Practical: Matching in Hypergraphs

MIKA ROTHER, Heidelberg University, Germany

Finding large matchings in hypergraphs is a major problem in the field of combinatorial optimization. In this practical the goal is to find large hypergraph matchings. The underlying idea is to build a graph model based on a hypergraph structure and then solve it with maximum independent set solvers. The experiments show that this approach is quite useful for hypergraphs up to a certain size in order to find a large hypergraph matching.

1 INTRODUCTION

Finding a large hypergraph matching is a NP-complete problem, so there must be good heuristics and approaches to tackle this problem, so that one can find as large hypergraph matchings as possible. Our approach to do so, is to take a hypergraph, then define a relation between the hyperedges and the nodes of this hypergraph with the nodes and the edges of a graph and then built a graph model based on this relation. The defined relation, that will be further explained in Subsection 2.1, yields to a relation between hypergraph matching and the maximum independent set of a graph.

A hypergraph $\mathcal{H} = (V_H, E_H, \omega)$ has a set of n nodes $V_H = \{1, \dots, n\}$ and a set of hyperedges $E_H \subseteq \mathcal{P}(V_H) \setminus \emptyset$. Moreover, the hyperedges may be weighted. The nodes $v \in V$ contained in a hyperedge are called *pins*, and the set of nodes of one hyperedge is called *net*. The hyperedge weights are defined by $\omega : E_H \longrightarrow \mathbb{R}_{\geq 1}$. Furthermore, hypergraphs can be r-partite and d-uniform. According to a definition of Jafarpour-Golzari and Zaare-Nahandi [1], an r-partite ($r \geq 2$) hypergraph $\mathcal{H}$, is a hypergraph which $V_H(\mathcal{H})$ can be partitioned to r subsets such that for every two vertices v_1, v_2 in one partition, v_1, v_2 do not lie in any hyperedge. A hypergraph $\mathcal{H}$ is called d-uniform, if all its hyperedges have the same cardinality d.

A graph $G = (V_G, E_G, c)$ also contains nodes V_G and edges E_G, but here an edge runs only between two nodes $v_1, v_2 \in V_G$. The node weights c are defined by $c : V_G \longrightarrow \mathbb{R}_{\geq 1}$.

A matching in a hypergraph $\mathcal{H}$ is a subset $\mathcal{M} \subseteq E_H$ such that all pairs of hyperedges $e_i, e_j \in E_H$ have an empty intersection, i. e., no common node in their nets. The goal is to find the largest possible hypergraph matching, which from here on is defined by $\mathcal{M}_{\max}$. Now this is given by either the *maximum-cardinality matching (MCM)* or the *maximum-weight matching (MWM)*. For the MCM, $\mathcal{M}_{\max}$ corresponds to the hypergraph matching $\mathcal{M}$ where the set of hyperedges contained in $\mathcal{M}$ is the largest. In contrast, for the MWM, $\mathcal{M}_{\max}$ is given by the $\mathcal{M}$ where the sum of the weights of the hyperedges in $\mathcal{M}$ is largest. If the hypergraph under consideration is not weighted, then MCM and MWM are equivalent.

An independent (node) set of a graph is defined as a subset $U \subseteq V_G$, so that all nodes in U are not adjacent to each other. The independent set is called maximal if no more nodes from V_G can be added to U without U no longer being an independent set. Finally, the maximum independent set is the maximal independent set whose cardinality is the largest of all. Since in Section Algorithm Design we define a relation between the maximum hypergraph matching and the maximum independent set of a graph, the value of the maximum independent set from here on is also defined by $\mathcal{M}_{\max}$.

Author's address: Mika Rother, rothermika@gmail.com, Heidelberg University, Im Neuenheimer Feld 205, Heidelberg, Baden-Württemberg, Germany, 69120.

To make comparisons with current state-of-the-art algorithms that compute hypergraph matchings, the *Karp-Sipser* algorithms that Uçar et al. [5] had used are compared with the *maximum independent set solver (MISS) algorithms* from KaMIS [2]. Since the MISS compute a maximum independet set of a graph, we have to built the graph model first. How exactly this works will now be described in Algorithm Design.

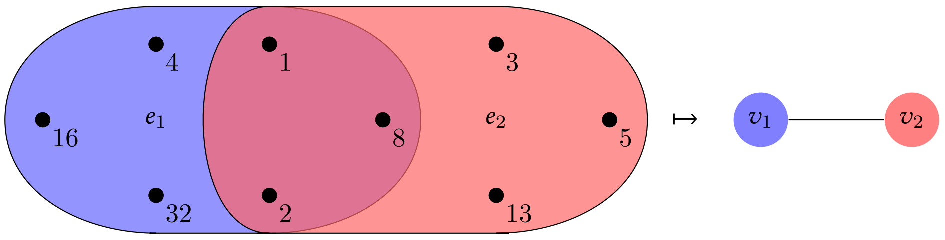
2 ALGORITHM DESIGN

2.1 Hypergraph matching and maximum independent set

First of all, the question arise how exactly to built a graph model based on a hypergraph and how to compute the hypergraph matching of the original hypergraph with the graph afterwards. Let us first consider the following hypergraph $\mathcal{H} = (V_H, E_H, \omega_H)$. This hypergraph has nodes, hyperedges, and weighted hyperedges as described in Section 1. The idea now is to map each hyperedge in $\mathcal{H}$ to a node of a graph. As described in Algorithm 1, the edges between nodes in the resulting graph are then given by whether the original hyperedges contained common pins, i. e., their nets are **not** disjoint. We denote the constructed graph from here on only by $\mathcal{G}$. We define the mapping function f by

$$\begin{aligned}
&f : \mathcal{H} \to \mathcal{G}, \\
&e_i \mapsto v_i, \quad \forall i : e_i \in E_H, v_i \in V_G \\
&(e_i, e_j) \mapsto \left\{ \begin{array}{ll} \emptyset, & \text{if } e_i \cap e_j = \emptyset \\ e = (v_i, v_j) \in E_G, & \text{else} \end{array} \right\}, \quad \forall i, j, i \neq j : e_i, e_j \in E_H
\end{aligned}$$

To give an example: Consider the hyperedges $e_1 = \{1, 2, 4, 8, 16, 32\}$ and $e_2 = \{1, 2, 3, 5, 8, 13\}$ and then built the graph model as just described, then the hyperedges e_1 and e_2 are mapped to the nodes (in the graph) v_1 and v_2. And since e_1 and e_2 have common nodes ($\{1, 2, 8\}$), there also exists an edge between v_1 and v_2 in the resulting graph. (This example is illustrated below). Besides, it does not matter how many nodes two hyperedges share during the construction of the graph model. As long as there is at least one node in both, there will always be an edge in the constructed graph.



Lemma 1. *If we map the hyperedges to nodes, and make edges between these nodes if the hyperedges contain common nodes, that then the maximum hypergraph matching is equivalent to the maximum independent set of the constructed graph model, i. e., $\mathcal{M}_{\max}$ is the same for both.*

Proof. Given the definition of the *maximum cardinality matching*, as in Section 1, there is a subset $\mathcal{M}_{\max} \subseteq E_H$, such that all pairs of hyperedges $e_i, e_j \in \mathcal{M}_{\max}$ have an empty intersection, i. e., $e_i \cap e_j = \emptyset$. The mapping function above, $f : \mathcal{H} \to \mathcal{G}$, now maps all hyperedges to nodes, i. e., $e_i \mapsto v_i$, $\forall i : e_i \in E_H$. All pairs of hyperedges in $\mathcal{M}_{\max}$ that are mapped by f have an empty intersection, so all this pairs of nodes in $\mathcal{G}$ are not adjacent to each other, because of the definition of f. So all these nodes are part of a maximal independent set in $\mathcal{G}$. And because they were mapped by hyperedges that are in the maximal cardinality matching, the maximal independent set is also the maximum independent set. □

2.2 Algorithm to construct the graph model

Based on this idea, we develop an algorithm that takes a hypergraph $\mathcal{H}$ and constructs a graph model as described in Subsection 2.1. HgrToGraph contains a description of the algorithm in the form of pseudocode.

Algorithm 1 HgrToGraph

```
 1: Input Hypergraph $\mathcal{H}$
 2: procedure Construct Graph Model
 3:     $result \leftarrow [[T_1], \dots, [T_n]]$    // a vector of vectors where $n = |E_H|$
 4:     $S \leftarrow 0$    // this variable counts all not disjoint hyperedges
 5:     for $e \in E_H$ do
 6:         $T \leftarrow []$    // saving all overlapping edges, initially $T$ is empty
 7:         for $v \in e$ do
 8:             for $e' \in E_H$ where $v \in e'$ do
 9:                 if $e$ is not $e'$ then $T \leftarrow T \cup \{e'\}$
10:         sort $T$ and erase duplicates;
11:         $result[e] \leftarrow T$
12:         $S \leftarrow T.size$
13:         $T.clear$    // clear the array for next iteration
14:     $S \leftarrow S/2$    // algorithm counts every hyperedge twice
15:     return $result$    // $result$ contains all information of $\mathcal{G}$
```

This algorithm, as well as the whole program for the construction of a graph model from a hypergraph can be found in [3]. The program contains beside the algorithm HgrToGraph also some further components, which are described in the following subsections 2.3-2.5 in more detail. One additional remark, the algorithm also works for edge-weighted hypergraphs, but all experiments in Section 3 are made only with unweighted hypergraphs.

2.3 Data structures

First of all, let's have a look at the format of the input and output files. The input file logically contains information about the hypergraph, while the output file contains the data about the graph model. Both files are in hMetis format. On the one hand this makes it easier to built $\mathcal{G}$ from $\mathcal{H}$ and on the other hand the files have to be in hMetis format to be solved by KaMIS.

2.4 Reduction of hyperedges

After the algorithm has delivered first results, the next step is to use r-partite hypergraphs to built the graph model to make comparisons with [5]. Thereby, synthetically generated hypergraphs are used on the one hand and real-life data on the other hand. For both approaches we use the same data as [5]. To generate the synthetic hypergraphs we use an *Matlab* program that generates k-out hypergraphs, where k is the ratio of hyperedges to nodes. For example, if $k = 2$, there are twice as many hyperedges as nodes. With the resulting graphs it is possible to work successfully, (more about this in Section 3).

The problem is then the real-life data sets that can be found in [4]. These are large tensors whose instances have several million hyperedges and are therefore too large to be solved by KaMIS, or so large that the algorithm HgrToGraph doesn't terminate.

Therefore, we decide to write a function that deletes those hyperedges that contain pins that are included in very many hyperedges i. e., that have a large edge degree. Thereby, it is possible to reduce the large instances from [4] such that they can be successfully used to built the graph model. However, a lot of hyperedges always have to be deleted in order for KaMIS to be able to deal with the generated graphs. Of course, this ensures that one can no longer compare the results of KaMIS with those of Uçar et al. [5] so accurately. Accordingly, the reduced hypergraph is also written to a file so that comparisons can be made with them as well.

2.5 Parallelization of the algorithm

Although we can finally create graph models based on the reduced hypergraphs, this still takes a long time in some cases. Another optimization that we make, is to parallelize Algorithm 1. So we rewrite the program to pass the number of threads that should be used to compute $\mathcal{G}$. For this the hyperedges in the first for loop in Algorithm 1 are divided between the threads. So that a thread that handles a hyperedge e_i writes its results at the ith position of the resulting array. Since the machine on which the calculations are done has 32 threads, significant speed-ups can be achieved. More detailed results on the speed-up by parallelization can be found in Subsection 3.5.

3 EXPERIMENTAL RESULTS

The experiments are performed on a computer equipped with Intel Xeon Silver 4216 CPU and 93GB RAM. Therefore the machine has 32 threads.

3.1 k-out hypergraphs

As already mentioned in Subsection 2.4, the first instances with which we make comparisons, are k-out hypergraphs. In generating these instances, the following parameters are varied: the dimension r, k, and the number of hyperedges n, with $r = \{3, 9\}$, $k = \{2, 4, 8\}$, and $n = \{1000, 10000\}$. The results for 3-partite and 9-partite hypergraphs can be observed in fig. 1 and in fig. 2.

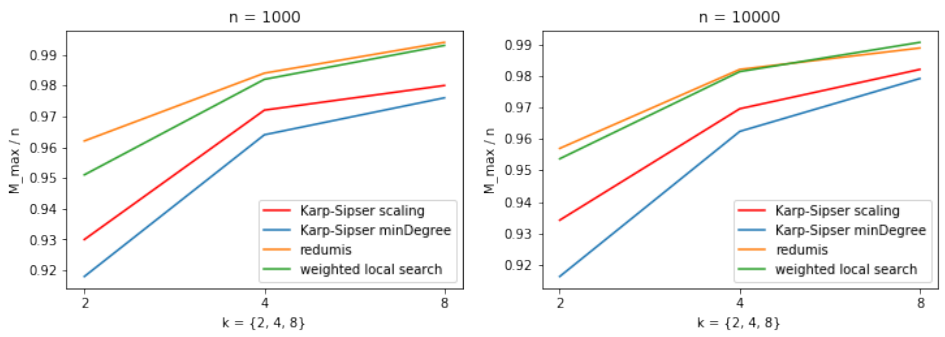


Fig. 1. Comparision of $\mathcal{M}_{\max}$ for 3-partite hypergraphs of the *Karp-Sipser* algorithms used by [5] with two algorithms from KaMIS for $k = \{2, 4, 8\}$ for once $n = 1000$ and once $n = 10000$ hyperedges.

In all cases, it can be clearly seen that the KaMIS algorithms achieve better values for $\mathcal{M}_{\max}$ and thus find a greater hypergraph matching.

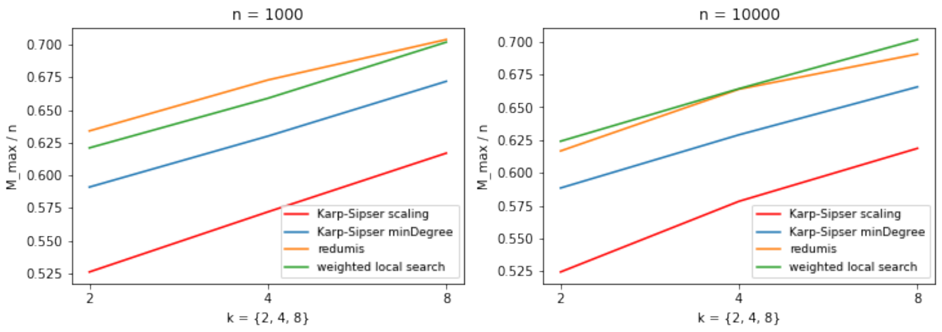


Fig. 2. Comparision of $\mathcal{M}_{\max}$ for 9-partite hypergraphs like in fig. 1

3.2 Real-life data and reduced hypergraphs

In this section, we compare the results of the *Karp-Sipser* algorithms operating on the original tensors from [4] with the reduced hypergraphs. Before we get to the evaluation, let us first describe how many hyperedges are removed during the reduction.

Three tensors were used for comparison, *uber*, *nips*, and *nell-2*, (all three can be found in [4]). The following table shows how many hyperedges the original instances have and how many the reduced.

Tensor	$\lvert E_H \rvert$ of the original hypergraph	$\lvert E_H \rvert$ of the reduced hypergraph
uber	3,309,490	205,912
nips	3,101,609	979,195
nell-2	76,879,419	765,415

As one can clearly see, the instances are sometimes even shrunk very significantly so that KaMIS can work on them. Accordingly, the results of fig. 3 show that the KaMIS algorithms unfortunately cannot achieve as good values for $\mathcal{M}_{\max}$ as those of Uçar et al. [5].

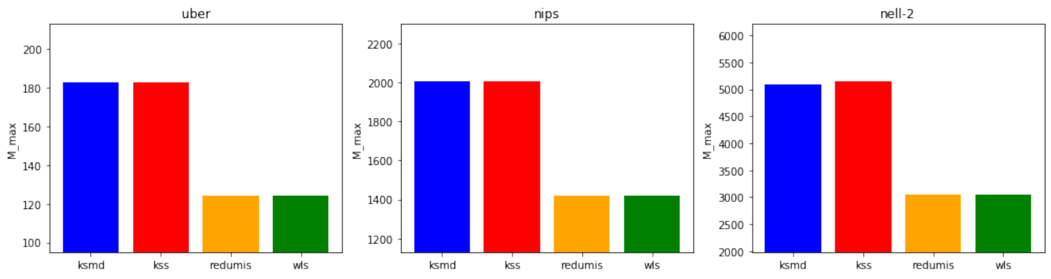


Fig. 3. Comparision of $\mathcal{M}_{\max}$ for real-life data from [4] of the *Karp-Sipser* algorithms with two algorithms from KaMIS.

3.3 Reduced hypergraphs only

To get a fairer result, the *Karp-Sipser* algorithms are also applied to the reduced hypergraphs to see if they perform as well as the KaMIS algorithms on the same instances.

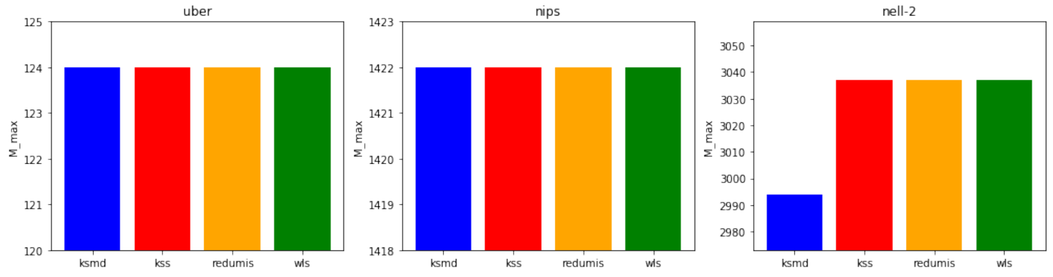


Fig. 4. Comparision of $\mathcal{M}_{\max}$ for the reduced instances from [4] of the *Karp-Sipser* algorithms with two algorithms from KaMIS.

Fig. 4 shows that in the case of reduced instances both approaches do not differ much. Only the *Karp-Sipser min-degree* algorithm cannot achieve as good a value for $\mathcal{M}_{\max}$ on the reduced hypergraph of the *nell-2* tensor as the other algorithms.

3.4 Trade-off between $\mathcal{M}_{\max}$ and speed

One additional thing we try to evaluate is the trade-off between $\mathcal{M}_{\max}$ and the speed of Algorithm 1. For this approach we reduce the number of hyperedges again and check how $\mathcal{M}_{\max}$ differs from the values evaluated in Section 3.3 and 3.4. For this experiment we use again the tensor *nips*, and use the algorithms *redumis* and *weighted-local-search (wls)*, which are both provided by KaMIS. The following table shows the results of this experiment:

$\|E_H\|$	$\mathcal{M}_{\max}$ (redumis)	$\mathcal{M}_{\max}$ (wls)	Speed in [s]
979,195	1422	1422	2.55377
715,306	1218	1218	1.58231
451,548	944	944	0.7588
227,518	623	623	0.302595
52,817	212	212	0.075519

3.5 Measuring speed-ups

Before we reach the conclusion, we will evaluate the speed-ups mentioned in Subsection 2.5 that can be achieved by parallelizing the HgrToGraph algorithm. Fig. 5 shows the speed of Algorithm 1 for different numbers of threads. The tensor *nips* is used as the file to built the graph model, with the full hyperedge set so that the reduced hypergraph exactly corresponds to the original hypergraph. It can be clearly seen that the more threads are used, the faster the algorithm runs as well.

4 CONCLUSION

The results Section 3 show that when the *Karp-Sipser* and the KaMIS algorithms run on the same files, the KaMIS algorithms perform at least as well, mostly even better. So the approach of converting the hypergraph $\mathcal{M}$ to the graph $\mathcal{G}$ definitely has its benefits. Another advantage that Algorithm 1 has over the *Karp-Sipser* codes is that it works not only for r-partite hypergraphs, but for all kinds of hypergraphs. However, the major drawback is that our approach does not work for files that are too large, so there is some limit to how large $\mathcal{H}$ can be.

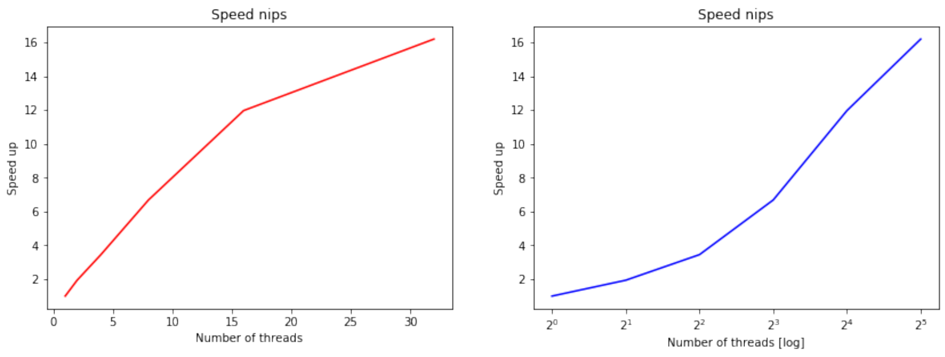


Fig. 5. Both plots show the same instance, where on the right-hand side the x-axis is logarithmic.

REFERENCES

[1] R. Jafarpour-Golzari and R. Zaare-Nahandi. Unmixed d-uniform r-partite hypergraphs. *to appear in Ars Combinatoria*, 05 2016.

[2] S. Lamm, P. Sanders, C. Schulz, D. Strash, and R. Werneck. Finding near-optimal independent sets at scale. 01 2016. doi: 10.1137/1.9781611974317.12.

[3] M. Rother. HyperMatch. https://github.com/suomika/HyperMatch, 2021.

[4] S. Smith, J. W. Choi, J. Li, R. Vuduc, J. Park, X. Liu, and G. Karypis. FROSTT: The formidable repository of open sparse tensors and tools, 2017. URL http://frostt.io/.

[5] B. Uçar, F. Dufossé, K. Kaya, and I. Panagiotas. *Effective Heuristics for Matchings in Hypergraphs*, pages 248–264. 11 2019. ISBN 978-3-030-34028-5. doi: 10.1007/978-3-030-34029-2_17.