Preliminary Report - q-Analog for $\mathcal{S}_2[2, 3, 19]$

Andy Berger* Andrew Elashkin†

September, 2019

1. Road to SS2 Orbits

We start by generating the field $GF(2, 19)$ itself, the minimal polynomial being $x^{19} + x^5 + x^2 + x^1 + 1$. The zero element is first generated, then using a generator element, α, we generate all other elements using their indices, i.e. $\alpha^0, \alpha^1, ..., \alpha^{2^{19}-2}$. Starting from α^0, each element is generated according to $\alpha^{i+1} = \alpha^i * \alpha$ where $\alpha^{19} = \alpha^5 + \alpha^2 + \alpha^1 + 1$.

After inserting the field into a file mapping each index i to its bit representation, e.g. $\alpha^{18} = 1000000000000000000$, $\alpha^{19} = 0000000000000010111$, etc.

In the 2D case, we have calculated the SS2 matrix which contains rows of orbits where each row contains the orbit following a list of all other i's in the orbit. The results have been saved to a file and loaded upon a data structure for use in functions further in the computation process. Each 2D subspace can be represented in the form (α^0, α^i) where α^i is independent of α^0, i.e. $i \not\equiv 0 \mod 2^{19} - 1$ which means they are not the same which is equivalent to independence under fields that satisfy $char(F) = 2$. Since all 2D subspaces are of this form they are uniquely identified by i.

For this reason, we take α^0 and for each $i = 1, ..., 2^{19} - 2$ we check if it already belongs to an orbit. If not, we generate the 6 differences equivalent under the normalizer of the Singer Subgroup, akin to the works in Braun, Etzion et al.[1], and for each such difference, d, generate $d * 2^k \mod 2^{19} - 1$ for $k = 0, ..., 18$. This process generates 114 different indices belonging to the same orbit whose first element is i, this will be reffered to as the orbit representative. We also mark each of these indices as being in said orbit so that when we get to those i's later they will be passed over. In total, similarly to [1], we get $\frac{\begin{bmatrix}19\\2\end{bmatrix}}{(2^{19}-1)*19} = 4599$ orbits each having $\frac{2^{19}-2}{4599} = 114$ 2D subspaces.

*Technion, Haifa, Israel; email: f
†Technion, Haifa, Israel; email: s

After the SS2 matrix generation we update the field file to hold the index of the orbit rep. for each index $i = 1, ..., 2^{19} - 2$. This sums up the data generated in the first part of the program which the parts herein rely on.

2. Road to SS3 Orbits

Each 3D subspace of $GF(2, 19)$ can be expressed as a trio $(\alpha^0, \alpha^i, \alpha^j)$ of independent elements. To ascertain independence we first check that each two elements are independent, i.e. different (as per the last section about 2D subspaces) - simply going over all i's starting with 1 and all j's starting with $i + 1$ while generating achieves this (lower j's were handled before as i's).

Then, we check that the trio is indepedent - there is no linear combination of 2 elements with coefficients over the field leading to the third element. Under $GF(2, 19)$ built around $GF(2)$ this is simply rendered as a sum. If there was such a combination from 2 of the elements to the other it would induce combinations from every 2 elements of the three to the one left, so we simply check that the equality $\alpha^0 + \alpha^i = \alpha^j$ does not hold.

We can calculate j such that $\alpha^0 + \alpha^i = \alpha^j$ and skip over it while generating orbit elements.

Generating a 3D orbit, which is expressed as such an independent trio $(\alpha^0, \alpha^i, \alpha^j)$ where i is an orbit rep. of one of the 4599 2D orbits, consists, therefore, of going over all orbit reps. and for each such rep., i, calculating the above "bad" $j \pmod{2^{19} - 1}$ and generating $(\alpha^0, \alpha^i, \alpha^j)$ for all other j's in $i + 1, ..., 2^{19} - 2$.

A more refined selection can be made by considering not all 3D orbits but only those that have all 2D subspaces they contain in different 2D orbits. A 3D subspace $(\alpha^0, \alpha^i, \alpha^j)$ contains the following 2D subspaces:

$$(\alpha^0, \alpha^i), (\alpha^0, \alpha^j), (\alpha^i, \alpha^j), (\alpha^0, \alpha^i{+}\alpha^j), (\alpha^i, \alpha^0{+}\alpha^j), (\alpha^j, \alpha^0{+}\alpha^i), (\alpha^0{+}\alpha^i, \alpha^0{+}\alpha^j)$$

We can see that any combination of two elements of the trio is independent of the third as we have proven above and so all 7 pairs form valid 2D subspaces. We now normalize each 2D subspace to present it with its first element being α^0, the way it appears in previous sections, in order to characterize it using only the second element. To do this we define $k, u, v \in \{0, ..., 2^{19} - 2\}$ such that $\alpha^k = \alpha^i + \alpha^j, \alpha^u = \alpha^0 + \alpha^j, \alpha^v = \alpha^0 + \alpha^i$ and so:

$(\alpha^i, \alpha^j) = (\alpha^0, \alpha^{j-i})$ $(\alpha^0, \alpha^i{+}\alpha^j) = (\alpha^0, \alpha^k)$ $(\alpha^i, \alpha^0{+}\alpha^j) = (\alpha^i, \alpha^u) = (\alpha^0, \alpha^{u-i})$
$(\alpha^j, \alpha^0 + \alpha^i) = (\alpha^j, \alpha^v) = (\alpha^0, \alpha^{v-j})$ $(\alpha^0 + \alpha^i, \alpha^0 + \alpha^j) = (\alpha^v, \alpha^u) = (\alpha^0, \alpha^{u-v})$

The contained 2D subspaces are now given as:

$$(\alpha^0, \alpha^i), (\alpha^0, \alpha^j), (\alpha^0, \alpha^{j-i}), (\alpha^0, \alpha^k), (\alpha^0, \alpha^{u-i}), (\alpha^0, \alpha^{v-j}), (\alpha^0, \alpha^{u-v})$$

It is possible, however, for some of the pairs to indicate the same 2D subspace or rather fall in the same 2D orbit. To fix this we simply check the data structure to

which the SS2 orbits were stored in the last section and if any two 2D subspaces exist such that their orbits have the same orbit rep. (and thus are in the same orbit) the 3D orbit containing them is disregarded. We can compute this check independly for each orbit in the same manner as generating all 3D orbits as we have mentioned above.

This per-orbit process can then be multi-threaded and run independently, generating all 4599 orbits into files that are concatenated in the end. On our machine, we use 3 threads that each generate a third of the orbits. Each orbit takes about 30 seconds to generate and so using threads here makes a significant improvement.

3. Generating Kramer-Mesner Files

Let us define a version of the Exact Cover problem we will be discussing in this article. We are given a set of numbers, $S = \{x_1, x_2, ..., x_n\}$, and a number of subsets, $S_1, S_2, ..., S_m \subseteq S$. The problem is described as choosing a number of these subsets $S_{i_1}, S_{i_2}, ..., S_{i_k} \in \{S_1, S_2, ..., S_m\}$ which are pairwise disjoint ($\forall j \neq k, S_{i_j} \cap S_{i_k} = \emptyset$) such that $\bigcup_{j=1,...,k} S_{i_j} = S$. Because the selected subsets are pairwise distoint each of the numbers can belong to only one of them, we can think of the number as covered by that subset.

Our objective is to find a set of 3D orbits such that each 2D orbit is contained in exactly one of them - the 2D orbits are the numbers and the 3D orbits the subsets containing them, a solution has each 2D orbit covered by one 3D orbit.

We represent the problem by taking the subsets and numbers as the rows and columns of a table, T, respectively. Each table cell, $T[row, col]$, contains either a 1 or 0: a boolean that is the result of the query *"Does the subset matching row contain the number matching col?"*. This table is called a Kramer-Mesner matrix - it is a standard representation of the Exact Cover problem.

A Kramer-Mesner matrix in our case is a table whose columns range from 0 to 4598 corresponding to the 2D orbits and whose rows each represent a 3D orbit with 0's and 1's in the columns of the 2D orbits that are contained in it.

We do not need to check every 3D subspace and generate all 2D subspaces contained in them. Generating the orbits for both the 3D and 2D subspaces is enough - an Exact Cover using the orbits will yield solutions when looking at the 3D Subspaces of the solution's orbits. Yet, even after all these reductions the task of generating all rows remains computationally unfeasible, since there are still $\frac{\begin{bmatrix}19\\3\end{bmatrix}}{(2^{19}-1)*19} = 86113647$ 3D orbits in total, each of which must hold the result bit for the above query for each of the 2D orbits making the program high in memory both in the sense of physical storage where files containing even just 5000000~ rows of the full Kramer-Matrix matrix take up tens of GB's and the sense of RAM where a program holding data structures containing even just

18000000~ rows takes up above 2GB's which is more than half of the capacity of an ordinary household computer, urging us to move to more powerful servers.

Finally, even having a high-powered server at our disposal would still have a problem with running the solver for the Exact Cover problem, as we will discuss further in the next subsection.

For these reasons and to better utilize the limited resources that we have, we will take only some of the rows. Since, after reducing 3D subspaces by means of grouping them into their respective orbits and only considering those containing 7 different 2D orbits, all current rows are identical in the sense of having the same probability in being in a viable solution we will use the probabilistic approach and take rows at random from each orbit.

4. Dancing Links Implementation

We have implemented Donald Knuth's Dancing Links algorithm from his article [2], in C++. The function signatures and more comments are found in the file `dancing_links.h`. It follows the pseudo-code presented in said article and has a solver function that, given a Kramer-Mesner matrix, constructs a Data Structure from it and runs the solver algorithm, outputting all solutions to a file where every solution is a row of the indices of the rows belonging to the solution. We have tested it on small Kramer-Mesner matrices of $4X4$ Sudokus, the solutions of which were found instantaneously.

It is worthy to mention that the Exact Cover problem was proved to be NP-Complete [3]

5. Generating Partial Solutions

After composing a `Dancing Links` class akin to the one described by Knuth's paper, we implement a partial solution function, `reduce`, that works with the dancing links representation: Given our random Kramer-Mesner table, it generates a partial solution by starting with an empty solution and selecting a random row in each iteration, adding it only if it doesn't intersect with the current solution. It puts the selected rows of the partial solution in one file. In another file it puts those rows again along with all rows of the Kramer-Mesner matrix that don't intersect with the partial solution, this file has only those columns that are not covered by the partial solution.

We start by taking 8 random rows from each 3D orbit many times, each time we are left with a Kramer-Mesner matrix. For each such matrix we generate a Dancing Links object out of it and proceed to run many reduction iterations on it. After generating a Kramer-Mesner matrix several times, each time coming up with the greatest partial solution we can get from it across all iterations, we take the number of rows in that greatest partial solution and write it down next to the number of rows taken from each orbit while generating the matrices (8 in this case) and, finally, increase the number of random rows from each orbit

and repeat the process. Increasing the amount of random rows in each orbit and subsequently the number of rows in the Kramer-Mesner files itself increases the sizes of the partial solutions we are getting. We have saved the best partial solution we got using this technique for every number of rows we are taking from each orbit in a table that is found in the next subsection.

After taking 1000 rows from each orbit we end up with a partial solution of 551 rows and problems start to arise: not only do the Kramer-Mesner files generated become too large to store on our computer (reaching over 20GB's) but also the amount of RAM used to store the table's Dancing Links representaion gets to the point where the program simply crashes on our ordinary computer. This brings forth the need for an efficient representation more suitable for running the reduction procedure. The Dancing Links representation saves 4 pointers for each 1 in the Kramer-Mesner matrix, indicating the four 1's nearest to it in all directions (another pointer is saved to get the top of the column the 1 is in). This plays an important part in the Dancing Links representation of the Algorithm X solver. However, for finding a partial solution by means of generating random rows this feature does not aid us. We then try to think of a new method of representation.

6. Matrix Class

Another way of representing the table looks at a row of seven 1's as the seven indices of the columns where those 1's are set. In a sense, now each 1 in the Kramer-Mesner matrix saves only one integer of the column index with no information on nearby 1's. We implement this in the Matrix class, described in `matrix.h`.

We also build a function that adds several random rows from a given 3D orbit directly to the Matrix class object, skipping the Kramer-Mesner file building process entirely. As a bonus, this also helps lower the running time of the generation process. We end up with an object small enough to hold up to 4000 rows per orbit on our computer and run the reduction algorithm (which works the same way as in the Dancing Links class) on it, emitting a solution of 569 rows.

A table is shown detailing how increasing the number of rows taken from each 3D orbit raises the partial solution size:

Lines taken	Max sol.
8	419
12	436
25	461
50	472
75	491
120	498
150	507

Lines taken	Max sol.
300	522
600	538
800	544
900	547
1000	551
1400	556*
1500	556
2000	563
4000	569

[*] Problem with file output,so no proof for this. Also where we made the representation switch from Dancing Links to Matrix class.

7. Road to 592

After representing the Kramer-Mesner matrix using the new Matrix class we take the same approach of finding a partial solution, skipping the Kramer-Mesner file generation altogether. We will detail how this helps us get to a partial solution of 592 rows:

Being no longer tied to taking only 1000 random rows from each orbit to the size of the Kramer-Mesner file generated we start taking as many as 4000 rows from each orbit. The maximal partial solution we get is now 569 rows long. We then generate ALL 3D orbit rows that comply with the partial solution of 569, resulting in a file that, while small, takes 6 hours to generate with 3 threads. The file contains, for all rows, only the columns not already covered by the rows of the 569 partial solution.

Then, we run the solver on the complying rows and the Kramer-Mesner file we get is small enough at this point for it to say: no solutions (what do you know?). So, we generate a max partial solution for it by running randomly like before. We get 23 as the best result, converte it from the reduced format (with 569's columns crossed out) to the original format with all 4599 columns and concatenate the 569 partial solution with the new 23 rows to get a partial solution of 592 rows.

8. Road to 593

Using the 569 partial solution yeilds a formidable result, but the number of complying Kramer-Mesner rows is somewhat small. The idea now is to use a smaller partial solution as our base and use it find enough complying rows so that the combined partial solution exceeds 592 rows. We need to find a "golden number" of sort for the number of rows for the base partial solution, one that is

not too small having too many complying rows to fit in our memory and one not too large so that there are not enough complying rows to make the combined solution exceed 592.

We end up taking the smallest partial solution we have generated, 419, and, using it as the partial solution on which the aforementioned calculations are based, do the whole thing again. It takes us 6 (multi-threaded) hours but the kramer file is much bigger, a few MB's in size.

As a result, we are not able to run the solver and get the solution so we have to resort into getting a partial solution, the best one giving us enough rows to gain a concatenated solution of 593.

Trying for 200, 300 and even 350 would prove to yield kramer files that are unfeasibly large (we generated some orbits as a start) and so we didn't try going there. Using 400 rows only gave us only 591 rows in the end, which was a letdown. So regrettably, we move on.

9. Reducing and Deducing

We now settle on a number of rows between 419 and 569, 500. We use a function that generates a partial solution of 500 rows with as many threads as we can, generates the reduced Kramer-Mesner table from it, runs the solver and, if it fails, runs a number of partial solution iterations on the reduced table and takes the best one it finds and, if the concatenated partial solution exceeds the current best one, saves the concatenated result to a file. We run this process several times to see what we can get.

This function is `reduce_and_deduce` from `main.cpp`. Right now, it runs for 10 iterations but that can easily be changed. Running it locally on our machine takes about 6 hours per iteration. Using a server, we managed to have 10 iterations take about 8 hours. We were only ever successful in getting only 593 again from the many iterations we had run.

10. Notes

We only generate Kramer-Mesner rows whose $j \geq 2 * i$.

Also, it is worthy to mention that not all partial solutions can lead to a full solution, as we have seen in our results. Having all possible combinations of orbit rows with seven 1's is $\binom{4599}{7}$ which is around 1×10^{14} and the kramer rows are only around 86×10^6 so the chances of finding 657 rows that don't intersect are kind of low. We just have to generate many times and hope for the best.

11. Useful Functions

main.cpp:

`SS2_generator`

Generates a given number of 2 dimensional subspaces of a field under the normalizer of the Singer subgroup

`SS3_generator_aux`

Generates a 3D subspace orbit off a 2D orbit index.

`kramer_generator_aux` Generates a number of random rows (3D subspaces that belong to a 3D subspace orbit with the given index). Only rows whose 7 2D contained subspaces fall in different orbits are counted.

`reduced_kramer_generator_aux` Generates all of an orbit's random rows, that on top of all of what the above function details also comply with a number of columns (don't intersect with them).

`matrix_generator_aux` Generates a number of random rows and stores them and stores them inside a matrix object instead of a file.

`partial_solution` Tries to find a partial solution to the problem by generating random Kramer rows from each of the orbits and choosing from them at random, adding a row only if it complies with the par. sol.

`partial_increase` Experiment: Takes a few partial solutions that were previously generated and tries to add some rows from one to the other.

`overly_ambitious` Experiment: Takes a small number of rows from each orbit for the Kramer-Mesner matrix and tries to solve it using Dancing Links.

`build_reduced_kramer_from_partial_solution` Takes all columns already covered by a given partial sol. and generates all rows that comply with them, the generated reduced Kramer file has entries only in other columns.

`reduced_file_to_original` Takes a reduced Kramer file and and restores the (zero) entries in the missing columns.

`reduce_and_deduce` Generates a partial solution (or uses an existing one) and tries to generate all rows that comply with it, then tries to solve them using Dancing Links and eventually finds a max par. sol. for them, if the combined par. sol. exceeds the current best par. sol. the reduced rows are restored and concatenated to the first par. sol. giving a combined result.

dancing_links.h:

`cols_arr`

Recieves an array signifying a bit array of columns and sets the columns that are covered in the `Dancing Links` object

`is_matrix_partial_solution` Returns `true` if each column of the object is covered by at most one 1.

`portmanteau` Takes two `Dancing Links` objects and tries to add all rows of one to the other but only rows that comply with it are added. Useful when concatenating two small partial solutions that agree with each other.

`reduce` Chooses random rows from the object and tries to add them to a partial solution, after running a few given iterations and chooses the best partial solution it has generated.

`choose_col` The rule by which we choose which column to reduce by, as per [2]. Takes the leftmost row with the minimal number of ones.

`cover` Covers the column chosen as the similarly named function in [2] does.

`uncover` Uncovers the column chosen as the similarly named function in [2] does.

`solve` Employs the solving algorithm of `Dancing Links` and tries to find all solutions for the given object and output them to a file.

matrix.h:

`rows_map_to_vector` Auxilliary Function: Given a vector, inserts the rows of the matrix object to it.

`best_partial_solution` Experiment: tries to greedily search the object and find the best partial solution by taking every possible combination of compliant rows.

`attach_aux` Auxilliary Function: Takes a current partial solution and a vector containing rows of another partial solution and tries to add a given number of these rows to the first par. sol.

`detach_some_and_attach_some_more` Experiment: Takes an a partial solution and a vector containing rows of another partial solution and tris to detach a given number of rows from the first partial solution and then add a higher given number of compliant rows from the other par. sol.

`reduce` Chooses random rows from the object and tries to add them to a partial solution, after running a few given iterations and chooses the best partial solution it has generated.

`add_row` Adds a row of an SS3 subspace to the matrix object.

12. References

1. EXISTENCE OF q-ANALOGS OF STEINER SYSTEMS, Braun, Etzion, et al.

2. Dancing Links, Donald E. Knuth.

3 About NP-Completeness of the Exact Cover problem, Michael R. Garey and David S. Johnson,Computers and Intractability(San Francisco:Freeman, 1979).

4. Also about NP-Completeness of the Exact Cover problem, Reducibility among combinatorial problems, Richard M. Karp.