Random sampling of finite graphs with constraints

Evelina Andersson
Abstract

Little is known about generated random graphs in which a certain type of subgraph is forbidden to exist as a subgraph. In this project a computer program has been developed for generating this kind of graphs and examine some of their properties, such as the number of edges and their chromatic numbers. The program was used to generate triangle free graphs, four cycle free graphs, tetrahedron free graphs and octahedron free graphs with 100, 200, 300, 400 or 500 vertices and the probability functions $p(n) = 1$, $p(n) = \frac{1}{2}$, $p(n) = \frac{1}{\sqrt{n}}$ or $p(n) = \frac{1}{n}$ were used in order to determine if an edge should exist in the graph. After that the number of edges and the number of triangles and the chromatic numbers for each type of graphs were analyzed.
## Contents

1 Probability measures on graphs 3
   1.1 Background .................................................. 3
      1.1.1 Complete graph ........................................ 3
      1.1.2 Isomorphism and classes of graphs ................... 3
      1.1.3 The degree of a vertex ................................ 4
      1.1.4 Vertex colouring ....................................... 4
   1.2 Probability on graphs ..................................... 4
   1.3 Graphs with constraints .................................. 4
   1.4 Problems ................................................... 5

2 Implementation 6
   2.1 Representing the graph in the program .................. 6
   2.2 Calculate the value of the probability function ...... 6
   2.3 Generating graphs ......................................... 7
      2.3.1 Generating graphs based on edges ..................... 7
      2.3.2 Generating graphs based on vertices ................. 12
   2.4 Checking for forbidden subgraphs ....................... 17
      2.4.1 Triangle Free ........................................ 17
      2.4.2 Four Cycle Free ....................................... 19
      2.4.3 Tetrahedron Free ...................................... 21
      2.4.4 Octahedron Free ...................................... 22
   2.5 Graph colouring ............................................ 27
   2.6 Counting edges ............................................. 33
   2.7 Counting triangles ......................................... 34

3 Experiments 41

4 Results 45
   4.1 Triangle free graphs ...................................... 45
   4.2 Four cycle free graphs .................................... 47
   4.3 Tetrahedron free graphs ................................... 49
   4.4 Octahedron free graphs ................................... 51

5 Analysis of the results 53

6 Conclusions and future work 56
1 Probability measures on graphs

This section reviews some graph theoretic concepts and results of relevance for the present study. For more details about the theory, see Diestel (2005).

1.1 Background

A graph is a pair of sets $G = (V,E)$ where $V$ is the set of vertices in the graph and $E \subset [V]^2$, that is, $E$ consists of 2-element subsets. An edge is written as $\{v_0, v_1\}$ and means that it exists a connection between the vertices $v_0$ and $v_1$ in the graph. In this case we also say that $v_0$ and $v_1$ are adjacent.

Example 1.1. The graph $G = (\{1,2,3,4,5\}, \{\{1,4\}, \{1,5\}, \{3,4\}, \{3,5\}\})$ looks like this:

\begin{center}
\begin{tikzpicture}
  \foreach \x in {1,2,3,4,5} {\node (\x) at (2.5*\x:1cm) {\x};}
  \foreach \x in {1,2,3,4,5} {\foreach \y in {1,2,3,4,5} {\x != \y \orb \x < \y \implies {\draw (\x) -- (\y);}};}
\end{tikzpicture}
\end{center}

1.1.1 Complete graph

Two vertices $x$ and $y$ are neighbours or adjacent if the edge $\{x,y\}$ exists in the graph. A complete graph, $K_n$, consists of $n$ vertices which are pairwise adjacent, which means that from each vertex in $K_n$ it is possible to reach any one of the other vertices in $K_n$ by just following one edge.

Example 1.2. The complete graph, $K^3$, forms a triangle and consists of the vertices 1,2 and 3 and the edges $\{1,2\}$, $\{1,3\}$ and $\{2,3\}$:

\begin{center}
\begin{tikzpicture}
  \foreach \x in {1,2,3} {\node (\x) at (90-120*\x:1cm) {\x};}
  \foreach \x in {1,2,3} {\foreach \y in {1,2,3} {\x != \y \orb \x < \y \implies {\draw (\x) -- (\y);}};}
\end{tikzpicture}
\end{center}

Example 1.3. The complete graph, $K^4$, forms a tetrahedron and consists of the vertices 1,2,3 and 4 and the edges $\{1,2\}$, $\{1,3\}$, $\{1,4\}$, $\{2,3\}$, $\{2,4\}$ and $\{3,4\}$:

\begin{center}
\begin{tikzpicture}
  \foreach \x in {1,2,3,4} {\node (\x) at (90-120*\x:1cm) {\x};}
  \foreach \x in {1,2,3,4} {\foreach \y in {1,2,3,4} {\x != \y \orb \x < \y \implies {\draw (\x) -- (\y);}};}
\end{tikzpicture}
\end{center}

1.1.2 Isomorphism and classes of graphs

The graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are isomorphic, $G_1 \simeq G_2$, if there is a bijection $\phi : V_1 \rightarrow V_2$ with $\{x,y\} \in E_1 \Leftrightarrow \{\phi(x), \phi(y)\} \in E_2$ for all $x,y \in V_1$. The mapping $\phi$ is called an isomorphism.

Example 1.4. The graphs $G_1 = (\{a,b,c,d\}, \{\{a,b\}, \{a,d\}, \{b,c\}, \{c,d\}\})$ and $G_2 = (\{1,2,3,4\}, \{\{1,2\}, \{1,4\}, \{2,3\}, \{3,4\}\})$ are isomorphic.
A graph property $\mathcal{P}$ is a class of graphs that is closed under isomorphism. For example if $G$ contains a triangle, that is, three pairwise adjacent vertices, then the graphs isomorphic to $G$ also contain a triangle.

1.1.3 The degree of a vertex
The set of neighbours to a vertex $v$ in a graph $G$ is denoted $N_G(v)$ and the degree $d(v)$ of a vertex is the number of edges adjacent to $v$.

1.1.4 Vertex colouring
A vertex colouring of a graph $G$ is a mapping $c : V \to \{1, 2, \ldots, k\}$ with this property:

if $\{x, y\} \in E$ then $c(x) \neq c(y)$

The minimum number $k$ such that there is a vertex colouring $c : V \to \{1, 2, \ldots, k\}$ of $G$ is denoted by $X(G)$ and called the chromatic number of $G$.

1.2 Probability on graphs
Let the set $\mathcal{G}_n$ contain all the graphs with the vertices $\{1, 2, \ldots, n\}$. For every edge probability $0 < p \leq 1$ we can define a probability measure $P_n$ on $\mathcal{G}_n$ by assigning each $G \in \mathcal{G}_n$ the probability $P_n(G) = p^m(1 - p)^{(n^2/2) - m}$. In this way $(\mathcal{G}_n, P_n)$ is a probability space which we denote $\mathcal{G}_{n,p}$. If $p = 1/2$ then $P_n(G) = 2^{-n^2/2}$ for every $G \in \mathcal{G}_{n,p}$, so in this case $P_n$ is the uniform probability measure on $\mathcal{G}_n$.

Now, it is possible to examine the probability that a graph $G \in \mathcal{G}_{n,p}$ has a certain property $\mathcal{P}$ such as the number of edges or the chromatic number. For every property $\mathcal{P}$ let $P_n(\mathcal{P}) = P_n(\{G \in \mathcal{G}_{n,p} : G has \mathcal{P}\}) = \sum_{G \in \mathcal{G}_{n,p}} P_n(G)$. If $\lim_{n \to \infty} P_n(\mathcal{P}) = 1$ then we say that almost all $\mathcal{G}_{n,p}$ have $\mathcal{P}$.

1.3 Graphs with constraints
In this project has random graphs that do not have a subgraph isomorphic to a graph $H$ been generated and examined. That is, we have studied the following set for some choices of $H$: $\mathcal{F}_n(H) = \{G \in \mathcal{G}_{n,p} : G has H\}$. $\mathcal{F}_n(H)$ can be seen as a probability space with the uniform measure.

In the uniform probability measure on $\mathcal{F}_n(K^l+1)$ where $l \geq 2$, the probability that $G \in \mathcal{F}_n(K^l+1)$ has the chromatic number $l$ converges to 1 when $n \to \infty$. The probability that $G \in \mathcal{F}_n(K^l+1)$ has at least $cn^2$ edges converges to 1 when $n \to \infty$ (Kolaitis et al., 1987), where $0 < c < 1$ is a constant.
But how can graphs $G \in \mathcal{F}_n(H)$ be generated in such a way that the graphs are generated with
the probability $\frac{1}{|\mathcal{F}_n(H)|}$? There is no simple answer to this question, but it is possible to generate
$H$-free graphs by using this random procedure based on a construction studied in Erdős et al. (1995):

1. Start with a graph $G = (\{1, 2, \ldots, n\}, \emptyset)$ which contains $n$ vertices and no edges, that is $E = \emptyset$,
a forbidden graph $H$ that is not allowed to exist as a subgraph and a function $p(n) : \mathbb{N} \to [0, 1]$.

2. Let the set $\Omega_n$ contain all the edges that exist in the complete graph $K^n$ with vertex set
$\{1, \ldots, n\}$:

$$\Omega_n = \{\{v, w\} : v, w \in \{1, \ldots, n\} \text{ and } v \neq w\}$$

The set has the cardinality $|\Omega_n| = \binom{n}{2}$.

Let $\Pi_n$ be the set of all possible permutations of the edges in $\Omega_n$. The cardinality of $\Pi_n$
is $|\Pi_n| = \binom{n}{2}$! and $\Pi_n$ should be seen as a probability space with the uniform measure.

Take one of the permutations in $\Pi_n$ by random:

$$\pi = (e_1, e_2, \ldots, e_{\binom{n}{2}})$$

3. Examine for each edge $e_i$ in $\pi$, if $G = (\{1, 2, \ldots, n\}, E \cup \{e_i\})$ does not have a subgraph
isomorphic to $H$, then $E$ is set to $E \cup \{e_i\}$ with the probability $p(n)$ otherwise $E$ is set to $E$.

4. After all the edges in $\pi$ have been examined, the generated graph $G = (\{1, 2, \ldots, n\}, E)$ is
returned.

For each $G \in \mathcal{F}_n(H)$, let $\mu_n(G)$ denote the probability that the graph produced by this procedure is $G$.

A few properties of the generated graphs are known. For example the probability with the probability
measure $\mu_n$ that $G \in \mathcal{F}_n(K^3)$ has at most $cn^3/2\log(n)$ edges converges to 1 when $n \to \infty$ (Erdős
et al., 1995), for a constant $c$. Hence, $\mu_n$ is not the same as the uniform measure $P_n$ or $F_n(K^3)$.

1.4 Problems

We have seen that the generating measure is not the same as the uniform measure. We therefore
do not know what a typical chromatic number for $G \in \mathcal{F}_n(H)$ is when the generating measure is
being used, not even when $H = K^3$. The goal with this M. Sc. project is to examine the following
properties using experiments in which graphs which contains 100, 200, 300, 400 or 500 vertices and
when $H$ is a triangle, a four cycle, a tetrahedron or an octahedron.

- The chromatic number for a member of $\mathcal{F}_n(H)$,
- the number of edges for a member of $\mathcal{F}_n(H)$,
- the number of triangles (i.e. 3-cycles) for a member of $\mathcal{F}_n(H)$.

In order to create random graphs and examine the properties of random graphs, I have imple-
mented a program in Java. The program can generate random graphs with the vertices $1, \ldots, n$ using
different probabilities and forbid different subgraphs and examine the properties of the graphs. Note
that it is impossible for a computer to create real random numbers, instead pseudo random numbers
are created by a random number generator in the interval $(0, 1)$ from the $\text{Re}(0, 1)$-distribution (Alm
and Britton, 2008).
2 Implementation

The source code and the manual to the program that has been created in this project can be found in the appendices $I$ and $J$ which are available electronically. This section describes the algorithms used in the program for generating random graphs, to determine if a forbidden subgraph will exist in the graph after an edge has been inserted, the number of edges in the graph, the number of colours needed to colour the graph and the number of triangles in the graph. The program requires these arguments:

- $|V|$ - the number of vertices in the generated graph
- $H$ - the graph which may not occur as a subgraph
- $p(n)$ - the probability function used to determine if an edge is to be inserted in the graph with vertices $1,...,n$ if this does not create a subgraph isomorphic to $H$.
- vertices/edges - an option telling if the graph should be generated according to vertices $^1$ or edges
- output file - the file in which the graph and its properties is written to

2.1 Representing the graph in the program

The graph generated by the program is stored in a matrix $A$ of the size $|V| \times |V|$ where $V = \{1,..,n\}$ for some $n$. The matrix $A$ is called an adjacency matrix and has a column and a row for each vertex in the graph and each element $a_{ij}$ in $A$ has either the value 1 or the value 0. The value 1 means that there is an edge between the vertices $i$ and $j$ and the value 0 means that an edge between $i$ and $j$ does not exist (Tucker, 2002), see example 2.1.

Example 2.1. A graph consisting of the vertices $V = \{1,2,3,4\}$ and the edges $E = \\{(1,2), (1,4), (2,3), (3,4)\}$ is stored in the matrix below:

\[
\begin{pmatrix}
0 & 1 & 0 & 1 \\
1 & 0 & 1 & 0 \\
0 & 1 & 0 & 1 \\
1 & 0 & 1 & 0 \\
\end{pmatrix}
\]

2.2 Calculate the value of the probability function

The probability function given to the program can consist of any combination of the arithmetic operations, numbers, roots and a variable which represents the number of vertices in the graph $|V|$:

$^1$Only the generating procedure based on edges has been described in section 2 and the generating method based on vertices has been implemented as an alternative.
<table>
<thead>
<tr>
<th>Name</th>
<th>Syntax</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>addition</td>
<td>expression_1 + expression_2</td>
<td></td>
</tr>
<tr>
<td>substraction</td>
<td>expression_1 - expression_2</td>
<td></td>
</tr>
<tr>
<td>multiplication</td>
<td>expression_1 * expression_2</td>
<td></td>
</tr>
<tr>
<td>division</td>
<td>expression_1 / expression_2</td>
<td></td>
</tr>
<tr>
<td>expression with parentheses</td>
<td>( expression )</td>
<td></td>
</tr>
<tr>
<td>number</td>
<td>number</td>
<td>number ∈ N</td>
</tr>
<tr>
<td>variable</td>
<td>n</td>
<td>If the probability function contains n then n is replaced by</td>
</tr>
<tr>
<td>√(expression)</td>
<td>sqrt ( expression )</td>
<td></td>
</tr>
</tbody>
</table>

Before the given probability function, \( p(n) \), is used by the program, the value of \( p(n) \) is calculated.

The probability functions used in the experiments are:

<table>
<thead>
<tr>
<th>Probability function</th>
<th>The value used in the program</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p(n) = 1 )</td>
<td>1</td>
</tr>
<tr>
<td>( p(n) = 1/2 )</td>
<td>0.5</td>
</tr>
<tr>
<td>( p(n) = 1/\sqrt{</td>
<td>V</td>
</tr>
<tr>
<td>( p(n) = 1/n )</td>
<td>( \frac{1}{</td>
</tr>
</tbody>
</table>

### 2.3 Generating graphs

The program can generate graphs in two different ways, one based on edges in which the edges between the nodes are inserted randomly in the graph (Erdős et al., 1995), and one based on vertices in which the vertices are chosen randomly and all edges that can be attached to the chosen vertex without introducing the forbidden subgraph \( H \) are inserted with the probability \( p(n) \).

#### 2.3.1 Generating graphs based on edges

In the edge based version of generating a random graph, the program creates a random graph by randomly inserting, with the given probability, the edges in a graph that do not introduce the forbidden subgraph. The algorithm 1 performs these steps:

- At the line 1, the algorithm creates the square matrix \( A \) of the size \(|V| \times |V|\) in which the generated graph will be stored in.

- The loops at the lines 3 - 7 construct the set \( E_{test} \) which contains all the edges that will be tested during the creation of the graph:

\[
E_{test} = \{\{1,2\}, \{1,3\}, ..., \{1,n\}, \{2,3\}, \{2,4\}, ..., \{2,n\}, ..., \{n-1,n\}\}
\]

- The loop at the lines 8-14 examines all the edges in \( E_{test} \) in these steps:

  - At the line 9 an edge is chosen by random from \( E_{test} \) and stored in the variable \( e_{ij} \).
  
  - The condition at the line 10 creates a new random number, \( random \), and tests if it is lower than the value of the given probability function, \( probability \), and if the edge in \( e_{ij} \) can be inserted in the graph without introducing the forbidden subgraph. If this condition is true then \( e_{ij} \) is inserted in \( A \) at the line 11 and removed from \( E_{test} \) at the line 13.

- After the loop at the lines 8-14 has examined all the edges, the generated graph stored in the matrix \( A \) is returned at the line 15.
Algorithm 1 generateGraph(size, probability, subgraph)

1: $A \leftarrow$ new $MATRIX[size][size]$
2: $E_{test} \leftarrow \emptyset$
3: for $i := 1$ to $size - 1$ do
4:   for $j := i + 1$ to $size$ do
5:      $E_{test} = E_{test} \cup \{\{i, j\}\}$
6:   end for
7: end for
8: while ($|E_{test}| > 0$) do
9:   $e_{ij} \leftarrow$ take an edge by random from $E_{test}$
10:  if random $\leq$ probability $\land \neg$subgraph.exists($A, e_{ij}$) then
12:  end if
13:  $E_{test} = E_{test} \setminus \{e_{ij}\}$
14: end while
15: return $A$

Example 2.2. In this example has the program been given these arguments:

| $|V|$ | forbidden subgraph | probability function | generating method | output file |
|------|--------------------|----------------------|-------------------|-------------|
| 4    | triangle           | $p(\ n\ ) = 1$      | edges             | result.graph |

In the initial step the graph, $G_{4,1}$, only contains four vertices:

```
  1   2
  \
  4   3
```

$G_{4,1}$

In the next step the program creates the set $E_{test}$ which contains all the edges that will be tested if they can be inserted in the graph:

$E_{test} = \{\{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 3\}, \{2, 4\}, \{3, 4\}\}$

After that the algorithm takes one edge from $E_{test}$ by random and tests if it can be inserted in $G_{4,1}$ without violating the forbidden subgraph triangle:

After the algorithm has tested the edge $\{3, 4\}$, it removes $\{3, 4\}$ from $E_{test}$ and takes another edge from $E_{test}$ by random and tries to insert it in the graph:
Graph | Edges | Test | Result
---|---|---|---
\(G_{4,1}\) | \(E_{test} = \{\{1, 2\}, \{3, 4\}\}\) | Can the edge \(\{1, 2\}\) be inserted in \(G_{4,1}\)? | Yes, because the edges \(\{1, 2\}\) and \(\{3, 4\}\) do not form a triangle in \(G_{4,1}\)

After that the algorithm continues with removing the edge \(\{1, 2\}\) from \(E_{test}\) and takes a new edge by random from \(E_{test}\) and tries to insert it in the graph:

Graph | Edges | Test | Result
---|---|---|---
\(G_{4,1}\) | \(E_{test} = \{\{1, 3\}, \{1, 4\}, \{2, 3\}, \{2, 4\}\}\) | Can the edge \(\{1, 3\}\) be inserted in \(G_{4,1}\)? | Yes, because the edges \(\{1, 2\}\), \(\{1, 3\}\) and \(\{3, 4\}\) do not form a triangle in \(G_{4,1}\)

The algorithm will continue with trying to insert the rest of the edges in \(E_{test}\) one at a time until \(E_{test}\) is empty:

Graph | Edges | Test | Result
---|---|---|---
\(G_{4,1}\) | \(E_{test} = \{\{1, 4\}, \{2, 3\}, \{2, 4\}\}\) | Can the edge \(\{2, 3\}\) be inserted in \(G_{4,1}\)? | No, because the edges \(\{1, 2\}\), \(\{1, 3\}\) and \(\{2, 3\}\) form a triangle in \(G_{4,1}\)

\(G_{4,1}\) | \(E_{test} = \{\{1, 4\}, \{2, 4\}\}\) | Can the edge \(\{1, 4\}\) be inserted in \(G_{4,1}\)? | No, because the edges \(\{1, 3\}\), \(\{1, 4\}\) and \(\{3, 4\}\) form a triangle in \(G_{4,1}\)

\(G_{4,1}\) | \(E_{test} = \{\{2, 4\}\}\) | Can the edge \(\{2, 4\}\) be inserted in \(G_{4,1}\)? | Yes, because the edges \(\{1, 2\}\), \(\{1, 3\}\), \(\{2, 4\}\) and \(\{3, 4\}\) do not form a triangle in \(G_{4,1}\)

After the algorithm has examined all edges in \(E_{test}\), the generated graph stored in the matrix \(A\) looks like this:
Example 2.3. In this example has the program been given these arguments:

<table>
<thead>
<tr>
<th>V</th>
<th>forbidden subgraph</th>
<th>probability function</th>
<th>generating method</th>
<th>output file</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>triangle</td>
<td>( p(n) = 1 / n )</td>
<td>edges</td>
<td>result.graph</td>
</tr>
</tbody>
</table>

In the initial state the graph only contains four vertices:

\[
\begin{array}{cc}
1 & 2 \\
4 & 3 \\
\end{array}
\]

\( G_{4,1/4} \)

In the next step the program creates a set containing all the edges that will be tested if they can be inserted in the graph:

\[ E_{test} = \{\{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 3\}, \{2, 4\}, \{3, 4\}\} \]

After that the algorithm takes one edge from \( E_{test} \) by random and tests if it can be inserted in \( G_{4,1/4} \) without violating the forbidden subgraph triangle and the probability \( \frac{1}{4} \):

<table>
<thead>
<tr>
<th>Graph</th>
<th>Edges</th>
<th>Test</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>( G_{4,1/4} )</td>
<td>( E_{test} = {{1, 2}, {1, 3}, {1, 4}, {2, 3}, {2, 4}, {3, 4}} )</td>
<td>Can the edge ( {3, 4} ) be inserted in ( G_{4,1/4} ) if random ( &gt; \frac{1}{4} ) ?</td>
<td>No, because random ( &gt; \frac{1}{4} )</td>
</tr>
</tbody>
</table>

After the algorithm has tested \( \{3, 4\} \) it removes \( \{3, 4\} \) from \( E_{test} \) and takes another edge from \( E_{test} \) by random and tries to insert it in the graph:

<table>
<thead>
<tr>
<th>Graph</th>
<th>Edges</th>
<th>Test</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>( G_{4,1/4} )</td>
<td>( E_{test} = {{1, 2}, {1, 3}, {1, 4}, {2, 3}, {2, 4}} )</td>
<td>Can the edge ( {1, 2} ) be inserted in ( G_{4,1/4} ) if random ( &gt; \frac{1}{4} ) ?</td>
<td>No, because random ( &gt; \frac{1}{4} )</td>
</tr>
</tbody>
</table>
After that the algorithm continues with removing \( \{1, 2\} \) from \( E_{test} \) and takes a new edge by random from \( E_{test} \) and tries to insert it in the graph:

<table>
<thead>
<tr>
<th>Graph</th>
<th>Edges</th>
<th>Test</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2</td>
<td>( G_{4,1/4} ) ( E_{test} = {{1, 3}, {1, 4}, {2, 3}, {2, 4}} )</td>
<td>Can the edge ( {1, 3} ) be inserted in ( G_{4,1/4} )? if random &lt; ( \frac{1}{3} )?</td>
<td>Yes, because random &lt; ( \frac{1}{3} ) and the edge ( {1, 3} ) do not form a triangle in ( G_{4,1/4} )</td>
</tr>
</tbody>
</table>

After that the algorithm removes the edge \( \{1, 3\} \) from \( E_{test} \) and continue with the same procedure until \( E_{test} \) is empty:

<table>
<thead>
<tr>
<th>Graph</th>
<th>Edges</th>
<th>Test</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2</td>
<td>( G_{4,1/4} ) ( E_{test} = {{1, 4}, {2, 3}, {2, 4}} )</td>
<td>Can the edge ( {2, 4} ) be inserted in ( G_{4,1/4} )? if random &gt; ( \frac{1}{3} )?</td>
<td>No, because random &gt; ( \frac{1}{3} )</td>
</tr>
<tr>
<td>1 2</td>
<td>( G_{4,1/4} ) ( E_{test} = {{1, 4}, {2, 3}} )</td>
<td>Can the edge ( {2, 3} ) be inserted in ( G_{4,1/4} )? if random &gt; ( \frac{1}{3} )?</td>
<td>No, because random &gt; ( \frac{1}{3} )</td>
</tr>
<tr>
<td>1 2</td>
<td>( G_{4,1/4} ) ( E_{test} = {{1, 4}} )</td>
<td>Can the edge ( {1, 4} ) be inserted in ( G_{4,1/4} )? if random &gt; ( \frac{1}{3} )?</td>
<td>No, because random &gt; ( \frac{1}{3} )</td>
</tr>
</tbody>
</table>

After the algorithm has examined all edges in \( E_{test} \) the generated graph stored in the matrix \( A \) looks like this:

\[
\begin{array}{cc}
1 & 2 \\
\hline
4 & 3
\end{array}
\]

\( G_{4,1/4} \)
2.3.2 Generating graphs based on vertices

In the vertex based version of generating a random graph, the program creates a random graph by first taking a vertex \( v \) by random. After that all edges that can be attached to \( v \) without introducing the forbidden subgraph are inserted with the given probability. This procedure is performed for each vertex in the graph. The algorithm 2 used to generate graphs according to vertices performs these steps:

- At the line 1, the algorithm creates the square matrix \( A \) of the size \( size \times size \) in which the generated graph will be stored in.
- The set \( V \) which contains the vertices in the graph, is created at the lines 2-5:
  \[
  V = \{1, 2, ..., n\}
  \]
- The loop at the lines 6-18 traverses through the vertices in \( V \) and executes these steps:
  - At the line 7 a vertex \( v \) is taken by random from \( V \)
  - At the lines 8 - 11 the set \( E_{test} \) which contains all possible edges that can be connected between \( v \) and one of the vertices \( v + 1, ..., size - 1 \) or \( size \) is created:
    \[
    E_{test} = \{\{v, v + 1\}, \{v, size - 1\}, ..., \{v, size\}\}
    \]
  - the loop at the lines 12-18 loops through \( E_{test} \) and performs these steps:
    * At the line 13 the first edge in \( E_{test} \) is stored in the variable \( e_{ij} \).
    * The condition at the line 14 creates a new random number and tests if it is lower than the value of the given probability function, \( probability \), and if inserting \( e_{ij} \) in the graph does not introduce the forbidden subgraph. If this condition is true then \( e_{ij} \) is inserted in the matrix \( A \) at the line 15.
    * After the condition has tested \( e_{ij} \), \( e_{ij} \) is removed from \( E_{test} \) at the line 17.
  - After the loop at the lines 12-18 has examined all the edges in \( E_{test} \), \( v \) is removed from \( V \) at the line 19.
- After the loop at the lines 6-18 has examined all the vertices, the generated graph stored in the matrix \( A \) is returned at the line 19.
Algorithm 2 generateGraph(size, probability, subgraph)

1:  $A \leftarrow \text{new MATRIX}[\text{size}][\text{size}]$
2:  $V \leftarrow \emptyset$
3:  for $i := 1$ to size do
4:      $V = V \cup \{i\}$
5:  end for
6:  while ($|V| > 0$) do
7:      $v \leftarrow$ take a vertex by random from $V$
8:      $E_{\text{test}} \leftarrow \emptyset$
9:      for $j := v + 1$ to size do
10:         $E_{\text{test}} = E_{\text{test}} \cup \{\{v, j\}\}$
11:     end for
12:    while ($|E_{\text{test}}| > 0$) do
13:       $e_{ij} \leftarrow$ the first edge in $E_{\text{test}}$
14:       if random $\leq$ probability $\land \neg$ subgraph.exists($A, e_{ij}$) then
16:       end if
17:       $E_{\text{test}} = E_{\text{test}} \setminus \{e_{ij}\}$
18:    end while
19:  $V = V \setminus \{v\}$
20: end while
21: return $A$

Example 2.4. In this example has the program been given these arguments:

| $|V|$ | forbidden subgraph | probability function $p(n) = 1$ | generating method | output file |
|------|--------------------|----------------------------------|-------------------|-------------|
| 4    | triangle           |                                  | vertices          | result.graph |

In the initial step the graph only contains four vertices:

\[ \begin{array}{cc}
1 & 2 \\
4 & 3 \\
\end{array} \]

\[ G_{4,1} \]

In the first step the algorithm creates a set containing all the vertices in the graph:

\[ V = \{1, 2, 3, 4\} \]

In the next step the algorithm takes a vertex by random from $V$, for example vertex number 2, and creates a set containing all the edges that can be attached to vertex 2 and to the vertices labeled with a larger number than 2:

\[ E_{\text{test}} = \{\{2, 3\}, \{2, 4\}\} \]

After that the algorithm loops through $E_{\text{test}}$ and inserts all edges that can be inserted into the graph without introducing a triangle in the graph:
In the next step the algorithm removes vertex 2 from $V$ and takes another vertex by random from $V$, for example vertex 1, and all the edges that can be attached to vertex 1 are collected in a set $E_{test}$:

$$E_{test} = \{\{1,2\}, \{1,3\}, \{1,4\}\}$$

After that the algorithm loops through $E_{test}$ and inserts all edges that can be inserted in $G_{4,1}$ without introducing a triangle in the graph:

After that vertex 1 is removed from $V$ and another vertex is taken by random from $V$, for example vertex 3, and the edge $\{3,4\}$ that can be attached to the vertex 3 is stored in the set $E_{test}$:

$$E_{test} = \{\{3,4\}\}$$
After that the algorithm tests if the edge \(\{3, 4\}\) can be added to \(G_{4,1}\) without introducing a triangle in \(G_{4,1}\):

<table>
<thead>
<tr>
<th>Graph</th>
<th>Edges</th>
<th>Test</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>(G_{4,1})</td>
<td>(E_{\text{test}} = {{3, 4}})</td>
<td>Can the edge ({3, 4}) be inserted in (G_{4,1})?</td>
<td>No, because the edges ({2, 3}), ({3, 4}) and ({3, 4}) form a triangle in (G_{4,1})</td>
</tr>
</tbody>
</table>

After that vertex 3 is removed from \(V\) and a new vertex is taken by random from \(V\), for example vertex 4. Since this vertex is the vertex with the highest number in \(G_{4,1}\), the program does not generate any edges for this vertex, instead this vertex is removed from \(V\). Since \(V\) is now empty, the generated graph stored in the matrix \(A\) is returned by the algorithm:

\[
\begin{align*}
\begin{array}{c}
1 \\
4 \\
2 \\
3
\end{array}
\end{align*}
\]

\(G_{4,1}\)

Note that even if the algorithm starts in another vertex than 2 in example 2.4, the resulting graph will always be isomorphic to \(G_{4,1}\).

**Example 2.5.** In this example has the program been given these arguments:

| \(|V|\) | forbidden subgraph | probability function | generating method | output file |
|-------|-------------------|----------------------|------------------|-------------|
| 4     | triangle          | \(p(n) = 1/n\)      | vertices         | result.graph |

In the initial step the graph only contains four vertices:

\[
\begin{align*}
\begin{array}{c}
1 \\
4 \\
2 \\
3
\end{array}
\end{align*}
\]

\(G_{4,1/4}\)

In the first step the algorithm creates a set containing all the vertices in the graph:

\[
V = \{1, 2, 3, 4\}
\]

In the next step the algorithm takes a vertex by random, for example vertex 2, from \(V\) and creates a set containing all the edges that can be attached to vertex 2 and to the vertices labeled with a larger number than 2:

\[
E_{\text{test}} = \{\{2, 3\}, \{2, 4\}\}
\]

After that the algorithm loops through \(E_{\text{test}}\) and inserts the edges that can be inserted in \(G_{4,1/4}\) without introducing a triangle in \(G_{4,1/4}\) and without violating the probability \(\frac{1}{4}\):
<table>
<thead>
<tr>
<th>Graph</th>
<th>Edges</th>
<th>Test</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(G_{4,1/4})</td>
<td>(E_{test} = {{2, 3}, {2, 4}})</td>
<td>Can the edge {2, 3} be inserted in (G_{4,1/4}) if random (&gt; \frac{1}{4})?</td>
<td>No, because random (&gt; \frac{1}{4})</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Graph</th>
<th>Edges</th>
<th>Test</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(G_{4,1/4})</td>
<td>(E_{test} = {{2, 3}, {2, 4}})</td>
<td>Can the edge {2, 4} be inserted in (G_{4,1/4}) if random (&gt; \frac{1}{4})?</td>
<td>No, because random (&gt; \frac{1}{4})</td>
</tr>
</tbody>
</table>

After that the algorithm removes vertex 2 from \(V\) and another vertex is taken by random from \(V\), for example vertex 1, and all the edges that can be attached to the vertex \(v\) are collected in a set \(E_{test}\):

\[E_{test} = \{\{1, 2\}, \{1, 3\}, \{1, 4\}\}\]

After that the algorithm loops through \(E_{test}\) and inserts all the edges that can be inserted in \(G_{4,1/4}\) without introducing a triangle in \(G_{4,1/4}\) and without violating the probability \(\frac{1}{4}\):

<table>
<thead>
<tr>
<th>Graph</th>
<th>Edges</th>
<th>Test</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(G_{4,1/4})</td>
<td>(E_{test} = {{1, 2}, {1, 3}, {1, 4}})</td>
<td>Can the edge {1, 2} be inserted in (G_{4,1/4}) if random (&gt; \frac{1}{4})?</td>
<td>No, because random (&gt; \frac{1}{4})</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Graph</th>
<th>Edges</th>
<th>Test</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(G_{4,1/4})</td>
<td>(E_{test} = {{1, 2}, {1, 3}, {1, 4}})</td>
<td>Can the edge {1, 3} be inserted in (G_{4,1/4}) if random (&lt; \frac{1}{4})?</td>
<td>Yes, because random (&lt; \frac{1}{4}) and the edge {3, 4} does not form a triangle in (G_{4,1/4})</td>
</tr>
</tbody>
</table>
After that vertex 1 is removed from $V$ and another vertex is taken by random from $V$, for example vertex 3, and the edge $\{3,4\}$ is stored in the set $E_{test}$:

$$E_{test} = \{\{3,4\}\}$$

After that the algorithm tests if the edge $\{3,4\}$ can be inserted in $G_{4,1/4}$ without introducing a triangle in $G_{4,1/4}$ and without violating the probability $\frac{1}{4}$:

After that vertex 3 is removed from $V$ and a new vertex is taken by random from $V$, for example vertex 4. Since this vertex is the vertex with the highest number in $G_{4,1/4}$, the algorithm does not generate any edges for this vertex, instead this vertex is removed from $V$. Since $V$ is now empty, the generated graph stored in the matrix $A$ is returned by the algorithm:

$$G_{4,1/4}$$

2.4 Checking for forbidden subgraphs

The program can search for the subgraphs triangles, four cycles, tetrahedrons and octahedrons. This section describes the algorithms used for searching for these subgraphs in the graph and as arguments they require the graph and the edge the program tries to add to the graph.

2.4.1 Triangle Free

In order to examine if a graph $G = (V,E)$ will contain a triangle after the edge $e = \{e_0,e_1\}$ has been inserted in $G$, the method $\text{triangleExists}(G,e)$ goes through all the edges in $E$ and tries to find three vertices in the graph that can be combined into a triangle if $e$ is inserted in $G$, see figure 1.
The method only returns true if it can find three vertices that can be combined into a triangle, see example 2.6, otherwise the method returns false. The algorithm used in the method triangleExists(G = (V, E), e = {e₀, e₁}) performs these steps:

- The loop at the lines 1-5 traverses through all the vertices in the set V \ {e₀, e₁} and tests if the edges {e₀, v} and {e₁, v} exists in E. If this is true then the graph will contain a triangle between the vertices e₀, e₁ and v if the edge {e₀, e₁} is inserted in the graph. The method therefore returns true at the line 3.

- If a triangle was not found during the execution of the loop at the lines 1-5, then the method returns false at the line 6.

Algorithm 3 triangleExists(G=(V, E), e = {e₀, e₁})

1: for all v ∈ V \ {e₀, e₁} do
2:    if {e₀, v} ∈ E ∧ {e₁, v} ∈ E then
3:        return true
4:    end if
5: end for
6: return false

Example 2.6. This happens when the method triangleExists(G, e) examines if it possible to insert the edge e = {2, 3} in the graph G = (\{(1, 2, 3, 4), \{(1, 2), (2, 4), (3, 4)\}\}) without introducing a triangle:

Given:
G = (\{(1, 2, 3, 4), E = \{(1, 2), (2, 4), (3, 4)\}\})
and e = \{2, 3\}

In the first iteration of the loop at the lines 1-5, the loop examines vertex 1 in the graph. In this case the loop finds the edge \{1, 2\}.

Since the edges \{1, 2\} and \{2, 3\} do not form a triangle in the graph, the loop will now continue with examining vertex 2.
In the second iteration of the loop, the loop examines vertex 2. In this case the loop finds the edges \( \{1, 2\} \) and \( \{2, 4\} \).

Since the edges \( \{1, 2\}, \{2, 4\} \) and \( \{2, 3\} \) do not form a triangle, the loop will now continue with examining vertex 3.

In the third iteration of the loop, the loop examines vertex 3. In this case the loop finds the edge \( \{3, 4\} \).

Since the edges \( \{3, 4\} \) and \( \{2, 3\} \) do not form a triangle the loop will now continue with examining vertex 4.

In the fourth iteration of the loop, the loop examines vertex 4. In this case the loop finds the edges \( \{2, 4\} \) and \( \{3, 4\} \).

Since the edges \( \{2, 4\}, \{3, 4\} \) and \( \{2, 3\} \) form a triangle, the method will now return true.

\[ \square \]

### 2.4.2 Four Cycle Free

In order to examine if a graph \( G = (V, E) \) will contain a four cycle after the edge \( e = \{e_0, e_1\} \) has been inserted in \( G \), the method \( \text{fourCycleExists}(G, e) \) goes through all pair of the vertices \( < v_0, v_1 > \) where \( v_0 \in V \setminus \{e_0, e_1\} \) and \( v_1 \in V \setminus \{e_0, e_1, v_0\} \) and examines if the edges \( \{e_0, v_0\}, \{e_1, v_1\} \) and \( \{v_0, v_1\} \) exists in \( E \). If that is the case then a four cycle will be introduced in \( G \) if \( e \) is inserted in \( G \), see figure 4:

![Figure 2: The edges \( \{e_0, e_1\}, \{e_0, v_0\}, \{e_1, v_1\} \) and \( \{v_0, v_1\} \) form a four cycle between the vertices \( e_0, e_1, v_0, \) and \( v_1.\) The method only returns true if it can find four vertices that can be combined into a four cycle, see example 2.7, otherwise the method returns false. The algorithm 4 used in the method \( \text{fourCycleExists}(G = (V, E), e = \{e_0, e_1\}) \) performs these steps:

- The loop at the lines 1-7 traverses through all the vertices in the set \( V \setminus \{e_0, e_1\} \) and executes this step:
The loop at the lines 2-6 traverses through the vertices in the set $V \setminus \{e_0, e_1, v_0\}$ and tests if the edges $\{e_0, v_0\}$, $\{e_1, v_1\}$ and $\{v_0, v_1\}$ exists in $E$. If this is true then the graph will contain a four cycle between the vertices $e_0$, $e_1$, $v_0$ and $v_1$ if the edge $\{e_0, e_1\}$ is inserted in the graph so the method therefore returns $true$ at the line 4.

- If a four cycle was not found during the execution of the loops at the lines 1-7, then the edge $\{e_0, e_1\}$ can be inserted in $G$ without introducing a four cycle in $G$ and the method therefore returns $false$ at the line 8.

**Algorithm 4** fourCycleExists($G=(V, E)$, $e = \{e_0, e_1\}$)

1: for all $v_0 \in V \setminus \{e_0, e_1\}$ do  
2:   for all $v_1 \in V \setminus \{e_0, e_1, v_0\}$ do  
3:     if $\{e_0, v_0\} \in E \land \{e_1, v_1\} \in E \land \{v_0, v_1\} \in E$ then  
4:        return $true$  
5:     end if  
6:   end for  
7: end for  
8: return $false$

**Example 2.7.** This happens when the method fourCycleExists($G, e$) examines if it possible to insert the edge $e = \{2,3\}$ in the graph $G = (\{1, 2, 3, 4\}, \{\{1,2\}, \{1,4\}, \{3,4\}\})$ :

Given: 
$G = (\{1, 2, 3, 4\}, \{\{1,2\}, \{1,4\}, \{3,4\}\})$ 
and $e = \{2,3\}$

In the first iteration of the loops at the lines 1-7, the loops examine the vertices 1 and 2. In this case the loops find the edges $\{1,2\}$ and $\{1,4\}$.

Since the edges $\{1,2\}$, $\{1,4\}$ and $\{2,3\}$ do not form a four cycle, the loops will move to the next pair of vertices, 1 and 3.

In the second iteration of the loops, the loops examine the vertices 1 and 3. In this case the loops find the edges $\{1,2\}$, $\{1,4\}$ and $\{3,4\}$.

Since the edges $\{1,2\}$, $\{1,4\}$, $\{3,4\}$ and $\{2,3\}$ form a four cycle, the method will now return $true$. 

□
2.4.3 Tetrahedron Free

In order to examine if a graph $G = (V, E)$ will contain a tetrahedron after the edge $e = \{e_0, e_1\}$ has been inserted in $G$, the method $\text{tetrahedronExists}(G, e)$ goes through all pair of the vertices $<v_0, v_1>$ where $v_0 \in V \setminus \{e_0, e_1\}$ and $v_1 \in V \setminus \{e_0, e_1, v_0\}$ and examines if the edges $\{e_0, v_0\}$, $\{e_1, v_0\}$, $\{e_0, v_1\}$, $\{e_1, v_1\}$ and $\{v_0, v_1\}$ exist in $E$. If that is the case then a tetrahedron will be introduced in $G$ if $e$ is inserted in $G$, see figure 5:

![Figure 3: The edges $\{e_0, e_1\}$, $\{e_0, v_0\}$, $\{e_1, v_0\}$, $\{e_0, v_1\}$, $\{e_1, v_1\}$ and $\{v_0, v_1\}$ form a tetrahedron between the vertices $e_0$, $e_1$, $v_0$, and $v_1$.](image)

The method only returns $true$ if it can find four vertices that can be combined into a tetrahedron if $e$ is inserted in $G$, see example 2.8, otherwise the method returns $false$. The algorithm 5 used in the method $\text{tetrahedronExists}(G = (V, E), e = \{e_0, e_1\})$ performs these steps:

- The loop at the lines 1-7 traverses through all the vertices in the set $V \setminus \{e_0, e_1\}$ and executes these steps:
  - The loop at the lines 2-6 traverses through the vertices in the set $V \setminus \{e_0, e_1, v_0\}$ and tests if the edges $\{e_0, v_0\}$, $\{e_1, v_0\}$, $\{e_0, v_1\}$, $\{e_1, v_1\}$ and $\{v_0, v_1\}$ exist in $E$. If this is true then the graph will contain a tetrahedron between the vertices $e_0$, $e_1$, $v_0$ and $v_1$ if the edge $\{e_0, e_1\}$ is inserted in the graph. In this case the method returns $true$ at the line 4.

- If a tetrahedron was not found during the execution of the loop at the lines 1-7, then the edge $\{e_0, e_1\}$ can be inserted in $G$ without introducing a tetrahedron in $G$. In this case the method returns $false$ at the line 8.

Algorithm 5 $\text{tetrahedronExists}(G=(V, E), e = \{e_0, e_1\})$

1: for all $v_0 \in V \setminus \{e_0, e_1\}$ do
2:   for all $v_1 \in V \setminus \{e_0, e_1, v_0\}$ do
3:     if $\{e_0, v_0\} \in E \land \{e_1, v_0\} \in E \land \{e_0, v_1\} \in E \land \{e_1, v_1\} \in E \land \{v_0, v_1\} \in E$ then
4:       return $true$
5:     end if
6:   end for
7: end for
8: return $false$

Example 2.8. This happens when the method $\text{tetrahedronExists}(G, e)$ examines if it possible to insert the edge $e = \{2, 3\}$ in the graph $G = (\{1, 2, 3, 4\}, \{\{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 4\}, \{3, 4\}\})$ without introducing a tetrahedron in the graph:
Given:
\[ G = (\{1, 2, 3, 4\}, \{\{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 4\}, \{3, 4\}\}) \]
and \( e = \{2, 3\} \)

In the first iteration of the loops in the lines 1-7, the loops examine the vertices 1 and 2. In this case the loops find the edges \{1, 2\}, \{1, 3\}, \{1, 4\} and \{2, 4\}.

Since the edges \{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 4\} and \{2, 3\} do not form a tetrahedron in the graph, the loops will now continue with the next pair of vertices, 1 and 3.

In the second iteration of the loops, the loops examine the vertices 1 and 3. In this case the loops find the edges \{1, 2\}, \{1, 3\}, \{1, 4\} and \{3, 4\}.

Since the edges \{1, 2\}, \{1, 3\}, \{1, 4\}, \{3, 4\} and \{2, 3\} do not create a tetrahedron the loops will now examining the next pair of vertices, 1 and 4.

In the third iteration of the loops, the loops examine the vertices 1 and 4. In this case the loops find the edges \{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 4\} and \{3, 4\}.

Since the edges \{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 4\}, \{3, 4\} and \{2, 3\} form a tetrahedron in the graph the method will now return true.

\[ \square \]

### 2.4.4 Octahedron Free

In order to examine if a graph \( G = (V, E) \) will contain an octahedron after the edge \( e = \{e_0, e_1\} \) has been inserted in \( G \), the method \( \text{octahedronExists}(G, e) \) goes through all the vertices in \( V \) and all the edges in \( E \) and tries to find six vertices in the graph that can be combined into an octahedron, see figure 4.
The method only returns true if it can find six vertices that can be combined into an octahedron if $e$ is inserted in $G$, see example 2.9, otherwise the method returns false. The algorithm 6 used in the method octahedronExists($G, e$) performs these steps:

- The loop at the lines 1-15 traverses through all the vertices in the set $V \{e_0, e_1\}$ and executes these steps:
  - The loop at the lines 2-14 traverses through the vertices in the set $V \{e_0, e_1, v_0\}$ and tests if the edges $\{e_0, v_0\}, \{e_0, v_1\}, \{v_0, v_1\}$ and $\{e_1, v_0\}$ exist in $E$, see figure 5.

- If the edges exist in E, then the algorithm continues with executing the loop at the lines 4 - 12 and tests if the edges $\{e_0, v_2\}, \{v_1, v_2\}$ and $\{v_2, e_1\}$ exist in $E$, see figure 6.
Figure 6: The edges \( \{e_0, v_2\}, \{v_1, v_2\} \text{ and } \{v_2, e_1\} \) exist in the graph.

- If the edges exist in \( E \), then the algorithm continues with executing the loop at the lines 6 - 10 and tests if the edges \( \{e_1, v_3\}, \{v_0, v_3\}, \{v_1, v_3\} \text{ and } \{v_2, v_3\} \) exist in \( E \), see figure 7.

Figure 7: The edges \( \{e_1, v_3\}, \{v_0, v_3\}, \{v_1, v_3\} \text{ and } \{v_2, v_3\} \) exists in the graph.

- If the edges exists in the graph, then an octahedron has been found and \textit{true} is returned at the line 8.

- If a octahedron was not found during the execution of the loops at the lines 1-15, then the method returns \textit{false} at the line 16.
The pseudocode for the method `octahedronExists(G, e)` is:

```
Algorithm 6 octahedronExists(G=(V, E), e = \{e_0, e_1\})
1: for all v_0 ∈ V \ {e_0, e_1} do
2:   for all v_1 ∈ V \ {e_0, e_1, v_0} do
3:     if \{e_0, v_0\} ∈ E \{e_0, v_1\} ∈ E \{v_0, v_1\} ∈ E \{e_1, v_0\} ∈ E then
4:       for all v_2 ∈ V \ {e_0, e_1, v_0, v_1} do
5:         if \{e_0, v_2\} ∈ E \{v_0, v_2\} ∈ E \{v_1, v_2\} ∈ E \{e_1, v_1\} ∈ E then
6:           for all v_3 ∈ V \ {e_0, e_1, v_0, v_1, v_2} do
7:             if \{e_1, v_3\} ∈ E \{v_0, v_3\} ∈ E \{v_1, v_3\} ∈ E \{v_2, v_3\} ∈ E then
8:               return true
9:           end if
10:         end for
11:       end if
12:     end for
13:   end if
14: end for
15: return false
```

**Example 2.9.** This happens when the method `octahedronExists(G, e)` examines if it possible to insert the edge \(e = \{2, 3\}\) in the graph \(G = (\{1, 2, 3, 4, 5, 6\}, \{(1, 2), (1, 3), (1, 4), (1, 5), (2, 5), (2, 6), (3, 4), (3, 6), (4, 5), (4, 6), (5, 6)\})\):

Given:
\(G = (\{1, 2, 3, 4, 5, 6\}, \{(1, 2), (1, 3), (1, 4), (1, 5), (2, 5), (2, 6), (3, 4), (3, 6), (4, 5), (4, 6), (5, 6)\})\)
and \(e = \{2, 3\}\)
In the first iteration of the loops at the lines 1-15, the vertices 1, 2, 3 and 4 are examined. In this case the loops find the edges \{1, 2\}, \{1, 3\}, \{1, 4\}, \{1, 5\}, \{2, 5\}, \{2, 6\}, \{3, 4\}, \{3, 6\}, \{4, 5\} and \{4, 6\}.

Since the edges \{1, 2\}, \{1, 3\}, \{1, 4\}, \{1, 5\}, \{2, 5\}, \{2, 6\}, \{3, 4\}, \{3, 6\}, \{4, 5\} and \{2, 3\} do not form an octahedron in the graph, the loops will now continue with examining the vertices 1, 2, 3 and 5.

In the second iteration of the loops, the vertices 1, 2, 3 and 5 are examined. In this case the loops find the edges \{1, 2\}, \{1, 3\}, \{1, 4\}, \{1, 5\}, \{2, 5\}, \{2, 6\}, \{3, 4\}, \{3, 6\}, \{4, 5\} and \{5, 6\}.

Since the edges \{1, 2\}, \{1, 3\}, \{1, 4\}, \{1, 5\}, \{2, 5\}, \{2, 6\}, \{3, 4\}, \{3, 6\}, \{4, 5\}, \{5, 6\} and \{2, 3\} do not form an octahedron in the graph, the loops will now continue with examining the vertices 1, 2, 3 and 6.
In the third iteration, the method examines the vertices 1, 2, 3 and 6. In this case the loops find the edges \{1,2\}, \{1,3\}, \{1,5\}, \{2,5\}, \{2,6\}, \{3,4\}, \{3,6\}, \{4,5\} and \{4,6\}.

Since the edges \{1,2\}, \{1,3\}, \{1,4\}, \{1,5\}, \{2,5\}, \{2,6\}, \{3,4\}, \{3,6\}, \{4,5\}, \{4,6\} and \{2,3\} form an octahedron in the graph, the method will now return true.

2.5 Graph colouring

The algorithm used for colouring a graph is a backtracking version of DSATUR (Degree of Satisfac-tion) known as BSC(Backtracking Sequential Colouring). According to (Klotz, 2002) this algorithm finds the chromatic number for a given graph. The algorithm is sequential and starts with sorting the vertices according to their degrees, that is the number of edges incident to a vertex, in a non-increasing order, see figure 8.

Figure 8: The vertices in this graph are sorted in the order 1,2,4 and 3.

After that the algorithm assigns the colour with the smallest index, \(c_1\), to the vertex with the highest degree. In the next step the algorithm searches for the uncoloured vertex with the highest saturation, that is the vertex with the highest number of different colours found in the adjacent vertices, and colours it with the colour with the smallest index that does not exist in the neighbourhood of the vertex. This procedure is repeated until all the vertices in the graph has been coloured. After that the number of colours, \(c_{\text{max}}\), that have been used to colour the graph is stored and the algorithm continues with searching for the first vertex, \(v\), that was coloured with \(c_{\text{max}}\). Once \(v\) has been found, all the vertices that were coloured after \(v\) are uncoloured. In the next step the algorithm colours all the uncoloured vertices, but the colours that can be used must have an index that is less than or equal to \(c_{\text{max}}\) (Klotz, 2002).
The algorithm 7 used in BSC performs these steps:

- At the line 1 the array $V_c$ in which all the colours given to each vertex in the graph will be stored in is created.

- The vertices in the graph are sorted according to their degrees in non-increasing order at the line 2 and the order is stored in the array $V_s$.

- The variable $c_{max}$ contains the maximum number of colours required to colour the graph. In the worst case scenario, the number of colours are the same as the number of vertices in the graph. Since the graph has not been coloured yet, $c_{max}$ is assigned to $|V|$ at the line 3.

- The starting index is stored in the variable $start$ at the line 4.

- The vertex with the highest degree is stored in the variable $v$ at the line 5.

- The array $C$ which will contain the number of different colours used so far when the algorithm has reached a vertex is created at the line 5.

- At the line 8 the set of free colours, $C_{free}$, is created and in the initial step it contains, $c_1$, which is the colour with the smallest index.

- The loop at the lines 9 - 43 performs these steps:
  - The flag $back$ is set to $false$ at the line 10. This flag will be set to true if the algorithm cannot find a colour that can be used to colour a vertex in the graph.
  - The loop at the lines 11 - 24 performs these steps:
    - if $i > start$ then the vertex with the highest saturation and the colours that can be used for colouring are stored in the variables $v_s$ respective $C_{free}$ at the lines 13 and 14.
    - if $|C_{free}| > 0$ then $v_s$ is coloured with the colour with the smallest index found in $C_{free}$ at the line 17. At the line 18, the number of colours that have been used so far is stored in $C$.
    - if $|C| \leq 0$ then no colour that can be used for colouring has been found and the algorithm must now start to backtrack. Before the backtracking, the variables start and back are set to $i - 1$ respective true and after that the for-loop breaks at the line 20.
  - if $back$ is true and if $start \geq 0$ then the vertex at the index $start$ in the array $V_s$ is uncoloured at the line 25.
  - if $back$ is false then the algorithm managed to colour the graph. The colouring of the given graph is stored in the array $C_{res}$ and the number of colours needed to colour the graph is stored in $c_{max}$.
  - At the lines 30 - 33 the smallest index in $C_{res}$ which contain a vertex coloured with $c_{max}$ is found and stored in $i$.
  - At the line 34, $start$ is set to $i - 1$ and as a result, the algorithm will backtrack one step.
  - If $start < 0$ then the while-loop at the lines 9 - 43 breaks at the line 36.
  - The loop at the lines 38-40 loops through the vertices in $V_c$ and uncolours all vertices at the indices $[start, |V|]$.
  - At the line 41, $v_s$ is set to the new starting vertex located in $V_s[start]$.

- The optimal colouring for the given graph is returned at the line 44.
Algorithm 7 BSC(G = (V, E))

1: \( V_c = \text{new ARRAY}[|V|] \)
2: \( V_s \leftarrow \text{sortAccordingToTheHighestDegree}(V) \)
3: \( c_{max} = |V| \)
4: \( \text{start} = 0 \)
5: \( v_s = V_s[0] \)
6: \( C = \text{new ARRAY}[|V| + 1] \)
7: \( C[0] = c_1 \)
8: \( C_{free} = \{c_1\} \)
9: \textbf{while} \( \text{start} \geq 0 \) \textbf{do}
10: \( \text{back} = \text{false} \)
11: \textbf{for all} \( i \in [\text{start}, |V|) \) \textbf{do}
12: \textbf{if} \( i > \text{start} \) \textbf{then}
13: \( v_s \leftarrow \text{getVertexWithHighestSaturation}(G, V_s) \)
14: \( C_{free} = \text{getFreeColours}(V_c, N_G(v_s), c_{max}) \)
15: \textbf{end if}
16: \textbf{if} \( |C_{free}| > 0 \) \textbf{then}
17: \( V_c[v_s] = C_{free}.\text{first()} \)
18: \( C[i + 1] = \text{Math.max}(C_{free}.\text{first()}, C[i]) \)
19: \textbf{else}
20: \( \text{start} = i - 1; \text{back} = \text{true}; \text{break} \)
21: \textbf{end if}
22: \textbf{end for}
23: \textbf{if} \( \text{back} \) \textbf{then}
24: \textbf{if} \( \text{start} \geq 0 \) \textbf{then}
25: \( v_s = V_s[\text{start}]; V_c[v_s] = \text{nil} \)
26: \textbf{end if}
27: \textbf{else}
28: \( C_{res} = V_c \)
29: \( c_{max} = C[|V|] \)
30: \( i = 0 \)
31: \textbf{while} \( C_{res}[i] \neq c_{max} \) \textbf{do}
32: \( i++ \)
33: \textbf{end while}
34: \( \text{start} = i - 1 \)
35: \textbf{if} \( \text{start} < 0 \) \textbf{then}
36: \( \text{break} \)
37: \textbf{end if}
38: \textbf{for all} \( j \in [\text{start}, |V|] \) \textbf{do}
39: \( V_c[j] = \text{nil} \)
40: \textbf{end for}
41: \( v_s = V_s[\text{start}] \)
42: \textbf{end if}
43: \textbf{end while}
44: \textbf{return} \( C_{res} \)
The method \textit{getVertexWithHighestSaturation}(G = (V, E), V_s) used in the algorithm 7 to find the uncoloured vertex with the highest saturation performs these steps:

- At the beginning of the method, no vertices have been examined yet, so the variables that keep track of the highest value of saturation found so far, \( \hat{d} \), and which vertex that has this value, \( v_{\hat{d}} \), are both set to 0 at the lines 1 and 2.
- The loop at the lines 3-8 examines all the vertices \( v_s \in V_s \) and performs this step:
  - The condition at the line 4 tests if \( v_s \) has a greater saturation than the saturation found so far. If this is the case then the values of \( v_{\hat{d}} \) and \( \hat{d} \) and are replaced with \( v_s \) and its saturation \( d(v_s) \) at the lines 5 and 6.
- After the loop at the lines 3 - 8 has examined all vertices in \( V_s \), the vertex with the highest saturation, \( v_{\hat{d}} \), is returned at the line 9.

\begin{algorithm}
\textbf{Algorithm 8} getVertexWithHighestSaturation(G = (V, E), V_s)
\begin{algorithmic}[1]
\STATE \( \hat{d} = 0 \)
\STATE \( v_{\hat{d}} = 0 \)
\FORALL {\( v_s \in V_s \)}
\IF {\( d(v_s) > \hat{d} \)}
\STATE \( v_{\hat{d}} = v_s \)
\STATE \( \hat{d} = d(v_s) \)
\ENDIF
\ENDFOR
\RETURN \( v_{\hat{d}} \)
\end{algorithmic}
\end{algorithm}

The method \textit{getFreeColours}(V_c, V_n, c_{max}) used by the BSC algorithm to find a free colour with the smallest index that can be used to colour a vertex performs these steps:

- At the beginning of the algorithm, no colours have been examined yet, so the variables \( C_{\text{free}} \) and \( c_i \) are set to the empty set and the colour with the smallest index at the lines 1 and 2.
- The set \( C_{N_G(v)} \) which contains all the colours found in the neighbourhood \( N_G(v) \) is created at the lines 3 - 6.
- The loop at the lines 7 - 12 is executed as long as the colour \( c_i \) has a smaller index than the colour \( c_{\text{max}} \). During each iteration \( c_i \) is added to \( C_{\text{free}} \) if \( c_i \not\in C_{N_G(v)} \). Before the next iteration in the loop is executed, the current value of \( c_i \) is replaced with the next colour \( c_{i+1} \). When the index of \( c_i \) is the same as the index for \( c_{\text{max}} \), then the colours in \( C_{\text{free}} \) is returned at the line 13.
**Algorithm 9** getFreeColours($V_c, N_G(v), c_{max}$)

1: $C_{free} = \emptyset$
2: $c_i = c_1$
3: $C_{N_G(v)} = \emptyset$
4: for all $v_n \in N_G(v)$ do
5:   $C_{N_G(v)} = C_{N_G(v)} \cup \{V_c[v_n]\}$
6: end for
7: while $c_i < c_{max}$ do
8:   if $c_i \not\in C_{N_G(v)}$ then
9:     $C_{free} = C_{free} \cup \{c_i\}$
10: end if
11: $c_i = c_{i+1}$
12: end while
13: return $C_{free}$

**Example 2.10.** The BSC-algorithm starts with sorting the vertices in the graph according their degrees, that is the number of neighbours a vertex has, in non-increasing order. After that the set $C_{free}$ which contains the colours that can be used for colouring is set to $\{c_1\}$:

**Given:**
$G = (\{1, 2, 3, 4\}, \{\{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 4\}\})$

The vertices are sorted according to their degrees:
$V_{sorted} = \{1, 2, 4, 3\}$

The vertices of free colours that can be used for colouring is created:
$C_{free} = \{c_1\}$

The array which contains the number of colours that have been used is also created:
$C = [c_1, nil, nil, nil, nil]$  

After that the vertex with the highest degree is coloured with $c_1$ and the algorithm continues with finding the vertex with the highest saturation and colours it with the colour with the smallest index that does not exist in the neighbourhood of the vertex. This procedure continues until all the vertices in the graph have been coloured:

In the first iteration, the uncoloured node with the highest saturation, 1, is coloured with the colour $c_1$ which does not exist in its neighbourhood. After that the array which contains the number of colours that has been used is updated:
$C = [c_1, c_1, nil, nil, nil]$
In the second iteration, the uncoloured node with the highest saturation, 2, is coloured with $c_2$ which is the colour with the smallest index in $C_{\text{free}} = \{c_2, c_3, c_4\}$. After that the uncoloured vertex with the highest saturation is found and new free colours are found in its neighbourhood:

$v_s = 4$

$C_{\text{free}} = \{c_3, c_4\}$

The array which contains the number of colours that has been used is also updated:

$C = [c_1, c_1, c_2, \text{nil}, \text{nil}]$

In the third iteration, the uncoloured node with the highest saturation, 4, is coloured with $c_3$ which is the colour with the smallest index in $C_{\text{free}}$. After that the uncoloured vertex with the highest saturation is found and new free colours are found in its neighbourhood:

$v_s = 3$

$C_{\text{free}} = \{c_2, c_3, c_4\}$

The array which contains the number of colours that has been used also is updated:

$C = [c_1, c_1, c_2, c_3, \text{nil}]$

In the fourth iteration the uncoloured node with the highest saturation, 3, is coloured with $c_2$ which is the colour with the smallest index in $C_{\text{free}}$.

$C_{\text{free}} = \{c_2, c_3, c_4\}$

The array which contains the number of colours has been used is also updated:

$C = [c_1, c_1, c_2, c_3, c_2]$}

After that the algorithm examines if it should backtrack and try to recolour some of the vertices in the graph. After it has examined all possible colourings on the graph, the graph coloured with the smallest number of colours is returned:
2.6 Counting edges

Since the random graph is stored in a square matrix the number of edges in the random graph can be calculated by counting the number of 1:s in the upperhalf of the matrix.

The algorithm 10 used in the method numberOfEdges($G = (V, E)$) performs these steps:

- In the initial step at the line 1, the variable $\#edges$ in which the number of 1:s found in the upperhalf of the matrix will be stored in, is set to 0.
- The loops at the lines 2 - 8 traverse through each row in the square matrix and performs this step:
  - The condition at the line 4 tests if the value stored in the cell $a_{ij}$ is 1. If that is the case, then $\#edges$ is increased by 1 at the line 5.
- After the loops at the lines 2 - 8 has counted the number of 1:s in the upperhalf of the matrix, the value in $\#edges$ is returned.

Algorithm 10 $\text{numberOfEdges}(G=(V,E))$

1: $\#edges \leftarrow 0$
2: for $i := 1$ to $|V| - 1$ do
3:  for $j := i + 1$ to $|V|$ do
4:  if $a_{ij} = 1$ then
5:  $\#edges ++$
6: end if
7: end for
8: end for
9: return $\#edges$

Example 2.11. In the initial step of the algorithm the number of edges that have been found is 0:

$G = (\{1, 2, 3, 4\}, \{\{1, 2\}, \{1, 4\}, \{2, 3\}, \{3, 4\})$

$\#edges = 0$

In the first iteration, the algorithm traverses through the first row in the matrix which contains the elements $a_{12} = 1$, $a_{13} = 0$ and $a_{14} = 1$ and founds two elements that have the value 1. Since the value 1 means that there exists an edge from the vertex number 1 to another vertex in the graph, the variable $\#edges$ is increased by 2 in this case:
In the second iteration, the algorithm traverses through the second row in the matrix which contains the elements $a_{23} = 1$ and $a_{24} = 0$. Since the algorithm founds one element that has the value 1 the variable $\#edges$ is increased by 1:

\[
G = (\{1, 2, 3, 4\}, \{\{1, 2\}, \{1, 4\}, \{2, 3\}, \{3, 4\}\})
\]

$\#edges = 2$

In the third iteration, the algorithm traverses through the third row in the matrix which contains the element $a_{34} = 1$. Since the element has the value 1 the variable $\#edges$ is increased by 1:

\[
G = (\{1, 2, 3, 4\}, \{\{1, 2\}, \{1, 4\}, \{2, 3\}, \{3, 4\}\})
\]

$\#edges = 3$

After the algorithm has traversed the upperhalf of the matrix, the number of edges, 4, is returned.

\[
\]

2.7 Counting triangles

The algorithm for calculating the number of triangles in a graph is based on the recursive version of the depth first search-algorithm.

The algorithm performs these steps:

- At the line 1 the set $T$, which will contain all the triangles found in the graph, is set to the empty set ($\emptyset$).
- The loop at the lines 2-5 traverses through all $v \in V$ and executes these steps:
– The set \( \{v\} \) is assigned to the variable \( V_v \) which now contains the vertex in the graph the algorithm starts the search for triangles in.

– At the line 4 the recursive method \( findAllTriangles(G, V_v) \) is called. This method returns all triangles that are found in the graph when the search is started at the vertex \( v \) and all triangles returned by the method are added to \( T \).

• After the loop at the lines 2-5 has found all the triangles in the graph, the cardinality of \( |T| \) is returned at the line 6.

Algorithm 11 countTriangles(G=(V, E))

```
1: \( T = \emptyset \)
2: for all \( v \in V \) do
3: \( V_v = \{v\} \)
4: \( T = T \cup findAllTriangles(G, V_v) \)
5: end for
6: return \( |T| \)
```

The algorithm 12 used in the method \( findAllTriangles(G = (V, E), V_v) \) performs these steps:

• At the line 1 the set \( T \), which will contain all the triangles found in the graph when the search starts at the vertex \( v \), is empty.

• The condition at the line 2 tests if \( V_v \) contains at most three vertices. If the condition is true then these steps are performed:

  – The loop at the lines 3 - 7 examines all neighbours to the last visited vertex \( V_v.last() \) and performs these steps:

    * The condition at the line 4 tests if one of the vertices in the neighbourhood of the last visited vertex is the same as the vertex the algorithm started the search for triangles at, that is \( v \), and if \( V_v \) contains exactly three vertices. If this is the case then a triangle has been found and is formed by the vertices in \( V_v \). The set \( V_v \) is therefore added to \( T \) at the line 6.

    * The condition at the line 7 tests if \( V_v \) contains less than three vertices and the vertex \( v_n \) does not exist in \( V_v \). If this is the case then a recursive call is made with the arguments \( G \) and \( V_v \cup \{v_n\} \) at the line 8. The reason why \( v_n \) has been added to \( V_v \) is because the method has now visit \( v_n \). The recursive call returns the triangles it has found and they are added to \( T \).

• The set of triangles that have been found by the method is returned at the line 12.
Algorithm 12 findAllTriangles(G=(V, E), V_v)

1: $T = \emptyset$
2: if $|V_v| \leq 3$ then
3: for all $v_n \in N_G(V_v.last())$ do
4: if $V_v.head() = v_n \land |V_v| = 3$ then
5: $T = T \cup \{V_v\}$
6: end if
7: if $|V_v| < 3 \land v_n \not\in V_v$ then
8: $T = T \cup findAllTriangles(G, V_v \cup \{v_n\})$
9: end if
10: end for
11: end if
12: return $T$

Example 2.12. This happens when the algorithm $findTriangles(G, V_v)$ is executed on the graph $G = (\{1, 2, 3, 4\}, \{\{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 4\}, \{3, 4\}\})$ and starts at the vertex 1:

In the initial step is the set of triangles, $T$, is empty:

$$G = (\{1, 2, 3, 4\}, \{\{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 4\}, \{3, 4\}\})$$
$$T = \emptyset$$
$$V_v = \{1\}$$

The set $V_v$ contains only one vertex so the algorithm continues with finding all the neighbours to vertex 1 and stores them in the set $N_G$:

$$G = (\{1, 2, 3, 4\}, \{\{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 4\}, \{3, 4\}\})$$
$$T = \emptyset$$
$$V_v = \{1\}$$
$$N_G(V_v.last()) = \{2, 3, 4\}$$

After that a recursive call is made with the arguments $G$ and $V_v = \{1, 2\}$:

$$G = (\{1, 2, 3, 4\}, \{\{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 4\}, \{3, 4\}\})$$
$$T = \emptyset$$
$$V_v = \{1, 2\}$$
The set $V_v$ contains only two vertices so the algorithm continues with finding all the neighbours to vertex 2 and stores them in the set $N_G$:

$$G = (\{1, 2, 3, 4\}, \{\{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 4\}, \{3, 4\}\})$$
$$T = \emptyset$$
$$V_v = \{1, 2\}$$
$$N_G(V_v..last()) = \{1, 4\}$$

After that a recursive call is made with the arguments $G$ and $V_v = \{1, 2, 4\}$:

$$G = (\{1, 2, 3, 4\}, \{\{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 4\}, \{3, 4\}\})$$
$$T = \emptyset$$
$$V_v = \{1, 2, 4\}$$

The set $V_v$ contains three vertices so the algorithm continues with finding all the neighbours to vertex 4 and stores them in the set $N_G$:

$$G = (\{1, 2, 3, 4\}, \{\{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 4\}, \{3, 4\}\})$$
$$T = \emptyset$$
$$V_v = \{1, 2, 4\}$$
$$N_G(V_v..last()) = \{1, 2, 3\}$$

Since the first vertex in the set $N_G$ is the same vertex the algorithm started its search for triangles at and $|V_v| = 3$, the algorithm has now found a triangle and stores it in the set $T$:

$$G = (\{1, 2, 3, 4\}, \{\{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 4\}, \{3, 4\}\})$$
$$T = \{\{1, 2, 4\}\}$$
$$V_v = \{1, 2, 4\}$$
$$N_G(V_v..last()) = \{1, 2, 3\}$$

Since the number of vertices in $V_v$ is more than two vertices, the recursive call does not continue with examining the neighbours in $N_G$, instead it returns the set $T$: 

37
\[ G = (\{1, 2, 3, 4\}, \{\{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 4\}, \{3, 4\}\} ) \]
\[ T = \{1, 2, 4\} \]
\[ V_v = \{1\} \]
\[ N_G(V_v.\text{last}()) = \{1, 4\} \]

Since the algorithm has visited all the neighbours of the vertex 2 it returns the set \( T \):

\[ G = (\{1, 2, 3, 4\}, \{\{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 4\}, \{3, 4\}\} ) \]
\[ T = \{1, 2, 4\} \]
\[ V_v = \{1\} \]
\[ N_G(V_v.\text{last}()) = \{2, 3, 4\} \]

After that the algorithm continues with examining if a triangle can be found if it visits the neighbour with the number 3 so it therefore makes a recursive call with the arguments \( G \) and \( V_v = \{1, 3\} \):

\[ G = (\{1, 2, 3, 4\}, \{\{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 4\}, \{3, 4\}\} ) \]
\[ T = \{1, 2, 4\} \]
\[ V_v = \{1, 3\} \]

Since the set \( V_v \) contains two vertices, the algorithm continues with finding the neighbours of the vertex 3:

\[ G = (\{1, 2, 3, 4\}, \{\{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 4\}, \{3, 4\}\} ) \]
\[ T = \{1, 2, 4\} \]
\[ V_v = \{1, 3\} \]
\[ N_G(V_v.\text{last}()) = \{1, 4\} \]

In the next step, the algorithm makes a recursive call with the arguments \( G \) and \( V_v = \{1, 3, 4\} \):
The set $V_v$ contains three vertices so the algorithm continues with finding all the neighbours to vertex 4 and stores them in the set $N_G$:

Since the first vertex in the set $N_G$ is the same vertex the algorithm started its search for triangles at and $|V_v| = 3$, the algorithm has now found another triangle and adds it to the set $T$:

Since the number of vertices in $V_v$ is more than two vertices, the recursive call does not continue with examining the neighbours in $N_G$, instead it returns the set $T$:

Since the algorithm has now visit all edges in the set $V_v$, it returns $T$:
After that the algorithm continues with making a recursive call with the arguments $G$ and $V_v = \{1, 4\}$. Since this recursive call results in a set of triangles consisting of the triangles $T = \{\{1, 2, 4\}, \{1, 3, 4\}\}$, this call does not find any new triangles in the graph. In the final step the algorithm returns the triangles it has found in the graph:

$$T = \{\{1, 2, 4\}, \{1, 3, 4\}\}$$
3 Experiments

The program has been used to generate random graphs both according to edges and according to vertices and to examine the number of colours needed to colour each graph and the number of edges and triangles in each graph. The table 1 contains an overview of arguments given to the program to generate triangle free graphs. According to the table the first type of generated random graphs consisted of 100 vertices, the edges were inserted in the graph with the probability 1 if a triangle was not introduced in the graph. According to the second row in the table the second type of generated graphs consists 100 vertices, the edges were inserted in the graph with the probability $\frac{1}{2}$ if a triangle was not introduced in the graph. The rest of the rows in the table and in the tables 2, 3 and 4 follow the same overview over the arguments and each type of graphs was generated 100 times based on edges and 100 times based on vertices.

| | | | | | | Probability | Forbidden structure |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | triangle | four cycle | tetrahedron | octahedron |
| 100 | 200 | 300 | 400 | 500 | | | | | |
| x | 1 | | | | | | | | x |
| x | $\frac{1}{2}$ | | | | | | | | x |
| x | $\frac{1}{\sqrt{|V|}}$ | | | | | | | | x |
| x | $\frac{1}{n}$ | | | | | | | | x |
| x | 1 | | | | | | | | x |
| x | $\frac{1}{2}$ | | | | | | | | x |
| x | $\frac{1}{\sqrt{|V|}}$ | | | | | | | | x |
| x | $\frac{1}{n}$ | | | | | | | | x |
| x | 1 | | | | | | | | x |
| x | $\frac{1}{2}$ | | | | | | | | x |
| x | $\frac{1}{\sqrt{|V|}}$ | | | | | | | | x |
| x | $\frac{1}{n}$ | | | | | | | | x |
| x | 1 | | | | | | | | x |
| x | $\frac{1}{2}$ | | | | | | | | x |
| x | $\frac{1}{\sqrt{|V|}}$ | | | | | | | | x |
| x | $\frac{1}{n}$ | | | | | | | | x |

Table 1: This table contains an overview over the arguments given to the program during the generation of triangle free graphs.
<table>
<thead>
<tr>
<th>size</th>
<th>Probability</th>
<th>Forbidden structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>200</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>300</td>
<td>$\frac{1}{2}$</td>
<td>x</td>
</tr>
<tr>
<td>400</td>
<td>$\frac{1}{\sqrt{</td>
<td>V</td>
</tr>
<tr>
<td>500</td>
<td>$\frac{1}{n}$</td>
<td>x</td>
</tr>
<tr>
<td></td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td></td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td></td>
<td>$\frac{1}{2}$</td>
<td>x</td>
</tr>
<tr>
<td></td>
<td>$\frac{1}{\sqrt{</td>
<td>V</td>
</tr>
<tr>
<td></td>
<td>$\frac{1}{n}$</td>
<td>x</td>
</tr>
</tbody>
</table>

Table 2: This table contains an overview over the arguments given to the program during the generation of four cycle free graphs.
Table 3: This table contains an overview over the arguments given to the program during the generation of tetrahedron free graphs.
Table 4: This table contains an overview over the arguments given to the program during the generation of octahedron free graphs.

<table>
<thead>
<tr>
<th>size</th>
<th>Probability</th>
<th>Forbidden structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>x</td>
<td>triangle</td>
</tr>
<tr>
<td>200</td>
<td>x</td>
<td>four cycle</td>
</tr>
<tr>
<td>300</td>
<td>x</td>
<td>tetrahedron</td>
</tr>
<tr>
<td>400</td>
<td>x</td>
<td>octahedron</td>
</tr>
<tr>
<td>500</td>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>size</th>
<th>Probability</th>
<th>Forbidden structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>1</td>
<td>x</td>
</tr>
<tr>
<td>x</td>
<td>$\frac{1}{2}$</td>
<td>x</td>
</tr>
<tr>
<td>x</td>
<td>$\frac{1}{\sqrt{</td>
<td>V</td>
</tr>
<tr>
<td>x</td>
<td>$\frac{1}{n}$</td>
<td>x</td>
</tr>
<tr>
<td>x</td>
<td>1</td>
<td>x</td>
</tr>
<tr>
<td>x</td>
<td>$\frac{1}{2}$</td>
<td>x</td>
</tr>
<tr>
<td>x</td>
<td>$\frac{1}{\sqrt{</td>
<td>V</td>
</tr>
<tr>
<td>x</td>
<td>$\frac{1}{n}$</td>
<td>x</td>
</tr>
<tr>
<td>x</td>
<td>1</td>
<td>x</td>
</tr>
<tr>
<td>x</td>
<td>$\frac{1}{2}$</td>
<td>x</td>
</tr>
<tr>
<td>x</td>
<td>$\frac{1}{\sqrt{</td>
<td>V</td>
</tr>
<tr>
<td>x</td>
<td>$\frac{1}{n}$</td>
<td>x</td>
</tr>
<tr>
<td>x</td>
<td>1</td>
<td>x</td>
</tr>
<tr>
<td>x</td>
<td>$\frac{1}{2}$</td>
<td>x</td>
</tr>
<tr>
<td>x</td>
<td>$\frac{1}{\sqrt{</td>
<td>V</td>
</tr>
<tr>
<td>x</td>
<td>$\frac{1}{n}$</td>
<td>x</td>
</tr>
<tr>
<td>x</td>
<td>1</td>
<td>x</td>
</tr>
<tr>
<td>x</td>
<td>$\frac{1}{2}$</td>
<td>x</td>
</tr>
<tr>
<td>x</td>
<td>$\frac{1}{\sqrt{</td>
<td>V</td>
</tr>
<tr>
<td>x</td>
<td>$\frac{1}{n}$</td>
<td>x</td>
</tr>
</tbody>
</table>
4 Results

This section contains the results of the examination of the graphs. Note that the diagrams in the figures 9, 11, 13 and 15 have been plotted according to the number of vertices and the median values of #colours (# = number of), #edges and #triangles and that the diagrams in the figures 10, 12, 14 and 16 have been plotted according to the number of vertices and the mean values of #colours, #edges and #triangles. The left column of the diagrams contains the results of the graphs generated according to edges and the right column of the diagrams contains the results of the graphs generated according to vertices. The appendices A, B, C and D contains tables over the number colours and median, mean, variance and standard deviation for #edges and #triangles.

4.1 Triangle free graphs

The figure 9 contains diagrams with the median values of the results when triangles were forbidden in the generated random graphs.

(a) #colours needed to colour a triangle free graph generated according to edges.

(b) #colours needed to colour a triangle free graph generated according to vertices.

(c) #edges found in a triangle free graph generated according to edges.

(d) #edges found in a triangle free graph generated according to vertices.

Figure 9: The diagrams a) and c) show the result when the triangle free graphs have been generated according to edges. The diagrams b) and d) show the result when the triangle free graphs have been generated according to vertices.
The figure 10 contains diagrams with the mean values of the results when triangles were forbidden in the generated random graphs.

(a) #colours needed to colour a triangle free graph generated according to edges.

(b) #colours needed to colour a triangle free graph generated according to vertices.

(c) #edges found in a triangle free graph generated according to edges.

(d) #edges found in a triangle free graph generated according to vertices.

Figure 10: The diagrams a) and c) show the result when the triangle free graphs have been generated according to edges. The diagrams b) and d) show the result when the triangle free graphs have been generated according to vertices.
4.2 Four cycle free graphs

The figure 11 contains diagrams with the median values of the results when four cycles were forbidden in the generated random graphs.

(a) #colours needed to colour a four cycle free graph generated according to edges.  
(b) #colours needed to colour a four cycle free graph generated according to vertices.

(c) #edges found in a four cycle free graph generated according to edges.  
(d) #edges found in a four cycle free graph generated according to vertices.

(e) #triangles found in a four cycle free graph generated according to edges.  
(f) #triangles found in a four cycle free graph generated according to vertices.

Figure 11: The diagrams a), c) and e) show the result when the four cycle free graphs have been generated according to edges. The diagrams b), d) and f) show the result when the four cycle free graphs have been generated according to vertices.
The figure 12 contains diagrams with the mean values of the results when four cycles were forbidden in the generated random graphs.

(a) #colours needed to colour a four cycle free graph generated according to edges.
(b) #colours needed to colour a four cycle free graph generated according to vertices.

c) #edges found in a four cycle free graph generated according to edges.
(d) #edges found in a four cycle free graph generated according to vertices.

(e) #triangles found in a four cycle free graph generated according to edges.
(f) #triangles found in a four cycle free graph generated according to vertices.

Figure 12: The diagrams a), c) and e) show the result when the four cycle free graphs have been generated according to edges. The diagrams b), d) and f) show the result when the four cycle free graphs have been generated according to vertices.
4.3 Tetrahedron free graphs

The figure 13 contains diagrams with the median values of the results when tetrahedron were forbidden in the generated random graphs.

(a) #colours needed to colour a tetrahedron free graph generated according to edges.

(b) #colours needed to colour a tetrahedron free graph generated according to vertices.

(c) #edges found in a tetrahedron free graph generated according to edges.

(d) #edges found in a tetrahedron free graph generated according to vertices.

(e) #triangles found in a tetrahedron free graph generated according to edges.

(f) #triangles found in a tetrahedron free graph generated according to vertices.

Figure 13: The diagrams a), c) and e) show the result when the tetrahedron free graphs have been generated according to edges. The diagrams b), d) and f) show the result when the tetrahedron free graphs have been generated according to vertices.
The figure 14 contains diagrams with the mean values of the results when tetrahedrons were forbidden in the generated random graphs.

(a) #colours needed to colour a tetrahedron free graph generated according to edges.  
(b) #colours needed to colour a tetrahedron free graph generated according to vertices.

(c) #edges found in a tetrahedron free graph generated according to edges.  
(d) #edges found in a tetrahedron free graph generated according to vertices.

(e) #triangles found in a tetrahedron free graph generated according to edges.  
(f) #triangles found in a tetrahedron free graph generated according to vertices.

Figure 14: The diagrams a), c) and e) show the result when the tetrahedron free graphs have been generated according to edges. The diagrams b), d) and f) show the result when the tetrahedron free graphs have been generated according to vertices.
4.4 Octahedron free graphs

The figure 15 contains diagrams with the median values of the results when octahedrons were forbidden in the generated random graphs.

(a) #colours needed to colour a octahedron free graph generated according to edges.
(b) #colours needed to colour a tetrahedron free graph generated according to vertices.
(c) #edges found in a octahedron free graph generated according to edges.
(d) #edges found in a octahedron free graph generated according to vertices.
(e) #triangles found in a octahedron free graph generated according to edges.
(f) #triangles found in a octahedron free graph generated according to vertices.

Figure 15: The diagrams a), c) and e) show the result when the octahedron free graphs have been generated according to edges. The diagrams b), d) and f) show the result when the octahedron free graphs have been generated according to vertices.
The figure 16 contains diagrams with the mean values of the results when octahedrons were forbidden in the generated random graphs.

(a) #colours needed to colour a octahedron free graph generated according to edges.

(b) #colours needed to colour a tetrahedron free graph generated according to vertices.

(c) #edges found in a octahedron free graph generated according to edges.

(d) #edges found in a octahedron free graph generated according to vertices.

(e) #triangles found in a octahedron free graph generated according to edges.

(f) #triangles found in a octahedron free graph generated according to vertices.

Figure 16: The diagrams a), c) and e) show the result when the octahedron free graphs have been generated according to edges. The diagrams b), d) and f) show the result when the octahedron free graphs have been generated according to vertices.
5 Analysis of the results

In order to determine the number of colours needed to colour the generated graphs and how many edges and triangles they contain, the values in the different experiments were interpolated with respect to polynomial functions. The tool that have been used to calculate these interpolations is Matlab’s Curve fitting tool. Besides from finding the functions, the tool also calculates how well the function approximates the given points, that is the Goodness of fit, and uses the scores Due to Error(SSE), Ratio of the sum of square of regression(R-square), Degrees of Freedom Adjusted R-square and Root Mean Squared Error(RMSE). The result from Matlab’s Curve fitting tool can be found in the appendices E - H. The tables below contains a short summary of the results. Note that in each expression, \( n \) is the number of edges and \( c \) is a constant that can have different values in different cases.

<table>
<thead>
<tr>
<th>Triangle free graphs generated according to edges</th>
<th>( p = 1 )</th>
<th>( p = \frac{1}{2} )</th>
<th>( p = \frac{1}{\sqrt{n}} )</th>
<th>( p = \frac{1}{n} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>colour edges</td>
<td>Mean</td>
<td>Median</td>
<td>Mean</td>
<td>Mean</td>
</tr>
<tr>
<td>colour edges</td>
<td>( cn^\frac{5}{7} )</td>
<td>( cn^\frac{5}{7} )</td>
<td>( cn^\frac{5}{7} )</td>
<td>( cn^\frac{5}{7} )</td>
</tr>
<tr>
<td>edges</td>
<td>( cn^\frac{5}{7} )</td>
<td>( cn^\frac{5}{7} )</td>
<td>( cn^\frac{5}{7} )</td>
<td>( cn^\frac{5}{7} )</td>
</tr>
</tbody>
</table>

Table 5: The approximated number of colours and edges in the triangle free graphs generated according to edges.

<table>
<thead>
<tr>
<th>Triangle free graphs generated according to vertices</th>
<th>( p = 1 )</th>
<th>( p = \frac{1}{2} )</th>
<th>( p = \frac{1}{\sqrt{n}} )</th>
<th>( p = \frac{1}{n} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>colour edges</td>
<td>Mean</td>
<td>Median</td>
<td>Mean</td>
<td>Mean</td>
</tr>
<tr>
<td>colour edges</td>
<td>( \frac{2}{7} )</td>
<td>( \frac{2}{7} )</td>
<td>( \frac{2}{7} )</td>
<td>( \frac{2}{7} )</td>
</tr>
<tr>
<td>edges</td>
<td>( cn^\frac{2}{7} )</td>
<td>( cn^\frac{2}{7} )</td>
<td>( cn^\frac{2}{7} )</td>
<td>( cn^\frac{2}{7} )</td>
</tr>
</tbody>
</table>

Table 6: The approximated number of colours and edges in the triangle free graphs generated according to vertices.

<table>
<thead>
<tr>
<th>Four cycle free graphs generated according to edges</th>
<th>( p = 1 )</th>
<th>( p = \frac{1}{2} )</th>
<th>( p = \frac{1}{\sqrt{n}} )</th>
<th>( p = \frac{1}{n} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>colour edges</td>
<td>Mean</td>
<td>Median</td>
<td>Mean</td>
<td>Mean</td>
</tr>
<tr>
<td>colour edges</td>
<td>( cn^\frac{7}{7} )</td>
<td>( cn^\frac{7}{7} )</td>
<td>( cn^\frac{7}{7} )</td>
<td>( cn^\frac{7}{7} )</td>
</tr>
<tr>
<td>edges</td>
<td>( cn^\frac{7}{7} )</td>
<td>( cn^\frac{7}{7} )</td>
<td>( cn^\frac{7}{7} )</td>
<td>( cn^\frac{7}{7} )</td>
</tr>
<tr>
<td>triangles</td>
<td>( cn\frac{7}{7} )</td>
<td>( cn\frac{7}{7} )</td>
<td>( cn\frac{7}{7} )</td>
<td>( cn\frac{7}{7} )</td>
</tr>
</tbody>
</table>

Table 7: The approximated number of colours and edges in the four cycle free graphs generated according to vertices. Note that in the case where \( p = \frac{1}{n} \) the exponent in the expression for the number of triangles is negative. This implies that no triangles exist in this kind of graphs.
### Four cycle free graphs generated according to vertices

<table>
<thead>
<tr>
<th>p = 1</th>
<th>p = ( \frac{1}{2} )</th>
<th>p = ( \frac{1}{\sqrt{n}} )</th>
<th>p = ( \frac{1}{n} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>colour</td>
<td>Mean: ( cn^{\frac{2}{5}} ) Median: 4</td>
<td>Mean: ( cn^{\frac{2}{10}} ) Median: ( cn^{\frac{1}{10}} )</td>
<td>Mean: ( cn^{\frac{2}{10}} ) Median: ( cn^{\frac{1}{10}} )</td>
</tr>
<tr>
<td>edges</td>
<td>Mean: ( cn^{\frac{1}{3}} ) Median: ( cn^{\frac{1}{3}} )</td>
<td>Mean: ( cn^{\frac{7}{10}} ) Median: ( cn^{\frac{7}{10}} )</td>
<td>Mean: ( cn^{\frac{7}{10}} ) Median: ( cn^{\frac{7}{10}} )</td>
</tr>
<tr>
<td>triangles</td>
<td>Mean: ( cn^{\frac{2}{5}} ) Median: ( cn^{\frac{2}{5}} )</td>
<td>Mean: ( cn^{\frac{4}{10}} ) Median: ( cn^{\frac{4}{10}} )</td>
<td>Mean: ( cn^{\frac{4}{10}} ) Median: ( cn^{\frac{4}{10}} )</td>
</tr>
</tbody>
</table>

Table 8: The approximated number of colours and edges in the four cycle free graphs generated according to vertices.

### Tetrahedron free graphs generated according to edges

<table>
<thead>
<tr>
<th>p = 1</th>
<th>p = ( \frac{1}{2} )</th>
<th>p = ( \frac{1}{\sqrt{n}} )</th>
<th>p = ( \frac{1}{n} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>colour</td>
<td>Mean: ( cn^{\frac{2}{5}} ) Median: ( cn^{\frac{2}{5}} )</td>
<td>Mean: ( cn^{\frac{2}{5}} ) Median: ( cn^{\frac{2}{5}} )</td>
<td>Mean: ( cn^{\frac{2}{5}} ) Median: ( cn^{\frac{2}{5}} )</td>
</tr>
<tr>
<td>edges</td>
<td>Mean: ( cn^{\frac{1}{3}} ) Median: ( cn^{\frac{1}{3}} )</td>
<td>Mean: ( cn^{\frac{1}{3}} ) Median: ( cn^{\frac{1}{3}} )</td>
<td>Mean: ( cn^{\frac{1}{3}} ) Median: ( cn^{\frac{1}{3}} )</td>
</tr>
<tr>
<td>triangles</td>
<td>Mean: ( cn^{\frac{2}{5}} ) Median: ( cn^{\frac{2}{5}} )</td>
<td>Mean: ( cn^{\frac{1}{3}} ) Median: ( cn^{\frac{1}{3}} )</td>
<td>Mean: ( cn^{\frac{1}{3}} ) Median: ( cn^{\frac{1}{3}} )</td>
</tr>
</tbody>
</table>

Table 9: The approximated number of colours and edges in the tetrahedron free graphs generated according to edges.

### Tetrahedron free graphs generated according to vertices

<table>
<thead>
<tr>
<th>p = 1</th>
<th>p = ( \frac{1}{2} )</th>
<th>p = ( \frac{1}{\sqrt{n}} )</th>
<th>p = ( \frac{1}{n} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>colour</td>
<td>Mean: 3 Median: 3</td>
<td>Mean: ( cn^{\frac{2}{5}} ) Median: ( cn^{\frac{2}{5}} )</td>
<td>Mean: ( cn^{\frac{2}{5}} ) Median: ( cn^{\frac{2}{5}} )</td>
</tr>
<tr>
<td>edges</td>
<td>Mean: ( cn^{\frac{1}{3}} ) Median: ( cn^{\frac{1}{3}} )</td>
<td>Mean: ( cn^{\frac{1}{3}} ) Median: ( cn^{\frac{1}{3}} )</td>
<td>Mean: ( cn^{\frac{1}{3}} ) Median: ( cn^{\frac{1}{3}} )</td>
</tr>
<tr>
<td>triangles</td>
<td>Mean: ( cn^{\frac{1}{3}} ) Median: ( cn^{\frac{1}{3}} )</td>
<td>Mean: ( cn^{\frac{1}{3}} ) Median: ( cn^{\frac{1}{3}} )</td>
<td>Mean: ( cn^{\frac{1}{3}} ) Median: ( cn^{\frac{1}{3}} )</td>
</tr>
</tbody>
</table>

Table 10: The approximated number of colours and edges in the tetrahedron free graphs generated according to vertices.
<table>
<thead>
<tr>
<th>Octahedron free graphs generated according to edges</th>
<th>$p = 1$</th>
<th>$p = \frac{1}{2}$</th>
<th>$p = \frac{1}{\sqrt{n}}$</th>
<th>$p = \frac{1}{n}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>colour</td>
<td>Mean</td>
<td>Median</td>
<td>Mean</td>
<td>Median</td>
</tr>
<tr>
<td>edges</td>
<td>$cn^\frac{5}{2}$</td>
<td>$cn^\frac{5}{2}$</td>
<td>$cn^\frac{5}{2}$</td>
<td>$cn^\frac{5}{2}$</td>
</tr>
<tr>
<td>triangles</td>
<td>$cn^\frac{7}{2}$</td>
<td>$cn^\frac{7}{2}$</td>
<td>$cn^\frac{7}{2}$</td>
<td>$cn^\frac{7}{2}$</td>
</tr>
</tbody>
</table>

Table 11: The approximated number of colours and edges in the octahedron free graphs generated according to edges.

<table>
<thead>
<tr>
<th>Octahedron free graphs generated according to vertices</th>
<th>$p = 1$</th>
<th>$p = \frac{1}{2}$</th>
<th>$p = \frac{1}{\sqrt{n}}$</th>
<th>$p = \frac{1}{n}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>colour</td>
<td>Mean</td>
<td>Median</td>
<td>Mean</td>
<td>Median</td>
</tr>
<tr>
<td>edges</td>
<td>$cn^\frac{1}{20}$</td>
<td>$cn^\frac{1}{20}$</td>
<td>$cn^\frac{1}{20}$</td>
<td>$cn^\frac{1}{20}$</td>
</tr>
<tr>
<td>triangles</td>
<td>$cn^\frac{1}{5}$</td>
<td>$cn^\frac{1}{5}$</td>
<td>$cn^\frac{1}{5}$</td>
<td>$cn^\frac{1}{5}$</td>
</tr>
</tbody>
</table>

Table 12: The approximated number of colours and edges in the octahedron free graphs generated according to vertices.
6 Conclusions and future work

In this project has a computer program been developed that can generate and examine some properties of generated random graphs without some forbidden subgraph. The program was used to generate and examine some properties of triangle free graphs, four cycle free graphs, tetrahedron free graphs and octahedron free graphs. The graphs had 100, 200, 300, 400 or 500 vertices. The probability functions \( p(n) = 1 \), \( p(n) = \frac{1}{2} \), \( p(n) = \frac{1}{\sqrt{n}} \) and \( p(n) = \frac{1}{n} \) were used to determine if an edge should exist in the graph. The generating process was either based on edges or vertices.

According to the analysis of the results of the triangle free graphs the number of colours needed to colour the graphs was 2 when the graphs have been generated according to vertices and with the probability \( p = 1 \). This result was expected because all the generated graphs are formed as a star in this case. In the other cases the number of colours required to colour the graphs was higher than 2. For each one of the edge probabilities \( p(n) = 1 \), \( p(n) = \frac{1}{2} \), \( p(n) = \frac{1}{\sqrt{n}} \) and \( p(n) = \frac{1}{n} \) the average values for the number of edges were \( cn^\alpha \) for some numbers \( \alpha \) and \( c \) which depended on the edge probability and the forbidden graph. The number \( \alpha \) varied between 1 and 2, endpoints excluded.

According to the analysis of the results of the four cycle free graphs the average values of the number of colours needed to colour the graphs were almost \( n^c \), where the value of \( c \) varied between \( \frac{3}{50} \) and \( \frac{1}{3} \). For each one of the edge probabilities \( p(n) = 1 \), \( p(n) = \frac{1}{n} \), \( p(n) = \frac{1}{\sqrt{n}} \) and \( p(n) = \frac{1}{n} \) the average value for the number of edges was \( cn^\alpha \) for some numbers \( \alpha \) and \( c \) which depended on the edge probability and the forbidden graph. The number \( \alpha \) varied between 1 and \( \frac{7}{5} \). The average number of triangles were \( n^\beta \) where \( \beta \) varied between \( -\frac{1}{3} \) and \( \frac{4}{5} \).

According to the analysis of the results of the tetrahedron free graphs the average values of the number of colours needed to colour the graphs were almost \( n^c \), where the value of \( c \) varied between \( \frac{1}{20} \) and \( \frac{1}{2} \). For each one of the edge probabilities \( p(n) = 1 \), \( p(n) = \frac{1}{n} \), \( p(n) = \frac{1}{\sqrt{n}} \) and \( p(n) = \frac{1}{n} \) the average value for the number of edges was \( cn^\alpha \) for some numbers \( \alpha \) and \( c \) which depended on the edge probability and the forbidden graph. The number \( \alpha \) varied between 1 and 2, endpoints excluded. The average number of triangles were \( n^\beta \) where \( \beta \) varied between \( -\frac{2}{5} \) and \( \frac{12}{5} \).

According to the analysis of the results of the octahedron free graphs the average values of the number of colours needed to colour the graphs were almost \( n^c \), where the value of \( c \) varied between \( \frac{1}{20} \) and \( \frac{1}{2} \). For each one of the edge probabilities \( p(n) = 1 \), \( p(n) = \frac{1}{n} \), \( p(n) = \frac{1}{\sqrt{n}} \) and \( p(n) = \frac{1}{n} \) the average value for the number of edges was \( cn^\alpha \) for some numbers \( \alpha \) and \( c \) which depended on the edge probability and the forbidden graph. The number \( \alpha \) varied between 1 and \( \frac{3}{5} \). The average number of triangles were \( n^\beta \) where \( \beta \) varied between \( -\frac{1}{5} \) and \( \frac{6}{5} \).

A conclusion that can be drawn from these experiments is that generated graphs of the kind that have been examined tend to give ”sparse graphs” (the number of edges is atmost \( n^\alpha \) for some \( \alpha < 2 \)) whose chromatic number grows with the number of edges.

It is possible to use any kind of subgraph in the program by giving the program instructions how to detect a specific subgraph in a graph. Instructions how this can be done exists in the manual in the Appendix J. It is also possible to give other probability functions to the program. As a future work, this program can be used to generate other random graphs which, for example, does not contain any cubes and use other probability functions. No extra implementation is required to proceed with these experiments. However, if other properties than the number of edges, the number of triangles and the chromatic numbers are to be explored some implementation is required.
References


