Representation of Coalitional Games with Algebraic Decision Diagrams

Karthik Aadithya
Tomasz Michalak
Nicholas Jennings

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2011-8
http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-8.html

January 30, 2011

Representation of Coalitional Games with Algebraic Decision Diagrams

Karthik .V. Aadithya*‡, Tomasz P. Michalak†, and Nicholas R. Jennings†
*Department of Electrical Engineering and Computer Sciences, The University of California, Berkeley, CA, USA
†School of Electronics and Computer Science, University of Southampton, UK
‡Contact author. Email: kv.aadithya@gmail.com

***Abstract*—With the advent of algorithmic coalitional game theory, it is important to design coalitional game representation schemes that are both compact and efficient with respect to solution concept computation. To this end, we propose a new method for representing coalitional games. We show that our representation (a) is fully expressive (i.e., can be used to represent any coalitional game), (b) is compact (i.e., has size polynomial in the number of agents) for many games of practical interest, (c) enables polynomial time Banzhaf Index and Shapley Value computation, (d) enables polynomial time algorithms for several core-related questions, such as testing if a given vector is in the core, checking if the core is empty and computing the smallest ϵ such that the strong-ϵ core is non-empty, and (e) enables polynomial time cost of stability computation. To the best of our knowledge, no existing coalitional game representation offers all these advantages. The core data structure behind our representation is the Algebraic Decision Diagram (ADD), which is a widely applied and well-researched topic in the Electrical Engineering (EE) community. Borrowing ideas from the EE literature, we have also been able to prove a previously unknown, powerful, positive result that enables efficient solution concept computation for a wide range of coalitional games. Hence we are hopeful that our representation opens the doors for using the rich corpus of ADD-related EE literature for advancing the field of algorithmic coalitional game theory.**

I. Introduction

The study of interactions among *multiple, autonomous, self-interested intelligent agents, who can form coalitions* in order to achieve common goals, realize collective payoffs or share common goods/costs, is an important and recurring theme in multi-agent systems. Coalitional game theory provides a mathematical framework for modelling and analysing such interactions. Historically, the field of coalitional game theory was mainly concerned with *developing solution concepts* for predicting the outcomes of such interactions, namely, which coalitions would be formed and how the gains from co-operation would be divided amongst coalition members. More recently however, with the advent of *algorithmic game theory*, there is an increased emphasis on *developing efficient algorithms* for computing such solution concepts [1].

With the new emphasis on solution concept *computation*, the key question is: "How should coalitional games be represented so that solution concepts can be computed as efficiently as possible?". The traditional characteristic function representation (which maps every subset of agents to a real number) is no longer considered adequate because, irrespective of the game, it is always of length exponential in the number of agents. *Compactness* is, therefore, one of the key properties desirable in a representation scheme for coalitional games [2], [3]. However, elementary counting arguments show that any representation scheme that is *fully expressive* (i.e., can be used to represent any coalitional game) cannot be always compact (i.e., there will exist many games for which the representation would require exponential number of bits). Hence we focus on designing *a fully-expressive representation that is compact for most games of practical interest.*

Apart from *compactness*, the other key requirement is *computational efficiency*, i.e., the representation should enable *efficient (polynomial time) algorithms for answering game-theoretic questions* about widely accepted solution concepts such as the core, the Banzhaf Index and the Shapley Value [2], [3]. The most important problems in this context include (a) testing if a given payoff vector is in the core (`TEST CORE`), (b) checking if the core is empty (`EMPTY CORE`), (c) computing the smallest ϵ such that the strong-ϵ core is non-empty (ϵ-`CORE`), (d) computing the cost of stabilising the grand coalition (`CoS`), (e) computing the Banzhaf Indices of all agents (`BI`), and (f) computing the Shapley Values of all agents (`SV`) [4]–[8].

This paper presents a new method of representing coalitional games, under which *all the six problems above can be solved in polynomial time*. Our representation is *fully-expressive* and it is also *compact for many games of interest*. Moreover, our representation has a *highly intuitive construction*, i.e., given a description for a coalitional game in plain English (such as in Examples 1-5 of §II), the process of converting this description into our representation is often very natural and straightforward.

To the best of our knowledge, *no existing representation offers all the above advantages*. Indeed, our analysis in §II shows that existing representations are often inadequate — either they blow up in size or the solution concepts become computationally intractable — even for games that can be described fairly simply in words and analysed fairly easily using back-of-the-envelope combinatorics. To avoid such shortcomings, the *design philosphy* behind our representation is that: the process of constructing the representation from a word description should be as quick and intuitive as possible, and as easy as asking a series of questions of the form "What happens if a particular agent is present in the coalition? What happens otherwise?".

The core data structure behind our representation is the Algebraic Decision Diagram (ADD)[1] [9], which, as we show in §III, is ideally suited to implement the above design philosophy. ADDs are, in fact, well-known and widely used in the Electrical Engineering (EE) community to efficiently represent and analyse real-valued functions of boolean vector-valued arguments. To the best of our knowledge, this paper is the first to recognize that ADDs can also be highly useful, compact representations for coalitional games. For instance, in §III, we illustrate the power of ADDs to compactly represent coalitional games, using as examples the same easy-to-describe games on which existing representation schemes were earlier (in §II) shown to be inadequate.

Besides being compact, ADDs are also *computationally efficient*; indeed, in §IV, we demonstrate that *many game-theoretic questions can be reduced to efficiently solvable ADD problems*. Capitalizing on this, we develop efficient (polynomial time) ADD-based algorithms for solving the six problems above, namely, `TEST CORE`, `EMPTY CORE`, ϵ-`CORE`, `CoS`, `BI` and `SV`. For each of these problems, we present readily implementable pseudocode of a polynomial time solution.

Perhaps the greatest advantage of adopting an ADD-based coalitional game representation is the *rich corpus of EE literature on ADD construction, manipulation and analysis*. For example, drawing upon the EE literature on ADD construction for symmetric boolean functions [10], we have been able to prove (in §VI) that a wide range of coalitional games (roughly, all those games where the agents are partitioned into a fixed number of distinct "types", where the value of a coalition depends only on the number of agents of each type included in the coalition) are efficiently solvable for the core, cost of stability, Banzhaf Indices and Shapley Values. We believe that many more positive results will be discovered in the future, based on exploring the connection between ADDs and coalitional games. Our representation, therefore, opens the doors for using the vast EE literature on ADDs to advance the field of algorithmic coalitional game theory.

II. Previous work

We now present an overview of existing techniques for coalitional game representation. Backed by suitable examples, we highlight the *pros* and *cons* of each representation technique.

Characteristic function representation: This is the traditional way to represent a coalitional game. It consists of a tuple $\langle N, \nu \rangle$ where N is a

[1]also known as Multi-Terminal Binary Decision Diagram

set of agents and $\nu : 2^N \rightarrow \mathbb{R}$ is a *characteristic function* that maps every subset of agents to a real number, with $\nu(\emptyset) = 0$. Mathematically, this representation is also a *formal definition* for the concept of a coalitional game.

Pros: Fully expressive (by definition).
Cons: Always of length exponential in the number of agents; hence no solution concept can be computed efficiently.

Induced subgraph representation: This representation, proposed by Deng and Papadimitriou [1], consists of an undirected, edge-weighted graph $G(V, E)$ where V is a (finite) set of agents and $E : V \times V \rightarrow \mathbb{R}$ is a symmetric function that maps every *pair* of agents to a real number. The value $\nu(C)$ of a coalition $C \subseteq V$ is the total weight of all edges in the subgraph of G induced by C.

Pros: Always compact. BI and SV are easy (in P).
Cons: Not fully expressive. CoS and all core-related questions are hard (NP-Hard or worse).

Unrestricted Marginal Contribution Network (MC-Net) representation: This representation, proposed by Ieong and Shoham [2], consists of a (finite) set of "rules" of the form *Pattern* $\rightarrow$ *Value*, where *Pattern* is a *propositional formula* over the set of agents N and *Value* is a real number. Given a coalition $C \subseteq N$, the *Pattern* part of each rule is first evaluated under the truth assignment $x \iff x \in C$, for every $x \in N$. Then the *Value* parts of *only those rules whose Pattern part is satisfied by* C are summed up to yield the value $\nu(C)$.

Example 1. The 1-of-2 games $\mathbf{G_n^{1/2}}$. This is a family of games where the n^{th} game $G_n^{1/2}$ has $2n$ agents $\{x_1 \ ... \ x_n, \ y_1 \ ... \ y_n\}$. For every i, the agents x_i and y_i are *substitutes* for each other, but not for any other agent. Therefore, a coalition is *winning* (has a value 1) iff, for every i, at least one of $\{x_i, y_i\}$ is present in the coalition. All non-winning coalitions have zero value.

It is straightforward and intuitive to translate the above word description into an MC-Net for $G_n^{1/2}$. Indeed, the resulting MC-Net has only one rule:

$$\bigwedge_{i=1}^{n} (x_i \vee y_i) \rightarrow 1$$

Example 2. The 2-of-3 games $\mathbf{G_n^{2/3}}$. In this family, the n^{th} game $G_n^{2/3}$ has $3n$ agents $\{x_1 \ ... \ x_n, \ y_1 \ ... \ y_n, \ z_1 \ ... \ z_n\}$. For every i, any *pair* of agents in $\{x_i, y_i, z_i\}$ is a *substitute* for any other *pair*. Therefore, a coalition is *winning* iff, for every i, at least two of $\{x_i, y_i, z_i\}$ are present in the coalition.

Translating the above word description into an MC-Net is also intuitive and straightforward. The resulting MC-Net again has only one rule:

$$\bigwedge_{i=1}^{n} [(x_i \wedge y_i) \vee (y_i \wedge z_i) \vee (z_i \wedge x_i)] \rightarrow 1$$

Pros: Fully expressive. Compact and intuitive construction for many games of interest (such as the 1-of-2 and 2-of-3 games above).
Cons: All solution concepts are hard.

Basic MC-Net representation: This representation, also proposed by Ieong and Shoham [2], makes MC-Nets more tractable for BI and SV by imposing the restriction that the *Pattern* part of each MC-Net rule can only be a conjunction of literals. This restriction does not compromise the *fully expressive* property of MC-Nets. However, it seriously undermines the "intuitiveness" of MC-Net construction. For example, under the added restriction, it is no longer straightforward and intuitive to construct MC-Nets for the 1-of-2 and 2-of-3 games above. Indeed, it is proved in [3] that all basic MC-Nets for the 1-of-2 games with positive *Values* necessarily contain exponentially many rules. Thus, the added restriction also appears to have seriously compromised on compactness.

Pros: Fully expressive. Compact for some games (including all induced-subgraph games). SV and BI are easy.
Cons: Construction is often un-intuitive/exponential (even for easy-to-describe games like the 1-of-2 and 2-of-3 games). CoS and all core-related questions are hard.

Read-once MC-Net representation: This representation, proposed by Elkind et. al. [3], allows a larger class of *Patterns* compared to basic MC-Nets, without sacrificing on efficient BI and SV computation. In this representation, a *Pattern* can be any *read-once boolean formula*, i.e., any boolean formula wherein each variable appears at most once. Under this relaxed condition, some games that were previously intractable (using only basic MC-Net rules), now become tractable. For example, the 1-of-2 MC-Net above is not a basic MC-Net, but it is a read-once MC-Net. However, the 2-of-3 MC-Net above is neither basic nor read-once, so the 2-of-3 game is still intractable.

Pros: Fully expressive. Compact for many games for which no compact basic MC-Net is likely to exist (e.g., the 1-of-2 games). SV and BI are easy.
Cons: Construction can be un-intuitive/exponential (even for easy-to-describe games like the 2-of-3 games). CoS and all core-related questions are hard.

We now present additional pathological examples of easy-to-describe and easy-to-analyse games, for which constructing an MC-Net is nevertheless extremely un-intuitive and likely exponential.

Example 3. The Majority games $\mathbf{G_n^M}$. Here the n^{th} game G_n^M has $2n+1$ agents $\{x_1 \ ... \ x_{2n+1}\}$. A coalition is winning iff it is a majority, i.e., its size is more than n.

Majority games are of practical interest because they are one of the few classes of *weighted voting games* that are easy to analyse. For example, just by inspection, the SV of each agent in G_n^M is $\frac{1}{2n+1}$, while the BI of each agent is $\frac{1}{2^{2n}}\binom{2n}{n}$. The core of G_n^M is empty, the CoS is $n/(n+1)$ and the strong-ϵ core is non-empty for every $\epsilon \geq n/(2n+1)$. However, in spite of being so easy to describe and analyse, no compact MC-Net is known for the majority games.

Example 4. The Glove games $\mathbf{G_{m,n}^{gl}}$. The game $G_{m,n}^{gl}$ has $m+n$ agents $\{l_1 \ ... \ l_m, \ r_1 \ ... \ r_n\}$. Each *left agent* l_i has one *left* glove, while each *right agent* r_j has one *right* glove. The value of a coalition is the number of *pairs* of gloves held by the coalition, i.e., if a coalition C has θ_1 left agents and θ_2 right agents, then $\nu(C) = \min(\theta_1, \theta_2)$.

Glove games are of practical interest because they are one of the few classes of *coalitional resource games* that are easy to analyse. For instance, by recognizing that a left (right) agent makes a positive marginal contribution to a coalition iff the coalition already contains more right (left) agents than left (right) agents, elementary enumerative combinatorics is sufficient to solve the BI and SV problems. The core is also easily characterised: for $m < n$ ($m > n$), the core has exactly one payoff vector that assigns 1 to every left (right) agent and 0 to every right (left) agent; for $m = n$, the core comprises all vectors of the form $[\underbrace{x \ ... \ x}_{m \text{ times}}, \underbrace{y \ ... \ y}_{n \text{ times}}]$, where $x \geq 0, \ y \geq 0$ and $x + y = 1$. Again, despite their simplicity of analysis, no compact MC-Net representation is known for the glove games.

Example 5. The Square games $\mathbf{G_n^{sq}}$. In this family of games, G_n^{sq} has n agents $\{x_1 \ ... \ x_n\}$. The value of a coalition is the square of the size of the coalition, i.e., if a coalition C has θ members, then $\nu(C) = \theta^2$.

Square games are among the simplest examples of *super-additive games*. They are also easy to analyse. For example, just by inspection, the BI of each agent in G_n^{sq} is n (and so is the SV). It is also easy to test if a given vector is in the core: just sort the vector, compute a cumulative sum and check if the result is element-wise at least $[1, \ 4, \ 9 \ ... \ n^2]$. Again, in spite of this extreme simplicity, a compact MC-Net representation for G_n^{sq} is yet to be found.

The above discussion brings out three important shortcomings of existing coalitional game representations: (a) existing representations are not compact for many games of practical interest, (b) there is no standard procedure for translating a word description into the existing representations, and (c) CoS and all core-related questions are NP-Hard or worse in all the existing representations.

By contrast, our ADD-based representation (described in subsequent sections) is both *compact* and *intuitive to construct*, even for the pathological examples above. Moreover, under our representation, efficient (polynomial time) algorithms exist for BI, SV, CoS and all core-related

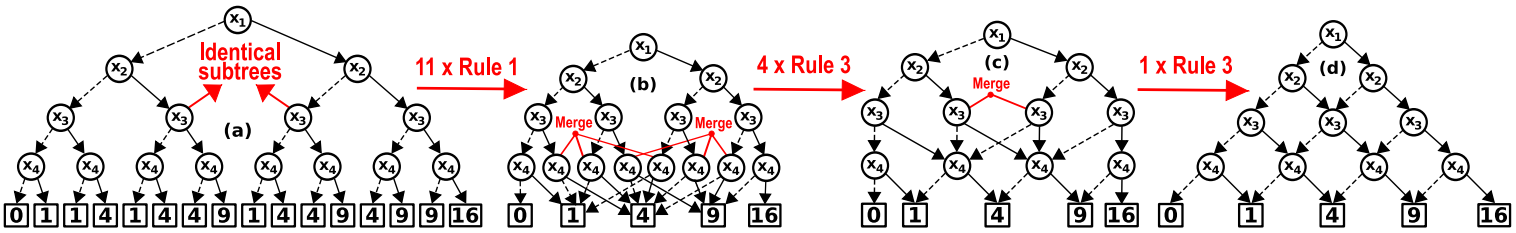

Fig. 1. ADD construction from decision tree for the square game G_4^{sq}.

questions.

III. Our representation based on Algebraic Decision Diagrams

In this section, we illustrate the power of ADDs to compactly represent coalitional games.

A. The key ADD idea: Remove duplication from decision trees

ADDs are, in essence, highly optimized representations for ordered decision trees. In general, a decision tree is of size exponential in the number of decision variables. However, the observation is that *most practically encountered decision trees contain a significant amount of duplication*, i.e., there exist many subtrees within the decision tree that are isomorphic to one another.

For example, consider the ordered decision tree for the square game G_4^{sq} from Eg. 5, which is shown in Fig. 1 (a). In the figure, each non-terminal node (decision node) is labelled with an agent (the corresponding decision variable). Moreover, each decision node has exactly two edges leading away from itself, one dashed and the other solid. The left (right) child of each decision node, obtained by following the dashed (solid) edge, corresponds to an *exclude (include)* decision, i.e., the agent is excluded from (included in) the coalition. Coalition values are specified by the terminal nodes. It is readily seen that this decision tree contains significant duplication (e.g., consider the identical sub-trees rooted at the nodes labelled x_3, as pointed out in Fig. 1 (a)).

The fundamental idea behind the ADD is that: *it is wasteful to maintain multiple identical copies of duplicated subtrees; instead, such isomorphic subtrees should be merged together*, thereby resulting in a much smaller (but equivalent) directed acyclic graph (DAG) [9], [11]. To this end, three *reduction rules* have been formulated for compressing a decision tree into a DAG [11]:

Rule 1: *Merge isomorphic terminal nodes.* That is, if two terminal nodes u and v carry the same value, delete u and redirect all its incoming edges to v.

Rule 2: *Delete dummy nodes.* That is, if the left child of a decision node u is the same as its right child, then delete u and redirect all its incoming edges to this (only) child.

Rule 3: *Merge isomorphic decision nodes.* That is, if two nodes u and v have (a) identical labels, (b) identical left children and (c) identical right children, delete u and redirect all its incoming edges to v.

For example, the decision tree of Fig. 1 (a) contains four isomorphic terminal nodes with value 1, six isomorphic terminal nodes with value 4 and four isomorphic terminal nodes with value 9. To get rid of all this duplication, Rule 1 (above) is applied 3+5+3=11 times in succession, resulting in the DAG of Fig. 1 (b). This DAG is not free from isomorphic nodes either. In fact, as shown in Fig. 1 (b), it has two sets of three isomorphic nodes each, which can be merged by applying Rule 3 four times in succession, thereby resulting in the DAG of Fig. 1 (c). This DAG again contains two isomorphic nodes (as shown in Fig. 1 (c)), which are merged by a single application of Rule 3. This results in the DAG of Fig. 1 (d), which is *maximally compressed* in the sense that it cannot be made smaller by any further application of Rules 1-3. Such a *maximally compressed* DAG (which can be shown to be a unique and canonical representation for the original decision tree) is called an Algebraic Decision Diagram. Thus, Fig. 1 (d) is an ADD representation for the square game G_4^{sq}.

B. A formal definition for the ADD-based representation

Having explained the fundamentals of ADDs, we now formally define our ADD-based representation for coalitional games. In this representation, a coalitional game is specified by a tuple $\langle N, <, G(V, E, L_V, L_E)\rangle$, where

- N is a finite set (the set of agents)
- $<$ is a strict total order defined on N
- $G(V, E, L_V, L_E)$ is a vertex-labelled, edge-labelled, directed acyclic graph (the ADD) that satisfies the following:
  - V is a finite set (the set of ADD vertices)
  - $E \subset V \times V$ is a finite set (the set of ADD edges)
  - $L_V : V \to N \cup \mathbb{R}$ is a function that labels each ADD vertex with either an agent (for non-terminal vertices) or a real number (for terminal vertices)
  - $L_E : E \to \{\texttt{SOLID, DASHED}\}$ is a function that labels each ADD edge as either `SOLID` or `DASHED`
  - G contains exactly one root/source vertex, i.e., exactly one vertex of in-degree zero
  - For all vertices u and v, if (u, v) is an edge in G, then $u < v$
  - For each non-terminal vertex u, there exists exactly one vertex v, called the left child of u, such that $(u, v) \in E$ and $L_E((u, v)) =$ `DASHED`
  - For each non-terminal vertex u, there exists exactly one vertex v, called the right child of u, such that $(u, v) \in E$ and $L_E((u, v)) =$ `SOLID`
  - The reduction rules 1-3 of the previous subsection cannot be used to simplify G any further.

C. Noteworthy properties of ADDs

As a consequence of the reduction rules of §III-A, ADDs have many interesting and useful properties. Of these, we now list the properties that are especially relevant to coalitional games.

Sub-ADDs as coalitional games: In an ADD, *every* node u can be thought of as the source node of a unique coalitional game rooted at u. Viewed this way, each ADD node represents a coalitional game in its own right. For instance, the root (source node) of Fig. 1 (d) represents the square game G_4^{sq}. The left child of the root represents another coalitional game, namely, a square game played by the agents $\{x_2, x_3, x_4\}$. The right child of the root represents yet another coalitional game, namely, the game played by agents $\{x_2, x_3, x_4\}$ where the value of a k-sized coalition is $(k+1)^2$. In general, given an ADD with agents $\{x_1 < x_2 < \ldots < x_n\}$, every decision node u with label x_i represents a unique coalitional game played by agents x_i to x_n, whose ADD representation is given by the sub-ADD rooted at u.

Reusability of sub-ADDs: As mentioned above, each sub-ADD of an ADD represents a unique coalitional game. Moreover, each sub-ADD, once created, can be "re-used" again and again at no extra cost. For example, in Fig. 1 (d), the sub-ADD rooted at the middle node labelled x_3 is used twice: once corresponding to the decision "exclude x_1 but include x_2" and once corresponding to the decision "include x_1 but exclude x_2". Likewise, the sub-ADDs rooted at the middle two nodes labelled x_4 are each used twice. In general, given an ADD, a sub-ADD rooted at node u is used as many times as the in-degree of u. This is analogous to dynamic programming: ADD nodes are like memoized solutions to dynamic programming sub-problems; a one-time effort is expended to create them, which pays back many times over. Thus, ADDs provide a

framework that allows simpler coalitional games (rooted at sub-ADDs) to be used as building blocks for constructing more complex coalitional games.

Relationship between an ADD node and its children: For every (non-terminal) ADD node u, there is an intuitive relationship between the coalitional game rooted at u and the coalitional games rooted at the children of u: suppose u has label x_i; then the left (right) child of u represents a coalitional game played by the agents x_{i+1} to x_n, that describes how to evaluate the characteristic function *in the absence (presence) of agent* x_i. In general, for every decision node u with label x_i, the left (right) child of u specifies what happens if agent x_i is excluded from (included in) the coalition. This observation forms the basis of our intuitive procedure (§III-D) for constructing an ADD from a word description of a coalitional game.

Compactness: Often, the reduction rules of §III-A are so powerful that they transform an exponential-sized decision tree into a polynomial-sized ADD [9], [11]. For instance, generalising the ADD construction of Fig. 1, we see that the ADD representation for G_n^{sq} would contain a total of $1+2+\ \ldots\ +(n+1) = (n+1)(n+2)/2$ nodes, which is polynomial in the number of agents n. This is true not only for square games; in fact, as seen from Table I, a polynomial-sized ADD exists (and in §III-D, we show how to construct it) for every single pathological example of §II.

Game	# agents	ADD size
$G_n^{1/2}$	$2n$	$2n+2$
$G_n^{2/3}$	$3n$	$4n+2$
G_n^{M}	$2n+1$	n^2+2n+3
$G_{m,n}^{gl}$	$m+n$	$\frac{1}{6}(\min(m,n))^3 + \frac{11}{6}\min(m,n) + mn + 1$
G_n^{sq}	n	$\frac{1}{2}n^2 + \frac{3}{2}n + 1$

TABLE I. ADD sizes for the pathological examples of §II, indicating that polynomial-sized ADD representations are possible even for games that have no known polynomial MC-Net.

Expressiveness: ADDs are *fully expressive* (i.e., they can be used to represent any coalitional game). This follows from a two-step reasoning: (1) every coalitional game can be represented as a decision tree, and (2) every decision tree can be transformed into an ADD by the reduction rules of §III-A [11].

Importance of agent ordering: Given a coalitional game, the size of its ADD representation often depends strongly on the agent ordering chosen [11]. For instance, consider the 1-of-2 games $G_n^{1/2}$: if the agent ordering is $<_1$: $x_1 < y_1 < x_2 < y_2 <\ \ \ldots\ \ < x_n < y_n$, the ADD size is $2n+2$; but if the agent ordering is $<_2$: $x_1 <\ \ \ldots\ \ < x_n < y_1 <\ \ \ldots\ \ < y_n$, the ADD size shoots up to $2^{n+1}-1$. To achieve compactness, it is therefore crucial to choose a "good" variable ordering. However, for a general ADD, the problem of finding an optimal variable ordering is NP-Hard [12]. Hence we suggest two guidelines that usually result in a good variable ordering: (1) place substitute agents close to each other, and (2) place "less significant" agents ahead of "more significant" agents. For example, the first guideline applied to $G_n^{1/2}$ suggests the ordering $<_1$ above. Similarly, for $G_n^{2/3}$, the first guideline suggests the ordering $x_1 < y_1 < z_1\ \ \ldots\ \ x_n < y_n < z_n$, which results in ADD size $4n+2$. To take another example: for the glove games $G_{m,n}^{gl}$, the first guideline would advocate grouping all the left agents together and all the right agents together. The second guideline would then decide which group to put first: if $m \leq n$ ($m > n$), the group of right (left) agents should come first, followed by the group of left (right) agents (within a group, the ordering is immaterial because of symmetry). This results in the ADD size shown in Table I.

D. The intuitive ADD construction procedure

In §III-A, we described three reduction rules for systematically constructing an ADD from a decision tree. However, for large games, it is not practical to build a decision tree and then convert it to an ADD. Rather, we need an intuitive method to construct an ADD directly from a word description of the coalitional game, i.e., bypassing the decision tree altogether. This subsection describes such a method.

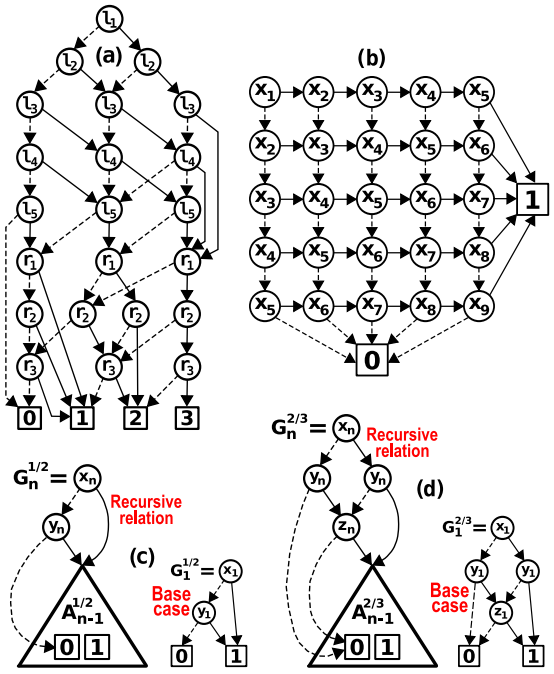

Fig. 2. ADD representations for the pathological examples of §II: (a) ADD for $G_{5,3}^{gl}$, (b) ADD for G_4^M, (c,d) Recursive relations illustrating ADD construction for $G_n^{1/2}$ and $G_n^{2/3}$ respectively.

Our method works in a bottom-up fashion, first constructing (sub) ADDs for simpler coalitional games, and then using these as building blocks for more complex coalitional games. At each decision node so constructed, the key idea is to ask the questions "What happens if this particular agent is excluded from the coalition? What happens otherwise?". If both these answers are identical, the current decision node is a dummy node (i.e., it should not even exist in the ADD). If the answers are different, then the answer to the former question yields the current decision node's left child, while the answer to the latter question yields the right child. This question-answer routine is continued recursively until a terminal node is reached. Moreover, at each decision node, a new child node is created only if no existing sub-ADD answers the corresponding exclusion/inclusion question; otherwise, we just draw an edge from the current decision node to the previously computed sub-ADD that answers the question (i.e., without incurring the cost of creating a new node).

Algorithm 1 illustrates the above question-answer method for the majority games G_n^M. At each decision node, the variable k keeps track of the current coalition size; the moment a majority is attained (or it is determined that no majority can be attained), the current decision node's relevant child is designated the appropriate terminal node (0 or 1). If both outcomes are possible, a recursive call is initiated with an updated value of k. A hash table is used to memoise sub-ADDs. Running this algorithm for $n = 4$ produces the ADD of Fig. 2 (b). From the figure, it is easy to generalise that the ADD for G_n^M would contain exactly $(n+1)^2+2$ nodes, as indicated in Table I.

Algorithm 4 produces ADD representations for the square games, using the same invariant as Algorithm 1.

Algorithm 3 applies the same question answer procedure to the glove games $G_{m,n}^{gl}$ (where $m \geq n$ is assumed without loss of generality). The pattern remains exactly the same as Algorithm 1: answer the exclusion/inclusion questions, use a hash-table that memoises sub-ADDs. But the invariant is more complicated. At each decision node, two variables θ_1 and θ_2 are maintained; θ_1 denotes the number of distinct pairs of gloves already present in the coalition, while θ_2 denotes the maximum number of pairs possible assuming that no more left agents will join the coalition. Running this algorithm with $(m,n) = (5,3)$ produces the ADD of Fig. 2 (a). The generalisation to arbitrary (m,n) is, however,

Algorithms 1-5 illustrate the intuitive construction of ADDs for the pathological examples of §II.

Algorithm 1: ADD creation for G_n^M: Run create_maj_ADD(n, 1, 0, $\emptyset$)

```
function create_maj_ADD(n, label, k, hash_table)
    // invariant: k = size of current coalition
    if hash_table contains key (label, k) then
        return hash_table[(label, k)];
    end
    // answer the exclusion question
    ADDNode left_child;
    if n + k < label then
        left_child = Terminal node with value 0;
    else
        left_child = create_maj_ADD(n, label+1, k, hash_table);
    end
    // answer the inclusion question
    ADDNode right_child;
    if k == n then
        right_child = Terminal node with value 1;
    else
        right_child = create_maj_ADD(n, label+1, k+1, hash_table);
    end
    // combine the exclusion and inclusion answers
    ADDNode curr = new ADDNode (decision variable = x_label);
    curr → left = left_child, curr → right = right_child;
    hash_table[(label, k)] = curr;
    return curr;
end
```

Algorithm 2: ADD creation for $G_n^{1/2}$: Run create_1_of_2_ADD(n)

```
function create_1_of_2_ADD(n)
    if n == 1 return the base case ADD of Fig. 2 (c);
    ADDNode prev = create_1_of_2_ADD(n-1);
    ADDNode X = new ADDNode (decision variable = x_n);
    ADDNode Y = new ADDNode (decision variable = y_n);
    ADDNode ZERO = Terminal node with value 0;
    X → left_child = Y, X → right_child = prev;
    Y → left_child = ZERO, Y → right_child = prev;
    return X;
end
```

Algorithm 3: ADD creation for $G_{m,n}^{gl}$: Run create_gl_ADD(m, n, 1, 0, 0, $\emptyset$)

```
function create_gl_ADD(m, n, label, θ1, θ2, hash_table)
    // invariants: θ1 = # paired gloves, θ2 = min(# unpaired
       left gloves, # undecided right gloves)
    if hash_table contains key (label, θ1, θ2) then
        return hash_table[(label, θ1, θ2)];
    end
    // answer the exclusion question
    ADDNode left_child;
    if label ≤ m then
        if label == m AND θ2 == 0 then
            left_child = Terminal node with value 0;
        else
            left_child = create_gl_ADD(m, n, label+1, 0, θ2, hash_table);
        end
    else
        if label == n + m then
            left_child = Terminal node with value θ1;
        else
            left_child = create_gl_ADD(m, n, label+1, θ1, min(θ2,m+n−label), hash_table);
        end
    end
    // answer the inclusion question
    ADDNode right_child;
    if label ≤ m then
        if θ2 == n − 1 then
            right_child = create_gl_ADD(m, n, m+1, 0, n, hash_table);
        else
            right_child = create_gl_ADD(m, n, label+1, 0, θ2+1, hash_table);
        end
    else
        if label == n + m then
            right_child = Terminal node with value θ1 + 1;
        else
            right_child = create_gl_ADD(m, n, label+1, θ1+1, θ2−1, hash_table);
        end
    end
    // combine the exclusion and inclusion answers
    ADDNode curr = new ADDNode(decision variable = (label ≤ m) ? l_label : r_label−m);
    curr → left = left_child, curr → right = right_child;
    hash_table[(label, θ1, θ2)] = curr;
    return curr;
end
```

Algorithm 4: ADD creation for G_n^{sq}: Run create_sq_ADD(n, 1, 0, $\emptyset$)

```
function create_sq_ADD(n, label, k, hash_table)
    // invariant: k = size of current coalition
    if hash_table contains key (label, k) then
        return hash_table[(label, k)];
    end
    // answer the exclusion question
    ADDNode left_child;
    if label == n then
        left_child = Terminal node with value k^2;
    else
        left_child = create_sq_ADD(n, label+1, k, hash_table);
    end
    // answer the inclusion question
    ADDNode right_child;
    if label == n then
        right_child = Terminal node with value (k+1)^2;
    else
        right_child = create_sq_ADD(n, label+1, k+1, hash_table);
    end
    // combine the exclusion and inclusion answers
    ADDNode curr = new ADDNode (decision variable = x_label);
    curr → left = left_child, curr → right = right_child;
    hash_table[(label, k)] = curr;
    return curr;
end
```

Algorithm 5: ADD creation for $G_n^{2/3}$: Run create_2_of_3_ADD(n)

```
function create_2_of_3_ADD(n)
    if n == 1 return the base case ADD of Fig. 2 (d);
    ADDNode prev = create_2_of_3_ADD(n-1);
    ADDNode X = new ADDNode (decision variable = x_n);
    ADDNode Y1 = new ADDNode (decision variable = y_n);
    ADDNode Y2 = new ADDNode (decision variable = y_n);
    ADDNode Z = new ADDNode (decision variable = z_n);
    ADDNode ZERO = Terminal node with value 0;
    X → left_child = Y1, X → right_child = Y2;
    Y1 → left_child = ZERO, Y1 → right_child = Z;
    Y2 → left_child = Z, Y2 → right_child = prev;
    Z → left_child = ZERO, Z → right_child = prev;
    return X;
end
```

lengthy and tedious; so we omit the proof for the $G_{m,n}^{gl}$ ADD size quoted in Table I.

Similarly, Algorithm 2 and Algorithm 5 produce ADD representations for the 1-of-2 and 2-of-3 games respectively. The recursive relations used here are much simpler; they are illustrated (along with the base case $n = 1$) in Fig. 2 (c) and Fig. 2 (d) respectively.

IV. Our representation: the algorithms

This section presents polynomial time ADD-based algorithms (along with readily implementable pseudocode) for solving the six key game-theoretic problems mentioned in §I: `TEST-CORE`, `EMPTY-CORE`, ϵ-`CORE`, `CoS`, `BI` and `SV`.

A. *TEST-CORE, EMPTY-CORE, ϵ-CORE and CoS*

Given a coalitional game $g = \langle N, \nu : 2^N \rightarrow \mathbb{R} \rangle$, with $\nu(\emptyset) = 0$. Consider a *payoff vector* $\vec{x}$ (which is a $|N|$ dimensional vector whose entries add up to $\nu(N)$) that maps every agent $a \in N$ to a payoff $\vec{x}[a]$. We say that a coalition $C \subseteq N$ is *happy* with the payoff vector $\vec{x}$ provided the sum of the payoffs of all agents belonging to C is at least $\nu(C)$, i.e., the vector $\vec{x}$ collectively assigns to coalition C a payoff that is at least as large as the intrinsic value $\nu(C)$. The *core* [4] of a coalitional game is the set of all payoff vectors $\vec{x}$ such that every coalition $C \subseteq N$ is *happy* with $\vec{x}$. The intuitive explanation is that the core contains all *stable* payoff divisions, i.e., all possible ways of distributing the value of the grand coalition among its members so that no subset of agents has any incentive to "break off" from the grand coalition.

However, the problem is that for many coalitional games, the core is empty (i.e., there exists no stable way to distribute the value of the grand coalition among its members) [4]. For such games, a solution concept called the strong ϵ-*core* [5] has been proposed, which uses a *weaker stability criterion*: for every coalition $C \subseteq N$, the collective payoff assigned to C should be at least $\nu(C) - \epsilon$. The intuition behind the strong ϵ-core is that no coalition $C \subseteq N$ would gain more than ϵ by breaking off from the grand coalition. In other words, if a penalty of ϵ is imposed for leaving the grand coalition, then there is no incentive for any subset of agents to break off. The challenge is to find the smallest ϵ such that this weaker condition can be satisfied by at least one payoff vector $\vec{x}$.

More recently, a new solution concept [6] based on *the cost of stabilising the grand coalition* has been proposed for games whose core is empty: here it is assumed that a *benevolent external party* would like to stabilise the grand coalition by awarding it a value Δ over and above its intrinsic value $\nu(N)$. The problem is to find the smallest Δ such that the amount $\nu(N) + \Delta$ can be distributed among the agents, leaving no coalition $C \subseteq N$ with an incentive to break off. This smallest Δ is called the *cost of stability (CoS)*.

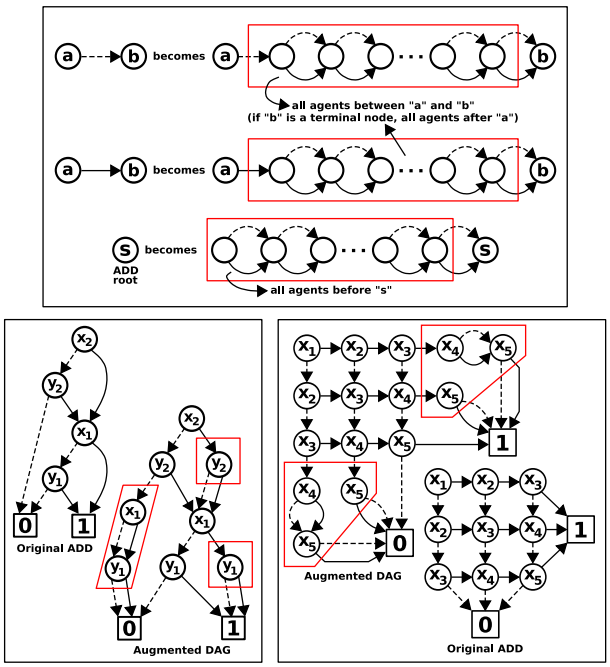

Fig. 3. Converting an ADD into an augmented DAG. Top: Rules to introduce intermediate nodes on ADD edges and before the ADD root. Bottom: Example conversions of the $G_2^{1/2}$ ADD (left) and the G_2^M ADD (right) to augmented DAGs (the newly introduced nodes are indicated by boxes around them).

Algorithms 6–8 describe polynomial-time ADD-based procedures for solving TEST-CORE, EMPTY-CORE, ϵ-CORE and CoS. These algorithms work by generating an *augmented DAG* from the given ADD. In the augmented DAG, *all agents occur on all paths, and in the correct order*. That is, if the game is played by agents $x_1 < x_2 < \ldots < x_n$, then *every complete path* (starting at the source node and ending at one of the terminal nodes) in the augmented DAG would correspond to the label sequence $x_1, x_2 \ldots x_n$, followed by a terminal node. Fig. 3 shows how to convert an ADD into an augmented DAG, by inserting intermediate nodes (if necessary) on each ADD edge and before the ADD root. Note that the concepts such as terminal/non-terminal nodes, dashed/solid edges, left/right children etc. apply equally well to augmented DAGs. Also note that the size of the augmented DAG is polynomial in the size of the original ADD.

Algorithm 6 tests if a given payoff vector $\vec{x}$ lies in the core of a coalitional game represented as an ADD. The key observation that makes the algorithm polynomial is that: it is not necessary to enumerate all coalitions C and check that they are happy with $\vec{x}$; rather, it is sufficient to check that at every terminal node u of the augmented DAG, the coalitions with value $L_{V'}(u)$ that receive the *least payoff* under $\vec{x}$ are happy. The *least payoff* at each terminal node is found using the critical path method for DAGs [13, Sec. 24.2]. Thus Algorithm 6 runs in time linear in the size of the augmented DAG.

Algorithm 6: TEST-CORE in polynomial time

Input: Coalitional game $\Gamma = \langle N, <, G(V, E, L_V, L_E)\rangle$, with $\nu(\emptyset) = 0$.
Payoff vector $\vec{x}_{test}$ of dimension $|N|$, whose entries add up to $\nu(N)$.
Output: TRUE if $\vec{x}_{test}$ is in the core of Γ. FALSE otherwise.
$G'(V', E', L_{V'}, L_{E'})$ = augmented DAG created from G;
// all further operations only on G'
foreach node $u \in V'$, **initialize** $d[u] = +\infty$;
initialize d[source node of G'] $= 0$;
vector⟨DAGNode⟩ topological_order = topological_sort(G');
foreach *DAGNode* u *in topological_order* **do**
  if u *is a non-terminal node* **then**
    d[left_child(u)] = min(d[left_child(u)], $d[u]$);
    d[right_child(u)] = min(d[right_child(u)], $d[u] + \vec{x}_{test}[L_{V'}(u)]$);
  else
    if $d[u] < L_{V'}(u)$ **then return** FALSE;
  end
end
return TRUE;

Algorithm 7 outlines a polynomial time procedure for EMPTY-CORE and CoS. The main idea is that: for each node u in the augmented DAG, a variable $d[u]$ maintains a lower bound on the *least payoff at* u, which is the shortest path length from the DAG source to u, where each SOLID edge (v, w) is assigned a weight $\vec{x}[v]$ and each DASHED edge is assigned weight zero. This lower bound is enforced by a set of *linear constraints*. Thus, EMPTY-CORE and CoS are both reduced to linear programming (LP) instances, for which well-known polynomial time techniques (e.g., Karmarkar's algorithm [14]) exist.

Algorithm 7: EMPTY-CORE and CoS in polynomial time

Input: Coalitional game $\Gamma = \langle N, <, G(V, E, L_V, L_E)\rangle$, with $\nu(\emptyset) = 0$.
Output: CoS of Γ and a payoff vector $\vec{x}_{\text{CoS}}$. The core is non-empty iff the returned CoS is 0 (if so, the returned $\vec{x}_{\text{CoS}}$ lies in the core; if not, the core is empty and the returned $\vec{x}_{\text{CoS}}$ stabilises the grand coalition while achieving CoS).
$G'(V', E', L_{V'}, L_{E'})$ = augmented DAG created from G;
// all further operations only on G'
initialize LPconstraints = $\{d$[source node of $G'] = 0\}$;
foreach *DAGEdge* $e = (u, v)$ *in* E' **do**
  if $L_{E'}(e) ==$ *DASHED* **then**
    LPconstraints.add($d[v] \le d[u]$)
  else
    LPconstraints.add($d[v] \le d[u] + \vec{x}(L_{V'}(u))$)
  end
end
foreach terminal node $u \in V'$ **do** LPconstraints.add($d[u] \ge L_{V'}(u) - \epsilon$);
LPconstraints.add($\sum_{a \in N} \vec{x}[a] = \nu(N)$)
$\langle opt, [\vec{d}_{\text{CoS}}, \vec{x}_{\text{CoS}}]\rangle$ = LPsolve(**min** $\sum_{a \in N} \vec{x}[a]$ **subj. to** LPconstraints);
CoS = $opt - \nu(N)$; **return** [CoS, $\vec{x}_{\text{CoS}}$];

Algorithm 8: ϵ-CORE in polynomial time

Input: Coalitional game $\Gamma = \langle N, <, G(V, E, L_V, L_E)\rangle$, with $\nu(\emptyset) = 0$.
Output: The smallest ϵ such that Γ has a non-empty strong-ϵ core (and a corresponding payoff vector $\vec{x}_\epsilon$)
$G'(V', E', L_{V'}, L_{E'})$ = augmented DAG created from G;
// all further operations only on G'
initialize LPconstraints = $\{d$[source node of $G] = 0\}$;
foreach *DAGEdge* $e = (u, v)$ *in* E' **do**
  if $L_{E'}(e) ==$ *DASHED* **then**
    LPconstraints.add($d[v] \le d[u]$)
  else
    LPconstraints.add($d[v] \le d[u] + \vec{x}(L_{V'}(u))$)
  end
end
foreach terminal node $u \in V'$ **do** LPconstraints.add($d[u] \ge L_{V'}(u) - \epsilon$);
LPconstraints.add($\sum_{a \in N} \vec{x}[a] = \nu(N)$)
$\langle \epsilon_{opt}, [\vec{d}_\epsilon, \vec{x}_\epsilon, \epsilon_{opt}]\rangle$ = LPsolve(**min** ϵ **subj. to** LPconstraints);
return $[\epsilon_{opt}, \vec{x}_\epsilon]$;

Algorithm 8 solves ϵ-CORE in polynomial time, using techniques very similar to Algorithm 7. The only difference is that a new variable ϵ is introduced into the LP constraints at the terminal DAG nodes. Minimising ϵ (subject to the LP constraints) is again an instance of LP (hence solved in polynomial time).

B. BI and SV

Given a coalitional game $g = \langle N, \nu : 2^N \to \mathbb{R}\rangle$, where $\nu(\emptyset) = 0$. The Banzhaf Index [7] $BI_g(x)$ of agent x in this game is defined by:

$$BI_g(x) = \frac{1}{2^{|N|-1}} \sum_{S \subseteq N \setminus \{x\}} [\nu(S \cup \{x\}) - \nu(S)]$$

The intuition is that $BI_g(x)$ is the *expected marginal contribution* made by the agent x to a *(uniformly) randomly chosen subset of* $N \setminus \{x\}$.

The Shapley Value [8] $SV_g(x)$ of agent x in the game g is defined by:

$$SV_g(x) = \frac{1}{|N|} \sum_{S \subseteq N \setminus \{x\}} \frac{1}{\binom{|N|-1}{|S|}} [\nu(S \cup \{x\}) - \nu(S)]$$

The intuition is that $SV_g(x)$ is the *expected marginal contribution* made by the agent x to the *subset of agents that occurs before* x *in a (uniformly) randomly chosen permutation of all the agents* N.

Algorithm 9 describes a polynomial time procedure for computing BI and SV, given a coalitional game represented as an ADD. The algorithm works by dynamic programming. The agents playing the game are denoted $x_1 < x_2 < \ldots < x_n$. The agent whose BI/SV is to be computed is denoted x_i.

To find BI(x_i), a dynamic programming sub-problem $\alpha_i^{out}(u)$ is defined at each node u of the given ADD. This sub-problem asks for the *number of subsets of* $\{x_1, x_2 \ldots L_V(u)\}$ *(where* $L_V(u)$ *is replaced by* x_n *if* u *is a terminal node), not containing* x_i*, under whose truth assignment there exists a path from the source node to* u. Another sub-problem, $\alpha_i^{in}(u)$, counts the subsets *containing* x_i that have a source-to-u path. Fig. 4 provides detailed equations for solving these sub-problems at the child nodes, using the solutions memoised at the parent nodes. Finally, the dynamic programming solutions at the terminal nodes are put together to compute BI(x_i) (the equations for this are also supplied by Fig. 4).

For computing SV(x_i), a dynamic programming sub-problem $\beta_i^{out}(u, m)$ is defined at each ADD node u, for every $0 \le m \le |N|$. This sub-problem asks for the *number of* m*-sized subsets of* $\{x_1, x_2 \ldots L_V(u)\}$ *(where* $L_V(u)$ *is replaced by* x_n *if* u *is a terminal node), not containing* x_i*, under whose truth assignment there exists a path from the source node to* u. Similarly, the sub-problem $\beta_i^{in}(u, m)$ counts the m-sized subsets *containing* x_i that have a source-to-u path. As before, Fig. 4 provides the equations for computing sub-problem solutions at child nodes using solutions memoised at parent nodes.

Complexity: For BI, (a) each ADD node is visited exactly once (in topological order), (b) at each non-terminal ADD node, it takes time $O(|N|)$ for updating the dynamic programming solutions at the child

Notation
$\circ\ N = \{x_1, x_2\ \ldots\ x_n\}$
$\circ\ x_1 \le x_2 \le\ \ldots\ \le x_n$

Initialization of the source node		
	Banzhaf Index	Shapley Value
Source node u	$u(x_j)$	$u(x_j)$
$j \le i$	$\alpha_i^{in}(u) = 0$ $\alpha_i^{out}(u) = 2^{j-1}$	$\beta_i^{in}(u,m) = 0$ $\beta_i^{out}(u,m) = \binom{j-1}{m}$
$j > i$	$\alpha_i^{in}(u) = 2^{j-2}$ $\alpha_i^{out}(u) = 2^{j-2}$	$\beta_i^{in}(u,m) = \binom{j-2}{m-1}$ $\beta_i^{out}(u,m) = \binom{j-2}{m}$

Algorithm 9: BI and SV in polynomial time

Input: Coalitional game $\Gamma = \langle N, <, G(V, E, L_V, L_E)\rangle$, Agent $x_i \in N$
Output: BI and SV of agent x_i in Γ

foreach node $u \in V$, **initialize** $\{\alpha_i^{in}(u), \alpha_i^{out}(u), \beta_i^{in}(u,m), \beta_i^{out}(u,m)\}$ all to zero;
initialize $\{\alpha_i^{in}(u), \alpha_i^{out}(u), \beta_i^{in}(u,m), \beta_i^{out}(u,m)\}$ for the source node *(use table on left)*;
initialize $\mathtt{BI}(x_i) = 0$, $\mathtt{SV}(x_i) = 0$;
vector⟨ADDNode⟩ topological_order = topological_sort(G);
foreach *node u in topological_order* **do**
  if *u is a non-terminal node* **then**
    use the dynamic programming table below to:
    update $\{\alpha_i^{in}(v), \alpha_i^{out}(v), \beta_i^{in}(v,m), \beta_i^{out}(v,m)\}$, where $v = \text{left\_child}(u)$
    update $\{\alpha_i^{in}(v), \alpha_i^{out}(v), \beta_i^{in}(v,m), \beta_i^{out}(v,m)\}$, where $v = \text{right\_child}(u)$
  else
    update $\mathtt{BI}(x_i)$ and $\mathtt{SV}(x_i)$ *using the final solution table below*
  end
end
return $\mathtt{BI}(x_i), \mathtt{SV}(x_i)$

Dynamic Programming: Computing the solutions to sub-problems at child nodes v using previously computed (and memoized) solutions at the parent nodes u

	Banzhaf Index		Shapley Value	
Non-terminal nodes u	$u(x_j) \dashrightarrow (x_k) v$	$u(x_j) \rightarrow (x_k) v$	$u(x_j) \dashrightarrow (x_k) v$	$u(x_j) \rightarrow (x_k) v$
$k \le i$	$\alpha_i^{in}(v) \mathrel{+}= 0$ $\alpha_i^{out}(v) \mathrel{+}= \alpha_i^{out}(u) 2^{k-j-1}$	$\alpha_i^{in}(v) \mathrel{+}= 0$ $\alpha_i^{out}(v) \mathrel{+}= \alpha_i^{out}(u) 2^{k-j-1}$	$\beta_i^{in}(v,m) \mathrel{+}= 0$ $\beta_i^{out}(v,m) \mathrel{+}= \sum_{l=0}^{m} \beta_i^{out}(u,l) \binom{k-j-1}{m-l}$	$\beta_i^{in}(v,m) \mathrel{+}= 0$ $\beta_i^{out}(v,m) \mathrel{+}= \sum_{l=0}^{m} \beta_i^{out}(u,l) \binom{k-j-1}{m-l-1}$
$k > i$ $j < i$	$\alpha_i^{in}(v) \mathrel{+}= \alpha_i^{out}(u) 2^{k-j-2}$ $\alpha_i^{out}(v) \mathrel{+}= \alpha_i^{out}(u) 2^{k-j-2}$	$\alpha_i^{in}(v) \mathrel{+}= \alpha_i^{out}(u) 2^{k-j-2}$ $\alpha_i^{out}(v) \mathrel{+}= \alpha_i^{out}(u) 2^{k-j-2}$	$\beta_i^{in}(v,m) \mathrel{+}= \sum_{l=0}^{m} \beta_i^{out}(u,l) \binom{k-j-2}{m-l-1}$ $\beta_i^{out}(v,m) \mathrel{+}= \sum_{l=0}^{m} \beta_i^{out}(u,l) \binom{k-j-2}{m-l}$	$\beta_i^{in}(v,m) \mathrel{+}= \sum_{l=0}^{m} \beta_i^{out}(u,l) \binom{k-j-2}{m-l-2}$ $\beta_i^{out}(v,m) \mathrel{+}= \sum_{l=0}^{m} \beta_i^{out}(u,l) \binom{k-j-2}{m-l-1}$
$k > i$ $j = i$	$\alpha_i^{in}(v) \mathrel{+}= 0$ $\alpha_i^{out}(v) \mathrel{+}= \alpha_i^{out}(u) 2^{k-j-1}$	$\alpha_i^{in}(v) \mathrel{+}= \alpha_i^{out}(u) 2^{k-j-1}$ $\alpha_i^{out}(v) \mathrel{+}= 0$	$\beta_i^{in}(v,m) \mathrel{+}= 0$ $\beta_i^{out}(v,m) \mathrel{+}= \sum_{l=0}^{m} \beta_i^{out}(u,l) \binom{k-j-1}{m-l}$	$\beta_i^{in}(v,m) \mathrel{+}= \sum_{l=0}^{m} \beta_i^{out}(u,l) \binom{k-j-1}{m-l-1}$ $\beta_i^{out}(v,m) \mathrel{+}= 0$
$k > i$ $j > i$	$\alpha_i^{in}(v) \mathrel{+}= \alpha_i^{in}(u) 2^{k-j-1}$ $\alpha_i^{out}(v) \mathrel{+}= \alpha_i^{out}(u) 2^{k-j-1}$	$\alpha_i^{in}(v) \mathrel{+}= \alpha_i^{in}(u) 2^{k-j-1}$ $\alpha_i^{out}(v) \mathrel{+}= \alpha_i^{out}(u) 2^{k-j-1}$	$\beta_i^{in}(v,m) \mathrel{+}= \sum_{l=0}^{m} \beta_i^{in}(u,l) \binom{k-j-1}{m-l}$ $\beta_i^{out}(v,m) \mathrel{+}= \sum_{l=0}^{m} \beta_i^{out}(u,l) \binom{k-j-1}{m-l}$	$\beta_i^{in}(v,m) \mathrel{+}= \sum_{l=0}^{m} \beta_i^{in}(u,l) \binom{k-j-1}{m-l-1}$ $\beta_i^{out}(v,m) \mathrel{+}= \sum_{l=0}^{m} \beta_i^{out}(u,l) \binom{k-j-1}{m-l-1}$

Final solution: Using the answers to dynamic programming sub-problems at terminal nodes u to compute $\mathtt{BI}(x_i)$ and $\mathtt{SV}(x_i)$

Banzhaf Index	Shapley Value
$\mathtt{BI}(x_i) \mathrel{+}= \frac{L_V(u)}{2^{n-1}} \left[\alpha_i^{in}(u) - \alpha_i^{out}(u)\right]$	$\mathtt{SV}(x_i) \mathrel{+}= \frac{1}{n} \sum_{m=0}^{n-1} \frac{L_V(u)}{\binom{n-1}{m}} \left[\beta_i^{in}(u,m+1) - \beta_i^{out}(u,m)\right]$

Fig. 4. Computation of answers to dynamic programming sub-problems from previously memoized solutions. Note that if v is a terminal node, k is assumed to be $n+1$. Also, $\binom{n}{k}$ is assumed to be 0 whenever $k > n$ or $k < 0$, with $\binom{0}{0} = 1$.

nodes (since this involves at most four N-bit multiplications), and (c) all other operations, such as initialization and obtaining the final solution, are insignificant compared to the update operation in terms of big-O complexity. Hence the complexity of BI is $O(|N||ADD|)$ (which is polynomial in both the number of agents and the size of the input ADD).

For SV, (a) each ADD node is visited exactly once (in topological order), (b) at each non-terminal ADD node, it takes time $O(|N|^2 \log |N|)$ for updating the dynamic programming solutions at the child nodes (although at first sight the complexity appears to be $O(|N|^3)$, we point out that all the SV update equations can be viewed as convolutions of two $|N|$-length sequences of $|N|$-bit numbers, which can be carried out in time $O(|N|^2 \log |N|)$ using Fast Fourier Transform (FFT) based techniques [15]), and (c) all other operations, such as initialization and putting together the final solution, are insignificant compared to the update operation in terms of big-O complexity. Hence the complexity of SV is $O(|N|^2 \log |N||ADD|)$.

V. Additional Observations

Here we make two observations that enable efficient analysis of coalitional games using ADDs.

Observation 1. The ADD-List. It is well-known that all read-once boolean formulas can be represented efficiently using ADDs. Specifically, for every read-once boolean formula on n variables, an ADD of size $O(n)$ can be constructed for it in linear time [16]. Thus, given any basic/read-once MC-Net, it is possible to very efficiently convert it into a *list of ADDs*, with one ADD per MC-Net rule. Each ADD in the list can then be solved individually for BI or SV (using Algorithm 9); then these values can be summed up to yield the desired agent's overall BI or SV (making use of the additivity property).

The above method, in fact, suggests a general strategy for analysing

unrestricted MC-Nets: (a) convert each MC-Net rule into an ADD, (b) solve each ADD individually for BI and SV (using Algorithm 9), and (c) use the additivity property for computing the overall BI and SV efficiently. Thus, ADDs are a powerful way to analyse unrestricted MC-Nets for BI and SV. As long as each MC-Net rule can be translated into a compact ADD, the method will be efficient; there is absolutely no requirement that the MC-Net rules should be basic/read-once. Therefore, ADDs enable a much larger class of MC-Net *Patterns* than was ever possible before.

Observation 2. Finding "high-yield" coalitions. Consider a situation where *agents need to be paid to join coalitions*. This is common in many real-life situations: e.g., companies need to pay employees in order for them to work together. One way to model this is a coalitional game $\langle N, \nu \rangle$, along with a *payment vector* $\vec{p}$ that specifies the payment required by each agent to participate. Given an total budget B, the natural question to ask is: how to maximise $\nu(C)$ such that all agents in C can be paid off within the budget? Although this problem is NP-Hard in general, it can be solved efficiently if g is represented as an ADD. Indeed, the algorithm is very similar to Algorithm 6 for TEST-CORE: at each terminal ADD node u, use the critical path method for DAGs to find the coalition with value $L_V(u)$, *that requires least payment*; of these, pick the coalition that yields the maximum value, while demanding a payment at most B.

VI. A New Theoretical Result

We now present a previously unknown, positive result showing that a wide range of coalitional games can in fact be solved efficiently (with respect to TEST-CORE, EMPTY-CORE, ϵ-CORE, CoS, BI and SV) using our ADD-based representation. To prove this, we draw upon the EE literature on ADD construction for symmetric boolean functions.

Definition. **k-Typed Coalitional Game (k-TCG).** A coalitional game $g = \langle N, \nu \rangle$ is said to be *k-typed* if the set N can be partitioned into k disjoint subsets $N_1, N_2 \ \ldots \ N_k$ whose union is N, such that the value $\nu(C)$ of *every* coalition C can be expressed as a function $f(n_1, n_2 \ldots n_k)$, where n_i denotes the number of agents of C belonging to N_i, for every $1 \leq i \leq k$.

The intuition is that, in a k-TCG, one can group the agents into k types, and the value of any coalition C would depend only on the number of agents of each type, who are in C.

Example 6. Consider a weighted voting game $W_n(w_1, w_2, w_3, q)$ played among $3n$ agents, n of whom have weight w_1, n of whom have weight w_2 and n of whom have weight w_3, where the winning quota is q. Then $W_n(w_1, w_2, w_3, q)$ is a 3-TCG because the value of every coalition C can be determined, knowing only the number of agents of each weight $\{w_1, w_2, w_3\}$ in C.

Example 7. The glove games $G^{gl}_{m,n}$ are 2-TCGs because the value of any coalition C can be determined, knowing only the number of left agents and the number of right agents in C.

Theorem. Let $g = \langle N, \nu \rangle$ be a k-TCG with n agents. Then g has an ADD representation containing at most $(1+n)^k(1+kn/2)$ nodes.

Proof sketch: The result follows by generalising a theorem outlined in [10], which states that all 1-TCGs played by n agents can be represented by ADDs containing at most $O(n^2)$ nodes. It is quite straightforward to generalise this construction to k-TCGs using the variable ordering: $\{\text{agents} \in N_1\} < \{\text{agents} \in N_2\} < \ldots < \{\text{agents} \in N_k\}$, where $N_1, N_2 \ldots N_k$ make up the k-partition of N (described in the definition above). Within a set N_i, the ordering of agents is immaterial. The ADD so constructed can be shown to contain at most $(1+n)^k(1+kn/2)$ nodes. Due to space constraints, we omit a detailed proof.

The above result shows that whenever k is bounded, k-TCGs can be represented compactly using ADDs. For example, all weighted voting games with at most k different weights (where k is bounded) can be compactly represented (and hence, efficiently solved for TEST-CORE, EMPTY-CORE, ϵ-CORE, CoS, BI and SV) using ADDs. Similarly, all coalitional skill/resource games where the agents can be classified into at most k *skill/resource profiles* can be represented and solved efficiently using ADDs. Thus, the above theorem at once proves that many games of practical interest, belonging to widely different categories of coalitional games, can all be compactly represented and efficiently solved using ADDs.

VII. Conclusions

Problem	Induced subgraph	Unrestricted MC-Net	Basic MC-Net	Read-once MC-Net	ADD
$\nu(C)$ given C	✓	✓	✓	✓	✓
TEST-CORE	×	×	×	×	✓
EMPTY-CORE	×	×	×	×	✓
ϵ-CORE	×	×	×	×	✓
CoS	×	×	×	×	✓
BI	✓	×	✓	✓	✓
SV	✓	×	✓	✓	✓

TABLE II. Comparing different representation schemes with respect to efficiency of solution concept computation. ✓ means the problem is in P and × means the problem is NP-Hard or worse.

In this paper, we have presented a new method, based on Algebraic Decision Diagrams, for representing coalitional games. We have demonstrated that ADDs are not only compact for many games of practical interest, but also computationally efficient for many solution concepts. Table II compares the efficiency of solution concept computation in our representation versus existing state-of-the-art techniques. As the table shows, no existing representation scheme offers advantages comparable to ADDs.

We have also presented the ADD-List, a new data structure that enables efficient BI and SV computation for unrestricted MC-Nets. With ADD-Lists, a much larger class of MC-Net *Patterns* can be handled, than was ever possible before. In short, ADDs offer all the advantages of state-of-the-art representations, and then some more!

We have also shown that ADDs can be applied to solve a new and interesting problem in coalitional game analysis: finding high-yield coalitions under a budget constraint.

Most importantly, in this paper, we have forged the first link between coalitional game theory and Algebraic Decision Diagrams. As a result, we have made it possible to borrow ideas from the huge EE literature on ADDs, and apply them to advance the field of algorithmic coalitional game theory. We have already demonstrated one such application: the k-typed coalitional games. We feel sure that more applications will be discovered in the future; thus the full impact of ADDs on coalitional game theory remains to be seen.

References

[1] X. Deng and C. Papadimitriou. On the complexity of cooperative solution concepts. *Mathematics of Operations Research*, 19(2):257–266, 1994.

[2] S. Ieong and Y. Shoham. Marginal contribution nets: A compact representation scheme for coalitional games. In *EC '05: Proceedings of the Sixth ACM Conference on Electronic Commerce*, pages 193–202, 2005.

[3] E. Elkind, L.A. Goldberg, P.W. Goldberg, and M. Wooldridge. A tractable and expressive class of marginal contribution nets and its applications. *Mathematical Logic Quarterly*, 55(4):362–376, 2009.

[4] A. Rapoport. *N-person game theory: Concepts and applications*. Dover Publications, 2001.

[5] L.S. Shapley and M. Shubik. Quasi-cores in a monetary economy with nonconvex preferences. *Econometrica*, 34(4):805–827, 1966.

[6] Y. Bachrach, E. Elkind, R. Meir, D. Pasechnik, M. Zuckerman, J. Rothe, and J. Rosenschein. The cost of stability in coalitional games. In *Algorithmic Game Theory*, volume 5814 of *Lecture Notes in Computer Science*, pages 122–134. Springer, Berlin, 2009.

[7] J.F. Banzhaf. Weighted voting does not work: A mathematical analysis. *Rutgers Law Review*, 19(2):317–343, 1965.

[8] L.S. Shapley. A value for n-person games. In *Classics in Game Theory*, pages 69–79. Princeton University Press, 1997.

[9] R.I. Bahar, E.A. Frohm, C.M. Gaona, G.D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic Decision Diagrams and their applications. *Formal Methods in System Design*, 10(2-3):171–206, 1997.

[10] D.E. Ross, K.M. Butler, and M.R. Mercer. Exact ordered Binary Decision Diagram size when representing classes of symmetric functions. *Journal of Electronic Testing*, 2(3):243–259, 1991.

[11] R.E. Bryant. Symbolic boolean manipulation with ordered Binary Decision Diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[12] C. Meinel and A. Slobodovà. On the complexity of constructing optimal ordered Binary Decision Diagrams. In *Mathematical Foundations of Computer Science*, volume 841 of *Lecture Notes in Computer Science*, pages 515–524. Springer, Berlin, 1994.

[13] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2 edition, 2001.

[14] N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4(4):373–395, 1984.

[15] J.W. Cooley, P.A.W. Lewis, and P.D. Welch. The Fast Fourier Transform and its applications. *IEEE Transactions on Education*, 12(1):27–34, 1969.

[16] M. Sauerhoff, I. Wegener, and R. Werchner. Optimal ordered Binary Decision Diagrams for read-once formulas. *Discrete Applied Mathematics*, 103(1-3):237–258, 2000.