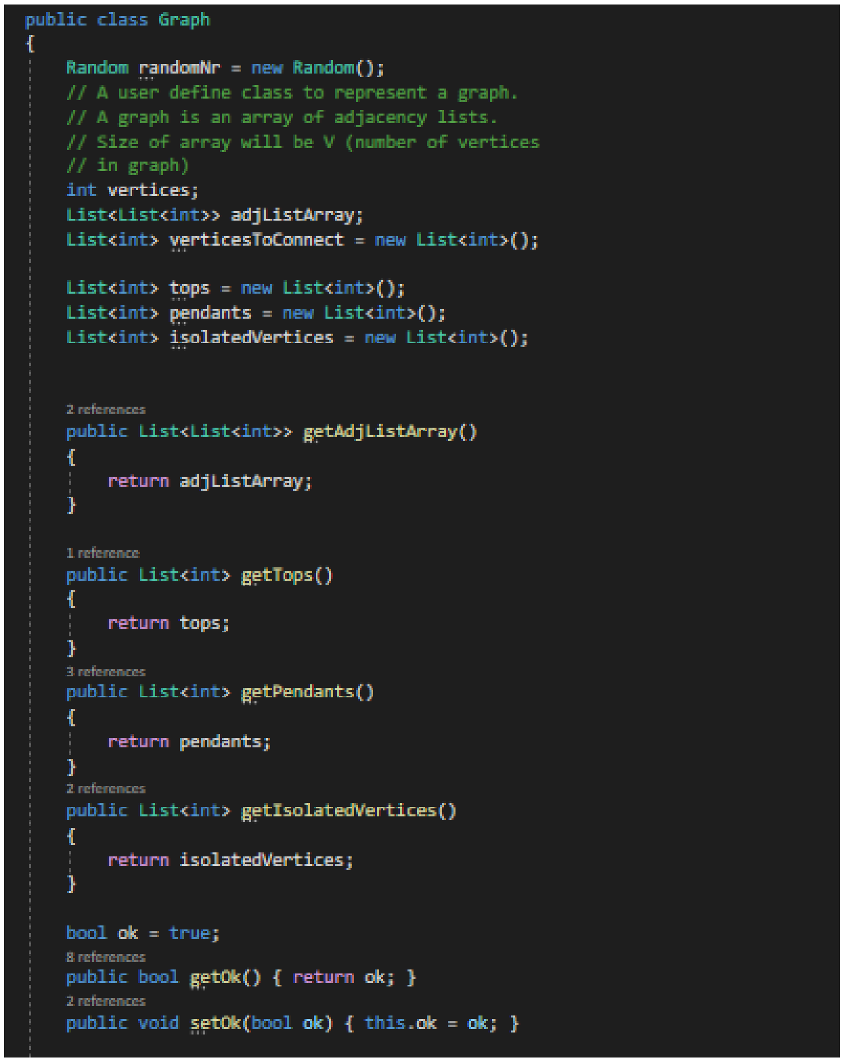
Fig. 1.4

The class contains the "brains" behind 3 main algorithms for the famous problem 'Vertex Cover':

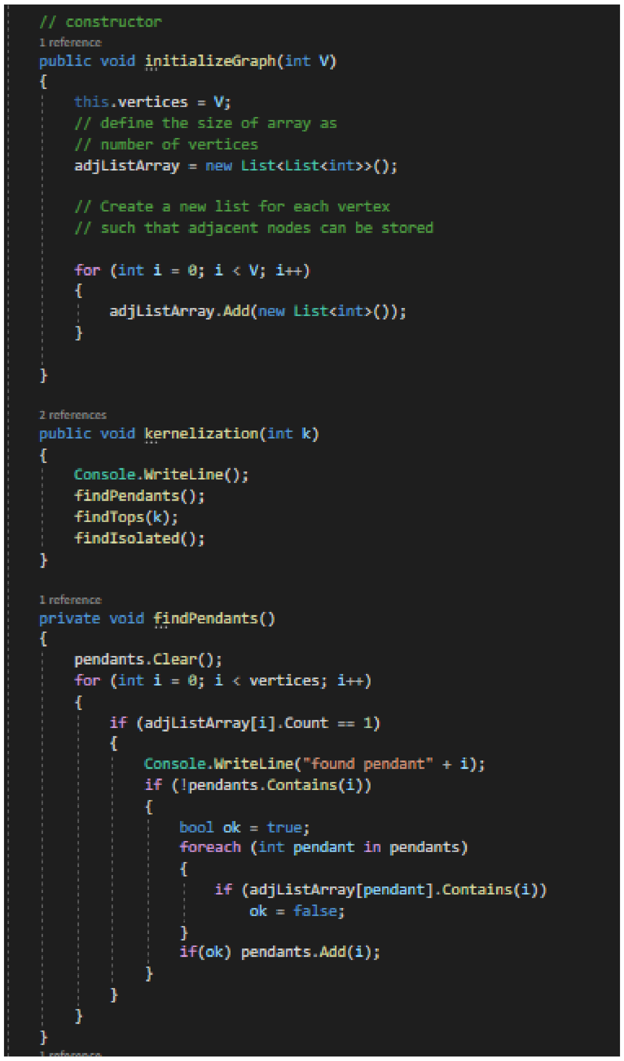
- The Brute Force approach (Fig. 1.3)
- The Enhanced Brute Force approach (Fig. 1.2)
- The Approximation Algorithm approach (Fig. 1.4)

The above algorithms will be later discussed in detail in the following sub-section: 1.2 Strategy.

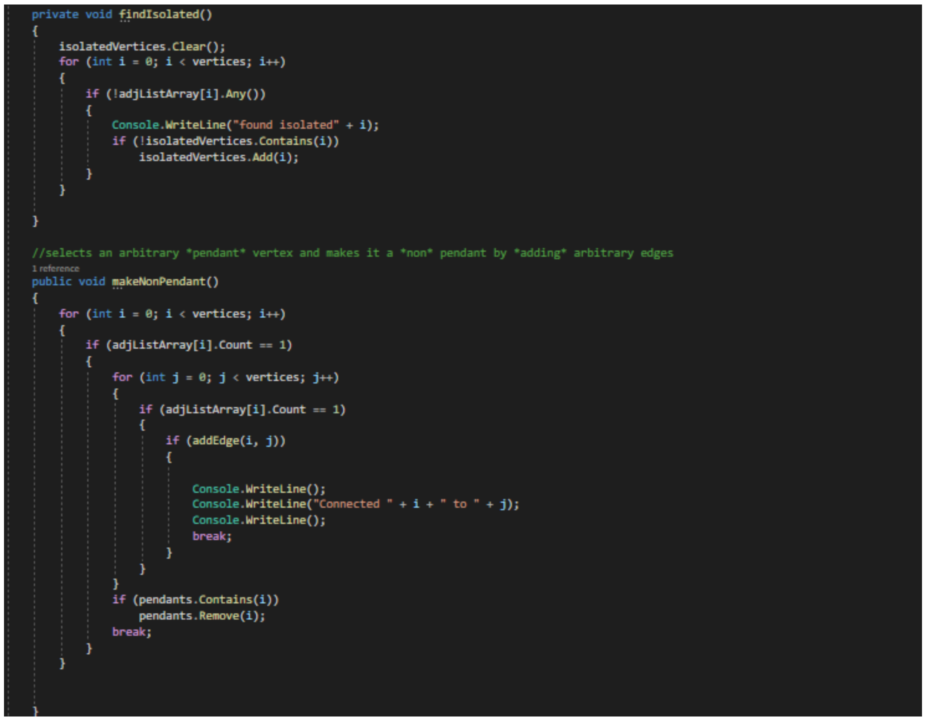
Equally important, the Graph class with all its crucial elements is shown:



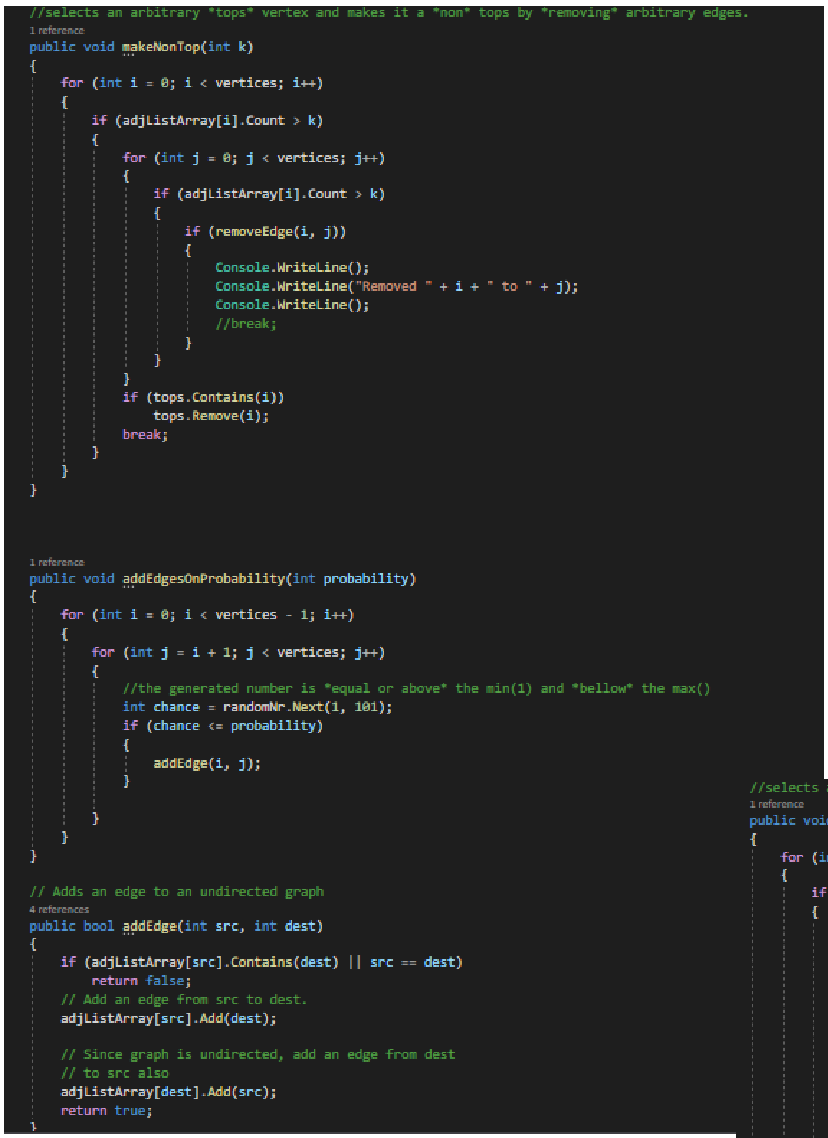
```
public class Graph
{
    Random randomNr = new Random();
    // A user define class to represent a graph.
    // A graph is an array of adjacency lists.
    // Size of array will be V (number of vertices
    // in graph)
    int vertices;
    List<List<int>> adjListArray;
    List<int> verticesToConnect = new List<int>();

    List<int> tops = new List<int>();
    List<int> pendants = new List<int>();
    List<int> isolatedVertices = new List<int>();


    2 references
    public List<List<int>> getAdjListArray()
    {
        return adjListArray;
    }

    1 reference
    public List<int> getTops()
    {
        return tops;
    }
    3 references
    public List<int> getPendants()
    {
        return pendants;
    }
    2 references
    public List<int> getIsolatedVertices()
    {
        return isolatedVertices;
    }

    bool ok = true;
    8 references
    public bool getOk() { return ok; }
    2 references
    public void setOk(bool ok) { this.ok = ok; }
```



```
// constructor
1 reference
public void initializeGraph(int V)
{
    this.vertices = V;
    // define the size of array as
    // number of vertices
    adjListArray = new List<List<int>>();

    // Create a new list for each vertex
    // such that adjacent nodes can be stored

    for (int i = 0; i < V; i++)
    {
        adjListArray.Add(new List<int>());
    }

}

2 references
public void kernelization(int k)
{
    Console.WriteLine();
    findPendants();
    findTops(k);
    findIsolated();
}

1 reference
private void findPendants()
{
    pendants.Clear();
    for (int i = 0; i < vertices; i++)
    {
        if (adjListArray[i].Count == 1)
        {
            Console.WriteLine("found pendant" + i);
            if (!pendants.Contains(i))
            {
                bool ok = true;
                foreach (int pendant in pendants)
                {
                    if (adjListArray[pendant].Contains(i))
                        ok = false;
                }
                if(ok) pendants.Add(i);
            }
        }
    }
}
```



```
private void findIsolated()
{
    isolatedVertices.Clear();
    for (int i = 0; i < vertices; i++)
    {
        if (!adjListArray[i].Any())
        {
            Console.WriteLine("found isolated" + i);
            if (!isolatedVertices.Contains(i))
                isolatedVertices.Add(i);
        }
    }

}

//selects an arbitrary *pendant* vertex and makes it a *non* pendant by *adding* arbitrary edges
1 reference
public void makeNonPendant()
{
    for (int i = 0; i < vertices; i++)
    {
        if (adjListArray[i].Count == 1)
        {
            for (int j = 0; j < vertices; j++)
            {
                if (adjListArray[i].Count == 1)
                {
                    if (addEdge(i, j))
                    {

                        Console.WriteLine();
                        Console.WriteLine("Connected " + i + " to " + j);
                        Console.WriteLine();
                        break;
                    }
                }
            }
            if (pendants.Contains(i))
                pendants.Remove(i);
            break;
        }
    }


}
```



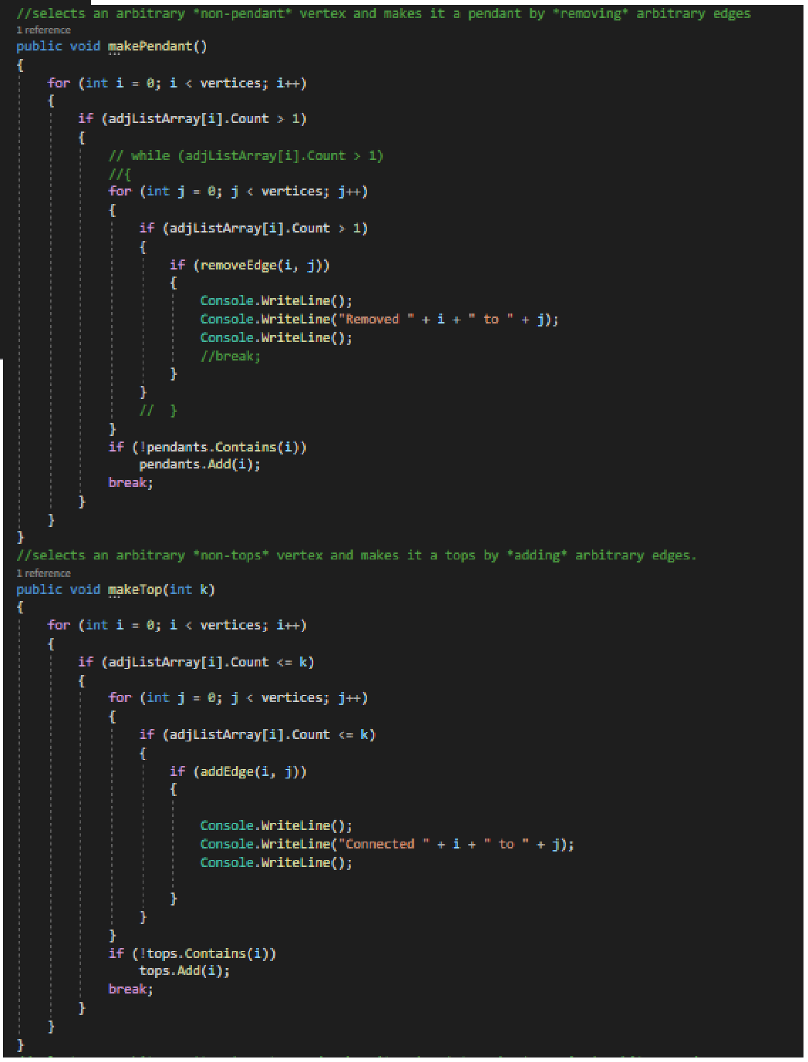
```
//selects an arbitrary *tops* vertex and makes it a *non* tops by *removing* arbitrary edges.
1 reference
public void makeNonTop(int k)
{
    for (int i = 0; i < vertices; i++)
    {
        if (adjListArray[i].Count > k)
        {
            for (int j = 0; j < vertices; j++)
            {
                if (adjListArray[i].Count > k)
                {
                    if (removeEdge(i, j))
                    {
                        Console.WriteLine();
                        Console.WriteLine("Removed " + i + " to " + j);
                        Console.WriteLine();
                        //break;
                    }
                }
            }
            if (tops.Contains(i))
                tops.Remove(i);
            break;
        }
    }
}



1 reference
public void addEdgesOnProbability(int probability)
{
    for (int i = 0; i < vertices - 1; i++)
    {
        for (int j = i + 1; j < vertices; j++)
        {
            //the generated number is *equal or above* the min(1) and *bellow* the max()
            int chance = randomNr.Next(1, 101);
            if (chance <= probability)
            {
                addEdge(i, j);
            }


        }
    }
}

// Adds an edge to an undirected graph
4 references
public bool addEdge(int src, int dest)
{
    if (adjListArray[src].Contains(dest) || src == dest)
        return false;
    // Add an edge from src to dest.
    adjListArray[src].Add(dest);

    // Since graph is undirected, add an edge from dest
    // to src also
    adjListArray[dest].Add(src);
    return true;
```

This class contains extremely useful methods that make not only the creation of a graph possible, but also the algorithms from the previously presented class work through various additional functions, such as:

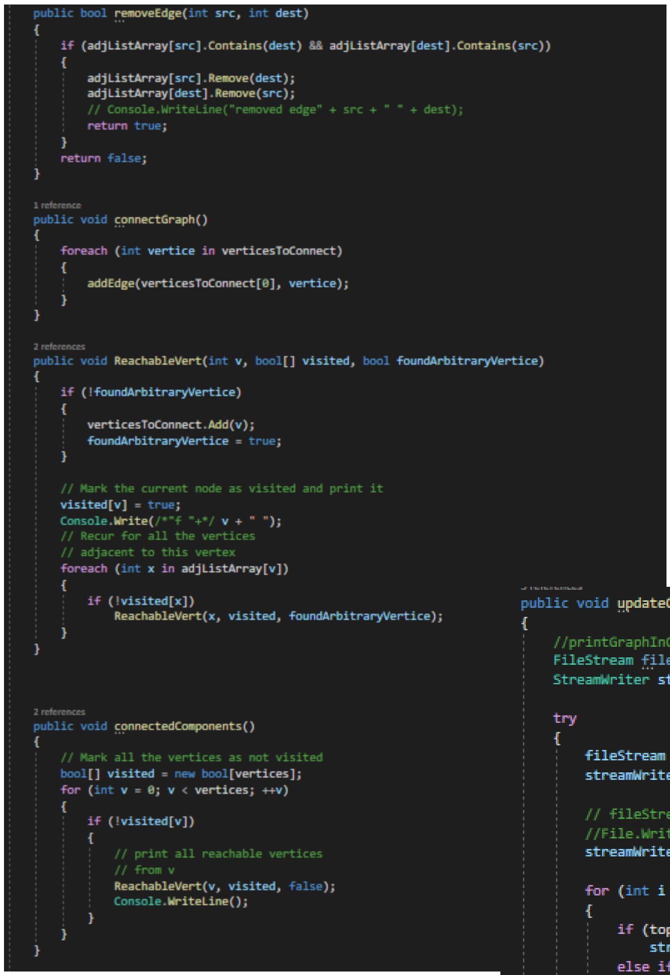
- Validate()
- VaidateSmart()

Moreover, the class presents also functions that make sure the graph is always updated:

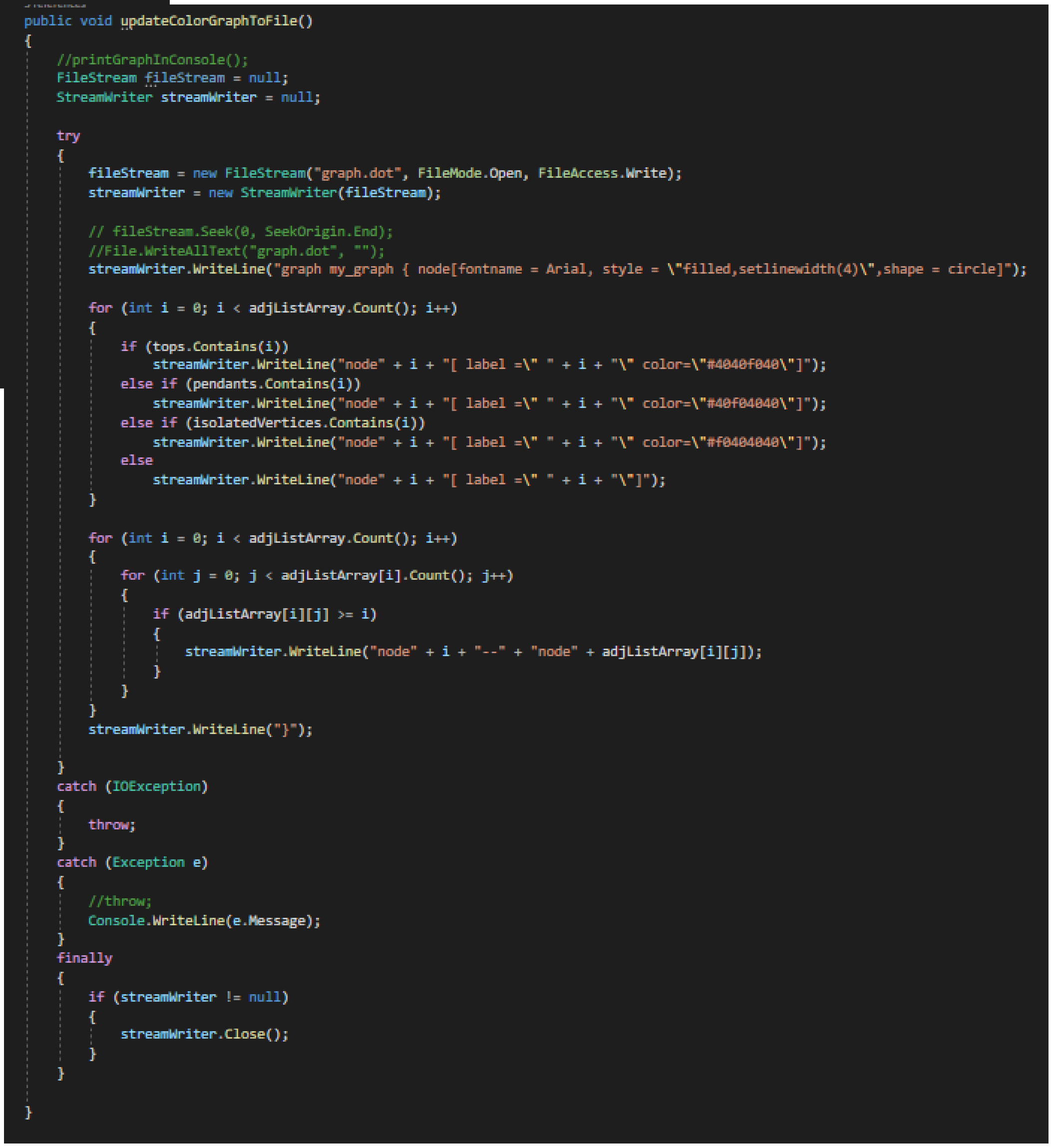
- updateColorGraphToFile()
- writeGraphToFile()



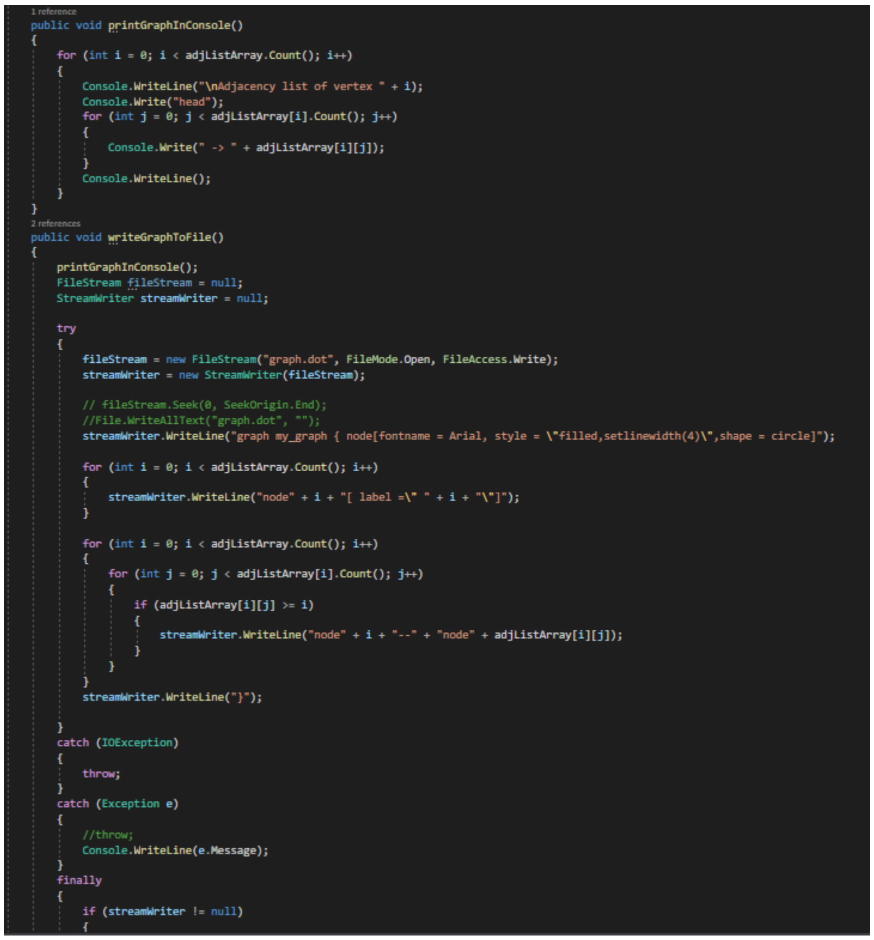
```
//selects an arbitrary *non-pendant* vertex and makes it a pendant by *removing* arbitrary edges
1 reference
public void makePendant()
{
    for (int i = 0; i < vertices; i++)
    {
        if (adjListArray[i].Count > 1)
        {
            // while (adjListArray[i].Count > 1)
            //{
            for (int j = 0; j < vertices; j++)
            {
                if (adjListArray[i].Count > 1)
                {
                    if (removeEdge(i, j))
                    {
                        Console.WriteLine();
                        Console.WriteLine("Removed " + i + " to " + j);
                        Console.WriteLine();
                        //break;
                    }
                }
                //  }
            }
            if (!pendants.Contains(i))
                pendants.Add(i);
            break;
        }
    }
}
//selects an arbitrary *non-tops* vertex and makes it a tops by *adding* arbitrary edges.
1 reference
public void makeTop(int k)
{
    for (int i = 0; i < vertices; i++)
    {
        if (adjListArray[i].Count <= k)
        {
            for (int j = 0; j < vertices; j++)
            {
                if (adjListArray[i].Count <= k)
                {
                    if (addEdge(i, j))
                    {

                        Console.WriteLine();
                        Console.WriteLine("Connected " + i + " to " + j);
                        Console.WriteLine();

                    }
                }
            }
            if (!tops.Contains(i))
                tops.Add(i);
            break;
        }
    }
}
```



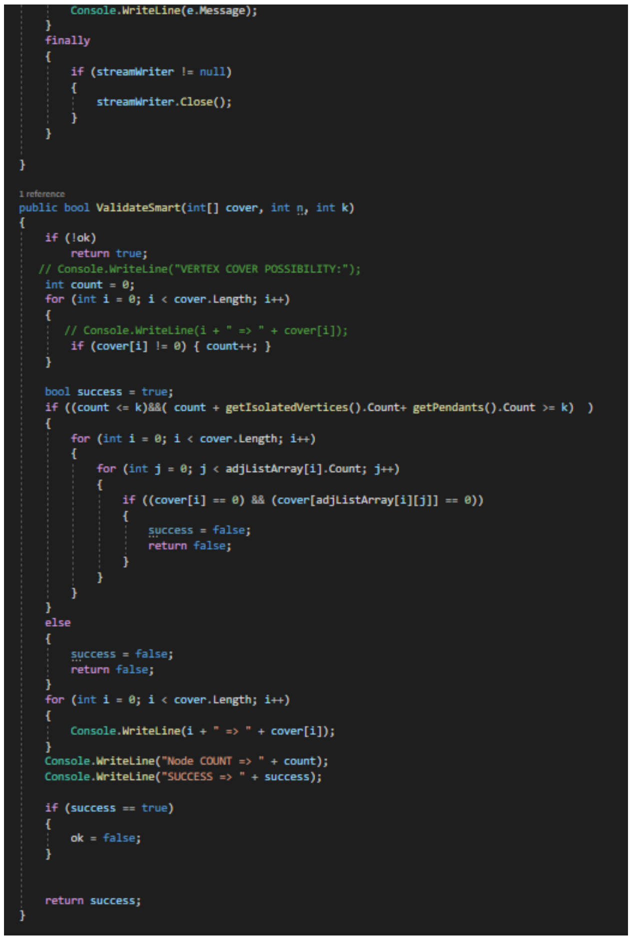
```
public bool removeEdge(int src, int dest)
{
    if (adjListArray[src].Contains(dest) && adjListArray[dest].Contains(src))
    {
        adjListArray[src].Remove(dest);
        adjListArray[dest].Remove(src);
        // Console.WriteLine("removed edge" + src + " " + dest);
        return true;
    }
    return false;
}

1 reference
public void connectGraph()
{
    foreach (int vertice in verticesToConnect)
    {
        addEdge(verticesToConnect[0], vertice);
    }
}

2 references
public void ReachableVert(int v, bool[] visited, bool foundArbitraryVertice)
{
    if (!foundArbitraryVertice)
    {
        verticesToConnect.Add(v);
        foundArbitraryVertice = true;
    }

    // Mark the current node as visited and print it
    visited[v] = true;
    Console.Write(/*"f "+*/ v + " ");
    // Recur for all the vertices
    // adjacent to this vertex
    foreach (int x in adjListArray[v])
    {
        if (!visited[x])
            ReachableVert(x, visited, foundArbitraryVertice);
    }
}


2 references
public void connectedComponents()
{
    // Mark all the vertices as not visited
    bool[] visited = new bool[vertices];
    for (int v = 0; v < vertices; ++v)
    {
        if (!visited[v])
        {
            // print all reachable vertices
            // from v
            ReachableVert(v, visited, false);
            Console.WriteLine();
        }
    }
}
```



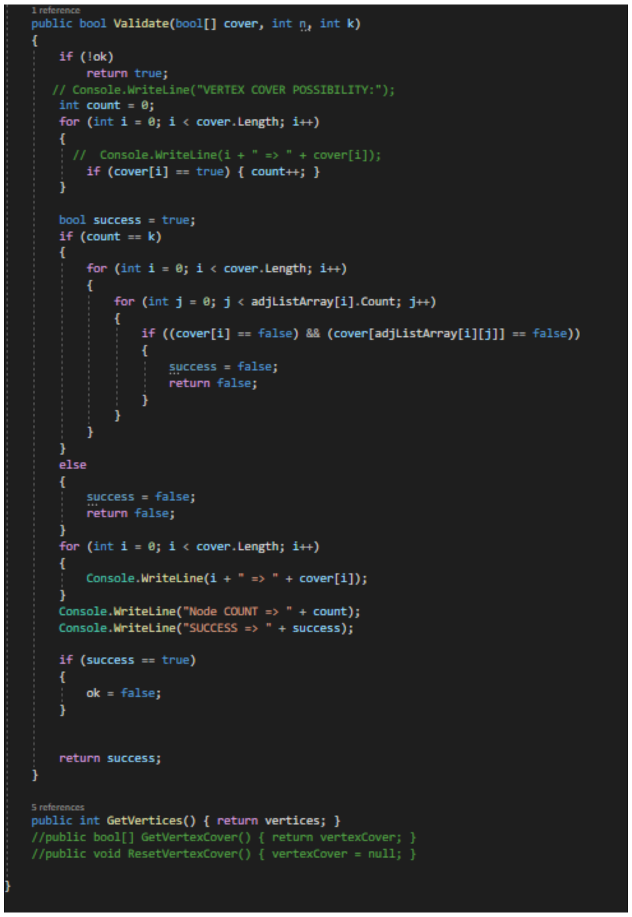
```
public void updateColorGraphToFile()
{
    //printGraphInConsole();
    FileStream fileStream = null;
    StreamWriter streamWriter = null;

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

        // fileStream.Seek(0, SeekOrigin.End);
        //File.WriteAllText("graph.dot", "");
        streamWriter.WriteLine("graph my_graph { node[fontname = Arial, style = \"filled,setlinewidth(4)\",shape = circle]");

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

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

}
```



```
    1 reference
    public void printGraphInConsole()
    {
        for (int i = 0; i < adjListArray.Count(); i++)
        {
            Console.WriteLine("\nAdjacency list of vertex " + i);
            Console.Write("head");
            for (int j = 0; j < adjListArray[i].Count(); j++)
            {
                Console.Write(" -> " + adjListArray[i][j]);
            }
            Console.WriteLine();
        }
    }
    2 references
    public void writeGraphToFile()
    {
        printGraphInConsole();
        FileStream fileStream = null;
        StreamWriter streamWriter = null;

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

            // fileStream.Seek(0, SeekOrigin.End);
            //File.WriteAllText("graph.dot", "");
            streamWriter.WriteLine("graph my_graph { node[fontname = Arial, style = \"filled,setlinewidth(4)\",shape = circle]");

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

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

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



```
            Console.WriteLine(e.Message);
        }
        finally
        {
            if (streamWriter != null)
            {
                streamWriter.Close();
            }
        }

    }

    1 reference
    public bool ValidateSmart(int[] cover, int n, int k)
    {
        if (!ok)
            return true;
       // Console.WriteLine("VERTEX COVER POSSIBILITY:");
        int count = 0;
        for (int i = 0; i < cover.Length; i++)
        {
          // Console.WriteLine(i + " => " + cover[i]);
            if (cover[i] != 0) { count++; }
        }

        bool success = true;
        if ((count <= k)&&( count + getIsolatedVertices().Count+ getPendants().Count >= k)  )
        {
            for (int i = 0; i < cover.Length; i++)
            {
                for (int j = 0; j < adjListArray[i].Count; j++)
                {
                    if ((cover[i] == 0) && (cover[adjListArray[i][j]] == 0))
                    {
                        success = false;
                        return false;
                    }
                }
            }
        }
        else
        {
            success = false;
            return false;
        }
        for (int i = 0; i < cover.Length; i++)
        {
            Console.WriteLine(i + " => " + cover[i]);
        }
        Console.WriteLine("Node COUNT => " + count);
        Console.WriteLine("SUCCESS => " + success);

        if (success == true)
        {
            ok = false;
        }


        return success;
    }
```

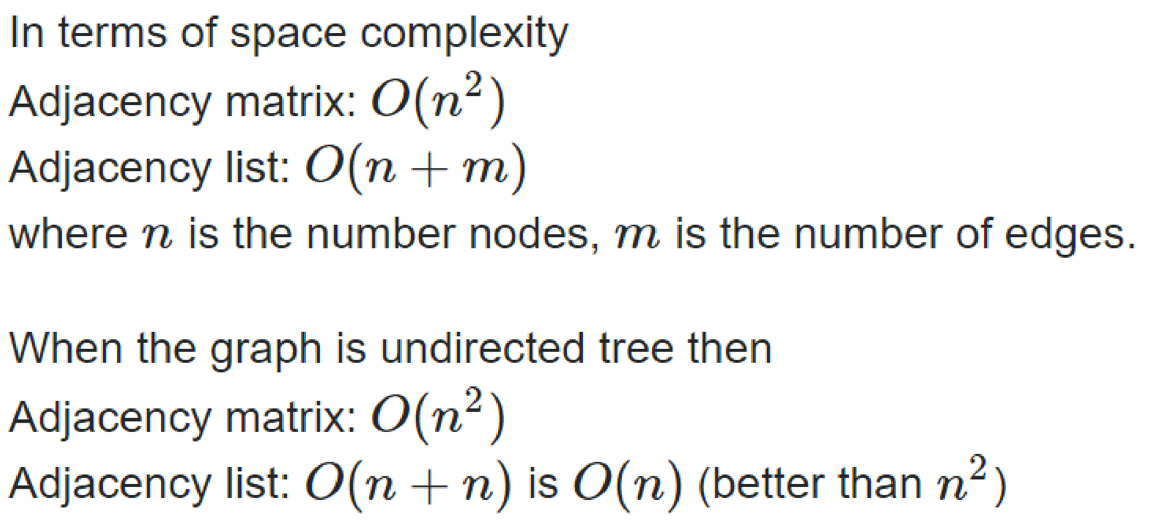


```
    1 reference
    public bool Validate(bool[] cover, int n, int k)
    {
        if (!ok)
            return true;
       // Console.WriteLine("VERTEX COVER POSSIBILITY:");
        int count = 0;
        for (int i = 0; i < cover.Length; i++)
        {
          //  Console.WriteLine(i + " => " + cover[i]);
            if (cover[i] == true) { count++; }
        }

        bool success = true;
        if (count == k)
        {
            for (int i = 0; i < cover.Length; i++)
            {
                for (int j = 0; j < adjListArray[i].Count; j++)
                {
                    if ((cover[i] == false) && (cover[adjListArray[i][j]] == false))
                    {
                        success = false;
                        return false;
                    }
                }
            }
        }
        else
        {
            success = false;
            return false;
        }
        for (int i = 0; i < cover.Length; i++)
        {
            Console.WriteLine(i + " => " + cover[i]);
        }
        Console.WriteLine("Node COUNT => " + count);
        Console.WriteLine("SUCCESS => " + success);

        if (success == true)
        {
            ok = false;
        }


        return success;
    }

    5 references
    public int GetVertices() { return vertices; }
    //public bool[] GetVertexCover() { return vertexCover; }
    //public void ResetVertexCover() { vertexCover = null; }

}
```

1.2 Strategy

Throughout the strategy-setting process, an impressive number of ideas have been brainstormed. This has proved to be very useful in the current choice of a strategy for our algorithms. Of all the generated ideas, the one worth mentioning, on which the whole code is based, is the decision of using adjacency list instead of adjacency matrix.
The reasons behind this choice is the following:

Even though in terms of space complexity, the matrix has better advantage, when it comes to time complexity, an adjacency list might turn out to be more appealing. (*When Are Adjacency Lists or Matrices the Better Choice?*, n.d.)



In terms of space complexity
Adjacency matrix: $O(n^2)$
Adjacency list: $O(n+m)$
where n is the number nodes, m is the number of edges.

When the graph is undirected tree then
Adjacency matrix: $O(n^2)$
Adjacency list: $O(n+n)$ is $O(n)$ (better than n^2)

Therefore, with the help of an adjacency list we might be able to gain more time and score a better performance.

Moreover, as mentioned before, the whole code revolves around 3 major algorithms that target the famous NP-complete problem “Vertex Cover”. A short description of the above strategies will be provided.

The Brute Force Approach

For this approach, we followed the global sketch that was provided to us in the assignment description.
The sketch is the following:

```
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);
      }
}
```

The method implemented following the steps depicted above can be found in the Algorithm class. The function tries recursively all possible combinations until either it finds one that matches the requirements or exhaust all options, suggesting that there is no solution for the given input.

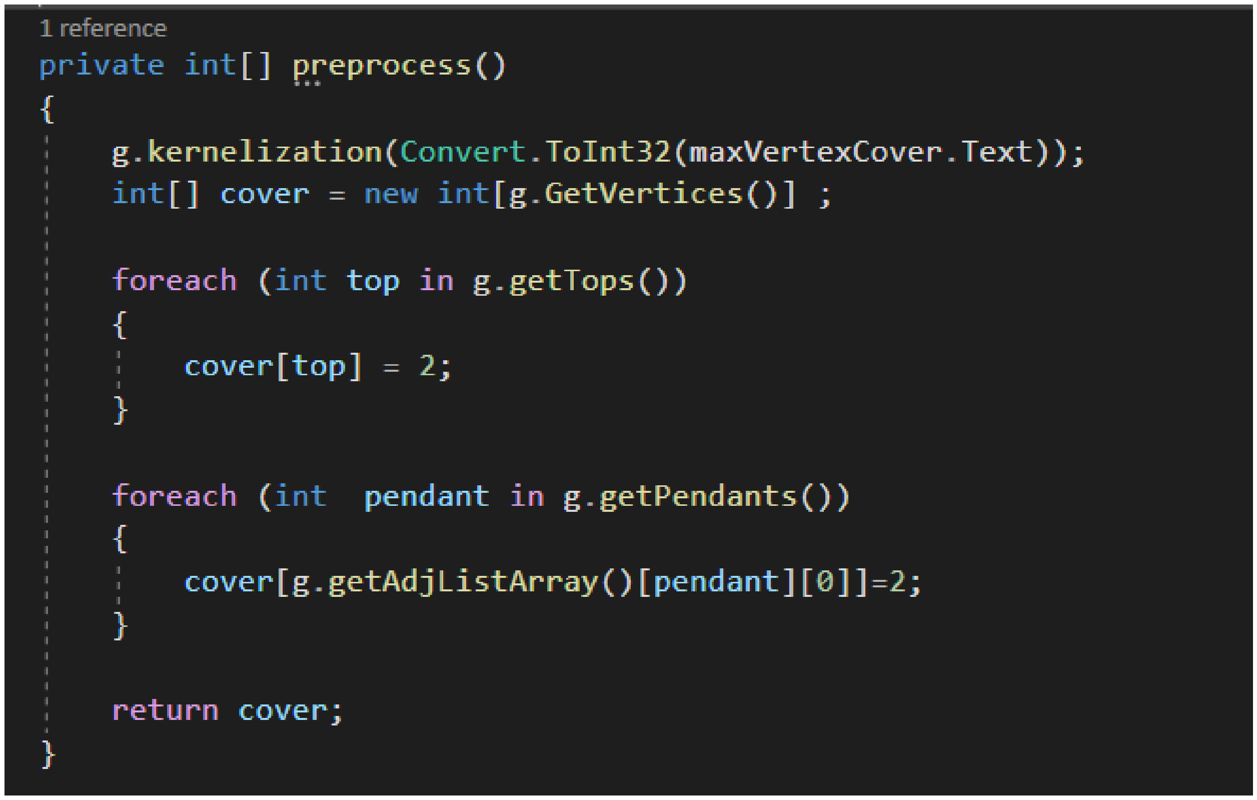
In a way, we can say that the function uses backtracking, thus creating a tree of all possible arrangements.

The Enhanced Brute Force Approach

The enhanced brute force works on the principle of kernelization of the graph for the vertex cover problem. This kernelization has the same logic that is explained on wikipedia but the code implementation is a bit different:

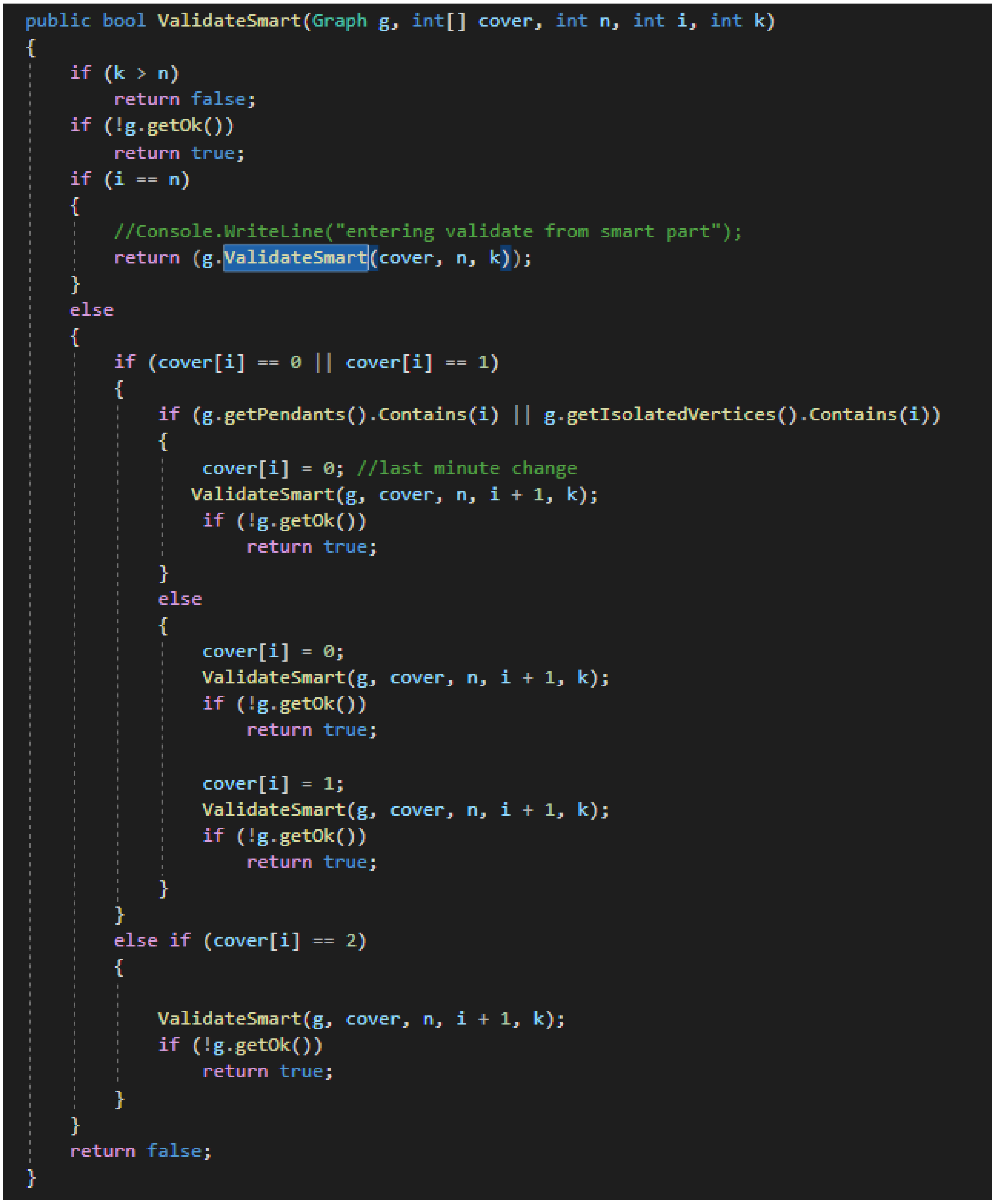
1) We call the preprocess() method. This method begins with the kernelization() method which will keep track of every pendant,top and isolated vertex in our graph. After that we create an initial vertex cover that will already have some vertices set to true(tops and the neighbours of

pendants).



```
1 reference
private int[] preprocess()
{
    g.kernelization(Convert.ToInt32(maxVertexCover.Text));
    int[] cover = new int[g.GetVertices()] ;

    foreach (int top in g.getTops())
    {
        cover[top] = 2;
    }

    foreach (int  pendant in g.getPendants())
    {
        cover[g.getAdjListArray()[pendant][0]]=2;
    }

    return cover;
}
```

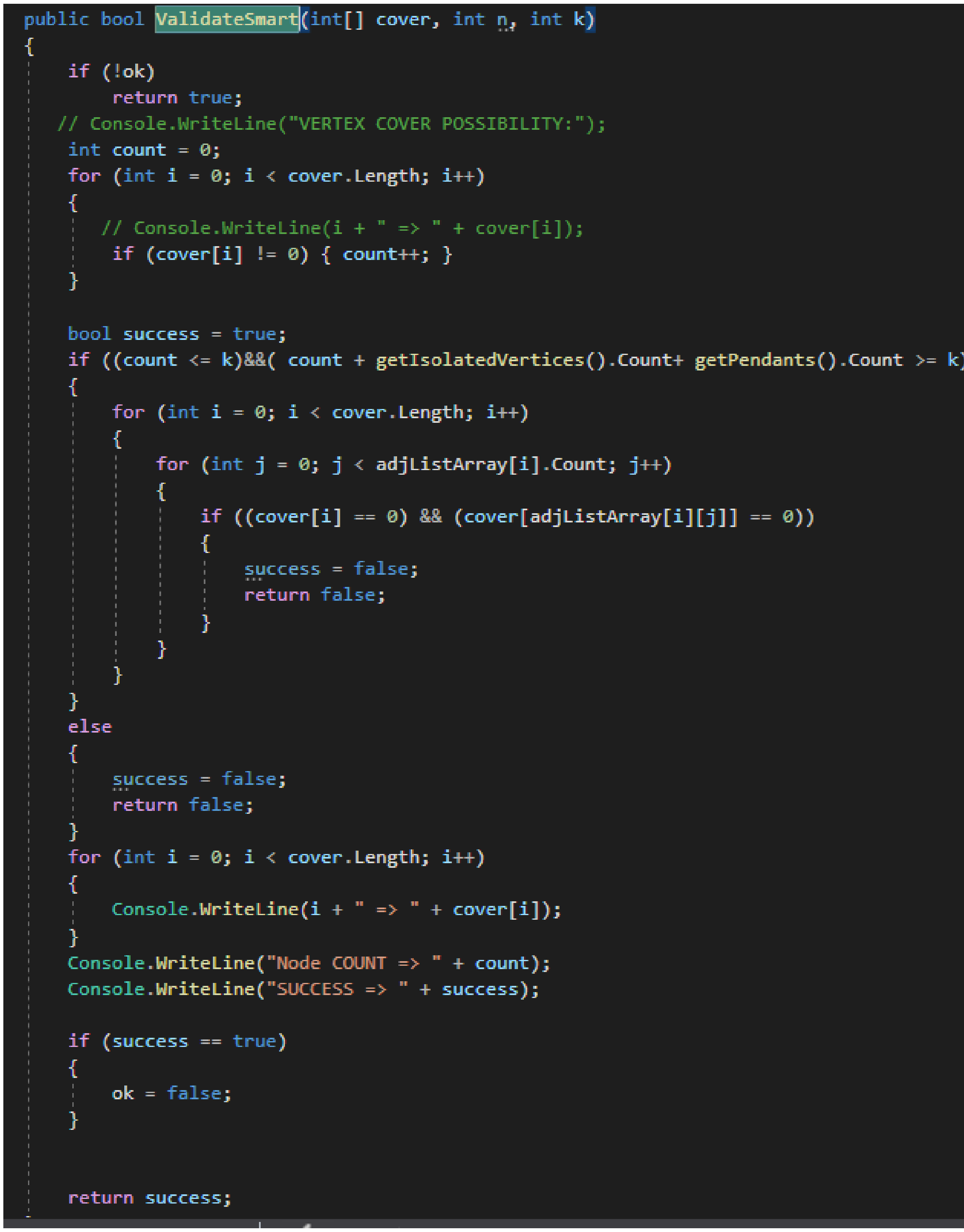
2) The next step is the core of our "smart" algorithm. What differentiates this algorithm from the simple brute force one is that based on our tops,pendants and isolated vertices we can automatically remove branches from our binary tree, thus reducing the exponential complexity. The process is as follows:
   a) We execute the initial smart algorithm and based on some input constraints we can return false from the beginning(for example vertex cover is larger than maximum nodes)
   b) check the initial cover for either a 0/1 or a 2:
      1. case 0/1: If in our graph the vertex we are currently checking is either a pendant or isolated we can automatically mark it as being false and continue the algorithm because they will not be part of the vertex cover. Otherwise we can continue building the normal binary tree with all the possibilities
      2. case 2: If the checked vertex is set as being true from the kernelization process so we know it is part of the vertex cover and we don't have to check the false branches of the binary tree.



```
public bool ValidateSmart(Graph g, int[] cover, int n, int i, int k)
{
    if (k > n)
        return false;
    if (!g.getOk())
        return true;
    if (i == n)
    {
        //Console.WriteLine("entering validate from smart part");
        return (g.ValidateSmart(cover, n, k));
    }
    else
    {
        if (cover[i] == 0 || cover[i] == 1)
        {
            if (g.getPendants().Contains(i) || g.getIsolatedVertices().Contains(i))
            {
                cover[i] = 0; //last minute change
               ValidateSmart(g, cover, n, i + 1, k);
                if (!g.getOk())
                    return true;
            }
            else
            {
                cover[i] = 0;
                ValidateSmart(g, cover, n, i + 1, k);
                if (!g.getOk())
                    return true;

                cover[i] = 1;
                ValidateSmart(g, cover, n, i + 1, k);
                if (!g.getOk())
                    return true;
            }
        }
        else if (cover[i] == 2)
        {

            ValidateSmart(g, cover, n, i + 1, k);
            if (!g.getOk())
                return true;
        }
    }
    return false;
}
```

c) if we finish the tree and are at the end of the execution then we enter a method that checks whether that branch has the best vertex cover
   i) if it has then we print the result and return true

ii) if it doesn't then we return false and continue with the other branches

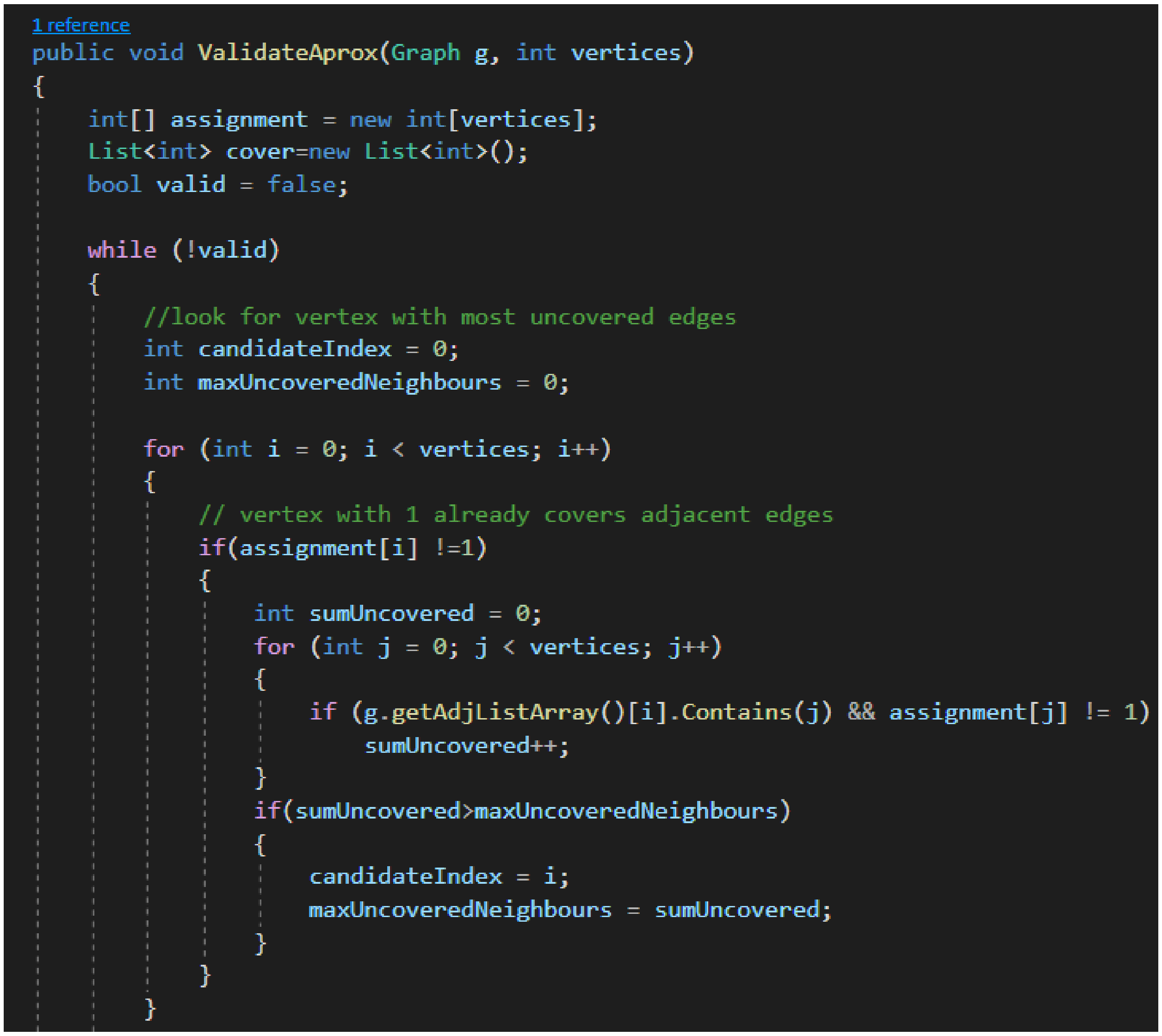


```
public bool ValidateSmart(int[] cover, int n, int k)
{
    if (!ok)
        return true;
   // Console.WriteLine("VERTEX COVER POSSIBILITY:");
    int count = 0;
    for (int i = 0; i < cover.Length; i++)
    {
      // Console.WriteLine(i + " => " + cover[i]);
        if (cover[i] != 0) { count++; }
    }

    bool success = true;
    if ((count <= k)&&( count + getIsolatedVertices().Count+ getPendants().Count >= k
    {
        for (int i = 0; i < cover.Length; i++)
        {
            for (int j = 0; j < adjListArray[i].Count; j++)
            {
                if ((cover[i] == 0) && (cover[adjListArray[i][j]] == 0))
                {
                    success = false;
                    return false;
                }
            }
        }
    }
    else
    {
        success = false;
        return false;
    }
    for (int i = 0; i < cover.Length; i++)
    {
        Console.WriteLine(i + " => " + cover[i]);
    }
    Console.WriteLine("Node COUNT => " + count);
    Console.WriteLine("SUCCESS => " + success);

    if (success == true)
    {
        ok = false;
    }


    return success;
```

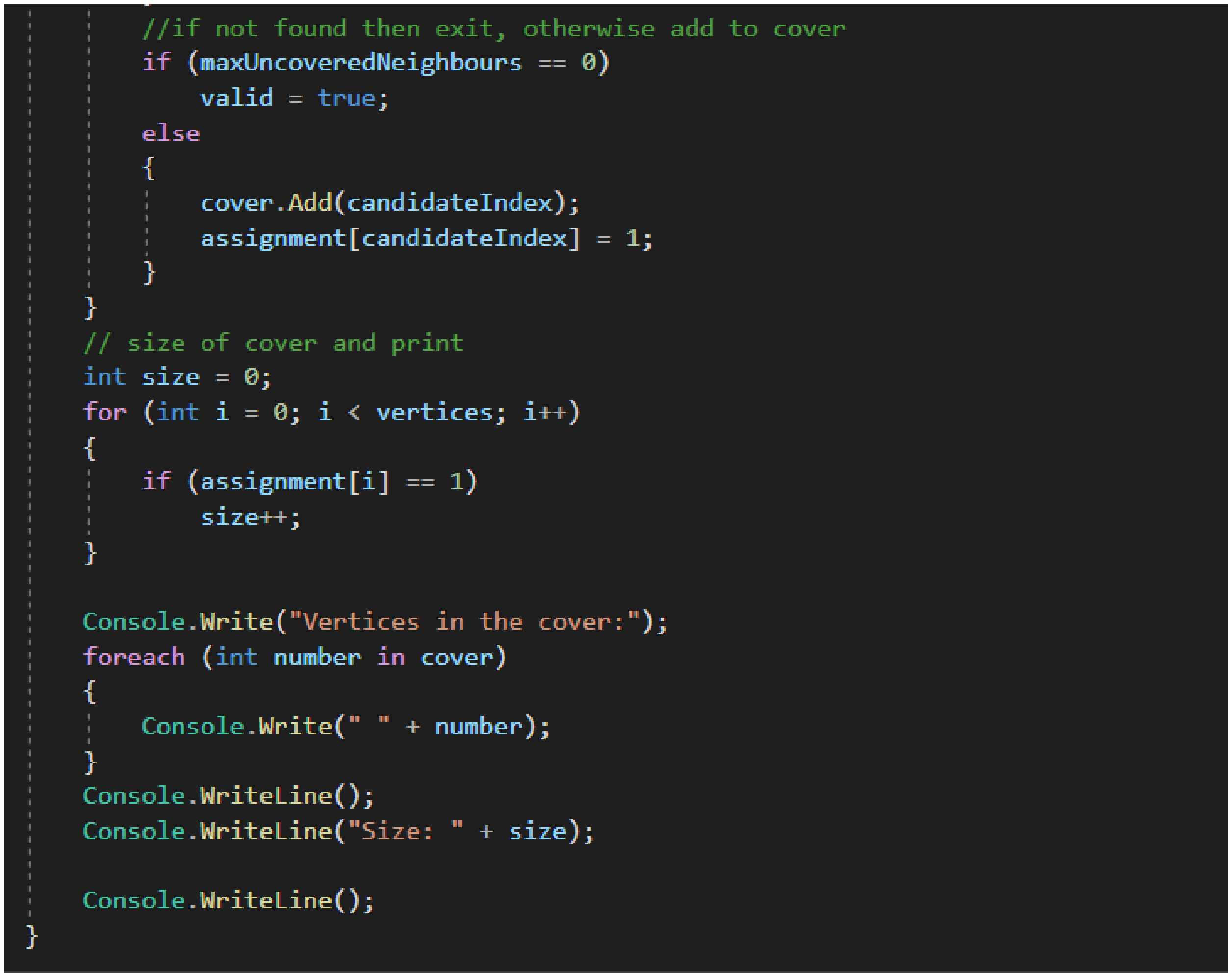
Because of how the “smart” algorithm omits certain branches of the built tree based on the kernelization process, the time required to run it is reduced. Further explanations are in the following code analysis section below.

The Approximation Algorithm Approach

Our approximation algorithm is based on the greedy approach for looking for a vertex cover in a graph. It was inspired by the examples given in the Udacity course.



```
public void ValidateAprox(Graph g, int vertices)
{
    int[] assignment = new int[vertices];
    List<int> cover=new List<int>();
    bool valid = false;

    while (!valid)
    {
        //look for vertex with most uncovered edges
        int candidateIndex = 0;
        int maxUncoveredNeighbours = 0;

        for (int i = 0; i < vertices; i++)
        {
            // vertex with 1 already covers adjacent edges
            if(assignment[i] !=1)
            {
                int sumUncovered = 0;
                for (int j = 0; j < vertices; j++)
                {
                    if (g.getAdjListArray()[i].Contains(j) && assignment[j] != 1)
                        sumUncovered++;
                }
                if(sumUncovered>maxUncoveredNeighbours)
                {
                    candidateIndex = i;
                    maxUncoveredNeighbours = sumUncovered;
                }
            }
        }
```



```
        //if not found then exit, otherwise add to cover
        if (maxUncoveredNeighbours == 0)
            valid = true;
        else
        {
            cover.Add(candidateIndex);
            assignment[candidateIndex] = 1;
        }
    }
    // size of cover and print
    int size = 0;
    for (int i = 0; i < vertices; i++)
    {
        if (assignment[i] == 1)
            size++;
    }

    Console.Write("Vertices in the cover:");
    foreach (int number in cover)
    {
        Console.Write(" " + number);
    }
    Console.WriteLine();
    Console.WriteLine("Size: " + size);

    Console.WriteLine();
}
```

The way this algorithm works is that it keeps searching through the vertices. For each iteration of searching, the algorithm counts how many connections each vertex has(without connections to vertexes already added to the cover from previous iterations). Then for this iteration it adds the vertex with the most connection to the cover and then checks if all edges have been added.

Further time complexity explanations are given in the section below.

2. Code Analysis

This section introduces the topics: asymptotic notation and performing test. The algorithm will be therefore analyzed based on its time complexity as well as on its performance.

TOPICS:

2.1. Time Complexity

Brute force approach:

For the brute force approach the calculation is very simple. Because the algorithm builds a binary tree with all the possible variations for the branches(true/false) for every vertex we have the following complexity: O(2^n). Other operations such as looping through the vertices and checks can be ignored because they have polynomial complexity which cannot be compared to the exponential one

Worst case: O(2^n)

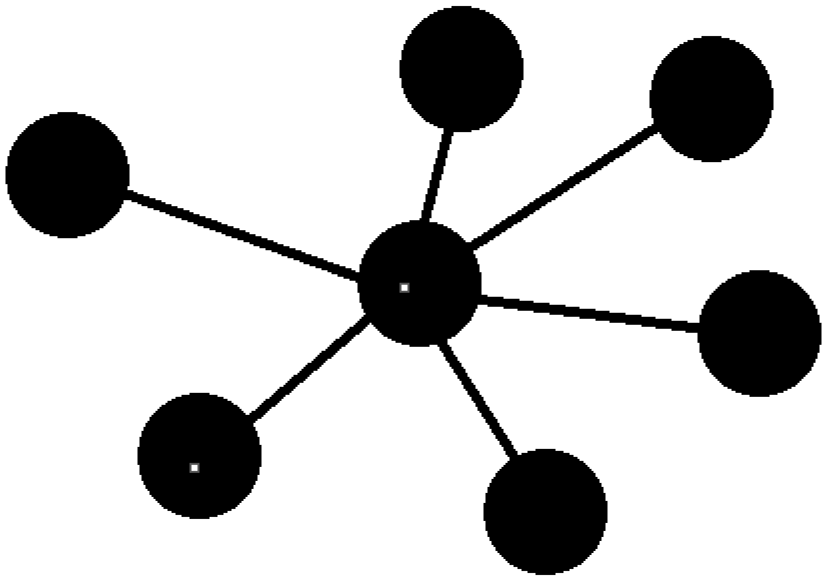
Average case: Ω(2^n)

Best case:Θ(2^n)

"Smart" brute force approach:

Because the "smart" brute force approach performs a preprocess the average case has the same exponential complexity but performs on average better depending on the graph. The preprocessing has polynomial time complexity.

An example of the best possible graph would be this one because it would get solved instantly by the kernelization process:



Worst case: O(2^n)

Average case: Ω(2^n)

Best case:Θ(n)

Approximation algorithm:

The approximation algorithm is the only one with polynomial complexity because it might not always give the optimum solution.

It is dependent on a new variable that we'll call V which checks whether we have found a possible vertex cover. This V can be anywhere in between 2(best case) and n(worst case) which gives us the following time complexity:

Worst case: O(n^3)

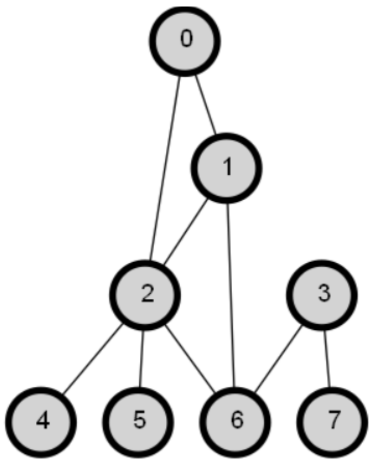
Average case: Ω(n^2*V)

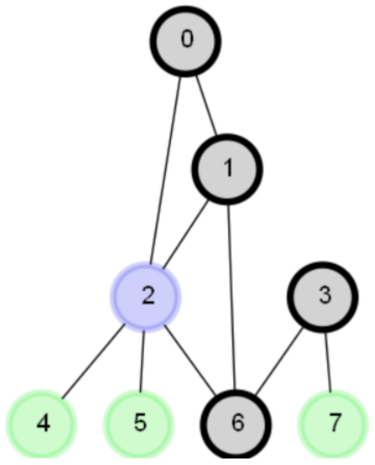
Best case:Θ(n^2)

2.2. Performance Test

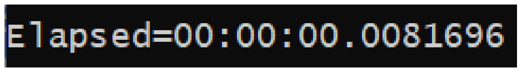
Given different input sizes, the average running time of the algorithm was the following:

1. For the graph consisting of 8 vertices with a probability of an edge of 30% and a given value k = 3, where k = the requested size k of the vertex cover, the following execution time was found:



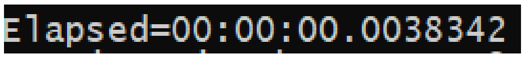


Brute force approach	Enhanced brute force approach	Approximation algorithm approach
8.1696≈8.17 ms	3.8342≈3.83 ms	4.0167≈4.02 ms



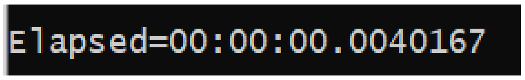
```
Elapsed=00:00:00.0081696
```

(Brute force approach)



```
Elapsed=00:00:00.0038342
```

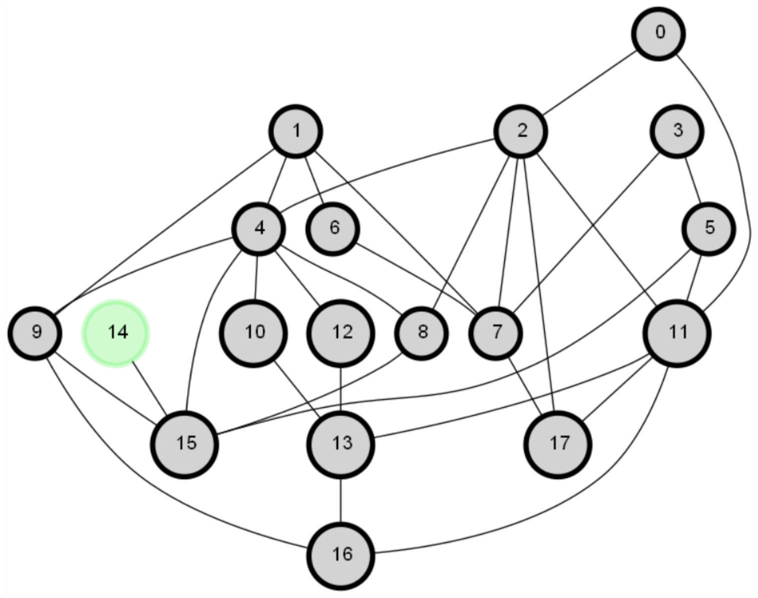
(Enhanced brute force approach)



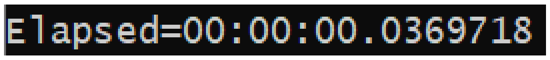
```
Elapsed=00:00:00.0040167
```

(Approximation algorithm approach)

2. For the graph consisting of 18 vertices with a probability of an edge of 20% and a given value k = 9, where k = the requested size k of the vertex cover, the following execution time was found:

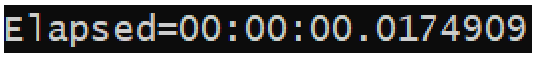


Brute force approach	Enhanced brute force approach	Approximation algorithm approach
36.97≈37.00 ms	17.49≈17.50 ms	4.0397≈4.04 ms



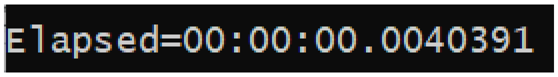
```
Elapsed=00:00:00.0369718
```

(Brute force approach)



```
Elapsed=00:00:00.0174909
```

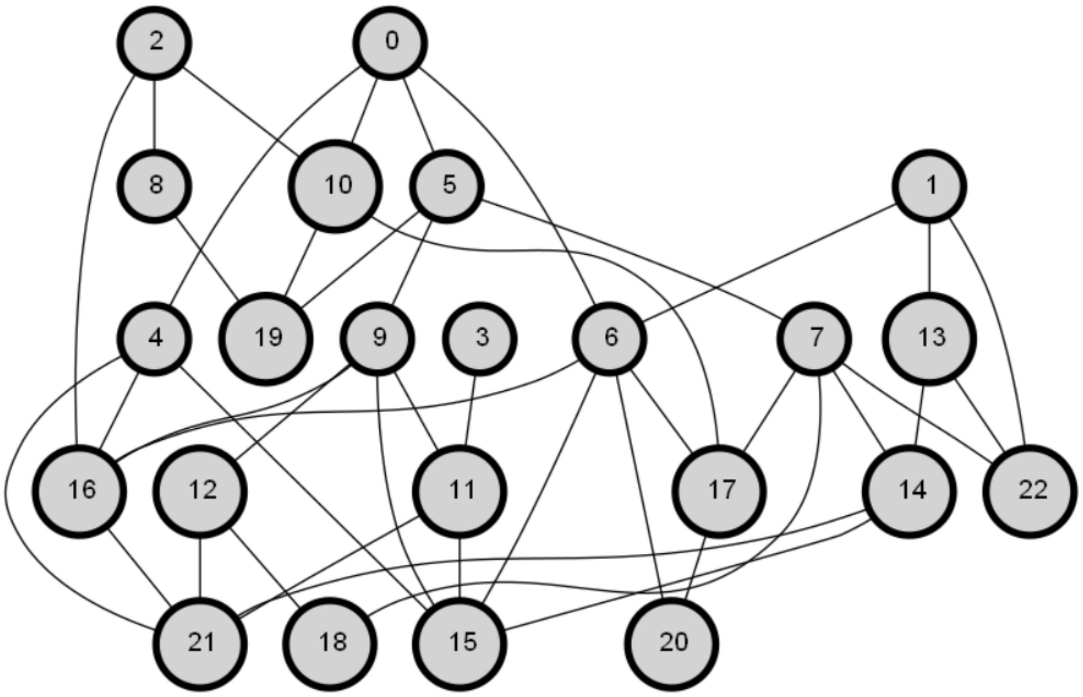
(Enhanced brute force approach)



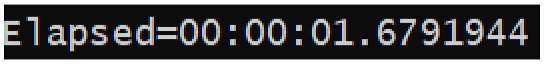
```
Elapsed=00:00:00.0040391
```

(Approximation algorithm approach)

3. For the graph consisting of 23 vertices with a probability of an edge of 20% and a given value k = 13, where k = the requested size k of the vertex cover, the following execution time was found:

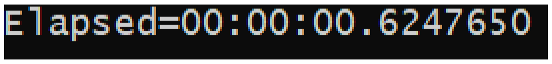


Brute force approach	Enhanced brute force approach	Approximation algorithm approach
1679.19≃1679.20 ms	624.76≃624.8 ms	5.056≃5.06 ms



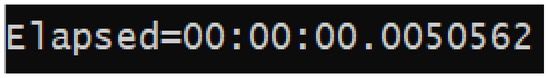
```
Elapsed=00:00:01.6791944
```

(Brute force approach)



```
Elapsed=00:00:00.6247650
```

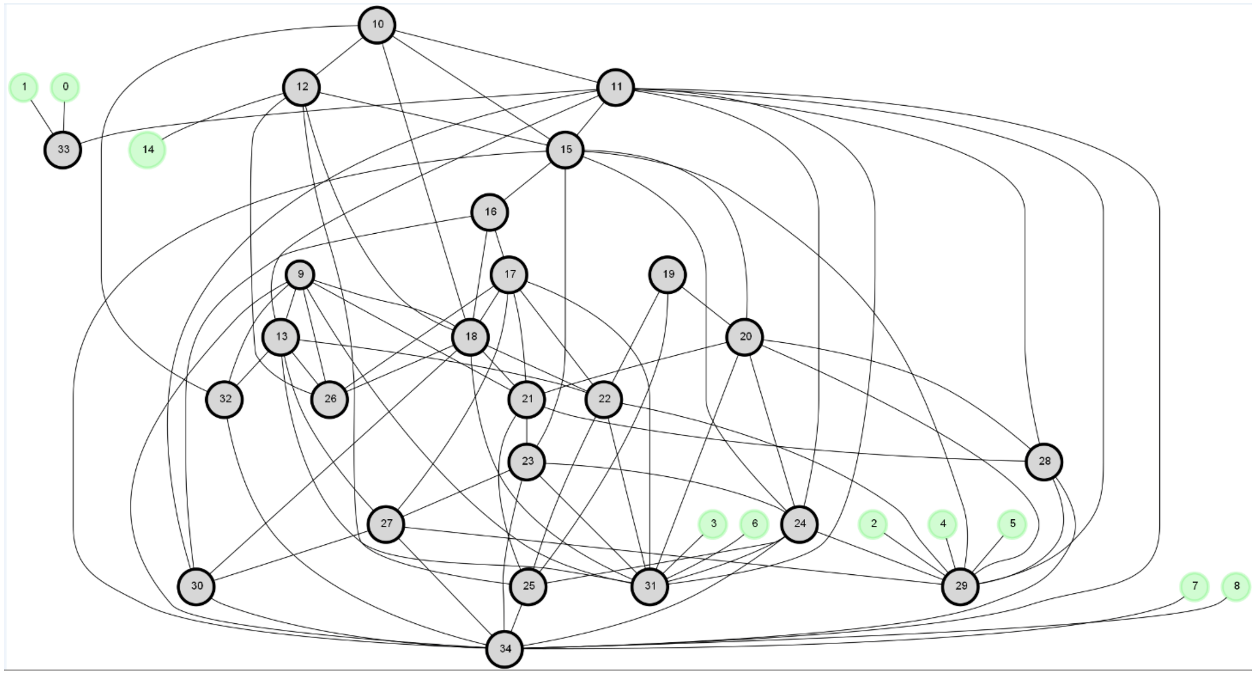
(Enhanced brute force approach)

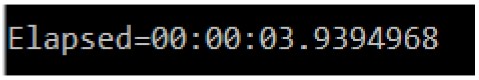


```
Elapsed=00:00:00.0050562
```

(Approximation algorithm approach)

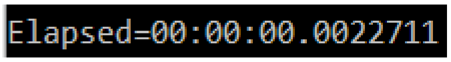
4. For the graph with 35 vertices and probability of an edge of 30 percent with pendants added to help the “smart” algorithm and a given value k = 19, where k = the requested size k of the vertex cover, the following execution time was found:





```
Elapsed=00:00:03.9394968
```

(Enhanced brute force approach)



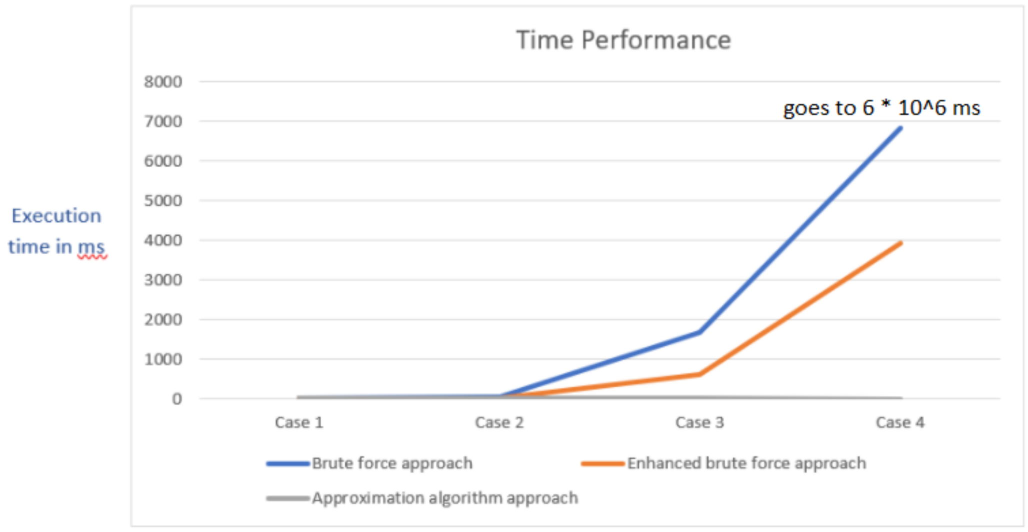
```
Elapsed=00:00:00.0022711
```

(Approximation algorithm approach)

For the brute force approach because it is too long we used a mathematical approximation based on the former value for 23 vertices and 1.67 seconds:
Difference between 35 and 23 vertices:12 vertices
1.67*(2^12)=6840 seconds
6840s/60=114 minutes

Brute force approach	Enhanced brute force approach	Approximation algorithm approach
114 minutes	3939 ms	2.27 ms

Based on the above results, the Time Performance graph (timeplot) could be drawn:



3. Bibliography

When are adjacency lists or matrices the better choice? (n.d.). StackExchange. Retrieved April 11, 2021, from https://cs.stackexchange.com/questions/79322/when-are-adjacency-lists-or-matrices-the-better-choice

Cormen, T. H., Leiserson, C. E., Rivest, R. E., & Stein, C. (n.d.). *Introduction to Algorithms* (Third Edition ed.).