An Anytime Algorithm for Simultaneous Coalition Structure Generation and Assignment

Fredrik Präntare and Fredrik Heintz

Linköping University
581 83 Linköping, Sweden
{firstname.lastname}@liu.se

Abstract. A fundamental problem in artificial intelligence is how to organize and coordinate agents to improve their performance and skills. In this paper, we consider simultaneously generating coalitions of agents and assigning the coalitions to independent tasks, and present an anytime algorithm for the *simultaneous coalition structure generation and assignment* problem. This optimization problem has many real-world applications, including forming goal-oriented teams of agents. To evaluate the algorithm's performance, we extend established methods for synthetic problem set generation, and benchmark the algorithm against *CPLEX* using randomized data sets of varying distribution and complexity.
We also apply the algorithm to solve the problem of assigning agents to regions in a major commercial strategy game, and show that the algorithm can be utilized in game-playing to coordinate smaller sets of agents in real-time.

Keywords: coalition structure generation · assignment problem.

1 Introduction

An important research challenge in the domain of 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. There are many approaches to this, including *task allocation* [7], *assignment* algorithms [4, 12, 15, 16, 29], *multi-agent reinforcement learning* [14], and organizational paradigms [10].

Coalition formation [11, 24] is a major coordination-paradigm and study of *coalitions* (flat goal-oriented organizations of agents) that has received extensive coverage in the literature over the past two decades [22]. This paradigm typically involves forming coalitions and allocating tasks, with applications in economics [30], planning [6], sensor fusion [5], wireless networks [9], and cell networks [32]. In cooperative games with transferable utility, coalition formation generally involves identifying *coalition structures* (sets of disjoint and exhaustive coalitions) that maximizes social welfare (utility) through *coalition structure generation* [20]. Coalition structure generation is NP-complete [23], and many algorithms have been presented that solves this problem, including algorithms based on dynamic programming [18, 31], tree-search [21], constraint optimization [27], and hybrid techniques [19]—each with their own strengths and weaknesses, making them suitable for solving different types of problems. Variations on the coalition structure generation problem also exist, e.g. with overlapping

coalitions—where agents have limited resources that they can use to partake in multiple coalitions at the same time [3, 8].

Coalition structure generation and assignment (of coalitional goals) are two processes for coordination that are often treated separately—including the majority of the previous examples. This is because traditional algorithms for coalition structure generation have no notion of independent coalitional goals, even though coalitions are often described as goal-oriented organizational structures. In instances for which coordination of multiple coalitions is important, this may generate suboptimal teams for achieving and accomplishing the tasks and goals at hand, and would typically require two different utility functions: one for deciding on which coalitions to form, and one for assigning them to tasks/goals. This is potentially disadvantageous, since it is often complicated to create good utility functions (or to generate realistic performance measures), and it is not necessarily a simple task to predict how the two utility 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.

To make this possible, and to address the aforementioned issues, we present an anytime algorithm that solves the *simultaneous coalition structure generation and assignment* problem by integrating coalition-to-task assignment into the formation of coalitions. We accomplish this by extending the coalition structure generation problem, and generating coalition structures for which each coalition is assigned to exactly one goal. Our algorithm can thus be used to create structured collaboration through explicit goal-orientation. Furthermore, our algorithm only requires one utility function, has the ability to prune large parts of the search space, can give worst-case guarantees on solutions, and always generates an optimal solution when run to completion.

To evaluate the algorithm's performance, we extend established methods for generating synthetic problem sets, and benchmark our algorithm against *CPLEX*—a commercial state-of-the-art optimization software. Our experiments are conducted to deduce whether the presented algorithm can handle difficult data sets efficiently. 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 our algorithm can be used to optimally solve a difficult game-playing problem in real-time. Apart from being applied to strategy games, our algorithm can potentially be used to solve many important real-world problems. It could, for example, be used to form optimal cross-functional 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 the algorithm is anytime and can return a valid solution even if it is interrupted prior to finishing a search, it can potentially be used in many real-world scenarios with real-time constraints as well, including time-critical systems for managing tactical decisions.

Note that this paper is the full-paper version of a previous extended abstract [17]. This version has been thoroughly revised and extended. The presented algorithm, its presentation, and the benchmarks herein, have all been significantly improved.

We begin by formalizing the problem that we solve in Section 2. Then, in Section 3, we describe our algorithm. In Section 4, we present the results from our experiments. Finally, in Section 5, we conclude with a summary.

2 Problem Formalization

The simultaneous coalition structure generation and assignment problem formalizes as:

Input: A set of agents $A = \{a_1, ..., a_n\}$, a list of tasks $T = \langle t_1, ..., t_m \rangle$, and the value $\boldsymbol{v}(C, t) \mapsto \mathbb{R}$ for assigning any coalition $C \subseteq A$ to any task $t \in T$.

Output: A list of coalitions $\langle C_1, ..., C_m \rangle$ that maximizes $\sum_{i=1}^{m} \boldsymbol{v}(C_i, t_i)$, where $C_i \subseteq A$ for $i = 1, ..., m$, $C_i \cap C_j = \emptyset$ for all $i \neq j$, and $\bigcup_{i=1}^{m} C_i = A$.

Note that we use the sum $\boldsymbol{V}(S) = \sum_{i=1}^{m} \boldsymbol{v}(C_i, t_i)$ to denote the value of a solution $S = \langle C_1, ..., C_m \rangle$ throughout this report. We also use the terms *agent* and *task* as abstractions (they can be substituted for any type of entities, e.g. resources, regions), and we use the conventions $n = |A|$ and $m = |T|$.

Now, with this in mind, and given the aforementioned input, we can also formalize this problem using a *binary integer programming* model:

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

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 1*). 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 1. *Binary coalition-encoding. Given a set of agents $A = \{a_1, ..., a_n\}$, and the non-negative integer $j < 2^n$ on binary form $j = b_1 2^0 + b_2 2^1 + ... + b_n 2^{(n-1)}$ 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 $k = 1, ..., n$. For example, if the coalition C^j has a binary coalition-encoding given by j over $\{a_1, ..., a_n\}$, 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 $j = 8 = 1000_2$.*

3 Algorithm Description

To solve this optimization problem, we propose an anytime search algorithm that utilizes branch-and-bound and a search space representation based on multiset permutations of integer partitions. By doing so, our algorithm always generates optimal solutions when run to exhaustion, and solutions with worst-case guarantees when interrupted prior to finishing a search. The algorithm consists of the following major steps:

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

These steps are described in the following subsections.

3.1 Partitioning of 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* (see *Definition 2*). In this representation, a list of non-negative integers $\langle p_1, ..., p_m \rangle$ represents all solutions $\langle C_1, ..., C_m \rangle$ with $|C_i| = p_i$ for $i = 1, ..., m$. Note that this is, technically speaking, a *refinement* (or an extension) of Rahwan, Ramchurn, Jennings and Giovannucci's search space representation for conventional coalition structure generation [21].

Definition 2. *Integer partition. An integer partition of $y \in \mathbb{N}$ is a multiset of positive integers $\{x_1, ..., x_k\}$ 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 $1 + 2 + 12 + 15 = 30$.

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. For example, 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. [1, 25]. 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 $\langle p_1, ..., p_m \rangle \in M_3$ represents the partition (subspace) that contains all solutions $\langle C_1, ..., C_m \rangle$ with $|C_i| = p_i$ and $C_i \subseteq A$ for $i = 1, ..., m$. 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 $O(1)$ per permutation, e.g. [26, 28].

The reason that partitions represented by the multiset permutations in M_3 cover the whole search space, is the fact that every coalition structure that consists of k agents can be mapped to exactly one of the integer partitions of k (see [21] 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 (i.e. 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.

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

3.2 Calculation of the Bounds for Partitions

To establish bounds for partitions (subspaces), so that the algorithm can make more informed decisions during search, let $\mathbb{C}_p \coloneqq \{X \subseteq A : |X| = p\}$, i.e. the set of all p-sized coalitions, and define:

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

We can now establish an upper and a lower bound for the value of the best possible solution in $\mathbb{S}_P$ as the sums $\sum_{i=1}^{m} \boldsymbol{M}(p_i, t_i)$ and $\sum_{i=1}^{m} \boldsymbol{Avg}(p_i, t_i)$, respectively. For proofs, see *Theorem 1* and *Theorem 2*. Note that this lower bound, that we base on the average values of coalitional values, is better than the one you would achieve by using the more straight-forward $\min \{\boldsymbol{v}(C, t) : C \in \mathbb{C}_p\}$. A proof for this follows directly from the definition of $\boldsymbol{Avg}(p, t)$.

Theorem 1. *$u_P = \sum_{i=1}^{m} \boldsymbol{M}(p_i, t_i)$ is an upper bound for the value of the best possible solution in the subspace $\mathbb{S}_P$ that is represented by $P = \langle p_1, ..., p_m \rangle$. In other words, $\sum_{i=1}^{m} \boldsymbol{v}(C_i, t_i) \leq u_P$ for all $\langle C_1, ..., C_m \rangle \in \mathbb{S}_P$.*

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

$$\boldsymbol{M}(p_i, t_i) = \boldsymbol{M}(|C_i|, t_i) \tag{1}$$

Since $\boldsymbol{v}(C_i, t_i) \leq \boldsymbol{M}(|C_i|, t_i)$ for $i = 1, ..., m$, 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 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 $\mathbb{S}_P$ that is represented by $P = \langle p_1, ..., p_m \rangle$. In other words:*

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

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

$$\overline{y_1, ..., y_k} \leq \max\{y_1, ..., y_k\} \tag{2}$$

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

$$|\mathbb{S}_P| = X_i \cdot |\mathbb{C}_{p_i}| \tag{3}$$

for some integer $X_i \in \mathbb{N}$ for $i = 1, ..., m$. 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 $t_1, ..., t_{i-1}, t_{i+1}, ..., t_m$. 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 $\mathbb{V}_P \coloneqq \{\sum_{i=1}^{m} \boldsymbol{v}(C_i, t_i) : \langle C_1, ..., C_m \rangle \in \mathbb{S}_P\}$, i.e. the set of the values of the solutions in $\mathbb{S}_P$, as follows:

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

From this and (2), we conclude:

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

□

Since the performance measure for each coalition-to-task assignment is assumed to be known, the bounds can, in practice, be calculated without having to enumerate or generate any solution. For instance, by enumerating all coalition-to-task values, the lower bounds can be calculated using a moving average.

3.3 Searching for Solutions using Branch-and-Bound

We search for solutions by searching one partition (subspace) at a time, and discard partitions that only contain suboptimal solutions (i.e. a partition is discarded when its upper bound is lower than the value of the best solution found so far). 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) partitions, and thus 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 partitions, that ultimately makes it possible to find better solutions more quickly, and use heuristics to guide search.

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 coalition-to-task utility function $\boldsymbol{v} \mapsto \mathbb{R}$, and we instead have to take advantage of domain-independent information (e.g. partitions and their bounds). It is possible to utilize 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 partitions that we use is defined as follows:

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

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

We use *Algorithm 1* and *Algorithm 2* to search a subspace $\mathbb{S}_P$ (represented by the multiset permutation $P \in M_3$) for $\arg\max_{S \in \mathbb{S}_P} \boldsymbol{V}(S)$. If interrupted before termination, these algorithms return the best feasible solution found so far, denoted S'. Note that we use a notation based on brackets to indicate an element at a specific position of a list or a vector. For example, the notation $S[j]$ corresponds to the coalition $C_j \in S$, and the notation $A[i]$ corresponds to the agent $a_i \in A$.

Algorithm 1 : `InitAndStartSearchSubspace`(A, T, P, S', u_P)
Initializes and starts the search procedure defined in *Algorithm 2*, thus searching $\mathbb{S}_P$.

1: **if** S' is uninitialized **then**
2: $\quad S' \leftarrow \emptyset_{|T|}$ ▷ S' is initialized to a list of $m = |T|$ empty coalitions.
3: **end if**
4: **return** `SearchSubspace`(A, T, P, u_P, 1, $\emptyset_{|T|}$, 0.0, S')

Algorithm 2 : `SearchSubspace`(A, T, P, u, i, $\overrightarrow{S}$, $\overrightarrow{v}$, S')
Recursively searches the subspace $\mathbb{S}_P$ represented by the multiset permutation P.

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.
      $\overrightarrow{v} \leftarrow \overrightarrow{v} + \boldsymbol{v}(\overrightarrow{S}[j], T[j])$
      $u \leftarrow u - \boldsymbol{M}(P[j], T[j])$
    end if
    if $S' = \emptyset_{|T|}$ **or** $\overrightarrow{v} + u > \boldsymbol{V}(S')$ **then** ▷ Check if a better solution is possible.
      $S'' \leftarrow$ `SearchSubspace`(A, T, P, u, $i+1$, $\overrightarrow{S}$, $\overrightarrow{v}$, S')
      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.
      $\overrightarrow{v} \leftarrow \overrightarrow{v} - \boldsymbol{v}(\overrightarrow{S}[j], T[j])$
      $u \leftarrow u + \boldsymbol{M}(P[j], T[j])$
    end if
    $\overrightarrow{S}[j] \leftarrow \overrightarrow{S}[j] \setminus \{A[i]\}$ ▷ Remove agent $A[i]$ from the coalition $\overrightarrow{S}[j]$.
  end if
end for
return S'

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 Subsection 3.1) 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:

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

where $Q_1 \prec Q_2$ denotes that the solutions represented by the group of multiset permutations consisting of the members $q_1, ..., q_m$ is searched before the solutions represented by the group of multiset permutations consisting of the members $p_1, ..., p_m$,

where $\{q_1, ..., q_m\} = Q_1$ and $\{p_1, ..., p_m\} = Q_2$, with $Q_1 \in M_2$ and $Q_2 \in M_2$. w_Q and f_Q are defined (similarly to the partition bounds), for all $Q \in M_2$, as follows:

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

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

4 Evaluation and Results

A common approach to evaluating optimization algorithms is to use standardized problem instances for benchmarking. To our knowledge, no such instances exist for the simultaneous coalition structure generation and assignment 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 coalition structure generation algorithms. The extended methods are then used to generate difficult problem sets of varying distribution and complexity that we use to benchmark our algorithm against *IBM ILOG CPLEX Optimization Studio*—a commercial state-of-the-art optimization software.

Larson and Sandholm [13] provided standardized synthetic problem instances for the coalition structure generation problem by using normal and uniform probability distributions to generate randomized coalitional values. Following Rahwan et al. [21], we denote these distributions *NPD* and *UPD*, 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*, a distribution that was proposed by Rahwan et al. [21] for benchmarking coalition structure generation algorithms. Our extensions of these probability distributions, to our task-dependent domain, are defined as follows:

- **UPD:** $\boldsymbol{v}(C, t) \sim |C| \cdot \boldsymbol{\mathcal{U}}(0, 1)$
- **NPD:** $\boldsymbol{v}(C, t) \sim |C| \cdot \boldsymbol{\mathcal{N}}(1, 0.1^2)$
- **NDCS:** $\boldsymbol{v}(C, t) \sim \boldsymbol{\mathcal{N}}(|C|, |C|)$

where $\boldsymbol{\mathcal{N}}(\sigma, \mu)$ and $\boldsymbol{\mathcal{U}}(a, b)$ are the normal and uniform distributions, respectively, given a coalition $C \subseteq A$ and a task $t \in T$.

The results of our experiments that were based on these distributions, and from applying the algorithm to a commercial strategy game, are presented in Subsection 4.2, and Subsection 4.3, respectively.

4.1 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 NDCS and NPD) and `uniform_real_distribution` (for UPD) from the *C++ Standard Library*. All tests were conducted using *Windows 10* (x64), an *Intel* 7700K 4200MHz CPU, and 16GB of 3000MHz DDR4 memory.

4.2 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 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.

Note that we, throughout this section, use the abbreviation *MP* (short for multiset permutation) to denote our algorithm.

The execution time to find an optimal solution for 8 tasks is plotted using a logarithmic scale in *Figure 1*, in which we benchmark MP against CPLEX with different numbers of agents, using problem sets generated with UPD, NPD and NDCS.

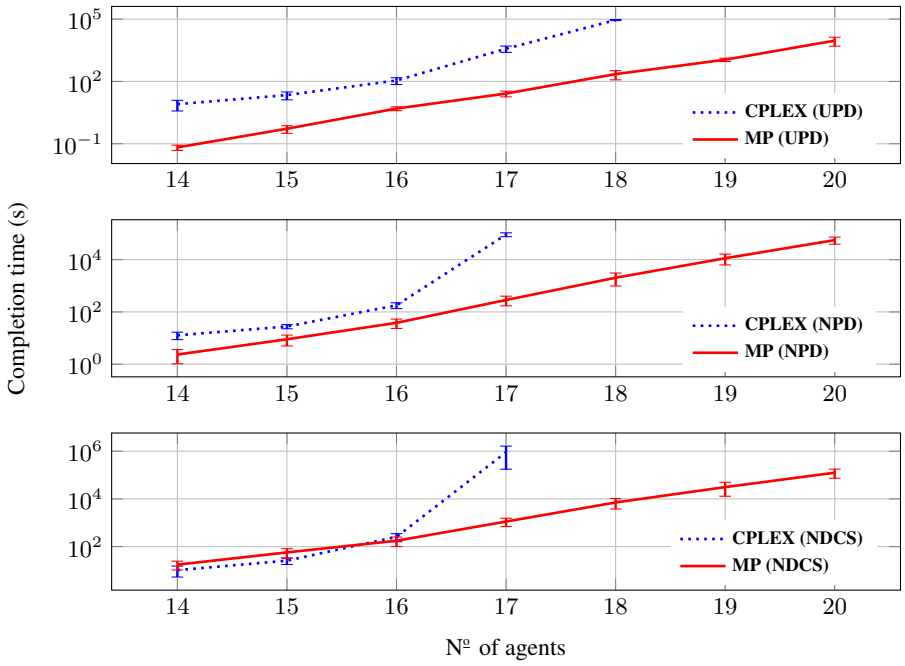

Fig. 1: Execution time for solving synthetic problems with 8 tasks. The values for the coalition-to-task assignments were generated using UPD (top), NPD (middle) and NDCS (bottom).

The results in these graphs show that our algorithm (MP) is considerably faster (by many orders of magnitude) than CPLEX for almost all distributions and problem sets. When there are more than 16 agents, CPLEX has difficulty finding optimal solutions within a reasonable time, especially for NPD and NDCS, as can be seen in the graphs

above. MP, however, manages to find optimal solutions for all problems (at least up to 20 agents) within a reasonable time. In these logarithmic graphs, MP is clearly linear, while CPLEX is not. Furthermore, MP is clearly sensitive to the distribution of utility values. A potential reason for this is that the efficiency of MP depends on its ability to discard partitions. Naturally, this ability is affected by the distribution of the utility values in the problem being solved.

We plot the execution time to find an optimal solution for 16 agents in *Figure 2*, and instead look at how the number of tasks (2 to 12) affect performance. We used 16 agents in these tests, since CPLEX didn't manage to find optimal solutions within a reasonable time for problems with more agents.

As can be seen in *Figure 2*: CPLEX demonstrates inconsistent behaviour for problems when varying the number of tasks. This includes increased execution time in easier problems with few (2 to 4) tasks. With this in mind, our algorithm is considerably faster for most problem sets, except for those with many (8 to 12) tasks generated by NDCS. A reason could be that, when we increase the number of tasks, MP consequently generates larger integer partitions. As a consequence, the blocks generated by MP also becomes much larger, since the number of possible multiset permutations grows exponentially in the number of tasks. These multiset permutations take a considerable time to generate, even if (or when) MP discards the entire partitions that they represent.

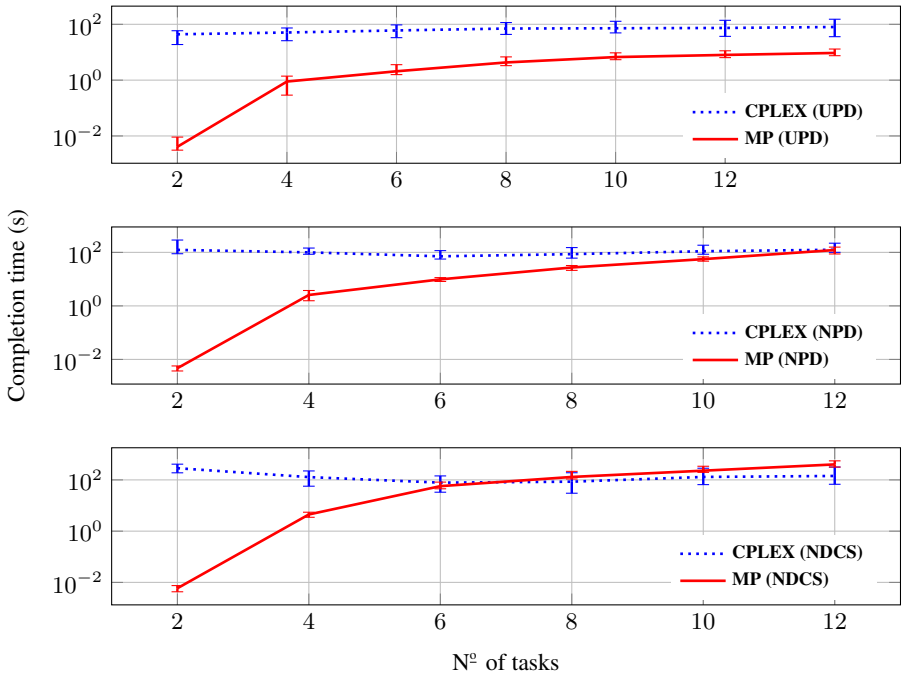

Fig. 2: Execution time to solve synthetic problems with 16 agents. Generated using UPD (top), NPD (middle) and NDCS (bottom).

In *Figure 3*, we look at the quality of the anytime solutions generated by MP. We used 12 agents and 8 tasks for this purpose, and interrupted the algorithm during search by only allowing it to evaluate a fixed number of solutions. The total number of possible solutions for 12 agents and 8 tasks is $8^{12} \approx 7 \times 10^{10}$. We show the value $V_{anytime}$ of the best solution that our algorithm has found after a number of evaluated solutions, divided by the value V_{opt} of an optimal solution, on the y-axis. In this experiment, all utility values were generated using NDCS.

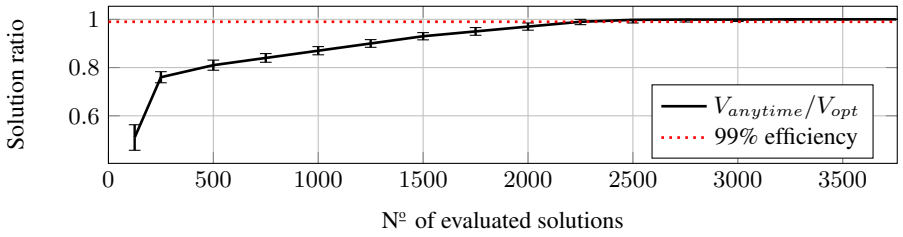

Fig. 3: The quality of anytime solutions when our algorithm is interrupted prior to finishing a complete search for problem sets based on NDCS with 12 agents, 8 tasks and $8^{12} \approx 7 \times 10^{10}$ possible solutions.

In this experiment, MP's execution time is roughly the same for evaluating any subsequent 1000 solutions, and was measured to 0.34 ± 0.27 seconds. Also, finding an optimal solution took 3.07 ± 1.75 seconds. This means that, after roughly 0.34 seconds, MP manages to find close to 90% efficient solutions, and after approximately 1 second, MP often manages to find 99% efficient solutions. For this execution time, CPLEX fails to find any solution at all.

4.3 Applying the Algorithm to Europa Universalis 4

To empirically show that the algorithm can be used to coordinate agents in a real-world scenario, we applied it to improve the coordination skills of computer-based players in *Europa Universalis 4* (EU4)—a very complex strategy game, in which agents are required to act and reason in real-time. This game is very popular, with many thousands of active players, and was developed and released commercially by the Swedish game development company *Paradox Development Studio*. Note that there are many reasons to why strategy games are ideal for empirically evaluating and testing AI algorithms, and other authors have discussed these reasons in earlier publications, see e.g. [2].

In a session of 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, coordinate armies, handle diplomacy, and wage war. To play this dynamic (and partially observable) game successfully, the players have to continuously solve simultaneous coalition structure generation and assignment problems by assigning their armies to different regions. Previously, the computer-based players in EU4 used an *ad hoc* anytime search algorithm to do so—a highly spe-

cialized algorithm designed for the context of EU4, inherently based on expert knowledge and heuristics.

In collaboration with the game's developers, we benchmarked our algorithm against theirs. To do so, we used the same problem sets (generated by the game) and utility function (based on expert knowledge and defined by the developers) for both algorithms. 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 $m \in [1, 35]$. However, there were at most $30^8 \approx 6.56 \cdot 10^{11}$ solutions for the largest problem sets that were generated by the game (i.e. problems with $n = 8$ armies and $m = 30$ regions).

The results from these experiments show that applying the algorithm to EU4 was a great success in terms of improving the computer-based players' performance (i.e. an increase of solution quality) and computational efficiency (i.e. 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 $1/20 \approx 0.05$ seconds)—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, 13922 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 functions that are used in EU4 were to change (e.g. due to environment alterations), the *ad hoc* algorithm might have to be altered. This is not the case for our algorithm, since it does not make any assumptions on coalitions' utility functions, or the game's rules. Therefore, our algorithm is potentially cheaper and easier to maintain. Finally, note that EU4's environment is not superadditive: Adding an agent to a coalition does not necessarily increase its value, since the regions' have supply-based limitations that can reduce larger coalitions' values.

5 Conclusions

In this paper, we presented an anytime algorithm that solves the simultaneous coalition structure generation and assignment problem by integrating assignment into the formation of coalitions. We are, to the best of our knowledge, the first to study and solve this 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 clearly demonstrate that our algorithm is superior to CPLEX in solving synthetic instances of the simultaneous coalition structure generation and assignment problem. 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. This is beneficial in many real-time systems (e.g. real-world multi-agent systems), in

which feasible solutions must be available fast, but optimality is not necessarily required. Apart from these properties, our algorithm is able to give worst-case guarantees on solutions.

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 previous algorithms.

For future work, it would be interesting to investigate other approaches to solving this problem, including dynamic programming and greedy algorithms. We also intend to analyze the algorithm's parallel computing performance, and look at the problem of simultaneous coalition structure generation and assignment with overlapping coalitions. Finally, it would be interesting to see if machine learning could be applied to solve large-scale simultaneous coalition structure generation and assignment problems, or increase our algorithm's performance by improving its search heuristics.

6 Acknowledgements

This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

Bibliography

[1] Andrews, G., Eriksson, K.: Integer partitions. Cambridge University Press (2004)

[2] Buro, M.: Real-time strategy games: A new ai research challenge. In: International Joint Conference on Artificial Intelligence. pp. 1534–1535 (2003)

[3] Chalkiadakis, G., Elkind, E., Markakis, E., Polukarov, M., Jennings, N.R.: Cooperative games with overlapping coalitions. Journal of Artificial Intelligence Research **39**, 179–216 (2010)

[4] Chu, P.C., Beasley, J.E.: A genetic algorithm for the generalised assignment problem. Computers & Operations Research **24**(1), 17–23 (1997)

[5] Dang, V.D., Dash, R.K., Rogers, A., Jennings, N.R.: Overlapping coalition formation for efficient data fusion in multi-sensor networks. In: AAAI. vol. 6, pp. 635–640 (2006)

[6] Dukeman, A., Adams, J.A.: Hybrid mission planning with coalition formation. Autonomous Agents and Multi-Agent Systems **31**(6), 1424–1466 (2017)

[7] Gerkey, B.P., Matarić, M.J.: A formal analysis and taxonomy of task allocation in multi-robot systems. The International Journal of Robotics Research **23**(9), 939–954 (2004)

[8] Habib, F.R., Polukarov, M., Gerding, E.H.: Optimising social welfare in multi-resource threshold task games. In: International Conference on Principles and Practice of Multi-Agent Systems. pp. 110–126. Springer (2017)

[9] Han, Z., Poor, H.V.: 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 (2009)

[10] Horling, B., Lesser, V.: A survey of multi-agent organizational paradigms. The Knowledge Engineering Review **19**(4), 281–316 (2004)

[11] Kelso Jr, A.S., Crawford, V.P.: Job matching, coalition formation, and gross substitutes. Econometrica: Journal of the Econometric Society pp. 1483–1504 (1982)

[12] Kuhn, H.W.: The hungarian method for the assignment problem. Naval Research Logistics (NRL) **2**(1-2), 83–97 (1955)

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

[14] Leibo, J.Z., Zambaldi, V., Lanctot, M., Marecki, J., Graepel, T.: Multi-agent reinforcement learning in sequential social dilemmas. In: Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems. pp. 464–473. International Foundation for Autonomous Agents and Multiagent Systems (2017)

[15] Munkres, J.: Algorithms for the assignment and transportation problems. Journal of the society for industrial and applied mathematics **5**(1), 32–38 (1957)

[16] Pentico, D.W.: Assignment problems: A golden anniversary survey. European Journal of Operational Research **176**(2), 774–793 (2007)

[17] Präntare, F., Ragnemalm, I., Heintz, F.: An algorithm for simultaneous coalition structure generation and task assignment. In: International Conference on Principles and Practice of Multi-Agent Systems. pp. 514–522. Springer (2017)

[18] Rahwan, T., Jennings, N.R.: An improved dynamic programming algorithm for coalition structure generation. In: Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems-Volume 3. pp. 1417–1420. International Foundation for Autonomous Agents and Multiagent Systems (2008)
[19] Rahwan, T., Michalak, T.P., Jennings, N.R.: A hybrid algorithm for coalition structure generation. In: AAAI. pp. 1443–1449 (2012)
[20] Rahwan, T., Michalak, T.P., Wooldridge, M., Jennings, N.R.: Coalition structure generation: A survey. Artificial Intelligence **229**, 139–174 (2015)
[21] Rahwan, T., Ramchurn, S.D., Jennings, N.R., Giovannucci, A.: An anytime algorithm for optimal coalition structure generation. Journal of Artificial Intelligence Research **34**, 521–567 (2009)
[22] Ray, D., Vohra, R.: Coalition formation. In: Handbook of game theory with economic applications, vol. 4, pp. 239–326. Elsevier (2015)
[23] Sandholm, T., Larson, K., Andersson, M., Shehory, O., Tohmé, F.: Coalition structure generation with worst case guarantees. Artificial Intelligence **111**(1-2), 209–238 (1999)
[24] Shehory, O., Kraus, S.: Methods for task allocation via agent coalition formation. Artificial intelligence **101**(1-2), 165–200 (1998)
[25] Stojmenović, I., Zoghbi, A.: Fast algorithms for genegrating integer partitions. International Journal of Computer Mathematics **70**(2), 319–332 (1998)
[26] Takaoka, T.: An o (1) time algorithm for generating multiset permutations. In: International Symposium on Algorithms and Computation. pp. 237–246. Springer (1999)
[27] Ueda, S., Iwasaki, A., Yokoo, M., Silaghi, M.C., Hirayama, K., Matsui, T.: Coalition structure generation based on distributed constraint optimization. In: AAAI. vol. 10, pp. 197–203 (2010)
[28] Williams, A.: 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 (2009)
[29] Yamada, T., Nasu, Y.: Heuristic and exact algorithms for the simultaneous assignment problem. European Journal of Operational Research **123**(3), 531–542 (2000)
[30] Yamamoto, J., Sycara, K.: 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 (2001)
[31] Yeh, D.Y.: A dynamic programming approach to the complete set partitioning problem. BIT Numerical Mathematics **26**(4), 467–474 (1986)
[32] Zhang, Z., Song, L., Han, Z., Saad, W.: Coalitional games with overlapping coalitions for interference management in small cell networks. IEEE Transactions on Wireless Communications **13**(5), 2659–2669 (2014)