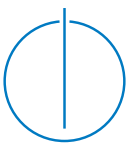
DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Spectral Methods to Find Small Expansion Sets on Hypergraphs

Franz Rieger



DEPARTMENT OF INFORMATICS

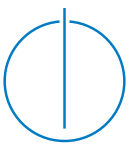
TECHNICAL UNIVERSITY OF MUNICH

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. Dr. rer. nat. 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 sets with low edge expansion on a graph can also be defined on hypergraphs. 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 the results are evaluated on random hypergraphs. For the creation of the hypergraphs, different algorithms are presented. Especially the approximation algorithm can be useful for finding a set of friends in a social network, which is represented by a hypergraph.

Contents

1 Introduction

This thesis revolves around finding small expansion sets on random hypergraphs, especially with an approximation algorithm derived from the findings in [4]. Additionally, for evaluating the implementation of the algorithm, a discussion on how to create random hypergraphs is involved.

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

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.

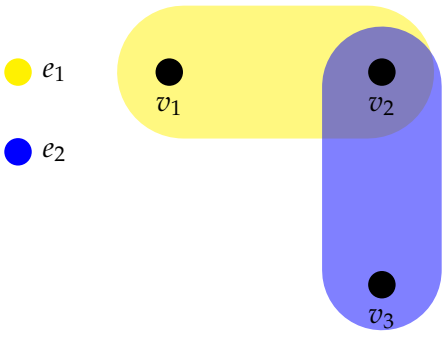


Figure 1.1: An example for a simple graph with three vertices and two edges.
$G = (\{v_1, v_2, v_3\}, \{\{v_1, v_2\}, \{v_2, v_3\}\})$

1.2 Hypergraphs

This thesis deals with a generalized form of 2-graphs, namely (hyper)graphs. 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 $w : E \to \mathbb{R}_+$. An example for a hypergraph can be seen in fig. 1.2.

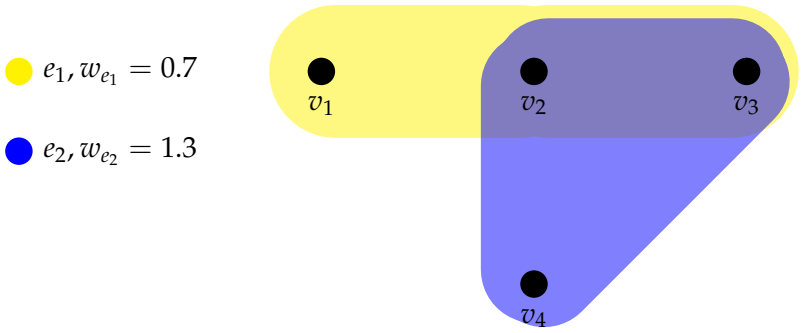


Figure 1.2: An example for a simple hypergraph with four vertices and two hyperedges. $G = (\{v_1, v_2, v_3, v_4\}, \{\{v_1, v_2, v_3\}, \{v_2, v_3, v_4\}\})$

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. Hence a 2-graph like defined in section 1.1 is a 2-uniform hypergraph. 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 n vertices and the m edges 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 perspective of the vertices by summing up the degrees of all the vertices (which equals to nd for regular graphs). Bu the count of connections can also be considered 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.

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. $\forall i \in [k-1] \exists e \in E : u_i, u_{i+1} \in e$. Hence, a path exists if it is possible to reach v_k by hopping from one vertex to another by moving along the edges, starting at v_1. 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. Thus, within a connected component it is possible to reach every vertex from every other vertex. If $S = V$, the whole graph consists of only one connected component, ergo it is called connected.

1.3 Cuts

On such hypergraphs certain properties can be observed, 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 defined by its cut-set $\emptyset \neq S \subsetneq V$, a non-empty strict subset of the vertices V. 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 going between the vertices of set S and all the remaining vertices $V \setminus S$. For weighted graphs, instead of the number of edges, their added weight is considered. 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 referred to as expansion) of a graph, which is the quotient of the summed weight of the edges crossing S and $V \setminus S$ and the minimum of the summed weight of all the vertices in S or $V \setminus S$. The formal notation which is introduced in the following and in chapter 2 is found on the article, on which the crucial approximation algorithm of this thesis is based [4].

The set of edges which are cut by a subset of vertices S contains all the edges, which contain at least one vertex in the set S and at least one other 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_v of a vertex v is defined by summing up the weights of its edges:

$$w_v := \sum_{e \in E : v \in e} w_e. \tag{1.4}$$

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

$$w(S) := \sum_{v \in S} w_v \tag{1.5}$$

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

$$w(F) := \sum_{e \in F} w_e \tag{1.6}$$

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

$$\Phi(S) := \frac{w(\partial S)}{w(S)}. \tag{1.7}$$

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

$$\forall \emptyset \neq S \subseteq V : 0 \leq \Phi(S) \leq 1 \tag{1.8}$$

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

$$\Phi(H) := \min_{\emptyset \subsetneq S \subsetneq V} \max\{\Phi(S), \Phi(V \setminus S)\}. \tag{1.9}$$

Here again, $0 \leq \Phi(H) \leq 1$ holds because of eq. (1.8).

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 only the vertices of one connection component, as the cut would contain no edges. For an example refer to fig. 1.3. As this thesis focuses on finding sets S with a low expansion value $\Phi(S)$, graphs with expansion 0 are a trivial special case. Therefore, only connected graphs shall be of interest here.

Note that for a graph H, which is obtained by connecting two connection components with an edge with a relatively 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, where every edge-weight takes the same value, ∂S is 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

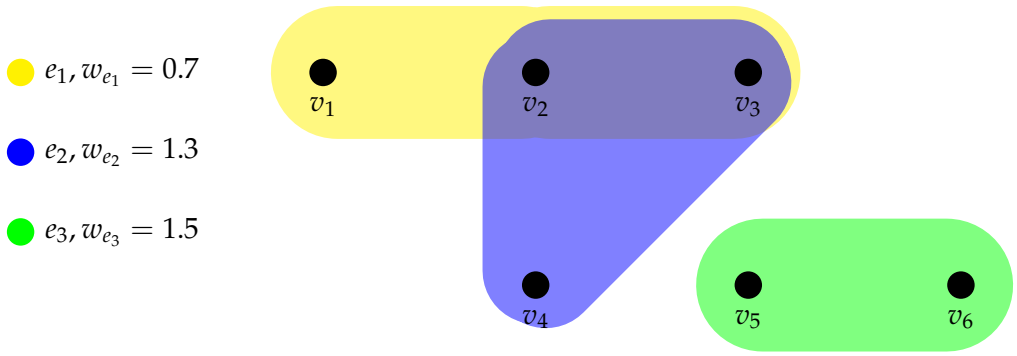


Figure 1.3: An example for a non-connected hypergraph with two connection components. For $S := \{v_5, v_6\}$ it can be verified that $\delta S = 0$, hence $\Phi(S) = \Phi(V \setminus S) = 0$.

problem, one of them will be focused on here: For certain applications like finding a group of friends in a social network, 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 most 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, the required 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 various creation methods are 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 elaborates on possible future work and completes the thesis.

2 Notation

For further use, especially for the upcoming theorems and the subroutines of algorithm 4, some notation of spectral properties of graphs shall be introduced.

In the algorithms, values are assigned to each vertex $v \in V$ in the form of vectors $f, g \in \mathbb{R}^V$.

In the so-called weighted space, 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 used weight matrix W of a hypergraph, which contains the vertices' weights on its diagonal, can be denoted as

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

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

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

Observe that $0 \leq D_w(f) \leq 2$ [4]. For a vector f where all entries take the same value it can be seen that $D_w(f) = 0$.

The discrepancy ratio important is because it is connected to the edge expansion $\Phi(S)$ of a set S. This can be seen by choosing f to be the indicator vector for the set, so

$$f_v = \begin{cases} 1, & \text{if } v \in S \\ 0, & \text{otherwise} \end{cases}. \tag{2.3}$$

Then, the nominator of $D_w(f)$ would sum over all the edges in δS, because if and only if one vertex of an edge is in S and the other one is not in S, the expression $(f_u - f_v)^2$ would be 1, (it is 0 otherwise). The denominator however, already computes $w(S)$, which makes the discrepancy ratio equal to the expansion $\Phi(S)$. Therefore, computing vectors with a low discrepancy ratio can help in finding an expansion set with a low expansion value.

It shall be seen in algorithm 7 that one can create a small expansion set if several orthogonal vectors whose maximal discrepancy ratio is low are found. The lowest value which can be achieved is described by a so-called orthogonal minimaximizer which is defined as follows for k mutually orthogonal non-zero vectors of the weighted space:

$$\xi_k := \min_{0 \neq f_1,\ldots,f_k; f_i \perp f_j} \max_{i \in [k]} D_w(f_i) \tag{2.4}$$

However, there is no efficient way of computing the value (and the corresponding vectors) known. Therefore, an approximation is given with algorithm 5.

3 Expansion Algorithms

In this chapter, different approaches for generating expansion sets S with a low edge expansion are discussed. First, brute-force algorithms for finding the sets with the lowest expansion for the whole graph, for just a set, and also for sets of all possible sizes are shown. Following, a more efficient approximation algorithm, which is based on the results of [4], for finding small expansion sets is presented.

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))
    bestS := null
    lowestExpansion := ∞
    for ∅ ≠ S ⊊ V do
        expansion := max{Φ(S), Φ(V \ S)}
        if expansion < lowestExpansion then
            lowestExpansion := expansion
            bestS := S
    return bestS
```

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

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

However, 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 tractable 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 examine the lowest expansion value of each possible size of the expansion set like in algorithm 2.

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

function BRUTEFORCEEDGEEXPANSIONSIZESGRAPH($H := (V, E, w)$)
  $bestSofSize := \{\}$
  $lowestExpansionOfSize := \{1 : \infty, 2 : \infty, \ldots, n-1 : \infty\}$
  for $\emptyset \neq S \subsetneq V$ **do**
    $expansion := \max\{\Phi(S), \Phi(V \setminus S)\}$
    if $expansion < lowestExpansionOfSize[|S|]$ **then**
      $lowestExpansionOfSize[|S|] := expansion$
      $bestSofSize[|S|] := S$
  return $bestSofSize$

In order to analyze the results from the approximation 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 computation of the expansion in algorithm 2 needs to be changed to get algorithm 3.

Algorithm 3 Brute-force expansion of sets for every size

function BRUTEFORCEEDGEEXPANSIONSIZESSETS($H := (V, E, w)$)
  $bestSofSize := \{\}$
  $lowestExpansionOfSize := \{1 : \infty, 2 : \infty, \ldots, n-1 : \infty\}$
  for $\emptyset \neq S \subsetneq V$ **do**
    $expansion := \Phi(S)$
    if $expansion < lowestExpansionOfSize[|S|]$ **then**
      $lowestExpansionOfSize[|S|] := expansion$
      $bestSofSize[|S|] := S$
  return $bestSofSize$

For algorithm 2 and algorithm 3 the above argument for exponential time complexity holds as well. Therefore, a more efficient approximation shall be given in the following section.

3.2 Approximation of small expansion sets

As described in [4], an algorithm for generating a random small expansion set can be derived by first creating a set of $k \geq 2$ non-zero vectors $f_1, \ldots, f_k$, which are orthogonal in the weighted space and have a low maximal discrepancy ratio $\xi = max_{f_1,\ldots,f_k} D_w(f_k)$. Then, the set of vectors can be inputted to algorithm 7 which is algorithm 1 in [4]. There, the vectors are used to create an set of vertices which are then returned. The combination of these steps results in algorithm 4, which takes a graph H and an integer $k \geq 2$ and returns a small expansion set. In the following, the subroutines of this algorithm shall be elaborated.

Algorithm 4 Find Small Expansion Set

function SMALLEXPANSIONSET(H, k)
  $f_1 \ldots, f_k$:=SAMPLESMALLVECTORS(H, k)
  return SMALLSETEXPANSION($H, f_1 \ldots, f_k$)

3.2.1 Creation of orthogonal vectors with low discrepancy ratio

As there is no known, efficient way of achieving the optimal value ξ_k, an approximation is given through solving a semidefinite programming (SDP) problem in the proof of theorem 8.1 in [4], which results in algorithm 5. This algorithm is called first and generates one vector f_i after another by repeatedly solving the SDP. It returns a set of non-zero orthonormal vectors $\{f_1, \ldots, f_k\}$, where each vector $f_i \in \mathbb{R}^V$ gives a value to each vertex. Because of theorem 2 it is of importance for algorithm 7 that the maximal discrepancy ratio $\xi := \max_{s \in [k]} D_w(f_s)$ is small, as explained before, the value of the discrepancy ratio is correlated with the value of the resulting expansion. At first, f_1 is set to be $\frac{\vec{1}}{||\vec{1}||_w}$, as this results in a minimal $D_w(f_1) = 0$. Following that, the other vectors $f_2, \ldots, f_k$ are sampled after another, using algorithm 6. This sampling of vectors after another where the discrepancy ratio of the next vector shall be minimal, given that it has to be orthogonal to the already constructed vectors, is also referred to as procedural minimizer, hence the name of the algorithm. In this way an approximation for the ideal vectors, which would achieve ξ_k as defined in eq. (2.4), is performed according to theorem 1.

Theorem 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 $i \in [k]$,*

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

Algorithm 5 Procedural Minimizer

function SAMPLESMALLVECTORS(H, k)
  $f_{:}1 = \frac{\vec{1}}{||\vec{1}||_w}$
  for $i = 2, \ldots, k$ **do**
    f_i :=SAMPLERANDOMVECTOR($H, f_1, \ldots, f_{i-1}$)
    $f_i = \frac{f_i}{||f_i||_w}$
  return $f_1, \ldots, f_k$

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 $k-1$ constraints ensure orthonormality to the already existing vectors. By sampling a vector from a gaussian normal distribution and multiplying it with all the $\vec{g_v}$, the coordinates of f are created. According to theorem 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.

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 $v \in V$
  $\vec{z} := sample(\mathcal{N}(0, I_n))$ ▷ random gaussian vector
  for $v \in V$ **do**
    $f(v) := \langle \vec{g_v}, \vec{z} \rangle$
  return f

3.2.2 Calculating a small expansion set

After the vectors $f_1, \ldots, f_k$ have been sampled, algorithm 4 calls algorithm 7. There, the vectors $f_1, \ldots, f_k$ are transformed to form u_v for $v \in V$. 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 $\tilde{u}_v$ which represents the candidate vertices for the expansion. With this subset, a vector X is constructed, which for each vertex represents a value, indicating a sort of priority of that vertex belonging to the next expansion set. X_v takes the value $||u_v||$ if $\tilde{u}_v \in \hat{S}$ and 0 otherwise. Then, X is sorted in decreasing order and all the prefixes of X are analyzed for the expansion value of their respecting vertices. The set of vertices S with the lowest expansion is returned. According to theorem 2, there is an upper bound on the expansion value of the expansion set, which this algorithm returns. Therefore, with the right input vectors, the algorithm can create useful results. Here again, in the implementation, this algorithm is repeated several times as its results are based on randomness and compared to the optimization of the SDP, the calculation is inexpensive and only takes a small amount of time. In that way, the best result of many, i.e. the one with the lowest expansion, can be returned.

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

function SmallSetExpansion($G := (V, E, w), f_1, \ldots, f_k$)
  assert $\xi == \max_{s \in [k]} \{D_w(f_s)\}$
  assert $\forall f_i, f_j \in \{f_1, \ldots, f_k\} \subset \mathbb{R}^n, i \neq j : f_i$ 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 $i \in V$ **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

Theorem 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 $r := \max_{e\in E} |e|$.

In algorithm 8 a number of $l := \lceil \frac{\log_2 k}{1-\log_2(1+\frac{2}{log_2 k})} \rceil$ assignments are sampled for each vertex $u \in V$ with the help of algorithm 9, which assigns a value $w_j(u) \in \{0, 1\}$ to each vertex u for $j \in [l]$. With that, for each vertex u, a word $W(u) = w_1(u)w_2(u)\cdots w_l(u)$ can be defined out of the assignments. Then, a random *word* is picked, depending on whether $n \geq 2^l$, either from $\{0, 1\}^l$ or from all the constructed words with the same probability. In the latter case, $|V| - |W|$ new, distinct words are constructed, if there are duplicates within the words $W(u)$. $|W|$ refers to the number of unique words in W. The constructed words are 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. In the implementation, if a word, is chosen that does not belong to any vertex, the process of sampling a word is repeated, as an empty S would not be useful.

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, which does not create a problem since in the Poisson process events happen continuously and t_0 can be set on any possible event, which naturally has predecessors. Additionally, a vector $g \in \mathbb{R}^k$ is created, where each component is sampled independently from a standard normal distribution $\mathcal{N}(0, 1)$. Then, for each vertex v and for $i = 1, 2, \ldots, l$ a 'time' $t = \langle g, \tilde{u}_v \rangle$ is calculated. Depending on whether the number of events that happened in the Poisson process between t and t_0, is even or odd, $w_i(v)$ is set to 0 or 1. Finally, w is returned.

This is the crucial point of the whole algorithm: For each vector $v \in V$, a $\tilde{u}_v$ was created, whose components $\tilde{u}_{v,1}, \ldots, \tilde{u}_{v,k}$ represent the entries of the orthogonal vectors with low maximal discrepancy ratio ξ of algorithm 5. Therefore, vertices $a, b \in V$ whose f-values are similar, i.e. $f_i(a) \approx f_i(b)$, also receive a similar time through the inner product, so $t_a = \langle g, \tilde{u}_a \rangle \approx \langle g, \tilde{u}_b \rangle = t_b$. Therefore, it is likely that they also receive the same assignment as in the Poisson process, it is unlikely for an event to happen in a

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

function OrthogonalSeparator($\{\tilde{u}_v\}_{v\in V}, \beta = \frac{99}{100}, \tau = k$)
  $l := \lceil \frac{\log_2 k}{1-\log_2(1+\frac{2}{log_2 k})} \rceil$
  $w :=$SampleAssignments($\{\tilde{u}_v\}_{v\in V}, l, V, \beta$)
  for $v \in V$ **do**
    $W(v) := w_1(v)w_2(v)\cdots w_l(v)$
  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($\{\tilde{u}_v\}_{v\in V}, l, V, \beta$)
  $\lambda := \frac{1}{\sqrt{\beta}}$
  $k := |\tilde{u}|$ ▷ number of entries for each vertex
  $g :=$ sample($\mathcal{N}(0, I_k)$) ▷ all components g_i are mutually independent
  $poisson\_process := N(\lambda)$ ▷ N is a Poisson process on $\mathbb{R}$ with rate λ
  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

short difference of time. So, with high probability, $\forall i \in [k] : w_i(a) = w_i(b)$, which means they get the same word. As only one word gets selected in algorithm 8, that means that those vertices (or, to be precise, their $\tilde{u}$-values) are returned together to algorithm 7 if their word is selected. There, the sorting ensures that not all the combinations of the returned vertices need to be checked, which would be inefficient, as just checking the prefixes is sufficient. The fact that vectors $f_1, \ldots, f_k$ with low discrepancy ratios show this characteristic is due to the spectral properties of hypergraphs, which are discussed in [4]. There, a Laplacian operator for hypergraphs is defined and the discrepancy ratio as well as the edge expansion are connected to the eigenvalues of said Laplacian.

It can also be seen why the number of vertices in the expansion set returned by algorithm 7 decreases with a higher k according to theorem 2: As k increases, l also increases. Therefore, the words get longer, and it is more likely for two vertices to have a letter in their word which differs, making it less likely they are sampled together.

4 Random hypergraph generation

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

The initial aim 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, where 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 by sampling from all graphs

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

The naive 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 vertices are distinguishable). This makes the algorithm impracticable.

4.2 Adding random edges

Therefore, a simple algorithm to generate graphs which follows some of the intentions will be elaborated in the following. Algorithm 11 simply samples edges by repeatedly

Algorithm 11 Generate by adding random edges

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

randomly choosing r vertices of V. This makes the algorithm guaranteed to terminate, as it contains no conditional loops. As all the operations, especially the sampling can be performed in polynomial time complexity and no conditional loops or recursive calls are 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 there is a $H = (V, E, w)$ which 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. Additionally, the algorithm does not guarantee to have no doubled edges. No statement can be made on whether the probability that one graph is constructed is equal to any other graph, as for that a more specific graph model would be required. In discussion of the other algorithms, this question shall be disregarded. The reader may be referred to [6] for further reference.

Algorithm 12 Generate random graph with upper bound on degrees

function GENERATERANDOMGRAPHBOUNDDEGREES($n, r, d, weightDistribution$)
  $E := \emptyset$
  $V := \{v_1, \ldots, v_n\}$
  $w = \{\}$
  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 $H = (V, E, w)$

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 repeatedly samples edges as long as there are at least r vertices left which have a degree lower than d. It is again guaranteed to terminate, and the resulting graph is r-uniform, all possible graphs can be constructed and there is no guarantee for the graph to have unique edges, following a similar argumentation to algorithm 11. Also, 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 and a maximum degree 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 (at most r) vertices do not have degree d in the end, because they have not been sampled before, but there are not at least $r - 1$ other vertices left, which do not have degree d yet and would accept another edge. An example of such a situation can be seen in fig. 4.1.

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 as needed for a full edge are sampled from the vertices, which have the next higher degree. Therefore, the degrees of the vertices are increased step by step and there is an upper bound on the difference in degrees $|deg(v) - deg(u)| <= 1$. Therefore, it is not possible for some vertices to have

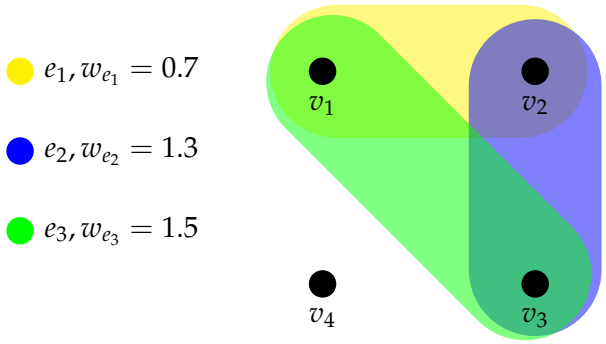


Figure 4.1: An example for a non-connected 2-uniform hypergraph which could have been created by algorithm 12.

Algorithm 13 Generate random hypergraph, sampling from lowest degrees

function GENERATESAMPLESMALLESTDEGREES($n, r, d, weightDistribution$)
  $E := \emptyset$
  $V := \{v_1, \ldots, v_n\}$
  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 maximum degree already, while others do not have any edges yet, which might make it impossible for them to connect.

The algorithm is guaranteed to terminate, and of polynomial time complexity for 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, it is not guaranteed that all the possible graphs can be constructed: This algorithm basically constructs the edges by d separate r-matchings, where a matching is a set of edges which encompasses each vertex in the graph exactly once. But not every graph can be dissembled into d r-matchings, as seen in the counterexample of fig. 4.2, which can not be constructed by algorithm 13. But since the graph in the counterexample is not connected, which would be desired, it can not be excluded at this point that all the possible graphs within the requirements can be constructed by algorithms like this one.

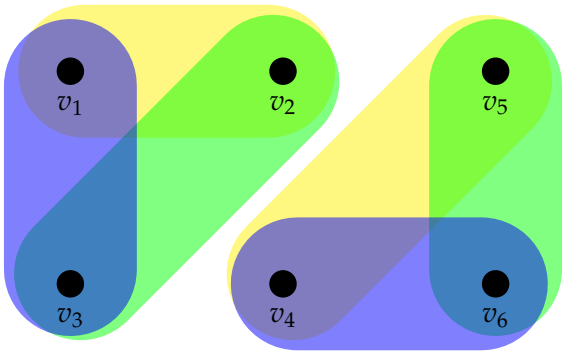


Figure 4.2: An example for a 2-regular 2-uniform hypergraph which can not be constructed by algorithm 13.

Again, this graph is not necessarily connected, and some edges might be doubled, as it proves challenging to avoid the following situation: Assume a graph is being generated, 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 arise to solve this problem could 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 $\binom{n}{r}$, the number of all possible first edges. 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. These options shall be discussed in the following.

4.5 Resampling whole graph until connected

Algorithm 14 resamples the whole graph, if the graph constructed by algorithm 11 is not connected. This way, the algorithm loses the property of guaranteed terminating. The expected runtime would depend on the probability of creating a graph which fulfills the requirements. However, this shall not be analyzed here.

This algorithm can be extended by checking for more properties like containing no doubled edges or 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 drastically, which is also not analyzed here.

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 constructed 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 out of those one vertex each, such that $u \in e, v \in f$. Then, if the vertices do not belong to the same connected 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 connected component, it shall be ensured that the number of connected components does not increase, as there are some situations where this operation can split a connected component into two separate components.

As the graph initially created by algorithm 13 is d-regular, 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 $connected\_component(u) \neq connected\_component(v)$ **then**
      $e := (e \cup \{v\}) \setminus \{u\}$
      $f := (f \cup \{u\}) \setminus \{v\}$
  return $G := (V, E, w)$

4.7 Creation of spanning tree

One other approach for generating connected graphs is to ensure the graph is connected in the beginning by first creating a tree like algorithm 16 does. Therefore, at first, the main connection component is created by sampling a random edge. Then, until the graph is connected, for every new edge one vertex v of degree $deg(v) = 1$ from the main connection component is sampled along with $r - 1$ non-connected vertices. In case less than $r - 1$ non-connected vertices remain, all of them will be combined to one edge and the remaining vertices are chosen from those with degree 1. 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 analogously to algorithm 13.

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 and extended in other ways as indicated before. Therefore, it is important to note that this study of creation algorithms is not exhaustive. More sophisticated random graph models are discussed in [6], [7]. As only algorithms 14, 15 and 16 can be implemented efficiently and also guarantee to produce connected graphs, which are required for evaluating the algorithms of chapter 3, only they were implemented. The details of the implementation are discussed in the next chapter.

Algorithm 16 Generate random graph by creating a spanning tree

function GENERATEWITHSPANNINGTREE($n, r, d, weightDistribution$)
$V := \{v_1, \ldots, v_n\}$
$w = \{\}$
$firstEdge := choice(V, r)$
$w(firstEdge) = sample(weightDistribution)$
$E := \{firstEdge\}$
while $\{v \in V | deg(v) = 0\} \neq \emptyset$ **do** ▷ create tree
if $|\{v \in V | deg(v) = 0\}| \geq r - 1$ **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
$nextEdgeVertices := \{v \in V | deg(v) = 0\} \cup$
$choice(\{v \in V | deg(v) = 1\}, r - |\{v \in V | deg(v) = 0\}|)$
$nextEdge := nextEdgeVertices$
$E := E \cup \{nextEdge\}$
$w(nextEdge) := sample(weightDistribution)$
while $|\{v \in V | deg(v) < d\}| \geq r$ **do** ▷ fill up degrees
$smallestDegreeVertices := \{v \in V | deg(v) = \min_{u \in V} deg(u)\}$
if $|smallestDegreeVertices| >= r$ **then**
$nextEdgeVertices := sample(smallestDegreeVertices, r)$
▷ draw without replacement
else
$secondSmallestDegreeVertices := \{v \in V | deg(v) = \min_{u \in V} deg(u) + 1\}$
$nextEdgeVertices :=$
$sample(secondSmallestDegreeVertices, r - |smallestDegreeVertices|)$
$nextEdgeVertices := smallestDegreeVertices \cup nextEdgeVertices$
$nextEdge := nextEdgeVertices$
$E := E \cup \{nextEdge\}$
$w(nextEdge) := sample(weightDistribution)$
return $G := (V, E, w)$

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

Table 4.1: Comparison of the properties the creation algorithms and their graphs.

5 Implementation

The discussed algorithms of chapter 3 were implemented in order to analyze the performance. In order to have graphs on which the performance can be verified, some of the hypergraph creation algorithms of chapter 4 were implemented as well. This chapter shall give an overview over the used technologies and the structure of the code.

5.1 Technologies

The focus of the implementation is less on fast execution time but on demonstrating feasibility of the algorithms. Therefore, Python 3 in combination with several libraries is utilized. For vector representation and operations, NumPy [8] proved useful and is therefore used. In order to optimize the semidefinite programming problem sDP 1, the commonly used SciPy [9] was chosen over less commonly used tools like cvxpy or cvxopt as an implementation using them proved problematic due to lack of information resources about them. For storing the results of the evaluation on graphs, Pickle is employed. In order to plot the graphs, Matplotlib [10], in special PyPlot was utilized.

5.2 Code structure

For representing hypergraphs, a simple implementation tailored to the requirements was created in order to comfortably being able to implement the graph creation algorithms of chapter 4 as well as the small expansion algorithms of chapter 3. Therefore, several classes are used to represent the whole 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. 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.

algorithm	method
1	*brute_force_hypergraph_expansion*
2, 3	*brute_force_hypergraph_expansion_each_size* (has an option on how to calculate the expansion)
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.

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

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 set of the connection components, which 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.

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 of the events after as well as before the time $t_0 = 0$. With the method *get_number_events_happened_until_t*, the number of events which happened between t_0 and the given t is returned. For convenient handling of the vectors f generated by algorithm 6, the class *vertex_vector* is used to access $f(v)$.

In order to evaluate the performance of the various algorithms and to create the plots, several scripts are collected in *evaluation.py*. After evaluating, the results are saved in a object of class *Graph_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 https://github.com/riegerfr/Bachelor-s-thesis-edge-expansion/tree/master/hypergraph-implementation.

6 Evaluation

For the evaluation of the algorithms, some parameters need to be chosen in order to get reasonable results. Therefore, the edge-weight distribution is always set to be a uniform distribution on $[0.1, 1.1]$. As long as not mentioned differently, the number of graphs per algorithm is 3 and $n = 20, r = 3, d = 3, k = 2$. The choices of the parameters are justified in the respecting sections. In this chapter, especially in the plots, the term "approximation algorithm" refers to algorithm 4, whilst with "brute-force algorithm" usually algorithm 3 is meant.

6.1 Input graph sizes

For evaluating the implementation of algorithm 4 in comparison to the brute-force solution, the maximal size of graphs with respect to n, which can be brute-forced in a reasonable time is determined to asses the scalability of the different algorithms. The results for the runtime of the algorithms for different numbers of vertices can be seen in fig. 6.1. 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 takes around 18 s compared to 38 s for computing a small expansion set with algorithm 4. As the brute-forcing time roughly doubles for each additional vertex, it is infeasible 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 $r = 6$.

Additionally, for analyzing the constant C in theorem 2, a plot of as many graphs as possible seems to be ideal. So not only for brute-forcing but also for executing the estimation algorithm, the graph should not be too large. In conclusion, half a minute seems to be a reasonable runtime for computing one expansion set the algorithm. Therefore, n is set to be 20 for further analysis.

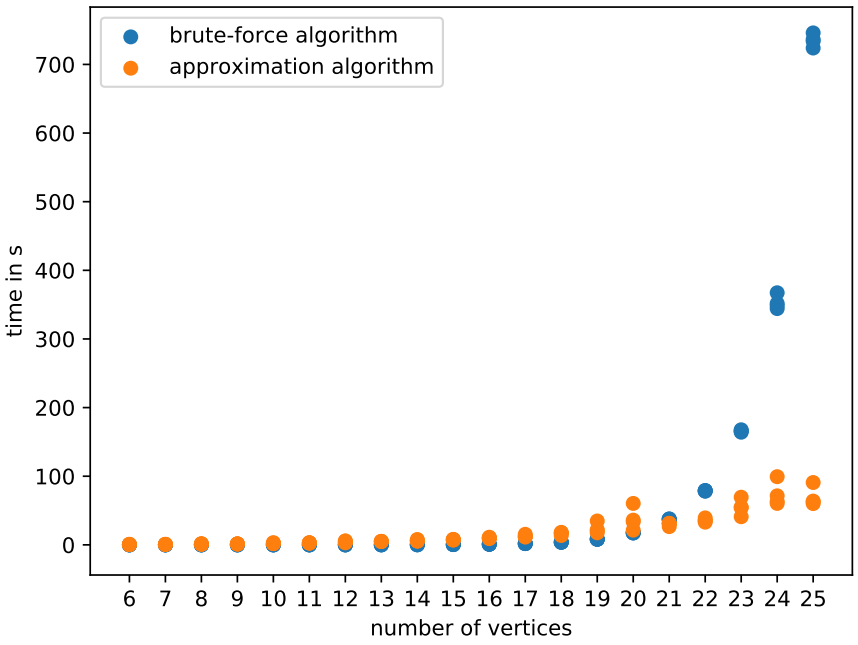


Figure 6.1: Plot of the number of vertices n in the graphs against the time for computing solutions. It can be seen that the brute-force algorithm takes a long time for larger graphs, while the approximation algorithm's time only increases slowly.

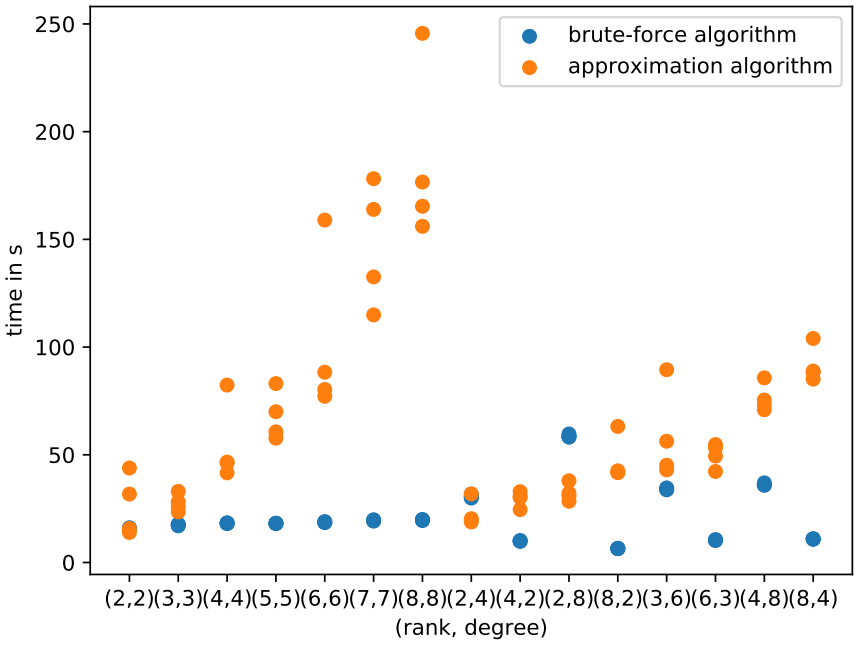


Figure 6.2: Plot of runtimes for different (rank, degree) combinations.

6.2 Rank degree combinations

As the graphs generated by the algorithms have uniform ranks and regular degrees (except those generated by algorithm 14), it needs to be determined which combination of r and d should be chosen. As already mentioned, because of eq. (1.1) they can not be chosen freely, since not for all combinations r-uniform d-regular graphs exist on n vertices, as $m = \frac{nd}{r}$ might not be integer. As all of the variables need to be 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, r) was 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, interestingly 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.

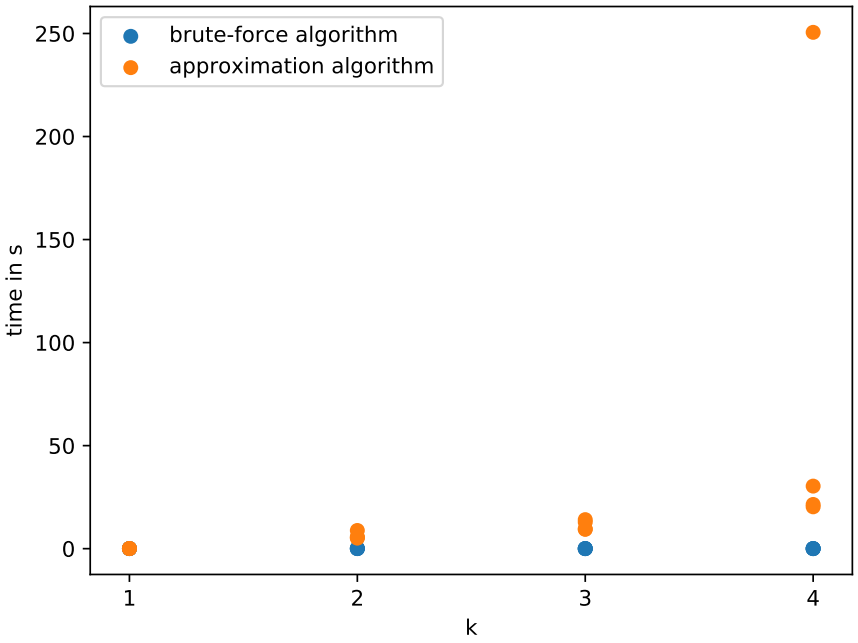


Figure 6.3: Plot of k against the runtime for the approximation algorithm.

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 anyway, as the spectral properties of the graph would not be taken advantage of, as in algorithm 5, for $k = 1$, only the unit vector would be returned. As seen in fig. 6.3, for higher k, the runtime for 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 ks, sometimes 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.

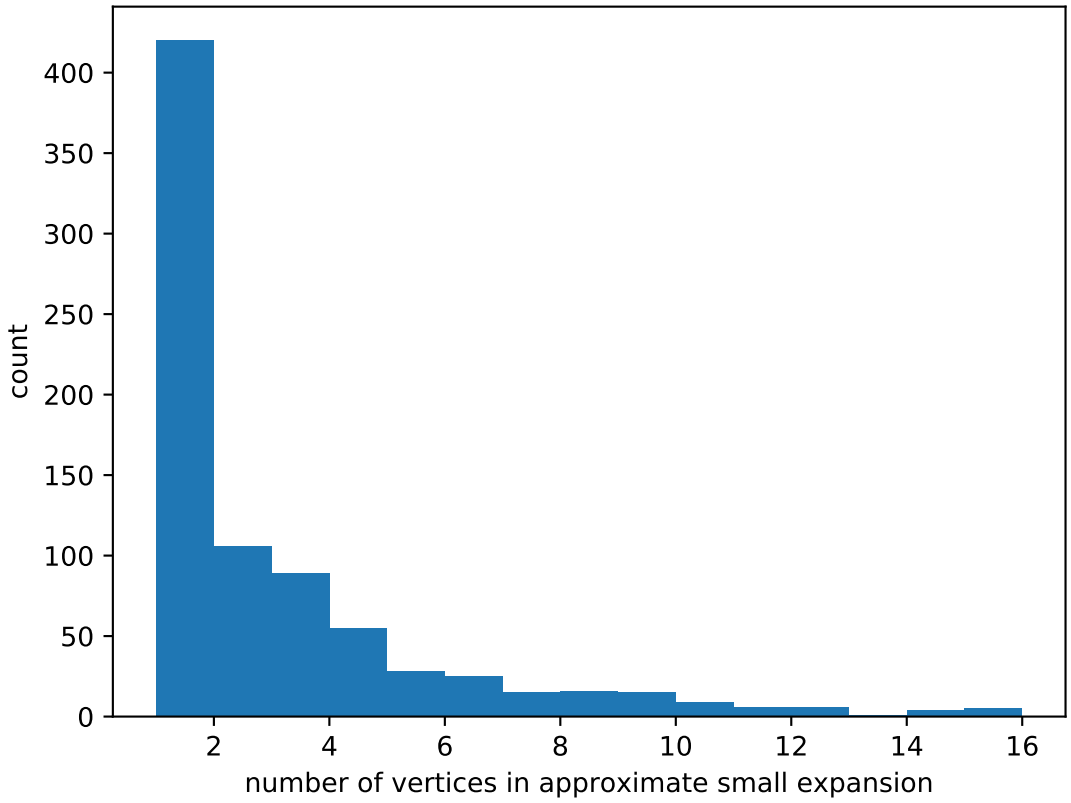


Figure 6.4: Plot of sizes of small expansions.

Since higher values of k were not feasible in eq. (3.3) due to numerical issues of the implementation as well as time constraints, the upper limit of the expansion size $|S| < \frac{24|V|}{k}$ in theorem 2 could not be verified as $k < 5$ here, but a $k > 24$ would be required.

One aspect which might be noted is that most of the expansions have size 1, which does not prove any insight, as the expansion of every subset S with just one vertex $v \in S$ takes value 1, as $\delta S = w_v = w(S)$ (if every edge contains at least two vertices). In a real-world application of this algorithm, these sets might be discarded.

6.5 Random graphs comparison

As the expansion for non-connected graphs is always 0, which is not interesting, only algorithms which guarantee to return connected graphs are considered, namely 14, 15 and 16.

The expansion of the graphs is evaluated against each other via brute-force. The results can be seen in fig. 6.5. Interestingly, no big differences in the expansion values can be determined. The average expansion value of 14 was slightly higher than the value of 15 and 16, which might be the case because 14 did return non-regular graphs.

6.6 Estimation of C

As according to theorem 2 a small value of C would be favourable to ensure that the expansion sets generated by algorithm 4 have small expansions values. For estimating C, the inequality of theorem 2 can be changed to:

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

In eq. (6.1), log refers to the natural logarithm, requiring a value of $k \geq 3$. As for higher values of k the at times the calculation takes very long, it is set to $k = 3$ here.

However, the plot in fig. 6.6 does not show a clear picture, as the calculated values for C of each run of algorithm 4 lie between 0.38 and 2.71. However, one could hypothesize that the value of C indeed is not much bigger than 2.71, since no bigger estimates appeared and most of the values were close to it. It might be a coincidence, but the value is similar to Euler's number e, which might give some inspiration for further proofs on upper bounds of C. However, for higher confidence, this should be tested on a large scale, especially with higher k and other combinations of the other parameters and graph construction algorithms.

6.7 Comparison of expansion values

Finally, the quality of the expansion values is analyzed. Here it seems reasonable to compare only expansion sets of the same size with another. As seen in fig. 6.7, the lowest expansion value found by the brute force algorithm is, as expected, always at least as low as the value achieved by the small expansion set found by the approximation algorithm. However, when comparing the best expansion values, found in 100 repetitions of the steps after solving the SDP in algorithm 4, against the best one-percentile of values as found by the brute-force algorithm, the approximation algorithm performs better in

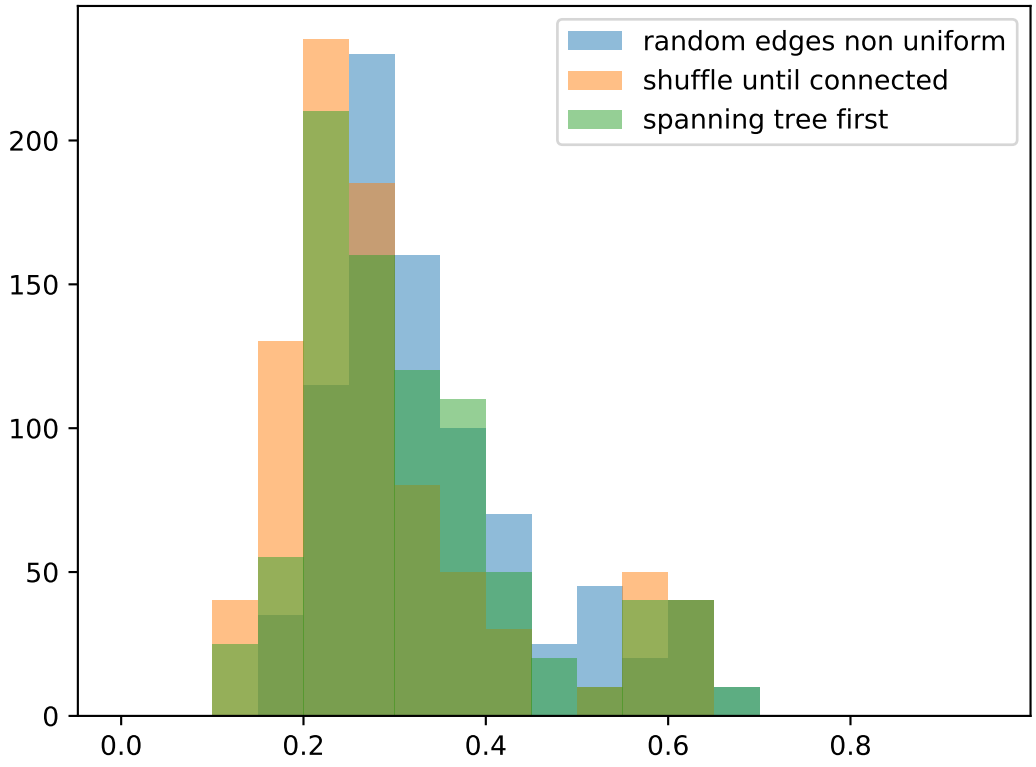


Figure 6.5: Plot of the lowest expansion values of graphs created by different algorithms for each size of the expansion set. For the calculation algorithm 2 was utilized.

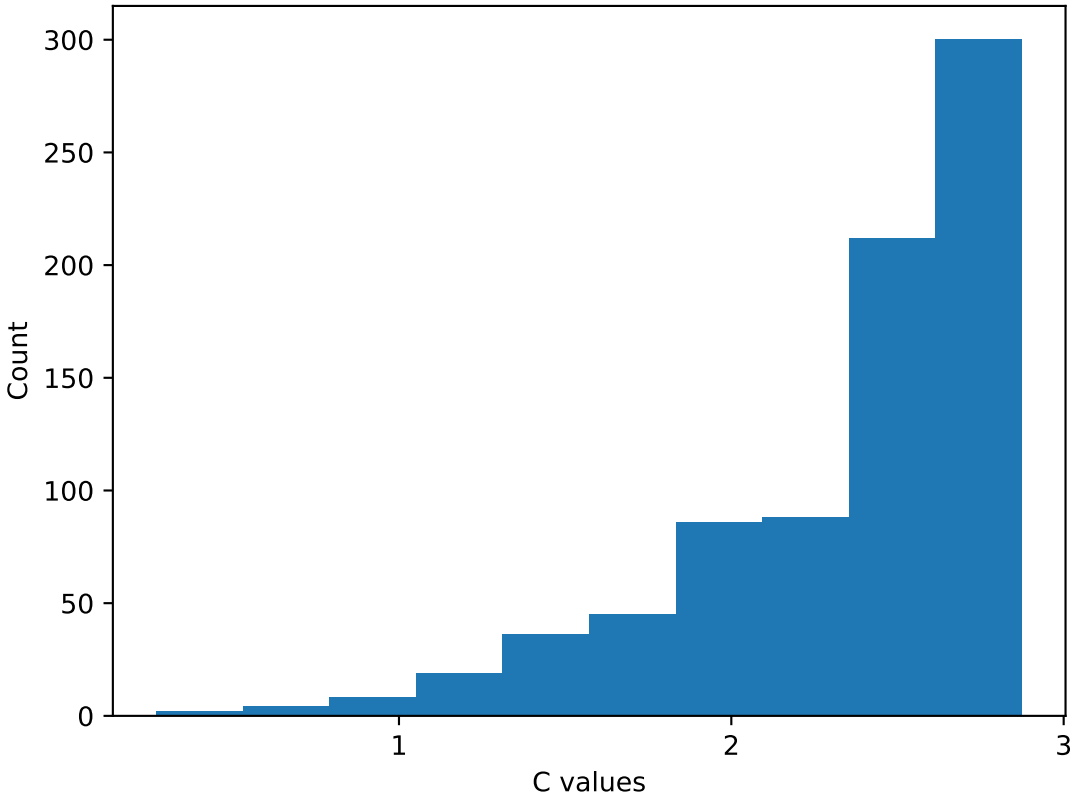


Figure 6.6: Plot of estimates for the constant C in theorem 2.

around 21% of times, even though that is not the aim of the approximation algorithm. Compared to the average brute-force value, which is the average expansion over all sets, it can be seen that the approximation algorithm performs significantly better than random guessing, as most of the entries lie above the line. The same can be seen when the median expansion value is considered, which is a positive result.

As the performance of the algorithm is now evaluated, possible applications can be discussed in the next chapter.

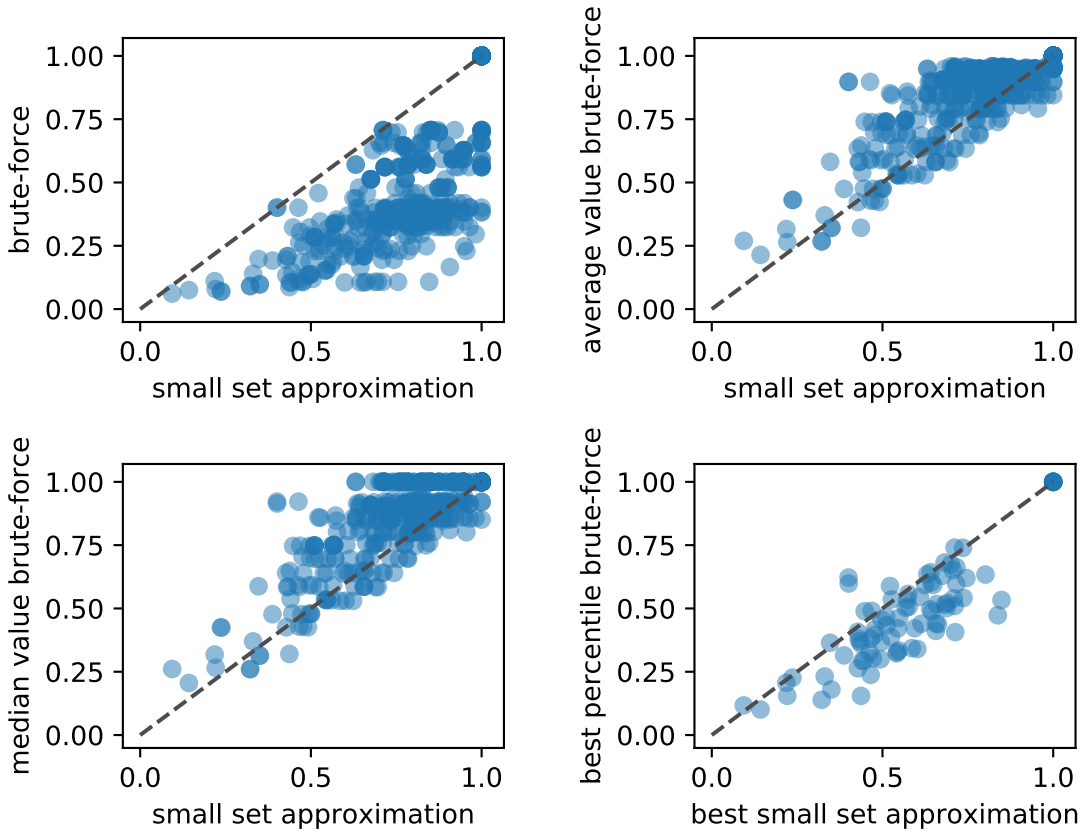


Figure 6.7: Plot of the expansion values achieved by the small expansion set approximation algorithm against the expansion set of the same size with the lowest expansion (as generated through the brute-force approach). Entries below the line signal that the expansion found by the approximation algorithm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

All in all, approximation algorithms for creating small expansion sets showed to be a stimulating topic from a theoretical perspective. However, they also have interesting applications in the real world as seen in chapter 7. In this thesis, it is shown in chapter 3 that by making use of the spectral properties of hypergraphs, through creating orthogonal vectors, using an SDP, one can find sets of strongly connected vertices with a low edge expansion. This was verified with an implementation which is described in chapter 5 and the results are evaluated in chapter 6. Additionally, finding an algorithm for creating random hypergraphs which fulfill certain properties chapter 4 proved to be appetizing for further research.

The biggest challenge in this thesis were the extraction and understanding of the approximation algorithm and the algorithms it depends on. Furthermore, developing algorithms for the creation of random graphs with the placed properties demonstrated to be tricky. During implementation, especially finding a suitable optimizer for the SDP and tuning the right parameters for it not to get stuck were challenging.

Several aspects for further work can be noted. 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 require 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 machine 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. From a theoretical perspective, the algorithms can be analyzed for improvement potential and bounds for the value of the constant can be investigated. Finally, the construction of random hypergraphs can be analyzed further and the used ideas combined in other ways.

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.