Autonomous Agents and Multi-Agent Systems (2020) 34:29
https://doi.org/10.1007/s10458-020-09450-1

# An anytime algorithm for optimal simultaneous coalition structure generation and assignment

**Fredrik Präntare**[1] · **Fredrik Heintz**[1]

Published online: 3 March 2020


**Abstract**
An important research problem in artificial intelligence is how to organize multiple agents, and coordinate them, so that they can work together to solve problems. Coordinating agents in a multi-agent system can significantly affect the system's performance—the agents can, in many instances, be organized so that they can solve tasks more efficiently, and consequently benefit collectively and individually. Central to this endeavor is coalition formation—the process by which heterogeneous agents organize and form disjoint groups (coalitions). Coalition formation often involves finding a coalition structure (an exhaustive set of disjoint coalitions) that maximizes the system's potential performance (e.g., social welfare) through coalition structure generation. However, coalition structure generation typically has no notion of goals. In cooperative settings, where coordination of multiple coalitions is important, this may generate suboptimal teams for achieving and accomplishing the tasks and goals at hand. With this in mind, we consider simultaneously generating coalitions of agents and assigning the coalitions to independent alternatives (e.g., tasks/goals), and present an anytime algorithm for the simultaneous coalition structure generation and assignment problem. This combinatorial optimization problem has many real-world applications, including forming goal-oriented teams. To evaluate the presented algorithm's performance, we present five methods for synthetic problem set generation, and benchmark the algorithm against the industry-grade solver CPLEX using randomized data sets of varying distribution and complexity. To test its anytime-performance, we compare the quality of its interim solutions against those generated by a greedy algorithm and pure random search. Finally, we also apply the algorithm to solve the problem of assigning agents to regions in a major commercial strategy game, and show that it can be used in game-playing to coordinate smaller sets of agents in real-time.

**Keywords** Coalition structure generation · Assignment · Coordination · Coalition formation · Combinatorial optimization

✉ Fredrik Präntare
fredrik.prantare@liu.se

[1] Linköping University, 581 83 Linköping, Sweden

## 1 Introduction

A major research challenge in artificial intelligence is to solve the problem of how to organize and coordinate multiple artificial entities (e.g., agents) to improve their performance, behaviour, and/or capabilities. In multi-agent systems, this problem has been thoroughly studied, since the coordination of agents in a multi-agent system can significantly affect the system's performance—agents can, in many instances and settings, be organized so that they can cooperate and work together to solve tasks more efficiently [19].

There are many approaches to this, including *task allocation* [15], *assignment* algorithms [9,25,27,50], *task specification trees* [12,23], *multi-agent reinforcement learning* [38,42], and *coalition formation* [21,37]. The latter is a paradigm for coordination that has received extensive coverage in the literature over the past two decades [33,35], and typically involves both forming *coalitions* (flat goal-oriented organizations of agents) and allocating tasks, with potential applications in many disciplines, including economics [51], sensor fusion [10], waste-water treatment systems [11], wireless networks [18], strategy games [28], and small cell networks [54].

Deciding on which coalitions to form typically involves evaluating different *coalition structures* (sets of disjoint and exhaustive coalitions) and solving a *coalition structure generation* (CSG) problem. Subsequently, coalition formation proceeds by forming the coalitions in the (evaluated) coalition structure with the highest performance measure. The formed coalitions may then be used to perform tasks, or execute plans, that require several artificial entities to be accomplished efficiently.

From an algorithmic perspective, coalition structure generation and assignment are two major coordinative processes that are generally treated as separate paradigms (including in all previous examples). Even though coalitions are often described as goal-oriented organizational structures, conventional CSG algorithms (e.g., for characteristic function games and other similar games) have no explicit notion of goals. In instances for which coordination of multiple coalitions is important, using such algorithms may generate suboptimal teams for achieving and accomplishing the tasks and goals at hand. Also, if combined with a typical task allocation or assignment algorithm (e.g., the *Hungarian algorithm* that was introduced by Kuhn [22]), we would require two different functions for expressing a coalition's value: one for deciding on which coalitions to form, and one for assigning/allocating them to alternatives. This is potentially disadvantageous, since it is often complicated to create good utility/value functions (or to generate realistic performance measures), and it is not necessarily a simple task to predict how the two functions influence the quality of generated solutions. Also, there are many settings and scenarios in which the utility of a team not only depends on its members and the environment, but also on the task/goal it is assigned to. It would therefore be beneficial if algorithms for coalition structure generation could take advantage of goal-orientation.

In light of these observations, and to address the aforementioned issues, we introduce the *simultaneous coalition structure generation and assignment* (SCSGA) problem, in which goal-orientation is central to the generation of coalition structures. Furthermore, we present three different algorithms to solve it:

- an optimal anytime branch-and-bound algorithm (this paper's main contribution);
- a greedy non-optimal algorithm for benchmarking anytime solutions and generating initial lower bounds for the quality of problems' optimal solutions; and
- a pure random search algorithm to use as a baseline when benchmarking other anytime algorithms.

These algorithms integrate coalition-to-alternative assignment into the formation of coalitions by generating *ordered* coalition structures, for which each possible enumeration of coalitions correspond (bijectively) to a specific assignment of alternatives. Our algorithms can thus be used to create structured collaboration through explicit goal-orientation, and they only require one function (analogous to the characteristic function) for representing a coalition's potential performance/utility.

To evaluate our algorithms' performance, we present five different methods for generating synthetic problem sets (of which three are extended from previous methods for benchmarking CSG algorithms). We also benchmark our optimal anytime algorithm against *CPLEX*—a commercial state-of-the-art optimization software developed by IBM—to deduce whether it can handle difficult data sets with sufficient efficiency. Moreover, we also apply our algorithm to solve the problem of simultaneously forming and assigning groups of armies to regions in the commercial strategy game *Europa Universalis 4*, and empirically show that it can be used to optimally solve a difficult game-playing problem in real-time. Note that this is, to our knowledge, the first time an algorithm for coalition structure generation of this calibre has been used in a real-world application (a complex multi-agent system) to considerably improve autonomous agents' computational efficiency and decision-making.

Finally, apart from being applied to strategy games, SCSGA algorithms can potentially be used to solve many important real-world problems. They could, for example, be used to form optimal cross-functional/multi-disciplinary teams aimed at solving a set of problems; to assist in the organization and coordination of subsystems in an artificial entity (e.g., a robot); or to allocate tasks in multi-agent systems (e.g., multi-robot facilities). Since our branch-and-bound algorithm is anytime (i.e., it can return a valid solution even if it is interrupted prior to finishing a search), it can also be used in many real-world scenarios with real-time constraints as well, such as in time-critical systems for managing tactical decisions.

Note that this paper is a significantly extended and thoroughly revised version of two previous papers [28,29]. More specifically, in this paper, we provide a more thorough review of related algorithms and domains together with examples and descriptions of a few potential applications. Moreover, the presented algorithm, its presentation, and the benchmarks herein, have all been significantly improved. Additionally, we provide two new algorithms for SCSGA, and develop two new methods for generating synthetic problem sets that more closely model certain real-world scenarios (which we also use to benchmark our algorithms). Several additional theorems with proofs are also provided that strengthen the validity of our claims.

The structure of this paper is organized as follows. We begin by discussing related work, CSG algorithms and similar domains in Sect. 2. We then formalize the SCSGA problem in Sect. 3. In Sects. 4 and 5, we describe our algorithms in detail. In Sect. 6, we present our experiments. Finally, in Sect. 7, we conclude with a summary.

## 2 Related work and motivation

The most commonly studied CSG problem is in the context of *characteristic function games* (CFGs) [33], in which the value of a coalition only depends on its members. It is defined as follows:

**Input:** A set of agents $A = \{a_1, \ldots, a_n\}$, and the function $\boldsymbol{v}(C) \mapsto \mathbb{R}$, known as the *characteristic function*, that corresponds to the value (e.g., expected utility) of the coalition $C \subseteq A$. $\boldsymbol{v}(\emptyset) = 0$ is assumed.
**Output:** A *coalition structure* $CS$ over $A$ (see Definition 1) that maximizes the sum of its coalitions' values $\sum_{C \in CS} \boldsymbol{v}(C)$.

**Definition 1** *Coalition structure* A *coalition structure* $CS = \{C_1, \ldots, C_{|CS|}\}$ over the agents $A$ is a set of coalitions with $C_i \subseteq A \backslash \emptyset$ for $i = 1, \ldots, |CS|$, $C_i \cap C_j = \emptyset$ for all $i \neq j$, and $\bigcup_{i=1}^{|CS|} C_i = A$. For example, $\{\{a_1, a_3\}, \{a_2\}\}$ and $\{\{a_1\}, \{a_2\}, \{a_3\}\}$ are two different coalition structures over $A = \{a_1, a_2, a_3\}$.

This type of coalition structure generation problem has been thoroughly studied. It is NP-complete [36], and a multitude of algorithms have been presented to solve it using different approaches, such as dynamic programming [30,52], tree-search [34], and hybrid techniques [32]. Variations on the CSG problem also exist, e.g., with overlapping coalitions, where agents have limited resources that they can use to partake in multiple coalitions [8,17]. Also, even though it is typically computationally difficult to generate high-quality coalition structures (since the search space grows in $\omega(n^{n/2})$ and $\mathcal{O}(n^n)$ for $n$ agents), there are certain CSG problem instances that can be solved optimally in polynomial time, see e.g., [14]. There are also concise representations of the characteristic function that can be used to reduce a CSG problem's computational complexity by sacrificing expressiveness [45,47].

Note that it is possible to define the characteristic function so that a coalition's value corresponds to the value of an optimal solution to a *distributed constraint optimization* problem among the coalition's members [46]. This approach, albeit arguably more goal-oriented, still lacks an explicit notion of coalitional goals, since coalitions' purposes are not explicit during their generation, and because coalitions have the same performance measure regardless of their individual goals.

Furthermore, the CSG problem has been studied in the context of other games as well. For example, in *partition function games* (PFGs) (initially proposed by Lucas and Thrall [44]), a coalition's value not only depends on its members, but also on the way all other agents are partitioned. Hence, in PFGs, we are interested in *embedded coalitions* (Definition 2)—a notion with which the CSG problem for PFGs is defined as follows:

**Input:** A set of agents $A = \{a_1, \ldots, a_n\}$, and the function $\boldsymbol{w}(C, CS) \mapsto \mathbb{R}$, known as the *partition function*, that corresponds to the value (e.g., potential utility) of the embedded coalition $(C, CS)$ over $A$.
**Output:** A coalition structure $CS$ over $A$ that maximizes $\sum_{C \in CS} \boldsymbol{w}(C, CS)$.

**Definition 2** *Embedded coalition* An *embedded coalition* over the agents $A$ is a pair $(C, CS)$, where $CS$ is a coalition structure over $A$, and $C$ is a coalition with $C \in CS$.

Observe that CFGs are a special case of PFGs—in other words, CFGs form a proper subclass of PFGs. Furthermore, CSG in this setting is highly computationally challenging since the value of a coalition may depend on the partitioning of all other agents, thus taking what is known as *externalities* (i.e., the coalitions' exerted influence over each other) into consideration. This has the consequence that each coalition $C \subseteq A$ can have as many different values as there are ways to partition the remaining agents $A \backslash C$. Also, in general, you

cannot optimally solve a CSG problem for PFGs without enumerating all possible coalition structures. Clearly, this type of "brute-force" is not feasible for most reasonably realistic problems, since the number of partitions of a set with $n$ elements is equal to the $n$th *Bell number* $B_n$, for which the following holds:

$$\alpha n^{n/2} \leq B_n \leq n^n$$

for some positive real number $\alpha$ (see [36] for proof)—with the consequence that explicitly representing a partition function requires $\mathcal{O}(n^n)$ real numbers [33].

Now, to address these aforementioned issues, researchers have studied and developed algorithms for certain types of PFG representations and more limited classes of the partition function. For example, Rahwan et al. [31] and Epstein et al. [13] developed algorithms constrained to games with:

- *negative externalities*, in which merging any two coalitions is never beneficial to the other coalitions; and
- *positive externalities*, where merging two coalitions is never detrimental to other existing coalitions.

Furthermore, Skibski et al. [39] presented a graphical representation based on rooted directed trees called *partition decision trees*, which Zha et al. [53] then used to solve the CSG problem using (1) a depth-first branch-and-bound algorithm, and (2) a *maximum satisfiability* (MaxSAT) encoding together with an off-the-shelve solver.

However, there are many settings and scenarios in which the expected future utility of a team not only depends on its members (as in CFGs), or the way all agents are partitioned (as in PFGs), but also on the its collective goal (e.g., its purpose, or the task/job it is assigned to). We illustrate one such scenario in Example 1.

***Example 1*** Suppose we aim to coordinate staff (agents) at a hospital by forming several heterogeneous multi-disciplinary healthcare teams aimed at helping a number of patients in the best possible way. Since doctors and nurses may have many different specializations (e.g., radiology, neurosurgery, oncology), and patients typically have a wide range of different disorders and illnesses (e.g., cancer, infection, heart disease), the best teams typically depend on the patients that need to be treated, and they may require the participation of several specialist types. Thus, if we fail to take the patients into consideration, we may form teams that are suboptimal (or arbitrarily bad). Ideally, we would instead like to pair each patient with the group that maximizes the hospital's aggregated global utility.

Arguably, conventional CSG algorithms fail to model this multi-faceted interplay between teams (coalitions) and their goals/ambitions in a satisfying way. Although *games with alternatives* (initially introduced by Bolger [4], and further developed and studied in e.g., [1,2,5,6,26]) captures this interaction between coalitions and their goals, no CSG algorithms have been developed for them, and mainly voting situations have been considered. In this type of game, there is a set of players (agents) $A = \{a_1, \ldots, a_n\}$ with a set of alternatives $T = \{t_1, \ldots, t_m\}$, and each player must choose exactly one alternative. Furthermore, $C_i$ is defined to be the set of players who choose alternative $t_i$, and the vector $\langle C_1, \ldots, C_m \rangle$ is called an *arrangement* of the players $A$ among the alternatives $T$. If $S$ is such an arrangement, then, if $C \in S$, the function $\boldsymbol{w}(C, S) \mapsto \mathbb{R}$ corresponds to $C$'s worth, given that the other players choose alternatives as specified by $S$. Thus, in this context, we are also interested in embedded coalitions and externalities, since when valuing a coalition's worth/utility, the way that all other players choose alternatives is taken into consideration.

A formalism related to games with alternatives was developed and analyzed by Grabisch and Rusinowska [16]. In their work, they presented a multi-choice framework, in which each agent has to choose an action under the influence of others. To the best of our knowledge, algorithmically forming coalitions/teams using their framework has neither been studied nor analyzed.

Moreover, in the context of both CFGs and PFGs, it is possible to design the value function (i.e., characteristic or partition function) so that algorithms for optimal CSG can be used to generate certain types of goal-oriented coalition structures. This can be accomplished by first including additional entities ("special elements") that each represent a specific task/goal/alternative in the CSG input's agent set, and then defining the value function in a way so that unwanted coalitions never exist in optimal coalition structures. We exemplify this approach in Example 2 with a real-world scenario and application.

***Example 2*** Suppose we have several students $s_1, \ldots, s_n$ enrolled at a university. These students come from different curricula and backgrounds, and have a diverse set of distinct skills and preferences. This semester, they are taking a course, in which each of them will be assigned to one of a few different projects $p_1, \ldots, p_m$. The students assigned to a project have to work together to complete it. More to the point, the course's teachers aim to assign the students to the different projects, while still maximizing the students' knowledge exchange (by e.g., making sure that each group is diverse enough), and making sure that they are satisfied with the course (by e.g., making sure that the students are assigned to projects that they find interesting and relevant). The problem of forming such project groups can be modelled as a CSG problem in the CFG context. In more detail, let the CSG input's agent set be:

$$A = \{s_1, \ldots, s_n\} \cup \{p_1, \ldots, p_m\}$$

and define the characteristic function as follows:

$$\boldsymbol{v}(C) = \begin{cases} 0 & \text{if } C = \emptyset \\ v_C & \text{if } |C \cap \{p_1, \ldots, p_m\}| = 1 \\ -\infty & \text{otherwise} \end{cases}$$

where $C \subseteq A$, and $v_C$ is a real number that represents the utility (e.g., suitability) of the students in $C \cap \{s_1, \ldots, s_n\}$ being assigned to the project $p \in C \cap \{p_1, \ldots, p_m\}$. Of course, $v_C$ needs to be defined in a way so that the aforementioned intricacies and details are taken into consideration.

In a general sense, albeit perhaps theoretically valid, this way of handling goal-orientation is typically blunt and has many disadvantages. For example, in practice, it potentially leads to much worse computational performance for CSG algorithms than necessary, and for CSG in CFGs, it makes the search space grow in $\mathcal{O}((m+n)^{m+n})$ instead of $\mathcal{O}(n^n)$. This is not only costly, but also typically difficult to work with (in both practice and theory), and it is not always clear how to include non-finite values for valid coalitions. Also, for non-optimal CSG algorithms and algorithms with anytime characteristics, this approach may generate coalition structures that could arguably be regarded as infeasible due to containing coalitions with value $-\infty$. Generally speaking (although special cases may exist), when modelling a problem this way, we either have to accept the risk of generating suboptimal (or arbitrarily bad) coalition structures for goal-oriented domains, and/or sacrifice computational performance and brevity. In light of these observations, we now define the SCSGA problem—a type of CSG problem with which we can avoid these drawbacks.

## 3 Problem formalization

The simultaneous coalition structure generation and assignment problem is formalized as follows:

**Input:** A set of agents $A = \{a_1, \ldots, a_n\}$, a list of alternatives $T = \langle t_1, \ldots, t_m \rangle$ (e.g., tasks/goals), and the value function $\boldsymbol{v}(C, t) \mapsto \mathbb{R}$, called the *utility function*, that represents the potential value (e.g., performance measure) for assigning any coalition $C \subseteq A$ to any alternative $t \in T$.
**Output:** An *ordered coalition structure* (see Definition 3) $\langle C_1, \ldots, C_m \rangle$ over $A$ that maximizes the sum $\sum_{i=1}^{m} \boldsymbol{v}(C_i, t_i)$.

**Definition 3** *Ordered coalition structure* The list $S = \langle C_1, \ldots, C_{|S|} \rangle$ is an *ordered coalition structure* over a set of agents $A$ if $C_i \subseteq A$ for $i = 1, \ldots, |S|$, $C_i \cap C_j = \emptyset$ for all $i \neq j$, and $\bigcup_{i=1}^{|S|} C_i = A$. For example, $\langle \{a_1, a_3\}, \emptyset, \{a_2, a_4\} \rangle$ is an ordered coalition structure over the agents $\{a_1, a_2, a_3, a_4\}$.

A real-world situation that can be modelled using this formalization is illustrated in Example 3. Furthermore, note that both Examples 1 and 2 can be modelled as SCSGA problems in a straightforward manner.

***Example 3*** The *Médecins Sans Frontières* (commonly known as Doctors Without Borders) has over 40 000 field staff deployed in more than 70 countries [20]. Deciding on whom to deploy where is a difficult problem that can be modelled as a SCSGA problem. In more detail, let the staff be the input's agent set $A$; the deployment locations (e.g., countries) be the list of different tasks $T$; and the utility function $\boldsymbol{v}(C, t)$ yield a value that represents how beneficial it is to deploy a certain team $C \subseteq A$ (a subset of the personnel) at a specific deployment location $t \in T$. The output's ordered coalition structure corresponds to an optimal deployment of the staff.

Moreover, given the aforementioned input, we can also formalize the SCSGA problem using a *binary integer programming* model:

$$
\begin{aligned}
\text{maximize} \quad & \sum_{j=0}^{2^n-1} \sum_{k=1}^{m} x_{jk} \cdot \boldsymbol{v}(C^j, t_k) \\
\text{subject to} \quad & \sum_{j=0}^{2^n-1} \sum_{k=1}^{m} x_{jk} \cdot y_{ij} = 1 \qquad i = 1, \ldots, n \\
& \sum_{k=1}^{m} x_{jk} \leq 1 \qquad j = 1, \ldots, 2^n - 1 \\
& \sum_{j=0}^{2^n-1} x_{jk} = 1 \qquad k = 1, \ldots, m \\
& x_{jk} \in \{0, 1\}
\end{aligned}
$$

where $y_{ij} = 1$ if agent $a_i \in C^j$, $y_{ij} = 0$ if not, and $C^j$ is a coalition defined through its *binary coalition-encoding* given by $j$ over $A$ (see Definition 4). Note that $x_{jk} = 1$ if and

only if coalition $C^j$ is to be assigned to task $t_k$, and that $C^0 = \emptyset$ is the only coalition that can be assigned to multiple tasks. The first constraint ensures disjoint and exhaustive coalitions, while the second and third constraints ensures coalition-to-task bijections.

**Definition 4** *Binary coalition-encoding* Given a set of agents $A = \{a_1, \ldots, a_n\}$, and the non-negative integer $j < 2^n$ on binary form $j = b_1 2^0 + b_2 2^1 + \cdots + b_n 2^{(n-1)}$ with $b_i \in \{0, 1\}$ for all $i \in \mathbb{N}$, we say that the coalition $C^j \subseteq A$ has a binary coalition-encoding given by $j$ over $A$ if and only if $b_k = 1 \iff a_k \in C^j$ for $k = 1, \ldots, n$. For example, if the coalition $C^j$ has a binary coalition-encoding given by $j$ over $\{a_1, \ldots, a_n\}$, we have $C^0 = \emptyset$ for $j = 0$, $C^3 = \{a_1, a_2\}$ for $j = 3 = 11_2$, and $C^8 = \{a_4\}$ for $j = 8 = 1000_2$.

Observe that the SCSGA problem corresponds to a CSG problem for games with alternatives *without externalities* (i.e., coalitions' values are not affected by the way non-members are partitioned)—consequently, we are the first to develop and study the algorithmic process of generating coalition structures for this constrained class of games with alternatives. Also, note that we use the notion *task* to denote our analogy of an alternative throughout this paper, and that we use the terms *solution* and *ordered coalition structure* interchangeably. Moreover, the sum $\boldsymbol{V}(S) = \sum_{i=1}^{m} \boldsymbol{v}(C_i, t_i)$ is used to denote the value of a solution $S = \langle C_1, \ldots, C_m \rangle$. We also use the terms *agent* and *task* as abstractions (they can be substituted for any type of entities, e.g., resources, regions, intentions, goals), and we use the conventions $n = |A|$ and $m = |T|$ when it improves readability. Finally, there are $m^n$ possible solutions to a SCSGA problem instance, since there are $m$ possible tasks to assign each of the $n$ agents to. Consequently, albeit much improved over the aforementioned $\mathcal{O}((m+n)^{m+n})$, exhaustive search at $\mathcal{O}(m^n)$ is still costly and typically not feasible. Also, observe that there are no restrictions on the integer $m$ other than that it is positive—a SCSGA problem instance can thus have more tasks than there are agents.

## 4 Optimal anytime branch-and-bound algorithm

To solve this optimization problem, we propose an anytime branch-and-bound algorithm in conjunction with a search space representation based on multiset permutations of size-$m$ integer partitions. By using branch-and-bound, our algorithm always generates optimal solutions when run to exhaustion, and solutions with worst-case guarantees when interrupted prior to finishing a search. This algorithm, that we abbreviate *MP* (short for multiset permutation), consists of the following major steps:

I. Partitioning of the search space.
II. Calculation of the bounds for subspaces.
III. Searching for solutions using branch-and-bound.

These steps are described in the next three subsections.

### 4.1 Partitioning the search space

To partition the search space, we use a search space representation that is based on *multiset permutations* (ordered arrangements) of *integer partitions* (Definition 5). In this representation, a list of non-negative integers $\langle p_1, \ldots, p_m \rangle$ *represents* all solutions $\langle C_1, \ldots, C_m \rangle$ with $|C_i| = p_i$ for $i = 1, \ldots, m$ (see Definition 6). Note that this is technically a refinement of Rahwan, Ramchurn, Jennings and Giovannucci's search space representation for conventional coalition structure generation [34].

**Definition 5** *Integer partition* An integer partition of $y \in \mathbb{N}$ is a multiset of positive integers $\{x_1, \ldots, x_k\}$ such that:

$$\sum_{i=1}^{k} x_i = y.$$

For example, the multiset $\{1, 1, 2\}$ is an integer partition of 4 since $1 + 1 + 2 = 4$, and $\{1, 2, 12, 15\}$ is an integer partition of 30 since $1 + 2 + 12 + 15 = 30$.

**Definition 6** *MP-representation* A list of non-negative integers $\langle p_1, \ldots, p_m \rangle$ *represents* the ordered coalition structure $\langle C_1, \ldots, C_m \rangle$ if $p_i = |C_i|$ for $i = 1, \ldots, m$.

In more detail, we generate all multiset permutations of $m$-sized non-negative integer partitions of $n$. We use the following three steps to do so:

1. First, generate the set $M_1$ of all integer partitions of $n$ that has $m$ or fewer elements (addends). If $n = 4$ and $m = 3$, then $M_1 = \{\{4\}, \{3, 1\}, \{2, 2\}, \{2, 1, 1\}\}$. Algorithms that can be used to generate these integer partitions already exist, e.g., [3,41]. In our case, order is of no concern, and it is trivial to exclude integer partitions that have more than $m$ elements, so any algorithm can potentially be used.
2. Generate $M_2$ by appending zeros to the integer partitions in $M_1$ (that we generated during *step 1*) until all of them have $m$ elements. For example, if $n = 4$ and $m = 3$, then $M_2 = \{\{4, 0, 0\}, \{3, 1, 0\}, \{2, 2, 0\}, \{2, 1, 1\}\}$.
3. Now, let $M_3$ be the set of all multiset permutations of the multisets in $M_2$. For example, if $n = 4$ and $m = 3$, then $M_3 =$
   $\{ \langle 4, 0, 0\rangle, \langle 0, 4, 0\rangle, \langle 0, 0, 4\rangle, \langle 0, 2, 2\rangle, \langle 2, 0, 2\rangle, \langle 2, 2, 0\rangle,$
   $\langle 3, 1, 0\rangle, \langle 3, 0, 1\rangle, \langle 0, 3, 1\rangle, \langle 1, 3, 0\rangle, \langle 1, 0, 3\rangle, \langle 0, 1, 3\rangle,$
   $\langle 2, 1, 1\rangle, \langle 1, 2, 1\rangle, \langle 1, 1, 2\rangle \}$.
   Each multiset permutation $\langle p_1, \ldots, p_m \rangle \in M_3$ represents the subspace that contains all solutions $\langle C_1, \ldots, C_m \rangle$ with $|C_i| = p_i$ and $C_i \subseteq A$ for $i = 1, \ldots, m$. For instance, if $n = 4$ and $m = 3$, the multiset permutation $\langle 3, 1, 0\rangle$ then represents $\langle \{a_1, a_2, a_3\}, \{a_4\}, \emptyset\rangle$, $\langle \{a_1, a_2, a_4\}, \{a_3\}, \emptyset\rangle$, $\langle \{a_1, a_3, a_4\}, \{a_2\}, \emptyset\rangle$, and $\langle \{a_2, a_3, a_4\}, \{a_1\}, \emptyset\rangle$. Note that there exists several known algorithms that can generate these multiset permutations in $\mathcal{O}(1)$ per new permutation, e.g., [43,49].

Note that every coalition structure that consists of $k$ agents can be mapped to exactly one of the integer partitions of $k$ (see [34] for proof). For example, the coalition structure $\{\{a_1, a_2\}, \{a_3\}\}$ can be mapped to $\{2, 1\}$, and $\{\{a_1, a_2, a_3\}\}$ to $\{3\}$. In *step 1*, we generate the partitions that correspond to these mappings. We then remove unnecessary coalition structures in *step 2*, so that we only look at coalition structures that can represent valid solutions (namely $m$-sized coalition structures). Finally, in *step 3*, we refine the representation of the search space that was generated in *step 2*, by taking advantage of the fact that we are only interested in coalition-to-task bijections. Consequently, the solutions represented by the multiset permutations in $M_3$ cover the whole search space, as shown in Theorem 1. With this in mind, define:

- $I_n$ to be set of all integer partitions of $n$;
- $Z_n$ to be the set of all *zero-inclusive integer partitions* (see Definition 7) of $n$;
- $S_J$ to be the set of all multiset permutations of the multiset $J$.

We can now, more clearly and compactly, define $M_1$, $M_2$ and $M_3$ as follows:

- $M_1 := \{J \in I_n : |J| \leq m\}$;
- $M_2 := \{J \in Z_n : |J| = m\}$;
- $M_3 := \bigcup_{J \in M_2} S_J$.

**Definition 7** *Zero-inclusive integer partition* A zero-inclusive integer partition of $y \in \mathbb{N}$ is a multiset of non-negative integers $\{x_1, \ldots, x_k\}$ such that:

$$\sum_{i=1}^{k} x_i = y.$$

For example, the multiset $\{0, 0, 1, 3, 4, 10, 100\}$ is a zero-inclusive integer partition of 118 since $0 + 0 + 1 + 3 + 4 + 10 + 100 = 118$.

**Theorem 1** *The subspaces represented by the multiset permutations in $M_3$ cover the whole search space.*

***Proof*** By contradiction. Assume that a solution $\langle C_1, \ldots, C_m \rangle$ is not represented by any element in $M_3$. Formally, this means that there is no list of non-negative integers $\langle p_1, \ldots, p_m \rangle \in M_3$ with $|C_i| = p_i$ for $i = 1, \ldots, m$.

Now, let $Q = \langle |C_1|, \ldots, |C_m| \rangle$. Since $Q \notin M_3$ (by our assumption), and $|Q| = m$ (by definition), it must be the case that $\{|C_1|, \ldots, |C_m|\}$ is not a zero-inclusive integer partition of $n$ (otherwise, we have that $Q \in M_3$). In other words, $\sum_{i=1}^{m} |C_i| \neq n$. This is a contradiction, since $\sum_{i=1}^{m} |C_i| = n$ follows directly from Definition 3. □

Given any multiset permutation $P = \langle p_1, \ldots, p_m \rangle \in M_3$ generated through the aforementioned process, let $\mathbb{S}_P$ denote the set of all solutions $\langle C_1, \ldots, C_m \rangle$ with $|C_i| = p_i$ and $C_i \subseteq A$ for $i = 1, \ldots, m$. In other words, let $\mathbb{S}_P$ be the subspace that contains all solutions represented by the multiset permutation $P \in M_3$.

### 4.2 Calculating the bounds for subspaces

To establish bounds for the subspaces in our search space representation, so the algorithm can make more informed decisions during search, let $\mathbb{C}_p := \{X \subseteq A : |X| = p\}$, namely the set of all $p$-sized coalitions, and define:

- $\boldsymbol{Avg}(p, t) := \frac{1}{|\mathbb{C}_p|} \sum \{\boldsymbol{v}(C, t) : C \in \mathbb{C}_p\}$;
- $\boldsymbol{M}(p, t) := \max \{\boldsymbol{v}(C, t) : C \in \mathbb{C}_p\}$.

We can now establish a lower and an upper bound for the value of the best possible solution in $\mathbb{S}_P$ as the sums $l_P := \sum_{i=1}^{m} \boldsymbol{Avg}(p_i, t_i)$ and $u_P := \sum_{i=1}^{m} \boldsymbol{M}(p_i, t_i)$, respectively. For proofs, see Theorems 2 and 3. This lower bound, based on the average utility values of coalition-to-task assignments, is better than the more straightforward (and intuitive) $\sum_{i=1}^{m} \min \{\boldsymbol{v}(C, t_i) : C \in \mathbb{C}_{p_i}\}$. See Theorem 4 for proof.

**Theorem 2** *$l_P = \sum_{i=1}^{m} \boldsymbol{Avg}(p_i, t_i)$ is a lower bound for the value of the best possible solution in the subspace $\mathbb{S}_P$ where $P = \langle p_1, \ldots, p_m \rangle$. In other words:*

$$l_P \leq \max_{\langle C_1,\ldots,C_m\rangle \in \mathbb{S}_P} \left\{ \sum_{i=1}^{m} \boldsymbol{v}(C_i, t_i) \right\}.$$

***Proof*** Recall that, for the arithmetic mean $\overline{y_1, \ldots, y_k}$ of a finite set $\{y_1, \ldots, y_k\} \subset \mathbb{R}$, the following holds:

$$\overline{y_1, \ldots, y_k} \leq \max\{y_1, \ldots, y_k\}. \tag{1}$$

Now, since there are $|\mathbb{C}_p|$ coalitions of size $p \in P$, we have:

$$|\mathbb{S}_P| = x_i \cdot |\mathbb{C}_{p_i}| \tag{2}$$

for some integer $x_i \in \mathbb{N}$ for $i = 1, \ldots, m$. This is because there are $|\mathbb{C}_{p_i}|$ different coalitions that can be assigned to task $t_i$, and for each coalition assigned to $t_i$, we have $x_i$ ways of assigning coalitions to the other tasks $t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_m$. Following this argument, there are exactly $x_i$ solutions in $\mathbb{S}_P$ for which any coalition $C$ with $|C| = p_i$ is the $i^{th}$ coalition. Based on this and (3), we can calculate the arithmetic mean of $\mathbb{V}_P := \{\sum_{i=1}^{m} \boldsymbol{v}(C_i, t_i) : \langle C_1, \ldots, C_m\rangle \in \mathbb{S}_P\}$, namely the set of the values of the solutions in $\mathbb{S}_P$, as follows:

$$\begin{aligned}
\overline{\mathbb{V}_P} &= \frac{1}{|\mathbb{S}_P|} \sum_{i=1}^{m} \sum_{C \in \mathbb{C}_{p_i}} \left\{ x_i \cdot \boldsymbol{v}(C, t_i) \right\} \\
&= \sum_{i=1}^{m} \sum_{C \in \mathbb{C}_{p_i}} \left\{ \frac{x_i}{|\mathbb{S}_P|} \cdot \boldsymbol{v}(C, t_i) \right\} \\
&= \sum_{i=1}^{m} \sum_{C \in \mathbb{C}_{p_i}} \left\{ \frac{1}{|\mathbb{C}_{p_i}|} \cdot \boldsymbol{v}(C, t_i) \right\} \\
&= \sum_{i=1}^{m} \left\{ \frac{1}{|\mathbb{C}_{p_i}|} \sum_{C \in \mathbb{C}_{p_i}} \boldsymbol{v}(C, t_i) \right\} \\
&= \sum_{i=1}^{m} \boldsymbol{Avg}(p_i, t_i).
\end{aligned}$$

From this and (2), we conclude:

$$\sum_{i=1}^{m} \boldsymbol{Avg}(p_i, t_i) \leq \max_{\langle C_1,\ldots,C_m\rangle \in \mathbb{S}_P} \left\{ \sum_{i=1}^{m} \boldsymbol{v}(C_i, t_i) \right\}.$$

□

**Theorem 3** *$u_P = \sum_{i=1}^{m} \boldsymbol{M}(p_i, t_i)$ is an upper bound for the value of the best possible solution in the subspace $\mathbb{S}_P$ where $P = \langle p_1, \ldots, p_m\rangle$. In other words:*

$$\sum_{i=1}^{m} \boldsymbol{v}(C_i, t_i) \leq u_P$$

*for all $\langle C_1, \ldots, C_m\rangle \in \mathbb{S}_P$.*

***Proof*** If $\langle C_1, \ldots, C_m \rangle \in \mathbb{S}_P$, then $p_i = |C_i|$ for $i = 1, \ldots, m$. From this, it follows that:

$$\boldsymbol{M}(p_i, t_i) = \boldsymbol{M}(|C_i|, t_i). \tag{3}$$

Since $\boldsymbol{v}(C_i, t_i) \leq \boldsymbol{M}(|C_i|, t_i)$ for $i = 1, \ldots, m$, we have:

$$\sum_{i=1}^{m} \boldsymbol{v}(C_i, t_i) \leq \sum_{i=1}^{m} \boldsymbol{M}(|C_i|, t_i).$$

Based on this, and (1), we conclude that:

$$\sum_{i=1}^{m} \boldsymbol{v}(C_i, t_i) \leq \sum_{i=1}^{m} \boldsymbol{M}(p_i, t_i).$$

□

**Theorem 4** $\sum_{i=1}^{m} \min\{\boldsymbol{v}(C, t_i) : C \in \mathbb{C}_{p_i}\}$ *is a lower bound for the value of the best possible solution in the subspace* $\mathbb{S}_P$ *where* $P = \langle p_1, \ldots, p_m \rangle$*, and it is a worse lower bound than* $\sum_{i=1}^{m} \boldsymbol{Avg}(p_i, t_i)$*. In other words:*

$$\sum_{i=1}^{m} \min\{\boldsymbol{v}(C, t_i) : C \in \mathbb{C}_{p_i}\} \leq \sum_{i=1}^{m} \boldsymbol{Avg}(p_i, t_i) \leq \max_{\langle C_1, \ldots, C_m \rangle \in \mathbb{S}_P} \left\{ \sum_{i=1}^{m} \boldsymbol{v}(C_i, t_i) \right\}.$$

***Proof*** Recall that, for the arithmetic mean $\overline{y_1, \ldots, y_k}$ of a finite set $\{y_1, \ldots, y_k\} \subset \mathbb{R}$, the following holds:

$$\min\{y_1, \ldots, y_k\} \leq \overline{y_1, \ldots, y_k}.$$

Therefore, it follows that:

$$\min\{\boldsymbol{v}(C, t) : C \in \mathbb{C}_p\} \leq \frac{1}{|\mathbb{C}_p|} \sum \{\boldsymbol{v}(C, t) : C \in \mathbb{C}_p\}.$$

From this, and since $\boldsymbol{Avg}(p, t) = \frac{1}{|\mathbb{C}_p|} \sum \{\boldsymbol{v}(C, t) : C \in \mathbb{C}_p\}$ by definition, we have:

$$\min\{\boldsymbol{v}(C, t) : C \in \mathbb{C}_p\} \leq \boldsymbol{Avg}(p, t).$$

Now, based on this, and Theorem 2, we conclude:

$$\sum_{i=1}^{m} \min\{\boldsymbol{v}(C, t_i) : C \in \mathbb{C}_{p_i}\} \leq \sum_{i=1}^{m} \boldsymbol{Avg}(p_i, t_i) \leq \max_{\langle C_1, \ldots, C_m \rangle \in \mathbb{S}_P} \left\{ \sum_{i=1}^{m} \boldsymbol{v}(C_i, t_i) \right\}.$$

□

Since the performance measure for each coalition-to-task assignment is assumed to be known, these bounds can, in practice, be calculated without having to enumerate or generate any solutions. For instance, by enumerating all coalition-to-task values, of which there exists a total number of $m2^n$, the lower bounds can be calculated using a moving average. Also, we can calculate an upper bound for the solutions represented by the multiset permutations in the set $M \subseteq M_3$ according to Theorem 5, and thus also calculate an upper bound to the optimal solution for any SCSGA problem, as shown in Corollary 1, in the same manner.

**Theorem 5** *$U_M := \max_{P \in M} u_P$ is an upper bound for the optimal solution in the search space represented by $M \subseteq M_3$. In other words, if $S$ is a solution represented by a multiset permutation in $M \subseteq M_3$, then $\boldsymbol{V}(S) \leq U_M$.*

***Proof*** Let $\mathcal{S}_M := \bigcup_{P \in M} \mathbb{S}_P$, namely all solutions that are represented by the multiset permutations in $M$. We now want to prove that $\boldsymbol{V}(S) \leq U_M$ for all $S \in \mathcal{S}_M$. With this in mind, note that the following holds:

$$\max_{S \in \mathcal{S}_M} \boldsymbol{V}(S) = \max_{P \in M} \left\{ \max_{S \in \mathbb{S}_P} \boldsymbol{V}(S) \right\}. \tag{4}$$

According to Theorem 3, we have:

$$\max_{S \in \mathbb{S}_P} \boldsymbol{V}(S) \leq u_P.$$

Consequently:

$$\max_{P \in M} \left\{ \max_{S \in \mathbb{S}_P} \boldsymbol{V}(S) \right\} \leq \max_{P \in M} u_P.$$

From this, and (4), we conclude:

$$\max_{S \in \mathcal{S}_M} \boldsymbol{V}(S) = \max_{P \in M} \left\{ \max_{S \in \mathbb{S}_P} \boldsymbol{V}(S) \right\} \leq \max_{P \in M} u_P = U_M.$$

In other words:

$$\max_{S \in \mathcal{S}_M} \boldsymbol{V}(S) \leq U_M.$$

□

**Corollary 1** *$U_{M_3} := \max_{P \in M_3} u_P$ is an upper bound for the optimal solution. In other words, if $S^*$ is an optimal solution, then $\boldsymbol{V}(S^*) \leq U_{M_3}$.*

***Proof*** This follows directly from Theorem 5. □

### 4.3 Searching for solutions using branch-and-bound

We search for solutions by searching one subspace at a time, and discard subspaces that only contain suboptimal solutions when a subspace's upper bound is lower than or equal to (1) the *value of the best solution evaluated so far*, or (2) the *largest lower bound of all remaining subspaces*. With this in mind, consider the following observation: Finding a better solution than the best that we have found can potentially make it possible to discard (additional) subspaces. Thus, if we find better solutions earlier, we can potentially reduce execution time by decreasing the search space that we need to consider. To potentially take advantage of this observation, we design a mechanism, based on defining a precedence order that dictates the order for which we search subspaces, that ultimately makes it possible to find better solutions more quickly by using heuristics to guide search. In more detail, we use a variation of *best-first branch-and-bound* to search such promising subspaces first (and to discard subspaces that cannot possibly contain an optimal solution).

Note that the efficiency induced by any search order depends on the problem that is being solved. In our case, we assume that there exists no a priori knowledge in regards to the

domain, except for the utility function, and we instead have to take advantage of information that exists for all domains (e.g., subspaces and their bounds). It is possible to use potential domain-specific information when it is available, which is likely a more efficient strategy for solving many real-world problems. In any case, the domain-independent order of precedence for searching subspaces that we use is defined as follows:

$$P_1 \prec P_2 \text{ if } u_{P_1} + l_{P_1} > u_{P_2} + l_{P_2}$$

where $P_1 \prec P_2$ denotes that the subspace represented by the multiset permutation $P_1 \in M_3$ is searched before the subspace represented by $P_2 \in M_3$. $u_P$ and $l_P$ are defined as in the previous subsection.

With this in mind, we use Algorithm 1 to search a subspace $\mathbb{S}_P$ (represented by the multiset permutation $P \in M_3$) for $\arg\max_{S \in \mathbb{S}_P} \boldsymbol{V}(S)$ by running the search procedure `SearchSubspace`$(P, u_P, 1, \emptyset_{|T|}, 0.0, \emptyset_{|T|})$, where $\emptyset_{|T|}$ is a list of $m = |T|$ empty coalitions, and $u_P$ is a upper bound for the subspace represented by $P$ (defined in the previous subsection). If interrupted before termination, this procedure returns the best feasible solution found so far, denoted $S'$. Note that Algorithm 1 is a variation of *depth-first branch-and-bound*, and that we use a notation based on brackets to indicate an element at a specific position of a list or vector. For example, the notation $S[j]$ corresponds to the coalition $C_j \in S$, and the notation $A[i]$ corresponds to the agent $a_i \in A$.

To address the high memory requirements for generating and storing many multiset permutations (required for generating the precedence order), it is possible to generate and store multiset permutations in memory-bounded blocks (distinct sets of multiset permutations). These blocks can sequentially be generated and searched during partitioning. The more blocks we use, the less memory is required. In our case, we use each set $Q \in M_2$ generated in *step 2* during the partitioning phase (described in Sect. 4.1) to represent a block. In other words, each disjoint group of distinct multiset permutations *in which all multiset permutations have the same members* is searched in sequence according to some criterion. The particular criterion that we use is defined as:

$$Q_1 \prec Q_2 \text{ if } w_{Q_1} + f_{Q_1} > w_{Q_2} + f_{Q_2}$$

where $Q_1 \prec Q_2$ denotes that the solutions represented by the group of multiset permutations consisting of the members $q_1, \ldots, q_m$ is searched before the solutions represented by the group of multiset permutations consisting of the members $p_1, \ldots, p_m$, where $\{q_1, \ldots, q_m\} = Q_1$ and $\{p_1, \ldots, p_m\} = Q_2$, with $Q_1 \in M_2$ and $Q_2 \in M_2$. $w_Q$ and $f_Q$ are defined (similarly to the subspace bounds), for all $Q \in M_2$, as follows:

- $w_Q := \sum_{q \in Q} \{\max_{i=1,\ldots,m} \boldsymbol{M}(q, t_i)\}$;
- $f_Q := \sum_{q \in Q} \{\frac{1}{m} \sum_{i=1,\ldots,m} \boldsymbol{Avg}(q, t_i)\}$.

**Algorithm 1** : `SearchSubspace`$(P, \overrightarrow{u}, \overrightarrow{v}, \overrightarrow{S}, S', i, A, T)$
Recursively searches the subspace $\mathbb{S}_P$ represented by the multiset permutation $P$ using depth-first branch-and-bound. The input parameter $\overrightarrow{u}$ is an intermediary real-valued upper bound that corresponds to how much the current tentative value $\overrightarrow{v}$ can increase at subsequent recursion steps deeper in the recursion. $\overrightarrow{S}$ contains an intermediate partial solution, while $S'$ stores the best solution found so far. The non-negative integer $i$ equals the algorithm's current recursion depth (or, from another perspective, the index of the agent that we are currently assigning to a task). The set $A$ contains the agents in the SCSGA problem being solved, and $T$ corresponds to the list of tasks that we are assigning the agents to.
**Output:** $\arg\max_{S \in \mathbb{S}_P} \boldsymbol{V}(S)$.

**if** $i > |A|$ **then** ▷ All agents have been assigned to a coalition in $\overrightarrow{S}$.
  **return** $\overrightarrow{S}$
**end if**
**for** $j = 1, ..., |T|$ **do**
  **if** $|\overrightarrow{S}[j]| \neq P[j]$ **then**
    $\overrightarrow{S}[j] \leftarrow \overrightarrow{S}[j] \cup \{A[i]\}$ ▷ Assign agent $A[i]$ to the coalition $\overrightarrow{S}[j]$.
    **if** $|\overrightarrow{S}[j]| = P[j]$ **then** ▷ Update the intermediary values.
      $\overrightarrow{v} \leftarrow \overrightarrow{v} + \boldsymbol{v}(\overrightarrow{S}[j], T[j])$
      $\overrightarrow{u} \leftarrow \overrightarrow{u} - \boldsymbol{M}(P[j], T[j])$
    **end if**
    **if** $S' = \emptyset_{|T|}$ **or** $\overrightarrow{v} + \overrightarrow{u} > \boldsymbol{V}(S')$ **then** ▷ Check if a better solution is possible.
      $S'' \leftarrow$ `SearchSubspace`$(P, \overrightarrow{u}, \overrightarrow{v}, \overrightarrow{S}, S', i+1, A, T)$
      **if** $S' = \emptyset_{|T|}$ **or** $\boldsymbol{V}(S'') > \boldsymbol{V}(S')$ **then**
        $S' \leftarrow S''$ ▷ Update the best solution found so far.
      **end if**
    **end if**
    **if** *interrupt* has been requested **then**
      **return** $S'$
    **end if**
    **if** $|\overrightarrow{S}[j]| = P[j]$ **then** ▷ Reset the intermediary values.
      $\overrightarrow{v} \leftarrow \overrightarrow{v} - \boldsymbol{v}(\overrightarrow{S}[j], T[j])$
      $\overrightarrow{u} \leftarrow \overrightarrow{u} + \boldsymbol{M}(P[j], T[j])$
    **end if**
    $\overrightarrow{S}[j] \leftarrow \overrightarrow{S}[j] \backslash \{A[i]\}$ ▷ Remove agent $A[i]$ from the coalition $\overrightarrow{S}[j]$.
  **end if**
**end for**
**return** $S'$

$w_Q$ and $f_Q$ can, similarly to subspace bounds, be computed without having to enumerate or generate any solutions. Moreover, the algorithm can search these blocks in parallel using separate processes. Also, these blocks can be partitioned into several smaller parts (e.g., sub-blocks) to further decrease memory usage.

Note that, even though this algorithm is anytime in the sense that it can return a solution at any time during its search procedure, it still needs to generate a number of integer partitions before its search procedure can begin. However, this number, known as the partition function $\boldsymbol{p}(n)$, is relatively small—especially when compared to the number of possible solutions $m^n$. For example, the values of $\boldsymbol{p}(n)$ for $n = 1, \ldots, 20$ are (OEIS sequence $A000041$ [40]):

1, 2, 3, 5, 7, 11, 15, 22, 30, 42, 56, 77, 101, 135, 176, 231, 297, 385, 490, 627.

Finally, we now show how to calculate a worst-case guarantee on an anytime solution's quality. Let $S'$ be an anytime solution generated by our algorithm (i.e., the algorithm was interrupted before its search procedure was completed, and $S'$ is the best intermediary solution that was found), and let $R \subseteq M_3$ be the set of all multiset permutations that represent the subspaces that remain to be searched. An upper bound for the optimal solution can now be calculated as:

$$U_R = \max \left\{ \boldsymbol{V}(S'), \max_{P \in R} u_P \right\}.$$

This follows directly from Theorem 5. Consequently, if $S^*$ is an optimal solution, the following holds:

$$\boldsymbol{V}(S') \leq \boldsymbol{V}(S^*) \leq U_R.$$

Now, let $\rho = U_R / \boldsymbol{V}(S')$. As a consequence, the value $\boldsymbol{V}(S')$ of our anytime solution is at worst-case a factor $\rho$ worse than the value $\boldsymbol{V}(S^*)$ of an optimal solution—in other words $\boldsymbol{V}(S^*) \leq \rho \boldsymbol{V}(S')$, since:

$$\boldsymbol{V}(S^*) \leq U_R = \boldsymbol{V}(S') \frac{U_R}{\boldsymbol{V}(S')} = \rho \boldsymbol{V}(S').$$

## 5 Simple non-optimal algorithms

Due to the computational complexity of the optimal SCSGA problem, we consider two non-optimal algorithms that generate feasible solutions more efficiently: One algorithm that uses a greedy strategy to make locally optimal choices when constructing a solution, and a second anytime algorithm that continuously generates random solutions (and keeps track of the best) until it is interrupted.

### 5.1 Agent-based greedy algorithm

Our greedy algorithm, Algorithm 2, is abbreviated *AG* for agent greedy. It efficiently generates a solution by sequentially assigning agents to coalitions/tasks: First, it initializes a $m$-sized ordered coalition structure that has no agents assigned to any of its coalitions. Then, it sequentially assigns each agent to the coalition that would (locally) increase the value of the solution the most (or decrease its value the least). Moreover, this algorithm has a worst-case time-complexity of $\mathcal{O}(|T||A|)$, and it requires $\mathcal{O}(|T| + |A|)$ memory for storing the solution that it generates.

**Algorithm 2** : `AgentBasedGreedyAlgorithm`$(A, T)$
Greedily generates a solution by sequentially assigning agents to tasks. The set $A$ contains the agents in the SCSGA problem being solved, and $T$ corresponds to the list of tasks that we are assigning the agents to.
**Output:** A size-$m$ ordered coalition structure over $A$.

$S \leftarrow \emptyset_{|T|}$ ▷ $S$ is initialized to a list of $m = |T|$ empty coalitions.
**for** $i = 1, ..., |A|$ **do**
  $k \leftarrow 0$
  $u \leftarrow -\infty$
  **for** $j = 1, ..., |T|$ **do**
    $u' \leftarrow \boldsymbol{v}(S[j], T[j])$
    $S[j] \leftarrow S[j] \cup \{A[i]\}$ ▷ Temporarily assign agent $A[i]$ to the coalition $S[j]$.
    $u'' \leftarrow \boldsymbol{v}(S[j], T[j])$
    **if** $u'' - u' > u$ **then**
      $k \leftarrow j$ ▷ Update the best candidate coalition to add agent $A[i]$ to.
      $u \leftarrow u'' - u'$
    **end if**
    $S[j] \leftarrow S[j] \backslash \{A[i]\}$ ▷ Remove agent $A[i]$ from the coalition $S[j]$.
  **end for**
  $S[k] \leftarrow S[k] \cup \{A[i]\}$ ▷ Assign agent $A[i]$ to the coalition $S[k]$.
**end for**
**return** $S$

### 5.2 Pure random search algorithm

Our second non-optimal algorithm, Algorithm 3, is a pure random search algorithm abbreviated *PRS*. This algorithm continuously samples solutions from the entire SCSGA search space by assigning each agent to a randomly (following the discrete uniform probability distribution $\mathcal{DU}$) selected coalition. This process is only halted once the algorithm is interrupted, for example by an external event. It also keeps track of the best solution found so far. Following this procedure, each possible solution is clearly equally likely to occur (per sample), since each agent has an equal probability of being assigned to each task. Moreover, each ordered coalition structure is generated in $\mathcal{O}(|A| + |T|)$ time, and for a problem with $k \in \mathbb{N}$ optimal solutions, each sample has a probability of $km^{-n}$ for being optimal.

**Algorithm 3** : `PureRandomSearchAlgorithm`$(A, T)$
Continuously samples solutions from the entire search space, while keeping track of the best solution it finds, which it returns once interrupted. The set $A$ contains the agents in the SCSGA problem being solved, and $T$ corresponds to the list of tasks that we are assigning the agents to.
**Output:** A size-$|T|$ ordered coalition structure over $A$.

$S' \leftarrow \emptyset_{|T|}$ ▷ $S'$ is initialized to a list of $m = |T|$ empty coalitions.
**while** *interrupt* has not been requested **do**
  $S \leftarrow \emptyset_{|T|}$
  **for** $i = 1, ..., |A|$ **do**
    $r \leftarrow \mathcal{DU}(1, |T|)$ ▷ Assign a random integer between 1 and $|T|$ (both inclusive) to $r$.
    $S[r] \leftarrow S[r] \cup A[i]$ ▷ Assign agent $A[i]$ to the coalition $S[r]$.
  **end for**
  **if** $S' = \emptyset_{|T|}$ or $\boldsymbol{V}(S) > \boldsymbol{V}(S')$ **then**
    $S' \leftarrow S$ ▷ Update the best solution found so far.
  **end if**
**end while**
**return** $S'$

### 5.3 Feasible (suboptimal) solutions in conjunction with branch-and-bound

Apart from generating feasible solutions quickly, non-optimal algorithms (e.g., those described in the previous subsections) can also be used to generate an initial solution for branch-and-bound algorithms to reduce the initial lower bound for optimal solutions. For example, for our anytime algorithm described in Sect. 3: If we generate a solution $S'$, then, if $\boldsymbol{V}(S') \geq u_P$ for any $P \in M_3$, clearly all solutions in $\mathbb{S}_P$ can be discarded.

Non-optimal algorithms can also be used to potentially improve the lower bound for subspaces by generating an initial solution for each subspace. This initial solution can then be used to prioritize subspace-selection during search—for example, as described in the previous section, when using best-first branch-and-bound. By doing so, subspaces can potentially be discarded earlier. This approach can thus potentially decrease the total execution time by making it possible for an algorithm to make more informed decisions during search. However, this is only practical if the non-optimal algorithm is sufficiently efficient.

With this in mind, and to make this possible, we now extend Algorithm 2 to only construct solutions with fixed (predetermined) coalition-sizes, so that we can use it in conjunction with the MP algorithm. In more detail, this extension, Algorithm 4, uses a list of non-negative integers $P = \langle p_1, \ldots, p_m \rangle$, where $\sum_{i=1}^{m} p_i = n$, to only generate solutions $\langle C_1, \ldots, C_m \rangle$ with $|C_i| = p_i$ for $i = 1, \ldots, m$.

This algorithm has the same worst-case characteristics as Algorithm 2. In other words, it has a worst-case time-complexity of $\mathcal{O}(|T||A|)$, and a worst-case memory consumption of $\mathcal{O}(|T| + |A|)$.

**Algorithm 4** : `AgentBasedFCSGreedyAlgorithm`$(P, A, T)$
Given that $P = \langle p_1, \ldots, p_m \rangle$ represents the subspace $\mathbb{S}_P \neq \emptyset$, this algorithm greedily generates a solution $S \in \mathbb{S}_P$ to a SCSGA problem instance by sequentially assigning agents to tasks. The set $A$ contains the agents in the SCSGA problem being solved, and $T$ corresponds to the list of tasks that we are assigning the agents to.
**Output:** An ordered coalition structure $\langle C_1, \ldots, C_m \rangle$ over $A$ with $|C_i| = p_i$.

$S \leftarrow \emptyset_{|T|}$ ▷ $S$ is initialized to a list of $m = |T|$ empty coalitions.
**for** $i = 1, ..., |A|$ **do**
  $k \leftarrow 0$
  $u \leftarrow -\infty$
  **for** $j = 1, ..., |T|$ **do**
    **if** $|S[j]| < P[j]$ **then**
      $u' \leftarrow \boldsymbol{v}(S[j], T[j])$
      $S[j] \leftarrow S[j] \cup \{A[i]\}$ ▷ Temporarily assign agent $A[i]$ to the coalition $S[j]$.
      $u'' \leftarrow \boldsymbol{v}(S[j], T[j])$
      **if** $u'' - u' > u$ **then**
        $k \leftarrow j$ ▷ Update the best candidate coalition to add agent $A[i]$ to.
        $u \leftarrow u'' - u'$
      **end if**
      $S[j] \leftarrow S[j] \backslash \{A[i]\}$ ▷ Remove agent $A[i]$ from the coalition $S[j]$.
    **end if**
  **end for**
  $S[k] \leftarrow S[k] \cup \{A[i]\}$ ▷ Assign agent $A[i]$ to the coalition $S[k]$.
**end for**
**return** $S$

## 6 Evaluation and results

A common approach for evaluating optimization algorithms is to use standardized problem instances for benchmarking. To our knowledge, no such instances exist for the SCSGA problem. We therefore translate standardized problem instances from a similar domain. More specifically, we extend established methods for synthetic problem set generation used for benchmarking CSG algorithms. The extended methods are then used to generate difficult problem sets of varying distribution and complexity that we use to benchmark our algorithms.

Larson and Sandholm [24] provided standardized synthetic problem instances for the coalition structure generation problem by using normal and uniform probability distributions to generate randomized values for coalitions. Following Rahwan et al. [34], we denote these distributions *NPD* (normal probability distribution) and *UPD* (uniform probability distribution), respectively.

To benchmark our algorithm, we extend these distributions to our domain, so that we also take tasks into consideration. In addition to NPD and UPD, we also extend and use *NDCS* (normally distributed coalition structures)—a distribution that was proposed by Rahwan et al. [34] for benchmarking coalition structure generation algorithms. Our extensions of these probability distributions, to our task-dependent domain, are defined as follows:

- **UPD:** $\boldsymbol{v}(C, t) \sim |C| \cdot \boldsymbol{\mathcal{U}}(a, b)$, where $a = 0$ and $b = 1$;
- **NPD:** $\boldsymbol{v}(C, t) \sim |C| \cdot \boldsymbol{\mathcal{N}}(\mu, \sigma^2)$, where $\mu = 1$ and $\sigma = 0.1$;
- **NDCS:** $\boldsymbol{v}(C, t) \sim \boldsymbol{\mathcal{N}}(\mu, \sigma^2)$, where $\mu = |C|$ and $\sigma = \max(\sqrt{|C|}, \epsilon)$;

for all $C \subseteq A$ and $t \in T$, where $\boldsymbol{\mathcal{N}}(\mu, \sigma^2)$ and $\boldsymbol{\mathcal{U}}(a, b)$ are the normal and uniform distributions, respectively, and $0 < \epsilon \ll 1$. For our experiments, we use $\epsilon = 10^{-9}$.

In addition to these extensions, we also define and use two additional distributions for generating synthetic problem instances, with the purpose to more closely model certain

types of simplified real-world task-dependent scenarios. We denote these *NSD* (normal skill-based distribution) and *NRD* (normal relation-based distribution). NSD models that agents may have different skills that alter their suitability for handling certain tasks, while NRD provides a simplistic model for the phenomenon that an agent's utility is potentially also dependent on the other agents in the coalition/team as well (i.e., an agent's contributions to a coalition depends both on the coalition's goal, and the agent's relationship to others). With this in mind, define:

- **Skill-level:** $s(a,t) \sim \mathcal{N}(\mu, \sigma^2)$ for all $a \in A$ and $t \in T$;
- **Relational utility:** $r(\{a,b\},t) \sim \mathcal{N}(\mu, \sigma^2)$, for all $\{a,b\} \in \binom{A}{2}$ and $t \in T$;

where $\mu = 1$, $\sigma = 0.1$, and $\binom{X}{2} = \{\{a,b\} : a,b \in X, a \neq b\}$, namely the set of all size-2 subsets of $X$. An interpretation of $s(a,t)$ is that it represents agent $a$'s suitability (or skill-level) for handling task $t$, while $r(\{a,b\},t)$ represents agent $a$'s potential utility for working together with agent $b$ towards completing task $t$. We now define NSD and NRD as follows:

- **NSD:** $v(C,t) := \sum_{a \in C} s(a,t)$;
- **NRD:** $v(C,t) := \sum_{\{a,b\} \in \binom{C}{2}} r(\{a,b\},t)$;

for all $C \subseteq A$ and $t \in T$. Note that we expect AG (namely Algorithm 2, the agent-based greedy algorithm) to always generate an optimal solution for problem sets generated with NSD, since, for this distribution, an agent's contribution to a coalition is not affected by the coalition's other members.

The results of our experiments that were based on these distributions, and from applying the algorithm to a commercial strategy game, are presented in Sects. 6.2, and 6.3, respectively.

### 6.1 Implementation and hardware

Our algorithm was implemented in *C++11*, and all synthetic problem sets were generated using the random number generators `normal_distribution` (for NPD, NSD, NRD and NDCS) and `uniform_real_distribution` (for UPD) from the *C++ Standard Library*. All tests were conducted using *Windows 10* (x64), an *Intel* 7700K 4.2GHz CPU, and 16GB of 3GHz DDR4 memory. We used version 12.5 and 12.8 of *IBM ILOG CPLEX Optimization Studio* for our CPLEX benchmarks.

### 6.2 Results of the synthetic experiments

The result of each experiment was produced by calculating the average of the resulting values (i.e., time measurements and numerical values of solution quality) from 50 generated problem sets per probability distribution and experiment. Also, to compete on equal terms, both CPLEX and our MP algorithm were only allowed to use a single CPU thread during all tests (even though both approaches support parallel computing). Furthermore, the algorithms did not have any a priori knowledge of the problems that they were given to solve, and we use the abbreviation *MP+AG* to denote using MP in conjunction with AG's extended version (Algorithm 4) to generate initial solutions, and calculating (potentially) better lower bounds for subspaces before searching them. Finally, following best practice, we plot the 95% confidence interval in all graphs. The statistical significance of the means' differences can thus be compared, since if two different series have non-overlapping confidence intervals, it is equivalent to that the null hypothesis is rejected for a $t$-test with $\alpha = 0.05$.

The execution time to find an optimal solution for 8 tasks is plotted using a logarithmic scale in Fig. 1. The results in these graphs show that our algorithm (MP) is considerably

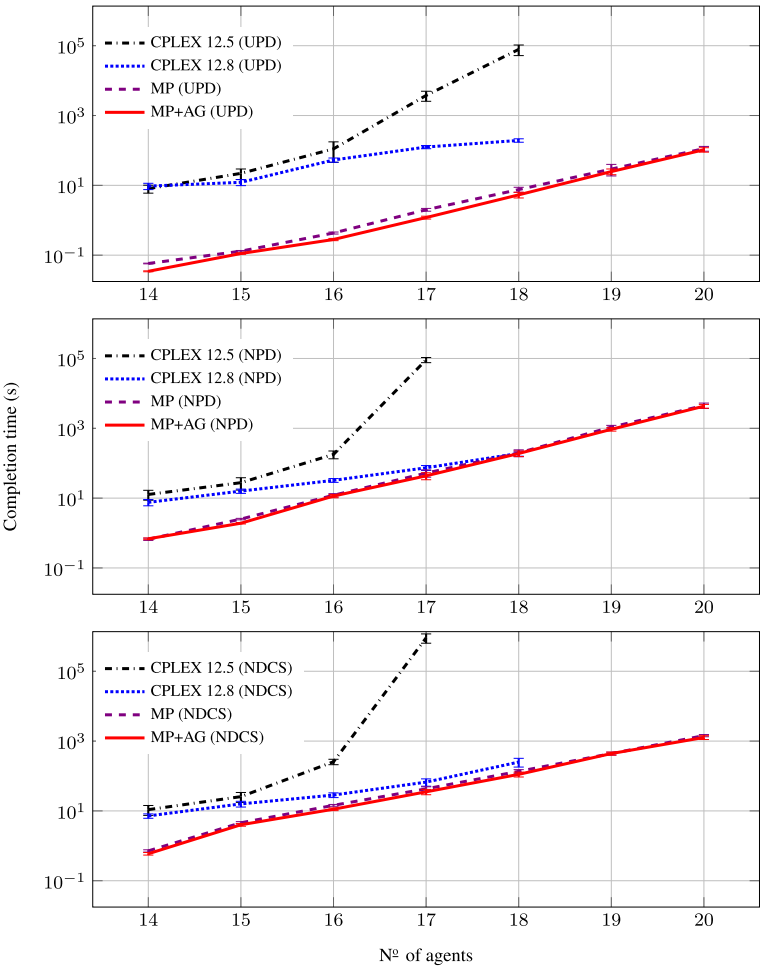


**Fig. 1** Execution time for optimally solving synthetic problems with 8 tasks. The values for the coalition-to-task assignments were generated using **UPD** (top), **NPD** (middle) and **NDCS** (bottom)

faster (often by many orders of magnitude) than CPLEX for all distributions and almost all problem sets. For example, for 16 agents and UPD, our algorithm completes its search in approximately 1% of the time that CPLEX needs.

For more than 18 agents, CPLEX's search procedure always crashed due to running out of memory. MP, however, managed to find optimal solutions for all problems within a reasonable time frame. In these logarithmic graphs, MP and CPLEX 12.8 are clearly linear, while CPLEX 12.5 is not. Furthermore, our benchmarks show that MP's search efficiency is sensitive to the distribution of utility values. This was expected, since MP is dependent on its ability to discard subspaces, and this ability is affected by the distribution of utility values in the problem being solved.

Using MP in conjunction with Algorithm 2 (the agent-based greedy algorithm) slightly improved search times for most problem sets. However, comparing MP to MP+AG shows, in general, a rather low difference in performance. This indicates that the lower bounds

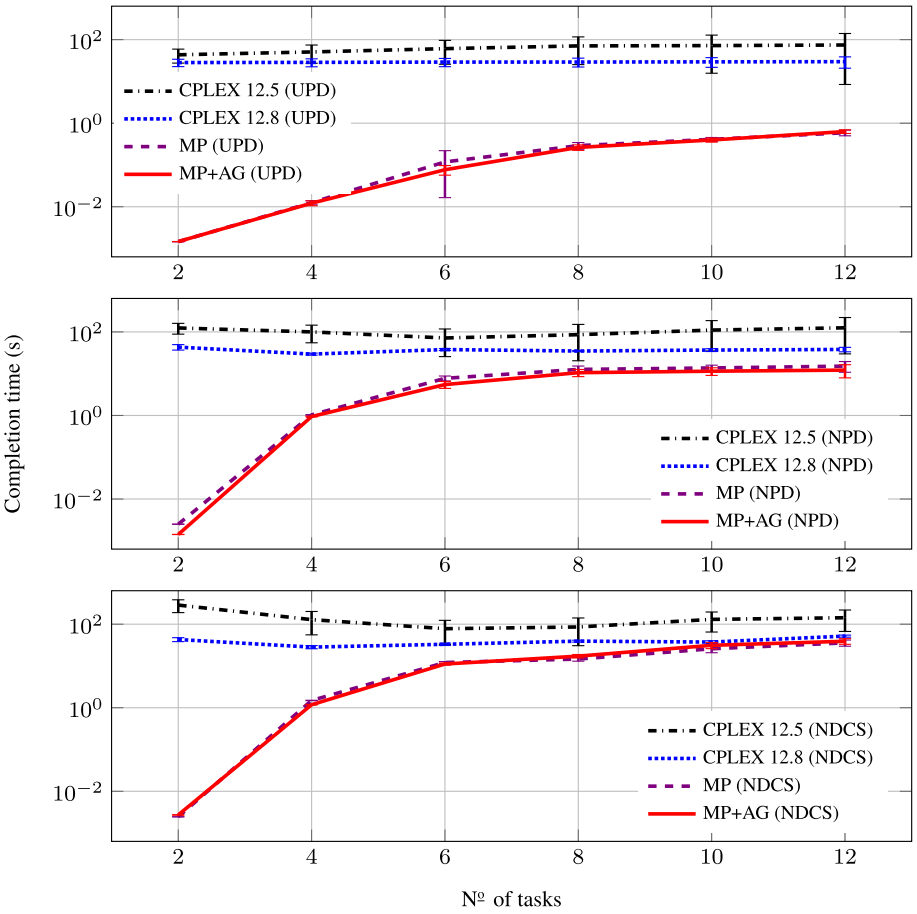


**Fig. 2** The execution time to optimally solve synthetic problems with 16 agents generated using **UPD** (top), **NPD** (middle) and **NDCS** (bottom)

calculated according to Theorem 3 are sufficiently tight (high) compared to the lower bounds generated by AG.

We plot the execution time to find an optimal solution for 16 agents in Fig. 2, and instead look at how the number of tasks (2 to 12) affect MP's performance. We used 16 agents in these benchmarks, since for problems with more agents, CPLEX 12.5 did not manage to find optimal solutions within a reasonable time frame, and CPLEX 12.8 often crashed due to insufficient memory for $|A| > 16$ (and it always crashed for $|A| > 18$).

As can be seen in Fig. 2, our algorithm is considerably faster than CPLEX for these problem sets as well (especially for problems with UPD-distributed utility values). Similarly as in previous benchmarks, all algorithms performed worst when the problem sets were generated with normal distributions.

For the benchmarks with few tasks (2 to 6), MP was extremely fast, and it did not need to search many subspaces (or evaluate many solutions) before it could guarantee that it had found an optimal ordered coalition structure.

In our next five benchmarks, we investigate the quality of the anytime solutions generated by MP and PRS. We used 13 agents and 14 tasks for this purpose, resulting in a total number of

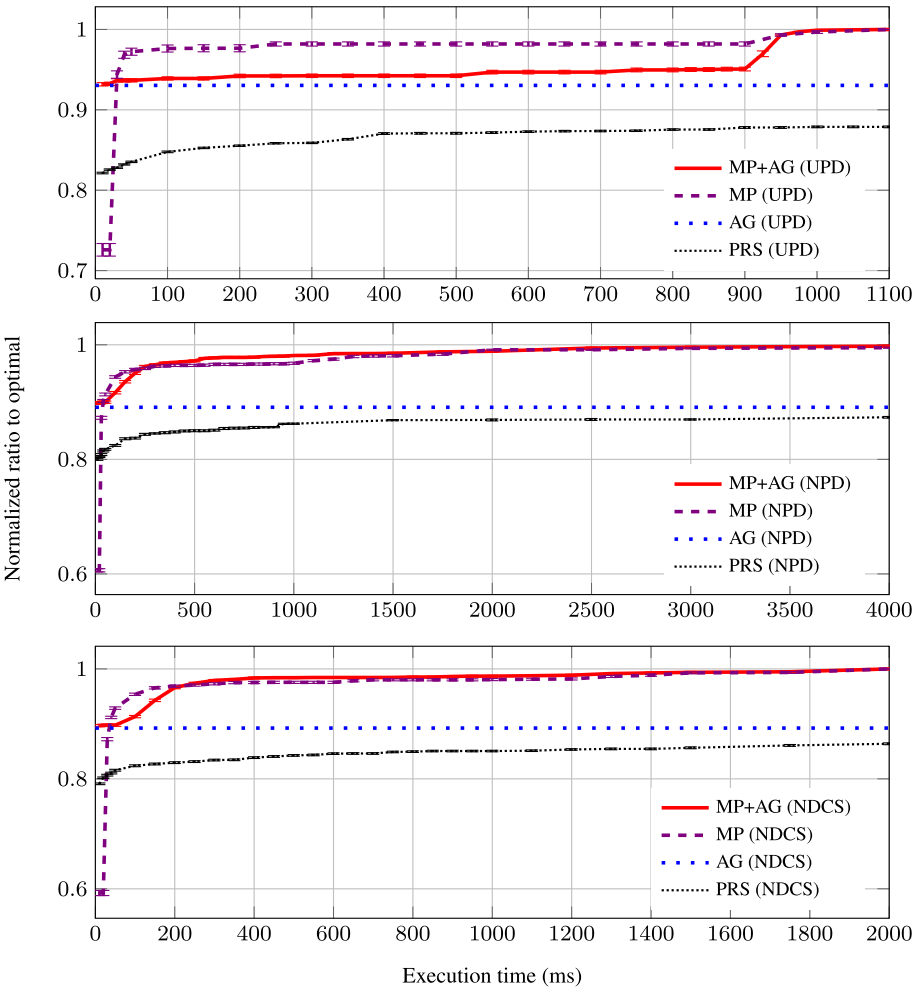


**Fig. 3** The normalized ratio to optimal obtained by the different algorithms for problem sets generated using **UPD** (top), **NPD** (middle) and **NDCS** (bottom) with 13 agents and 14 tasks

$14^{13} \approx 8 \times 10^{14}$ possible solutions. Our results from these experiments are shown in Figs. 3, 4 and 5. In these graphs, the execution time is shown on the x-axis, and the *normalized ratio to optimal* on the y-axis. This ratio, for a feasible solution $S'$, is defined as the following value:

$$\frac{\boldsymbol{V}(S') - \boldsymbol{V}(S_*)}{\boldsymbol{V}(S^*) - \boldsymbol{V}(S_*)}$$

where $S^*$ is an optimal solution, and $S_*$ is a lowest valued (i.e., worst) solution. We deem that this ratio gives a better indication to a solution's quality than using the seemingly more problematical $\boldsymbol{V}(S')/\boldsymbol{V}(S^*)$ (which is e.g., arguably misleading if the value of a solution is negative). Also, note that in these tests, PRS generated and evaluated approximately 4.4 million solutions per second, and for the execution time in these graphs, CPLEX failed to find any feasible solution at all.

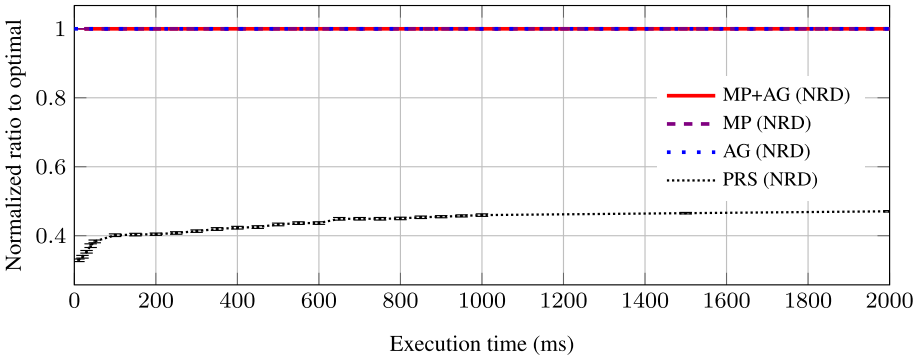


**Fig. 4** The normalized quality ratio of solutions obtained by the different algorithms for problem sets based on **NRD** with 13 agents and 14 tasks. Note that, for these problem sets, all algorithms, except PRS, always generated optimal solutions instantly

As can be seen in Fig. 3, both MP and MP+AG generated 90%-efficient solutions after roughly 50 ms for all three problem sets. Moreover, they performed similarly for the problem sets generated with NPD and NDCS, while MP generated intermediary solutions of higher quality in the top-most graph that represents the problems with UPD-distributed utility values. These results corroborates our earlier hypothesis that MP's lower bounds for subspaces, namely those that are based on Theorem 3, are sufficiently close (in value) to those generated by AG.

Furthermore, it took roughly the same time for MP and MP+AG to find optimal solutions in all benchmarks, and a better-than 99%-efficient solution is always found after 1 s for UPD, roughly 2 s for NPD, and approximately 1.5 s for NDCS—in other words, both MP and MP+AG found near-optimal solutions very rapidly for all distributions and benchmarks. Finally, as expected, PRS generated the worst solutions for all problem sets and execution times, except for when MP was interrupted before it had managed to generate any intermediary solution at all. In such cases, MP degenerates to generate an almost arbitrarily bad solution, while MP+AG returns an ordered coalition structure greedily constructed by AG.

Our last two benchmarks, presented in Figs. 4 and 5, show the quality of the solutions found by our algorithms for problems generated with NSD and NRD. As expected for NSD, both AG and MP+AG generated an optimal solution instantly, while MP had to search much longer (roughly 6 s on average) before it could guarantee that an optimal solution had been found.

For NRD-distributed utility values, all algorithms, except PRS, always found optimal solutions instantly. This could indicate that AG is optimal for this type of problem as well, and that MP finds and searches the best subspace first with a very high probability when the utility values are NRD-distributed. This may seem unlikely, and it is perhaps more likely that most solutions, for this problem-type, are optimal. This is however not the case, since if most solutions are optimal, PRS would have generated near-optimal solutions in our NRD-benchmarks very rapidly, and our experimental data clearly shows the opposite to be true. What we can say, however, is that for NRD, both MP, MP+AG and AG generate close-to-optimal solutions very quickly with a seemingly high probability.

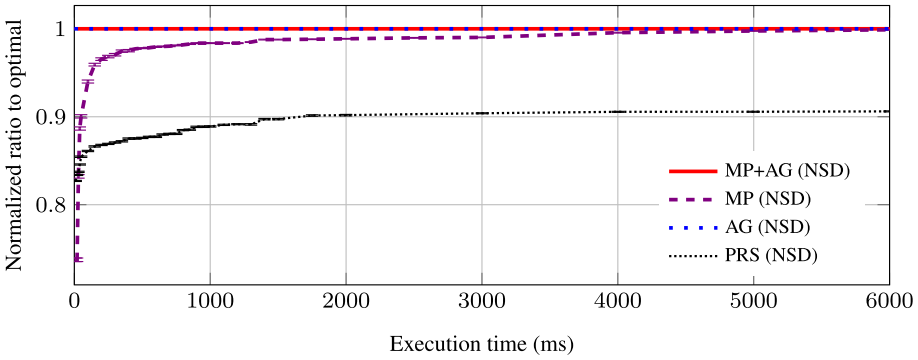


**Fig. 5** The normalized quality ratio of solutions obtained by the different algorithms for problem sets based on **NSD** with 13 agents and 14 tasks

### 6.3 Applying the MP algorithm to Europa Universalis 4

To empirically show that MP can be used to coordinate agents in a real-world scenario, we applied it to improve the coordination skills of computer-based players in the strategy game *Europa Universalis 4* (EU4)—a very complex partially observable *simultaneous move game*[1] (with many stochastic elements), in which players are required to act and reason in real-time.[2] This game is very popular, with more than one million copies sold worldwide [48], and it has many thousands of active players. Moreover, it was developed by the Swedish game development company *Paradox Development Studio*, and released commercially in late 2013. A screenshot showing EU4, from the perspective of a player playing as Sweden, is shown in Fig. 6.

In EU4, hundreds of simulated countries, both computer- and human-controlled alike, face off against each other, and have to coordinate themselves to defeat their opponents—they have to form alliances, administer their land, conduct trade, invest in new technologies, steer armies, manage diplomacy, and wage war. To handle this multi-faceted complexity, a computer-based player consists of several distinct computational subsystems, each hand-crafted to manage an important aspect of the game.

In particular, and more importantly pertaining to our subject at hand, there is one such subsystem that makes decisions in regards to which *region of interest* (a set of provinces that the game-playing agent deems important) that each of its player's different *armies*[3] should be deployed (assigned) to. This is typically a very difficult problem to solve—not only because the armies are heterogeneous, but also since the regions are complicated spatial systems themselves. Moreover, they affect each other, and are continuously transformed/altered as a consequence of stochastic processes (e.g., random events, the environment, battles), and player interactions (e.g., wars, edicts). See Fig. 7 for a map portraying the game's different regions and provinces. Note that, once an army has been deployed to a specific region, there's

[1] A simultaneous move game is a game in which players have to perform/choose actions without knowing the actions that will be performed/chosen by the other players.

[2] With real-time, we mean that the game continuously updates itself (and the game's world state) at a high rate, even when a player is not performing any actions. The game is thus not turn-based.

[3] Simply put, an army in EU4 is a combative entity that consists of a set of regiments. A regiment is a number of homogeneous soldiers that are either classed as infantry, cavalry or artillery. In addition, each army can be commanded by a general. A general can improve the army's skills, such as increasing its movement speed, or upgrade its siege ability when it is trying to take control of a province.


**Fig. 6** A screenshot showing EU4's user interface and a small portion of its world map (the game board). Note that, in this image, each small shield (with an accompanying army size number) represents an army positioned at a specific *province*—i.e., a small geographic area where armies can be stationed. A province also provides effects (e.g., increased income) to the player who controls it

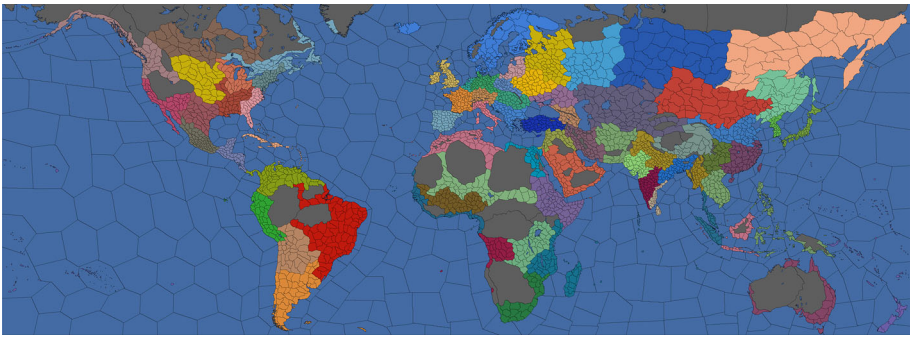

**Fig. 7** A map of the game's different regions and provinces. Each colour represents a specific region (which consists of a unique set of provinces)

another specialized system that handles the army's more direct low-level control, for example by deciding on exactly where to position individual armies. Finally, note that adding an agent to a coalition in EU4 may decrease its value, since the regions' have supply-based limitations that can reduce larger coalitions' values.

In light of these observations, to play this dynamic game successfully, computer-controlled players continuously try to assign their armies to the game's different regions. In more detail, in EU4, this problem can be (and is) modelled as follows:

**Input:** A set of armies $A = \{a_1, \ldots, a_n\}$, a list of regions $R = \langle r_1, \ldots, r_m \rangle$, and the utility function $\boldsymbol{v}(C, r) \mapsto \mathbb{R}$ that represents the value for assigning $C \subseteq A$ to $r \in R$.
**Output:** An ordered coalition structure $\langle C_1, \ldots, C_m \rangle$ that maximizes $\sum_{i=1}^{m} \boldsymbol{v}(C_i, r_i)$.

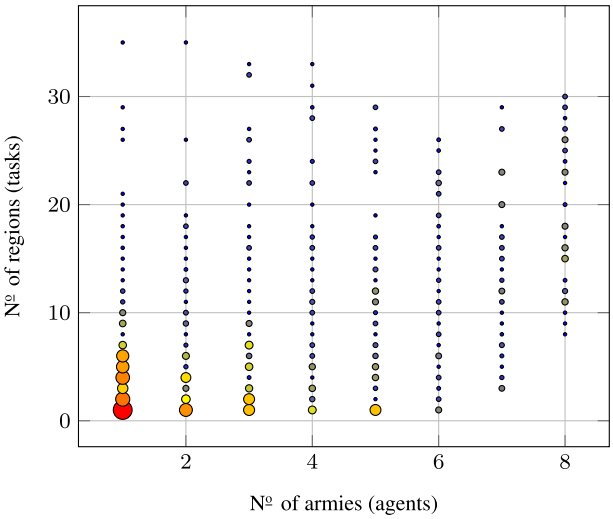


**Fig. 8** A visualization of the samples that were generated using EU4. A larger scatter mark indicates that there were more problem sets for that given number of agents and tasks

The computer-based players in EU4 solve this SCSGA problem using an *ad hoc* random search algorithm—a specialized non-optimal algorithm specifically designed for the context of EU4 that is inherently based on expert knowledge and domain-dependent heuristics to guide its search procedure.

In collaboration with the game's developers, we benchmarked MP against their algorithm. To do so, we used the same problem sets (generated by the game) for both algorithms. The utility function, which is defined by the developers, was given to us as a *black-box* function. We ran both algorithms while the game was playing, measured the algorithms' execution time, and compared the values of the solutions that the two algorithms generated. The following constraints held for all EU4 problem sets: $n \in [1, 8]$ and $m \in [1, 35]$—and there were at most $30^8 \approx 6.56 \times 10^{11}$ solutions for the largest problem sets that were generated by the game (namely problems with $n = 8$ armies and $m = 30$ regions). Note that all regions are never part of any problem set's input at the same time. This is because, typically, the game-playing agent is only interested in a few of them at a time, thus making it possible to dramatically decrease the problem's complexity by preventing the algorithm to consider certain solutions. A scatter plot of the different problems that were solved is shown in Fig. 8, as to give a hint on the suitability of using MP to solve this problem.

The results from running our experiments show that applying the algorithm to EU4 was a great success in terms of improving the computer-based players' performance (an increase of solution quality) and computational efficiency (reduction of execution time). In fact, our algorithm managed to find an optimal solution for all problems in less time than a game's frame (approximately $1/20 \approx 0.05$ s); and compared to the developer's algorithm, our algorithm decreased the execution time to, on average, 0.24% of theirs. Our algorithm also increased the numerical quality of solutions by, on average, 565% over theirs, and their algorithm seldom managed to find an optimal solution. These are the results from solving, in total, 13,922 problem sets that were generated while playing the game during 3 separate simulated sessions. Note that these results are not only promising in terms of performance, but also on the basis of generalization: If the utility/value functions that are used in EU4 were

to change (for example due to environment alterations as a result of game updates), their *ad hoc* algorithm might have to be altered. This is not the case for our algorithm, since it does not make any assumptions on the coalitions' utility functions or the game's rules. Therefore, our algorithm is potentially cheaper and easier to maintain. Also, there are many reasons to why strategy games are ideal for empirically evaluating and testing AI algorithms, and other authors have discussed these reasons extensively in earlier publications, see e.g., [7].

## 7 Conclusions

In this paper, we presented an anytime algorithm that solves the simultaneous coalition structure generation and assignment (abbreviated SCSGA) problem by integrating assignment into the formation of coalitions. We are, to the best of our knowledge, the first to study and solve this specific problem in a formal context.

Moreover, to benchmark the presented algorithm, we extended established methods for benchmarking coalition structure generation algorithms to our domain, and then used synthetic problem sets to empirically evaluate its performance. We benchmarked our algorithm against CPLEX, due to the lack of specialized algorithms for the simultaneous coalition structure generation and assignment problem.

Our results demonstrate that our algorithm is superior to CPLEX in solving synthetic instances of the simultaneous coalition structure generation and assignment problem. For example, when solving synthetic problem sets with 14 agents and 8 tasks, our algorithm finds an optimal solution in, on average, 5% of the time that CPLEX needs. Also, our algorithm does not have to search for very long before it can find high-quality solutions—even when interrupted prior to finishing a complete search. For example, it took our branch-and-bound algorithm less than 1 s to find an 95%-efficient solution in all of our benchmarks. This is potentially beneficial in many real-time systems (e.g., real-world multi-agent systems), in which feasible solutions must be available fast, but optimal coalition structures are not necessarily required. Apart from these properties, our algorithm is able to give worst-case guarantees on solutions. Moreover, our results indicate that SCSGA problems with utility values distributed in certain ways can be solved efficiently in linear time. Finally, by using our algorithm to improve the coordination of computer-based players in Europa Universalis 4, we demonstrated that it can be used to solve a real-world simultaneous coalition structure generation and assignment problem more efficiently than a previous approach. For example, our algorithm increased the numerical quality of solutions in this game by, on average, 565%, while simultaneously decreasing the execution time required to search for solutions.

For future work, it would be interesting to investigate other approaches to solving this problem, including dynamic programming and approximation algorithms. Also, problems with many agents are still computationally difficult to solve, and it would therefore be an important (and interesting) endeavor to investigate if machine learning, metaheuristic algorithms or Monte Carlo methods could be applied to solve difficult large-scale SCSGA problems.

**Acknowledgements** Open access funding provided by Linköping University. This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

## References

1. Albizuri, M. J., & Zarzuelo, J. M. (2000). Coalitional values for cooperative games with r alternatives. *Top*, *8*(1), 1–30.
2. Amer, R., Carreras, F., & Magaña, A. (1998). Extension of values to games with multiple alternatives. *Annals of Operations Research*, *84*, 63–78.
3. Andrews, G., & Eriksson, K. (2004). *Integer partitions*. Cambridge: Cambridge University Press.
4. Bolger, E. M. (1993). A value for games with n players and r alternatives. *International Journal of Game Theory*, *22*(4), 319–334.
5. Bolger, E. M. (2000). A consistent value for games with n players and r alternatives. *International Journal of Game Theory*, *29*(1), 93–99.
6. Bolger, E. M. (2002). Characterizations of two power indices for voting games with r alternatives. *Social Choice and Welfare*, *19*(4), 709–721.
7. Buro, M. (2003). Real-time strategy games: A new AI research challenge. In: *International joint conference on artificial intelligence* (pp. 1534–1535).
8. Chalkiadakis, G., Elkind, E., Markakis, E., Polukarov, M., & Jennings, N. R. (2010). Cooperative games with overlapping coalitions. *Journal of Artificial Intelligence Research*, *39*, 179–216.
9. Chu, P. C., & Beasley, J. E. (1997). A genetic algorithm for the generalised assignment problem. *Computers & Operations Research*, *24*(1), 17–23.
10. Dang, V. D., Dash, R. K., Rogers, A., & Jennings, N. R. (2006). Overlapping coalition formation for efficient data fusion in multi-sensor networks. *AAAI*, *6*, 635–640.
11. Dinar, A., Moretti, S., Patrone, F., & Zara, S. (2006). Application of stochastic cooperative games in water resources. In R.-U. Goetz & D. Berga (Eds.), *Frontiers in water resource economics* (pp. 1–20). Berlin: Springer.
12. Doherty, P., Heintz, F., & Landén, D. (2010). A distributed task specification language for mixed-initiative delegation. In *International conference on principles and practice of multi-agent systems* (pp. 42–57). Berlin: Springer.
13. Epstein, D., & Bazzan, A. L. (2013). Distributed coalition structure generation with positive and negative externalities. In *Portuguese conference on artificial intelligence* (pp. 408–419). Berlin: Springer.
14. Fatima, S., & Wooldridge, M. (2018). Computing optimal coalition structures in polynomial time. *Autonomous Agents and Multi-Agent Systems*, *33*, 1–49.
15. Gerkey, B. P., & Matarić, M. J. (2004). A formal analysis and taxonomy of task allocation in multi-robot systems. *The International Journal of Robotics Research*, *23*(9), 939–954.
16. Grabisch, M., & Rusinowska, A. (2010). A model of influence with an ordered set of possible actions. *Theory and Decision*, *69*(4), 635–656.
17. Habib, F. R., Polukarov, M., & Gerding, E. H. (2017). Optimising social welfare in multi-resource threshold task games. In *International conference on principles and practice of multi-agent systems* (pp. 110–126). Berlin: Springer.
18. Han, Z., & Poor, H. V. (2009). Coalition games with cooperative transmission: a cure for the curse of boundary nodes in selfish packet-forwarding wireless networks. *IEEE Transactions on Communications*, *57*(1), 203–213.
19. Horling, B., & Lesser, V. (2004). A survey of multi-agent organizational paradigms. *The Knowledge Engineering Review*, *19*(4), 281–316.
20. International, M. (2019). International activity report 2018. Retrieved February 25, 2020 from https://www.msf.org/international-activity-report-2018/.
21. Kelso, A. S, Jr., & Crawford, V. P. (1982). Job matching, coalition formation, and gross substitutes. *Econometrica: Journal of the Econometric Society*, *50*, 1483–1504.
22. Kuhn, H. W. (1955). The Hungarian method for the assignment problem. *Naval Research Logistics (NRL)*, *2*(1–2), 83–97.
23. Landén, D., Heintz, F., & Doherty, P. (2010). Complex task allocation in mixed-initiative delegation: A UAV case study. In *International conference on principles and practice of multi-agent systems* (pp. 288–303). Berlin: Springer.

24. Larson, K. S., & Sandholm, T. W. (2000). Anytime coalition structure generation: An average case study. *Journal of Experimental & Theoretical Artificial Intelligence*, *12*(1), 23–42.
25. Munkres, J. (1957). Algorithms for the assignment and transportation problems. *Journal of the Society for Industrial and Applied Mathematics*, *5*(1), 32–38.
26. Ono, R. (2001). Values for multialternative games and multilinear extensions. In M. Holler & G. Owen (Eds.), *Power indices and coalition formation* (pp. 63–86). Berlin: Springer.
27. Pentico, D. W. (2007). Assignment problems: A golden anniversary survey. *European Journal of Operational Research*, *176*(2), 774–793.
28. Präntare, F., & Heintz, F. (2018). An anytime algorithm for simultaneous coalition structure generation and assignment. In *International conference on principles and practice of multi-agent systems* (pp. 158–174). Berlin: Springer.
29. Präntare, F., Ragnemalm, I., & Heintz, F. (2017). An algorithm for simultaneous coalition structure generation and task assignment. In *International conference on principles and practice of multi-agent systems* (pp. 514–522). Berlin: Springer.
30. Rahwan, T., & Jennings, N. R. (2008). An improved dynamic programming algorithm for coalition structure generation. In *Proceedings of the 7th international joint conference on autonomous agents and multiagent systems* (Vol. 3, pp. 1417–1420). International Foundation for Autonomous Agents and Multiagent Systems.
31. Rahwan, T., Michalak, T., Wooldridge, M., & Jennings, N. R. (2012). Anytime coalition structure generation in multi-agent systems with positive or negative externalities. *Artificial Intelligence*, *186*, 95–122.
32. Rahwan, T., Michalak, T. P., & Jennings, N. R. (2012). A hybrid algorithm for coalition structure generation. In *AAAI* (pp. 1443–1449).
33. Rahwan, T., Michalak, T. P., Wooldridge, M., & Jennings, N. R. (2015). Coalition structure generation: A survey. *Artificial Intelligence*, *229*, 139–174.
34. Rahwan, T., Ramchurn, S. D., Jennings, N. R., & Giovannucci, A. (2009). An anytime algorithm for optimal coalition structure generation. *Journal of Artificial Intelligence Research*, *34*, 521–567.
35. Ray, D., & Vohra, R. (2015). Coalition formation. In H. P. Young & S. Zamir (Eds.), *Handbook of game theory with economic applications* (Vol. 4, pp. 239–326). Amsterdam: Elsevier.
36. Sandholm, T., Larson, K., Andersson, M., Shehory, O., & Tohmé, F. (1999). Coalition structure generation with worst case guarantees. *Artificial Intelligence*, *111*(1–2), 209–238.
37. Shehory, O., & Kraus, S. (1998). Methods for task allocation via agent coalition formation. *Artificial Intelligence*, *101*(1–2), 165–200.
38. Shoham, Y., Powers, R., & Grenager, T. (2003). Multi-agent reinforcement learning: A critical survey. *Web Manuscript*.
39. Skibski, O., Michalak, T. P., Sakurai, Y., Wooldridge, M., & Yokoo, M. (2015). A graphical representation for games in partition function form. In *Twenty-ninth AAAI conference on artificial intelligence*.
40. Sloane, N. J. A. (2019). The on-line encyclopedia of integer sequences, sequence a000041. Retrieved February 25, 2020 from https://oeis.org/A000041.
41. Stojmenović, I., & Zoghbi, A. (1998). Fast algorithms for genegrating integer partitions. *International Journal of Computer Mathematics*, *70*(2), 319–332.
42. Stone, P., & Veloso, M. (2000). Multiagent systems: A survey from a machine learning perspective. *Autonomous Robots*, *8*(3), 345–383.
43. Takaoka, T. (1999). An O(1) time algorithm for generating multiset permutations. In *International symposium on algorithms and computation* (pp. 237–246). Berlin: Springer.
44. Thrall, R. M., & Lucas, W. F. (1963). N-person games in partition function form. *Naval Research Logistics Quarterly*, *10*(1), 281–298.
45. Ueda, S., Iwasaki, A., Conitzer, V., Ohta, N., Sakurai, Y., & Yokoo, M. (2018). Coalition structure generation in cooperative games with compact representations. *Autonomous Agents and Multi-Agent Systems*, *32*(4), 503–533.
46. Ueda, S., Iwasaki, A., Yokoo, M., Silaghi, M. C., Hirayama, K., & Matsui, T. (2010). Coalition structure generation based on distributed constraint optimization. *AAAI*, *10*, 197–203.
47. Ueda, S., Kitaki, M., Iwasaki, A., & Yokoo, M. (2011). Concise characteristic function representations in coalitional games based on agent types. In *Twenty-second international joint conference on artificial intelligence*.
48. Wester, F., & Vajlok, A. (2016). Paradox interactive announces grand successes for grand strategy titles. Retrieved February 25, 2020 from https://paradoxinteractive.com/en/paradox-interactive-announces-grand-successes-for-grand-strategy-titles/.
49. Williams, A. (2009). Loopless generation of multiset permutations using a constant number of variables by prefix shifts. In *Proceedings of the twentieth annual ACM-SIAM symposium on discrete algorithms* (pp. 987–996). Society for Industrial and Applied Mathematics.

50. Yamada, T., & Nasu, Y. (2000). Heuristic and exact algorithms for the simultaneous assignment problem. *European Journal of Operational Research*, *123*(3), 531–542.
51. Yamamoto, J., & Sycara, K. (2001). A stable and efficient buyer coalition formation scheme for e-marketplaces. In *Proceedings of the fifth international conference on autonomous agents* (pp. 576–583). ACM.
52. Yeh, D. Y. (1986). A dynamic programming approach to the complete set partitioning problem. *BIT Numerical Mathematics*, *26*(4), 467–474.
53. Zha, A., Nomoto, K., Ueda, S., Koshimura, M., Sakurai, Y., & Yokoo, M. (2017). Coalition structure generation for partition function games utilizing a concise graphical representation. In *International conference on principles and practice of multi-agent systems* (pp. 143–159). Berlin: Springer.
54. Zhang, Z., Song, L., Han, Z., & Saad, W. (2014). Coalitional games with overlapping coalitions for interference management in small cell networks. *IEEE Transactions on Wireless Communications*, *13*(5), 2659–2669.