Computing Equitable Partitions

Tal Rastopchin and Gabby Masini

15 May 2020

1 Introduction

The goal of this project is to write code that calculates the equitable partitions of association schemes. There may be multiple equitable partitions which are equivalent under some automorphism of a given scheme. Thus, we calculate only the representatives of each equivalence class and create another function to calculate all isomorphic equitable partitions. We extended and used Izumi Miyamoto and Akihide Hanaki's code that computes properties of association schemes in order to help compute equitable partitions. All of the code can be found in the GitHub repository.

2 Background

An association scheme on a set X is a set S consisting of subsets of $X \times X$ forming a partition of $X \times X$ such that

- $1_x = \{(x, y) \in X \times X \mid y = x\} \in S$
- for each $p \in S$ the set $p* := \{(y, x) \in X \times X \mid (x, y) \in p\} \in S$
- for any three elements $p, q, r \in S$, there is a number a_{pq}^r such that for any $(x, y) \in r$, the number of $z \in X$ satisfying $(x, z) \in p$ and $(z, y) \in q$ is a_{pq}^r.

The relation matrix of this scheme is given by the points being ordered as indices, and the (i, j) entry is the relation p between those two points. Note, when $i = j$, the relation is 1_x.

Next we will define equitable partitions. Let X be a finite set, and order the elements. Then, refer to them as integers $1, 2, ..., n$. Given a subset $A \subseteq X$, we get a vector j_A where we define the i^{th} entry to be 1 if $i \in A$ and 0 if $i \notin A$. For example, j_X is the vector consisting of all 1's. Now, given some partition of X, which consists of subsets $A_1, A_2, ..., A_k$, we get a collection of vectors $j_{A_1}, j_{A_2}, ..., j_{A_k}$. Notice that the sum of these vectors is equal to j_X. Now, let S be a scheme on X. For each relation $p \in S$, we call the associated adjacency matrix for that relation σ_p. The (i, j) entry of σ_p is 1 if $(i, j) \in p$ and 0 otherwise. The partition $A_1, A_2, ..., A_k$ of X is equitable if for each relation

$p \in S$ and each $i \in \{1, 2, ..., k\}$, the vector $\sigma_p j_{A_i}$ can be written in the form $\sum_{l=1}^{k} b_{pi}^{l} j_{A_l}$ where each b_{pi}^{l} is a nonnegative integer.

3 Programming Conventions

In this section, we outline the programming conventions we used when creating our methods and algorithms in equitable_partitions.gap. Because we are computing equitable partitions of schemes, we need a data structure that represents a partition of a scheme. Miyamoto and Hanaki choose to represent their schemes as relational matrices. Given a scheme S on a set X with size $|X| = n$, the corresponding relational matrix R assumes some ordering of the elements of X. Miyamoto and Hanaki label both the relations of S and the points of X with the natural numbers; however, they choose to 0-index the relations, and 1-index the points. By convention they denote the 1_x relation by 0, so every relational matrix has a diagonal of 0s. To be consistent, we label the elements of X with the natural numbers $1, 2, \ldots, n$ with respect to the ordering given by a relational matrix. With this ordering, we can create data structures that represent partitions of schemes. To complete this project, it turns out that we needed three different partition representations.

3.1 The partition representation

The first representation we use is the same representation GAP natively uses to represent a partition of the set of elements $\{1, 2, \ldots, n\}$. GAP represents a partition of the set of n elements as a list of lists, where each sub-list contains elements from the universe we are partitioning. For example, suppose that $n = 3$ and we would like to represent the partition $\{\{1\}, \{2, 3\}\}$. As a list of lists, GAP would represent this partition as

```
[ [ 1 ], [ 2, 3 ] ]
```

In our code, as a convention we denote this representation of a partition as "partition." We use the partition representation when we convert from one representation of a partition to another.

3.2 The jvectors representation

The second representation we use encodes a partition with a collection of j vectors (the ones described in the definition of an equitable partition). GAP represents a row vector of length n as a list of length n. We represent a single j vector of length n as a 0-1 row vector of length n. A collection of j vectors together forms a partition, and we represent such a collection as a list of j vectors. Continuing with our previous example, suppose that we want to represent the partition $\{\{1\}, \{2, 3\}\}$ as a collection of j vectors. As a list of lists, we would represent this partition in GAP as

```
[ [ 1, 0, 0 ], [ 0, 1, 1 ] ]
```

In our code, as a convention we denote this representation of a partition as "jvectors." We use the jvectors representation when determining if a partition is equitable and when two partitions are isomorphic to each other.

3.3 The partition sequence representation

The third representation we use encodes a partition as a sequence of numbers. After defining and introducing partition sequences, we will show that each n-digit partition sequence corresponds to a partition of a set of size n.

Christopher French introduces partition sequences as follows. First, consider the set of all sequences of nonnegative integers of length n, where for digit i (we zero index i here), the number of instances of 0s that precede i is between 0 and i inclusive. We can order this set as follows. First, notice that every set of n digit sequences contains the all 0 sequence; this will be the first sequence in our ordering. Given a partition sequence x, to get to the next sequence, you increase x's last digit. If the last digit equals to the number of 0s that precede it, then we make the last digit a 0 and add 1 to the second-to-last digit, iteratively repeating this process until no digit is greater than the number of 0s preceding it. This sequence of partition sequences terminates when we get a sequence starting with 0 followed by $n-1$ 1s.

For example, let us consider all 3-digit sequences. We start with the all 0 sequence, 000. Then, we can increase the last digit twice without violating the number of 0s allowed to precede each digit, giving us the next two partition sequences 001 and 002. When we construct the next sequence after 002, as an intermediary step we add 1 and get 003. We see that 3 is larger than number of 0s proceeding it (2), and so we set the last digit to 0 and add 1 to the second to last digit, giving us 010. 010 has no digit in violation of the preceding 0 property, and so it is the next partition sequence in our sequence of partitions. We add 1 to the last digit, producing the partition sequence 011, which turns out to be a valid partition and is the terminating partition sequence. Then, all 3-digit sequences ordered give us

$$000,\ 001,\ 002,\ 010,\ \text{and } 011.$$

We now show that a sequence of n digit partition sequences enumerates the partitions of a set of size n by demonstrating how to convert from a partition sequence into a partition. To build the partition, you read the sequence from left to right, examining each digit one at a time. Every time you see a 0, start a new part of the partition with the corresponding digit. If you see a 1 in position i, put i in the first part. If you see a 2 in position i, put i in the second part, and so on.

For example, consider converting the partition sequence 002. We start with an empty set $\{\ \}$. The 1^{st} digit is 0, and so we create a new part containing the 1^{st} element, $\{\{1\}\}$. The 2^{nd} digit is 0, and so we create a new part containing the 2^{nd} element, $\{\{1\},\{2\}\}$. Lastly, the 3^{rd} digit is 2, and so we place the 3^{rd} element into the 2^{nd} part, giving us $\{\{1\},\{2,3\}\}$. This tells us that the partition sequence 002 corresponds to the partition $\{\{1\},\{2,3\}\}$.

We represent a n-digit partition sequence as a row vector of length m. As a row vector, we would represent the partition sequence 002 in GAP as

```
[ 0, 0, 2 ]
```

In our code, as a convention we denote this representation of a partition as a "partition sequence" or as a "sequence" (depending on the context). We will explain in detail why we needed this representation in the Algorithms section of this paper, but for now we will say that we used this representation to iterate over every possible partition of a finite set and to help us add partitions to a hash table.

3.4 Using these representations

Throughout our code, we represent partitions using partitions, jvectors, and partition sequences. We named each of our variables and methods to be clear about which representation they take as input and produce as output. For example, we have the methods `PartitionToJVectors, PartitionToSequence, JVectorsToPartition, JVectorsToSequence,` and `SequenceToPartition` which translate between the various representations of partitions. The only two methods which don't follow this rule are the `IsEquitablePartition` and `IsIsomorphic` functions, which both take jvectors as input. We chose to not use the term "jvectors" for the `IsEquitablePartition` method because we wanted the method to reflect the definition of an equitable partition; we exlude the term in the `IsIsomorphic` method for the same reason.

4 Methods

This section will give brief descriptions of the methods written and some examples of how the code could be used. Note, `as05` refers to the array of schemes of order 5 classified by Hanaki and Miyamoto.

4.1 `GroupToScheme`

`GroupToScheme` takes a group G and returns the corresponding association scheme's relational matrix. For example, if you wanted to compute the association scheme associated with S_3, then you could do the following:

```
gap> GroupToScheme(SymmetricGroup(3));
[ [ 0, 1, 2, 3, 4, 5 ],
  [ 1, 0, 3, 2, 5, 4 ],
  [ 2, 4, 0, 5, 1, 3 ],
  [ 4, 2, 5, 0, 3, 1 ],
  [ 3, 5, 1, 4, 0, 2 ],
  [ 5, 3, 4, 1, 2, 0 ] ]
```

4.2 IsEquitablePartition

`IsEquitablePartition` takes a scheme R and a partition of j vectors and returns whether or not the j vectors make up an equitable partition of R.

```
gap> IsEquitablePartition(as05[2], [ [ 1, 0, 0, 1, 0 ],
                                     [ 0, 1, 0, 0, 0 ],
                                     [ 0, 0, 1, 0, 1 ] ]);
true
```

4.3 IsIsomorphic

`IsIsomorphic` takes an automorphism group of a scheme and two sets of j vectors. It returns whether two sets of j vectors are isomorphic or not. For example, given the automorphism group of a scheme of order 5 and two equitable partitions, we can calculate the following.

```
gap> IsIsomorphic(AutomorphismGroupOfScheme(as05[2]),
                  [ [ 1, 0, 0, 1, 0 ],
                    [ 0, 1, 0, 0, 0 ],
                    [ 0, 0, 1, 0, 1 ] ],
                  [ [ 0, 1, 0, 0, 1 ],
                    [ 0, 0, 0, 1, 0 ],
                    [ 1, 0, 1, 0, 0 ] ]);
true
```

4.4 EquitablePartitions

`EquitablePartitions` takes a scheme R and computes its equitable partitions. It returns a two element vector of the total number of equitable partitions and a list of the representatives of the equivalence classes of equitable partitions. A faster, but slightly less space efficient version of this method is `EquitablePartitionsFast`. `EquitablePartitionsFast` takes a scheme R and computes its equitable partitions. It returns a two element vector of the total number of equitable partitions and a list of the representatives of the equivalence classes of equitable partitions. For example, `EquitablePartitions` and `EquitablePartitionsFast work` in the same way and could be run as follows.

```
gap> EquitablePartitionsFast(as05[2]);
[ 7,
  [ [ [ 1, 0, 0, 0, 0 ],
      [ 0, 1, 0, 0, 0 ],
      [ 0, 0, 1, 0, 0 ],
      [ 0, 0, 0, 1, 0 ],
      [ 0, 0, 0, 0, 1 ] ],
    [ [ 1, 0, 0, 1, 0 ],
      [ 0, 1, 0, 0, 0 ],
      [ 0, 0, 1, 0, 1 ] ],
```

```
    [ [ 1, 1, 1, 1, 1 ] ] ] ]
```

4.5 ComputeIsomorphicPartitions

`ComputeIsomorphicPartitions` takes a scheme R and a set of j vectors. It returns a list of all sets of j vectors isomorphic to it under the automorphism group of R. For example, given a specific set of j vectors on a scheme of order 5, we get the following.

```
gap> ComputeIsomorphicPartitions(as05[2], [ [ 1, 0, 0, 1, 0 ],
                                            [ 0, 1, 0, 0, 0 ],
                                            [ 0, 0, 1, 0, 1 ] ]);

[ [ [ 1, 0, 0, 1, 0 ], [ 0, 1, 0, 0, 0 ], [ 0, 0, 1, 0, 1 ] ],
  [ [ 1, 0, 0, 0, 1 ], [ 0, 0, 1, 0, 0 ], [ 0, 1, 0, 1, 0 ] ],
  [ [ 0, 1, 1, 0, 0 ], [ 1, 0, 0, 0, 0 ], [ 0, 0, 0, 1, 1 ] ],
  [ [ 0, 1, 0, 0, 1 ], [ 0, 0, 0, 1, 0 ], [ 1, 0, 1, 0, 0 ] ],
  [ [ 0, 0, 1, 1, 0 ], [ 0, 0, 0, 0, 1 ], [ 1, 1, 0, 0, 0 ] ] ]
```

4.6 Printing and Logging Methods

We have several methods for printing the equitable partition results in a nice format which include: `PrintEquitablePartitionsToFile`, `LogPartitionsScheme`, `LogPartitionsSchemeList`, `PrintPartitionsScheme`, and `PrintPartitionsSchemeList`. The way these methods can be used is described in detail in their documentation. For example, `LogPartitionScheme` can be used as follows to create a log file.

```
gap> LogPartitionsScheme("example.txt", as05[2],"scheme0502");
```

This command logs the equitable partitions to a newly created file, example.txt, in the format shown below.

```
#----------------------------------------
#Scheme name:
scheme0502 :=

# Number of Equitable Partitions: 7
# Number of Equivalence Classes: 3
[

# No. 1
[ [ 1, 0, 0, 0, 0 ],
  [ 0, 1, 0, 0, 0 ],
  [ 0, 0, 1, 0, 0 ],
  [ 0, 0, 0, 1, 0 ],
  [ 0, 0, 0, 0, 1 ] ],
```

```
# No. 2
[ [ 1, 0, 0, 1, 0 ],
  [ 0, 1, 0, 0, 0 ],
  [ 0, 0, 1, 0, 1 ] ],

# No. 3
[ [ 1, 1, 1, 1, 1 ] ]
];
```

4.7 Helper Methods

The aforementioned functions have several helper methods which include: `PrintMatrixToFile`, `MyZeroVector`, `ComputeCoefficients`, `VectorSpaceBasisFromPartition`, `PermuteJVector`, `CheckEquivalent`, `PermuteJVectors`, `PartitionToJVectors`, `CreateAuxiliary`, `NextPartitionSequence`, `SequenceToPartition`, `JVectorsToPartition`, `PartitionToSequence`,and `JVectorsToSequence`.

5 Algorithms

Here, the algorithms of the aforementioned methods will be described in more detail.

5.1 `GroupToScheme`

`GroupToScheme` takes a group G. It adds the elements to a dictionary where the key is some element $g \in G$ and the value is a relation from the set $\{0, 1, ..., |G| - 1\}$ assigned to that element (note that the identity element is always associated with the value 0), and these values help to determine the relations and order the group elements so they work as (i, j) indices for the relational matrix. Then, we calculate the relation matrix of the associated association scheme in a fashion similar to creating a Cayley table. To do this, we let the elements of the group be ordered and index the matrix according to this. Then, we multiply elements of the group, and insert them into the table such that the 0 relation is guaranteed to be on the diagonal. This is done by checking where the index of zero is in that particular column (i.e. the value of j), then inserting it in the row of the same index (so that we have $i = j$). Then, after computing all of the relations, we check to make sure it is an association scheme. If it isn't, we return an error; otherwise, we return the newly calculated scheme.

5.2 `IsEquitablePartition`

`IsEquitablePartition` takes a relational matrix `R` , a list of j vectors `jvectors`, and returns whether or not `jvectors` is an equitable partition of the

scheme S encoded by `R`. `IsEquitablePartition` does this by performing an exhaustive search, directly verifying that for each relation $p \in S$ and each $i \in \{1, 2, ..., k\}$, the vector $\sigma_p j_{A_i}$ can be written in the form $\sum_{l=1}^{k} b_{pi}^l j_{A_l}$ where each b_{pi}^l is a nonnegative integer. The exhaustive search works as follows. First, we use our `VectorSpaceBasisFromPartition` method to create a vector space `basis` of the set of vectors generated by `jvectors` over the rationals. Then, we compute the coefficients of $\sigma_p j_{A_i}$ with respect to our `basis`. By the definition of j vectors, if these coefficients exist, they must be nonnegative integers, and therefore these coefficients are our desired b_{pi}^l. So, we use our `ComputeCoefficients` method to determine if $\sigma_p j_{A_i}$ can be written in terms of `basis`. If they cannot, we halt the exhaustive search and return false. Otherwise, if the nonnegative numbers b_{pi}^l exist for every vector $\sigma_p j_{A_i}$, we return true. Note that this method always runs faster when it returns false. This is because it is easier to determine whether or not a partition is non-equitable as opposed to equitable.

5.3 IsIsomorphic

`IsIsomorphic` takes an automorphism group and two sets of j vectors. It starts by doing a length check on the j vectors because j vectors cannot be isomorphic if they are of different length. Then, for each automorphism in the automorphism group it checks the two sets of j vectors are equivalent under the given automorphism. It does this by using the `CheckEquivalent` method which works by permuting the j vectors in the first set of j vectors under a given automorphism, and then checking to see if it is in the second set of j vectors. `CheckEquivalent` returns true if all j vectors from the first set can be found in the second set after being permuted under the given automorphism and returns false otherwise. `IsIsomorphic` returns true if `CheckEquivalent` finds any two isomorphic j vector partitions. Otherwise, if `CheckEquivalent` always returns false, then the two j vector partitions are not equivalent under any automorphism, so `IsIsomorphic` returns false.

5.4 EquitablePartitions

To start, we will describe the important local variables which are used throughout `EquitablePartitions`. First, It takes a scheme `R`, and `n` represents its order. `numEquitablePartitions` represents the total number of equitable partitions. It starts at zero and will be incremented each time a new equitable partition is found. `representatives` represents the representatives of each equivalence class of equitable partitions. `automorphisms` represents the automorphism group of the scheme. `currentSequence` is used to iterate over all possible partitions, and it represents the current partition sequence and starts as a zero vector of length `n`.

Next, it loops through all possible partitions while the current sequence is still valid. Originally, we used GAP's `PartitionsSet` which generates all partitions of size `n` and looped through these; however, as the input schemes

grew larger the size of the set of partitions quickly became too large and used up too much memory as the Bell number grows exponentially. Thus, looping through sequences is much more memory efficient. In each iteration of the loop, it first transforms the current partition sequence into a set of j vectors. If the j vectors are an equitable partition of the given scheme, then check to see if the list of representatives is empty. If so, add it. Otherwise, check to see if any set of j vectors isomorphic to the original set of j vectors, using the `isIsomorphic` method, has already been added to the set of representatives using a boolean `previouslyDiscovered` which is only true if the partition of j vectors has already been discovered in the list `representatives`. Next, it checks to see if the j vectors have been discovered or not, and if not, then they are added to `representatives`. After this, we iterate to the next sequence using the method `nextPartitionSequence`, and this process is repeated until there are no more partitions. Finally, it returns the total number of equitable partitions and the list `representatives`.

5.5 `EquitablePartitionsFast`

To start, we will describe the important local variables which are used throughout `EquitablePartitionsFast`. First, It takes a scheme `R`, and `n` represents its order. `numEquitablePartitions` represent the total number of equitable partitions. It starts at zero and will be incremented each time a new equitable partition is found. `representatives` represents the representatives of each equivalence class of equitable partitions, and `representativeIndex` represents the current index and starts at 0. `automorphisms` represents the automorphism group of the scheme. `currentSequence` is used to iterate over all possible partitions, and it represents the current partition sequence and starts as a zero vector of length `n`. `equitablePartitionsTable` is a hash table where the keys are partition sequences and the values are the indices of the corresponding partition equivalence class representative. We use this hash table to keep track of all equitable partitions that have been discovered so far and all equitable partitions which are isomorphic to these under the automorphism group. Using this hash table, allows for quicker checking to see whether or not an equivalent partition has already been discovered, and it reduces the use of exhaustive methods like `IsEquitablePartition`.

`EquitablePartitionsFast` like `EquitablePartitions` iterates over partitions sequences while they are valid. However, the way new equitable partitions are stored and discovered is different. In each iteration of the while loop, it starts by converting the current sequence to a set of j vectors. Next, if the current sequence is in the table, continue to the next iteration. Otherwise, check if the j vectors form an equitable partition. If this is the case, add it to the list of representatives. Next, for all automorphisms in the automorphism group of the scheme compute equivalent sets of j vectors and convert them to partition sequences (note we sort the partitions in such a way that the partition sequence will always be computed to be the same). Then, add these newly created partition sequences to the hash table. If they can be added (i.e. it isn't already in

the table), we increment the total number of equitable partitions. After this, we find the next partition sequence and repeat until there are no more new partition sequences. Finally, it returns the total number of equitable partitions and the list `representatives`.

5.6 `ComputeIsomorphicPartitions`

`ComputeIsomorphicPartitions` takes a scheme `R` and a set of j vectors `jvectors`, and it computes all equivalent sets of j vectors under the automorphism group of the scheme `automorphsims`. This method uses a hash table `partitionTable` to keep track of the partitions and help remove duplicates. In the hash table, the keys are partition sequences, and the values are given by the automorphism that sends `jvectors` to that partition. `equivalenceClass` is a list that contains the entire equivalence class of isomorphic j vectors. So, for every automorphism in the automorphism group, this method permutes the j vectors under the given automorphism and tries to add them to the hash table. If they can be added then add that new permuted set of j vectors to `equivalenceClass`. Otherwise, continue to the next iteration. This is done for all automorphisms, and in the end, it returns `equivalenceClass` which is filled with isomorphic sets of j vectors.

6 Conclusion

6.1 Limitations

The most pressing limitation of our `EquitablePartitions` methods is that for a scheme S on a set X of size n, our methods visit every possible partition of a set of n elements. This means that the running time of each of these algorithms will be bounded from below by how many partitions are visited, regardless of how fast (on average) it takes to determine whether or not a specific partition is equitable. The Bell numbers tell us how many ways are there to partition a set of n elements. With the `EquitablePartitionsFast` method, we were able to compute the equitable partitions of a scheme on 10 elements in under a minute. Because Miyamoto and Hanaki have classified the association schemes up to order 34, we might use our methods to determine the equitable partitions of schemes up to order 34. But, it turns out that the Bell number grows very very fast: GAP tells us that Bell(n) for $n \in \{1, 2, \ldots, 10\}$ is

```
1, 2, 5, 15, 52, 203, 877, 4140, 21147, 115975.
```

To demonstrate that computing the equitable partitions of a scheme on 30 elements is intractable, suppose that `EquitablePartitionsFast` processes Bell(10) partitions in a minute. Further assume that for each iteration of the main loop in `EquitablePartitionsFast`, it runs in linear time (which in actuality it clearly

doesn't). Then, `EquitablePartitionsFast` processes Bell(30) partitions in

$$\begin{aligned}\text{Bell(30) / Bell(10) minutes} &= 846749014511809332450147 \text{ / } 115975 \text{ minutes} \\ &= 7.30113 \times 10^{18} \text{ minutes} \\ &= 1.38910388^{13} \text{ years.}\end{aligned}$$

Even if we could dramatically speed up how fast (on average) it takes to determine whether or not a specific partition is equitable, the total amount of partitions visited (in our case determined by the Bell number) will always be a lower bound on how fast we can compute the equitable partitions of a given scheme.

The second most pressing limitation of our `EquitablePartitions` methods is that they rely on exhaustive helper procedures. Even though the `EquitablePartitions` method uses a hash table to mitigate the cost calling the exhaustive `IsIsomorphic` method and uses the `IsEquitablePartition` more sparsely, it still relies on the `IsEquitablePartition` to disqualify non-equitable partitions. Both `EquitablePartitions` methods could be sped up with smarter and faster implementations of the `IsEquitablePartition` and `IsIsomorphic` methods.

A last limitation of our algorithms is that the `EquitablePartitionsFast` method uses more memory (the hash table) in order to speed up the isomorphism testing. When the `EquitablePartitionsFast` algorithm terminates, its hash table will contain every partition sequence corresponding to a discovered equitable partition. Even though partition sequences are relatively sparse representations of partitions of sets, running this algorithm on some scheme that is both 1) on a sufficiently large set and 2) has sufficiently many equitable partitions might use a lot of memory.

6.2 Future Work

There is much more that could be done with computing equitable partitions. Firstly, these methods could be made more efficient. The methods we wrote mainly relied on exhaustive searches, so there could be more efficient ways of looking for equitable partitions. For example, in the `EquitablePartitions` and `EquitablePartitionsFast` methods, we looked through all possible partitions, but it could be possible to iterate over the partitions in a different manner such that groups of partitions could be ruled out at once rather than looking through them one at a time. Because we know that (to an extent) the running time our our algorithms is lower bounded by the Bell number, it would be greatly beneficial if one could prove the existence of and construct much smaller class of potential partitions (closer to being polynomial in growth). `IsEquitablePartition` similarly takes an exhaustive approach to checking whether a partition is equitable or not; however, there may exist a better and more efficient approach. Another possibility for making the code more time efficient is implementing some sort of parallelism in the code. GAP has tasks which can be used in order

to add concurrency. Also, the conversions between different types of partitions could also possibly be made more efficient.

6.3 Results

We were able to calculate the equitable partitions of association schemes up to order 11. They can be found in our GitHub repository in the partition results. Note that we omit the first scheme of every order in Miyamoto and Hanaki's classification in this computation, because every partitioning of a scheme with one relation is equitable.

To further test our code, we created association schemes from groups using GroupToScheme, and tested the equitable partitions methods on them. The total number of equitable partitions matched the total number of subgroups and the number of equivalence classes of equitable partitions matched the number of conjugacy classes of subgroups for all groups that we tested. For example, we converted S_3 to an association scheme and got the following.

```
   gap> GroupToScheme(SymmetricGroup(3));
[ [ 0, 1, 2, 3, 4, 5 ],
  [ 1, 0, 3, 2, 5, 4 ],
  [ 2, 4, 0, 5, 1, 3 ],
  [ 4, 2, 5, 0, 3, 1 ],
  [ 3, 5, 1, 4, 0, 2 ],
  [ 5, 3, 4, 1, 2, 0 ] ]
```

Next, we computed the equitable partitions and got the following.

```
gap> PrintPartitionsScheme(s3, "s3");
#---------------------------------------
#Scheme name:
s3 :=

# Number of Equitable Partitions: 6
# Number of Equivalence Classes: 4
[

# No. 1
[ [ 1, 0, 0, 0, 0, 0 ],
  [ 0, 1, 0, 0, 0, 0 ],
  [ 0, 0, 1, 0, 0, 0 ],
  [ 0, 0, 0, 1, 0, 0 ],
  [ 0, 0, 0, 0, 1, 0 ],
  [ 0, 0, 0, 0, 0, 1 ] ],

# No. 2
[ [ 1, 0, 0, 0, 0, 1 ],
  [ 0, 1, 0, 1, 0, 0 ],
```

```
  [ 0, 0, 1, 0, 1, 0 ] ],

# No. 3
[ [ 1, 0, 0, 1, 1, 0 ],
  [ 0, 1, 1, 0, 0, 1 ] ],

# No. 4
[ [ 1, 1, 1, 1, 1, 1 ] ]
];
```

We ran this on several other small groups to verify that we always got the number of subroups and the number of conjugacy classes of subgroups.

References

[GAP20] The GAP Group. *GAP – Groups, Algorithms, and Programming, Version 4.11.0*, 2020.

[HM19] Akihide Hanaki and Izumi Miyamoto. Classification of association schemes with small vertices, 2019.

[Zie05] Paul-Hermann Zieschang. *Theory of Association Schemes*. Springer, 2005.