Distributing Coalition Value Calculations to Coalition Members

Luke Riley, Katie Atkinson, Paul E. Dunne and **Terry R. Payne**
Department of Computer Science
University of Liverpool
Liverpool, UK
{L.J.Riley, K.M.Atkinson, P.E.Dunne, T.R.Payne}@liverpool.ac.uk

Abstract

Within characteristic function games, agents have the option of joining one of many different coalitions, based on the utility value of each candidate coalition. However, determining this utility value can be computationally complex since the number of coalitions increases exponentially with the number of agents available. Various approaches have been proposed that mediate this problem by distributing the computational load so that each agent calculates only a subset of coalition values. However, current approaches are either highly inefficient due to redundant calculations, or make the benevolence assumption (i.e. are not suitable for adversarial environments). We introduce *DCG*, a novel algorithm that distributes the calculations of coalition utility values across a community of agents, such that: (i) no inter-agent communication is required; (ii) the coalition value calculations are (approximately) equally partitioned into shares, one for each agent; (iii) the utility value is calculated only once for each coalition, thus redundant calculations are eliminated; (iv) there is an equal number of operations for agents with equal sized shares; and (v) an agent is only allocated those coalitions in which it is a potential member. The *DCG* algorithm is presented and illustrated by means of an example. We formally prove that our approach allocates all of the coalitions to the agents, and that each coalition is assigned once and only once.

1 Introduction

Coalition formation is the process by which a number of agents partition themselves into temporary teams (i.e. coalitions), where each coalition collaborates to achieve mutually beneficial results. Coalition formation is a well studied research area in multi-agent systems and has a wide range of potential applications, including: electronic auctions/market places; communication networks; the smart grid; grid computing; distributed vehicle routing; sensor networks; multi-agent planning; and computational trust (Chalkiadakis, Elkind, and Wooldridge 2011).

Coalition formation can be divided into a three stage process (Sandholm et al. 1999): (1) calculating the utility value of each possible coalition; (2) finding a satisfiable set of coalitions; and (3) dividing the utility values of the coalitions into a stable distribution (where no agent can object to its assigned payoff). The first stage of coalition formation requires the agents to calculate the utility value of each coalition, which can be computationally complex as the number of coalitions that can form given a population of n agents is $2^n - 1$. Furthermore, the complexity of calculating an individual coalition's value can vary, and potentially be exponential (Sandholm and Lesser 1997). Thus, even if each agent only calculates the value of those coalitions that it can potentially form (i.e. $\frac{2^n}{2}$), then this can still result in a significant overlap of calculations, such that this redundancy can converge to 100%, as $\lim_{n\to\infty} \frac{2^n-1}{n2^{(n-1)}} = 0$.

Various studies have explored how to distribute coalition value calculations amongst agents, to reduce the computation cost for each agent, and possibly reduce the overall computation time (Shehory and Kraus 1995; 1996; 1998; Rahwan and Jennings 2007; Vinyals et al. 2012). Furthermore, as the agents themselves can potentially determine the value of the coalitions in which they participate, this eliminates the need for a trusted central authority/independent third-party responsible for determining the value calculations. However, existing approaches are not necessarily efficient: some only guarantee that every coalition value will be calculated *at least once*, potentially resulting in redundant calculations (Shehory and Kraus 1995; 1996; 1998); while others distribute the calculations unequally amongst the agents (Vinyals et al. 2012). The approach given in (Rahwan and Jennings 2007) does not suffer from these deficiencies but allocates some coalitions to agents that do not appear in them, which can be undesirable in adversarial environments where deception may occur.

We present a novel algorithm for distributing coalition value calculations, named the *Distributed Coalition Generation* (DCG) algorithm, that addresses these limitations by allocating all coalitions to agents that appear in them and balancing the computational load approximately evenly (wrt. share size and number of operations) across the agents. This process is completed without inter-agent communication in a way that eliminates redundant coalition value calculations.

Furthermore, the DCG algorithm can be combined with other approaches to complete the coalition formation process. For example, if agents want to find a core/ϵ-core stable solution (Osbourne and Rubinstein 1994), then DCG can be used as input to algorithms, such as (Cesco 1998; Wu 1977), that determine such solutions.

In this paper, the distributed coalition value calculation problem is presented and related work critiqued in Section 2. Section 3 describes the underlying coalition ordering mechanism and provides an example, while the DCG algorithm is presented in Section 4. Proofs for the properties of this work are presented in Section 5 which focus on how: (a) all coalitions are assigned; and (b) all coalitions are assigned once and only once. After the DCG algorithm and related theory has been fully introduced, comparisons with related work are made in Section 6. Possible lines of future work are detailed in Section 7 and the paper concludes in Section 8.

2 Related Work

The *characteristic function game* model of coalition formation (Osbourne and Rubinstein 1994) is denoted: $\mathcal{G} = \langle N, v \rangle$ where $N = \{1, 2, ..., n\}$ is the set of agents, and v is the *characteristic function* ($v(2^N) \rightarrow \mathbb{R}$) which assigns every possible coalition a real numeric payoff. To find an *outcome* of a characteristic function game, the value of each coalition needs to be calculated and the coalitions compared. As it assumes that each coalition's value is static, deterministic and independent of the other coalitions that could form, a *characteristic function game* has the property that each coalition's value needs to be calculated only once.

This property was originally exploited by Shehory and Kraus (Shehory and Kraus 1995; 1996; 1998) to reduce the number of redundant calculations. Instead of each agent calculating each coalition value in which it is a member, they introduced an algorithm (referred to here as SK) where each agent negotiated over which coalitions (that also comprised that agent) should be allocated to its value calculation share. However, this algorithm suffers from several limitations (Rahwan and Jennings 2007): (i) many messages need to be sent between the agents to facilitate the negotiation; (ii) there is *no* guarantee that the number of coalition value calculations performed by each agent is approximately equal; and (iii) there is no guarantee that every coalition value is calculated *once and only once* (SK only guarantees that every coalition value will be calculated *at least once*).

Rahwan and Jennings argued that the SK algorithm utilised the resources of the system *inefficiently*, and addressed this by proposing the *Distributed Coalition Value Calculation* algorithm (*DCVC*) (Rahwan and Jennings 2007). Their algorithm grouped coalitions into lists, and divided the lists into *shares*, one for each agent. They showed that: (i) no inter-agent communication was necessary; (ii) the agents' coalition value calculation shares were approximately equal; and (iii) each coalition value was calculated *once and only once*. However, a weakness of DCVC was that a coalition assigned to an agent i's share may not include i.

More recently, Vinyals *et al.* (Vinyals et al. 2012) proposed an algorithm (referred to as VBFR) that distributes the coalition value calculations to agents when they are connected in a network, with the property that each coalition must include member agents that are connected together in a graph. However their algorithm failed to evenly distribute the coalition value calculations (for a fully connected graph), because the number of calculations that an agent had to perform was correlated with an ID associated to each agent.

3 Preliminaries and Introductory Example

The DCG algorithm (in Section 4) exploits a novel method for representing and ordering coalitions, such that different coalitions can be allocated to each agent, in such a way as to facilitate the construction of *shares* (one per agent) that eliminate redundant coalition value calculations.

Ordering and Integer Partitions

In this paper, a coalition $C \subseteq \{1, 2, ..., n\}$ is represented as an ordered sequence of identifiers (IDs) that form a *coalition array*, where no agent appears more than once in any coalition, and where the coalition size $s = |C|$. An *integer increment value* between two contiguous agents i and j in a coalition array corresponds to the difference in the agents' IDs[1]. For example, if we have a coalition array $[3, 6, 1]$, then there are two integer increment values between the ID pairs $3, 6$ and $6, 1$. There is an additional increment between the last and the first agent IDs in the array; i.e. the ID pair $1, 3$. The *integer increment value* between two agents i and j can be decomposed into a *baseline increment* (which is assumed to be 1, since agent IDs are unique) and an *offset increment*, denoted $t_i = (j - i) - 1 \bmod n$ (i.e. integers modulo n). Thus, if $t_i = 0$, the difference between the IDs for agents i and j corresponds only to the baseline increment; whereas if $t_i \neq 0$, then t_i represents an additional offset increment. An *increment array* (IA) denoted $\underline{t} = \langle t_0, t_1, \dots, t_{s-1} \rangle$ therefore represents the offset increments between the identifiers of the coalition array. For example, given the coalition array $[3, 6, 1]$, the corresponding IA will be $\langle 2, 0, 1 \rangle$.

An *integer partition* of x is a combination of positive integers that add up to exactly x. The DCG algorithm uses integer partitions to identify the offset increments between consecutive pairs of IDs in the coalition array. The full set of integer partitions is denoted $\mathcal{I}(n - s)$; for example, given $n = 6$ and $s = 3$, $\mathcal{I}(n - s) = \{\{3\}, \{2, 1\}, \{1, 1, 1\}\}$. Increment arrays can be formed from an integer partition I for coalitions of size s, *only* when $I \in \mathcal{I}(n - s)$ and $|I| \leq s$, by including additional zero values to satisfy the property:

$$\sum_{i=0}^{s-1} t_i = (n - s)$$

For example, when $n = 6$ and $s = 3$, the integer partition $\{2, 1\}$ could be used to form various possible increment arrays: $\langle 2, 1, 0 \rangle$, $\langle 2, 0, 1 \rangle$, etc. The integer increment values corresponding to the increment array $\langle 2, 1, 0 \rangle$ result from the two following coalition arrays $[1, 4, 6]$ and $[2, 5, 1]$, as the ID pairs $1, 4$ and $2, 5$ share $(2 + 1)$, whereas the ID pairs $4, 6$ and $5, 1$ share $(1 + 1)$. As IAs are shared between coalition arrays, the new ordering method introduced in this paper divides the coalitions into 2-dimensional lists $L_{s,\underline{t}}$.

Each *increment array* $\underline{t}$ represents the necessary offset increments from one agent ID of the coalition array to the next. For agent i to generate a coalition C assigned to itself using $\underline{t}$, the first element of the coalition array will be i to *motivate*

[1]As agent IDs are in the range $[1, n]$, IDs modified using an integer increment will result in an ID modulo n. The agent ID n will be returned when the ID 0 is found because $0 \equiv n \pmod{n}$.

	L_3			
	$L_{3,\langle 3,0,0\rangle}$	$L_{3,\langle 2,1,0\rangle}$	$L_{3,\langle 2,0,1\rangle}$	$L_{3,\langle 1,1,1\rangle}$
CV_1	1,5,6	1,4,6	1,4,5	
CV_2	2,6,1	2,5,1	2,5,6	
CV_3	3,1,2	3,6,2	3,6,1	
CV_4	4,2,3	4,1,3	4,1,2	4,6,2
CV_5	5,3,4	5,2,4	5,2,3	5,1,3
CV_6	6,4,5	6,3,5	6,3,4	

Table 1: Coalition value calculation shares (CV) for all coalitions of size $s = 3$, when there are $n = 6$ agents.

i to compute the coalition's value. The second agent ID j in the coalition array will be $= (i + t_0 + 1) \bmod n$; and the third agent ID k will be $= ((i+t_0+1)+t_1+1) \bmod n$. This continues until the coalition's size s limit has been reached.

Table 1 presents a subset of the coalition arrays, grouped by IAs of size $s = 3$ for $n = 6$ agents. Each column represents a single list $L_{s,\underline{t}}$ for some IA $\underline{t}$, whereas the rows present the *coalition value calculation shares* (CVs) comprising the different coalition arrays with a common first element (where CV_i is agent i's share). The table represents all coalition arrays necessary for coalitions of size $s = 3$. To assign all of the coalitions, multiple IAs are needed; however, every coalition is assigned once and only once. Note that an integer partition may form more than one increment array; for example the two increment arrays $\langle 2, 1, 0\rangle$ and $\langle 2, 0, 1\rangle$ are formed from the {2,1} integer partition.

Four different IAs are required for all the coalitions to be allocated in the above example. The IA $\underline{t}^x = \langle 2, 0, 1\rangle$ is valid as $\{2, 1\}$ is a candidate integer partition of $\mathcal{I}(6 - 3)$ that satisfies $|\{2, 1\}| \le s = 3$. Yet as $|\{2, 1\}| \ne 3$, additional zeros are needed to fill up the IA to make the IA the required size s. If agent 2 used $\underline{t}^x$, the coalition array would comprise:
$= \{i, (i + t^x_0 + 1) \bmod n, ((i + t^x_0 + 1) + t^x_1 + 1) \bmod n\}$
$= \{2, (2 + 2 + 1) \bmod 6, ((2 + 2 + 1) + 0 + 1) \bmod 6\}$
$= \{2, 5, 0\} \equiv \{2, 5, 6\}$
In the above example, the ID 0 was generated. As $0 \equiv n \pmod n$, this is replaced with ID $= n = 6$ in this coalition.

Each IA should be used n times (once for each agent) unless the IA includes a sequence that is repeated throughout the IA. In Table 1, $\langle 1, 1, 1\rangle$ is the only IA with a repeated sequence, with $\{1\}$ being repeated $m = 3$ times. The number of times an IA with a repeating sequence should be used relates to the size of the repeating sequence and is given by r (introduced in the next subsection). The choice of agents that should use this type of IA will depend on the allocation of other coalitions, and is described in Section 4.

If any other IA was used other than the ones listed in Table 1, it would result in a coalition's value being calculated more than once. For example, if agent 6 used $t^y = \langle 1, 2, 0\rangle$, the coalition array $[6, 2, 5]$ would be generated despite this coalition being generated by agent 2 using $t^x = \langle 2, 0, 1\rangle$.

A Distributed Method for Coalition Generation

The *period of* $\underline{t}$, denoted by $\pi(\underline{t})$ is defined as:

$$\min_{1\le p\le s} \underline{t} = \langle t_0, t_1, ..., t_{p-1}, t_0, t_1, ..., t_{p-1}, ..., t_0, t_1, ..., t_{p-1}\rangle$$

Hence, $\underline{t}$ is formed by m identical copies of a sequence of length $\pi(\underline{t})$. Given $C \subseteq \{1, 2, ..., n\}$, an agent ID ag generates C from ag if $C = \{ag_1, ag_2, ..., ag_s\}$ and:

$$ag_i = \begin{cases} ag & \text{if} \quad i = 1 \\ (ag + \phi_i) \mod n & \text{if} \quad (ag + \phi_i) \mod n \ne 0 \\ n & \text{if} \quad (ag + \phi_i) \mod n = 0 \end{cases}$$

where:

$$\phi_i = \sum_{k=0}^{i-2} t_k + (i - 1)$$

Additionally, $C(ag, \underline{t})$ denotes the subset of $\{1, 2, ..., n\}$ generated by the IA $\underline{t}$ from agent ag. It is possible to demonstrate that each $\underline{t}$ only needs to be used by $r = (n \times \pi(\underline{t}))/s$ different agents. If more than r agents use $\underline{t}$ to generate a coalition, then repeated coalitions will be generated. For example, if the chosen IA from Table 1 is $\underline{t}^q = \langle 1, 1, 1\rangle$ then $r = (n \times \pi(\underline{t}^q))/s = (6 \times 1)/3 = 6/3 = 2$ agents should use $\underline{t}^q$, which is true as any other agent using $\underline{t}^q$ would repeat either coalition $\{1, 3, 5\}$ or $\{2, 4, 6\}$. Finally, if $\underline{t}^x$ and $\underline{t}^y$ generate the same coalition C for two different agents $i, j \in C$ (i.e. $C(i, \underline{t}^x) = C(j, \underline{t}^y)$), then $\underline{t}^x$ and $\underline{t}^y$ are classified as belonging to the same *equivalence class*, denoted $\underline{t}^x \approx \underline{t}^y$. For example, the IAs in the following equation belong to the same equivalence class: $C(2, \langle 2, 0, 1\rangle) = C(6, \langle 1, 2, 0\rangle) = \{2, 5, 6\}$. We write $\underline{t}^x \approx \underline{t}^y$ when $t^x = \langle t^y_k, ..., t^y_{s-1}, t^y_0, ..., t^y_{k-1}\rangle$ for some $0 \le k \le s - 1$. Section 5 proves that rather than considering *every* possible IA, it suffices only to consider a *single* representative from each equivalence class $\approx$.

4 The Distributed Coalition Generation Algorithm

The DCG algorithm used by each agent i to generate all of its coalitions in its coalition value calculation share is presented in Algorithm 1. The *balance* pointer is initialised, and references the next agent to calculate a coalition's value. It is similar to the α pointer in Rahwan and Jennings (2007), and its use by SingleSize ensures that all the agents' shares are either equal in size or have a difference in size of $+/-$ 1. Line 21 allows only the next r agents to calculate a coalition, starting from the agent referred to by the *balance* pointer, and continuing in an ascending order. If agent n (i.e. the agent with the highest ID) is assigned to calculate a coalition's value, and z more coalition value calculations are required, then line 21 also allows agents 1 to z to calculate a coalition's value.

The DCG algorithm calls SingleSize for every possible size of the coalitions. This function returns all coalitions of size s in agent i's share. The while loop (lines 18-30) uses a black box function, named build (lines 17 and 29), to determine the next IA of a new equivalence class. This new IA is used to generate another coalition for agent i, if i is one of the next r agents to be assigned a coalition (lines 19 to 23). Regardless of who uses the new IA, the *balance* pointer is updated (lines 24-27). When the build function returns *null*, this indicates that all the coalitions of size s for agent i's calculation share have been found.

Algorithm 1: The Distributed Coalition Generation (DCG) Algorithm

```
1: global int $balance := 1$;
2:
3: function DCG (int n, i) returns $\langle \mathcal{C}^s \rangle_{s=1}^n$;
4: Input: $\langle n, i \rangle$ $(1 \le i \le n)$; where $n$ is the number of
   agents and $i$ the agent ID.
5: Output: $\langle \mathcal{C}^s \rangle_{s=1}^n$; $\mathcal{C}^s = \{C_1^s, C_2^s, \dots, C_k^s\}$,
   $C_j^s \subseteq \{1, 2, \dots, n\}$, $|C_j^s| = s$; where $\langle \mathcal{C}^s \rangle_{s=1}^n$ is the
   collection of coalitions, of all sizes $1 \le s \le n$,
   assigned to agent $i$'s share.
6: begin
7: for (int $s = 1;\ s \le n;\ s++$) do
8:     $\mathcal{C}^s$ := SingleSize(n,s,i);
9: end for
10: return $\langle \mathcal{C}^1, \dots, \mathcal{C}^n \rangle$;
11: end;
12:
13: function SingleSize (int n, s, i) returns $\mathcal{C}^i$;
14: Input: $\langle n, s, i \rangle$ $(1 \le s \le n)$; where $n$ is the number
    of agents, $s$ the size of the coalitions and $i$ the
    agent ID.
15: Output: $\mathcal{C}^i$; $\mathcal{C}^i = \{C_1^i, C_2^i, \dots, C_k^i\}$,
    $C_j^i \subseteq \{1, 2, \dots, n\}$, $|C_j^i| = s$; where $\mathcal{C}^i$ is the
    collection of coalitions (of size $s$) assigned to agent
    $i$'s share.
16: begin
17: $t^y$ := build(n,s,0);
18: while ($t^y \neq null$) do
19:     $p := \pi(\underline{t})$;
20:     $r := (n \times p)/s$;
21:     if ($balance \le i < balance + r$) or ($balance + r$
        $> n$ and $1 \le i < balance + r - n$) then
22:         $C := C(i, \underline{t})$;
23:         $\mathcal{C}^i := \mathcal{C}^i \cup C$;
24:         $balance := balance + r$;
25:         if $balance > n$ then
26:             $balance := balance - n$;
27:         end if
28:     end if
29:     $t^y$ := build(n,s,y+1);
30: end while
31: return $\langle \mathcal{C}_1^i, \dots, \mathcal{C}_k^i \rangle$;
32: end
```

A possible implementation of the black box `build` function is presented in Algorithm 2. It relies on an indexing scheme that defines the function $place : \underline{t} \to \mathbb{N}_0$ to map an IA $\underline{t} = \langle t_0, \dots, t_{s-1} \rangle$ to a non-negative integer; i.e.

$$place(\langle t_0, t_1, \dots, t_{s-1} \rangle) = \sum_{i=0}^{s-1} (t_i \times (n - s + 1)^i)$$

Informally, $place(\underline{t})$ treats $\underline{t}$ as an integer expressed in base $n - s + 1$. With this convention, the number of distinct IAs is bounded by $(n - s + 1)^s$. For example, using the *place* function for the IA $t^x = \langle 2, 1, 0 \rangle$, would give the value $val = 2 + 4 + 0 = 6$ because: $(t_0^x = 2) \times (6 - 3 + 1)^0 = 2$; $(t_1^x = 1) \times (6 - 3 + 1)^1 = 4$; and $(t_2^x = 0) \times (6 - 3 + 1)^2 = 0$.

Algorithm 2: One possible method to find representative IAs from equivalence classes of $\approx$.

```
1: function decode (int n, s, k) returns $s$-tuple;
2: Input: $\langle n, s, k \rangle$; where $n$ is the number of agents,
   $s$ is the size of the coalition $(1 \le s \le n)$ and $k$ is
   the index position to convert to an $s$-tuple.
3: Output: $\langle t_0, t_1, \dots, t_{s-1} \rangle$ with
   $\sum_{i=0}^{s-1} t_i \times (n - s + 1)^i = k$
4: begin
5: $val := k$;
6: for (int $index = 0;\ index < s;\ index++$) do
7:     $t_{index} :=$ remainder$(val, n - s + 1)$;
8:     $val := (val - t_{index})/(n - s + 1)$;
9: end for
10: return $\langle t_0, t_1, \dots, t_{s-1} \rangle$;
11: end;
12:
13: function build (int n, s, y) returns $s$-tuple;
14: Input: $\langle n, s, y \rangle$; where $n$ is the number of agents,
    $s$ is the size of the coalition $(1 \le s \le n)$ and $y$ is
    the index position to start searching from
    $(0 \le y < (n - s + 1)^{s+1})$.
15: Output: $\langle t_0^k, t_1^k, \dots, t_{s-1}^k \rangle$; the next IA of a class
    $[\approx]_i$ not used so far.
16: begin
17: for (int $k = y;\ k < (n - s + 1)^{s+1};\ k++$) do
18:     $t^k = \langle t_0^k, \dots, t_{s-1}^k \rangle$ := decode(n,s,k);
19:     int $tot := \sum_{i=0}^{s-1} t_i^k$;
20:     if $tot = n - s$ and $\neg Used[k]$ then
21:         $Used[k] :=$ true
22:         $p := \pi(\underline{t}^k)$;
23:         for ($z = p - 1;\ z > 0;\ z--$) do
24:             $Used[place(\langle t_z^k, \dots, t_{s-1}^k, t_0^k, \dots, t_{z-1}^k \rangle)]$
                := true;
25:         end for
26:         return $t^k$;
27:     end if
28: end for
29: return $null$;
30: end;
```

This `build` function finds the next index value k ($> y$) of a representative of an equivalence class not used so far, while `decode` returns the s-tuple corresponding to the index value $val = k$. Each element of the s-tuple returned by `decode` is at least 0 and at most $n - s$; however, the s-tuple is not necessarily an IA as defined earlier (as it may not sum to $n - s$). This `build` function filters out any s-tuples returned by the `decode` function (lines 19-20) that either: (i) do not sum to $n - s$; or (ii) are a member of an equivalence class previously used. This `build` function knows which equivalence classes have been used so far since all the indexes of IAs of the same equivalence class are marked as used in the Boolean $Used$ array (lines 21-25). When this `build` function finds an s-tuple that is not filtered out by (i) and (ii), then this s-tuple is an IA of an equivalence class not previously used, and is returned (line 26). If no IA of a new equivalence class can be found, then $null$ is returned (line 29).

To illustrate how the `decode` function works: decoding $val = 6$ gives as expected $t^x = \langle 2,1,0\rangle$, as: $t_0^x = \mathbf{remainder}(6, 6-3+1) = \mathbf{remainder}(6,4) = 2$ while val becomes $val := (6-2)/(6-3+1) = 4/4 = 1$; $t_1^x = \mathbf{remainder}(1, 6-3+1) = \mathbf{remainder}(1,4) = 1$ while val becomes $val := (1-1)/(6-3+1) = 0/4 = 0$; and $t_2^x = \mathbf{remainder}(0, 6-3+1) = \mathbf{remainder}(0,4) = 0$.

Finally, if all the agents are required to only store one coalition in memory at a time (i.e. perform DCG with minimal memory requirements), then the following modifications to Algorithm 1 need to be made: (a) do not use the $\langle \mathcal{C}^s\rangle_{s=1}^n$ data object of line 3 and 10; (b) do not use the $\mathcal{C}^i$ data object of line 15 and 31; and (c) replace line 23 with *perform* $v(C)$.

5 Theoretical Evaluation

To evaluate the DCG algorithm, we prove that the algorithm will generate all of the coalitions possible for a community of n agents; and that each coalition is assigned once and only once, thus eliminating redundancy. We start with the following Lemma that shows that all coalitions will be generated:

Lemma 5.1. *Let $C \subseteq \{1,2,\dots,n\}$ with $|C| = s$. There is an IA $\underline{t}$ and $i \in C$ such that $C = C(i,\underline{t})$.*

Proof. Let $C = \{x, x_0, x_1, \dots, x_{s-2}\}$. Without loss of generality we may assume:

$$x < x_0 < x_1 < \cdots < x_i < x_{i+1} < \cdots < x_{s-2}$$

It suffices to find $\underline{t} = \langle t_0, \dots, t_{s-1}\rangle$ with $C(x,\underline{t}) = C$ and $\sum_{i=0}^{s-1} t_i = n - s$, i.e.:

$$\begin{aligned} t_0 &= x_0 - (x+1) \\ \cdots \\ t_k &= x_k - \left(\sum_{i=0}^{k-1} t_i + x + k + 1\right) \\ \cdots \\ t_{s-1} &= n - \sum_{i=0}^{s-2} t_i \end{aligned}$$

□

The next Lemma shows that IAs of an equivalent class generate the same coalitions:

Lemma 5.2. *If $\underline{t} \approx \underline{u}$ then*

$$\bigcup_{i=1}^{n} \{\ C(i,\underline{t})\ \} = \bigcup_{i=1}^{n} \{\ C(i,\underline{u})\ \}$$

Proof. Without loss of generality we may assume that $\underline{t} \approx \underline{u}$ is witnessed by the choice $k = s-1$, i.e.

$$\langle u_0, u_1, \dots, u_{s-1}\rangle = \langle t_{s-1},\ t_0,\ t_1, \dots, t_{s-3},\ t_{s-2}\rangle$$

Define φ_r for $1 \le r \le s+1$

$$\varphi_r = \begin{cases} 0 & \text{if} \quad r = 1 \\ \sum_{k=0}^{r-2} t_k + (r-1) & \text{if} \quad 2 \le r \le s+1 \end{cases}$$

Note that $\varphi_{s+1} = n$. Define ψ_r for $1 \le r \le s+1$ via

$$\psi_r = \begin{cases} 0 & \text{if} \quad r = 1 \\ t_{s-1} + \sum_{k=0}^{r-3} t_k + r-1 & \text{if} \quad 2 \le r \le s+1 \end{cases}$$

Comparing respective terms we see that for all $2 \le k \le s$:

$$\psi_k = \varphi_k + (t_{s-1} - t_{k-2})$$

This leads to the following:

$$C(i,\underline{t}) = \begin{cases} C(n - t_{s-1} + i-1, \underline{u}) & \text{if} \quad 1 \le i \le t_{s-1}+1 \\ C(i - t_{s-1}-1, \underline{u}) & \text{if} \quad t_{s-1}+2 \le i \le n \end{cases}$$

To see this, consider when $1 \le i \le t_{s-1}+1$. We have,

$$C(i,\underline{t}) = \bigcup_{k=1}^{s} \{i + \varphi_k\}$$

When $\lambda = n - t_{s-1} + i - 1$ the above is claimed to be:

$$\begin{aligned} C(\lambda,\underline{u}) &= \bigcup_{k=1}^{s} \{\lambda + \psi_k\} \\ &= \{\lambda\} \cup \bigcup_{k=2}^{s} \{\lambda + \varphi_k + t_{s-1} - t_{k-2}\} \end{aligned}$$

Consider the terms:

$$n - t_{s-1} + i - 1 + \varphi_k + t_{s-1} - t_{k-2}$$

For $2 \le k \le s$, from the fact that $\varphi_k = \sum_{j=0}^{k-2} t_j + k - 1$, these are equal to:

$$n - t_{s-1} + i - 1 + \sum_{j=0}^{k-3} t_j + k - 1 + t_{s-1} = n + i + \varphi_{k-1}$$

In total we have, for $1 \le i \le t_{s-1}+1$:

$$\begin{aligned} C(i,\underline{t}) &= \bigcup_{k=1}^{s} \{i + \varphi_k\} \\ C(\lambda,\underline{u}) &= \{n - t_{s-1} + i - 1\} \cup \\ &\quad \bigcup_{k=2}^{s} \{n + i + \varphi_{k-1}\} \end{aligned}$$

Given that $n + i + \varphi_{k-1}$ and $i + \varphi_{k-1}$ are congruent modulo n, which are elements of $C(i,\underline{t})$, the only terms unaccounted for are $\{n - t_{s-1} + i - 1\} \in C(\lambda,\underline{u})$ and $\{i + \varphi_s\} \in C(i,\underline{t})$. For these, however,

$$\begin{aligned} i + \varphi_s &= i + \sum_{j=0}^{s-2} t_j + s - 1 \\ &= i + (n - s - t_{s-1}) + s - 1 \\ &= i + n - t_{s-1} - 1 \end{aligned}$$

When $t_{s-1}+2 \leq i \leq n$ and $\omega = i - t_{s-1} - 1$, it is claimed that

$$C(i, \underline{t}) \;=\; \bigcup_{k=1}^{s} \{i + \varphi_k\}$$

corresponds to:

$$\begin{aligned} C(\omega, \underline{u}) &= \bigcup_{k=1}^{s} \{\omega + \psi_k\} \\ &= \{\omega\} \cup \bigcup_{k=2}^{s} \{\omega + \varphi_k + t_{s-1} - t_{k-2}\} \end{aligned}$$

Now inspecting the terms for $2 \leq k \leq s$:

$$i - t_{s-1} - 1 + \varphi_k + t_{s-1} - t_{k-2}$$

These, again, simplify to $i + \varphi_{k-1}$, so that

$$\begin{aligned} C(i, \underline{t}) &= \bigcup_{k=1}^{s} \{i + \varphi_k\} \\ C(\omega, \underline{u}) &= \{\omega\} \cup \bigcup_{k=2}^{s} \{i + \varphi_{k-1}\} \end{aligned}$$

When $1 \leq k \leq s-1$, the term $i+\varphi_k$ appears in both $C(i, \underline{t})$ and $C(\omega, \underline{u})$. For the terms $i+\varphi_s \in C(i, \underline{t})$ and $\omega \in C(\omega, \underline{u})$ we have already seen that $i + \varphi_s = i + n - t_{s-1} - 1$ is congruent modulo n with $i - t_{s-1} - 1 = \omega$. This establishes the claim of the Lemma. $\square$

Lemma 5.2 shows that IAs belonging to the same equivalence class of $\approx$ generate exactly the same set of coalitions. Our next two results establish that this is the *only* way in which two distinct IAs can produce the same coalition.

Lemma 5.3. *Let $C = \{x_1, x_2, \ldots, x_i, \ldots, x_s\}$ and $C = C(x_i, \underline{t})$ with $x_i < x_{i+1}$ for all $1 \leq i < s$. There is an IA $\underline{u}$, for which $\underline{t} \approx \underline{u}$ and $C(x_1, \underline{u})$ generates C in strictly increasing ordering of x_i, i.e. $\forall\, 2 \leq i \leq s$:*

$$x_i \in \left\{ x_1 + \sum_{k=0}^{i-2} u_i + i - 1, x_1 + \sum_{k=0}^{i-2} u_i + i - 1 - n \right\}$$

Proof. Given $\underline{t}$, suppose:

$$C(x_i, \underline{t}) \;=\; \{x_1, x_2, \ldots, x_i, \ldots, x_s\}$$

First observe that the terms

$$x_i \;+\; \sum_{k=0}^{r-2} t_k \;+\; r - 1 \;=\; x_i \;+\; \varphi_r$$

are strictly increasing. It follows that if $x_i \neq x_1$ there must be a *unique* index, p, for which:

$$x_i \;+\; \varphi_r \quad \text{is} \quad \begin{cases} \leq n & \text{if } r < p \\ > n & \text{if } r \geq p \end{cases}$$

In consequence, $x_1 \;=\; x_i + \varphi_p - n$ otherwise we cannot have $x_1 \in C(x_i, \underline{t})$. More generally, it must hold that:

$$\begin{aligned} x_k &= x_i + \varphi_{p+k-1} - n && \forall\, 1 \;\leq\; k \;\leq\; s - p + 1 \\ x_k &= x_i + \varphi_{p-(s-k)-1} && \forall\, s - p + 2 \;\leq k \;\leq s \end{aligned}$$

This, however, corresponds to the behaviour of the IA $\underline{u}$, whose definition is:

$$\underline{u} \;=\; \langle t_{p+1},\, t_{p+2},\, \ldots,\, t_{p+k},\, \ldots,\, t_s,\, t_0,\, \ldots,\, t_p \rangle$$

Clearly $\underline{u} \approx \underline{t}$ and $C(x_1, \underline{u}) = C(x_i, \underline{t})$ as claimed. $\square$

As a consequence of Lemma 5.3 we obtain:

Lemma 5.4. *Let $\underline{t}$ and $\underline{u}$ be IAs for which $\underline{t} \not\approx \underline{u}$, then:*

$$\bigcup_{i=1}^{n} \{C(i, \underline{t})\} \bigcap \bigcup_{i=1}^{n} \{C(i, \underline{u})\} \;=\; \emptyset$$

Proof. Suppose the contrary and that $C = \{x_1, \ldots, x_s\}$ can be generated by $C(x_i, \underline{t})$ and $C(x_j, \underline{u})$ for choices of $\underline{t}$ and $\underline{u}$ of different equivalence classes of $\approx$. As a consequence of Lemma 5.3 we know that there are IAs, $\underline{t}'$ and $\underline{u}'$ for which:

$$\begin{aligned} &\underline{t} \approx \underline{t}',\ \underline{u} \approx \underline{u}' \text{ and} \\ &C(x_1, \underline{t}') \;=\; C(x_i, \underline{t}) \;=\; C(x_j, \underline{u}) \;=\; C(x_1, \underline{u}') \end{aligned}$$

Furthermore, $C(x_1, \underline{t}')$ and $C(x_1, \underline{u}')$ produce the elements of C in increasing ordering of $x_i \in C$. This, however, is only possible if

$$x_i \;=\; x_1 \;+\; \sum_{k=0}^{i-2} t'_k \;+\; i - 1 \;=\; x_1 \;+\; \sum_{k=0}^{i-2} u'_k \;+\; i - 1$$

that is, $t'_i = u'_i$ for each $0 \leq i \leq s-1$. This, however, implies that $\underline{t} \approx \underline{u}$, in contradiction to our starting premise. $\square$

Finally, the following Theorem states that each IA chosen by the DCG algorithm is required to be used r times.

Theorem 5.5. *For any IA $\underline{t}$, and for all $1 \leq i \leq j \leq n$,*

$$C(i, \underline{t}) = C(j, \underline{t}) \Leftrightarrow$$

$$\exists\, 0 \leq r \leq \frac{(n-i)s}{n\pi(\underline{t})} \;:\; j = i + r\left(\frac{n\pi(\underline{t})}{s}\right)$$

Proof. We show that if and only if $j = i + (rn\pi(\underline{t})/s)$ for r in the range stated, then $C(i, \underline{t})$ and $C(j, \underline{t})$ are identical. This is completed using the previously introduced Lemmas. The proof is omitted for brevity. $\square$

6 Comparison to Related Work

Table 2 shows that unlike related approaches (Section 2), the DCG algorithm satisfies *all* of the following properties:

i *Communication is eliminated* as the algorithm determines the allocation of coalitions to each agent. Thus, no further coordination between the agents is necessary.

ii The inclusion of the *balance* pointer ensures that agents calculate *approximately equal* shares of coalitions, as it minimises the maximum difference between the size of any two agents' shares (to one).

iii *Redundancy is eliminated* as: (a) only one IA of each equivalence class is used; (b) each IA of a new equivalence class is used only r times; and (c) IAs of different equivalence classes cannot generate the same coalition.

Property	**SK**	**DCVC**	**VBFR**	**DCG**
Eliminates Comms.	No	Yes	Yes	*Yes*
Approx. Equal Shares	No	Yes	No	*Yes*
Eliminates Redundancy	No	Yes	Yes	*Yes*
Equal Operations	No	No	No	*Yes*
Coalition Self Assessed	Yes	No	Yes	*Yes*

Table 2: A comparison of the properties of the DCG algorithm and the three related algorithms (Section 2).

iv An *equal number of operations* will be performed by agents that are allocated equal sized shares, as each IA requires the exact same number of operations of additions to find the corresponding coalition.

v *Every coalition is assessed by one of its member agents* as the first agent generated for each coalition in agent i's allocated share is agent i itself.

The DCG algorithm is better suited for adversarial environments compared to the related approaches, as one of the agents in each coalition will be allocated that coalition's value calculation, while DCG additionally balances the value calculations efficiently across the community of agents in a manner that incurs no communication overheads.

7 Future Work

There are many intriguing avenues for future work. The DCG algorithm could be exploited by distributed coalition structure generation (CSG) algorithms to find the optimal coalition structure. The first such algorithm for solving the CSG problem optimally was D-IP (Michalak et al. 2010), which used the DCVC algorithm (Rahwan and Jennings 2007) as input. Using DCG as input to a distributed CSG algorithm could result in new and interesting properties compared to D-IP.

In competitive or adversarial environments, there is a possibility that self-interested agents may attempt to deceive others to gain an advantage. Within our DCG algorithm, an agent *may* benefit by artificially increasing the value of one (or more) of its coalition value calculations; however, we considered deception out of scope for this paper, and focussed instead on the properties of the DCG algorithm itself. An interesting line of research would be to investigate how to make DCG *incentive compatible*[2] or *near-incentive compatible*[3] (Blankenburg et al. 2005).

Finally, an investigation is underway to determine an optimal approach (in terms of efficiency) for the black box build function that is used within DCG. Preliminary results suggest that the optimal approach will run in time according to the number of equivalence classes of IAs and with storage requirements according to the number of agents.

8 Conclusions

In this paper, the *Distributed Coalition Generation Algorithm* was presented that distributes the coalition value calculations across a community of agents. This algorithm is based on a mechanism that exploits integer partitions to generate increment arrays that represent the difference between agent IDs when a coalition is represented in an ordered sequence. The algorithm has been evaluated theoretically, resulting in the proofs that: (a) all coalitions are assigned; and (b) each coalition is assigned once and only once.

[2]Where the agents fare best when they are truthful.

[3]Where the agents cannot determine how to lie profitably.

9 Acknowledgements

Luke Riley is grateful for support from the EPSRC. We also thank the anonymous reviewers for their useful comments.

References

Blankenburg, B.; Dash, R. K.; Ramchurn, S. D.; Klusch, M.; and Jennings, N. R. 2005. Trusted kernel-based coalition formation. In *Proceedings of the 4th International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 989–996.

Cesco, J. C. 1998. A convergent transfer scheme to the core of a TU-game. *Revista de Matematicas Aplicadas* 19:23–35.

Chalkiadakis, G.; Elkind, E.; and Wooldridge, M. 2011. *Computational Aspects of Cooperative Game Theory*. Morgan & Claypool Publishers.

Michalak, T.; Sroka, J.; Rahwan, T.; Wooldridge, M.; McBurney, P.; and Jennings, N. R. 2010. A distributed algorithm for anytime coalition structure generation. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent System (AAMAS)*, 1007–1014.

Osbourne, M. J., and Rubinstein, A. 1994. *A Course in Game Theory*. MIT Press.

Rahwan, T., and Jennings, N. R. 2007. An algorithm for distributing coalition value calculations among cooperating agents. *Artificial Intelligence* 171:535–567.

Sandholm, T. W., and Lesser, V. R. 1997. Coalitions among computationally bounded agents. *Artificial Intelligence* 94:99–137.

Sandholm, T. W.; Larson, K.; Andersson, M.; Shehory, O.; and Tohme, F. 1999. Coalition structure generation with worst case guarantees. *Artificial Intelligence* 111:209–238.

Shehory, O., and Kraus, S. 1995. Task allocation via coalition formation among autonomous-agents. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI)*, 655–661.

Shehory, O., and Kraus, S. 1996. Formation of overlapping coalitions for precedence-order task-execution among autonomous agents. In *Proceedings of the 2nd International Conference on Multiagent Systems (ICMAS)*, 330–337.

Shehory, O., and Kraus, S. 1998. Methods for task allocation via agent coalition formation. *Artificial Intelligence* 101:165–200.

Vinyals, M.; Bistaffa, F.; Farinelli, A.; and Rogers, A. 2012. Coalitional energy purchasing in the smart grid. In *Proceedings of the IEEE International Energy Conference & Exhibition (ENERGYCON)*, 848–853.

Wu, L. S.-Y. 1977. A dynamic theory for the class of games with nonempty cores. *SIAM Journal on Applied Mathematics* 32:328–338.