MIPRO 2017, May 22- 26, 2017, Opatija, Croatia

New Classes of Kochen-Specker Contextual Sets

Norman D. Megill* and Mladen Pavičić†
*Boston Information Group, 19 Locke Lane, Lexington MA 02420, U. S. A.
†Department of Physics, Nanooptics, Math.-Nat. Fakultät, Humboldt-Universität zu Berlin, Germany and
Center of Excellence for Advanced Materials and Sensing Devices (CEMS), Photonics and Quantum Optics Unit,
Ruđer Bošković Institute, Zagreb, Croatia
Email: *nm@alum.mit.edu, †mpavicic@irb.hr

***Abstract*—Finding Kochen-Specker contextual sets proves to be essential for quantum information and quantum computation in particular. It is therefore essential to find algorithms and programs which can generate arbitrary Kochen-Specker sets in a nearly-exhaustive manner. In this paper we present such generations for two new classes of Kochen-Specker sets. All sets from one of the classes are completely invisible to standard algorithms and programs from the literature as well as the upper part of sets from the second class. We also describe the methods and programs we used to obtain the sets on supercomputing clusters.**

I. Introduction

Kochen-Specker (KS) sets are sets of n-tuples of mutually orthogonal vectors from n-dim Hilbert space to which it is impossible to assign 1s and 0s in such a way that
(i) No two orthogonal vectors are both assigned the value 1;
(ii) In any group of n mutually orthogonal vectors, not all of the vectors are assigned the value 0.

KS sets not properly containing smaller KS subsets are called critical KS sets. If any orthogonal basis of a critical KS set is removed, it will no longer be a KS set. Only critical KS sets are relevant for experimental implementations.

KS sets can be represented as hypergraphs in which each vertex represents a vector and each edge an orthogonal basis.

In addition to presenting new results, this paper describes the methods and programs that we used. The collection of programs that helped us find these results are open source and freely available. Many of them are useful for working with hypergraphs generally, not just hypergraphs representing KS sets. The programs are normally run in a Unix or Linux command-line shell. Most of them perform a specific operation or test on a hypergraph. Because they all read and write a common language for hypergraphs, they can be chained (piped) to each other, and standard Unix utilities such as `grep` can be placed in the chain to filter desired results for the next stage. For very large problems such as finding the KS results described here, we can easily create scripts that can distribute the work over many CPUs in a supercomputing cluster.

For our computer representation, we encode hypergraphs using alphanumeric and other printable ASCII characters. We call these character strings *MMP hypergraphs*. Each vertex (vector) is represented by one of the following characters: `1 2 ...9 A B ...Z a b ...z ! " # $ % & ′ ( ) * - / : ; < = > ? @ [ \ ] ^ _ ` { | } ~` and then again all these characters prefixed by '+', then prefixed by '++', etc. Skipping of characters in this sequence is allowed.

Hypergraph edges (orthogonal bases) are encoded as a string of concatenated vertices, each edge containing as many vertices as there are dimensions, and edges are separated with commas. The complete hypergraph is terminated with a full stop (period) and is contained on a single line of text regardless of length. The order of the edges is irrelevant as is the order of vertices within each edge. The numbers of vertices and edges are unlimited. We often present MMP hypergraphs starting with edges forming the largest loop to facilitate their possible drawing.

A simple example of a hypergraph expressed with MMP notation is `123,3ab,a#++a.` which has 3 edges with 3 vertices per edge and 7 different vertices named `1`, `2`, `3`, `a`, `b`, `#`, and `++a`. (This example is for illustration and does not represent a KS set.)

II. Results

Waegell and Aravind recently constructed a KS set from the Witting polytope [1]. The set has 148 vertices and 265 edges—we denote it as 148-265—and it served us to generate over 300 types of smaller KS critical sets, where a *type* means a KS set with a particular number of vertices and edges. All KS sets that can be generated from this set build a *class* of KS sets which we shall call the 148-265 class. The 148-265 set itself we call the *master set*.

For the generation of KS critical sets from the 148-265, our approach turns out to be indispensable, since a standard approach in the literature, [2]–[8] using what are called parity proofs, turns out to be inapplicable.

A KS set is said to have a *parity proof* if its hypergraph has an odd number of edges and each vertex is common to an even number of edges. Any hypergraph satisfying this condition corresponds to a KS set, because it is impossible to assign an even number of 1s to an odd number of edges if we require exactly one 1 per edge. The converse fails, though; there are KS sets whose hypergraphs don't satisfy this condition.

We obtained over 300 types of KS, and none of them has a parity proof. Thus all the KS sets we found from the 148-265 class are completely invisible to a standard approach using parity proofs. In Fig. 1 we show a graphical representation of the statistics for the obtained KS from the 148-265 class together with two chosen hypergraphs.

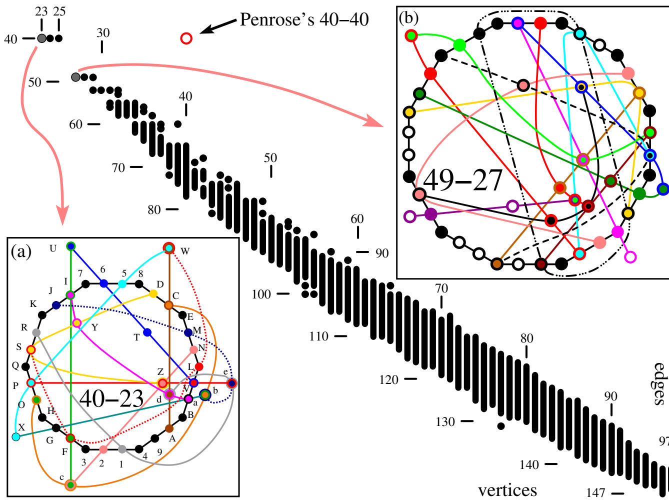

Fig. 1. KS criticals from the 4-dim 148-265 class; 40-40 hollow circle indicates the Penrose 40-40 non-critical KS set; inset (a) shows one of the smallest KS criticals generated from Penrose's 40-40 set; inset (b) shows one of the smallest KS criticals not contained in the 40-40 set.

The other class we considered is also mostly invisible in the standard approach using parity proofs. Waegell and Aravind considered a dual of 600-cell convex regular polytope and obtained a 300-675 master KS set [7]. From it, using parity-proof-based algorithms and programs they obtained 96 types of KS criticals, from 38-19 to 82-41. Then they commented that according to the parity proof method there are no sets smaller than 38-19 and said that they had not found any set larger than 82-41. In Fig. 2 we plotted the statistics of over 260 types we found (top left in red and bottom right in blue) together with their 96 types (top left in cyan).

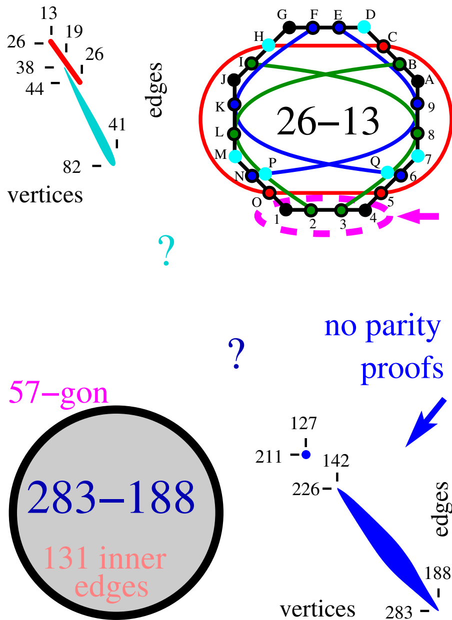

Fig. 2. KS criticals from the 4-dim 300-675 class; there are 250 types of KS critical sets from the upper part of the class (211-127 till 283-188); 26-13 is the smallest KS critical and 283-188 the largest we obtained.

In Fig. 2 we show that, first, there are 250 large KS criticals, from 211-127 to 283-188. Since none of these has a parity proof, Waegell and Aravind did not find them using the parity proof method. We also found the following small types: 26-13, 30-15, 32-17, 33-17, and 34-17, which are all smaller than 38-19. The reason they did not see them is that they are generated directly from large subsets of the master set that do not have parity proofs and are invisible in the parity proof approach. So, our approach is indispensable for generating all possible KS sets, large as well as small.

In particular, we not only revealed the aforementioned huge KS criticals but also showed that the 300-675 class partially overlaps the 60-75 class we generated in [9] (all 26-13,…,44-26 we obtained from the 300-675 master set can also be deduced from the master set 60-75 which defines the 60-75 class). (Their 38-19 is not from the 60-75 class.)

In order to prove that a set has the KS property, we must show that it does not admit an assignment of 0s and 1s such that exactly one vertex on each edge has a 1 assigned to it. Rather than rely on indirect parity proofs, our approach tests this condition directly. Our program `states01` (see Sec. IV-A) does this with a backtracking algorithm that either finds such a state (showing the set is not KS) or exhausts all possibilities (showing the set is KS). In the latter case, it means that at least one edge will be left over without it being possible to assign a 1 to any of its vertices, regardless of how the other edges have their 1 assigned. When a KS set is represented graphically as, e.g., 26-13 in Fig. 2, the KS property instances may be visualised (dashed magenta ellipse indicating such an edge). The parity proofs used by the standard programs in the literature are implicitly contained in our general proofs. However, `states01` can also reproduce the literature results with an additional option that checks whether a KS set also has a parity proof.

III. Algorithms and Programs

A. Overall philosophy

Our programs have a more or less uniform way of specifying the options to control functionality, and learning to use them is straightforward. Most options available for a program perform a single task that is well documented, and the programs can be easily chained via pipes to perform complex functions. All programs written by us have a built-in `--help` option, e.g. `states01 --help`, that provides full documentation of the program's options and features as well as providing examples of the program's use. We have been careful to ensure the help documentation accurately describes the program behaviour, so that in principle the programs can be used by other workers without assistance from us.

A great deal of work has been put into making the programs' runtimes as small as practical, often with good results. Not only does this let us obtain results faster, it reduces the expense of the computation resources needed. A common situation with the types of problems that some of our programs work with is that their run times inherently grow exponentially with MMP hypergraph size in the worst case, which as far as we know is theoretically unavoidable. To overcome this

limitation in a practical sense, a number of heuristic techniques were developed to suppress the exponential behaviour.

For example, the program `states01` iterates through the edges; at each edge, it either makes a trial 0/1 assignment to the unassigned vertices on the edge or backtracks to try another path when an assignment is not possible. A very successful heuristic, which we call the "clustering algorithm," starts with edges that have the most connections (i.e. shares the most vertices) with other edges. This greatly increases the probability of an early assignment conflict, so that backtracking will exhaust all possible assignment paths more quickly when proving that an MMP hypergraph is a KS set.

Since it seems impossible to avoid exponential behaviour completely, all programs having such behaviour incorporate user-settable timeouts. MMP hypergraphs which timed out are flagged in the output and can be put aside to study later, whether it is running them longer or using them to study and improve the algorithms further.

There are occasional situations where the exponential behaviour manifests itself and becomes problematic, i.e. the program gets "stuck," when the hypergraph traversal coincidentally encounters an unlucky pattern. Typically these arise as a long sequence of edges or vertices of length k that do not have early conflicts, and the program will "count" through 2^k possibilities for each later path it backtracks from when a conflict is eventually found. We don't have a good method for identifying these situations in advance; instead, we put them aside to deal with later when they time out. Empirically, we have found that randomising the edge order with our program `mmpshuffle` will often break the unlucky pattern. This has been quite successful in allowing our programs to run to completion for problematic hypergraphs, although it may require some manual trial and error.

In some cases the number of possible MMP hypergraphs that need to explored are astronomical and even with a supercomputer cluster may be unfeasible to explore. For this reason, some programs such as `mmpstrip` have the ability to take random samples of the search space so that we can statistically characterise the properties of the whole search space. We have put care into the randomisation process in two ways. First, for the seed we combine different sources of entropy such as the time of day and the process ID so that it will be unlikely that parallel runs on a supercomputer cluster will use the same random number sequence. Second, we want our results to be reproducible: all programs using random numbers will print out the seed that resulted from the entropy, and the user can specify that seed to replicate the program exactly run if desired.

B. Standard input and output formats

In order to allow the programs to present their outputs in a standard way and communicate to other programs (as well as to humans interpreting the outputs), additional information is sometimes prefixed and suffixed to the MMP hypergraph. An optional *MMP prefix* is any sequence of printable characters (that may include spaces) that ends with a space. An optional *MMP suffix* is a vector assignment enclosed in braces `{...}`; it is described below but in particular does not contain any spaces. An MMP hypergraph with a prefix and/or suffix is called an *MMP line.* An example of an MMP line with both a prefix and suffix is "`#2(1/7=22%) fail:: 1234,1567.{{1={0,0,0,1}}`", where "`1234,1567.`" is the MMP hypergraph proper. If an MMP line has a space, that means it has a prefix which ends at the last space on the line. If an MMP has characters after the full stop ending the MMP hypergraph, those characters are assumed to be an MMP suffix. In all cases, the MMP line ends with a new line character in the computer file containing it (i.e. there must be exactly one MMP hypergraph on each line).

The prefix information depends on what information the program needs to present to the user, and also to allow the program output to be filtered with programs such as `grep` or `sed`, matching for example the string "`fail:: `", as part of a Unix pipe. The format of each program's MMP prefix, if it has one, is documented by the program's `--help` command. Some programs preserve the MMP prefix or suffix while others rewrite them with the program's results or strip them off. The unrestricted format of the prefix (other than its trailing space) gives complete flexibility for a program to present its output information in whatever form is convenient as well as to add future information as a program's capabilities are enhanced.

The optional MMP suffix is a list of partial or complete vector assignments to vertices. Each vector assignment consists of a vertex name followed by = followed by a comma-separated list of complex number expressions in braces, with a complex number for each dimension. The assignments are comma-separated, with the entire list surrounded by braces. An example is `{a={0,1,0,0},++B={0,0,1,i*sqrt(2)}}`. The syntax for the complex number expressions is documented by the `--help` option of the `vecfind` program. Programs that renumber MMP hypergraph vertices or extract subsets of the MMP hypergraph will also modify any vector assignment suffix to keep it consistent.

Overall, the ability to chain programs via pipes and filters, sharing a standard hypergraph "language," allows sophisticated processing to be performed in a relatively simple and robust way. The starting input set can consist of millions or billions of MMP hypergraphs, providing an ideal problem for a supercomputer cluster.

C. Finding KS sets

In order for an MMP hypergraph to correspond to a KS set, there must exist an assignment of vectors to the vertices such that the orthogonality conditions specified by the edges are satisfied. Second, there must not exist an assignment (sometimes called a "colouring") of 0/1 (non-dispersive or classical) probability states to the vertices such that each edge has exactly one vertex assigned to 1 and its others assigned to 0.

For a given MMP hypergraph, we use two programs as filters for these two conditions. The program `vecfind` attempts to find an assignment of vectors to vertices. The program `states01` determines whether or not a 0/1 colouring is possible that meets the above requirement. We will describe their algorithms below.

One basic method in our approach—carried out by means of the program `mmpstrip`—has been to generate successive subsets from a master KS set with many redundancies i.e. that is far from being critical. There are several of these that have been identified in the literature. The goal is to find smaller critical subsets that may be of use in future experiments.

IV. Summary of programs

In this section we summarise the functionality of the programs used in our KS work. In all cases, a program's `--help` option can be consulted for more detail. The input to most programs is a list of MMP hypergraphs (in a file or from the standard input), and the output is zero or more MMP hypergraphs for each input MMP hypergraph. The programs can be chained with Unix pipes to perform more complex filtering and processing of MMP hypergraphs (which we will also call just MMPs in this section for brevity).

Most of our programs are written in the C language for efficiency and designed to operate primarily in Unix environments at the command-line or shell-scripting level. They are written in strict ANSI C so that they will work with different operating systems and compilers. We have been careful to keep the help documentation updated to accurately describe the available options so that they can be used by workers outside of our group. Typically the help includes simple examples that can be reproduced while learning to use the program.

A. *Program* `states01`

This program determines whether or not there exists an assignment of 0/1 (non-dispersive or classical) probability states to the vertices of an MMP hypergraph such that each edge has exactly one vertex assigned to 1 and its others assigned to 0. If not, then the MMP is a candidate for a KS set.

By default, an exhaustive search, using a backtracking algorithm, is done to determine whether an assignment is possible. Optionally, a faster search can be done using the parity proof method (`-p` option). While MMPs with parity proofs are definitely KS sets, MMPs that don't have parity proofs may or may not be KS sets. One purpose of the `-p` option is to be able to reproduce work done by others that is based on parity proofs.

Since the backtracking algorithm is exponentially slow in the worst case, the `-t` option allows a timeout to be specified. We have found that for typical MMPs of interest, timeouts occur relatively infrequently.

The `-c` option tests whether or not a KS candidate is a critical KS. It does this by first checking that the MMP is a KS set, then checking that it ceases to be a KS set when any edge is removed.

For non-critical KSs, the `-r` option will randomly discard edges until a critical KS is obtained. For highly redundant starting KSs, different critical KSs will be produced each run (unless a fixed random number seed is specified to reproduce a previous run).

B. *Program* `vecfind`

In order to determine whether a given KS candidate (i.e. one admitting no 0/1 states) is a true KS set, we must show that a non-conflicting assignment of mutually orthogonal vectors to each edge is possible. In general, this is a difficult problem. In some cases, symmetry of a master starting MMP allows assignment from a small collection of possible vectors; for example, the 60-60 MMP of [5] can be assigned with vectors with components from the set $\{0,\ 1,\ \tau=(1+\sqrt{5})/2, 1/\tau\}$ and their negatives.

The `vecfind` program will search for an assignment from a set of vector components, or a set of vectors, provided by the user. It will show an assignment if one is possible, or tell the user that it is impossible. If a set of vector components is provided, internally `vecfind` will compute all possible non-proportional vectors constructed from them and use that set for the search.

The algorithm assigns vectors from the given or computed list of vectors to the vertices. If a further assignment from the list is not possible (meaning it can't achieve a mutually orthogonal assignment of vectors from the list to an edge containing the vertex), it will backtrack and continue with the next vector in the list at the backtrack point.

Unlike the `states01` backtracking algorithm, there may be thousands of vectors to try at each vertex rather than just the two states 0 and 1. Both programs have exponential behaviour worst case, but clearly it is much more severe with `vecfind` since the number of assignments to reject grows as v^n where v is the number of vectors to try and n is the number of vertices. Because of this, we have invested a large effort into heuristics that experimentally have speeded up the search significantly, sometimes orders of magnitude, for typical MMPs and vector sets. For example, we use a "dynamic" version of the clustering algorithm of `states01`, in which the next vertex to try is the one with the largest number of potential conflicts based on the vector assignments so far, the idea being to encourage earlier backtracking. These efforts have proved moderately successful in that we can find vector assignments already known to the literature (such as the 60-60 mentioned above) almost instantly. However, some large MMPs still cause the program to be excessively slow, particularly when no assignment is possible, and therefore `vecfind` also has a `-t` timeout option.

The output of `vecfind` is the input MMP with a prefix indicating whether the assignment was successful and other information, along with a suffix containing a vector assignment to the vertices. If the assignment was successful, all vertices will be in this list, otherwise the best partial assignment it could find is used for the list. An MMP with a partial vector assignment suffix can be used as the input of a second run of `vecfind` with a different set of vectors to try on the unassigned vertices, and the assignments made by the suffix will remain the same.

The vector components are complex numbers that may be specified directly as in "4+5*i" or they may be specified with expressions involving arithmetic operations and common functions and constants such as sqrt, e, and pi, as in

"e^(2*pi*i/3)". The syntax and available functions are documented in the `--help` output. A built-in calculator, invoked with the option `-calc`, allows the user to check that a given expression evaluates to the expected complex number.

C. Program mmpstrip

Much of our work has involved starting with a very large starting or master KS set, such as the 60-60 (non-critical) KS set mentioned above, and exploring and characterising the smaller critical KS sets that are embedded. This has been a very fruitful effort, allowing us to find millions of non-isomorphic critical KS subsets in this case. This was an unexpected find and means that there is no key critical KS that characterises the 60-60, but instead it provides a rich source of critical KSs that can be used for physical experiments.

To assist studying such master KS sets, the program `mmpstrip` provides a flexible means of selecting subsets from the master. After subsets are selected with `mmpstrip`, they are filtered for ones that are KS sets (have no colouring) using `states01`, then filtered again to eliminate isomorphic ones using the program `shortd`.

From an input MMP hypergraph with n edges, `mmpstrip` will strip k edges so as to produce all $\binom{n}{k}$ subsets with a simple combinatorial algorithm. Partial output sets can be generated by means of start and end parameters. An increment parameter specified by option `-i` can be applied to skip all but every ith output line for partial sampling of the output subsets if the full output is too large. With the option `-c1`, the program will calculate in advance how many output hypergraphs will result, in order to help the user choose optimal parameter settings.

Alternatively, the option `-r` will choose random samples from the master set in order to lessen the chance of a biased selection in case there is a subtle pattern that repeats every ith hypergraph. As with our other programs offering random sampling, the random number seed is displayed and may be used to reproduce the run exactly if desired.

The option `-u` will discard unconnected MMP hypergraphs (i.e. those consisting of two or more disconnected parts). By default, the vertices in output MMPs are relabelled so as to avoid gaps in the vertex naming, although this behaviour may be suppressed with the `-n` option.

The above options are the ones most commonly used and are described in more detail the `mmpstrip --help` output. There are several other options also described in the help documentation.

D. Program shortd

Two MMP hypergraphs are *isomorphic* if one can be transformed into the other via reordering edges, reordering vertices within an edge, and renaming the vertices.

The program `shortd`, written by Brendan McKay, renumbers an input MMP hypergraph into a canonical form, so that two isomorphic MMPs will have the same canonical form and can be easily identified. The program is extremely fast and is based on McKay's extensive theoretical work on graphs and hypergraphs.

We routinely use `shortd` to eliminate isomorphic MMPs from a list, for example after producing a collection of MMP subsets with `mmpstrip`.

E. Program subgraph

This program, described in Ref. [10], will check whether an MMP is isomorphic to a subset of a larger MMP. It uses an algorithm suggested by Brendan McKay. If the two MMPs are the same size, it will tell us whether the two MMPs are isomorphic. While `shortd` will do the same thing much faster, `subgraph` will show the actual isomorphism in case there is one, rather than providing just a yes/no answer.

F. Program loop

Two edges are *connected* if they share a vertex. A *loop* is a set of connected edges in which each shared vertex is shared with exactly two edges in the set, in other words a chain of connected edges where the last edge connects to the first.

The program `loop` by default will list all possible loops in an MMP hypergraph. The list of loops can assist drawing the MMP; in particular, a large loop can be selected as the main circle for the drawing, with other edges drawn with lines to complete the MMP.

`loop` has a number of options that can be seen in its `--help` listing. In particular, the program can list only the b largest loops that were found (`-b` option), it can stop searching after m loops are found (`-m` option), and a timeout can be set to give up and continue from the next edge.

G. Program mmpshuffle

This program is primarily used to randomly scramble order of the edges of an MMP hypergraph and the order of the vertices in each edge, producing an isomorphic MMP with components in a different order. This can be useful to help break exponential behaviour of certain programs caused by unlucky patterns in the MMP. For example, if `states01` times out with `./states01 -t100000 < in.mmp > out.mmp`, we can try `./mmpshuffle -r < in.mmp | states01 -t100000 > out.mmp` to see if a different edge order is more successful. There are also options in `mmpshuffle` to rename the vertices, eliminating gaps in naming, and to reverse the order of edges and vertices.

When the MMP has a vector assignment suffix, `mmpshuffle` will also rename the vertices specified in the assignment accordingly.

H. Program mmptag

This is a small utility program with several functions. It can strip off MMP prefixes and suffixes, it can add prefixes specifying the number of edges and vertices, and it can produce a statistical breakdown of the number of edges and vertices in a large collection of MMP hypergraphs.

I. Program mmpxlate

This program can translate from and to several other hypergraph formats found in the literature. We are open to adding more formats to meet the needs of other groups.

V. Running programs on clusters

We have empirically found that for most tasks it is much more efficient to distribute sequential tasks through, for instance, HTCondor grid scheduling, to nodes in the grid, than to parallelize jobs. An example of our procedures is the following one. HTCondor software allows for two input-output parameters, say i and j. In the first step, we might strip edges from a master set, say 148-265, and filter them with `sed`, `states01`, again `sed`, and `shortd`, so as to generate non-isomorphic KS sets in, say 148-j, $j = 165, \ldots, 264$, i.e., 100 output files, each cut to a chosen number of lines, say 1 million; each line contains a KS set i-j where i depends on j in a rather involved manner depending of how many vertices, if any, were stripped together with stripped edges. We then `grep` KS sets into i-j files each of which contains only i-j KS sets. In the next step, we use these files as input files to generate non-isomorphic critical KS sets i-j-c-k, $k = 1, \ldots, 20$, via `states01` and `shortd`. Each line in these output files contains an l-m final critical KS set, where l and m are not related to i and j except for the inequalities $l \leq i$ and $m \leq j$. Some KS sets are so intricate that they ask for parallelizing tasks, and that takes us to the next section.

VI. Future work

Most of our work up to now has been with MMP hypergraphs that are small enough so that thousands or millions can be characterised on each CPU in a supercomputer cluster in a few days (so in the end we could end up with billions of non-isomorphic KS sets). However, some recent very large hypergraphs, such as the 300-675 mentioned above, stress the limits of some of our programs in that they may take days or weeks to test just one MMP hypergraph. We are planning to give these programs the ability to partition the run for a single MMP hypergraph into different sections that can be run in parallel on different CPUs in a supercomputer cluster. In particular, we are currently working on `states01` and `vecfind` to add this feature.

We are encouraged towards this goal by our success in earlier work with Hilbert lattice equations, where we were able to verify that a huge equation failed in a very large lattice counterexample (the 7oa equation and the Peres lattice of Ref. [11]) with a run that would have taken years if done on a single CPU. By carefully modifying our program `latticeg` described there, we able to partition the run into small pieces that were distributed among many CPUs in a supercomputer cluster, leading to the answer in a few days. This established the independence of this new Hilbert lattice equation from earlier members 3oa through 6oa in the OA (orthoarguesian) family. Incidentally, that work used an MMP hypergraph in the completely different role of representing a Greechie diagram (a kind of orthomodular lattice), showing the flexibility of applications for the MMP notation.

VII. Discussion

An important point to take away from this article is the power of a standard, compact, and precisely specified format for representing hypergraphs, which in our case is the MMP hypergraph notation. It provides the common language that allows all of our hypergraph processing programs to communicate with each other. The code needed to parse MMP hypergraphs into internal arrays, and conversely to generate them from internal arrays, is relatively simple and very fast; it can easily be incorporated into other programs whether written by us or someone else.

The ability to pass MMP hypergraphs from one program to another via Unix pipes, along with Unix filters (`grep`, `sed`, etc.) in between, has allowed us to automate processing of massive hypergraph collections and divide the work among supercomputer CPUs in an efficient way.

As mentioned in Section VI, MMP hypergraphs can be used for other applications involving hypergraphs. The `mmpxlate` program can translate from and to several other hypergraph notations, and we are always looking to add more. As the `mmpxlate` collection grows, we anticipate it will become a generally useful translation tool between hypergraph notations via MMP hypergraphs.

We note that all of the MMP hypergraph processing programs described above, other than `states01` and `vecfind`, can be used for general hypergraph work and are not specialised for the study of KS sets. Most of the programs have been used for many years with a wide variety of MMP hypergraphs, and all are free from known bugs. With the aid of their built-in `--help` option, they are usually no harder to learn to use than a new Unix command.

Acknowledgement

M.P. acknowledges a support by the Croatian Science Foundation through project IP-2014-09-7515, and the Ministry of Science, Education, and Sport of Croatia through the CEMS funding. Computational support was provided by the cluster Isabella of the Zagreb University Computing Centre and by the Croatian National Grid Infrastructure.

References

[1] M. Waegell and P. K. Aravind, "The Penrose dodecahedron and the Witting polytope are identical in $\mathbb{CP}^3$," *ArXiv:1701.06512*, January 2017.

[2] P. Lisoněk, P. Badziąg, J. R. Portillo, and A. Cabello, "Kochen-Specker set with seven contexts," *Phys. Rev. A*, vol. **89**, pp. 042 101–1–7, 2014.

[3] M. Planat, "On small proofs of the Bell-Kochen-Specker theorem for two, three and four qubits," *Eur. Phys. J. Plus*, vol. **127**, pp. 86–1–11, 2012.

[4] M. Planat and M. Saniga, "Five-qubit contextuality, noise-like distribution of distances between maximal bases and finite geometry," *Phys. Lett. A*, vol. **376**, pp. 3485–3490, 2012.

[5] M. Waegell and P. K. Aravind, "Parity proofs of the KochenSpecker theorem based on 60 complex rays in four dimensions," *J. Phys. A*, vol. **44**, pp. 505 303–1–15, 2011.

[6] ——, "Proofs of Kochen-Specker theorem based on a system of three qubits," *J. Phys. A*, vol. **45**, pp. 405 301–1–13, 2012.

[7] ——, "Parity proofs of the KochenSpecker theorem based on 120-cell," *Found. Phys.*, vol. **44**, pp. 1085–1095, 2014.

[8] ——, "Parity proofs of the KochenSpecker theorem based on the Lie algebra E8," *J. Phys. A*, vol. **48**, pp. 225 301–1–17, 2015.

[9] N. D. Megill, K. Fresl, M. Waegell, P. K. Aravind, and M. Pavičić, "Probabilistic generation of quantum contextual sets," *Phys. Lett. A*, vol. **375**, pp. 3419–3424, 2011.

[10] M. Pavičić, N. D. Megill, and J.-P. Merlet, "New Kochen-Specker sets in four dimensions," *Phys. Lett. A*, vol. **374**, pp. 2122–2128, 2010.

[11] M. Pavičić, B. D. McKay, N. D. Megill, and K. Fresl, "Graph approach to quantum systems," *J. Math. Phys.*, vol. **51**, pp. 102 103–1–31, 2010.