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 was to find large matchings. The underlying idea was to convert a hypergraph into a graph 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 matching.

1 INTRODUCTION

A hypergraph $\mathcal{H} = (V_H, E_H, c_H, \omega_H)$ has a set of n nodes $V_H = \{1, \ldots, n\}$ and a set of hyperedges $E_H \subseteq \mathcal{P}(V_H) \setminus \emptyset$. Moreover, both the nodes and the hyperedges may be weighted. The nodes $v_i \in V$ contained in a hyperedge are called *pins*, and the set of nodes of one hyperedge is called *net*. The node weights are defined by $c_H : V_H \longrightarrow \mathbb{R}_{\geq 1}$ and the hyperedge weights by $\omega_H : E_H \longrightarrow \mathbb{R}_{\geq 1}$. Since hypergraphs with node weights cannot be converted to a graph by the algorithm, they were not considered further. Thus, in the following, all hypergraphs are unweighted or have only weighted hyperedges. 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 $G = (V_G, E_G, c_G, \omega_G)$ 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_G are defined the same as c_H and the edge weights ω_G are defined similar as ω_H.

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, as mentioned in the introduction, to find the largest possible matchings, 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 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 Algorithm Design the relation between the maximum matching of a hypergraph and the maximum independent set of a graph is employed, the value of the maximum independent set from here on is also defined by $\mathcal{M}_{\max}$.

To make comparisons with current state-of-the-art algorithms that compute hypergraph matchings, the *Karp-Sipser* algorithms that Uçar et al. [7] had used were compared with the *maximum*

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

independent set solver (MISS) algorithms from KaMIS [2]. Since the MISS compute a maximum independet set of a graph, the hypergraphs had to be converted to it first. How exactly this conversion algorithm works will now be described in Algorithm Design.

2 ALGORITHM DESIGN

2.1 Matching and maximum independent set

First of all, the question arose how exactly to convert a hypergraph into a graph and how to compute the 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 1. The idea now is to convert each hyperedge in $\mathcal{H}$ to a node of a graph. 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.

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 convert them to a graph 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. Besides, it does not matter how many nodes two hyperedges share during the conversion. As long as there is at least one node in both, there will always be an edge in the converted graph.

It is now easy to consider that if we map the hyperedges to nodes, and make edges between these nodes if the hyperedges contain common nodes, that then the maximum matching of the hypergraph is equivalent to the maximum independent set of the converted graph, i. e., $\mathcal{M}_{\max}$ is the same for both.

2.2 Conversion algorithm

Based on this idea, we then developed an algorithm that takes a hypergraph $\mathcal{H}$ and converts it to a graph as described in 2.1. We denote the converted graph from here on only by $\mathcal{G}$. HgrToGraph contains a description of the algorithm in the form of pseudocode.

Algorithm 1 HgrToGraph

```
1:  procedure CONVERSION
2:      Input Hypergraph $\mathcal{H}$
3:      result ← [[$temp_1$], ..., [$temp_n$]]    // a vector of vectors where $n = |E_H|$
4:      overlapping_edges ← 0    // this variable counts all not disjoint hyperedges
5:      for he ∈ $E_H$ do
6:          temp ← []    // saving all overlapping edges
7:          for pin ∈ he do
8:              for he2 ∈ $E_H$ where pin ∈ he2 do
9:                  if he is not he2 then temp.push_back(he2)
10:         sort temp; erase duplicates;
11:         result[he] ← temp
12:         overlapping_edges ← temp.size
13:         temp.clear    // clear the array for next iteration
14:     overlapping_edges ← overlapping_edges/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 conversion of 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 2.3-2.6 in more detail. One additional remark, the algorithm also works for edge-weighted hypergraphs, but all experiments in section 3 were 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 converted graph. Both files are created in hMetis format. On the one hand this makes it easier to convert $\mathcal{H}$ to $\mathcal{G}$ and on the other hand the files have to be in hMetis format to be solved by KaMIS.

To read in the hypergraph file, the framework KaHyPar [5] was used, which provides a fundamental data structure for building hypergraphs, and facilitates the work with the help of many functions. So after the hypergraphs could be converted, first tests with different benchmark sets were made, which were used among others by Schlag [4]. Thereby no comparisons with [7] could be made yet, because their algorithms only work for r-partite hypergraphs. However, it was possible to successfully test some instances and thus establish the functionality of the program.

2.4 Reduction of hyperedges

After the algorithm delivered first results, the next step was to convert r-partite hypergraphs to make comparisons with [7]. Thereby, synthetically generated hypergraphs were used on the one hand and real-life data on the other hand. For both approaches we used the same data as [7]. To generate the synthetic hypergraphs we used 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 was possible to work successfully, (more about this in 3).

The problem was then the real-life data sets that can be found in [6]. These are large tensors whose instances have several million hyperedges and were therefore too large to be solved by KaMIS, or so large that the algorithm HgrToGraph did not terminate.

Therefore, it was decided 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 was possible to reduce the large instances from [6] such that they could be successfully converted. However, a lot of hyperedges always had to be deleted in order for KaMIS to be able to deal with the generated graphs. Of course, this ensured that one could no longer compare the results of KaMIS with those of Uçar et al. [7] so accurately. Accordingly, the reduced hypergraph was also written to a file so that comparisons could be made with them as well.

2.5 Parallelization of the algorithm

Although the reduced hypergraphs could finally be converted, this still took a long time in some cases. Another optimization that was therefore made was to parallelize 1. So the program was rewritten to pass the number of threads that should be used to compute $\mathcal{G}$. Since the machine on which the calculations were done has 32 threads, significant speed-ups could be achieved. More detailed results on the speed-up by parallelization can be found in 3.4.

2.6 Bottleneck outstream

One thing that unfortunately could not be parallelized was the outstream of the file for $\mathcal{G}$. In the final program, this was by far the most time-consuming part, especially for the larger instances. To solve this problem, a few small things were programmed in, such as a buffer, which helped to increase the performance somewhat, but no further actions could be taken to really reduce the runtime drastically.

All in all, the conversion with the final program did not take more than an hour for any instance in the end, so that enough instances could be tested in the final experiments.

3 EXPERIMENTAL RESULTS

3.1 k-out hypergraphs

As already mentioned in 2.4, the first instances with which comparisons were made were k-out hypergraphs. In generating these instances, the following parameters were 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.

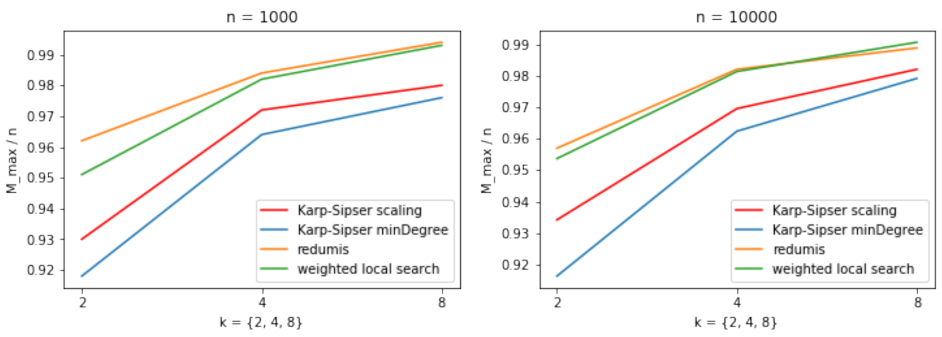

Fig. 1. Comparision of $\mathcal{M}_{\max}$ for 3-partite hypergraphs of the *Karp-Sipser* algorithms used by [7] with two algorithms from KaMIS for $k = \{2, 4, 8\}$ for once $n = 1000$ and once $n = 10000$ hyperedges.

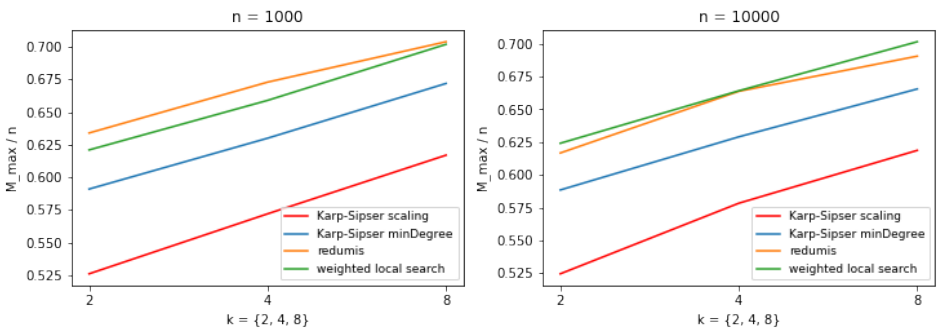

Fig. 2. Comparision of $\mathcal{M}_{\max}$ for 9-partite hypergraphs like in fig. 1

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

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 [6] with the reduced hypergraphs. Before we get to the evaluation, let us first describe how many hyperedges were removed during the reduction.

Three tensors were used for comparison, *uber*, *nips*, and *nell-2*, (all three can be found in [6]). The following table shows how many hyperedges the original instances had 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 were sometimes even shrunk very significantly so that KaMIS could work on them. Accordingly, the results of fig. 3 show that the KaMIS algorithms unfortunately could not achieve as good values for $\mathcal{M}_{\max}$ as those of Uçar et al. [7].

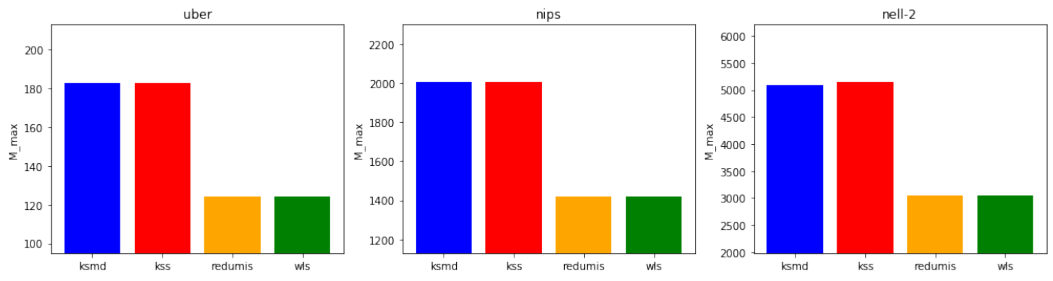

Fig. 3. Comparision of $\mathcal{M}_{\max}$ for real-life data from [6] of the *Karp-Sipser* algorithms with two algorithms from KaMIS.

3.3 Reduced hypergraphs only

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

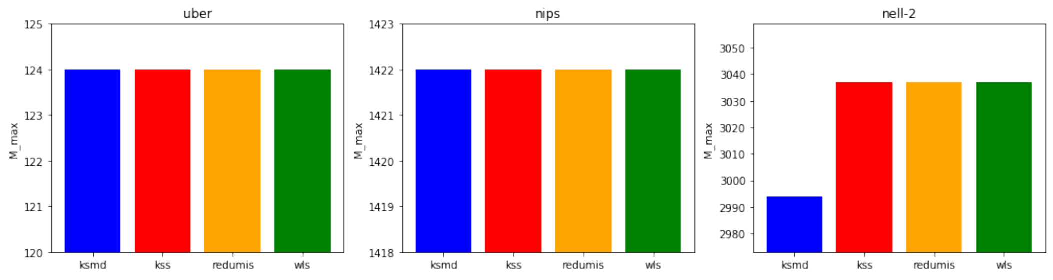

Fig. 4. Comparision of $\mathcal{M}_{\max}$ for the reduced instances from [6] of the *Karp-Sipser* algorithms with two algorithms from KaMIS.

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

3.4 Measuring speed-ups

Before we reach the conclusion, we will evaluate the speed-ups mentioned in 2.5 that could be achieved by parallelizing the HgrToGraph algorithm. Fig. 5 shows the speed of the algorithm 1 for different numbers of threads. The tensor *nips* was used as the file to convert, with **no** hyperedges removed so that the reduced hypergraph exactly corresponded to the original hypergraph. It can be clearly seen that the more threads were used, the faster the algorithm ran as well.

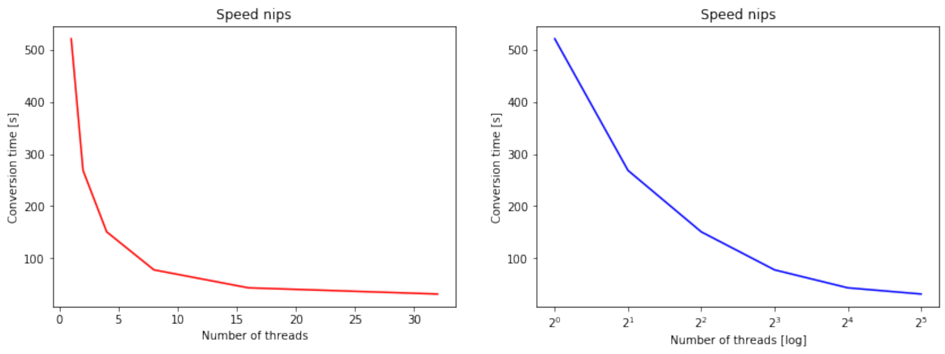

Fig. 5. Both plots show the same instance to be converted, where on the right-hand side the x-axis is logarithmic.

4 CONCLUSION

The results in the section 3 have shown 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 the 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.

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. Schlag. Benchmark sets used in the dissertation of sebastian schlag, 2019. 46.12.02; LK 01.
[5] S. Schlag, V. Henne, T. Heuer, H. Meyerhenke, P. Sanders, and C. Schulz. k-way hypergraph partitioning via n-level recursive bisection. 11 2015. doi: 10.1137/1.9781611974317.5.
[6] 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/.
[7] 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.