THE CONTAINER METHOD

AND ITS APPLICATIONS

THE CONTAINER METHOD

AND ITS APPLICATIONS

Thesis

to obtain the degree of Bachelor of Science
at the Delft University of Technology,
to be defended publicly on Thursday July 10th, 2025 at 14:15.

by

Joachim ANEMAET

Student number:	5764440	
Project duration:	April 22, 2025 – July 10, 2025	
Thesis committee:	Dr. A. (Anurag) Bishnoi,	TU Delft, supervisor
	A. (Ananth) Ravi,	TU Delft, supervisor
	Dr. J.A.M. (Joost) de Groot,	TU Delft, senior

An electronic version of this thesis is available at https://repository.tudelft.nl.

Contents

LAYMEN'S SUMMARY

Suppose you have n points on a flat surface and that no more than three of them lie on the same straight line. How large a group of points can you always find such that at most two of them lie on a straight line?
This is an example of a problem in discrete geometry. In figure 1, one of these cases is illustrated, namely the case for 6 points. We can check that the size of the largest group of points we can make that has at most two points on a straight line is 4. An example of such a 4 point set is the set A, circled in red. Now the problem wants this number for any arrangement of n points. This specific arrangement minimises the max number of points such that at most two of those points lie on a line. In 1991, Füredi proved that you can always find a subset of size roughly $\sqrt{n \log n}$, but this had an upper bound that was not really tight. That is where the hypergraph container method comes into play. This method gives a strong upper bound on the problem. In this thesis we will look at this container method, how it works and how it can be applied to such problems.

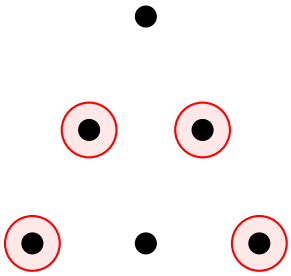


Figure 1: Case $n = 6$

SUMMARY

IN this thesis, we explore the method of hypergraph containers and its applications to problems in extremal combinatorics and discrete geometry. We start by introducing containers on triangle-free graphs and use it to give a lower bound on the number of triangle-free graphs on n vertices. We then generalise to the container algorithm for independent sets in graphs, which is a version of regular graphs. The proof of this theorem is an algorithm. We study this algorithm and apply it to the Erdős-Rényi random graph $G(n,p)$. After this we expand to hypergraphs, specifically 3-uniform hypergraphs. Finally we look at the application to a problem in discrete geometry:
Given n points in the Euclidean plane $\mathbb{R}^2$, with at most three on any line, how large a subset are we guaranteed to find in general position (i.e., with at most two on any line)?
A strong upper bound is obtained by using the hypergraph container method.

Preface and Acknowledgements

THIS thesis was written as part of the Bachelor's final project for the Applied Mathematics programme at TU Delft. It is dedicated to the Hypergraph Container Method and its applications, a topic I chose out of an interest in graph theory, and partly because I had previously enjoyed a course taught by Dr. Anurag Bisnoi.

The topic proved to be abstract and technically demanding, and it took considerable time to gain a proper understanding of the underlying ideas. I am therefore especially grateful to my supervisors, Dr. Anurag Bisnoi and Ananth Ravi, for their continuous guidance throughout the project. The regular meetings were not only intellectually valuable, but also engaging and motivating, and played a key role in helping me stay on track.

I would also like to thank Dr. Joost de Groot for acting as the independent committee member during the defence, and Mark Albertsma for his practical help with LaTeX whenever technical issues arose.

Working on this thesis has been a challenging yet rewarding experience. It gave me the opportunity to explore a rich and intricate topic, and to develop both my mathematical understanding and research skills in the process.

Joachim Anemaet
Delft, July 2025

NOTATION

THROUGHOUT the thesis we will use the following notation.

Definition 0.1 (Asymptotic notation)**.** let f, g be functions from $\mathbb{N}$ to $\mathbb{R}$. We use the following asymptotic notation:

- $f = o(g)$ if $\lim_{n\to\infty} f/g = 0$,
- $f = O(g)$ if there exists a constant $C > 0$ and $n_0 \in \mathbb{N}$ such that $|f(n)| \leq Cg(n)$ for all $n \geq n_0$,
- $f = O_\epsilon(g)$ if for every fixed $\epsilon > 0$, there exists a constant $C = C(\epsilon) > 0$ and $n_0 \in \mathbb{N}$ such that $|f(n,\epsilon)| \leq C(\epsilon) \cdot g(n,\epsilon)$ for all $n \geq n_0$,
- $f = \Omega(g)$ if $g = O(f)$, and
- $f = \Theta(g)$ if $f = O(g)$ and $f = \Omega(g)$.

Definition 0.2. We write $f(n) \ll g(n)$ if $\frac{f(n)}{g(n)} \to 0$ as $n \to \infty$.

Definition 0.3. We write with high probability, or w.h.p., when the probability that the event under consideration occurs tends to 1 as $n \to \infty$. This event should depend on n.

1

INTRODUCTION

THE method of (hypergraph) containers is a powerful tool that can help characterise the typical structure and/or answer extremal questions about families of discrete objects with a prescribed set of local constraints [1]. These questions arise in extremal graph theory, additive combinatorics, discrete geometry, coding theory, and Ramsey theory. These problems usually avoid forbidden structures, such as the family of H-free graphs [1]. We might wonder what we can say about the independent sets of such graphs. This is where the (hypergraph) container method comes into play.

The aim of this thesis is to make this rather technical concept of the hypergraph container method more easily accessible. From section 2.2, we will go through chapter 11 of Yufei Zhao's lecture notes [2]. Here we will build up the needed knowledge for the applications, and also implement the graph container algorithm in Python [2]. We will also be going through an easy application, which is counting triangle-free graphs on n vertices. Many of the definitions throughout this thesis come from the lecture notes of the courses Graph Theory [3] and Extremal Combinatorics [4].

In the final chapter, we will go through an application. This application is worked out in [5] but is quite technical. Here the aim is to get an intuition of how the hypergraph container method can be applied to this problem.

[1] H-free graphs are graphs that do not contain H as a subgraph. See definition 2.3.
[2] https://www.python.org.

2

PRELIMINARIES

THE container method works with graphs. In this chapter we will give the necessary background information from the field of (extremal) graph theory. We will also see how we can use the bipartite graph $K_{\lfloor n/2 \rfloor, \lceil n/2 \rceil}$ to obtain a lower bound for the number of triangle-free graphs on n edges.

2.1. GRAPHS

LET us start by defining graphs.

Definition 2.1. A graph G is a pair of sets (V, E), where V is nonempty and E is a set of pairs of vertices, that is, $E \subseteq \{\{u, v\} : u, v \in V, u \neq v\}$. The set V is known as the set of vertices and the set E is known as the set of edges.

A graph is a nice way to visualise a structure. You could see this as vertices being related to each other if they are connected by an edge.

Remark 2.2. We call $V(G)$ the set of vertices of the graph G and $E(G)$ the set of edges of G. In this thesis, we will use E and $E(G)$ interchangeably, similar to V and $V(G)$.

Definition 2.3. Given a graph $G = (V, E)$, a subgraph of G is a graph $H = (V', E')$ such that $V' \subseteq V$ and $E' \subseteq E \cap \binom{V'}{2}$.

Remark 2.4. With $\binom{V}{2}$ we mean all combinations of 2 vertices from the set V. This correspond to the set of all possible edges that can occur in the graph $G = (V, E)$.

We see that a subgraph is a smaller graph obtained from the original graph. We need to make sure that for any edge we take from the original graph G, we also take the vertices that belong to the edge. See the red subgraph in figure 2.1.

Definition 2.5. A complete graph is a graph where $E = \binom{V}{2}$ and $|V| = n$. We denote this graph by K_n.

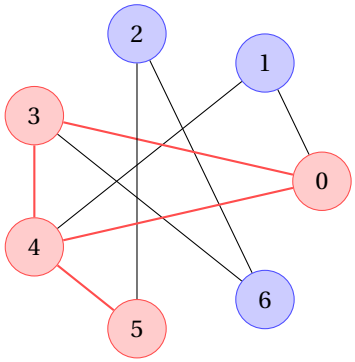


Figure 2.1: Example of a subgraph.

In a complete graph, every vertex is connected by every other vertex with an edge. In this way, we obtain a "completed graph" .

Example 2.6. For $n = 3$, we have the complete graph K_3. This is what we call a triangle. This is shown in figure 2.2.

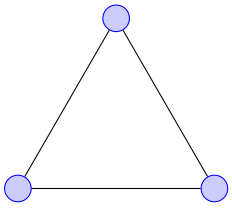


Figure 2.2: The complete graph K_3, also known as a triangle.

Definition 2.7. A triangle free graph is a graph that does not contain K_3 as a subgraph.

Example 2.8. An example of a triangle-free graph is the famous Petersen graph. This graph is illustrated in figure 2.3.

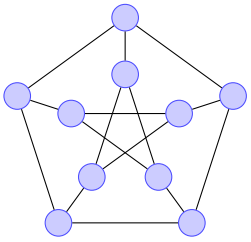


Figure 2.3: The Petersen graph.

Definition 2.9. Two vertices $u, v \in V$ in a graph $G = (V, E)$ are called adjacent if $\{u, v\} \in E$ and non-adjacent if $\{u, v\} \notin E$. A set of pairwise adjacent vertices in a graph is called a clique and a set of pairwise non-adjacent vertices is called an independent set (or a co-clique).

Remark 2.10. A singular vertex is both a clique and an independent set on its own.

Example 2.11. Let $G = (V, E)$ be the graph with $V = \{a, b, c\}$ and $E = \{\{a, b\}, \{b, c\}\}$ as in figure 2.4.

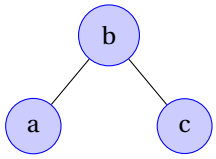


Figure 2.4: Cliques and independent sets in a graph.

Then the cliques of G are $\{a\}, \{b\}, \{c\}, \{a, b\}$ and $\{b, c\}$. The independent sets of G are $\{a\}, \{b\}, \{c\}$ and $\{a, c\}$.

Definition 2.12. Let $G = (V, E)$ a graph and $u \in V$. The set $N(u) = \{v \in V : \{u, v\} \in E\}$ is known as the neighbourhood of u. The size of this set is known as the degree of the vertex u, denoted by $\deg(u) = |N(u)|$.

Example 2.13. We see that degree of a vertex v is the number of edges that are connected to v. This is illustrated in figure 2.5. We see that vertex v has degree 4.

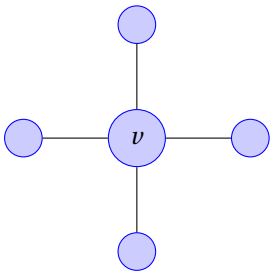


Figure 2.5: The degree of vertex v is 4.

Definition 2.14. A graph $G = (V, E)$ is called bipartite if the vertex set V can be partitioned into two independent subsets A, B. We denote the complete bipartite graph by $K_{m,n}$ where $|A| = m$ and $|B| = n$.

2.2. LOWER BOUND ON TRIANGLE-FREE GRAPHS

IMAGINE that we are trying to find the number of triangle-free graphs on n vertices. If we take $K_{\lfloor n/2 \rfloor, \lceil n/2 \rceil}$, where $\lfloor x \rfloor$ rounds x down to the nearest integer and $\lceil x \rceil$ rounds x

2

up to the nearest integer, we have a graph on n vertices. Note that this graph is triangle-free as a bipartite graph cannot contain a triangle [1]. This implies that all subgraphs of this graph are also triangle-free. This gives us a lower bound for the number of triangle-free graphs on n vertices, namely $2^{\lfloor n/2 \rfloor \lceil n/2 \rceil} = 2^{\lfloor n^2/4 \rfloor}$ subgraphs, as we have $\lfloor n/2 \rfloor \cdot \lceil n/2 \rceil$ edges. So we have at least $2^{\lfloor n^2/4 \rfloor}$ triangle-free graphs. This is illustrated in figure 2.6. The oldest result in extremal graph theory is the following theorem. This gives an upper bound on the number of edges in a triangle-free graph G on n edges. The theorem was proven by Mantel [6] in 1907.

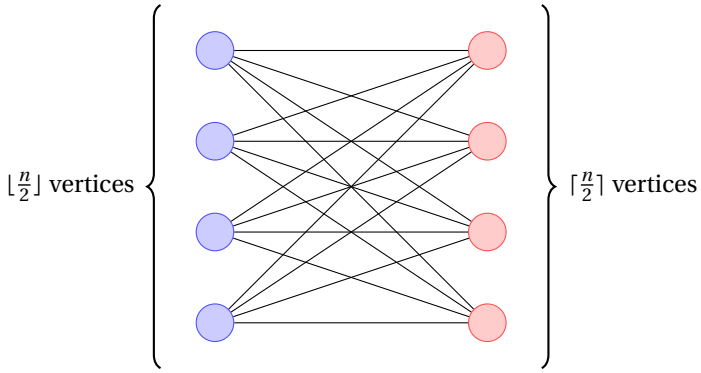


Figure 2.6: $K_{\lfloor n/2 \rfloor, \lceil n/2 \rceil}$

Theorem 2.15 (Mantel's Theorem)**.** A triangle-free graph G on n vertices has at most $\left\lfloor \frac{n^2}{4} \right\rfloor$ edges.

Remark 2.16. If we use $K_{\lfloor n/2 \rfloor, \lceil n/2 \rceil}$, we obtain equality in Mantel's theorem, as found in the lower bound found before using figure 2.6.

Definition 2.17. For events $A_1, \ldots, A_n$, we define the union bound as

$$\mathbb{P}\left(\bigcup_{i=1}^{n} A_i \right) \leq \sum_{i=1}^{n} \mathbb{P}(A_i).$$

We will now try to get an upper bound. One could think of using the union bound to determine the amount of triangle-free graphs on n vertices, but that would have too many events. Perhaps we can use Mantel's theorem to bound over all maximal triangle-free graphs? This will also be too wasteful. It turns out that the container method is a more efficient union bound that we can apply to get a better bound [2]. We will prove this in section 3.1.
Before we can define containers, we first need to explain what hypergraphs are.

Definition 2.18. A hypergraph H is a pair (V, E), where:

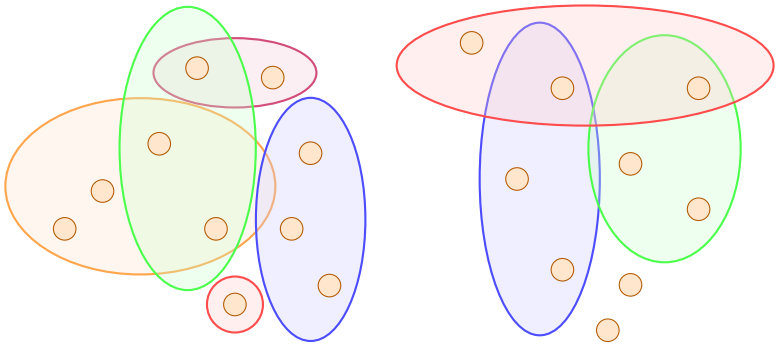
- V is a nonempty set of vertices,

[1] You cannot split a triangle into 2 independent sets.

- E is a set of nonempty subsets of V, called hyperedges.

In other words, each edge of a hypergraph is a set of vertices. We say that H is a r-uniform hypergraph if every hyperedge has exactly r vertices; that is, $E \subseteq \binom{V}{r}$.

Example 2.19. Let us look at two examples of hypergraphs. In figure 2.7a we can see a hypergraph H_1 and in figure 2.7b we see a 3-uniform hypergraph H_2.



(a) Hypergraph H_1 (b) 3-uniform hypergraph H_2

Figure 2.7: Example of two hypergraphs

An independent set in a hypergraph is defined similarly as its regular graph counterpart. Given a hypergraph H, an independent set of H is a set of vertices such that no hyperedge is fully contained within the independent set.

Definition 2.20. An independent set $I \subseteq V$ in a hypergraph $H = (V, E)$ is a set such that $\forall e \in E : e \nsubseteq I$.

We are now in a position to "define" containers. Given a hypergraph H with controlled degrees [2], we can find containers with the following properties:

- Each container is a subset of vertices of the hypergraph H.
- Every independent set of the hypergraph H is a subset of some container.
- The total amount of containers is relatively small.
- Each container is not too large. In fact, it is not much larger than the maximum size of an independent set.

[2] See conditions in theorem 5.2.

3

CONTAINERS FOR TRIANGLE-FREE GRAPHS

IN this chapter, we will introduce the concept of containers. The container method is a useful tool in combinatorics. We start with containers for triangle-free graphs to give an intuition for the container method, and later we will expand to hypergraphs.

3.1. APPLICATION TO TRIANGLE-FREE GRAPHS

AN example of an application of the container method is to count the number of triangle-free graphs on n vertices. We can model this as a 3-uniform hypergraph. This goes as follows:
We have the original graph G, where

- $|V(G)| = n$, and
- a triangle in G is a set of edges $\{e_1, e_2, e_3\} \subseteq E(G)$.

We now model this as the following hypergraph H:

- $V(H) = E(K_n)$, and
- $E(H) = \{\{e_1, e_2, e_3\} \subseteq E(K_n) : e_1, e_2, e_3 \text{ form a triangle}\}$.

Here, an independent set of $V(H)$ corresponds to a triangle-free graph in G. See example 3.1.

Example 3.1. Let us see a visual example of how a graph G can be turned into a 3-uniform hypergraph H that can be used to count the triangle-free graphs in G. In figure 3.1 you can see how a graph G on the left with five vertices is transformed to the corresponding hypergraph H on the right. Here the graph has two triangles: a red one and a blue one. These are coloured just for the visualisation. The dotted lines do not exist in this graph, but are for reference to understand how they transform to the hypergraph. They visualise the independent sets within G.

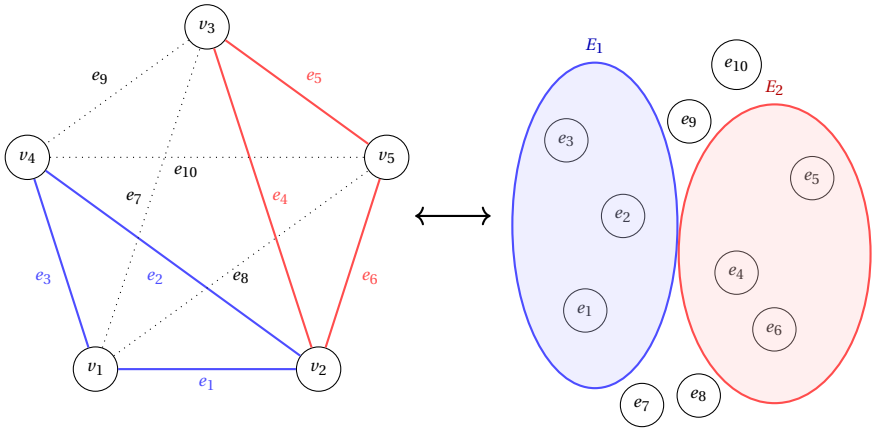


Figure 3.1: Example of a transformation from a regular graph to a hypergraph.

We will now formulate our first theorem using containers. This theorem is from [2].

Theorem 3.2 (Containers for Triangle-Free Graphs)**.** For every $\epsilon > 0$, there exists a $C > 0$, such that for every $n \in \mathbb{N}$, there is a collection $\mathcal{C}$ of graphs on n vertices, with

$$|\mathcal{C}| \leq n^{Cn^{3/2}}$$

such that

(a) every $G \in \mathcal{C}$ has at most $(\frac{1}{4} + \epsilon)n^2$ edges, and

(b) every triangle-free graph is contained in some $G \in \mathcal{C}$.

This theorem tells us that for every n, we can make a collection $\mathcal{C}$ of containers such that the number of containers is smaller than $n^{Cn^{3/2}}$ and every triangle-free graph is contained in one of these containers. Since every $G \in \mathcal{C}$ has at most $\left(\frac{1}{4} + \epsilon\right) n^2$ edges, the largest triangle free graph will have this amount of edges too.

Theorem 3.2 is an older theorem, and uses regular graphs instead of hypergraphs. You can, however, still view this as a result of using hypergraphs:

- The collection $\mathcal{C}$ of graphs on n vertices is equal to a collection of vertices from $V(H)$.
- Every graph $G \in \mathcal{C}$ has at most $\left(\frac{1}{4} + \epsilon\right) n^2$ edges, thus every subset of a container has at most $\left(\frac{1}{4} + \epsilon\right) n^2$ points.
- Every triangle-free graph is contained in some $G \in \mathcal{C}$. This corresponds to the fact that every independent set in $V(H)$ is in a container.

Using theorem 3.2, we can prove the following result.

Theorem 3.3 (Erdős, Kleitman, Rothschild, 1973)**.** The number of triangle-free graphs on n vertices is $2^{n^2/4 + o(n^2)}$.

Proof. Let $\epsilon > 0$ be arbitrarily small. Let $\mathcal{C}$ be as in theorem 3.2. Then by said theorem,

(a) every $G \in \mathcal{C}$ has at most $(\frac{1}{4}+\epsilon)n^2$ edges,

(b) and every triangle-free graph is contained in some $G \in \mathcal{C}$.

Then we have

$$\begin{aligned} |\mathcal{C}| \cdot 2^{(\frac{1}{4}+\epsilon)n^2} &\leq 2^{(\frac{1}{4}+\epsilon)n^2} \cdot n^{Cn^{3/2}} \\ &= 2^{(\frac{1}{4}+\epsilon)n^2} \cdot 2^{Cn^{3/2}\log n} \\ &= 2^{(\frac{1}{4}+\epsilon)n^2 \cdot Cn^{3/2}\log n}. \end{aligned}$$

And since $\epsilon > 0$ was arbitrarily small, we conclude that the number of triangle-free graphs on n vertices is $2^{(\frac{1}{4}+o(1))n^2} = 2^{n^2/4+o(n^2)}$. □

We will need the following two definitions to understand the Erdős-Stone-Simonovits theorem.

Definition 3.4. We write $\mathrm{ex}(n, H)$ for the maximum number of edges in a n-vertex graph G without H as a subgraph.

Definition 3.5. A k-colouring of a graph $G = (V, E)$ is a labelling $f : V \to \{1, 2, \ldots, k\}$. It is called a proper k-colouring if for all $\{x, y\} \in E$, $f(x) \neq f(y)$. A graph is called k-colourable if it has a proper k-colouring. The chromatic number $\chi(G)$ is the minimum k such that G is k-colourable.

The following theorem allows you to count not only triangle-free graphs, but any H-free graph. The Erdős-Stone-Simonovits theorem says that for a non-bipartite graph H,

$$\mathrm{ex}(n, H) = \left(1 - \frac{1}{\chi(H)-1} + o(1)\right)\binom{n}{2}, \tag{3.1}$$

and then the number of H-free graphs on n vertices is equal to $2^{(1+o(1))\,\mathrm{ex}(n,H)}$. One might wonder if this theorem gives a better bound on the number of triangle-free graphs than theorem 3.3.

$$\begin{aligned} \mathrm{ex}(n, K_3) &= \left(1 - \frac{1}{\chi(K_3)-1} + o(1)\right)\binom{n}{2} \\ &= \left(1 - \frac{1}{2} + o(1)\right)\binom{n}{2} \\ &= \left(\frac{1}{2} + o(1)\right)\binom{n}{2}. \end{aligned}$$

Using the second part of the theorem, we conclude that the number of triangle-free graphs on n vertices is equal to

$$2^{(1+o(1))\left(\frac{1}{2}+o(1)\right)\binom{n}{2}} = 2^{\left(\frac{1}{2}+o(1)\right)\frac{n(n-1)}{2}} = 2^{\frac{n(n-1)}{4}+o(n^2)}.$$

We see that theorem 3.3 gives a much better bound, showing how important the container method is.

3.2. MANTEL'S THEOREM IN RANDOM GRAPHS

LET us now see a first problem where the container method can be applied. Theorem 3.8 is not a direct application of the container method, as it was originally proven without the use of it. It, however, can still be proven using the container method, which might give a better understanding of how containers can be used.

We first have to define what the random graph $G(n,p)$ is.

Definition 3.6. We call $G(n,p)$ or $G_{n,p}$ the random graph on n vertices with probability p. This graph is obtained by independently taking each of the $\binom{n}{2}$ pairs of vertices as an edge with probability p.

In remark 3.9 we will make use of the expectation.

Definition 3.7. The expectation of a random variable X is defined by

$$\mathbb{E}[X] = \sum_{x \in X(\Omega)} x \cdot \mathbb{P}(X = x).$$

Theorem 3.8. If $p \gg 1/\sqrt{n}$, then with probability $1 - o(1)$, every triangle-free subgraph of $G(n,p)$ has at most $(\frac{1}{4} + o(1))pn^2$ edges.

Remark 3.9. The statement is false for $p \ll 1/\sqrt{n}$. Indeed, we have that the expected number of triangles [7] is

$$\begin{aligned}
&\mathbb{E}\big[\text{number of triangles in } G(n,p)\big] \\
&= \sum_{i<j<k} \mathbb{P}\big(\text{triangle } \{i,j,k\} \text{ is present in } G(n,p)\big) \\
&= (\text{number of triangles in } K_n) \times \mathbb{P}\big(\text{each triangle is present}\big) \\
&= \binom{n}{3} p^3 \\
&= O(n^3 p^3).
\end{aligned} \tag{3.2}$$

Note that the expected number of edges [7] is equal to

$$\begin{aligned}
&\mathbb{E}\big[\text{number of edges in } G(n,p)\big] \\
&= \sum_{i<j} \mathbb{P}\big(\text{edge } \{i,j\} \text{ is present in } G(n,p)\big) \\
&= (\text{number of edges in } K_n) \times \mathbb{P}\big(\text{each edge is present}\big) \\
&= \binom{n}{2} p.
\end{aligned} \tag{3.3}$$

Thus, we have w.h.p. $\frac{n^2 p}{2}$ edges. As $n^3 p^3 \ll n^2 p$, we can remove one edge of each of the $O(n^3 p^3)$ triangles to make the graph triangle-free, which is $o(n^2 p)$ edges.

Proof. We will prove a weaker result for $p \gg \frac{\log n}{\sqrt{n}}$. Let $\epsilon > 0$ be arbitrarily small. Let $\mathcal{C}$ be a set of containers for n-vertex triangle-free graphs satisfying theorem 3.2. For every $G \in \mathcal{C}$, $e(G) \leq \left(\frac{1}{4} + \epsilon\right) n^2$, so applying the Chernoff bound, theorem 5.0.7 from [2],

$$\mathbb{P}\left(e(G \cap G(n,p)) > \left(\frac{1}{4} + 2\epsilon\right) n^2 p\right) \leq e^{-\Omega_\epsilon(n^2 p)}.$$

Since every triangle-free graph is contained in some $G \in \mathcal{C}$, we take a union bound over $\mathcal{C}$ to obtain

$$\begin{aligned}
&\mathbb{P}\left(G(n,p) \text{ has a triangle-free subgraph with} > \left(\frac{1}{4} + 2\epsilon\right) n^2 p \text{ edges}\right) \\
&\leq \sum_{G \in \mathcal{C}} \mathbb{P}\left(e(G \cap G(n,p)) > \left(\frac{1}{4} + 2\epsilon\right) n^2 p\right) \\
&\leq |\mathcal{C}| e^{-\Omega_\epsilon(n^2 p)} \\
&\leq e^{O_\epsilon(n^{3/2} \log n - \Omega_\epsilon(n^2 p))} \\
&= o(1),
\end{aligned}$$

provided that $p \gg \frac{\log n}{\sqrt{n}}$. □

4

GRAPH CONTAINERS

IN this section we will look at the container method for independent sets in graphs. We will take a look at the following theorem from [2].

Theorem 4.1 (Container Theorem for Independent Sets in Graphs)**.** For every $c > 0$, there exists a $\delta > 0$ such that the following holds.
Let $G = (V, E)$ be a graph with average degree d and maximum degree at most cd. There exists a collection $\mathcal{C}$ of subsets of V with

$$|\mathcal{C}| \leq \binom{|V|}{\leq \frac{2\delta|V|}{d}} \tag{4.1}$$

such that

- every independent set I of G is contained in some $C \in \mathcal{C}$,
- $|C| \leq (1-\delta)|V|$ for every $C \in \mathcal{C}$.

Definition 4.2. A maximal independent set is an independent set that cannot be extended with another vertex of the graph. In other words, it is not contained in any larger independent set.

This theorem tells us that if we have a graph with a controlled degree (a constant factor times the average degree), there is a collection of containers $\mathcal{C}$. Equation 4.1 in the theorem tells us something about the number of containers in the collection $\mathcal{C}$. It shows there are not too many containers. Here, $\binom{n}{\leq i} = \binom{n}{0} + \binom{n}{1} + \cdots + \binom{n}{i}$. The most valuable result are the two points below. As every independent set I of G is contained in a container C, the maximal independent set is also. The second point gives an upper bound on the size of a container. Essentially it tells us we have removed $\delta|V|$ vertices from the graph. As the maximal independent set is contained in this container, it also gives an upper bound on the size of this set. The theorem is proved using an algorithm from [2] that outputs the container and also a fingerprint S, related to this container.

Algorithm 4.3 (The Graph Container Algorithm)**.** We input a maximal independent set $I \subseteq V$. The output is a "fingerprint" $S \subseteq I$ of size smaller than $\frac{2\delta|V|}{d}$, and a container $C \supseteq I$ which only depends on S. In the algorithm we will maintain a partition $V = A \cup S \cup X$, where

- A is the set with available vertices, initially $A = V$.
- S is the current fingerprint, initially $S = \emptyset$.
- X is the set with the excluded vertices, initially $X = \emptyset$.

We will call the *max-degree order* of $G[A]$ the ordering of A by the degree of its vertices in $G[A]$. We order it by putting the vertex with the largest degree first, and break ties according to some arbitrary predetermined ordering of V. In section 4.1, the algorithm breaks ties by ordering by vertex number in a descending order. Now we state the algorithm.

While $|X| < \delta|V|$:

1. Let v be the first vertex of $I \cap A$ in the max-degree order on $G[A]$.
2. Add v to S.
3. Add the neighbours of v to X.
4. Add vertices preceding v in the max-degree order on $G[A]$ to X.
5. Remove from A all the new vertices added to $S \cup X$.

With this algorithm and the fingerprint obtained by it, we get a stronger theorem [2] that is relevant for some applications.

Theorem 4.4 (Graph Container Theorem with Fingerprints)**.** For every $c > 0$, there exists a $\delta > 0$ such that the following holds.
Let $G = (V, E)$ be a graph with average degree d and maximum degree at most cd. Writing $\mathcal{I}$ for the collection of independent sets of G, there exist functions

$$S : \mathcal{I} \to 2^V \text{ and } A : 2^V \to 2^V$$

(one only needs to define $A(\cdot)$ on sets in the image of S) such that, for every $I \in \mathcal{I}$,

- $S(I) \subseteq I \subseteq S(I) \cup A(S(I))$,
- $|S(I)| \leq \frac{2\delta|V|}{d}$,
- $|S(I) \cup A(S(I))| \leq (1-\delta)|V|$.

We see that the fingerprint is a subset of the independent set and hence the container that belongs to the independent set. Somehow, this fingerprint tells us something about the container it is in. We also see the bounds from theorem 4.1 back in the last 2 bullet points.

4.1. IMPLEMENTING THE ALGORITHM

To better understand this algorithm, the choice was made to implement it into an existing coding language. This was done by writing the necessary code for Python, linked in appendix A.1.
The following observations were made:

- The union of the output sets S and A is the container produced by the algorithm, and $I \subseteq S \cup A$.
- We see that if the output set A is empty, the maximal independent set I is a container itself.
- Each container is the union of all the maximal independent sets it contains.
- All maximal independent sets with the same fingerprint belong to the same container.
- We see that all vertices of S together with their neighbouring vertices cover the whole graph except for the vertices in A.
- The input maximal independent set I should always be sorted on reversed order to keep consistency throughout the algorithm.
- If there exists a $v \in V(G)$ with $\deg(v) = n-1$, then v itself is a container.

To give an illustration of the algorithm, we will look at an example of running the algorithm on $G_{25,0.5}$.

Example 4.5 (Running the Graph Container Algorithm on $G_{25,0.5}$). We run the algorithm on the graph from figure 4.1. The input was the maximal independent set $I = \{24, 15, 8, 6\}$, and the output was the fingerprint $S = \{24, 8\}$ and the container $S \cup A = \{20, 5, 6, 24, 8, 15\}$. It is clear that $I \subseteq S \cup A$. The code that generated this example can be found in appendix A.3.

Let us go through the algorithm step by step now. We run the algorithm on the graph of figure 4.1. As input, it has the graph G itself and maximal independent set $I = \{24, 15, 8, 6\}$. We use a command from the [8] plugin to find this maximal independent set.

First, we initialise by making sets and calculating the average degree we will need later in the algorithm.

- Create A, S, V.
- Calculate the average degree d.
- Calculate δ [1].
- Create the max-degree order as $G[A]$.

[1] Here we used that $\delta < 1/4c(c+1)$, obtained from [9]. In the algorithm we implemented this as $\delta = 1/4c(c+1) \cdot 0.999$ with $c = m/d$ with m the maximum degree

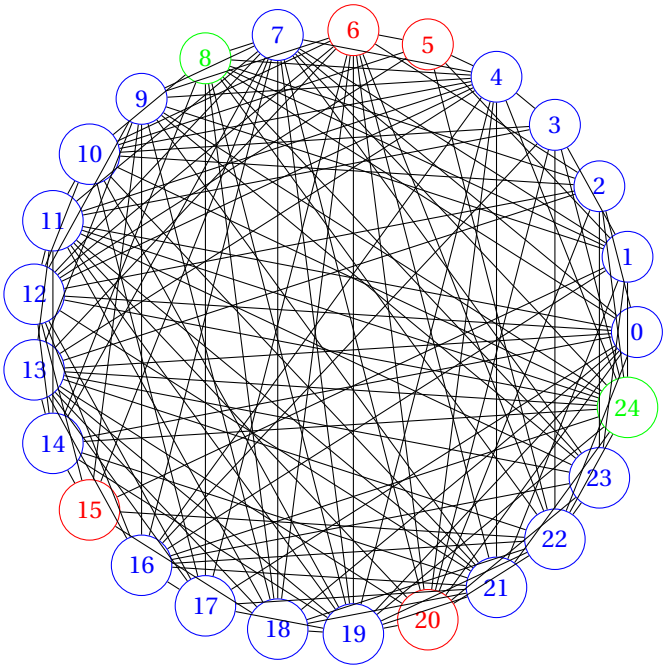


Figure 4.1: Algorithm applied to $G_{25,0.5}$ with maximal independent set $\{24,15,8,6\}$.

We are going to run the algorithm on $G(25,0.5)$, with independent set $I = \{24,15,8,6\}$. The max-degree order on $G[A]$ is

$$G[A] : \{24,19,13,12,7,11,4,22,21,16,10,9,8,6,0,20,18,23,14,1,17,2,3,15,5\}.$$

We have the following constants:

- Average degree $d = 11.68$,
- $\delta = 0.732650576444924$.

We now have the following partition which we will add vertices from and to throughout the algorithm.

- $A = \{24,19,13,12,7,11,4,22,21,16,10,9,8,6,0,20,18,23,14,1,17,2,3,15,5\}$,
- $S = \{\}$,
- $X = \{\}$.

We now start the first iteration of our algorithm. We know that $|X| = 0$ and $\delta|V| \approx 18{,}316$, so we start the loop. We start by creating the intersection on $G[A]$, which is

$$I \cap A = \{24,8,6,15\}. \tag{4.2}$$

The steps are executed as follows:

1. The first vertex with highest degree in $I \cap A$ is $v = 24$.
2. We add this v to S.
3. Now we add all neighbours of v to X, which are

$$N(v) = \{1,2,3,4,7,9,10,11,12,13,14,16,19,21,22\}. \tag{4.3}$$

4. The vertices preceding v from $G[A]$ are now added to X. There are none left to add as they have already been added in the previous step.
5. From A, we remove all these new vertices from $S \cup X$. Then, our sets look as follows:
   - $A = \{8,6,0,20,18,23,17,15,5\}$,
   - $S = \{24\}$,
   - $X = \{1,2,3,4,7,9,10,11,12,13,14,16,19,21,22\}$.

This concludes the first iteration of the algorithm.
We continue with the second iteration of our algorithm. We now have $|X| = 15$ and $\delta|V| \approx 18.316$, so we continue the loop. We again calculate the intersection, which is equal to

$$I \cap A = \{8,6,15\}.$$

We now go through the same steps again:

1. The first vertex in $I \cup A$ is $v = 8$.
2. We add v to S.
3. We add the neighbours of v to X, which are

$$\{0,1,2,4,7,10,14,17,18,19,22,23\}. \tag{4.4}$$

4. We add the vertices preceding v from $G[A]$ to X. There are again none to add.
5. From A, we remove all these new vertices from $S \cup X$. Our sets now look as follows:
   - A : $\{6,20,15,5\}$,
   - S : $\{24,8\}$,
   - X : $\{0,1,2,3,4,7,9,10,11,12,13,14,16,17,18,19,21,22,23\}$.

We then move on to the third iteration. We see that $|X| = 20$ and $\delta|V| \approx 18.316$, so $|X| \not< \delta|V|$. We conclude that we terminate the algorithm. The algorithm now outputs

- the fingerprint $S = \{24,8\}$,
- the container $S \cup A = \{20,5,6,24,8,15\}$,
- and $I = \{24,15,8,6\}$ to check if it is contained in the container.

This example used one of the many maximal independent sets I. To better understand that the fingerprints are unique, in table 4.1 we see every maximal independent set I and the output of the algorithm. We used [10] to find all of these maximal independent sets. We see here that independent sets with the same fingerprint belong to the same container. We see that this algorithm "proves" the theorem as it meets the output conditions. We have $\delta \approx 0.733$ and $d = 11.68$.

- The amount of unique containers in table 4.1 is $|\mathcal{C}| = 66$ and this is clearly less than

$$\binom{|V|}{\leq \frac{2\delta|V|}{d}} \approx \binom{25}{\leq 3.138} = \binom{25}{0} + \binom{25}{1} + \binom{25}{2} + \binom{25}{3} = 2626.$$

- Every independent set is contained in some $C \in \mathcal{C}$ as every independent set is a subset of some maximal independent set I which is contained in a container.
- The last statement in the theorem says that each container C is smaller than $(1-\delta)|V| \approx 6.675$. This is also satisfied since the largest container has 6 points.

Table 4.1: All maximal independent sets and their outputs.

Maximal independent set I	Fingerprint S	Container $S \cup A$
{6,4,1,0}	{0,4,6}	{0,1,4,6,23,15}
{7,6,3}	{6,7}	{3,6,7}
{9,5,3,0}	{0,9}	{0,9,3,5}
{9,8,5,3}	{8,9}	{3,20,5,8,9}
{10,1,0}	{0,10}	{0,1,23,10,15}
{10,9,0}	{9,10}	{0,20,9,10,14}
{11,6,1}	{11,6}	{1,11,6,15}
{11,9,8,5}	{9,11}	{17,2,5,8,9,11}
{13,10,1}	{10,13}	{1,10,13,15}
{13,10,7}	{13,7}	{3,21,5,7,10,13}
{13,12,1}	{12,13}	{1,3,8,12,13}
{13,12,8,3}	{12,13}	{1,3,8,12,13}
{14,6,4,1}	{4,6,14}	{1,4,6,14}
{15,8,6,3}	{8}	{3,20,5,6,8,15}
{15,13,10}	{10,13}	{1,10,13,15}
{16,14,6}	{16,6}	{16,0,6,14,15}
{16,15,5,0}	{16,0}	{16,0,5,15}
{16,15,6,0}	{16,6}	{16,0,6,14,15}
{16,15,10,0}	{16,0,10}	{16,0,10,15}
{16,15,11,5,2}	{16,2,11}	{16,2,5,11,15}
{16,15,11,8,5}	{16,8,11}	{16,5,6,8,11,15}
{16,15,11,8,6}	{16,8,11}	{16,5,6,8,11,15}
{17,7,5,3,2}	{17,7}	{17,2,3,5,7}
{17,11,9,5,2}	{9,11}	{17,2,5,8,9,11}
{17,14,9,5,3,2}	{9,14}	{17,2,3,5,9,14}
{17,15,4,2}	{17,4}	{17,2,4,15}
{17,15,5,3,2}	{17}	{17,2,3,5,15}
{17,15,11,5,2}	{17,11}	{17,2,5,11,15}
{18,16,14,5,2}	{16,18}	{16,18,2,5,14}
{18,17,14,4,2,1}	{18,4,14}	{1,18,17,4,2,14}
{18,17,14,5,3,2}	{17,18,14}	{17,18,2,3,5,14}
{19,2,1}	{1,19}	{1,2,19}
{19,9,5,3,2}	{9,19}	{2,19,3,5,9}
{19,12,1}	{19,12}	{1,19,3,23,12}
{19,15,5,3,2}	{2,19}	{2,19,3,5,15}
{19,16,15,5,2}	{16,19}	{16,2,19,5,15}
{20,9,8,5}	{8,9}	{3,20,5,8,9}
{20,12,8}	{8,12}	{8,3,12,20}
{20,14,10,9}	{9,10}	{0,20,9,10,14}
{20,16,15,8,5}	{16,8}	{16,20,5,6,8,15}
{20,16,15,10}	{16,10,20}	{16,18,20,10,14,15}
{20,17,14,9,5}	{9,20}	{17,20,5,9,14}

{20, 18, 12, 1}	{12, 20}	{1, 18, 12, 20}
{20, 18, 14, 10, 1}	{10, 20}	{1, 18, 20, 10, 14, 15}
{20, 18, 16, 14, 5}	{16, 20}	{16, 18, 20, 5, 14, 15}
{20, 18, 16, 14, 10}	{16, 10, 20}	{16, 18, 20, 10, 14, 15}
{20, 18, 17, 14, 1}	{18, 20}	{1, 18, 17, 20, 5, 14}
{20, 18, 17, 14, 5}	{18, 20}	{1, 18, 17, 20, 5, 14}
{21, 7, 5, 3, 2}	{21, 7}	{2, 3, 21, 5, 7}
{21, 13, 7, 5, 3}	{13, 7}	{3, 21, 5, 7, 10, 13}
{21, 15, 5, 3, 2}	{2, 21}	{2, 3, 21, 5, 15}
{21, 15, 13, 8, 5, 3}	{21, 13}	{3, 21, 5, 8, 13, 15}
{21, 20, 15, 8, 5}	{8, 21}	{3, 20, 21, 5, 8, 15}
{22, 10, 7}	{22, 7}	{17, 2, 21, 22, 7, 10}
{22, 12, 1}	{12, 22}	{1, 12, 22}
{22, 14, 10, 1}	{10, 22}	{1, 22, 9, 10, 14}
{22, 14, 10, 9}	{10, 22}	{1, 22, 9, 10, 14}
{22, 17, 7, 2}	{22, 7}	{17, 2, 21, 22, 7, 10}
{22, 17, 11, 2, 1}	{11, 22}	{1, 17, 2, 22, 9, 11}
{22, 17, 11, 9, 2}	{11, 22}	{1, 17, 2, 22, 9, 11}
{22, 17, 14, 4, 2, 1}	{4, 22}	{1, 17, 2, 4, 22, 14}
{22, 17, 14, 9, 2}	{9, 22}	{17, 2, 22, 9, 14}
{22, 21, 7, 2}	{22, 7}	{17, 2, 21, 22, 7, 10}
{23, 14, 6, 3}	{6, 23}	{3, 6, 23, 14, 15}
{23, 14, 6, 4}	{4, 6, 23}	{4, 6, 23, 14, 15}
{23, 15, 6, 3, 0}	{0, 6}	{0, 1, 3, 6, 23, 15}
{23, 15, 6, 4, 0}	{0, 4, 6}	{0, 1, 4, 6, 23, 15}
{23, 15, 10, 0}	{0, 10}	{0, 1, 23, 10, 15}
{23, 17, 15, 4}	{4, 23}	{17, 4, 23, 14, 15}
{23, 17, 15, 5, 3}	{23}	{17, 3, 5, 23, 14, 15}
{23, 18, 12, 3}	{18, 12}	{1, 18, 3, 23, 12}
{23, 18, 14, 10}	{10, 18}	{1, 18, 23, 10, 14}
{23, 18, 17, 14, 4}	{18, 4, 23}	{17, 18, 4, 23, 14}
{23, 18, 17, 14, 5, 3}	{18, 23}	{17, 18, 3, 5, 23, 14}
{23, 19, 12, 3}	{19, 12}	{1, 19, 3, 23, 12}
{23, 19, 15, 5, 3}	{19, 23}	{19, 3, 5, 23, 15}
{23, 21, 15, 5, 3, 0}	{0, 21}	{0, 3, 21, 5, 23, 15}
{24, 15, 8, 6}	{24, 8}	{20, 5, 6, 24, 8, 15}
{24, 20, 15, 8, 5}	{24, 8}	{20, 5, 6, 24, 8, 15}
{24, 20, 17, 15, 5}	{24, 20}	{17, 18, 20, 5, 24, 15}
{24, 20, 18, 17, 5}	{24, 20}	{17, 18, 20, 5, 24, 15}
{24, 23, 15, 5, 0}	{24, 0}	{0, 5, 23, 24, 15}
{24, 23, 15, 6, 0}	{24, 6}	{0, 6, 23, 24, 15}
{24, 23, 17, 15, 5}	{24, 23}	{17, 5, 23, 24, 15}
{24, 23, 18, 17, 5}	{24, 18}	{17, 18, 5, 23, 24}

5

HYPERGRAPH CONTAINERS

We have seen theorem 4.1 for graphs, but now now want to extend to hypergraphs. In particular, we will be looking at 3-uniform hypergraphs.

Remark 5.1. Given an r-uniform hypergraph H and $1 \le l < r$, we write

$$\Delta_l(H) = \max_{A \subseteq V(H):|A|=l} \big(\text{the number of edges containing A}\big).$$

With this we can formulate the container theorem [2] for 3-uniform hypergraphs.

Theorem 5.2 (Container Theorem for 3-uniform Hypergraphs)**.** For every $c > 0$ there exists a $\delta > 0$ such that the following holds.
Let H be a 3-uniform hypergraph with average degree $d \ge \delta^{-1}$ and

$$\Delta_1(H) \le cd \quad \text{and} \quad \Delta_2(H) \le c\sqrt{d}.$$

Then there exists a collection $\mathcal{C}$ of subsets of $V(H)$ with

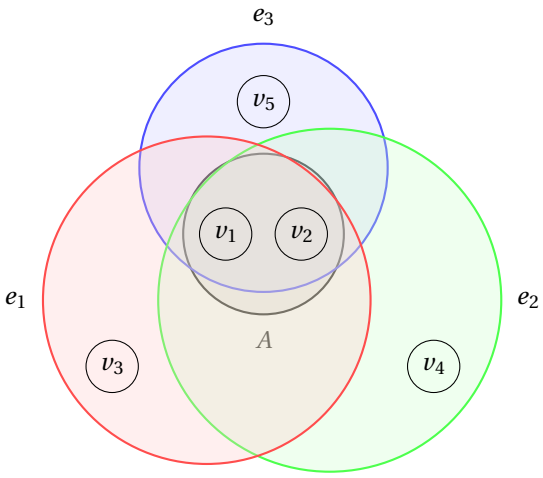
$$|\mathcal{C}| \le \binom{v(H)}{\le v(H)/\sqrt{d}}$$

such that

- every independent set of H is contained in some $C \in \mathcal{C}$,
- $|C| \le (1-\delta)v(H)$ for every $C \in \mathcal{C}$.

Let us try to understand how the theorem works. The input of the theorem is a 3-uniform hypergraph H with average degree d. There are 2 conditions for the theorem:

- The first condition, $\Delta_1(H) \le cd$, means that Δ_1, which is the maximum number of edges containing a 1-vertex subset of $v(H) = |V(H)|$, equal to the maximum degree of any vertex. So we require the maximum degree to be smaller than a constant multiplied by the average degree.



5

Figure 5.1: Example of a hypergraph H with $\Delta_2 = 3$

- The second condition, $\Delta_2 \leq c\sqrt{d}$, requires Δ_2, which is the maximum number of edges containing a two-vertex subset of $v(H)$ to be smaller than a constant multiplied with the square root of the average degree. This Δ_2 is equal to the largest "shared" degree of two vertices. This is illustrated in figure 5.1. Here A is the set with two vertices as $l = 2$, and the hyperedges e_1, e_2, e_3 all have one vertex outside of A, and A is contained within all three of the hyperedges. We repeat this for the other options of two vertex sets as Δ_2 is the max of these, but these options will have either zero or one hyperedge containing A. We see that here $\Delta_2 = 3$ with the A as in figure 5.1, as there are three outgoing edges from A, which is the maximum.

The output of the theorem gives us a collection $\mathcal{C}$ of subsets of $v(H)$, which are the containers. Every independent set will be in one of such containers. And then

$$|\mathcal{C}| \leq \binom{v(H)}{\leq v(H)/\sqrt{d}}$$

gives us a bound of the number of containers. $|C| \leq (1-\delta)v(H)$ gives us the same bound on the size of the containers again, hence also a bound on the maximal independent set. This is a valuable result that we can use to prove problems.

We will again prove this with an algorithm from [2] that has an independent set $I \subseteq v(H)$ as input and gives a fingerprint $S \subseteq I$ and a container $C \supset I$.

Algorithm 5.3 (Container Algorithm for 3-uniform Hypergraphs)**.** Throughout the algorithm, we will maintain

- a fingerprint S, initially $S = \emptyset$.

- a 3-uniform hypergraph A, intitally $A = H$,
- a graph G of "forbidden" pairs on $V(H)$, initially $G = \emptyset$.

The algorithm:
While $|S| \leq \frac{v(H)}{\sqrt{d}} - 1$:

- Let u be the first vertex in I in the max-degree order on A.
- Add u to S.
- Add $\{x, y\}$ to $E(G)$ whenever $\{u, x, y\} \in E(H)$.
- Remove from $V(A)$ the vertex u as well as all vertices proceeding u in the max-degree order on A.
- Remove from $V(A)$ every vertex whose degree in G is larger than $c\sqrt{d}$.
- Remove from $E(A)$ every edge that contains an edge of G.

We will see that after the algorithm terminates we either

- have removed many vertices from $V(A)$,
- or have a final graph G with at least $\Omega(\sqrt{d}n)$ edges and maximum degree $O(\sqrt{d})$, so that we can apply the graph container lemma (algorithm 4.3 from theorem 4.1) to G.

The most important part to note here is that this algorithm reduces the hypergraph to a regular graph to which the graph container algorithm can be applied. This algorithm has also been implemented in Python, but visualisation is hard for hypergraphs. For now, this is left as future work. See chapter 7.

6

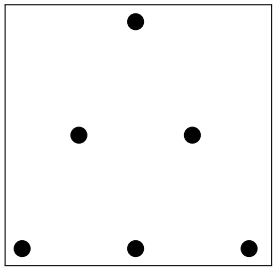
Applying Containers to a Problem in Discrete Geometry

We will now try to apply the container method to a problem. Erdős posed the following problem in [11]:
Given n points in the Euclidean plane $\mathbb{R}^2$, with at most three on any line, how large a subset are we guaranteed to find in general position (i.e., with at most two on any line)?

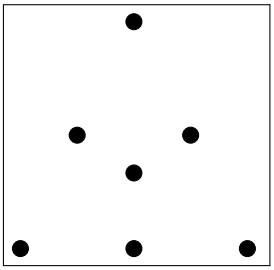
We will start with an intuitive approach for a problem that approaches the threshold of the Erdős problem from the other direction. Instead of trying to find the largest subset with at most two on a line we try to find the smallest set such that every subset of this size contains three points on a line as in theorem 6.1, ignoring the set size $n^{\frac{5}{6}+o(1)}$.
Note that for $n \leq 5$ there are no cases in which a subset smaller than n itself guarantees three points on a line. For $n = 6$ to $n = 13$ structures are found. These are drawn in figure 6.1. The structures indicate that every subset of size $s = \lceil n^{5/6} \rceil$ contains three points on a line. These structures are not unique, as illustrated in figure 6.1d and figure 6.1e.

Füredi proved [12] that you can always find a subset of size $\Omega\left(\sqrt{n \log n}\right)$. He also gave a construction in which the largest set has size $o(n)$. We can obtain a stronger upper bound using the method of hypergraph containers.

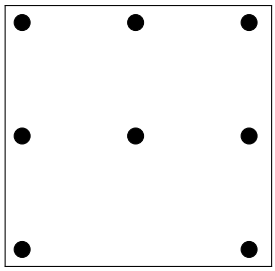
Theorem 6.1. There exists a set $S \subset \mathbb{R}^2$ of size n, containing no four points on a line, such that every subset of S of size $n^{\frac{5}{6}+o(1)}$ contains three points on a line.



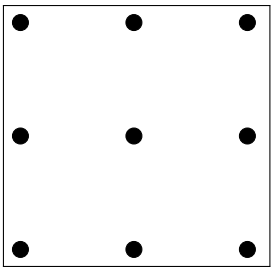
(a) Case $n = 6, s = 5$



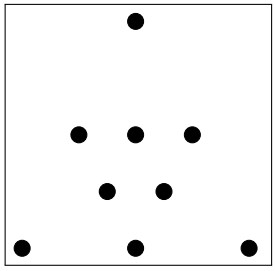
(b) Case $n = 7, s = 5$



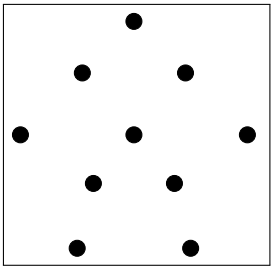
(c) Case $n = 8, s = 6$



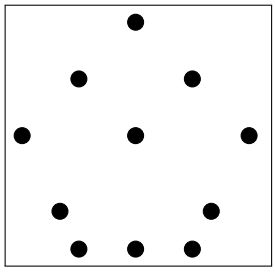
(d) Case $n = 9, s = 7$



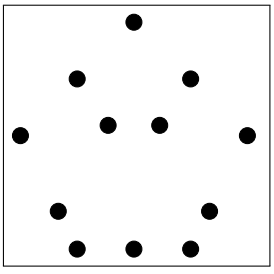
(e) Case $n = 9, s = 7$



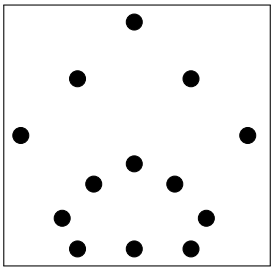
(f) Case $n = 10, s = 7$



(g) Case $n = 11, s = 8$



(h) Case $n = 12, s = 8$



(i) Case $n = 13, s = 9$

Figure 6.1: Cases $n \in \{6, 7, 8, 9, 10, 11, 12, 13\}$

6.1. APPLYING THE CONTAINER METHOD

To prove theorem 6.1 we will be modelling the problem as hypergraphs. Indeed, we see that we can make a 3-uniform hypergraph $\mathcal{H}$. We say that the edge $\{x, y, z\}$ exists if and only if x, y, z are 3 points that lie on a line. If we have an independent set in $\mathcal{H}$, it means that we have no occurrence of 3 points on a straight line within that set of points. We are now in a position to apply the hypergraph container method.

The idea is that the container method gives us a collection of containers such that we know both the size and the number of containers. The most important part here is the size of the containers. The method gives us an upper bound on the size of a container, and as every independent set is contained in a container, also the maximal independent set is. Here we note that the maximal independent set in the hypergraph $\mathcal{H}$ corresponds to the maximal subset with no 3 points on a line in the discrete geometry problem. This means that any set larger than this will have 3 points on a line, which is exactly what we are looking for.

7

CONCLUSION AND DISCUSSION

In this thesis we learnt about the (hypergraph) container method. We learnt how it works and also how it can be applied to problems. We began by exploring triangle-free graphs and the application of the container method to count the number of triangle-free graphs on n vertices. After this we looked at the graph container theorem for independent sets in graphs. This algorithms proof consists of an algorithm. We looked into this algorithm and implemented it on the Erdős–Rényi random model $G(n,p)$. In chapter 5 we extended this theorem to hypergraphs, specifically 3-uniform hypergraphs. With this knowledge we looked at an application in discrete geometry. We started with an intuitive approach and later tried to understand how the container theorem for 3-uniform hypergraphs can be applied to this problem. There is still much more to work on. This thesis focusses a lot on the intuition behind applying the (hypergraph) container method, but avoids a lot of technical details. For example, the proof of the problem in chapter 6. There are also a lot more applications of the method, in various mathematical areas like extremal graph theory, additive combinatorics, discrete geometry, coding theory, and Ramsey theory. Lastly, the container theorem for 3-uniform hypergraphs also has a proof by algorithm. This algorithm has been implemented in Python, but in this thesis we did not look into it much. This is because it is very hard to visualise hypergraphs.

BIBLIOGRAPHY

[1] Wikipedia. (2025). Container method — Wikipedia, the free encyclopedia [Online].

[2] Zhao, Y. (2024). *Probabilistic methods in combinatorics* (Lecture Notes, Course 18.226 (Fall 2022)) (Last updated June 18, 2024). Massachusetts Institute of Technology. https://yufeizhao.com/pm/probmethod_notes.pdf

[3] Bishnoi, A. (2025). *Graph theory* (Lecture Notes, Course Graph Theory) (Last updated on March 24, 2025). Delft University of Technology.

[4] Anurag Bishnoi, D. G., Wouter Cames van Batenburg. (2025). *Extremal combinatorics* (Lecture Notes, Course Extremal Combinatorics). Delft University of Technology.

[5] Balogh, J., Morris, R., & Samotij, W. (2018). The method of hypergraph containers. https://arxiv.org/abs/1801.04584

[6] Mantel, W. (1907). Vraagstuk xxviii. *Wiskundige Opgaven met de Oplossingen*, *10*(2), 60–1.

[7] Park, J. (2023). Threshold phenomena for random discrete structures. *Notices of the American Mathematical Society*, *70*(10), 1615–1624. https://www.ams.org/notices/202310/rnoti-p1615.pdf

[8] Hagberg, A. A., Schult, D. A., & Swart, P. J. (2008). Exploring network structure, dynamics, and function using networkx. In G. Varoquaux, T. Vaught, & J. Millman (Eds.), *Proceedings of the 7th python in science conference* (pp. 11–15).

[9] Morris, R. (2016). *The method of hypergraph containers* (Lecture Notes) (Available at https://www.ime.usp.br/~spschool2016/wp-content/uploads/2016/07/Morris.pdf). São Paulo School in Combinatorics, IME–USP.

[10] Stein, W., et al. (2025). *Sage Mathematics Software (Version x.y.z)* [`http://www.sagemath.org`]. The Sage Development Team.

[11] Erdös, P. (1984). Some old and new problems in combinatorial geometry. *Northholland Mathematics Studies*, *87*, 129–136. https://api.semanticscholar.org/CorpusID:6725213

[12] Füred, Z. (1991). Maximal independent subsets in steiner systems and in planar sets. *SIAM Journal on Discrete Mathematics*, *4*(2), 196–199. https://doi.org/10.1137/0404019

A

CODE

IN this appendix one can find the code of the algorithm that was implemented in section 4. We use the `NetworkX` library [8] to visualise the output.

A.1. THE GRAPH CONTAINER ALGORITHM

THIS is the code written using a custom class `random_graph` and `random_3uniform_hypergraph`. There already exists an Erdős-Rényi model within the `NetworkX` library, but that does not work for hypergraphs. The regular graph container algorithm is implemented using this model in A.2.

```
# -*- coding: utf-8 -*-
"""
Created on Tue May  6 14:08:58 2025

@author: joachim
"""
import numpy as np
import random as rd
from itertools import combinations
import networkx as nx
import matplotlib.pyplot as plt


class random_graph(object):
    V: list
    E: list
    A: dict

    def __init__(self,n: int,p: float=1) -> None:
        self.n = n
```

A

```
        self.V = list()
        self.E = list()
        self.A = dict()
        for i in range(n):
            self.V.append(i)
            self.A[i] = list()
        for i,j in combinations(self.V,2):
            p2 = rd.random()
            if p2 < p:

                self.E.append((i,j))
                self.A[i].append(j)
                self.A[j].append(i)
    def add_edge(self,edge):
        self.E.append(edge)
        self.A[edge[0]].append(edge[1])
        self.A[edge[1]].append(edge[0])

class random_3uniform_hypergraph(object):
    V: list
    E: list
    A: dict

    def __init__(self,n: int,p: float=1) -> None:
        self.n = n
        self.V = list()
        self.E = list()
        self.A = dict()
        for i in range(n):
            self.V.append(i)
            self.A[i] = set()
        for i,j,k in combinations(self.V,3):
            p2 = rd.random()
            if p2 < p:

                self.E.append((i,j,k))
                self.A[i].add(j)
                self.A[i].add(k)
                self.A[j].add(i)
                self.A[j].add(k)
                self.A[k].add(i)
                self.A[k].add(j)
    def remove_edge(self,edge):
        self.E.remove(edge)

```

```
# def sort_degree(G):
    # return {k: v for k, v in sorted(G.A.items(),key=lambda item:
     ↪ len(item[1]),reverse=True)}

def sort_degree(G):
    degrees = {k: len(G.A[k]) for k in sorted(G.A,reverse=True)}
    return {k:v for k, v in sorted(degrees.items(),key=lambda item:
     ↪ item[1],reverse=True)}


def avg_degree(G):
    return sum(len(k) for k in G.A.values())/len(G.A.values())

def find_indepset(H):
    G = nx.empty_graph(H.n)
    for x,y,z in H.E:
        G.add_edges_from(combinations([x,y,z],2))
    return sorted(nx.maximal_independent_set(G),reverse=True)

def draw_indepset(S,I,G):
    G2 = nx.empty_graph(G.n)
    G2.add_edges_from(G.E)
    pos = nx.circular_layout(G2)
    nx.draw_networkx(G2,pos=pos)
    nx.draw_networkx_nodes(G2,pos=pos,nodelist=I,node_color='red',
     ↪ label='I')
    nx.draw_networkx_nodes(G2,pos=pos,nodelist=S,node_color='green',
     ↪ label='S')
    plt.legend()



def graph_container_algorithm(I,G,draw=True):
    # Sort A on max-degree
    A = sort_degree(G)
    print("Running algorithm on $G(",n,",",p,")$, with independent
     ↪ set $I=",I,"$")
    # print("$G[A]:", A,"$")
    # Initialise
    d = avg_degree(G)
    # print("Average degree:",d)
    c = list(A.values())[0]/d
    delta = 1/4*c*(c+1)*0.999
    # print("$\delta=",delta,"$")
    S = set()
```

A

```
106         X = set()
107         # The algorithm itself
108         iteration = 1
109         while len(X) < delta*len(G.V):
110             # print("Iteration",iteration)
111             # print("$|X|=",len(X),",\delta|V|=",delta*len(G.V),"$")
112             # Step 1: intersection of I and A in max-degree order on
                ↪ G[A]
113             union = {k:A[k] for k in [x for x in I if x in A.keys()]}
114             unionsorted = {k:v for k,v in sorted(union.items(),key=lambda
                ↪ item: item[1], reverse=True)}
115             # print("$I \cap A=",unionsorted,"$")
116             # print("$A:",A,"$")
117             # print("$S:",S,"$")
118             # print("$X:",X,"$")
119             # Step 2: add v to S
120             try:
121                 v = list(unionsorted.keys())[0]
122                 # print("v:",v)
123             except: break
124             S.add(v)
125             # Step 3: Add the neighbours of v to X
126             removed = set(G.A[v])
127             # print("Step 3:",removed)
128             for i in removed:
129                 X.add(i)
130             # Step 4: Add vertices preceding v in the max-degree order on
                ↪ G[A] to X
131             step4 = list()
132             idx = list(A.keys()).index(v)
133             for i in list(A.keys())[:idx]:
134             # idx = list(A.keys()).index(v)
135             # for i in list(A.keys())[:idx]:
136                 X.add(i)
137                 removed.add(i)
138                 step4.append(i)
139             # print("Step 4:",step4)
140             # Step 5: Remove all vertices added to S and X from A
141             # print("Step 5:",removed)
142             removed.add(v)
143             for i in removed:
144                 for k in list(A.keys()):
145                     if k == i:
146                         del A[k]
147             iteration += 1
```

```
148             # print("$A:",A,"$")
149             # print("$S:",S,"$")
150             # print("$X:",X,"$")
151         if draw==True:
152             G2 = nx.empty_graph(G.n)
153             G2.add_edges_from(G.E)
154             pos = nx.circular_layout(G2)
155             nx.draw_networkx(G2,pos=pos)
156             nx.draw_networkx_nodes(G,pos=pos,nodelist=A,
                ↪  node_color='red',label='A')
157             nx.draw_networkx_nodes(G,pos=pos,nodelist=S,
                ↪  node_color='green',label='S')
158             nx.draw_networkx_nodes(G,pos=pos,nodelist=X,
                ↪  node_color='blue',label='X')
159             plt.legend()
160         if len(S) > np.ceil(2*delta*len(G.V)/d):
161             print('oh no!!')
162         SvA = S.union(A)
163         print("S:" ,S)
164         print("SvA ",SvA)
165         print("I:",I)
166 
167         # print("Testing if theorem holds:")
168         # print(len(S), "<=", np.ceil(2*delta*len(G.V)/d))
169         # print(len(SvA), "<=" , (1-delta)*len(G.V))
170         # print(print("Iteration",iteration))
171         # print("$|X|=",len(X),",\delta|V|=",delta*len(G.V),"$")
172         return set(A.keys()),S,X
173 
174 
175 def hypergraph_container_algorithm(I,H,draw=True):
176         # Sort A on max-degree
177         c = sort_degree(H)[0]
178         S = set()
179         d = avg_degree(H)
180         A = H
181         G = random_graph(H.n,0)
182         while len(S) < H.n/np.sqrt(d)-1:
183             # Step 1:let u be the first vertex in I\cupA in the
                ↪  max-degree order on A
184             union = {k:A.A[k] for k in [x for x in I if x in A.A.keys()]}
185             unionsorted = {k:v for k,v in sorted(union.items(),key=lambda
                ↪  item: item[1], reverse=True)}
186             try: u = list(unionsorted.keys())[0]
187             except: break
```

A

```
188             # Step 2: add u to S
189             S.add(u)
190             # Step 3: add xy to E(G) whenever uxy in E(H)
191             for x,y in combinations(H.V,2):
192                 if (u,x,y) in H.E:
193                     G.add_edge((x,y))
194             # Step 4: remove from V(A) the vertex u as well as all
                ↪ vertices proceeding
195             # u in the max_degree order on A
196             maxdegreeorder = sort_degree(A)
197             idx = list(maxdegreeorder.keys()).index(u)
198             for i in list(maxdegreeorder.keys())[:idx]:
199                 A.V.remove(i)
200                 A.A.pop(i)
201             A.V.remove(u)
202             A.A.pop(u)
203             # Step 5: remove from V(A) every vertex whose degree in G is
                ↪ larger than c*sqrt(d)
204             for k in A.V:
205                 if len(G.A[k]) > c*np.sqrt(d):
206                     A.V.remove(k)
207                     A.A.pop(k)
208             # Step 6: remove from E(A) every edge that contains an edge
                ↪ of G.
209             for x,y in G.E:
210                 for edge in A.E:
211                     if x in edge and y in edge:
212                         A.remove_edge(edge)
213     return S,A,G
214
215
216 n = 10
217 p = 0.7
218 latex = 'n'
219 # n = int(input("n: "))
220 # p = float(input("p: "))
221 # latex = input("Print latex code? (y/n)")
222
223 RG = random_graph(n,p)
224 G = nx.empty_graph(RG.n)
225 G.add_edges_from(RG.E)
226 I = sorted(nx.maximal_independent_set(G),reverse=True)
227 G = RG
228 A,S,X = graph_container_algorithm(I,G)
229
```

```
# H = random_3uniform_hypergraph(n,p)
# I = find_indepset(H)
# S,A,G = hypergraph_container_algorithm(I,H)
# print(I,S,A)

# print(S.intersection(I) == S)

#print(nx.to_latex_raw(G))


if latex == 'y':
    G2 = nx.empty_graph(G.n)
    G2.add_edges_from(G.E)
    pos = nx.circular_layout(G2)
    node_color = dict()
    for i in A:
        node_color[i] = 'red'
    for i in S:
        node_color[i] = 'green'
    for i in X:
        node_color[i] = 'blue'
    latex_code =
     ↪ nx.to_latex_raw(G2,pos,node_options=node_color,tikz_options=
     ↪ '[scale=4,every node/.append style={circle}]')
    print(latex_code)
```

A.2. THE GRAPH CONTAINER ALGORITHM USING THE ERDŐS-RÉNYI GRAPH

IN this section we look at the code implemented using the Erdős-Eényi model. This is standard within the `NetworkX` library so it might be easier to understand. It does not contain the hypergraph container algorithm, as this library does not work with hypergraphs.

```
# -*- coding: utf-8 -*-
"""
Created on Tue May  6 14:08:58 2025

@author: joachim
"""
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt


```

A

```
12 def sort_degree(G):
13     degrees = dict(G.degree(sorted(G.nodes,reverse=True)))
14     return {k:v for k, v in sorted(degrees.items(),key=lambda item:
       ↪ item[1],reverse=True)}
15
16 def find_max_indep_set(G):
17     return sorted(nx.maximal_independent_set(G),reverse=True)
18
19 def avg_degree(G):
20     degrees = dict(G.degree(G.nodes))
21     return sum(degrees.values())/len(degrees)
22
23 def draw_indepset(S,I,G):
24     pos = nx.circular_layout(G)
25     nx.draw_networkx(G,pos=pos)
26     nx.draw_networkx_nodes(G,pos=pos,nodelist=I,node_color='red',
       ↪ label='I')
27     nx.draw_networkx_nodes(G,pos=pos,nodelist=S,node_color='green',
       ↪ label='S')
28     plt.legend()
29
30 def container_algorithm(I,G,draw=True):
31     # Sort A on max-degree
32     A = sort_degree(G)
33     # Initialise
34     d = avg_degree(G)
35     c = list(A.values())[0]/d
36     delta = 1/4*c*(c+1)*0.99
37     print(delta)
38     # Initialise
39     S = set()
40     X = set()
41     # The algorithm itself
42     while len(X) < delta*len(G.nodes):
43         # Step 1: intersection of I and A in max-degree order on
           ↪ G[A]
44         union = {k:A[k] for k in [x for x in I if x in A.keys()]}
45         unionsorted = {k:v for k,v in sorted(union.items(),key=lambda
           ↪ item: item[1], reverse=True)}
46         # Step 2: add v to S
47         try:v = list(unionsorted.keys())[0]
48         except: break
49         S.add(v)
50         # Step 3: Add the neighbours of v to X
51         removed = set(G.neighbors(v))
```

```
52          for i in removed:
53              X.add(i)
54          # Step 4: Add vertices preceding v in the max-degree order on
             ↪ G[A] to X
55          idx = list(A.keys()).index(v)
56          for i in list(A.keys())[:idx]:
57              X.add(i)
58              removed.add(i)
59          # Step 5: Remove all vertices added to S and X from A
60          removed.add(v)
61          for i in removed:
62              for k in list(A.keys()):
63                  if k == i:
64                      del A[k]
65      if draw == True:
66          pos = nx.circular_layout(G)
67          nx.draw_networkx(G,pos=pos)
68          nx.draw_networkx_nodes(G,pos=pos,nodelist=A,
             ↪ node_color='red',label='A')
69          nx.draw_networkx_nodes(G,pos=pos,nodelist=S,
             ↪ node_color='green',label='S')
70          nx.draw_networkx_nodes(G,pos=pos,nodelist=X,
             ↪ node_color='blue',label='X')
71          plt.legend()
72      return set(A.keys()),S,X
73
74  n = 25
75  p = 0.5
76  G = nx.erdos_renyi_graph(n,p)
77  I = find_max_indep_set(G)
78  # print(d <= 2*delta*len(G.nodes))
79  A,S,X = container_algorithm(I,G)
80
81  # print(S.intersection(I) == S)
82  print("S: ",S)
83  print("SvA: ",S.union(A))
84  print("I: ",I)
85  # nx.to_latex_raw(G)
```

A

A.3. EXAMPLE 4.5

THIS section contains the code that has been used to generate example 4.5.

```
# -*- coding: utf-8 -*-
"""
Created on Sun Jun 29 15:59:03 2025

@author: joachim
"""
import numpy as np
import tabulate as tb

from graph_container_algorithm import
 ↪  random_graph,sort_degree,avg_degree


def graph_container_algorithm(I,G):
    # Sort A on max-degree
    A = sort_degree(G)
    print("Running algorithm on $G(25,0.5)$, with independent set
     ↪  $I=",I,"$")
    print("$G[A]:", A,"$")
    # Initialise
    d = avg_degree(G)
    print("Average degree:",d)
    c = list(A.values())[0]/d
    delta = 1/4*c*(c+1)*0.999
    print("$\delta=",delta,"$")
    S = set()
    X = set()
    # The algorithm itself
    iteration = 1
    while len(X) < delta*len(G.V):
        print("Iteration",iteration)
        print("$|X|=",len(X),",\delta|V|=",delta*len(G.V),"$")
        # Step 1: intersection of I and A in max-degree order on
         ↪  G[A]
        union = {k:A[k] for k in [x for x in I if x in A.keys()]}
        unionsorted = {k:v for k,v in sorted(union.items(),key=lambda
         ↪  item: item[1], reverse=True)}
        print("$I \cap A=",unionsorted,"$")
        print("$A:",A,"$")
        print("$S:",S,"$")
        print("$X:",X,"$")
        # Step 2: add v to S
```

```
        try:
            v = list(unionsorted.keys())[0]
            print("v:",v)
        except: break
        S.add(v)
        # Step 3: Add the neighbours of v to X
        removed = set(G.A[v])
        print("Step 3:",removed)
        for i in removed:
            X.add(i)
        # Step 4: Add vertices preceding v in the max-degree order on
        ↪ G[A] to X
        step4 = list()
        idx = list(A.keys()).index(v)
        for i in list(A.keys())[:idx]:
        # idx = list(A.keys()).index(v)
        # for i in list(A.keys())[:idx]:
            X.add(i)
            removed.add(i)
            step4.append(i)
        print("Step 4:",step4)
        # Step 5: Remove all vertices added to S and X from A
        print("Step 5:",removed)
        removed.add(v)
        for i in removed:
            for k in list(A.keys()):
                if k == i:
                    del A[k]
        iteration += 1
        print("$A:",A,"$")
        print("$S:",S,"$")
        print("$X:",X,"$")
    if len(S) > np.ceil(2*delta*len(G.V)/d):
        print('oh no!!')
    SvA = S.union(A)
    print("S:" ,S)
    print("SvA ",SvA)
    print("I:",I)

    # print("Testing if theorem holds:")
    # print(len(S), "<=", np.ceil(2*delta*len(G.V)/d))
    # print(len(SvA), "<=" , (1-delta)*len(G.V))
    # print(print("Iteration",iteration))
    # print("$|X|=",len(X),",\delta|V|=",delta*len(G.V),"$")
    return I,S,SvA
```

A

```
G = random_graph(25,0)


for i in [
    (0,2), (0,7), (0,8), (0,11), (0,12), (0,13),(0,14), (0,17),
    (0,18), (0,19),(0,20), (0,22), (1,3), (1,5),(1,7), (1,8),
    (1,9), (1,15),(1,16), (1,21), (1,23), (1,24),(2,6), (2,8),
    (2,10), (2,12),(2,13), (2,20), (2,23), (2,24),(3,4), (3,10),
    (3,11), (3,16),(3,20), (3,22), (3,24), (4,5),(4,7), (4,8),
    (4,9), (4,10),(4,11), (4,12), (4,13), (4,16),(4,19), (4,20),
    (4,21),(4,24),(5,6), (5,10), (5,12), (5,22),(6,9), (6,10),
    (6,12), (6,13),(6,17),(6,18), (6,19), (6,20),(6,21), (6,22),
    (7,8), (7,9),(7,11), (7,12), (7,14),(7,15),(7,16), (7,18),
    (7,19), (7,20),(7,23), (7,24), (8,10), (8,14),(8,17), (8,18),
    (8,19), (8,22),(8,23), (9,12), (9,13), (9,15),(9,16), (9,18),
    (9,21), (9,23),(9,24), (10,11), (10,12), (10,17),(10,19),
    (10,21),(10,24), (11,12),(11,13), (11,14), (11,18), (11,19),
    (11,20), (11,21), (11,23), (11,24),(12,14), (12,15), (12,16),
    (12,17),(12,21), (12,24), (13,14), (13,16),(13,17), (13,18),
    (13,19), (13,20),(13,22), (13,23), (13,24), (14,15),(14,19),
    (14,21), (14,24), (15,18),(15,22), (16,17), (16,21), (16,22),
    (16,23), (16,24), (17,19), (17,21),(18,19), (18,21), (18,22),
    (19,20),(19,21), (19,22), (19,24), (20,22),(20,23), (21,24),
    (22,23), (22,24)]:
    G.add_edge(i)


MaxIndepSets = [[6, 4, 1, 0], [7, 6, 3], [9, 5, 3, 0], [9, 8, 5, 3],
[10, 1, 0], [10, 9, 0], [11, 6, 1], [11, 9, 8, 5],
[13, 10, 1], [13, 10, 7], [13, 12, 1], [13, 12, 8, 3],
[14, 6, 4, 1], [15, 8, 6, 3], [15, 13, 10], [16, 14, 6],
[16, 15, 5, 0], [16, 15, 6, 0], [16, 15, 10, 0], [16, 15, 11, 5, 2],
[16, 15, 11, 8, 5], [16, 15, 11, 8, 6], [17, 7, 5, 3, 2],
[17, 11, 9, 5, 2], [17, 14, 9, 5, 3, 2], [17, 15, 4, 2],
[17, 15, 5, 3, 2], [17, 15, 11, 5, 2],[18, 16, 14, 5, 2],
[18, 17, 14, 4, 2, 1], [18, 17, 14, 5, 3, 2], [19, 2, 1],
[19, 9, 5, 3, 2], [19, 12, 1], [19, 15, 5, 3, 2], [19, 16, 15, 5, 2],
[20, 9, 8, 5], [20, 12, 8],[20, 14, 10, 9], [20, 16, 15, 8, 5],
[20, 16, 15, 10], [20, 17, 14, 9, 5], [20, 18, 12, 1],
[20, 18, 14, 10, 1],[20, 18, 16, 14, 5], [20, 18, 16, 14, 10],
[20, 18, 17, 14, 1], [20, 18, 17, 14, 5],[21, 7, 5, 3, 2],
[21, 13, 7, 5, 3], [21, 15, 5, 3, 2], [21, 15, 13, 8, 5, 3],
[21, 20, 15, 8, 5], [22, 10, 7], [22, 12, 1], [22, 14, 10, 1],
[22, 14, 10, 9], [22, 17, 7, 2], [22, 17, 11, 2, 1],
```

```
[22, 17, 11, 9, 2],[22, 17, 14, 4, 2, 1], [22, 17, 14, 9, 2],
[22, 21, 7, 2], [23, 14, 6, 3],[23, 14, 6, 4], [23, 15, 6, 3, 0],
[23, 15, 6, 4, 0], [23, 15, 10, 0],[23, 17, 15, 4],
[23, 17, 15, 5, 3], [23, 18, 12, 3], [23, 18, 14, 10],
[23, 18, 17, 14, 4],[23, 18, 17, 14, 5, 3], [23, 19, 12, 3],
[23, 19, 15, 5, 3],[23, 21, 15, 5, 3, 0], [24, 15, 8, 6],
[24, 20, 15, 8, 5],[24, 20, 17, 15, 5],[24, 20, 18, 17, 5],
[24, 23, 15, 5, 0],[24, 23, 15, 6, 0], [24, 23, 17, 15, 5],
[24, 23, 18, 17, 5]]

table = list()
containers = list()

for i in MaxIndepSets:
    I,S,SvA = graph_container_algorithm(i,G)
    table.append([set(I),S,SvA])
    if SvA not in containers:
        containers.append(SvA)

headers = ('I','S','S v A')
tabled = tb.tabulate(table,tablefmt='fancy_grid',headers=headers) #
 ↪  Change fancy_grid to latex for latex output
print(tabled)
```