Algorithm for Creating Partitions from a Set of Subsets (ACPSS)

Andrew J. Collins

September 1, 2026

Abstract

This paper formally proves that the Algorithm for Creating Partitions from a Set of Subsets (ACPSS) correctly terminates. The purpose of the paper is to generate all possible partitions from a specially ordered set of subsets. This set of subsets is assumed to contain all singleton sets and to be pre-ordered in a special manner. The algorithm is part of an algorithm framework, which is used to find the core partitions of hedonic games. However, the algorithm is applicable to any set of subsets that obey the assumed rules.

1 Introduction

This paper formally proves that the Algorithm for Creating Partitions from a Set of Subsets (ACPSS), which is presented in this paper, terminates and, upon termination, has generated all possible partitions from a specially ordered set of subsets, and only those partitions. This algorithm is part of an algorithmic framework used to solve hedonic games (Collins, 2026).

The motivation for developing this algorithm framework is the need to numerically investigate hedonic games using Monte Carlo methods. The empirical properties of hedonic games have previously been reported, providing insight into their nature (Etemadidavan and Collins, 2021; Collins et al., 2022); however, this new algorithm framework provides a means to investigate games of much larger size.

Within the algorithmic framework, a special ordered set of subsets is used called the Subset Rational Coalition Lists (SRCL) (Collins, 2026), which is an advancement of Individually Rational Coalition Lists (IRCL), as coined by Elkind and Wooldridge (2009). However, any set of subsets can be used that obeys the properties described below; hence, the ACPSS is applicable to a wide variety of uses beyond the algorithmic framework.

This paper is arranged as follows. First, all notation is introduced and described. Using this notation, the ACPSS and its subroutines are described. Finally, the proof of ACPSS termination and correctness are provided.

2 Notation

There is a requirement to generate all feasible partitions from a generic set of subsets of $N = \{1, 2, ..., n\}$. The algorithm presented in this paper does just that. However, the algorithm is complex, and there is a need to define some of its key variables first. The specially ordered set of subsets is presented first, special indices of those subsets, and then, finally, algorithm-specific variables.

2.1 Subsets Under Consideration

Consider a partial list of subsets of N, which is represented by F. F is a vector of size $m = |F|$, and there are a number of criteria for F:

1. F is assumed to be in *reverse size order* (largest to smallest), then lexicographic order for subsets of the same size.
2. *if* $i \in N \implies \{i\} \in F$

These criteria for the set F are important for our algorithm to work. The second criterion guarantees that there exists at least one partition that can be created from elements of F, i.e., the partition created from singleton subsets. Since F is ordered, the a^{th} element can be represented as $f(a)$. It is assumed that $f(0) = \emptyset$.

2.2 Indices of F

Although each subset can be represented by a unique number using binary form, separate indices for each element of F are used, which are based on its position in the vector F. This is done for the iterative purposes of the algorithm. The set B_0 is the ordered collection of these indices:

$$B_0 = \langle 1, 2, ..., m \rangle$$

This means that the ath element of F, using B_0 indexing, can be represented as $b_0(a)$. Note that a value of zero represents the empty set; thus it is added that $b_0(0) = 0$ though not part of the set B_0.

The creation of B_0 is useful, in our algorithm, for navigating the list F through its indices. Similarly, a vector of indices J_0 is created, which is useful to navigate B_0. J_0 is a vector of some indices of B_0 such that the corresponding element of B_0 is the index for the element F that has the smallest index value for a given subset size, up to n. Hence, J_0 is a set of indices of indices (like a pointer of pointers in C++). J_0 is called the *size locator set*. It is defined as follows (remember that F is in reverse size order):

$$J_0(x) = argmin_{j \in \{1,2,...,|B_0|\}} \{j : |f(b_0(j))| \leq x\}$$

Since a partition is made up of disjoint subsets, understanding which subsets are disjoint from each other is important. In our algorithm, it is assumed that it is more efficient to predetermine which sets of F are disjoint (and not greater in order). Thus:

$$B_a = \{b \in B_0 : a < b, f(a) \cap f(b) = \emptyset\}$$

Note that B_a is a list of indices, not the subsets themselves. Note also that this means that a is excluded from the set B_a. It is possible that B_a is empty for a given a, but only if a is either the grand coalition, because all the singleton sets are included in F, or F is a singleton set a. As with the set B_0, a *size locator* set for B_a is defined:

$$J_a(x) = argmin_{j \in \{1,2,...,|B_a|\}} \{j : |f(b_a(j))| \leq x\}$$

It is important to note that the B_a are vectors whose elements are indices of F, whereas the J_a are vectors whose elements are indices of B_a. Note that $f(a)$ is not a vector but a subset. This *double indexing* is important for the algorithm navigation.

2.3 Partial Partition

In the proofs, the concept of *partial partition* is used. A partial partition is any collection of disjoint subsets; formally, a partial partition $P(d) = \{P_1, P_2, \ldots, P_d\}$:

$$P_i \cap P_j = \emptyset \quad \forall i \neq j \qquad \bigcup P_i \subseteq N$$

Within our discussion, it is assumed that $\forall P_i \in F$ and the corresponding index in B_0 is a_i.

Given a partition P, where at least one member is not a singleton, and $\forall P_i \in F$, the partial partition P^* is defined as all non-singleton members of P. Note that P might equal P^*.

Lemma 1. *Any finite partition $P_1, \ldots, P_d$ can be rearranged in reverse lexicographic order, as required by F.*

Proof. Trivial. This is just a rearrangement of the finite set of subsets □

WLOG assume $P_1, \ldots, P_d$ in order.

Symbol	Meaning
n	Number of elements
N	$\{1, 2, .., n\}$
F	A list subset of N in reversed size order, largest first, then lexicographic order. It contains all singleton subsets
$f(a)$	a th element of F
m	Number of elements in F
B_0	$(1, 2, ..., m)$
$j_0(x)$	Index number j of first value in B_0 s.t. $\|f(b_0(j))\| \leq x$
B_a	List of indexes of F, indexed by b, s.t. $a < b$, $f(a) \cap f(b) = \emptyset$, in reversed size order, largest first, then lexicographic order.
$j_a(x)$	Index number j of the first value in B_a s.t. $\|f\left(b_a(j)\right)\| \leq x$

Table 1: List of variables that need to be determined beforehand for the algorithms

Symbol	Meaning
C	A set of values from B_0 which represent the current partition or partial partition under consideration
$l(j)$	Indexes used in the algorithm, which indicate the current minimum index of $B_{c(j)}$ under consideration for the $j+1$ coalition in C. Note $C(0) = 0$.
L	$(l(0), l(1), l(2), ..., l(n-1))$
k	The last index position of the last position in C to be filled
r	Represents the number of empty spots available in the current partial partition

Table 2: List of variables used within the algorithms

2.4 Algorithm Variables

There are a number of variables used in the algorithm, which are discussed here. $C \subseteq B_0$ is a list of indexes of F, which represent the current partial partition under consideration in the algorithm:

$$if \quad a, c \in C \implies f(a) \cap f(c) = \emptyset$$

In the algorithm, elements are added to C until a partition is created. k represents the index of C for which the last position was filled. If $k = 0$, it implies that no positions have been filled. A partition must contain all elements of N within its subsets, r represents the number of unfilled elements in the current partial partition under consideration, that is:

$$r = n - |\cup_{a \in C} f(a)|$$

That r holds this property is proved below. The algorithm works by filling up C to make a partition. It is feasible that to turn a partial partition into a full partition, only singleton partitions can be used. ψ is a Boolean value that represents whether the current partial partition should be filled with singletons to create a partition. Assuming that $k+1$ is the current position in C under consideration, l_k is the current lowest index of $B_{c(k)}$ that is under consideration for position $k+1$ in C; l_k is important for iterating through the disjoint subsets for each of the given positions in C.

Note that at most only $n-1$ coalitions need to be considered in our algorithm because if n coalitions are in a partition, then that partition is made up exclusively of singleton coalitions, which are evaluated separately.

These variables discussed so far are summarized in Table 1; these values need to be determined before the start of the algorithm. A summary list of variables used internally to the algorithms is given in Table 2. Unless otherwise stated, all ordered lists are assumed to use index base 1.

Now that all the data structures and variables have been defined, the details of the algorithm are described.

3 Algorithm

This section describes, in detail, the partition generator algorithm (algorithm 2) and its subroutines. An important subroutine is the *StepBack* (algorithm 1), which is used to remove the rightmost non-zero value of C after a partition is created. As such, the *StepBack* algorithm is discussed first.

3.1 StepBack

The StepBack subroutine removes the last non-singleton coalition after a partition has been formed. Within the Partition Generator algorithm, only the non-singleton coalitions are stored and processed; if a partial partition requires singletons for completion before printing, the *AddSingletons* subroutine will do this.

Algorithm 1 StepBack Sub-routine

Require: $F \in P(P(N))$
Require: $C \in \mathbb{N}^{n-1}, L \in \mathbb{N}^{n-1}$
Require: $r \geq 0$
Require: $k \geq 1$
$r \leftarrow r + |f(c(k))|$
$c(k) \leftarrow 0$
$k \leftarrow k - 1$
$l(k) \leftarrow l(k) + 1$
$l(k+1) \leftarrow ... \leftarrow l(n-1) \leftarrow 1$ ▷ Note (1)

NOTE (1) This step is not strictly necessary as implied due to earlier steps in the *Partition Generator* Algorithm, but it is included for completeness.

After adding a coalition to C, a partition will only form if either the current partial partition is a partition (i.e., all players are allocated to a coalition), or the remaining coalitions can only be singleton coalitions. Thus, if a partition is formed, then it is not possible to add further subsets to form another partition. Hence, the most recently added coalition can be safely removed, and other possibilities compatible with the remaining members of C can be considered. The *StepBack* algorithm ensures the indexing variables are correct to enable the iterative process to happen.

The *Stepback* resets the last-added coalition of C to zero (as well as all coalitions that follow, but this is not a necessary step). The size of the new, smaller partial partition, r, is determined, and the coalition count, k, is reduced. Note that it is only needed to consider up to $n-1$ coalitions in a partition, since the only partition with n coalitions is made up exclusively of singleton coalitions, which are generated separately.

Most importantly, the tracking index, $l(k)$, for the now last coalition is increased by one. Since, as mentioned, a partition only forms when no other non-singleton coalitions can be added to the current partial partition, increasing the index tracking ensure that the same partition is not generated multiple times.

3.2 Other Subroutines

There are three further subroutines used in the main algorithm, which are not explicitly defined:

- *AddSingletons(C)*: This subroutine adds the singleton coalitions to a partial partition to make it a partition
- *PrintPartition(C)*: This subroutine prints out the partition in whatever form is required
- CheckSubsetCompatibility(C, c): This subroutine returns true is c can be added to C and it still remains a partial partition (i.e., $f(c)$ is disjoint to all elements of $\cup_{a \in C} f(a)$).

3.3 Partition Generator Algorithm

The main *Partition Generator* Algorithm is described in Algorithm 2. The algorithm starts with the largest coalition in F and continues until a partial partition that starts with a singleton coalition,($c(1) \leftarrow b_0(1)$), is about to be considered ($l_0 < j_0(1)$).

Algorithm 2 Partition Generator Algorithm

Require: $n > 1$
  $k \leftarrow 1$
  $l(0) \leftarrow ... \leftarrow l(n) \leftarrow 1$
  $c(0) \leftarrow 0$ ▷ Note (1)
  $c(1) \leftarrow B_0(1)$
  $c(2) \leftarrow ... \leftarrow c(n) \leftarrow 0$ ▷ Note (2)
  $r \leftarrow n - |f(c(1))|$
  while $l(0) < j_0(1)$ **do**
    $\psi \leftarrow true$
    if $r > 1$ **then**
      $u \leftarrow max\{j_{c(k)}(r), l(k)\}$
      $v \leftarrow j_{c(k)}(1) - 1$
      if $u \leq v$ **then**
        for w = u to v **do**
          if CheckSubsetCompatibility$(C, b_{c(k)}(w))$ **then**
            $c(k+1) \leftarrow b_{c(k)}(w)$
            $l(k+1) = 1$
            $r \leftarrow r - |f(c(k+1))|$
            $k \leftarrow k + 1$
            $\psi \leftarrow false$
            $w \leftarrow v + 1$
          end if
        end for
      end if
    else if $r = 0$ **then**
      PrintPartition
      StepBack
      $\psi \leftarrow false$
    end if
    if ψ **then**
      AddSingletons
      PrintPartition
      StepBack
    end if
  end while
  AddSingletons
  PrintPartition

NOTE (1) Zero means the empty partition case (i.e., no subsets are allocated to the partition yet). Note (2) Could limit to $n-1$ because the partition of singleton sets is considered separately

During its main loop, the algorithm adds compatible coalitions to the current partial partition, C, until either all players are accounted for in coalitions or only adding singletons is possible (remember that the SRCL contains all singleton coalitions). Once this stage is reached, singletons are added, and the resultant partition is printed out for evaluation. Finally, the *StepBack* algorithm is implemented so the next possible partition can be determined and considered.

By compatible, it means coalitions that are disjoint from all other existing coalitions, in the partial coalition, and the coalition's size is equal to or smaller than all those in the current partial partition. The algorithm only considers the next coalitions that are compatible from the last added coalition to the current partial partition. This is done by iterating through $B_{c(k)}$. As only the considered coalition will be definitely compatible with the last added coalition, there is a need to check it against all the other coalitions currently in C; this is done using the *CheckSubsetCompatibility* subroutine. Also, there is little point in checking a coalition if its size is larger than the number of remaining players; hence, lists $J_{c(k)}$ are used to ensure coalitions of size r or smaller are considered.

The last two lines of the algorithm enable the singleton partition to be printed out. A complete proof that the ACPSS algorithm does indeed print out all possible partitions, generated for F, is provided below.

4 Proof of ACPSS

This section provides a formal proof that the *Algorithm for Creating Partitions from a Set of Subsets* (ACPSS) does indeed generate all possible partitions of F.

The formal proof is deconstructed into a series of lemmas, propositions, and theorems. The results build upon each other until the final theorem is presented. Initially, the lemmas are focused on properties and boundaries of the variables C and r from the algorithm. This is used to prove a theorem that the algorithm will terminate, assuming a finite set F is considered.

Finally, using an inductive approach, it is shown that each possible P will be printed out by the algorithm.

4.1 Lemmas and Propositions

There are a number of propositions that are used in the main theorem proofs. These propositions mainly represent qualities of the various variables in the algorithm. These propositions require some new variables, not present in the original algorithm, to be defined.

4.1.1 The value of $c(.)$

Consider $t \in \mathbb{Z}$ tracks changes to k. A $t \leftarrow t+1$ can be placed directly in the algorithm; one below the $k \leftarrow k+1$ in the main algorithm and once below the $k \leftarrow k-1$ in the *StepBack* subroutine. Initially, t is set to 1. Let k_t be the value of k at step t.

Note that all changes to $c(.)$ are accompanied by a change in k, so therefore, a change with t also. This means we can index these changes with t: C_t, k_t

Lemma 2. $c_t(j) = 0 \quad \forall j > k_t$

Corollary. $c(j) = 0 \quad \forall j > k$

Proof. By induction.
Case $t = 1$:
$k = 1 \Rightarrow C(2) = \cdots = C(n) = 0$ by initialization.

Case: if true for t, what about the next step?
There are only three possibilities:

ⓐ Algorithm terminates before next change in k

ⓑ $t \leftarrow t+1$ after $k \leftarrow k+1$ in the main algorithm

ⓒ $t \leftarrow t+1$ after $k \leftarrow k-1$ in the *StepBack* subroutine

Case ⓐ – technically, no t+1 step, but included for completeness. $c_t(\cdot)$ only changes if accompanied by a change in k_t. $\therefore$ no change in k_t means no change in $c_t(\cdot)$.

$$c_t(j) = 0 \quad \forall\, j > k_t = k_{t+1}$$

Case ⓑ – $k \uparrow$ in *main* algorithm. Based on an increase in t:

$$k_{t+1} = k_t + 1 \quad \text{and only } c_{t+1}(k_t{+}1) > 0 \text{ changes during this step}$$

Since $c_t(j) = 0\ \forall j > k_t$ previously,

$$c_t(j) = 0 \quad \forall\, j > k_t{+}1 = k_{t+1}$$

Case ⓒ – $k \downarrow$ in *StepBack* subroutine.

$$k_{t+1} = k_t - 1 \quad \text{and only change in } C \text{ is} \quad c_{t+1}(k_t) = 0 \quad \text{and} \quad c_t(j) = 0\ \forall\, j > k_t$$

$$\therefore \quad c_{t+1}(j) = 0 \quad \forall\, j > k_t - 1 = k_{t+1}$$

□

The values of the other $c(.)$ can also be determined using a similar approach:

Lemma 3. *Given* $n > 0$
$c_t(j) > 0 \quad \forall j : 0 < j \leq k_t$

Corollary. $c(j) > 0 \quad \forall j : 0 < j \leq k$

Proof. By induction, the t notation is interjected.
Case $t = 1$:
$k_1 = 1$, since all singletons belong to F, $F \neq \emptyset$ then $b_0(1) > 0$, $\therefore c_1(1) > 0$.
Case true for t, what about $t+1$?
There are only three possibilities:

ⓐ Algorithm terminates before next change in k

ⓑ $t \leftarrow t+1$ after $k \leftarrow k+1$ in the main algorithm

ⓒ $t \leftarrow t+1$ after $k \leftarrow k-1$ in the *StepBack* subroutine

Case ⓐ – technically, no t+1 step, but holds as in lemma 2.

Case ⓑ – $t \leftarrow t+1$ after $k \leftarrow k+1$ in the main algorithm. $k_{t+1} = k_t + 1$ and only $c_{t+1}(k_t + 1)$ changes from its value in C_t. Since $w \leq j_{c_t(k_t)}(1) \implies b_{c_t(k_t)}(w) > 0$, by definition of $J_{c_t(k)}$ and ordering of F (since must represent a non-singleton subset).

$$\therefore \quad c_{t+1}(k_{t+1}) > 0$$

Since no other c_{t+1} changes.

$$c_{t+1}(j) > 0 \quad \forall j : 0 < j \leq k_{t+1}$$

Case ⓒ – $k \downarrow$: $k_{t+1} = k_t - 1$ and only $c_{t+1}(k_t - 1) = 0$ changes from it value in C_t.

$$\therefore \quad c_{t+1}(j) > 0 \quad \forall j : 0 < j \leq k_t - 1 = k_{t+1}$$

□

Lemma 4. *C is always a Partial Partition.*

Proof. The initial partial partition has one subset $f(c(1))$. A new c is added to C only if it does not intersect with any other member of C; so the partial partition status is maintained at each *CheckSubsetCompatibility* step. □

4.1.2 Final Singleton Partition Printed Out

Proposition 1 (Singleton Partition Output)**.** *The algorithm will print out the singleton partition after the final AddSingleton line is reached.*

Proof. There are two cases to reach the final *AddSingleton* subroutine:

ⓐ Never enter the **while** loop, because $l(0) \not< \jmath_0(1)$

ⓑ After exiting the **while** loop

Case ⓐ: Initially, $l(0) = 1$ so for this to occur, $\jmath_0(1) = 1$, $\therefore$ all partitions are singletons by definition of ordering of F and J_0.

$$\therefore \quad c(1) = b_0(1) = 1 \quad \text{and} \quad f(1) = (1)$$

So adding further singletons adds the rest of the singletons to the Partition.

Case ⓑ: Can only exit the **while** loop after $l(0)$ increases *(note:* $\jmath_0(1)$ *is fixed).*
Can only increase $l(.)$ in *StepBack* subroutine, where $l(0)$ can only increase when $k = 0$.

$$\therefore \quad \text{by lemma 2} \quad c(j) = 0 \quad \forall\, j > k = 0$$

$\therefore C$ is empty and *AddSingleton* subroutine adds all singletons before *PrintParition*. □

Lemma 5. *If* $C = \langle 0, 0, \ldots, 0 \rangle$*, then*

$$\mathit{CheckSubsetCompatibility}(C, c) = \textsc{True} \quad \forall\, c \in B_0$$

Proof. Trivial, since $f(c) \in F$, therefore $\langle f(c) \rangle$ is a Partial Partition. □

4.1.3 Properties of r

We can index both r and c, by t, as we did previously for k. Changes to t are assumed to occur around the k change steps in the algorithm.

Lemma 6.

$$r_t = n - |f(c_t(1))| - \cdots - |f(c_t(n))|$$

Corollary. $r = n - |f(c(1))| - \cdots - |f(c(n))|$

Proof. By induction, using t.
Case: $t = 1$
$r_1 = n - |f(c(1))|$ and $C = \langle c(1), 0, 0, \ldots, 0 \rangle$. Since $f(0) = \emptyset$, then $|f(0)| = 0$:

$$\begin{aligned} r_1 &= n - |f(c_0(1))| - |f(0)| - \cdots - |f(0)| \\ &= n - |f(c_0(1))| - |f(c_0(2))| - \cdots - |f(c_0(n))| \end{aligned}$$

Case: Given true for t, show $t + 1$: Note that r_t only changes around changes of k_t (and t) $\therefore$ Only three possibilities:

ⓐ Algorithm terminates before next change in r

ⓑ $r_{t+1} \leftarrow r_t - |f(c_t(k_t + 1))|$

ⓒ $r_{t+1} \leftarrow r_t + |f(c_t(k_t))|$

Case ⓐ – technically, no t+1 step, so no change in r.

Case ⓑ – Before change in r, $c_{t+1}(k_t{+}1)$ added to C_t to make new $C_{t+1} = \langle c_t(1), c_t(2), \ldots, c_t(k_t), c_{t+1}(k_t{+}1), c_t(k_t+2), \ldots, c_t(n)\rangle = \langle c_{t+1}(1), c_{t+1}(2), \ldots, c_{t+1}(k_t), c_{t+1}(k_t+1), c_{t+1}(k_t+2), c_{t+1}(n)\rangle$, since no other c_t changes at this step. Previous to the addition of $c_{t+1}(k_t+1)$, $c_t(k_t+1) = 0$, by lemma 2, $\therefore |f(c_t(k_t+1))| = 0$.

$$\begin{aligned} r_{t+1} &\leftarrow r_t - |f(c_{t+1}(k_t+1))| \\ &= n - |f(c_t(1))| - |f(c_t(2))| - \cdots - |f(c_t(k_t))| - |f(c_t(k_t+1))| - |f(c_t(k_t+2))| - \\ &\cdots - |f(c_t(n))| - |f(c_{t+1}(k_t+1))| \\ &= n - |f(c_{t+1}(1))| - |f(c_{t+1}(2))| - \cdots - |f(c_{t+1}(k_t))| - 0 - |f(c_{t+1}(k_t+2))| - \\ &\cdots - |f(c_{t+1}(n))| - |f(c_{t+1}(k_t+1))| \\ &= n - |f(c_{t+1}(1))| - |f(c_{t+1}(2))| - \cdots - |f(c_{t+1}(n))| \end{aligned}$$

By lemma 2, $c_t(k_t+1) = 0 \qquad \therefore |f(c_t(k_t+1))| = 0$.

Case ⓒ – $c_t(k_t)$ removed from C_t to make $C_t = \langle c_t(1), c_t(2), \ldots, c_t(k_t-1), 0, c_t(k_t+1), \ldots, c_t(n)\rangle$. $k_{t+1} = k_t - 1$. Note that $c_{t+1}(k_t) = 0$

$$\begin{aligned} r_{t+1} &= r_t + |f(c_t(k_t))| \\ &= n - |f(c_t(1))| - |f(c_t(2))| - \cdots - |f(c_t(k_t-1))| - |f(c_t(k_t))| - |f(c_t(k_t+1))| - \\ &\cdots - |f(c_t(n))| + |f(c_t(k_t))| \\ &= n - |f(c_t(1))| - |f(c_t(2))| - \cdots - |f(c_t(k_t-1))| - 0 - |f(c_t(k_t+1))| - \\ &\cdots - |f(c_t(n))| \\ &= n - |f(c_{t+1}(1))| - |f(c_{t+1}(2))| - \cdots - |f(c_{t+1}(k_t-1))| - |f(0)| - |f(c_{t+1}(k_t+1))| - \\ &\cdots - |f(c_{t+1}(n))| \\ &= n - |f(c_{t+1}(1))| - |f(c_{t+1}(2))| - \cdots - |f(c_{t+1}(k_t-1))| - |f(c_{t+1}(k_t))| - |f(c_{t+1}(k_t+1))| - \\ &\cdots - |f(c_{t+1}(n))| \end{aligned}$$

□

4.1.4 Algorithm Termination

A new variable σ is introduced to help demonstrate that the algorithm will terminate. First, the limits of l are looked at.

Lemma 7. *The maximum value of* $l(i) \le m, \quad$ *for* $m > 1$, $n > 1$.

Proof. Consider i such that $l(i)$ increases to m. Therefore, $k = i$, at that increase which occurs in the *StepBack* algorithm.

This can only occur in the *StepBack* algorithm.

Consider the next pass on the main algorithm loop: $u \ge m$. But:

$$j_{c(k)}(1) < m \qquad \text{since } |B_{c(k)}| < m$$

Therefore, since $v = J_{c(k)} - 1 < m = u, \implies$ skip the **"for"** loop, resulting in ψ remains *true* and a *StepBack* subroutine is instigated:

$$k \to k-1 \quad \text{and} \quad l(i) = 1$$

$\therefore l(i)$ cannot increase beyond m. □

Next, we introduce a standard result from number theory, specifically, the Fundamental Theorem of Numeration. Note that we are using variables here that do not relate to any other variables used so far, but are used for this standalone proof only.

Proposition 2. $\forall\, a_i < x$, *with* $x \geq 2$, $x, a_i \in \mathbb{N}$, $n \geq 1$*:*

$$x^n > a_0\, x^{n-1} + a_1\, x^{n-2} + \cdots + a_{n-1}$$

Proof.

$$\begin{aligned} a_0\, x^{n-1} + a_1\, x^{n-2} + \cdots + a_{n-1} &\leq (x-1)x^{n-1} + (x-1)x^{n-2} + \cdots + (x-1) \\ &= x^n - 1 \\ &< x^n \end{aligned}$$

□

The analysis now reverts to using algorithm-defined variables.
This proposition is the reason why the decimal numerical system is possible, and it is using that concept that a maximum value of σ is determined

$$\sigma = l(0) \times (m+1)^n + l(1) \times (m+1)^{n-1} + \cdots + l(n)$$

Now, properties of σ can be shown.

Lemma 8. *The maximum value of* σ *satisfies* $\sigma < m\,(m+1)^{n+1}$.

Proof. By lemma 7:

$$\begin{aligned} \sigma &\leq m(m+1)^n + m(m+1)^{n-1} + \cdots + m \\ &= m\left((m+1)^n + (m+1)^{n-1} + \cdots + 1\right) \end{aligned}$$

By Proposition 2:

$$\sigma < m(m+1)^{n+1}$$

□

Lemma 9. σ *is strictly increasing in the algorithm.*

Proof. Changes to σ only occur when there are changes to $l(.)$. Consider changes to $l(\cdot)$; these only happen in *StepBack* subroutine. In this step, l(k) increases by one. Let's call the new σ value σ^*. With this increase, it does not matter that $l(k+1), \ldots, l(n-1)$ are reduced to 1, because even if each $l(k+1), \ldots, l(n)$ was at its maximum m (by lemma 7), proposition 2 show us that:

$$(m+1)^{n-k} \;>\; m\,(m+1)^{n-(k+1)} + \cdots + m$$

$$\begin{aligned} \sigma^* &= l(0) \times (m+1)^n + \cdots + (l(k)+1) \times (m+1)^{n-k} + 1 \times (m+1)^{n-k-1} + \cdots + (m+1) + 1 \\ &= l(0) \times (m+1)^n + \cdots + l(k) \times (m+1)^{n-k} + 1 \times (m+1)^{n-k} + 1 \times (m+1)^{n-k-1} + \cdots + (m+1) + 1 \\ &> l(0) \times (m+1)^n + \cdots + l(k) \times (m+1)^{n-k} + (m\,(m+1)^{n-k-1} + \cdots + m) + 1 \times (m+1)^{n-k-1} + \\ &\quad \cdots + 1 + 1 \\ &= l(0) \times (m+1)^n + \cdots + l(k) \times (m+1)^{n-k} + (m+1) \times (m+1)^{n-k-1} + \cdots + (m+1) + 1 \\ &> l(0) \times (m+1)^n + \cdots + l(k) \times (m+1)^{n-k} + l(k+1) \times (m+1)^{n-k-1} + \cdots + l(n) \quad \text{by lemma 7} \\ &= \sigma \end{aligned}$$

Hence, σ increases. □

Theorem 3. *The algorithm terminates.*

Proof. The *StepBack* algorithm is always reached in finite steps, since $|f(c(k+1))| \geq 1$ and, thus, r is decreased each loop. In every finite number of steps σ is increasing (by Lemma 9). So in finite steps σ reaches, at most, the maximum given by Lemma 8, and therefore the algorithm terminates. □

4.1.5 Partial Partition

This section's lemmas and propositions are used to show that, given a partial partition, it is always reached. First, we show that the starting subset in the partition is reached.

Lemma 10. *In the algorithm, every value of $l(0)$ is considered from $l(0) = 1$ to $J_0(1)$.*

Proof. The algorithm terminates, by Theorem 3 so it reaches $J_0(1)$; it starts at 1. The only change in $l(0)$ occurs in the *StepBack* subroutine, which increments in steps of one, so it passes through all values. □

Lemma 11. *b_{P_1} will be a member of C at some point.*

Proof. Say P_1 is the b_{P_1} value in F, with $1 \leq b_{P_1} < j_0(1)$ since $|P_1| > 1$, so $b_{P_1} \in B_0$.

Consider the case when *StepBack* turns $l(0) = b_{P_1}$ by lemma 10. Note also that $k = 0$ at this point.

$$r = n > |P_i| \quad \text{and} \quad C = (0, 0, \ldots, 0) \quad \text{by Lemma 2}$$

$$\Rightarrow \quad j_0(n) \leq b_{p_i} \quad \text{because } j_0(n) \text{ is min and } |P_i| \leq n$$

(Remember: subset ordering is in reverse size order.)

After the *StepBack*, a new loop is started in the main algorithm:

$$u = b_{p_1}, \qquad b_{p_1} \leq j_0(1) - 1 = v \quad \text{because all singletons are in } F$$

First, $w = b_{p_1}$, and, by Lemma 5,

$$\textit{CheckSubsetCompatibility}(\emptyset,\, b_0(w)) = \textsc{True}$$

$$\therefore \quad C(1) \to B_0(w) = B_0(b_{p_1})$$

$$\therefore \quad P_1 \text{ is reached.}$$

□

Lemma 12. *If $\textit{CheckSubsetCompatibility}(C, c) = \textsc{True}$, then $|f(c)| \leq r$.*

Proof. By Lemma 6:

$$r = n - |f(c(1))| - \cdots - |f(c(k))|$$

Suppose for contradiction that $|f(c)| > r$. Then:

$$|f(c(1))| + \cdots + |f(c(k))| + |c| > n$$

However:

$$\bigcup c(i) \cup c \;\subseteq\; N$$

$$\left|\bigcup c(i) \cup c\right| \leq |N| = n$$

Since the sets are disjoint:

$$|f(c(1))| + \cdots + |f(c(k))| + |f(c)| \leq n \qquad \text{contradiction}$$

□

This lemma says that if a compatible subset then it will be visited.

Lemma 13. *Given a newly created $C(k^*)$ with $l(k^*) = 1$, $k^* > 0$.*
If $\exists\, l^* \geq 1$ such that

$$b_{c(k)}(l^*) = b^* \qquad \left|f\left(b^*\right)\right| > 1$$

and CheckSubsetCompability$(C,\, b^*) =$ TRUE,
then $l(k^*)$ will reach l^* and $c(k^*+1) = b^*$.

Proof. The algorithm only terminates when $k = 0$, and k^* only reduces by steps of one in the *StepBack* algorithm.

$\therefore$ Just need to show that *StepBack* is never invoked, at k, until $l(k^*) \geq l^*$.

There are three cases in which *StepBack* is invoked:

ⓐ $r = 0$

ⓑ $r = 1$

ⓒ $\psi = True$

Since $CheckSubsetCompatibility(C, b^*) = True \implies |f(b^*)| \leq r$, by lemma 12.

Case ⓐ: Invoked with $r = 0$.

Since $C(k') = 0\ \forall\, k' > k^*$ by lemma 2, and, by lemma 6,

$$r = n - |c(1)| - \cdots - |c(k^*)|$$

which is the same value of r as at the start, since $c(1), \ldots, c(k^*)$ have not had the opportunity to change because $k^* + 1 \geq k^*$, but by premise:

$$\therefore \quad r \geq \left|f(b^*)\right| > 1 > 0 \qquad \textit{Contradiction.}$$

Case ⓑ: Similar argument for $r = 1$.

Case ⓒ: Invoked by $\psi =$ TRUE.

This can only occur if no *CheckSubsetCompatibility* is found. However,

$$\textit{CheckSubsetCompatibility}(C,\, b^*) = \textsc{True} \ \Rightarrow\ F(c(k^*)) \cap F(b^*) = \emptyset$$

$$\therefore \quad b^* \in B_{c(k^*)}$$

Since $|f(b^*)| > 1$ and all singletons are in $B_{c(k)}$:

$$l^* < j_{c(k)}(1) \quad \therefore \quad l^* \leq J_{c(k)}(1) - 1$$

and

$$l^* \geq j_{c(k)}(r) \quad \text{otherwise would not be compatible}$$

$\therefore$ For $w \neq b^*$, l^* must be out of bounds, i.e. $l^* > v$ or $l^* < u$ but $l^* \leq J_{c(k)}(1) - 1 = v$, and:

$$u \geq l^* \ \Rightarrow\ \max\left(j_{c(k*)}(r),\ l(k*)\right) \geq l^* \ \Rightarrow\ l(k*) \geq l^*$$

Since $l(k)$ increments by one, $l(k)$ must have been l^* at some point. □

Lemma 14. *If $P(1), \ldots, P(d)$ are reached in the algorithm, then $P(d+1)$ is reached.*

Proof. Since $P(1), \ldots, P(d+1)$ form a partial partition, by definition:

$$\textit{CheckSubsetCompatibility}\big(\{a_1, \ldots, a_d\},\ a_{d+1}\big) = \textsc{True}$$

Since $P(d+1)$ is not a singleton and $|P(d+1)| > 1$:

$$\therefore \quad |f(a_{d+1})| > 1$$

Set

$$r = n - |P(1)| - \cdots - |P(d)|$$

Since partial partition: $r > |P(d+1)|$

Since $P(d) \cap P(d+1) = \emptyset$:

$$\Rightarrow \quad a_{d+1} \in B_{p(d)}$$

$$\therefore \quad \exists\, l_{a_{d+1}} \quad \text{s.t.} \quad b_{p(d)}\big(l_{a_{d+1}}\big) = a_{d+1} \qquad \therefore \quad l_{a_{d+1}} \geq 1$$

Since $P(1), \ldots, P(d)$ are reached: $c(k^*) = a_d$ for some k^*, and $l(k^*) = 1$.
$\therefore$ Apply lemma 13 with $b^* = a_{d+1}$. □

Lemma 15. *Given k, let $\hat{l}(k)$ be such that*

$$\hat{l}(k) = \max\big\{b : \textit{CheckSubsetCompatibility}(C, b) = \textit{True}, |f(b)| > 1\big\}$$

Then $\hat{l}(k)+1$ is reached by $l(k)$.

Proof. By lemma 13, $\hat{l}(k)$ reached $\therefore$, at that point, it would increase k to $k+1$. For $\hat{l}(k)+1$ not to be reached implies *StepBack* at k occurs before *StepBack* subroutine occurs at $k+1$, which would increase $l(k)$ by one. However, since $b_{c(k)}(w)$ is valid for $w = \hat{l}(k)$, an increase in k occurs.

Since the algorithm terminates, shown by theorem 3, through an increase in $l(k)$ when $k = 0$, so this implies another *StepBack* subroutine at $k+1$ occurs as only time, in algorithm, k decreases:

$$\therefore \quad \hat{l}(k) \uparrow \text{to } \hat{l}(k)+1$$

□

Lemma 16. *If P^* is reached in the algorithm, then P is printed out.*

Proof. Consider when the final member of P^* is added to C, at step k.

$$\therefore \quad k = d, \quad l_k = 1$$

Then either:

Case ⓐ:

$$\exists\ \max \hat{b} \in B_{p(k)} \quad \text{s.t.} \quad \textit{CheckSubsetCompatibility}(C,\ \hat{b}) = \textsc{True}$$

Case ⓑ:

$$\nexists\ \hat{b} \in B_{p(k)} \quad \text{s.t.} \quad \textit{CheckSubsetCompatibility}(C,\ \hat{b}) = \textsc{True}$$

Case ⓐ:
Since $f(\hat{b})$ is compatible, by lemma 12:

$$|f(\hat{b})| \leq r$$

Consider $\hat{l}$ s.t. $b_{c(k)}(\hat{l}) = \hat{b}$, where $\forall\, b \in B_{p(k)},\ |f(b)| > 1$.
Then by lemma 13, $l(k)$ will reach $\hat{l}$ and $C(k+1) = \hat{b}$.

Now $\hat{l}+1$ is reached by Lemma 15 and $c(k+1)=0$ at the *StepBack* subroutine:

$$\nexists\ w \quad \text{s.t.} \quad \mathit{CheckSubsetCompatibility}\big(C,\, b_{c(k)}(w)\big) = \textsc{True}$$

$$\therefore \quad \psi \text{ remains } \textsc{True}$$

$\therefore$ add singleton and P is printed out.

Case ⓑ:
Same as in $\hat{l}+1$ above. □

4.1.6 Major Proofs

This section now brings together all the proofs.

Proposition 4. *The algorithm prints out the singleton partition.*

Proof. Since the algorithm terminates, by Theorem 3, this occurs due to Proposition 1. □

Proposition 5. P^* *is reached*

Proof. By induction on d, initial case: $d=1$: P^* reached by lemma 11.

Inductive step: If feasible, partial partition reaches $P_1,\ldots,P_d$, then $P_1,\ldots,P_{d+1}$ is reached, by Lemma 14

□

Theorem 6. *Given any feasible partition of F, it is printed out by the algorithm.*

Proof. If P is a collection of singleton sets, then printed by Proposition 1. Otherwise, assume that P contains at least one non-singleton subset. By proposition 5, P^* reached. Then P is printed by Lemma 16. □

---

5 Conclusions

The paper presents and proves that the *Algorithm for Creating Partitions from a Set of Subsets* (ACPSS) correctly terminates, i.e., after it has generated all possible partitions of F. On closer inspection of the algorithm, there are a number of numerical efficiencies that could be made; for example, the algorithm values *l(k)* could be updated to take into account of the subsets that previously failed the *CheckSubsetCompatibility*. However, these improvements add complexity, and they would require a more advanced proof, which is left to future work.

References

A. J. Collins. Subset rational coalitions lists: An improved algorithm for finding the core partition of hedonic games. *mimeo*, pages 1–22, 2026.

A. J. Collins, S. Etemadidavan, and W. Khallouli. Generating empirical core size distributions of hedonic games using a monte carlo method. *International Game Theory Review*, 24(3):1–28, 2022.

E. Elkind and M. Wooldridge. Hedonic coalition nets. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems-Volume 1*, pages 417–424, 2009.

S. Etemadidavan and A. J. Collins. An empirical distribution of the number of subsets in the core partitions of hedonic games. *Operations Research Forum*, 2(4):65, 2021.

