DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Spectral Methods to Find Small Expansion Sets on Hypergraphs

Franz Rieger

DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Spectral Methods to Find Small Expansion Sets on Hypergraphs

Spektrale Methoden zum Finden kleiner Expansionsmengen auf Hypergraphen

Author:	Franz Rieger
Supervisor:	Prof. Susanne Albers
Advisor:	Dr. T.-H. Hubert Chan
Submission Date:	15. March 2019

I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15. March 2019 Franz Rieger

Acknowledgments

This thesis was written under the supervision of Dr. T.-H. Hubert Chan at the University of Hong Kong.

Abstract

The problem of finding set with low (edge) expansion on a graph can also be defined on hypergraphs. The expansion is the quotient of the summed weight of the edges crossing S and $V \setminus S$ and the summed weight of all the edges in S. In this thesis a brute force approach as well as an approximation algorithm for obtaining small sets with a low expansion are discussed, implemented and evaluated on different random hypergraphs. For the creation of the hypergraphs different algorithms are presented.

Contents

1 Introduction

To introduce the reader to the topic, a short overview of graphs and their generalization - hypergraphs - is given. Afterwards, the problem of cuts, especially edge expansion, shall be introduced.

1.1 2-Graphs

In graph theory a 2-graph $G := (V, E)$ is defined as a set of n vertices $V = \{v_1, \ldots, v_n\}$ and a set of m edges $E = \{e_1, \ldots, e_m\}$ where each edge $e_i = \{v_k, v_l\} \in E$ connects two vertices $v_k, v_l \in V$. A 2-graph can be seen in fig. 1.1. Note that in this thesis, for better overview, an edge is not displayed as a line between the vertices but as a coloured shape around the vertices.

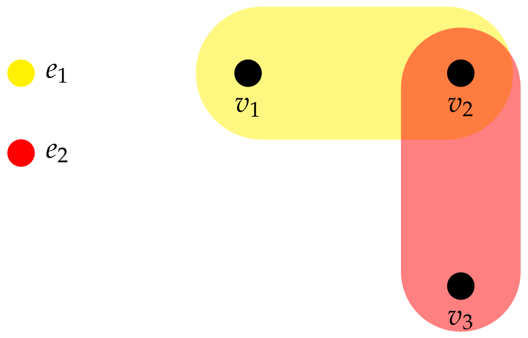

Figure 1.1: An example for a simple graph with three vertices and two edges.
$G = (\{v_1, v_2, v_3\}, \{\{v_1, v_2\}, \{v_2, v_3\}\})$

1.2 Hypergraphs

This thesis mostly deals with a generalized form of simple graphs, namely hypergraphs. A weighted, undirected hypergraph $H = (V, E, w)$ consists of a set of n vertices $V = \{v_1, \ldots, v_n\}$ and a set of m (hyper-)edges $E = \{e_1, \ldots, e_m | \forall i \in [i] : e_i \subseteq V \wedge e_i \neq \emptyset\}$ where every edge e is a non-empty subset of V and has a positive weight $w_e := w(e)$, defined by the weight function $w : E \rightarrow \mathbb{R}_+$. An example for a hypergraph can be seen in fig. 1.2.

The weight w_v of a vertex v is defined by summing up the weights of its edges: $w_v := \sum_{e \in E : v \in e} w_e$. Accordingly, a subset $S \subseteq V$ of vertices has weight $w_S := \sum_{v \in S} w_v$ and a subset $F \subseteq E$ of edges has weight $w_F = \sum_{e \in F} w_e$.

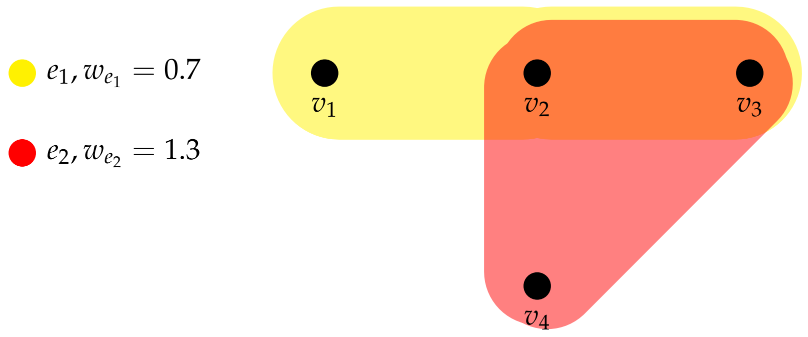

Figure 1.2: An example for a simple hypergraph with four vertices and two hyperedges. $G = (\{v_1, v_2, v_3, v_4\}, \{\{v_1, v_2, v_3\}, \{v_2, v_3, v_4\}\})$

The degree of a vertex $v \in V$ is defined as $deg(v) := |\{e \in E : v \in e\}|$, which is the count of edges which v is in contact with. A hypergraph where every vertex has exactly degree d, formally $\forall v \in V : deg(v) = d$, is called d-regular. A hypergraph where every edge contains exactly r vertices, formally $\forall e \in E : |e| = r$ is called r-uniform.

For d-regular, r-uniform hypergraphs, the following correlation with the number of vertices n and the number of edges m holds:

$$nd = mr \tag{1.1}$$

This can be verified in the following way: If a 'connection' is defined to be the where an edge and a vertex connect, one can count these connections from the vertices' perspective by summing up the degrees of all the vertices (which equals to nd for regular graphs). But one can also consider the count of connections from the edges perspective by summing up the ranks of the edges, which equates to mr for uniform graphs. As both ways count the same number of connections, eq. (1.1) holds. (TODO: source?)

A path between two vertices $v_1, v_k \in V$ is a list of vertices $v_1, v_2 \ldots, v_k$ where each tuple of vertices following another is connected by an edge, i.e. $\forall i \in [k-1] \exists e \in E : u_i, u_{i+1} \in e$. A connected component in a undirected graph is a subset of vertices $S \subseteq V$ where for every two vertices $u, v \in S$ there exists a path, between u and v. If $S = V$, the whole graph is called connected.

1.3 Cuts

On such hypergraphs certain properties can be described, which are of theoretical interest but also have influence on the behaviour of a system which is described by such a graph. Some of these properties are so called cuts. A cut is described by its cut-set $\emptyset \neq S \subsetneq V$, a non-empty strict subset of the vertices. Interesting cuts are for example the so called minimum cut or the maximum cut which are defined by the minimum (or maximum) number of edges (or their added weight for weighted graphs) going between S and $V \setminus S$. Formally, this can be expressed by the following equation:

$$MinCut(G) := \min_{\emptyset \subsetneq S \subsetneq V} \sum_{e \in E : \exists u,v \in e : u \in S \wedge v \in V \setminus S} w_e \tag{1.2}$$

For computing the minimum cut the Stoer–Wagner algorithm can be used, which has a polynomial time complexity in the number of vertices [1]. The maximum cut problem however, is known to be NP-hard [2].

1.4 Edge Expansion

The cut on which this thesis focuses on is the so-called edge expansion (also just expansion), which is the quotient of the summed weight of the edges crossing S and $V \setminus S$ and the summed weight of all the vertices in S. The formal notation is introduced in the following and oriented on the article [4], on which the crucial approximation algorithm of this thesis is based.

The set of edges which are cut by S contains all the edges, which have at least one vertex in S and at least one vertex in $V \setminus S$ and is defined as

$$\partial S := \{e \in E : e \cap S \neq \emptyset \wedge e \cap (V \setminus S) \neq \emptyset\}. \tag{1.3}$$

The weight $w(S)$ of a set S of vertices is defined as the summed weight of all the vertices in the set:

$$w(S) := \sum_{v \in S} w_v \tag{1.4}$$

The weight $w(F)$ of a set F of edges is defined as the summed weight of all the edges in the set:

$$w(F) := \sum_{e \in F} w_e \tag{1.5}$$

With that, the edge expansion of a non-empty set of vertices $S \subseteq V$ is defined by

$$\Phi(S) := \frac{w(\partial S)}{w(S)}. \tag{1.6}$$

For a better understanding of the expansion, observe that $\Phi(S)$ is bounded:

$$\forall \emptyset \neq S \subseteq V : 0 \leq \Phi(S) \leq 1 \tag{1.7}$$

The first inequality holds because the edge-weights are positive. The second inequality holds because $W(S) \geq W(\partial S)$, as $W(S)$ takes at least every edge (and therefore the corresponding weight), which is also considered by $W(\partial S)$, into account.

With this, the expansion of a graph H is defined as

$$\Phi(H) := \min_{\emptyset \subsetneq S \subsetneq V} \max\{\Phi(S), \Phi(V \setminus S)\}. \tag{1.8}$$

Here again, $0 \leq \Phi(H) \leq 1$ holds because of eq. (1.7).

In order to comprehend the edge expansion of a graph better, some special cases shall be considered. For non-connected graphs $\Phi(H) = 0$ holds, which can be verified by observing a S which contains of the vertices of one connection component, for an example refer to fig. 1.3. As this thesis focuses on finding sets S with a low expansion $\Phi(S)$, graphs with expansion 0 are a special case. Therefore only connected graphs shall be of interest here.

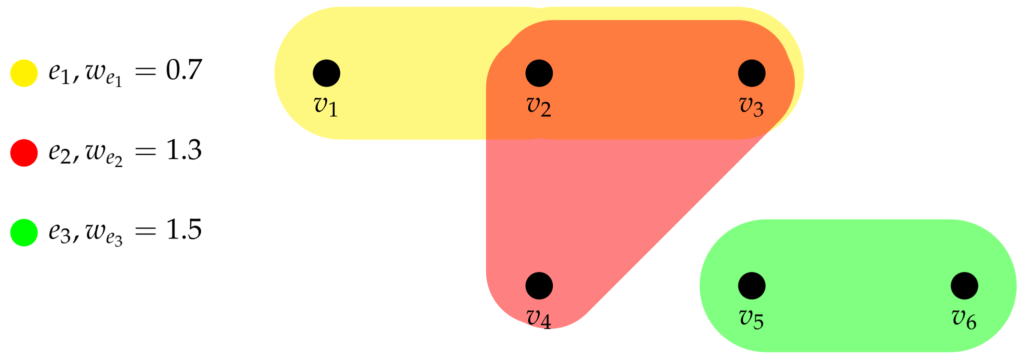

Figure 1.3: An example for a non-connected hypergraph with two connection components. For $S := \{v_5, v_6\}$ it can be verified that $\delta S = 0$, hence $\Phi(S) = \Phi(V \setminus S) = 0$.

Observe that for a graph H, which is obtained by connecting two connection components with an edge with small weight, $\Phi(H)$ takes a small value, which can be seen when S is chosen to be one of the previously separated connection components. For a fully connected graph, where each vertex shares at least one edge with every other vertex, which has equal edge-weights, ∂S will be relatively high for every $S \subsetneq V$. Therefore $\Phi(S)$ and ultimately also $\Phi(H)$ will take a high value.

The problem of computing the expansion $\Phi(H)$ on a hypergraph is NP-hard, as it is already NP-hard on 2-uniform-graphs, a special case of hypergraphs [3]. However, there exist polynomial time approximation algorithms for some relaxations of this problem, one of them will be focused on here: For certain applications, it can be interesting to find small expansion sets S, where the vertices are strongly connected within the set but only have a weak connection to the rest of the vertices, hence a set with a low expansion value $\Phi(S)$ is desired. Here, small refers to the number of vertices, so $|S|$ should be low compared to the total number of vertices $n = |V|$. In the presented algorithm, with high probability, sets which have at max a constant fraction $\frac{1}{c}$ of the total number of vertices $|V|$ are computed, formally $|S| \leq \frac{|V|}{c}$.

Finding such a S will be achieved by algorithm 4, which was deducted from results in [4]. As the algorithm uses spectral properties of graphs, some notation is introduced in chapter 2. In chapter 3 this algorithm and the algorithms it is based on as well as brute force solutions are presented. The involved constant shall be estimated in a empirical manner by running the algorithm multiple times on different random graphs, whose creation is discussed in chapter 4. Details regarding the implementation of the algorithms can be found in chapter 5, which is followed by the evaluation of the results in chapter 6. Possible applications of the algorithm are discussed in chapter 7 and finally, chapter 8 discusses future work and completes the thesis.

TODO: random graphs

TODO: fix small/low big/high everywhere

2 Notation

For further use, especially for the facts and the algorithms called by algorithm 4, some noation shall be introduced. The notation used in this thesis is orientated on [4].

The weight matrix of a hypergraph, which contains the vertices' weights on its diagonal can be denoted as

$$W := \begin{pmatrix} w_{v_1} & 0 & 0 & \dots & 0 \\ 0 & w_{v_2} & 0 & \dots & 0 \\ 0 & 0 & w_{v_3} & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & w_{v_n} \end{pmatrix} \in \mathbb{R}_{0+}^{n \times n}. \tag{2.1}$$

The discrepancy ratio of a graph, given a non-zero vector $f \in \mathbb{R}^V$ is defined as

$$D_w(f) := \frac{\sum_{e \in E} w_e \max_{u,v \in e}(f_u - f_v)^2}{\sum_{u \in V} w_u f_u^2}. \tag{2.2}$$

To understand the discrepancy ratio TODO Observe that $0 \leq D_w(f) \leq 2$ [4].

In the weighted space, in which the discrepancy ratio is defined like above, for two vectors $f, g \in \mathbb{R}^V$ the inner product is defined as $\langle f, g \rangle_w := f^T W g$. Accordingly, the norm is $||f||_w = \sqrt{\langle f, f \rangle_w}$. If $\langle f, g \rangle_w = 0$, f and g are said to be orthonormal in the weighted space.

The discrepancy ratio can also be defined on the so called normalized space with $x \in \mathbb{R}^n$:

$$\mathcal{D}(x) = D_w(W^{-\frac{1}{2}} x) \tag{2.3}$$

A so called orthogonal minimaximizer can be defined as follows. For k mutually orthogonal non-zero vectors of the weighted space:

$$\xi_k := \min_{x_1, \dots, x_k, orthogonal} \max_{i \in [k]} \mathcal{D}(x_i) \tag{2.4}$$

TODO:explain weighted space + normalized space

Todo explain minimaximizer, Discrepancy ration in ... space, gamma 2 Todo: weighted in which space orthogonal Laplacian? Eigenvalues? TODO: move content to other chapters?

3 Expansion Algorithms

In the following chapter different approaches for generating small expansion sets S as well as the edge expansion of hypergraphs are discussed.

3.1 Brute Force

One obvious approach for generating the edge expansion $\Phi(H)$ of a hypergraph H is to brute-force the problem like in algorithm 1.

Algorithm 1 Brute-force edge expansion on a hypergraph

```
function BruteForceEdgeExpansion($H := (V, E, w)$)
    $best\_S := null$
    $lowest\_expansion := \infty$
    for $\emptyset \neq S \subsetneq V$ do
        $expansion := \max\{\Phi(S), \Phi(V \setminus S)\}$
        if $expansion < lowest\_expansion$ then
            $lowest\_expansion := expansion$
            $best\_S := S$
    return $best\_S$
```

As algorithm 1 iterates over all the possible subsets $\emptyset \neq S \subsetneq V$, it computes

$$\arg \min_{\emptyset \subsetneq S \subsetneq V} \max\left(\Phi(S), \Phi(V \setminus S)\right) = \Phi(H). \tag{3.1}$$

There are $2^{|V|} - 2 = 2^n - 2 \in O(2^n)$ combinations for $\emptyset \neq S \subsetneq V$, namely all the $2^{|V|}$ subsets of V excluding the empty set $\emptyset$ and V itself. Hence, this algorithm is of exponential time complexity in n and is therefore not efficient for larger graphs as evaluated in fig. 6.1.

For the purpose of analyzing the graph creation algorithms in chapter 4, it can be insightful to observe the lowest expansion of each possible size (in the number of vertices) like in algorithm 2.

Algorithm 2 Brute-force edge expansion of a hypergraph for every size

```
function BruteForceEdgeExpansionSizes($H := (V, E, w)$)
    $best\_S\_of\_size := \{\}$
    $lowest\_expansion\_of\_size := \{1 : \infty, 2 : \infty, \ldots, n-1 : \infty\}$
    for $\emptyset \neq S \subsetneq V$ do
        $expansion := \max \Phi(S), \Phi(V \setminus S)$
        if $expansion < lowest\_expansion\_of\_size[|S|]$ then
            $lowest\_expansion\_of\_size[|S|] := expansion$
            $best\_S\_of\_size[|S|] := S$
    return $best\_S\_of\_size$
```

In order to analyze the results from algorithm 4, which only computes $\Phi(S)$, the expansion of a set S, not the whole graph, it makes sense to compare it with the best result possible for the same size of S. Therefore, just the line for the computation of the expansion algorithm 2 needs to be changed to get algorithm 3.

Algorithm 3 Brute-force edge expansion of sets for every size

```
function BruteForceEdgeExpansionSizes($H := (V, E, w)$)
    $best\_S_o f_s ize := \{\}$
    $lowest\_expansion\_of\_size := \{1 : \infty, 2 : \infty, \ldots, n-1 : \infty\}$
    for $\emptyset \neq S \subsetneq V$ do
        $expansion := \Phi(S)$
        if $expansion < lowest\_expansion\_of\_size[|S|]$ then
            $lowest\_expansion\_of\_size[|S|] := expansion$
            $best\_S\_of\_size[|S|] := S$
    return $best\_S\_of\_size$
```

For algorithm 2 and algorithm 3 the above argument of for exponential complexity holds as well.

3.2 Approximation of small expansion sets

As described in [4], an algorthm for generating a random small expansion set can be derived.

For a given Graph H we can find a small expansion set with algorithm 4, where $k \geq 2$ is an integer.

Algorithm 4 Find Small Expansion Set

```
function SmallExpansionSet($H, k$)
    $f_1 \dots, f_k$ :=SampleSmallVectors($H, k$)
    return SmallSetExpansion($H, f_1 \dots, f_k$)
```

In the main algorithm, the first call to algorithm 5, returns a set of orthonormal vectors $\{f_1, \dots, f_k\}$, where each vector $f_i \in \mathbb{R}^V$ gives a value to each vertex. Because of fact 2 it is of importance for algorithm 7 that $\xi := \max_{s \in [k]} D_w(f_s)$ is small. At first, f_1 is set to be $\frac{\vec{1}}{||\vec{1}||_w}$. Following that, the other vectors $f_2, \dots, f_k$ are sampled after another, using algorithm 6. This sampling of vectors after another where the discrepancy ratio of the next vector shall be minimal is also referred to as procedural minimizers. In this way an approximation for the ideal vectors, which achieve ξ_k as defined in eq. (2.4) is achieved.

Algorithm 5 Procedural Minimizer

```
function SampleSmallVectors($H, k$)
    $f_:1 = \frac{\vec{1}}{||\vec{1}||_w}$
    for $i = 2, \dots, k$ do
        $f_i$ :=SampleRandomVector($H, f_1, \dots, f_{i-1}$)
    return $f_1, \dots, f_k$
```

SDP 1 *SDP for minimizing g, (SDP 8.3 in [4])*

$$\begin{aligned} \underset{g}{\text{minimize}} \quad & SDPval := \sum_{e \in E} w_e \max_{u,v \in e} ||\vec{g}_u - \vec{g}_v||^2 \\ \text{subject to} \quad & \sum_{u \in V} w_v ||\vec{g}_v||^2 = 1, \\ & \sum_{u \in V} w_v f_i(v) \vec{g}_v = \vec{0}, \quad \forall i \in [k-1] \end{aligned}$$

In algorithm 6, SDP 1 is solved in order to generate vectors $\vec{g}_v \in \mathbb{R}^n$ for $v \in V$. The idea behind the vector $\vec{g}_v$ is to represent the coordinate v in the next vector f, which is being created. Therefore, $\vec{g}_v$ in the SDP relates to f_v in the discrepancy ratio defined in eq. (2.2). The first constraint limits the norm of the vector whilst the following ensure orthonormality to the already existing vectors. By sampling a vector from a gaussian and multiplying it with all the $\vec{g}_v$, the coordinates of f are created. According to fact 1, the maximal discrepancy ratio $\max_{s \in [k]} D_w(f_s)$, among the vectors which are returned by algorithm 6, is small with high probability. Therefore, in the implementation of algorithm 6, the steps after solving the SDP are repeated several times and the f with the smallest $D_w(f)$ is returned.

Fact 1 *(Theorem 8.1 in [4]) There exists a randomized polynomial time algorithm that, given a hypergraph $H = (V, E, w)$ and a parameter $k < |V|$, outputs k orthonormal vectors $f_1, \ldots, f_k$ in the weighted space such that with high probability, for each $i \in [k]$,*

$$D_w(f_i) \le \mathcal{O}(i\xi_i \log r). \tag{3.2}$$

Algorithm 6 Sample Random Vector (Algorithm 3 in [4])

```
function SampleRandomVector($H, f_1, \ldots, f_{i-k}$)
    Solve SDP 1 to generate vectors $\vec{g_v} \in \mathbb{R}^n$ for $v \in V$
    $\vec{z} := sample(\mathcal{N}(0, I_n))$
    for $v \in V$ do
        $f(v) :=< \vec{g_v}, \vec{z} >$
    return $f$
```

After sampling of the vectors $f_1, \ldots, f_k$, algorithm 4 calls algorithm 7. There the vectors $f_1, \ldots, f_k$ are flipped to form u_v. Each u_v represents the f-values for a vector $v \in V$. After normalizing each u_i to $\tilde{u}_i$, they are handed over to algorithm 8, which returns a subset of the $\tilde{u}_v$s. With this subset, a vector X is constructed. X_i takes the value $||u_v||$ if $\tilde{u}_v \in \hat{S}$, otherwise 0. Then, X is sorted in decreasing order. Afterwards all the prefixes of X are analyzed for the expansion of their respecting vertices. The set of vertices S with the lowest expansion is returned.

Algorithm 7 Small Set Expansion (according to Algorithm 1 in [4])

```
function SmallSetExpansion($G := (V, E, w), f_1, \dots, f_k$)
    assert $\xi == \max_{s\in[k]}\{D_w(f_s)\}$
    assert $\forall f_i, f_j \in \{f_1, \dots, f_k\} \subset \mathbb{R}^n, i \neq j$ : $f_i$ and $f_j$ orthonormal in weighted space
    for $v \in V$ do
        for $s \in [k]$ do
            $u_v(s) := f_s(v)$
    for $v \in V$ do
        $\tilde{u}_v := \frac{u_v}{||u_v||}$
    $\hat{S} :=$ OrthogonalSeparator($\{\tilde{u}_v\}_{v\in V}, \beta = \frac{99}{100}, \tau = k$ )
    for $i \in S$ do
        if $\tilde{u}_v \in \hat{S}$ then
            $X_v := ||u_v||^2$
        else
            $X_v := 0$
    $X :=$ sort $list(\{X_v\}_{v\in V})$
    $V := [v]_{\text{in order of X}}$
    $S := \arg\min_{\{P \text{ is prefix of } V\}} \phi(P)$
    return $S$
```

In algorithm 8 a number of $l := \lceil \frac{\log_2 k}{1-\log_2 k} \rceil$ assignments are sampled for each vertex $u \in V$ with the help of algorithm 9. Assignment $j \in [l]$ assigns a value $w_j(u) \in \{0, 1\}$ to each vertex u. With that, for each vertex u, a word $W(u) = w_1(u)w_2(u) \cdots w_l(u)$ can be defined. Then a random word is picked, depending on the size of n and 2^l, either from $\{0, 1\}^l$ or from all the constructed words with the same probability. In this case, $|V| -$ #distinct words in W are constructed and added to the set of words to pick from. Following, a value $r \in (0, 1)$ is chosen uniformly at random. Only those vertices v whose word $W(v)$ equals the chosen *word* and whose vector $\tilde{u}_v$ is smaller than r get selected into the set S which is returned to algorithm 7.

For sampling the assignments algorithm 9 uses a Poisson process on $\mathbb{R}$ with rate λ. It is important to note that for one call of the algorithm, the times on which the events happen do not change. Therefore, given a time t, the process returns the number of events which have happened between $t_0 = 0$ and t. Observe that $t \in \mathbb{R}$ can also take negative values. Additionally, a vector $g \in \mathbb{R}^n$ is sampled where each component is sampled independently from $\mathcal{N}(0, 1)$. Then for each vertex v and for $i = 1, 2, \dots, l$ a 'time' $t = \langle g, \tilde{u}_v \rangle$ is calculated. Depending on whether the number of events happened in the Poisson process until t, is even or odd, $w_i(v)$ is set to 0 or 1. Finally, w is returned.

Algorithm 8 Orthogonal Separator (combination of Lemma 18 and algorithm Theorem 10 in [5] (also Fact 6.7 in [4]))

```
function OrthogonalSeparator($\{\tilde{u}_v\}_{v \in V}, \beta = \frac{99}{100}, \tau = k$)
    $l := \lceil \frac{\log_2 k}{1 - \log_2 k} \rceil$
    $w :=$SampleAssignments($l, V, \beta$)
    for $v \in V$ do
        $W(u) := w_1(v) w_2(v) \cdots w_l(v)$
    if $n \geq 2^l$ then
        $word := random(\{0,1\}^l)$                                        ▷ uniform
    else
        $words := set(w(v) : v \in V)$                                  ▷ no multiset
        $words = words \uplus \{w_1, \ldots, w_{|V|-|words|} \in \{0,1\}^l\}$
        $word := random(words)$                                            ▷ uniform
    $r := uniform(0,1)$
    $S := \{v \in V : ||u_v||^2 \geq r \wedge W(u) = word\}$
    return $S$
```

Algorithm 9 Sample Assignments (proof of Lemma 18 in [5])

```
function SampleAssignments($l, V, \beta$)
    $\lambda := \frac{1}{\sqrt{\beta}}$
    $g :=$ sample($\mathcal{N}(0, I_n)$)          ▷ all components $g_i$ are mutually independent
    $poisson\_process := N(\lambda)$           ▷ N is a Poisson process on $\mathbb{R}$ with rate $\lambda$
    for $i = 1, 2, \ldots, l$ do
        for $v \in V$ do
            $t := \langle g, \tilde{u}_v \rangle$
            $poisson\_count := poisson\_process(t)$       ▷ # events between $t = 0$ and $t_v$
            if $poisson\_count \mod 2 == 0$ then
                $w_i(v) := 1$
            else
                $w_i(v) := 0$
    return $w$
```

Fact 2 *(Theorem 6.6 in [4]) Given a hypergraph $H = (V, E, w)$ and k vectors $f_1, f_2, \ldots, f_k$ which are orthonormal in the weighted space with $\max_{s\in[k]} D_w(f_s) \leq \xi$, the following holds: Algorithm 7 constructs a random set $S \subsetneq V$ in polynomial time such that with $\Omega(1)$ probability, $|S| \leq \frac{24|V|}{k}$ and*

$$\phi(S) \leq C \min\{\sqrt{r \log k}, k \log k \log\log k \sqrt{\log r}\} \cdot \sqrt{\xi}, \tag{3.3}$$

where C is an absolute constant and $r := \max_{e\in E} |e|$.

Todo: this makes use of the spectral properties... Rayleigh, laplacian, cheeger,

4 Random hypergraph generation

In order to evaluate the algorithms of chapter 3 hypergraphs are require as inputs. However, instead of creating a few hypergraphs by hand, they shall be randomly generated in order to have a diverse array of graphs.

The initial task was to find an algorithm which creates random r-uniform, d-regular, connected hypergraphs with no doubled edges in an effective manner which is guaranteed to terminate. Effective refers to a polynomial time complexity in the number of vertices, rank and the number of edges or the desired degrees respectively. Additionally, every graph which fulfills these criteria shall be created with equal probability.

However, this intention showed to be a non-trivial challenge. Therefore, several different approaches which fulfill some of these criteria will be discussed and their resulting graphs shall also be analyzed by their edge expansion.

4.1 Creating all Graphs

Algorithm 10 Generate sampling from all Graphs

```
function GenerateAllGraphs($n, r, d, weightDistribution$)
    $H_{n,r,d,weightDistribution} = \{H : H$ is d-regular, r-uniorm and connected with unique edges$\}$
    return $H = choose(H_{n,r,d,weightDistribution})$        ▷ uniformly at random
```

The first approach generates every possible connected, d-regular, r-uniform graph with unique edges with the same probability. However, (even with ignoring the weight distribution) creating of all the graphs which fulfill these properties is very expensive as there are alone $\binom{n}{r}$ possibilities for the first edge already (if the ordering is considered). This makes the algorithm impracticable.

4.2 Adding random edges

To start with, a simple algorithm to generate graphs which follows some of the intentions shall be discussed. Algorithm 11 simply samples edges by repeatedly randomly choosing r vertices of V. This algorithm is guaranteed to terminate, as it contains no

Algorithm 11 Generate by adding random edges

```
function GenerateAddRandomEdges($n, r, numberEdges, weightDistribution$)
    $E := \emptyset$
    $V := \{v_1, \dots, v_n\}$
    $w = \{\}$
    for $1, \dots, numberEdges$ do
        $nextEdge := sample(V, r)$
        $E := E \cup \{nextEdge\}$
        $weight(nextEdge) := sample(weightDistribution)$
    return $H = (V, E, w)$
```

conditioned loops. As all the operations, especially the sampling can be performed in polynomial time complexity and no conditioned loops or recursive calls is performed, polynomial time complexity can be assumed. Furthermore, the resulting graph is uniform, as all the edges contain exactly r vertices.

As there is no restriction on how the edges are to be added, every graph which fulfills the abovementioned criteria can be constructed. This can be verified by the following argument: Assume a $H = (V, E, w)$ can not be constructed by algorithm 11. Say $m := |E|$. Chose any edge $e \in E$ and remove it (and the corresponding weight) to construct $H' = (V, E', w')$. It can be seen that $|E'| = |E| - 1 = m - 1$. Hence, if this process is repeated until no edges are left, one can execute the algorithm's main loop and with non-zero probability chose exactly those edges (and their corresponding weights) which have been removed in the opposite order. In the end one would end up with exactly H again, contradicting that it can not be constructed.

However, the algorithm might never sample one vertex $v \in V$, therefore the rank of this vertex would be 0 which does make the graph possibly non-regular (as other vertices would have a degree > 0 and also not connected. Also, the algorithm does not guarantee to have no doubled edges.

4.3 Bound on vertex degrees

The first idea of algorithm 11 can be improved by ensuring the degree of the vertices do not exceed d as shown in algorithm 12. This algorithm repeadetly samples as long as there are at least r vertices left which have a degree lower than d. It is again terminating, and the resulting graph is r-uniform and all the possible graphs can be constructed.

Again, this algorithm is of polynomial runtime complexity, as there can be an upper bound on the number of executions of the loop: A graph on n vertices with rank r

Algorithm 12 Generate random graph with upper bound on degrees

```
function GenerateRandomGraphBoundDegrees($n, r, d, weightDistribution$)
    $E := \emptyset$
    $V := \{v_1, \dots, v_n\}$
    $w = \{\}$
    while $|\{v \in V | deg(v) < d\}| \geq r$ do
        $nextEdge := sample(\{v \in V | deg(v) < d\}, r)$        ▷ draw without replacement
        $E := E \cup \{nextEdge\}$
        $weight(nextEdge) := sample(weightDistribution)$
    return $H = (V, E, w)$
```

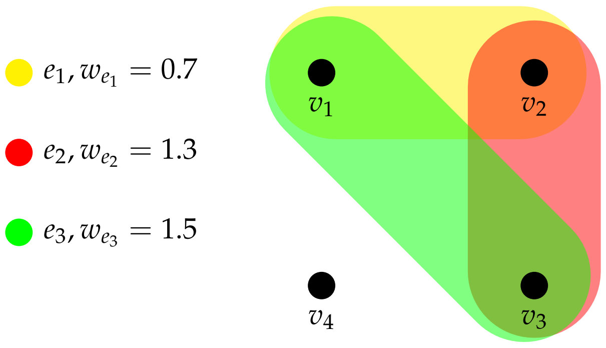

Figure 4.1: An example for a non-connected 2-uniform hypergraph which could have been created by algorithm 12.

and degree at max d can have at most $m \leq \frac{nd}{r}$ edges according to eq. (1.1). As every execution of the loop creates an edge, the loop will execute at most m times.

However, it is not guaranteed that this graph is connected and it is possible that some ($< r$) vertices do not have degree d in the end, because they have not been sampled before. An example of such a situation can be seen in fig. 4.1. Also, it does not guarantee to have no doubled edges.

4.4 Sampling from low degree vertices

To overcome these problems, the edges could only be sampled from the vertices with the smallest degrees like in algorithm 13. If at some point there are less than r vertices which share the lowest degree, as many vertices, which have the next higher degree, as needed for a full edge are sampeled.

The algorithm is guaranteed to terminate, and of polynomial time complexity for

Algorithm 13 Generate random hypergraph, sampling from lowest degrees

```
function GenerateSampleSmallestDegrees($n, r, d, weightDistribution$)
    $E := \emptyset$
    $V := \{v_1, \dots, v_n\}$
    while $|\{v \in V | deg(v) < d\}| \geq r$ do
        $smallestDegreeVertices := \{v \in V | deg(v) = \min_{u \in V} deg(u)\}$
        if $|smallestDegreeVertices| >= r$ then
            $nextEdgeVertices := sample(smallestDegreeVertices, r)$
        else
            $secondSmallestDegreeVertices := \{v \in V | deg(v) = \min_{u \in V} deg(u) + 1\}$
            $nextEdgeVertices := sample(secondSmallestDegreeVertices, r - |smallestDegreeVertices|)$
            $nextEdgeVertices := smallestDegreeVertices \cup nextEdgeVertices$
        $nextEdgeWeight := sample(weightDistribution)$
        $nextEdge := nextEdgeVertices$
        $E := E \cup \{nextEdge\}$
        $w(e) := nextEdgeWeight$
    return $G := (V, E, w)$
```

the same reasons as algorithm 12. Again, the resulting graph is r-uniform, but it is also d-regular, assuming there exists an integer m for the combination of n, r and d in eq. (1.1).

However, not all the possible graphs can be constructed: This algorithm basically constructs the edges by d r-matchings. TODO: source/example But not every graph can be dissembled into d r-matchings.

Again this graph is not necessarily connected and some edges might be doubled, as it proves challenging to avoid the following situation: Say a graph is being generated and only one edge is missing and r vertices have degree d-1. However, there already exists an edge consisting of those r vertices, hence the next edge would be a doubled edge. The first idea which might occur to solve this problem might be to keep a track of the combinations of vertices which are still possible as edges, combined with their remaining number of connections until they reach degree d, from the beginning. Then one could avoid choosing paths which end up with doubled edges. However, this seems to be virtually impossible due to the sheer number of combinations in $\binom{n}{r}$. Therefore, one other remaining way for ensuring unique edges is to resample the graphs (as whole or just some edges) if there are doubled edges.

To solve this, there are again several options which shall be discussed in the following.

4.5 Resampling whole graph until connected

Algorithm 14 resamples the whole graph, if the graph is not connected. This way, the algorithm loses the property of guaranteed terminating. The time complexity would depend on the probability of creating a graph which fulfills the requirements. However, this shall not be analyzed here.

This algorithm can be exteded by checking for more properties like no doubled edges and regular degrees and resample if those conditions are not met. However, more restrictions would decrease the chance of a created graph to fulfill all of the restrictions, possibly increasing the number of repetitions by an exponential level.

Algorithm 14 Generate random graph with resampling

```
function GenerateRandomGraph($n, r, d, weightDistribution$)
    $G :=$ GenerateAddRandomEdges($n, r, \frac{nd}{r}, weightDistribution$)
    while not connected$(G)$ do
        $G :=$ GenerateAddRandomEdges($n, r, \frac{nd}{r}, weightDistribution$)
    return $G := (V, E)$
```

4.6 Swapping edges at random

Instead of resampling the whole graph, one could also modify the graph by changing the edges in some way. In algorithm 15, as long as the graph is not connected, two edges $e, f \in E$ are selected and two vertices $u \in e, v \in f$ in those edges. Then, if the vertices do not belong to the same connection component, they are removed from their respective edges and added to the other one. By only 'swapping' if they do not belong to the same connection component, it shall be ensured that the number of connection components does not increase, as there are some situations where this operation can split a connected component into two separate components.

As the graph created by algorithm 13 is d-regular, this algorithm 15 will not change that, as for every edge which is removed from a vertex, another one is added. This also holds for the edges, therefore the graph is also r-uniform. Doubled edges can still occur and it is not guaranteed that the algorithm terminates, as it might never swap those edges and vertices which would be needed for connecting the graph.

Algorithm 15 Generate by randomly swapping edges,

```
function GenerateSwapEdges($n, r, d, weightDistribution$)
    $G :=$ GenerateSampleSmallestDegrees($n, r, d, weightDistribution$)
    while not Connected(G) do
        $e, f := sample(E, 2)$
        $u := sample(e)$
        $v := sample(f)$
        if $connection\_component(u) \neq connection\_component(v)$ then
            $e := (e \cup \{v\}) \setminus \{u\}$
            $f := (f \cup \{u\}) \setminus \{v\}$
    return $G := (V, E, w)$
```

4.7 Creating of spanning tree

The idea behind algorithm 16 is to ensure the graph is connected in the beginning by creating a spanning tree. Afterwards the edges are sampled from the vertices of lowest degree like in algorithm 13 in order to ensure regularity.

Therefore, the graphs generated will again be d-regular and r-uniform and might contain doubled edges. The algorithm is guaranteed to terminate and of polynomial time complexity. TODO The algori connected terminating polynomial time complexity all possible graphs all with equal probability

4.8 Overview

An overview of the properties of the discussed algorithms can be seen in table 4.1. The different ideas used in these algorithms can also be combined in other ways as indicated before. Therefore, it is important to note that this study of creation algorithms is not at all exhaustive. More sophisticated random graph models are discussed in [6], [7].

Implemented: algorithm 14, algorithm 15, algorithm 16 TODO

Algorithm 16 Generate random graph by creating a spanning tree

```
function GenerateWithSpanningTree($n, r, d, weightDistribution$)
    $V := \{v_1, \ldots, v_n\}$
    $E := choice(V, r)$
    while $\{v \in V | deg(v) = 0\} \neq \emptyset$ do
        if $|\{v \in V | deg(v) = 0\}| \geq r$ then
            $nextEdgeTreeVertex := choice(\{v \in V | deg(v) = 1\})$    ▷ get one tree node
            $nextEdgeVertices := choice(\{v \in V | deg(v) = 0\}, r - 1) \cup \{nextEdgeTreeVertex\}$
        else
            $nextEdgeVertices := \{v \in V | deg(v) = 0\}, \cup choice(\{v \in V | deg(v) > 0\}, |\{v \in V | deg(v) = 0\}|)$
        $nextEdgeWeight := sample(weightDistribution)$
        $nextEdge := nextEdgeVertices$
        $E := E \cup \{nextEdge\}$
        $w(e) := nextEdgeWeight$
    while $|\{v \in V | deg(v) < d\}| \geq r$ do
        $smallestDegreeVertices := \{v \in V | deg(v) = \min_{u \in V} deg(u)\}$
        if $|smallestDegreeVertices| >= r$ then
            $nextEdgeVertices := sample(smallestDegreeVertices, r)$    ▷ draw without replacement
        else
            $secondSmallestDegreeVertices := \{v \in V | deg(v) = \min_{u \in V} deg(u) + 1\}$
            $nextEdgeVertices := sample(secondSmallestDegreeVertices, r - |smallestDegreeVertices|)$
            $nextEdgeVertices := smallestDegreeVertices \cup nextEdgeVertices$
        $nextEdgeWeight := sample(weightDistribution)$
        $nextEdge := nextEdgeVertices$
        $E := E \cup \{nextEdge\}$
        $w(e) := nextEdgeWeight$
    return $G := (V, E, w)$
```

property \Algorithm	10	11	12	13	14	15	16
r-uniform	✓	✓	✓	✓	✓	✓	✓
d-regular	✓	✗	✗	✓	✗	✓	✓
unique edges	✓	✗	✗	✗	✗	✗	✗
connected	✓	✗	✗	✗	✓	✓	✓
terminating	✓	✓	✓	✓	✗	✗	✓
polynomial time complexity	✗	✓	✓	✓	?	?	✓
all possible graphs	✓	? ✓	?	?	?	?	?
all with equal probability	✓	?	?	?	?	?	?

Table 4.1: Comparison the properties of the graphs by the creation algorithms.

5 Implementation

The discussed algorithms of chapter 3 and some of were implemented in order to verify the results. For that some of the hypergraph creation algorithms of chapter 4 were implemented as well.

5.1 Technologies

The focus of the implementation was less on performance optimization but on demonstrating feasibility. Therefore, Python 3 in combination with several libraries was used. For vector representation and operations, NumPy [8] proved useful and was therefore used. In order to optimize the semidefinite programming problem (SDP), the commonly used SciPy [9] was chosen over tools like cvxpy or cvxopt as their implementation proved problematic due to lack of information about them as they are less known.

For storing the results of the evaluation, Pickle was used. In order to plot the graphs, Matplotlib [10], in special PyPlot was utilized.

5.2 Code

For representing hypergraphs, a simple implementation tailored to the requirements was created in order to comfortably being able to implement the graph creation algorithms as well as the small expansion algorithms. Therefore, several classes were used to represent the graph.

In order to represent the vertices, the class *vertex* is used. As important attributes, it contains the set of edges it belongs to and the vertices' weight w_v for a vertex v (defined like in section 1.2). Except for its constructor, the method *add_to_edge* is used by the construction algorithms when a new edge, containing v, is added. Additionally, for the resampling in algorithm 15, the method *recompute_weights_degrees* is needed to update the attributes of the vertex after an edge is changed.

Edges are represented by the class *edge*, which contains a set of vertices and an attribute *weight* to represent the weight w_e of an edge e and the set of vertices.

A whole Graph H is encapsulated by the class *Graph*, which contains sets of the vertices as well as edges and as needed for the construction in algorithm 15 also a

algorithm	method
1	*brute_force_hypergraph_expansion*
2, 3	*brute_force_hypergraph_expansion_each_size*
	(has an option on how to calculate)
4	*generate_small_expansion_set*
5	*generate_small_discrepancy_ratio_vertex_vectors*
6	in *generate_small_discrepancy_ratio_vertex_vectors*
7	*generate_small_expansion_set*
8	*create_random_orthogonal_seperator*
9	*assign_words_to_vertices*
14	*create_connected_graph_random_edge_adding_resampling*
15	*create_connected_graph_low_degrees_shuffel_edges_until_connected*
16	*create_connected_graph_spanning_tree_low_degrees*

Table 5.1: Mapping the algorithms to the respective methods in the implementation.

set of the connection components of the graph. They are computed by the method *compute_connection_components*. This class also contains the algorithms of chapter 3 and 4 as described in table 5.1.

Connection components are represented by the class *ConnectionComponent*, which contains the set of vertices in that component.

Furthermore, to generate small expansions, a static Poisson process in positive as well as negative time is required. Therefore, the class *Poisson_Process* was created. The constructor takes λ and then calculates and holds the times when the events happened after as well as before the time 0. With the method *get_number_events_happened_until_t*, the number of events which happened between $t_0 = 0$ and the given t is returned. For convenient handling of the vectors f generated by algorithm 6, *vertex_vector* is used to access $f(v)$.

For evaluating the algorithms and in order to create the plots several scripts are collected in *evaluation.py*. After evaluating, the results are saved in a object of class *log*, which contains, among others, the graph, the smallest expansion found by brute-force and the small set expansion found by the approximation algorithm.

The implementation can be found on github. TODO

6 Evaluation

For the evaluation of the algorithms some parameters should be chosen in order to get reasonable results. The edge-weight distribution is always set to be a uniform distribution on $[0.1, 1.1]$. As long as not mentioned differently, $n = 20, r = 3, d = 3, k = 3$. The choices of the constants are justified in the respecting sections.

6.1 Input graph sizes

For evaluating the implementation of algorithm 4 in comparison to the brute-force solution, the maximal size of graphs (in n) which can be brute-forced in a reasonable time is determined. In general, it is favourable to generate as large as possible graphs in order to see how the algorithm performs.

However, the as the brute forcing time correlates with around $\mathcal{O}(2^n)$, with the available resources, for $n = 20$ brute-forcing already took around 18 s compared to 38 s for computing a small set expansion. As the brute-forcing time roughly doubles for each additional vertex, it is unfeasible to set n much higher. Even though 21 vertices would be possible, due to eq. (1.1) an uneven number would unnecessarily limit the possible combinations of ranks and degree as evaluated in fig. 6.2, as for example, there would be no integer value for m if n = 21, d= 3 and r = 6.

Aditionally for analyzing the constant in fact 2, a plot of as many graps as possible seems to be ideal. So not only for brute-forcing but also for executing the estimation algorithm, the graph shouldn't be too big. In conclusion, half a minute seems to be a reasonable time for one run of the algorithm. The times needed for different number of vertices can be seen in fig. 6.1.

Therefore n is set to be 20 for further analysis.

6.2 Rank degree combinations

As the graphs generated by the algorithms have uniform ranks and regular degrees, it needs to be determined which combination of r and d should be chosen. As already mentioned, they can not be chosen freely, not for all combinations r-uniform d-regular graphs exist on n vertices, e of equation eq. (1.1). As all of the variables need to be

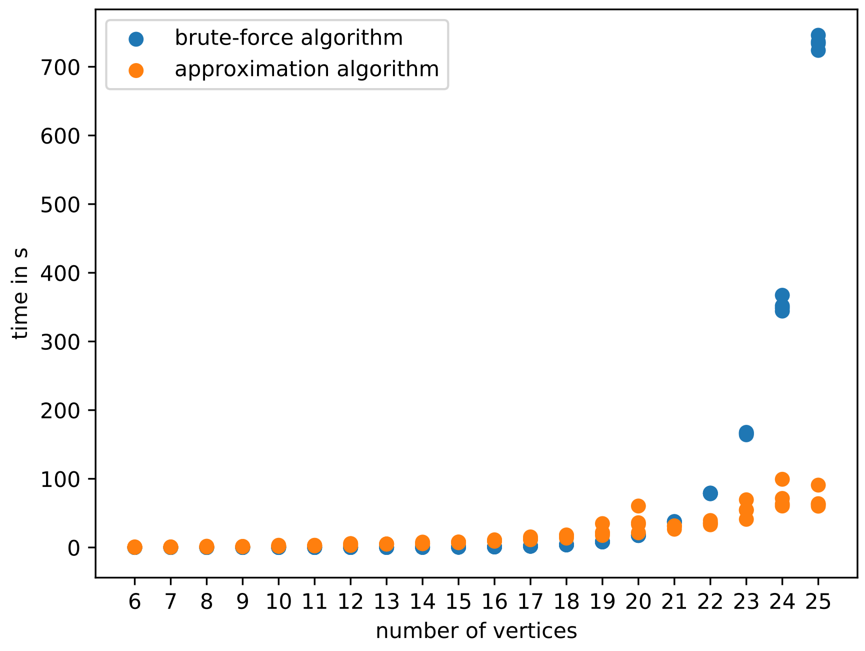

Figure 6.1: Plot of the number of vertices n against the time for computing solutions. It can be seen that the brute-force algorithm takes a long time for larger graphs, while the approximation algorithm's time only increases slowly.

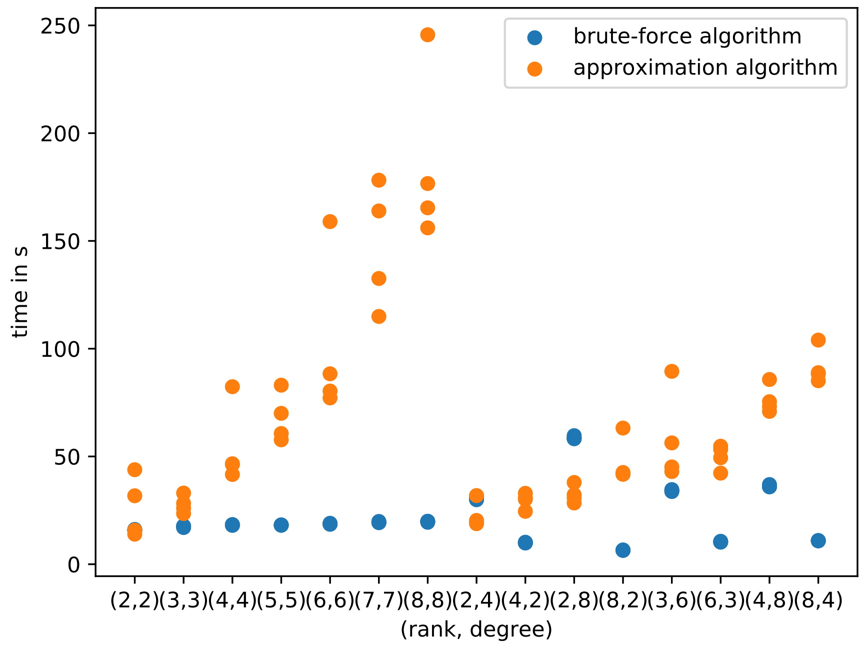

Figure 6.2: Plot of times for rank degree combinations.

non-negative integers, one can ensure to never violate that constraint for any n by setting $d = r$. An evaluation of the time constraints can be seen in fig. 6.2. Interestingly, for equal r and d, the brute-force algorithm's time did almost not increase when d and r were increased from (2,2) to (8,8). The small set approximation algorithm's times however, seem to increase linearly with r and d. For other combinations, the brute-force algorithm needed more time for comparatively high $\frac{d}{r}$ ratios.

For easier evaluation of the other properties, the rank and degree were chosen to be $r = 3$ and $d = 3$, as with a rank of 3 it is explicitly demonstrated that the algorithms work on hypergraphs.

6.3 Evaluation of k

The value of k plays an important role in the properties of the graph as well as the runtime. As described in algorithm 5, k vectors are constructed. As for k = 1, no SDP must be solved, the algorithm terminates quickly. However, this does not make sense anyways, as fact 2 does only hold for $k \geq 2$. As seen in fig. 6.3, for higher k, the time for

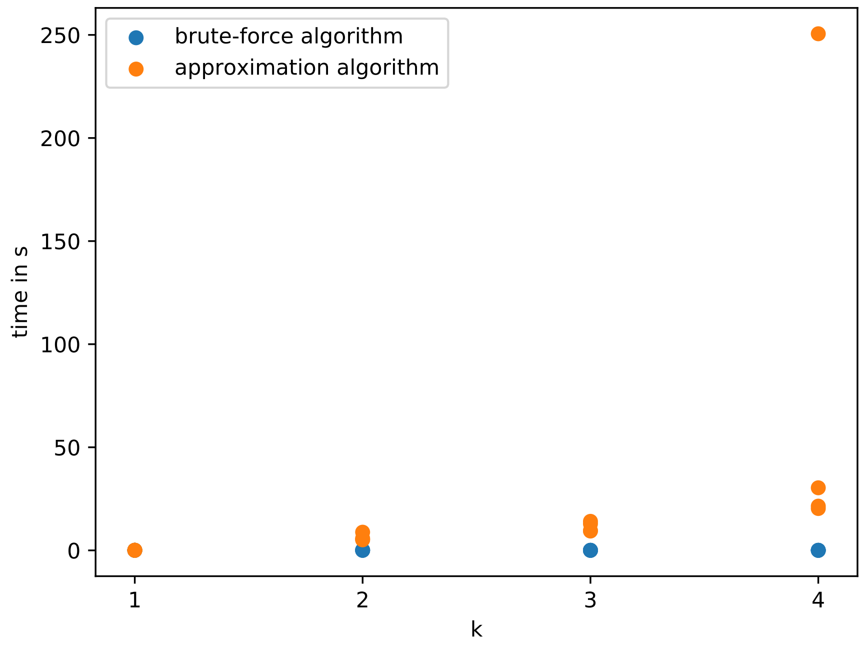

Figure 6.3: Plot of k against time.

each small expansion set approximation increases. Profiling the algorithm also showed that most of the time was spent on solving the SDP. Unfortunately for higher k, the optimization takes unreasonably long, presumably due to numerical instabilities.

6.4 Small expansion sizes

As seen in fig. 6.4, most of the sampled small expansion sets have a small number of vertices. TODO: verify: As k increases, this trend continues, going along the trend of getting smaller sets for larger k as indicated in fact 2.

Since higher values of k were not feasible in eq. (3.3) due to numerical issues of the implementation as well as time constraints, $|S| < \frac{24|V|}{k}$ can't be verified as k<5 here. C is desirable, as it would lead to small expansions.

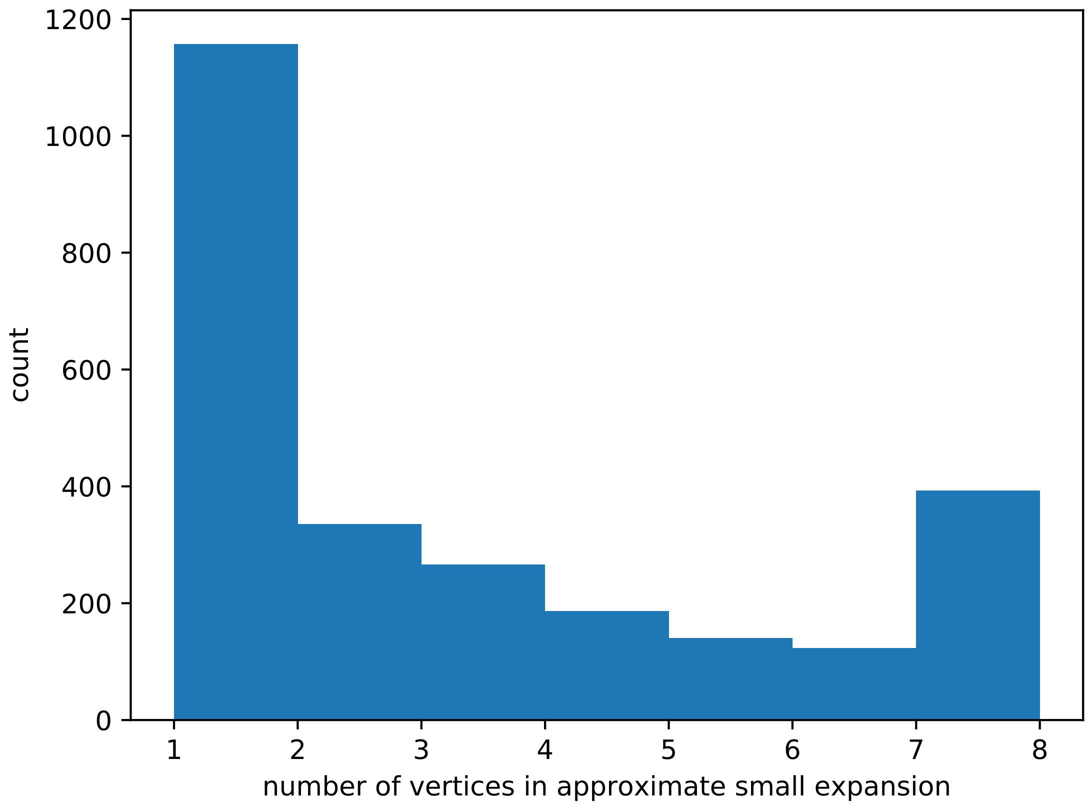

Figure 6.4: Plot of sizes of small expansions.

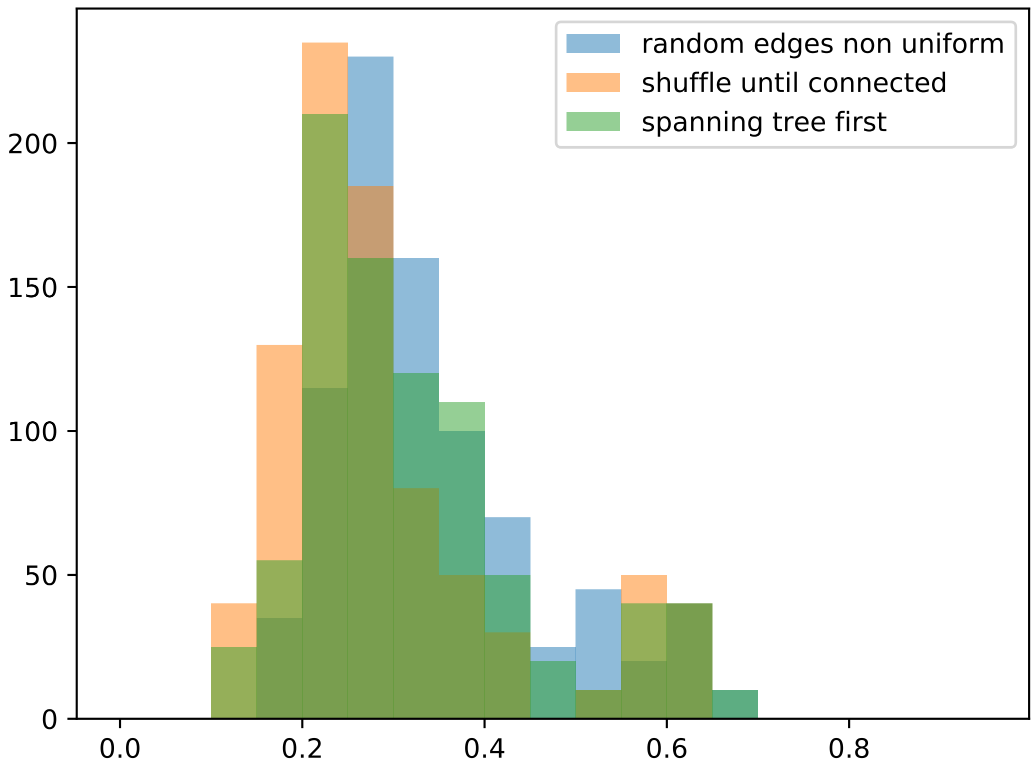

Figure 6.5: Plot of the lowest expansion values of the graphs created by different algorithms for each size of the expansion set. **??**

6.5 Random graphs comparison

As the expansion for non-connected graphs is always 0, only algorithms which guarantee to return connected graphs will be considered, namely 14, 15 and 16.

The expansion of the graphs is be evaluated against each other via brute-force. The results can be seen in **??**.

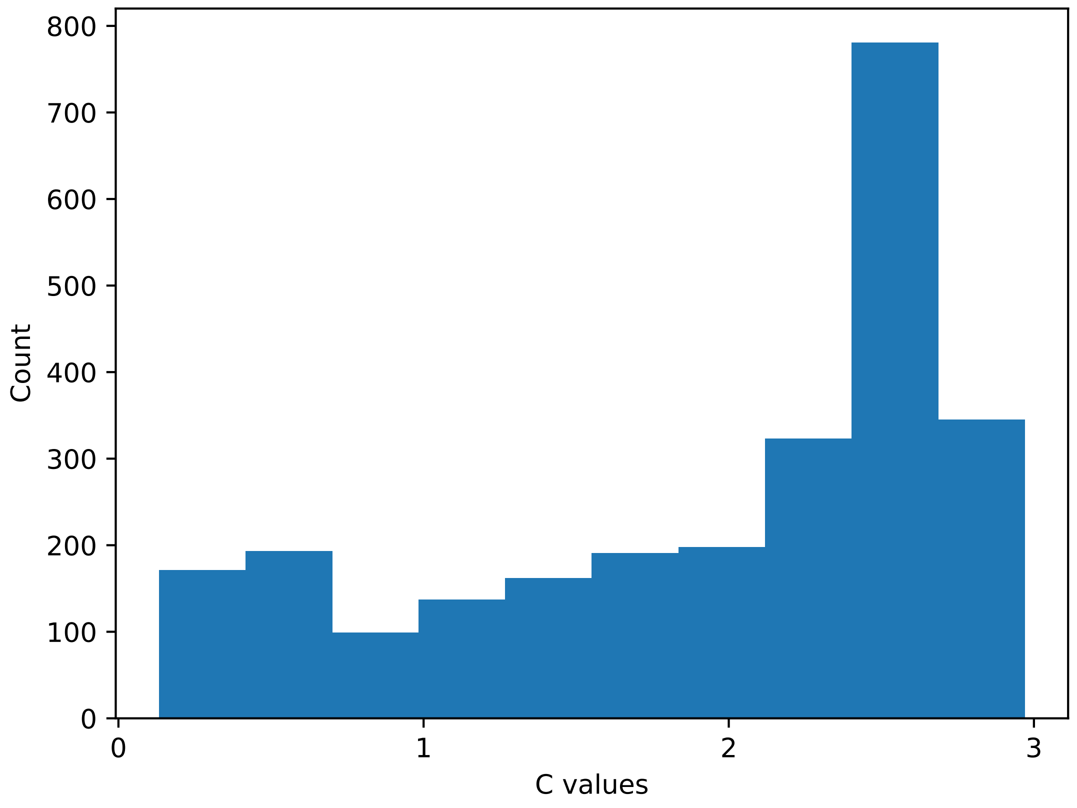

Figure 6.6: Plot of estimates for C

6.6 Estimation of C

As according to fact 2 a small value of C would be favourable to ensure small estimated expansions. For estimating C, the inequality of fact 2 can be changed to:

$$C \geq \frac{\phi(S)}{\min\{\sqrt{r \log k}, k \log k \log \log k \sqrt{\log r}\} \cdot \sqrt{\xi}} \tag{6.1}$$

log refers to the natural logarithm here.

However, fig. 6.6 does not show a clear picture. Since all the results were between ... and ..., one could hypothesize that C is at max .... TODO

6.7 Comparison of expansion values

Finally, the quality of the results is analyzed. As seen in fig. 6.7, the set with the lowest expansion value found by the brute force algorithm is, as expected, always at least as low as the value found by the approximation algorithm. Howeverm when comparing the best expansion values against the best one-percentile value as found by the brute-force algorithm, the approximation algorithm performs better ...% of times. However, the approximation algorithm

TODO

Therefore improvements

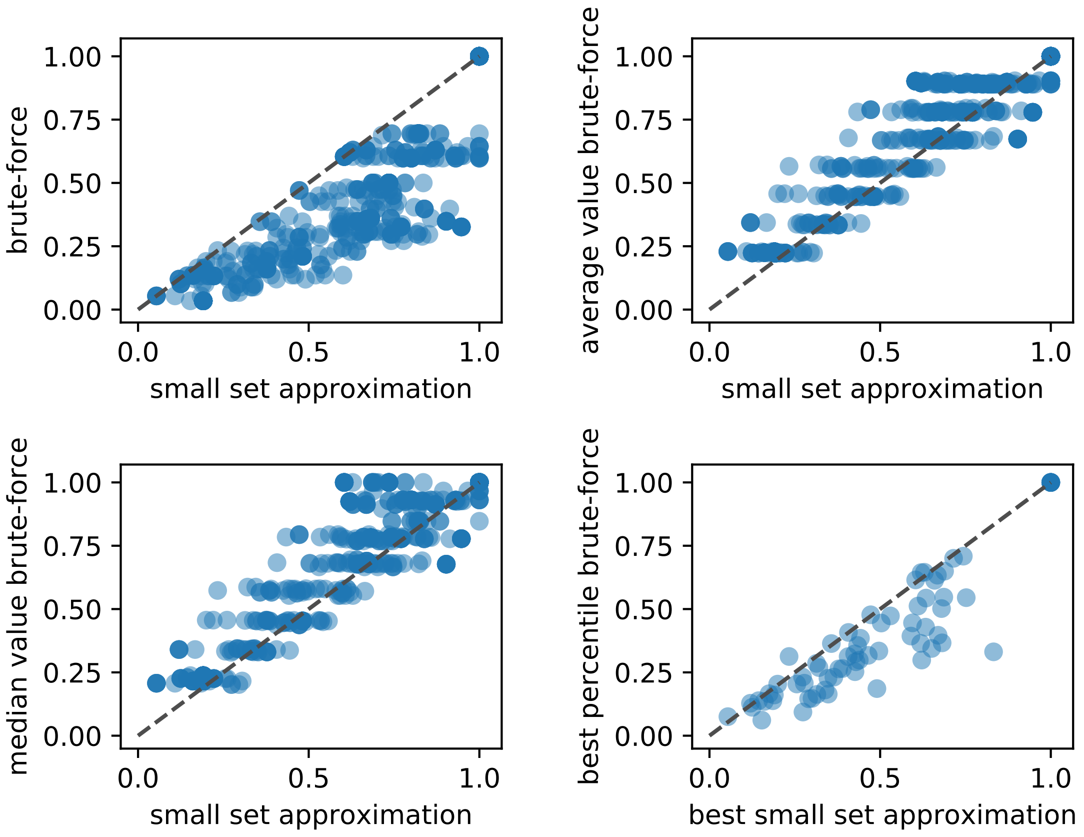

Figure 6.7: Plot of the expansion values achieved by the small expansion set approximation algorithm against the expansion set of the same size with the lowest expansion (as generated through the brute-force approach). Entries below the line signal that the expansion found by the approximation algorithm was worse than the set found by the brute force algorithm.

7 Applications

Hypergraphs can be used to model social networks in which users interact with each other. One could represent the users as vertices and the hyperedges as interactions between users. If for example ten users discuss a post with each other, they would receive an edge with a weight depending on the intensity of the interaction. If for example, interest lies in finding a group of close friends, which interact mostly and most intensely with each other, one can use the approximation algorithm, algorithm 4 in order to find such a group. The algorithm would be called with a k adjusted to the total number of users and the favoured number of users. Further discussion of applications like these can be found in[7].

For further applications of random hypergraphs, the reader is referred to [6].

One other possible application of the discussed algorithm might be to solve puzzle games like Rummikub[1]. There the stones would be the vertices in a hypergraph and the edges and their weights would indicate how well specific stones can be combined with another. As only some of the stones are visible usually, in order to win the game it is required to find a new combination which can take in some of the private stones of each player. A small expansion set could be a good heuristics for starting the search for new combinations. Admittedly, the possible actions in the game are limited but results might be also used in other matching-like problems.

[1]https://en.wikipedia.org/wiki/Rummikub

8 Resume and Further Work

Several aspects for further work can be found. For one, the efficiency of the implementation can be improved. As the bottleneck of the implementation is the SDP optimization, biggest improvements can be achieved there. One could try adjusting the options for the current optimizer, try different optimizers altogether and as well speed up the calculation of the $SDPvalue$ and the constraints. As of now, for better overview, these calculations rerquire accessing different Python-dictionaries, which could be handled more efficiently. Another aspect for the future is evaluating the algorithms on a larger scale with a more powerful computer and more time. Not only the n, r, d and k can be increased, but also the number of repetitions per graph. As a result, this would give a more precise picture about the properties of the algorithms, especially the value of the constant C of eq. (6.1). Furthermore, even more combinations of ranks r and degrees d, can be evaluated. Also, for more insight the algorithms could be evaluated on, non-uniform graphs.

The biggest challenge in this thesis were the extraction and understanding of the estimation algorithm and the algorithms it depends on. Furthermore, developing algorithms for the creation of random graphs with specific properties proved tricky. During implementation especially finding a suitable optimizer for the SDP and finding the right parameters for it were challenging.

All in all, small set approximations showed to be an interesting topic from a theoretical perspective, but they also have applications in the real world. Especially finding an algorithm for creating a random hypergraph proved appetizing.

List of Figures

List of Tables

Bibliography

[1] M. Stoer and F. Wagner, "A simple min-cut algorithm," *Journal of the ACM (JACM)*, vol. 44, no. 4, pp. 585–591, 1997.

[2] R. M. Karp, "Reducibility among combinatorial problems," in *Complexity of computer computations*, Springer, 1972, pp. 85–103.

[3] V. Kaibel, "On the expansion of graphs of 0/1-polytopes," in *The Sharpest Cut: The Impact of Manfred Padberg and His Work*, SIAM, 2004, pp. 199–216.

[4] T. H. Chan, A. Louis, Z. G. Tang, and C. Zhang, "Spectral properties of hypergraph laplacian and approximation algorithms," *CoRR*, vol. abs/1605.01483, 2016. arXiv: 1605.01483.

[5] A. Louis and Y. Makarychev, "Approximation Algorithms for Hypergraph Small Set Expansion and Small Set Vertex Expansion," in *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques (APPROX/RANDOM 2014)*, K. Jansen, J. D. P. Rolim, N. R. Devanur, and C. Moore, Eds., ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 28, Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2014, pp. 339–355, ISBN: 978-3-939897-74-3. DOI: 10.4230/LIPIcs.APPROX-RANDOM.2014.339.

[6] G. Ghoshal, V. Zlatić, G. Caldarelli, and M. E. Newman, "Random hypergraphs and their applications," *Physical Review E*, vol. 79, no. 6, p. 066 118, 2009.

[7] Z.-K. Zhang and C. Liu, "A hypergraph model of social tagging networks," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2010, no. 10, P10005, 2010.

[8] T. Oliphant, *NumPy: A guide to NumPy*, USA: Trelgol Publishing, [Online; accessed March 2019], 2006.

[9] E. Jones, T. Oliphant, P. Peterson, *et al.*, *SciPy: Open source scientific tools for Python*, [Online; accessed March 2019], 2001.

[10] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing In Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007. DOI: 10.1109/MCSE.2007.55.