FACS: Fast code-based Algorithm for Coalition Structure Generation

Paper ID: 1717

Abstract

Finding the optimal coalition structure is an NP-complete problem that is computationally challenging even under quite restrictive assumptions. In this paper, we propose a new algorithm for the Coalition Structure Generation (CSG) problem that can be run with more than 28 agents while using a complete set of coalitions as input (i.e. all the $2^n - 1$ possible coalitions for a set of n agents). The current state-of-the-art limit for exact algorithms is 27 agents. Our algorithm uses a novel representation of the search space of coalition structures and a new code-based search technique. We propose an effective heuristic search method to efficiently explore the space of coalition structures using our code based technique and show that our method outperforms existing state-of-the-art algorithms by multiple orders of magnitude while providing good quality solutions.

1 Introduction

Coalition formation has received considerable attention in recent research work, and its usefulness has been proved in many applications such as e-commerce (Tsvetovat and Sycara 2000), distributed sensor networks (Dang et al. 2006), etc. In this paper, we propose a new algorithm for the coalition structure generation problem that can be run with more than 28 agents while using a complete set of coalitions as input (i.e. all the $2^n - 1$ possible coalitions for a set of n agents). The fastest exact algorithms to date for solving the coalition structure generation problem are hybrid solutions called ODP-IP (Michalak et al. 2016) and ODSS (Changder et al. 2020) that combine IDP (Rahwan and Jennings 2008) and IP (Rahwan et al. 2009). ODP-IP and ODSS are very efficient for solving many problem instances when the number of agents is fewer than 27. In case there is enough time to perform the calculations required to find the exact result, exact algorithms are still practical. However, in some cases, when agents do not have sufficient time, an approach that gives good enough quality solutions, within a reasonable time, is more valuable (Wu and Ramchurn 2020; Farinelli et al. 2013). (Wu and Ramchurn 2020) showed that their proposed algorithm CSG-UCT outperforms C-Link (Farinelli et al. 2013). Subsequently, we compare FACS with CSG-UCT and show their results. The contributions of this paper are:

1. We propose a novel search method of coalition structures. It consists of an innovative heuristic algorithm (FACS) and a novel representation of the search space.
2. FACS can be run with more than 28 agents with a complete set of coalition values as input, where the limit of exact algorithms is 27 agents [1].
3. We show that our method, which combines an original heuristic and a novel search space representation is faster than the state-of-the-art algorithms by several orders of magnitude.

The rest of this paper is structured as follows. Section 2 formalizes the coalition structure generation problem and introduces some preliminaries. Section 3 presents our novel representation of the search space, while Section 4 details our coalition structure generation algorithm (FACS). Section 5 provides the results of the empirical evaluation. Finally, Section 6 concludes the paper.

2 CSG Problem formulation and preliminaries

The coalition structure generation problem is defined by a set $\mathcal{A} = \{a_1, a_2, ..., a_n\}$ of n agents where the efficiency of any subset of $\mathcal{A}$ is determined by a characteristic function v that assigns a real value to every subset of $\mathcal{A}$ called a coalition. A coalition structure $\mathcal{CS}$ is a partition of the set of agents $\mathcal{A}$ into disjoint coalitions $\mathcal{CS} = \{\mathcal{C}_1, \mathcal{C}_2, ..., \mathcal{C}_k\}$, where $k = |\mathcal{CS}|$ and $\forall \mathcal{C}_i \in \mathcal{CS}$, $\mathcal{C}_i \neq \emptyset$ and $\mathcal{C}_i \in p(\mathcal{A})$, a set of all the possible coalitions in $\mathcal{A}$. $\mathcal{CS}$ satisfies the following constraints: $\bigcup_{j=1}^{k} \mathcal{C}_i = \mathcal{A}$ and for all $i, j \in \{1, 2, ..., k\}$ where $i \neq j$, $\mathcal{C}_i \cap \mathcal{C}_j = \emptyset$. The set of all coalition structures is $\Pi(\mathcal{A})$. The value of a coalition structure $\mathcal{CS}$ is assessed as the sum of the values of its composing coalitions: $V(\mathcal{CS}) = \sum_{\mathcal{C} \in \mathcal{CS}} v(\mathcal{C})$. The coalition structure generation problem takes as input a set of agents $\mathcal{A}$ and aims at finding the optimal coalition structure $\mathcal{CS}^* \in \Pi(\mathcal{A})$, where $\mathcal{CS}^* = \arg\max_{\mathcal{CS} \in \Pi(\mathcal{A})} V(\mathcal{CS})$.

[1] Note that the code of the algorithm and the data sets are provided in the aaai supplementary material.

3 Novel Search Space Representation

First, let us recall the principle of the integer partition (IP) graph (Rahwan et al. 2007), which divides the space of all the coalition structures into subspaces that are each represented by an integer partition of n. For instance, for $n = 4$ agents, the set of integer partitions is: $\{[4], [1,3], [2,2], [1,1,2], [1,1,1,1]\}$. Each partition of n sums to n. The size of a partition $|\mathcal{P}|$ denotes the number of parts it contains. For example, the size of the partition $[1,1,2]$ is 3 because it contains 3 parts. In the IP graph, each partition of n is represented by a node, where two adjacent nodes are connected if and only if the partition in level l can be reached from another partition in level $l-1$ by splitting only an integer. Figure 1 shows a four-agent example of an integer partition graph. In this graph, each partition $\mathcal{P}$ represents a set of coalition structures in which the size of the coalitions matches the parts of $\mathcal{P}$. For example, the node [1,1,2] represents all coalition structures that contain two coalitions of size 1 and one coalition of size 2.

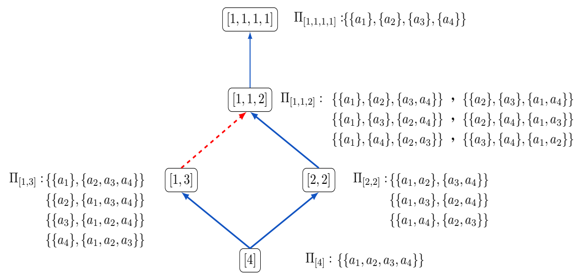
$[1,1,1,1]$
$\Pi_{[1,1,1,1]}: \{\{a_1\},\{a_2\},\{a_3\},\{a_4\}\}$
$[1,1,2]$
$\Pi_{[1,1,2]}: \{\{a_1\},\{a_2\},\{a_3,a_4\}\}$, $\{\{a_2\},\{a_3\},\{a_1,a_4\}\}$
$\{\{a_1\},\{a_3\},\{a_2,a_4\}\}$, $\{\{a_2\},\{a_4\},\{a_1,a_3\}\}$
$\{\{a_1\},\{a_4\},\{a_2,a_3\}\}$, $\{\{a_3\},\{a_4\},\{a_1,a_2\}\}$
$\Pi_{[1,3]}: \{\{a_1\},\{a_2,a_3,a_4\}\}$
$\{\{a_2\},\{a_1,a_3,a_4\}\}$
$\{\{a_3\},\{a_1,a_2,a_4\}\}$
$\{\{a_4\},\{a_1,a_2,a_3\}\}$
$[1,3]$
$[2,2]$
$\Pi_{[2,2]}: \{\{a_1,a_2\},\{a_3,a_4\}\}$
$\{\{a_1,a_3\},\{a_2,a_4\}\}$
$\{\{a_1,a_4\},\{a_2,a_3\}\}$
$[4]$
$\Pi_{[4]}: \{\{a_1,a_2,a_3,a_4\}\}$

Figure 1: A four-agent example of the integer partition graph. The bottom node $[4]$ represents the coalition structure formed by the grand coalition $\{a_1, a_2, a_3, a_4\}$. The top node $[1,1,1,1]$ represents the coalition structure formed by all the singleton coalitions $\{a_1\}, \{a_2\}, \{a_3\}, \{a_4\}$.

In our coalition structure representation, we codify each coalition $\mathcal{C}_i$ with a code (i.e., an index-based code). As a result, each agent of a given coalition $\mathcal{C}_i$ will be encoded with a number i, which corresponds to the index of the coalition to which it belongs. By doing so, each coalition structure is defined as a vector of size n where n represents the number of agents. Each position p in the vector identifies the agent $p_{/p=1..n}$ while the code in position p of the vector identifies the coalition to which the agent p belongs. Hence, a coalition structure will be encoded by a set of $x_i \geq 0$ numbers. Let us consider the following example. Let $\mathcal{A}$ be a set of $n = 10$ agents. Consider the coalition structure $\mathcal{CS} = \{\{a_1\}, \{a_2, a_3, a_4\}, \{a_5, a_6, a_7, a_8, a_9, a_{10}\}\}$ containing three coalitions: $\mathcal{C}_0 = \{a_1\}$, $\mathcal{C}_1 = \{a_2, a_3, a_4\}$ and $\mathcal{C}_2 = \{a_5, a_6, a_7, a_8, a_9, a_{10}\}$. The coalitions $\mathcal{C}_0, \mathcal{C}_1$ and $\mathcal{C}_2$ are encoded with codes 0, 1 and 2, respectively. $\mathcal{CS}$ will be encoded by the vector of codes $[x_1\ x_2\ ...\ x_{10}]$, where $x_{i/i=1..10} = 0 \Leftrightarrow a_i \in \mathcal{C}_0$, $x_{i/i=1..10} = 1 \Leftrightarrow a_i \in \mathcal{C}_1$ and, $x_{i/i=1..10} = 2 \Leftrightarrow a_i \in \mathcal{C}_2$. Figure 2 shows the form of the coalition structure $\mathcal{CS}$ using this representation. $\mathcal{CS}$ is encoded here with the vector of codes [0 1 1 1 2 2 2 2 2 2], where the number of different codes equals the number of coalitions forming the corresponding coalition structure and the size of the vector equals n. $\mathcal{CS}$ has three coalitions encoded with the codes 0, 1 and 2. Agent a_1 is in the coalition $\mathcal{C}_0$, this is why the number associated with a_1 in $\mathcal{CS}$ is 0. Agents a_2, a_3 and a_4 are in the coalition $\mathcal{C}_1$ and hence, the number associated with these agents is 1. The rest of the agents form the coalition $\mathcal{C}_2$.

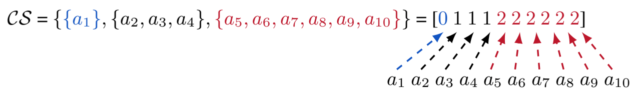
$\mathcal{CS} = \{\{a_1\}, \{a_2, a_3, a_4\}, \{a_5, a_6, a_7, a_8, a_9, a_{10}\}\} = [0\ 1\ 1\ 1\ 2\ 2\ 2\ 2\ 2\ 2]$
$a_1\ a_2\ a_3\ a_4\ a_5\ a_6\ a_7\ a_8\ a_9\ a_{10}$

Figure 2: Representation of the coalition structure $\mathcal{CS} = \{\{a_1\}, \{a_2, a_3, a_4\}, \{a_5, a_6, a_7, a_8, a_9, a_{10}\}\}$ in the new method with the vector of codes [0 1 1 1 2 2 2 2 2 2].

Any permutation of these numbers provides a different coalition structure, as shown in Figure 3. In this example, the first coalition is formed by agent a_3, the second coalition by agents a_2, a_4, a_6 and the last coalition is formed by agents $a_1, a_5, a_7, a_8, a_9, a_{10}$. Thus, the coalition structure is represented with the vector of codes [2 1 0 1 2 1 2 2 2 2].

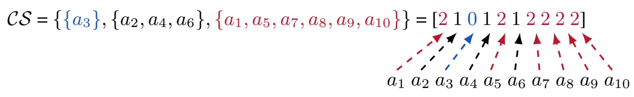
$\mathcal{CS} = \{\{a_3\}, \{a_2, a_4, a_6\}, \{a_1, a_5, a_7, a_8, a_9, a_{10}\}\} = [2\ 1\ 0\ 1\ 2\ 1\ 2\ 2\ 2\ 2]$
$a_1\ a_2\ a_3\ a_4\ a_5\ a_6\ a_7\ a_8\ a_9\ a_{10}$

Figure 3: Representation of the coalition structure $\mathcal{CS} = \{\{a_3\}, \{a_2, a_4, a_6\}, \{a_1, a_5, a_7, a_8, a_9, a_{10}\}\}$ in the new method with the vector [2 1 0 1 2 1 2 2 2 2]. Here, agent a_3 is encoded with the code 0 which represents the coalition $\mathcal{C}_0$, so $a_3 \in \mathcal{C}_0$. Likewise, agents a_2, a_4 and a_6 are encoded with the code 1 which represents the coalition $\mathcal{C}_1$, so $a_2, a_4, a_6 \in \mathcal{C}_1$. Similarly, agents a_1, a_5, a_7, a_8, a_9 and a_{10} are encoded with the code 2 representing the coalition $\mathcal{C}_2$, thus $a_1, a_5, a_7, a_8, a_9, a_{10} \in \mathcal{C}_2$. The coalition structure $\mathcal{CS}$ belongs to the node $[1,3,6]$ in the integer partition graph.

Generating all the combinations of these numbers guarantees exploration of all the coalition structures represented by each node. Let us now generalize this representation for the entire integer partition graph. For each node in level l, we have l coalitions, we then codify each coalition $\mathcal{C}_i$ with a code (i.e., a representative number) $i \in \mathcal{N} = \{0, 1, .., l-1\}$. Then, to represent with a vector of codes each coalition structure $\mathcal{CS}$ of this node, each coalition $\mathcal{C}_i$ of size k is depicted by k times the number (code) i. For example, for the node $[1,3,6]$ that contains three coalitions where $|\mathcal{C}_0| = 1$, $|\mathcal{C}_1| = 3$ and $|\mathcal{C}_2| = 6$, each vector representing a coalition structure of this node will contain one 0 for the coalition of size 1, three 1 for the coalition of size 3, and six 2 for the coalition of size 6 (see Figures 2 and 3). Figure 4 shows the vectors which represent all the coalition structures for a set of $n = 4$ agents. Each vector in Figure 4 corresponds to a coalition structure in Figure 1. This new representation of the coalition structures in the integer partition graph is based on coalition indexes, unlike the original version in (Rahwan et al. 2007). This new representation makes it possible to work with integers while searching in the space of coalition structures. This is particularly significant for the remainder of the paper where we use this representation to compute the coalition structures.

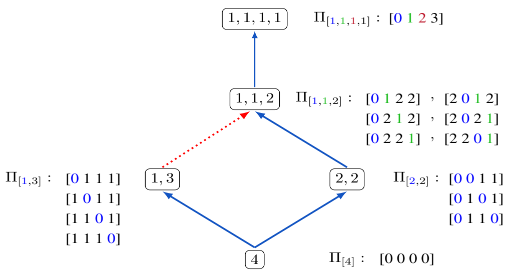

Figure 4: An example with four agents. The node $[1,3]$ has four coalition structures, each formed by two coalitions of size one and three. Only two numbers are needed to codify these coalitions. 0 codifies the first coalition of size 1, while 1 codifies the second coalition of size 3. Consequently, the coalition structures are represented by the vectors [0 1 1 1], [1 0 1 1], [1 1 0 1] and [1 1 1 0].

4 Code-based Coalition Structure Generation Algorithm

FACS partially enumerates the coalition structures of each node of the integer partition graph. Specifically, we start by generating all the integer partitions of n and searching each node from the bottom level of the integer partition graph up to the top level. Once all the possible partitions of the integer n are generated, the FACS algorithm proceeds in two main phases. The first phase involves generating, for each possible partition, the codes (i.e., the representative numbers) that can represent any coalition structure that belongs to it. These codes are then used in the second phase, which involves generating the different permutations of the numbers obtained at the end of phase 1. These permutations are then used to compute directly the coalition structures.

4.1 Phase 1: Code generation for each partition

The main purpose of this phase is to generate the codes for the coalition structures of each partition. As shown in Figure 2, the set of codes with which we compute the combinations contains repeated numbers. Thus, using a naive method for calculating combinations is not efficient due to the high computational load, especially for a large number of agents. Therefore, FACS partially generates the permutations. Consider the previous example of the node $[1,3,6]$ (see Figure 2). To enumerate the coalition structures represented by this node, we use the codes $0,1$ and 2 as shown in Figure 2. Rather than computing all the combinations of the 10 numbers (that can represent each of the 10 agents of the coalition structure), FACS calculates the combinations of the three codes, $\{0,1,2\}$ that represent the coalitions. We then obtain these combinations: $\vartheta = \{\{0,1,2\},\{0,2,1\},\{1,0,2\},\{1,2,0\},\{2,0,1\},\{2,1,0\}\}$. For each combination in ϑ, FACS calculates the first coalition structure of the combination called initialization. Each code in the combination will then be repeated as many times as the size of the coalition it represents. This means that in each initialization, the agents are assigned to the coalitions by respecting the order of these coalitions in the combination. For example, the initialization of $\{0,1,2\}$ is [0 1 1 1 2 2 2 2 2 2] because the coalition encoded with 0 is of size 1, the coalition encoded with 1 is of size 3 and the coalition encoded with 2 is of size 6 (see Figure 2). Likewise, the initialization of $\{0,2,1\}$ is [0 2 2 2 2 2 2 1 1 1] because we start by completing the coalition encoded with 0 by one agent (which corresponds to its size of 1), then we complete the coalition encoded with 2 by the next six agents, and finally the coalition encoded with 1 by the last three agents. Similarly, the initialization of $\{2,0,1\}$ is [2 2 2 2 2 2 0 1 1 1]. The different initializations thus obtained will then be used in the second phase to generate the different permutations.

Algorithm 1 shows how FACS generates the initializations. Given a combination of l codes as input, Algorithm 1 considers these codes in order of appearance and fills the initialization vector of size n starting from agent a_1 (line 1) to agent a_n. The first loop (line 2) iterates l times by processing one code at a time. On the other hand, loop 2 (line 3) fills the next k elements of the initialization vector with the code of the coalition at the index j of the combination. This is formulated by $code(j)$ in line 4. The value of k is defined by the size of the j^{th} coalition being processed. It is expressed as $size(j)$ in line 3. To illustrate the operations of the algorithm, let us consider the node $[1,3,6]$ and the related combination $\{2,0,1\}$. The first iteration of loop 1 ($j=0$ in line 2) considers the first code of the combination, namely 2, relating to coalition $\mathcal{C}_2$ and hence $size(0) = size(\mathcal{C}_2) = 6$. The second loop will then fill the first 6 elements ($k=1$ to $size(j)$ in line 3) of the initialization vector by 2, the code of $\mathcal{C}_2$ ($code(j)$ in line 4). By doing so, we assign the agents $a_1, ..., a_6$ to $\mathcal{C}_2$. The same processing is then applied to the remaining codes of the considered combination. Finally, we obtain the initialization vector [2 2 2 2 2 2 0 1 1 1].

Algorithm 1: Constructing an initialization

```
   Input: A combination $\zeta$ of the codes of a node that
          contains $i$ coalitions.
   Output: An initialization vector: Init (see section 4.1).
1  $a \leftarrow 1$ ▷ $a$ is a variable initialized with the index of the
   first agent $a_1$
2  for $j = 0$ to $i-1$ do
3      for $k = 1$ to $size(j)$ do ▷ $size\{j\}$ returns the size
       of the coalition in index $j$ of the combination $\zeta$
4          Init[$a$] $\leftarrow code(j)$ ▷ $code(j)$ returns the code of
           the coalition in index $j$ of the combination $\zeta$
5          $a \leftarrow a + 1$
6      end
7  end
8  Return Init
```

4.2 Phase 2: Coalition structure generation

The main purpose of this second phase is to generate the permutations of the codes composing the vectors obtained at the end of phase 1. Each newly generated vector of codes after each permutation will then be associated with a different coalition structure. For each initialization vector performed in phase 1, FACS computes the permutations as follows. First, FACS starts with the first agent (first number) and permutes its code with each of the codes of the other agents. Then, FACS moves to the next agent and applies the same permutation operations to its code. This process is then iterated until FACS reaches the last code of the concerned vector. A four-agent example is shown in Figure 5. This example concerns the node $[1, 1, 2]$ of level 3 (see Figure 4) that represents the coalition structures with two coalitions of size 1 and one coalition of size 2. The first coalition structure that FACS affects to this node is represented by the vector [0 1 2 2] (see line 1 of Figure 5), which is the initialization of the combination $\{0, 1, 2\}$ (we have one 0 for the coalition of size 1 and one 1 for the second coalition of size 1 followed up by two codes 2 for the coalition of size 2). In the second line of Figure 5, we swap the first two codes of the vector [0 1 2 2] and get the vector [1 0 2 2]. This vector represents the coalition structure $\{\{a_2\}, \{a_1\}, \{a_3, a_4\}\}$. In the third line, we swap the first code with the third code and obtain the vector that represents the coalition structure $\{\{a_3\}, \{a_2\}, \{a_1, a_4\}\}$. Line 4 reflects the last permutation applied to the first code. However, in line 5, we move to the second agent and swap the second code with the third one and get the coalition structure $\{\{a_1\}, \{a_3\}, \{a_2, a_4\}\}$ represented by the vector [0 2 1 2]. Line 6 reflects the last permutation of the initialization.

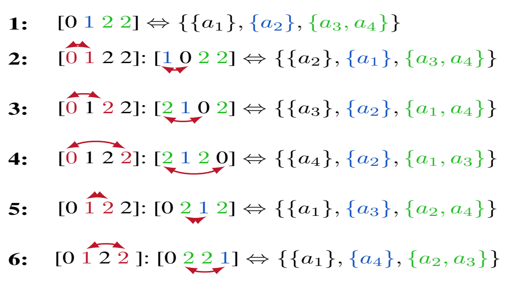
1: [0 1 2 2] ⇔ $\{\{a_1\}, \{a_2\}, \{a_3, a_4\}\}$
2: [0 1 2 2]: [1 0 2 2] ⇔ $\{\{a_2\}, \{a_1\}, \{a_3, a_4\}\}$
3: [0 1 2 2]: [2 1 0 2] ⇔ $\{\{a_3\}, \{a_2\}, \{a_1, a_4\}\}$
4: [0 1 2 2]: [2 1 2 0] ⇔ $\{\{a_4\}, \{a_2\}, \{a_1, a_3\}\}$
5: [0 1 2 2]: [0 2 1 2] ⇔ $\{\{a_1\}, \{a_3\}, \{a_2, a_4\}\}$
6: [0 1 2 2]: [0 2 2 1] ⇔ $\{\{a_1\}, \{a_4\}, \{a_2, a_3\}\}$

Figure 5: An illustration of FACS when searching the node $[1, 1, 2]$ of the integer partition graph in Figure 4.

Algorithm 2 shows how FACS generates different vectors that represent the coalition structures given an initialization. It starts with the first agent. The first loop (line 2) iterates i times by considering one coalition at a time. On the other hand, loop 2 (line 3) ensures that the permutation does not occur between the codes of agents that belong to the same coalition. By doing so, we reduce the generation of duplicated coalition structures. Thus, permutation always occurs between the agent's code under processing and the agents' codes of the following coalitions. This operation is performed by the third loop in line 4. After each permutation, a new code vector is generated. Finally, by moving to the next code (agent) in line 7, we ensure that the same processing is applied to the remaining codes of the considered coalition. For clarity, the pseudo-code for evaluating the coalition structures is provided in Algorithm 4.

Algorithm 2: Permuting the codes

```
Input: An initialization Init and the number of
       coalitions $i$ in a node.
Output: A set of code vectors $\mathcal{PM}$ resulting from the
        permutations.
1  $a \leftarrow 1$ ▷ $a$ is a variable initialized with the index of the
   first agent $a_1$
2  for $j = 0$ to $i - 1$ do
3      for $k = 1$ to size($j$) do ▷ size{$j$} returns the size
       of the coalition to which agent $a$ belongs
4          for $l$ = size($j$) + $a$ to $n$ do
5              Permutation $\leftarrow$ permute(Init[$a$], Init[$l$])
6              add Permutation to $\mathcal{PM}$
7          end
8          $a \leftarrow a + 1$ ▷ move to the next agent
9      end
10 end
11 Return $\mathcal{PM}$
```

Pruning combinations. We showed earlier how FACS generates the permutations in a node of the IP graph. Note that computing all the permutations for each combination is not efficient because it generates some coalition structures repeatedly such as $\{\{a_1, a_3\}, \{a_2, a_4\}\}$ and $\{\{a_2, a_4\}, \{a_1, a_3\}\}$, especially for nodes that are far from the root (the bottom node). Thus, to address this issue, FACS does not consider all the initialization vectors of a given node but only those that are most likely to generate a large number of different coalition structures. Figure 6 shows an example of a repeated generation of coalition structures when developing two initialization vectors of the same node. For instance, let us consider the node $[2, 2]$ of level 2, which represents the coalition structures with two coalitions of size 2. To enumerate these coalition structures, FACS uses two initializations, namely, [0 0 1 1] (initialization of $\{0, 1\}$) and [1 1 0 0] (initialization of $\{1, 0\}$). In Figure 6, we can see that in line 2 we have permuted the first and third codes resulting in the vector that represents the coalition structure $\{\{a_2, a_3\}, \{a_1, a_4\}\}$. In line a, we considered the second initialization. Line e shows the permutation of the second and fourth codes, which results in the same coalition structure computed in line 2. Broadly speaking, if it is found that two coalitions or more have the same size (i.e. we have repeated coalition sizes in the node), then at least two combinations would order the code of each coalition differently. However, the size of the corresponding coalitions would remain the same and therefore the treatment of both combinations would generate the same coalition structures. For instance, for the node $[2, 2]$, the two initializations [0 0 1 1] and [1 1 0 0] represent the same coalition structure which is $\{\{a_1, a_2\}, \{a_3, a_4\}\}$ (or $\{\{a_3, a_4\}, \{a_1, a_2\}\}$). Hence, there is no need to compute the permutations for both initializations.

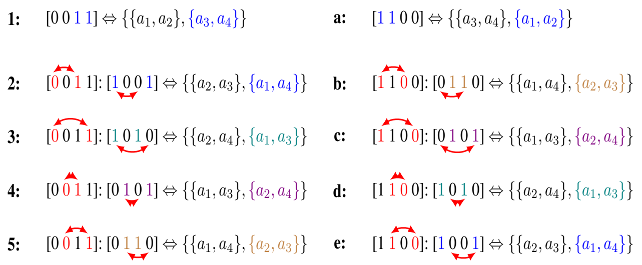
1: [0 0 1 1] ⇔ {{a1, a2}, {a3, a4}}
a: [1 1 0 0] ⇔ {{a3, a4}, {a1, a2}}
2: [0 0 1 1]: [1 0 0 1] ⇔ {{a2, a3}, {a1, a4}}
b: [1 1 0 0]: [0 1 1 0] ⇔ {{a1, a4}, {a2, a3}}
3: [0 0 1 1]: [1 0 1 0] ⇔ {{a2, a4}, {a1, a3}}
c: [1 1 0 0]: [0 1 0 1] ⇔ {{a1, a3}, {a2, a4}}
4: [0 0 1 1]: [0 1 0 1] ⇔ {{a1, a3}, {a2, a4}}
d: [1 1 0 0]: [1 0 1 0] ⇔ {{a2, a4}, {a1, a3}}
5: [0 0 1 1]: [0 1 1 0] ⇔ {{a1, a4}, {a2, a3}}
e: [1 1 0 0]: [1 0 0 1] ⇔ {{a2, a3}, {a1, a4}}

Figure 6: An example of repeated generation of coalition structures when searching the node [2,2] of the IP graph. FACS finds that both combinations compute the same coalition structures (each combination orders the coalition codes differently but the size of the corresponding coalitions remains the same). Thus, FACS does not compute the permutations for the two initializations but only for one.

Algorithm 3 shows how FACS avoids redundant coalition structures. Each combination distinctly orders the coalitions. The first loop iterates over the set of combinations. For each combination in position j (line 1), the second loop iterates over the rest of the combinations to test whether the order of coalition sizes is the same as in the combination in position j or not. If so, it does not consider any combination that has the same coalition order with the one in position j. In line 4, $size(\mathcal{C}_i^j)$ returns the size of the coalition in index i of the combination in position j.

```
Algorithm 3: Combination pruning
Input: A set $\vartheta$ of $m$ combinations of a partition $p$.
Output: A set of non redundant combinations.
1 for $j = 1$ to $m$ do
2     for $k = j + 1$ to $m$ do
3         if $\forall i_{/i=1..|p|}, size(\mathcal{C}_i^j) = size(\mathcal{C}_i^k)$ then ▷ $size\{\mathcal{C}_i^k\}$ returns the size of the coalition in index $i$ of the combination in position $k$
4             Remove the combination $k$ from $\vartheta$
5         end
6     end
7 end
8 Return $\vartheta$
```

By computing the permutations on different initializations instead of computing the combinations of only one initialization (Figure 2), FACS becomes faster but loses the ability to return the optimal solution because some coalition structures may not be generated with this method. For example, the coalition structure presented in Figure 3 is skipped by FACS because no permutation of the initializations yields it.

Moreover, for the nodes that belong to the first 5 levels, FACS calculates the permutations for all the combinations. However, for nodes that are on level i where $i \in \{6, ..., n\}$, FACS computes the permutations for only two initializations. We found that calculating more permutations will be extremely time-consuming, without significantly improving the solution quality. For nodes that are far from the root, the number of coalitions, and hence of codes, is large. Thus, calculating the combinations for the codes in phase 1 would take more time than the actual permutations in phase 2.

For example, for the top node that contains the singleton coalitions, this means that FACS would need to compute the combinations of n codes whilst there is only one possible coalition structure. Of course, this exception is handled and FACS only generates one coalition structure but this example simply shows that we do not need to compute all the combinations in phase 1 for the nodes that are far from the root. We choose only two initializations to lower the cost of searching in more combinations while ensuring that the coalition structures generated are different. The two combinations chosen are those that order the codes in an ascending order for the first one and in a descending order for the second one. This makes the resulting two initializations as different as possible, i.e. the order of coalition sizes in the two combinations would never be the same (except for the case where all the coalitions have the same size).

Theorem 1 *Given any partition, the FACS algorithm guarantees that each agent will belong to any coalition of size k at least $2 \times k$ times.*

Proof 1 *For each node in the IP graph, FACS computes the permutations for at least two combinations. Thus, it suffices to prove that for each combination, every agent will belong to each coalition of size k at least k times. Now, consider any coalition $\mathcal{C}$ of size k. The result is immediate when the agents belong to $\mathcal{C}$. Thus, for the remainder of the proof, we will assume that the agents do not belong to $\mathcal{C}$. In the previous section, we stated that the code of any agent a_i will be permuted with the code of all the other agents and hence with each agent in the coalition $\mathcal{C}$. Thus, any agent a_i will have the code that represents the coalition $\mathcal{C}$, k times, which proves that a_i will belong to the coalition $\mathcal{C}$ of size k at least k times.*

Theorem 2 *The FACS algorithm is anytime.*

Proof 2 *To compute the coalition structures, FACS starts with an initial coalition structure and then improves it over time. When the execution is stopped, it can still return the currently best solution that is better than the previous best ones. Therefore, FACS is anytime.*

Algorithm 4 details the coalition structure generation algorithm. FACS iterates over the nodes of the integer partition graph that covers all the partitions of integer n, given n agents. Algorithm 4 starts by storing the value of each coalition (see the loop in lines 1-5). The optimal coalition structure is first set to be the grand coalition, before its value improves over time as described in lines 9-26. In more detail, FACS processes one node of the IP graph at a time as follows: after generating the set of codes for the given node (line 11), it computes the combinations and then calls Algorithm 3 to filter them (see lines 12-13). For each combination, FACS generates the initialization vector (line 15) and computes the permutations (line 16) by calling Algorithm 1 and 2, respectively. Each permutation, which represents a new coalition structure, is then considered to, eventually, determine the new best one, as described by lines 17-23.

Algorithm 4: The FACS algorithm

```
Input: Set of all possible coalitions $p(\mathcal{A})$ and the value
       $v(\mathcal{C})$ of each coalition $\mathcal{C} \in p(\mathcal{A})$.
Output: The best coalition structure $\mathcal{CS}^+$ and its value.
for $i = 1$ to $n$ do
    for $\mathcal{C} \subseteq \mathcal{A}$, where $|\mathcal{C}| = i$ do
        $\mathcal{V}(\mathcal{C}) \leftarrow v(\mathcal{C})$
    end
end
Generate all integer partitions of $n$ and store them in the
  set of partitions $\varphi$
$\mathcal{CS}^+ \leftarrow \mathcal{A}$ ▷ $\mathcal{CS}^+$ is initialized with the grand coalition
$\mathcal{V}^+ \leftarrow V(\mathcal{A})$
for $i = 1$ to $n$ do
    for $p \in \varphi$, where $|p| = i$ do
        $\mathcal{N} \leftarrow \{0, 1, .., i-1\}$ ▷ $\mathcal{N}$ is the set of codes
          corresponding to the coalitions in the partition $p$
        Generate combinations of $\mathcal{N}$ and store them in
          the set $\vartheta$
        $\vartheta \leftarrow$ Combination_pruning($\vartheta$) ▷ call of
          Algorithm 3
        foreach $\varsigma \in \vartheta$ do
            Init $\leftarrow$ Constructing_an_initialization($\varsigma$) ▷
              call of Algorithm 1
            $\mathcal{PM} \leftarrow$ permuting_the_codes(Init) ▷ call of
              Algorithm 2, $\mathcal{PM}$ is a set of code vectors
            foreach $\mathcal{CS} \in \mathcal{PM}$ do ▷ each permutation
              represents a coalition structure $\mathcal{CS}$
                $\mathcal{V} = \sum_{j=0}^{i-1} \mathcal{V}(\mathcal{C}_j)$, where $\mathcal{C}_j \in \mathcal{CS}$
                if $\mathcal{V} > \mathcal{V}^+$ then
                    $\mathcal{V}^+ \leftarrow \mathcal{V}$
                    $\mathcal{CS}^+ \leftarrow \mathcal{CS}$
                end
            end
        end
    end
end
Return $\mathcal{CS}^+$, $\mathcal{V}^+$
```

5 Empirical Evaluation

We evaluated the performance of our algorithm on several value distributions used in the literature and compared it with ODP-IP (Michalak et al. 2016) and CSG-UCT (Wu and Ramchurn 2020). For ODP-IP and CSG-UCT, we used the code provided by the authors. All algorithms were implemented in Java and all of our evaluations were performed on a DELL PowerEdge R740, which has the configuration of an Intel(R) Xeon(R) Gold 5120 CPU at 2.20 GHz with 384GB of RAM at 2666 MHz. We considered two studies to extensively evaluate our technique. The first one considered sets of agents up to 27 agents in order to compare FACS to ODP-IP and CSG-UCT, while the second one focused on comparing FACS to CSG-UCT with sets of agents of more than 27 agents. We only compared with CSG-UCT because ODP-IP cannot be run with more than 27 agents. In the first series of experiments, we used the following two performance metrics to evaluate the performance of the algorithms. First, we computed the **execution time** required to find a solution. This metric shows how long it takes to find the optimal solution or a near-optimal solution in the case of an exact algorithm or a heuristic, respectively. Thereafter, we computed the **solution quality** $= \frac{v(\mathcal{CS}^+)*100}{v(\mathcal{CS}^*)}\%$, which is computed as the ratio between the value of the best coalition structure $\mathcal{CS}^+$ found by an algorithm after completion and the value of the optimal solution $\mathcal{CS}^*$ found by ODP-IP. This measure shows how good a solution found is with regard to the optimal one. We considered different value distributions to extensively evaluate our technique. For each distribution, we randomly generated multiple problem instances varying the number of agents involved. Tests were conducted using different value distributions in order to obtain different results for the ODP-IP and CSG-UCT algorithms. However, it is worth noting that the runtime of our algorithm is not affected by the value distribution. We considered the following distributions: Agent-based Uniform, Modified Normal (Rahwan, Michalak, and Jennings 2012), Agent-based Normal, Beta, Exponential (Michalak et al. 2016), Normal (Rahwan et al. 2007), Uniform (Larson and Sandholm 2000), Modified Uniform (Service and Adams 2010), Normally Distributed Coalition Structures (NDCS) (Rahwan et al. 2009), Weibull and F (Changder et al. 2020).

Figure 7 illustrates the execution time of FACS, ODP-IP and CSG-UCT for each of the above distributions. For each point on Figure 7 we conduct an average of 50 tests. As we can see, for all the value distributions, our algorithm outperforms all the considered algorithms. For example, with 25 agents for the Normal distribution, the ratio of ODP-IP time and our algorithm time is 75.9, meaning that our algorithm takes 0.013% of the time taken by ODP-IP to provide a good quality solution (99%). This shows that use of the new representation of the search space with the proposed heuristic algorithm provides promising results for the CSG problem.

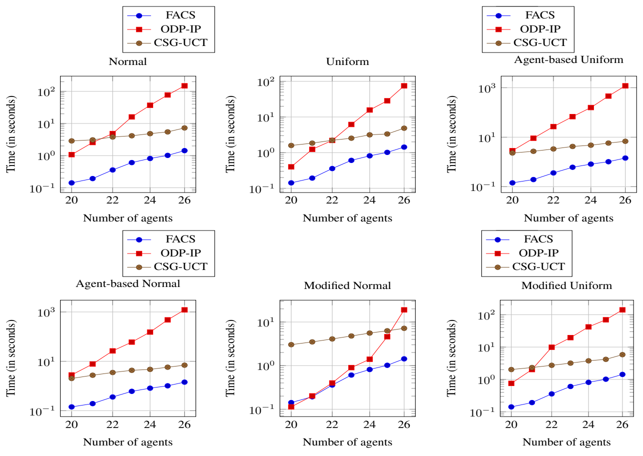

Figure 7: Time performance in seconds of FACS, ODP-IP and CSG-UCT for a number of agents between 20 and 26.

Next, we evaluate the effectiveness of FACS by comparing the quality of the solutions found by our algorithm with the solution quality of the optimal coalition structure.

Table 1 reports the quality of the solutions provided by FACS and those provided by CSG-UCT, when run to completion given different value distributions. For each distribution, we took the average solution quality produced by the two algorithms for a number of agents between 20 and 26. As we can see, FACS outperforms CSG-UCT with better solution quality in some value distributions and remains competitive with other distributions, thus confirming the effectiveness of our algorithm.

Distribution	Solution Quality	
	FACS	CSG-UCT
Agent-based normal	99%	99%
Beta	99%	99%
Uniform	99%	98%
Normal	98%	95%
Agent-based uniform	95%	89%
F Distribution	87%	85%
Exponential	80%	71%
Modified normal	84%	94%
NDCS	84%	90%
Modified uniform	81%	95%
Weibull Distribution	86%	94%

Table 1: Solution Quality of FACS and CSG-UCT when run to completion for sets of agents between 20 and 26. Here, the comparison is made with ODP-IP, which always provides the optimal solution when run to completion.

As mentioned above, FACS can be run with more than 28 agents while using the values of all possible coalitions. To demonstrate this, we conducted experiments on several value distributions and showed the solution quality of FACS and CSG-UCT given different numbers of agents (between 28 and 30 agents). As no exact algorithm is run for these sets of agents, the solution quality shown in Figure 8 is calculated as follows: $\frac{v(\mathcal{CS})}{max(v(\mathcal{CS}^+_{FACS}), v(\mathcal{CS}^+_{CSG-UCT}))}$, where $v(\mathcal{CS}^+_{FACS})$ and $v(\mathcal{CS}^+_{CSG-UCT})$ are the values of the best solutions provided by FACS and CSG-UCT, respectively. As we can see from Figure 8, FACS still yields good quality solutions when run with more than 28 agents, compared to those provided by CSG-UCT.

Next, we evaluate the anytime property of FACS. Figure 9 shows the time required to provide a certain solution quality for 30 agents. Here, time is measured in milliseconds. As can be seen, FACS still provides good quality solutions. For instance, for the Normal, Uniform and Beta distributions, solution quality exceeds 99% after about 10 milliseconds.

6 Conclusion

In this paper, we proposed a new algorithm for solving the coalition structure generation problem. It uses a complete set of coalitions as input and can be run with more than 28 agents. Our algorithm is based on an innovative heuristic algorithm (FACS) and a novel representation of the search space where each coalition of the integer partition graph is codified by an index-based code. The principle of FACS is hence to generate the coalition structures by permuting these codes. The main features of our algorithm are : (1) FACS went beyond the limit of 27 agents stated by exact algorithms; (2) It is an anytime algorithm; (3) It finds a high quality solution. Our performance analysis and empirical evaluation prove that our algorithm is effective and faster than the state-of-art-algorithms by several orders of magnitude for different numbers of agents while experimenting a wide range of value distributions.

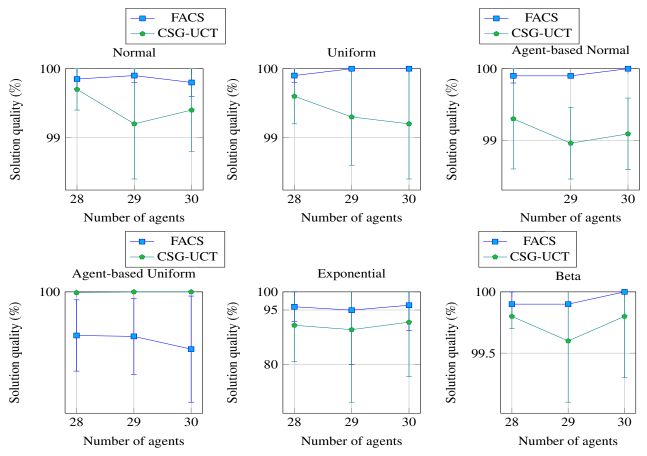

Figure 8: Solution quality of FACS and CSG-UCT when run to completion for a number of agents between 28 and 30. We recall that ODP-IP cannot run with more than 27 agents.

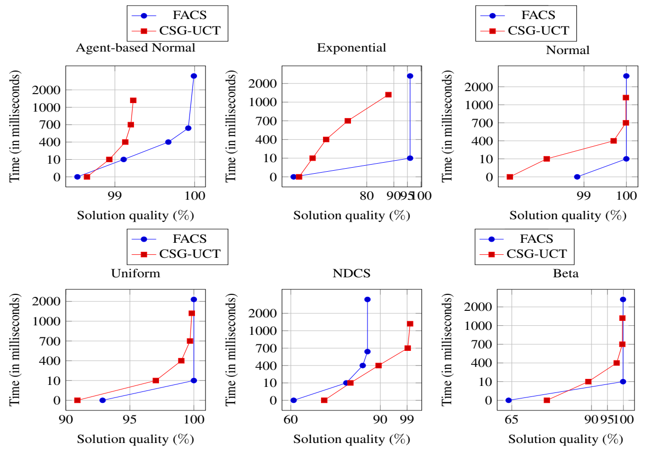

Figure 9: Time taken to produce a certain solution quality by FACS and CSG-UCT for 30 agents. Here, solution quality is calculated by using the formula: $\frac{v(\mathcal{CS})}{max(v(\mathcal{CS}^+_{FACS}), v(\mathcal{CS}^+_{CSG-UCT}))}$, where $v(\mathcal{CS})$ is the current best solution of the algorithm.

References

Changder, N.; Aknine, S.; Ramchurn, S. D.; and Dutta, A. 2020. ODSS: Efficient Hybridization for Optimal Coalition Structure Generation. In *Proc. of AAAI*, 7079–7086.

Dang, V. D.; Dash, R. K.; Rogers, A.; and Jennings, N. R. 2006. Overlapping coalition formation for efficient data fusion in multi-sensor networks. In *Proc. of AAAI*, volume 6, 635–640.

Farinelli, A.; Bicego, M.; Ramchurn, S.; and Zuchelli, M. 2013. C-link: A hierarchical clustering approach to large-scale near-optimal coalition formation. In *Proc. of IJCAI*, 407–413.

Larson, K. S.; and Sandholm, T. W. 2000. Anytime coalition structure generation: an average case study. *Journal of Experimental & Theoretical Artificial Intelligence* 12(1): 23–42.

Michalak, T.; Rahwan, T.; Elkind, E.; Wooldridge, M.; and Jennings, N. R. 2016. A hybrid exact algorithm for complete set partitioning. *Artificial Intelligence* 230: 14–50.

Rahwan, T.; and Jennings, N. R. 2008. An improved dynamic programming algorithm for coalition structure generation. In *Proc. of the 7th international joint conference on Autonomous agents and multiagent systems-Volume 3*, 1417–1420. International Foundation for Autonomous Agents and Multiagent Systems.

Rahwan, T.; Michalak, T.; and Jennings, N. R. 2012. A hybrid algorithm for coalition structure generation. In *Proc. of AAAI*, 1443–1449.

Rahwan, T.; Ramchurn, S. D.; Dang, V. D.; and Jennings, N. R. 2007. Near-Optimal Anytime Coalition Structure Generation. In *Proc. of IJCAI*, volume 7, 2365–2371.

Rahwan, T.; Ramchurn, S. D.; Jennings, N. R.; and Giovannucci, A. 2009. An anytime algorithm for optimal coalition structure generation. *Journal of artificial intelligence research* 34: 521–567.

Service, T.; and Adams, J. 2010. Approximate Coalition Structure Generation. In *Proc. of AAAI*, 854–859.

Tsvetovat, M.; and Sycara, K. 2000. Customer coalitions in the electronic marketplace. In *Proc. of the fourth international conference on Autonomous agents*, 263–264.

Wu, F.; and Ramchurn, S. D. 2020. Monte-Carlo Tree Search for Scalable Coalition Formation. In *Proc. of IJCAI*, 407–413.