Anytime Heuristic and Monte Carlo Methods for Large-Scale Simultaneous Coalition Structure Generation and Assignment

Fredrik Präntare, Herman Appelgren and Fredrik Heintz
Linköping University
581 83 Linköping, Sweden
{fredrik.prantare, herman.appelgren, fredrik.heintz}@liu.se

Abstract

Optimal simultaneous coalition structure generation and assignment is computationally hard. The state-of-the-art can only compute solutions to problems with severely limited input sizes, and no effective approximation algorithms that are guaranteed to yield high-quality solutions are expected to exist. Real-world optimization problems, however, are often characterized by large-scale inputs and the need for generating feasible solutions of high quality in limited time. In light of this, and to make it possible to generate better feasible solutions for difficult large-scale problems efficiently, we present and benchmark several different anytime algorithms that use general-purpose heuristics and Monte Carlo techniques to guide search. We evaluate our methods using synthetic problem sets of varying distribution and complexity. Our results show that the presented algorithms are superior to previous methods at quickly generating near-optimal solutions for small-scale problems, and greatly superior for efficiently finding high-quality solutions for large-scale problems. For example, for problems with a thousand agents and values generated with a uniform distribution, our best approach generates solutions 99.5% of the expected optimal within seconds. For these problems, the state-of-the-art solvers fail to find any feasible solutions at all.

1 Introduction

The *coalition structure generation* (CSG) problem is a major algorithmic challenge in cooperative game theory and multi-agent systems. It is central to coalition formation (Osborne and Rubinstein 1994), and it has been studied extensively in many contexts (Rahwan et al. 2015).

In other research fields (e.g., operations research), *assignment* algorithms have been used to allocate tasks and coordinate entities. These algorithms seek to find an optimal matching between the elements of sets, with the objective to maximize the aggregated value of assignments. There are numerous variations on this problem. See (Pentico 2007) for an extensive summary.

From an algorithmic perspective, CSG and assignment are two major processes for coordination that are typically treated separately: CSG as a *partitioning* problem, and assignment as a *matching* problem. (Präntare and Heintz 2020) showed that this separation may potentially lead to arbitrarily bad coalition structures (or an unmotivated exponential increase in computational complexity) when distinct coalitional goals are involved—for example when each coalition that is generated has a specific task that it needs to perform.

To remedy this, they presented the *simultaneous coalition structure generation and assignment* (SCSGA) problem—a difficult combinatorial optimization problem that generalizes both the *linear assignment problem* (Kuhn 1955), and the CSG problem for *characteristic function games* (Sandholm et al. 1999; Rahwan et al. 2015). This is a central problem in both artificial intelligence, operations research, and algorithmic game theory; with applications in optimal task/resource allocation (Präntare 2017), winner determination for combinatorial auctions (Sandholm et al. 2002), and team/coalition formation (Präntare and Heintz 2020). More formally, the *SCSGA problem* is defined as follows:

Input: a tuple $\langle A, T, \boldsymbol{v} \rangle$, where $A = \{a_1, ..., a_n\}$ is a set of agents, $T = \langle t_1, ..., t_m \rangle$ is a tuple of alternatives (e.g., tasks), and $\boldsymbol{v} : 2^A \times T \mapsto \mathbb{R}$ is a function that maps a value to every possible pairing of a coalition $C \subseteq A$ to an alternative $t \in T$.

Output: an *ordered coalition structure* (Definition 1) $\langle C_1, ..., C_m \rangle$ over A that maximizes $\sum_{i=1}^{m} \boldsymbol{v}(C_i, t_i)$.

Definition 1. The tuple $\langle C_1, ..., C_m \rangle$ is an *ordered coalition structure* over A if $C_i \cap C_j = \emptyset$ for all $i \neq j$, and $\bigcup_{i=1}^{m} C_i = A$. For example, $\langle \{a_1, a_3\}, \emptyset, \{a_2, a_4\} \rangle$ is an ordered coalition structure over the agents $\{a_1, a_2, a_3, a_4\}$. We omit the notion "over A" when it improves readability.

The only algorithm that has been developed for this problem is an optimal anytime branch-and-bound algorithm (denoted *MP*) by (Präntare and Heintz 2020) that uses a search space representation based on multiset permutations of integer partitions to prune large portions of the search space. Even though this algorithm performs well in practice for limited input sizes, and greatly outperforms the industry-grade solver *CPLEX*, there is no proven guarantee that it can find an optimum without evaluating all m^n possible ordered coalition structures. Moreover, this algorithm is impractical for solving problems with more than just a few agents. Just for preprocessing, it requires at least e.g., $m2^n = 10 \times 2^{30} \approx 10^{10}$ operations for $n = 30$ agents and $m = 10$ alternatives.

Furthermore, in general, we do not expect to find an efficient SCSGA approximation algorithm, and we need to enumerate all the $m2^n$ possible values of the value function to make any worst-case guarantee on the quality of a *feasible solution*—i.e., a possibly suboptimal ordered coalition structure. In other words, to find a feasible solution guaranteed to be within any bound from optimum, we first need to scan the whole input and evaluate every possible pairing of a coalition $C \subseteq A$ to an alternative $t \in T$. It would thus be interesting to investigate if, when and how low-complexity algorithms can generate feasible solutions of high-enough quality for problems with large-scale inputs and limited computation budgets.

In light of this, and to make it possible to generate better coalition structures for difficult large-scale problems, we develop, benchmark, and present several different centralized, anytime algorithms that use general-purpose heuristics and Monte Carlo techniques to guide search. To summarize, our main contributions are:

- We give a poly-time reduction from CSG to SCSGA. As a corollary, SCSGA is, as expected, NP-complete.
- We develop, present and benchmark five different heuristic and Monte Carlo methods for centralized SCSGA: 1) a *Monte Carlo tree search* variant; 2) a *steepest-ascent hill climb* method that we use to improve solution-quality; 3) a *random-restart greedy* algorithm; 4) a *simulated annealing* method; and 5) a *hybrid* approach based on local search. Most of our algorithms are based on utilizing permutations of agent sets to efficiently generate and evaluate different feasible solutions. As a by-product, we develop a framework for designing and implementing efficient random-restart CSG and SCSGA algorithms.
- We empirically show that our algorithms outperform current state-of-the-art for both small and large input-sizes when generating anytime, feasible solutions.
- We establish a baseline for future large-scale SCSGA research by providing initial theory and empirical data. In addition to this, we develop a new approach to estimating a solution's quality in two standard benchmarks, and present two new difficult synthetic problem sets.

This paper is structured as follows. We begin by presenting related heuristic algorithms and similar work in Section 2. Then, in Section 3, we define important concepts and prove the problem's intractability. In Section 4, we describe our algorithms. In Section 5, we present our experiments. Finally, in Section 6, we conclude with a summary.

2 Related Work

One of the first centralized, metaheuristic CSG algorithms was devised by (Sen and Dutta 2000). Their algorithm is a genetic algorithm that starts by creating a randomly generated initial set of coalition structures called the *population pool*. The algorithm then continues to update the population pool by repeatedly recombining its coalition structures to generate new ones. Later, (Keinänen 2009) used the well-known *simulated annealing* method to generate feasible solutions. (Di Mauro et al. 2010) proposed an algorithm that works by first using a greedy search strategy to generate an initial coalition structure, and then using local search to gradually find better ones. More recently, (Yeh and Sugawara 2016) devised an *ant colony optimization* heuristic, (Farinelli et al. 2017) presented an algorithm inspired by data clustering methods, and (Wu and Ramchurn 2020) developed a Monte Carlo tree search CSG algorithm.

Common for these methods is that they are specifically designed for solving CSG problems without alternatives. This has the implication that they: 1) allow coalition structures of any size (i.e., not only size-m, where m is the number of alternatives); 2) do not take the coalitions' order within the coalition structures into consideration; and 3) do not allow empty coalitions in solutions. Having these three properties arguably render them unsuitable for SCSGA unless e.g., they are redesigned from the ground up.

3 Basic Concepts and Complexity

For the remainder of this paper, we use the terms *solution* and *ordered coalition structure* interchangeably. We use $\boldsymbol{V}(S) = \sum_{i=1}^{m} \boldsymbol{v}(C_i, t_i)$ to denote the value of a solution $S = \langle C_1, ..., C_m \rangle$, and the conventions $n = |A|$ and $m = |T|$ when it improves readability. We define a *SCSGA problem instance* by its input-tuple $\langle A, T, \boldsymbol{v} \rangle$. Finally, recall that CSG for *characteristic function games* is formalized as:

Input: a tuple $\langle A, \boldsymbol{u} \rangle$, where $A = \{a_1, ..., a_n\}$ is a set of agents, and $\boldsymbol{u} : 2^A \mapsto \mathbb{R}$ is a function that maps a value to every coalition $C \subseteq A$. $\boldsymbol{u}(\emptyset) = 0$ is assumed.

Output: a *coalition structure* (see Definition 2) $\{C_1, ..., C_m\}$ over A that maximizes $\sum_{i=1}^{m} \boldsymbol{u}(C_i)$.

Definition 2. $CS = \{C_1, ..., C_m\}$ is a *coalition structure* over the set A if $C_i \cap C_j = \emptyset$ for all $i \neq j$, and $\bigcup_{i=1}^{m} C_i = A$. For example, $\{\{a_1, a_3\}, \{a_2\}\}$ and $\{\{a_1\}, \{a_2\}, \{a_3\}\}$ are coalition structures over $\{a_1, a_2, a_3\}$. Moreover, we often omit "over A" for brevity.

Similarly as for SCSGA, we define a *CSG problem instance* by its input-tuple $\langle A, \boldsymbol{u} \rangle$, and $\boldsymbol{U}(CS) = \sum_{C \in CS} \boldsymbol{u}(C)$ is used to denote the value of a coalition structure CS.

3.1 Complexity

We prove that CSG is reducible to SCSGA in Theorem 1. SCSGA's NP-completeness follows from this in Theorem 2.

Definition 3. A *SCSGA-corresponding problem instance* to a CSG problem instance $\langle A, \boldsymbol{u} \rangle$ is a SCSGA problem instance $\langle A, T, \boldsymbol{v} \rangle$ with $|T| = |A|$ (i.e., $m = n$) and $\boldsymbol{v}(C, t) = \boldsymbol{u}(C)$ for all $t \in T$ and $C \subseteq A$.

Lemma 1. *If $\mathcal{P}$ is a CSG problem instance, and $\mathcal{Q}$ is a* SCSGA-corresponding problem instance *(Definition 3) to $\mathcal{P}$; then, if the solution $\langle C_1, ..., C_m \rangle$ is optimal to $\mathcal{Q}$, it holds that the coalition structure $\{C_1, ..., C_m\}$ is optimal to $\mathcal{P}$.*

Proof. By contradiction. Without loss of generality, assume $S^* = \langle C_1, ..., C_m \rangle$ is optimal to $\mathcal{Q} = \langle A, T, \boldsymbol{v} \rangle$, and that the coalition structure $CS' = \{C_1, ..., C_m\}$ is suboptimal to

$\mathcal{P} = \langle A, \boldsymbol{u} \rangle$. Then, by definition of optimality, there exists a coalition structure $CS^* = \{K_1, ..., K_{|CS^*|}\}$ over A with:

$$\boldsymbol{U}(CS^*) = \sum_{i=1}^{|CS^*|} \boldsymbol{u}(K_i) > \sum_{i=1}^{m} \boldsymbol{u}(C_i) = \boldsymbol{U}(CS'). \quad (1)$$

Furthermore, from Definition 3, we have:

$$\boldsymbol{U}(CS') = \sum_{i=1}^{m} \boldsymbol{u}(C_i) = \sum_{i=1}^{m} \boldsymbol{v}(C_i, t_i) = \boldsymbol{V}(S^*).$$

This together with (1) gives:

$$\sum_{i=1}^{|CS^*|} \boldsymbol{u}(K_i) > \boldsymbol{V}(S^*). \quad (2)$$

Now, if $|CS^*| < |A|$, let $K_j = \emptyset$ for $|CS^*| < j \leq |A|$. Then, the ordered coalition structure $S' = \langle K_1, ..., K_{|A|} \rangle$ is a valid solution to Q. Note that this is also a valid solution if $|CS^*| = |A|$, and that $|CS^*| \leq |A|$ always holds, since CS^* is a set of disjoint subsets of A. Consequently, we have:

$$\boldsymbol{V}(S') = \sum_{i=1}^{|CS^*|} \boldsymbol{v}(K_i, t_i) + \sum_{j=|CS^*|+1}^{|A|} \boldsymbol{v}(\emptyset, t_i).$$

This with Definition 3 (and $\boldsymbol{u}(\emptyset) = 0$ by definition) gives:

$$\boldsymbol{V}(S') = \sum_{i=1}^{|CS^*|} \boldsymbol{u}(K_i) + \sum_{j=|CS^*|+1}^{|A|} \boldsymbol{u}(\emptyset) = \sum_{i=1}^{|CS^*|} \boldsymbol{u}(K_i).$$

From this and (2), it follows that:

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

This is a contradiction, since the solution S' cannot have a higher value than the optimal solution S^*. □

Theorem 1. *An algorithm that can solve the SCSGA problem can also solve the CSG problem (with only a polynomial increase in computation time after its SCSGA-corresponding problem instance has been solved).*

Proof. This follows directly from the proof of Lemma 1, since it provides a linear-time procedure to convert a CSG problem instance P to an equivalent SCSGA problem instance Q, and another linear-time procedure to convert Q's optimal solution to a coalition structure optimal to P.

(Note that a similar reduction can be made in the opposite direction. Here's the main idea: Let the alternatives be represented as agents in the CSG problem, and invalid coalitions assigned value $-\infty$. It is then easy to show that an optimal solution to the CSG problem can be converted in polynomial time to a corresponding optimal SCSGA solution.) □

Theorem 2. *The SCSGA problem is* NP*-complete.*

Proof. NP-hardness follows directly from the fact that CSG is NP-hard (Sandholm et al. 1999) together with Theorem 1.

The corresponding decision problem for SCSGA is as follows: Given a SCSGA problem instance $\langle A, T, \boldsymbol{v} \rangle$ and a number $k \in \mathbb{R}$, does there exist a size-m ordered coalition structure with value at least k? This problem is in NP, since we can verify a solution's value in linear time by summing the values of its m coalition-to-alternative assignments. □

Note that SCSGA is analogous to a version of the *winner determination problem* (WDP) when *all possible bids* are given as input and there is no auctioneer; leading to the fact that CSG, SCSGA and WDP are in essence closely related problems. Albeit not surprising, this makes inexact approaches more interesting to investigate, since we already know that WDP is computationally hard to approximate (Sandholm et al. 2002). Moreover, it is in general not tractable to use WDP algorithms to solve SCSGA problems. One reason for this is because they are often designed to handle only a small number of bids, while SCSGA problems have $m2^n$ possible coalition-to-alternative assignments.

4 Algorithms

In this section, we present our SCSGA algorithms. The algorithms we present and investigate here use randomness to search different parts of the search space. Our greedy, heuristic algorithms find local optima quickly by using the agents' *marginal contributions* (see Definition 4). We also use randomized permutations of the input's agent set A to quickly generate new "starting points" for the algorithms (as we show in Subsection 4.5). These permutations typically correspond to the order for which the agents join the different coalitions. Also, note that we use the notation $X[i]$ for the i^{th} element of a vector or tuple X.

Definition 4. An agent a's *marginal contribution* $\boldsymbol{\Delta}_a(C, t)$ to the coalition $C \subseteq A \backslash \{a\}$ assigned to the alternative $t \in T$ is given by the identity: $\boldsymbol{\Delta}_a(C, t) = \boldsymbol{v}(C \cup \{a\}, t) - \boldsymbol{v}(C, t)$.

4.1 Permutative Greedy Algorithm

Algorithm 1 (abbreviated *GA*) generates a solution by sequentially assigning agents to alternatives in a greedy fashion. It requires $\mathcal{O}(nm)$ operations per run.

Note that this algorithm may generate arbitrarily bad solutions for certain types of problem instances—for example, when the value function is defined as follows:

$$\boldsymbol{v}(C, t) = \begin{cases} -|C|^2 & \text{if } |C| < |A| \\ \infty & \text{otherwise (i.e., } |C| = |A|) \end{cases}$$

for all $C \subseteq A$ and $t \in T$. This is because, for this type of problem, GA would try to keep all coalitions small (since it is the locally best strategy), while having only one large coalition would yield a better (infinitely valued) solution.

4.2 Permutative Steepest-Ascent Hill Climb

Algorithm 2 (abbreviated *HC*) works by taking an ordered coalition structure as input, and then continuously attempting to improve it by changing its alternative-assignments on an agent-to-agent basis. The algorithm terminates when this strategy cannot improve the solution's value any further (i.e., when a local optimum has been reached), or if a user-defined computation budget is exhausted. It requires $\mathcal{O}(nm)$ operations per "improvement", and it is combined with a *random sampling* method in Algorithm 3.

Algorithm 1 : `PermutativeGreed`($\mathcal{P}$, K)
$\mathcal{P} = \langle A, T, \boldsymbol{v} \rangle$ is a SCSGA problem instance, and the vector $K = \langle a_{k_1}, ..., a_{k_n} \rangle$ is a permutation of A.

$S \leftarrow \emptyset_m$ //S is initialized to a vector of m empty coalitions.
for $a = a_{k_1}, ..., a_{k_n}$ **do**
  $i^* \leftarrow \arg\max_{j \in \{1,...,m\}} \{\boldsymbol{\Delta}_a(C_j, t_j)\}$
  $S[i^*] \leftarrow S[i^*] \cup \{a\}$
return S

Algorithm 2 : `PermutativeHillClimb`($\mathcal{P}$, K, S)
$\mathcal{P} = \langle A, T, \boldsymbol{v} \rangle$ is a SCSGA problem instance, the vector $K = \langle a_{k_1}, ..., a_{k_n} \rangle$ is a permutation of A, and $S = \langle C_1, ..., C_m \rangle$ is an ordered coalition structure over A.

$success \leftarrow$ true
while $success =$ true, and budget not exhausted **do**
  $success \leftarrow$ false
  for $a = a_{k_1}, ..., a_{k_n}$ **do**
    $i \leftarrow j$ such that $a \in C_j$
    $i^* \leftarrow \arg\max_{j \in \{1,...,m\}} \{\boldsymbol{\Delta}_a(C_j \setminus \{a\}, t_j)\}$
    if $\boldsymbol{\Delta}_a(C_{i^*} \setminus \{a\}, t_{i^*}) > \boldsymbol{\Delta}_a(C_i \setminus \{a\}, t_i)$ **then**
      $S[i] \leftarrow S[i] \setminus \{a\}$; $S[i^*] \leftarrow S[i^*] \cup \{a\}$
      $success \leftarrow$ true //succeeded at improving S.
return S

Algorithm 3 : `RandomPermutativeHillClimb`($\mathcal{P}$, K)
$\mathcal{P} = \langle A, T, \boldsymbol{v} \rangle$ is a SCSGA problem instance, and the vector $K = \langle a_{k_1}, ..., a_{k_n} \rangle$ is a permutation of A.

$S \leftarrow$ `GenerateSolution()` //e.g., uniformly.
return `PermutativeHillClimb`($\mathcal{P}, K, S$)

Algorithm 4 : `PermutativeMCTS`($\mathcal{P}$, K)
$\mathcal{P} = \langle A, T, \boldsymbol{v} \rangle$ is a SCSGA problem instance, and the vector $K = \langle a_{k_1}, ..., a_{k_n} \rangle$ is a permutation of A.

$S \leftarrow \emptyset_m$
for $i = 1, ..., n$ **do**
  $G \leftarrow \langle a_{k_i}, ..., a_{k_n} \rangle$
  $root \leftarrow$ node representing S and G
  for $1, ..., RolloutsPerLevel$ **do**
    $node \leftarrow$ `TreePolicy`($root$)
    $\delta \leftarrow$ `RolloutPolicy`($node, G$)
    `Backpropagate`($node, \delta$)
  $j \leftarrow$ `BestAction`($root$); $S[j] \leftarrow S[j] \cup \{a_{k_i}\}$
return best solution found during rollouts

4.3 Permutative Monte Carlo Tree Search

Algorithm 4 (abbreviated *MCTS*) models a SCSGA problem instance as a decision tree, and uses Monte Carlo techniques to build and search it. MCTS locates a promising node, samples a feasible solution reachable from it, and uses the solution's value to guide future search. This process is repeated a number of times on each level of the decision tree. For every level, the tree's root node is updated to represent a solution S over $A \setminus U$, and a permutation $G = \langle a_{g_1}, ..., a_{g_{|G|}} \rangle$ of $U \subseteq A$, where U contains the agents that have not been assigned to an alternative yet. The root node always has m children, representing the m ways to assign a_{g_1} to an alternative. In detail, the following steps are applied each iteration:

1. `TreePolicy`: Recursively selects a node reachable from the root that maximizes:
$$v_x + c\sqrt{\frac{\ln n_p}{n_x}} + \sqrt{{\sigma_x}^2 + \frac{d}{n_x}}, \quad (3)$$
where v_x is the average value of previous rollouts through node x; n_x and n_p are the number of rollouts through node x and its parent p, respectively; and ${\sigma_x}^2$ is the variance of rollout results through node x. c and d are constants, where higher values encourages exploration. This formula was proposed by (Schadd et al. 2012) for solving puzzles with Monte Carlo tree search. Leaf nodes reached through this process are expanded by appending their possible child nodes to them.
2. `RolloutPolicy`: Takes a solution S over $A \setminus U$, and a permutation G of U, where U is the set of unassigned agents, and then produces an outcome δ. For producing δ, we use a random rollout, with which each unassigned agent is assigned to an alternative by drawing from a uniform distribution. We also store the best solution generated through any rollout.
3. `Backpropagate`: Saves the rollout's outcome δ to $node$, and then backpropagates it to all its ancestors. When $RolloutsPerLevel$ rollouts have been performed, `BestAction` selects the alternative to which a_{k_i} should be assigned. As is common practice, a child with the most rollouts is chosen (Browne et al. 2012).

4.4 Simulated Annealing

Algorithm 5 : `SimulatedAnnealing`($\mathcal{P}$)
$\mathcal{P} = \langle A, T, \boldsymbol{v} \rangle$ is a SCSGA problem instance.

$S \leftarrow$ `GenerateSolution()`; $S^* \leftarrow S$; $t \leftarrow \infty$
while computation budget is not exhausted **do**
  $k \sim \boldsymbol{\mathcal{U}}(1, m)$; $l \sim \boldsymbol{\mathcal{U}}(1, n)$; $p \sim \boldsymbol{\mathcal{U}}(0, 1)$
  $i \leftarrow j$ such that $a_l \in C_j$
  $S' \leftarrow S$; $S'[i] \leftarrow S'[i] \setminus \{a_l\}$; $S'[k] \leftarrow S'[k] \cup \{a_l\}$
  if $\boldsymbol{V}(S') > \boldsymbol{V}(S)$ or $p < P(S', S, S^*, t)$ **then**
    $S \leftarrow S'$
    if $\boldsymbol{V}(S) > \boldsymbol{V}(S^*)$ **then** $S^* \leftarrow S$
  $t \leftarrow (1/\text{computation budget portion used}) - 1$
return S^*

Algorithm 5 (abbreviated *SA*) uses a modification of the well-known metaheuristic optimization method *simulated annealing* (Kirkpatrick, Gelatt, and Vecchi 1983). Starting from a random solution, it generates new solutions by randomly "moving" an agent between two alternatives. A new solution S' generated from a solution S is accepted if $\boldsymbol{V}(S') > \boldsymbol{V}(S)$, or otherwise with probability:

$$P(S', S, S^*, t) = \exp\left(\frac{\boldsymbol{V}(S') - \boldsymbol{V}(S)}{t\boldsymbol{V}(S^*)}\right)$$

where S^* is the best solution found so far, S and S' are solutions, and $t \in \mathbb{R}$ decreases as the computation budget is exhausted. SA requires $\mathcal{O}(1)$ time per iteration.

Algorithm 6 : `RandomRestartAlgorithm`($\mathcal{P}$)
$\mathcal{P} = \langle A, T, \boldsymbol{v} \rangle$ is a SCSGA problem instance.

1: $K \leftarrow \langle a_1, ..., a_n \rangle$; $S^* \leftarrow \emptyset_m$; $S^*[1] \leftarrow A$
2: **while** computation budget is not exhausted **do**
3: $K \leftarrow$ `Shuffle`(K) //e.g., using Fisher–Yates.
4: $S \leftarrow$ `PermutativeSolver`($\mathcal{P}, K$)
5: **if** $\boldsymbol{V}(S) > \boldsymbol{V}(S^*)$ **then** $S^* \leftarrow S$
6: **return** S^*

4.5 Extensions

We extend the algorithms from the previous subsections with random-restart functionality in Algorithm 6, making it possible to use them to continuously find better solutions in an anytime fashion. `PermutativeSolver` thus implements Algorithm 1, 3 or 4. With Algorithm 3 or 4, and enough time, this method converges to a global optimum.

In our benchmarks, we also combined Algorithm 1 with Algorithm 2 in attempt to find better solutions even faster. This hybrid (abbreviated *HY*) works by using the permutative greedy algorithm to continuously generate initial high-quality solutions that are then passed on to the steepest-ascent hill climb algorithm (using line 1 in Algorithm 3) for further improvement using local search.

5 Benchmarks and Experiments

Our main goals with the empirical evaluation are to investigate: 1) how different input-sizes, value distributions and time constraints affect the algorithms; 2) the algorithms' performances relative to each other; 3) how the myopic nature of the greedy algorithms make them compare to the state-of-the-art; and 4) if the algorithms can generate high-quality solutions in short execution time for difficult large-scale problems. To accomplish these goals, and in accordance with (Präntare and Heintz 2020), we use *UPD* (uniform probability distribution) and *NPD* (normal probability distribution) for generating difficult problem instances:

- **UPD:** $\boldsymbol{v}(C,t) \sim \boldsymbol{\mathcal{U}}(0,1)$; and
- **NPD:** $\boldsymbol{v}(C,t) \sim \boldsymbol{\mathcal{N}}(\mu, \sigma^2)$, with $\mu = 1$ and $\sigma = 0.1$;

for all $C \subseteq A$ and $t \in T$. $\boldsymbol{\mathcal{U}}$ and $\boldsymbol{\mathcal{N}}$ are the uniform and normal distributions, respectively. We also define/use two new distributions: *SUPD* (sparse UPD) and *SNPD* (sparse NPD):

- **SUPD:** $\boldsymbol{v}(C,t) \sim \boldsymbol{\mathcal{U}}(0,1)$ with probability 0.01, else $\boldsymbol{v}(C,t) \sim \boldsymbol{\mathcal{U}}(0,0.1)$; and
- **SNPD:** $\boldsymbol{v}(C,t) \sim \boldsymbol{\mathcal{N}}(\mu_1, \sigma^2)$ with probability 0.01, else $\boldsymbol{v}(C,t) \sim \boldsymbol{\mathcal{N}}(\mu_2, \sigma^2)$, with $\mu_1 = 1$, $\mu_2 = 0.1$, $\sigma = 0.1$;

for all $C \subseteq A$ and $t \in T$. We expect these to be more difficult for greedy/myopic algorithms. Moreover, note that it is not possible to use the real-world SCSGA data presented in (Präntare and Heintz 2020) for our benchmarks. This is because that data is locked behind a non-disclosure agreement. Using synthetic experiments is thus the current best option for benchmarking, and we argue that an important future effort is to make real-world SCSGA data accessible.

Bear in mind that storing all values of the value function in a table requires $\omega(m2^n)$ memory and is thus not feasible for large-scale problems. For this reason, we only store the evaluated values, which we generate on-the-fly per query, and reuse them when they are needed again.

The result of each experiment was produced by computing the average of the resulting values from 20 generated problem sets per experiment. Following best practice, we plot the 95% confidence interval in all graphs.

All code was written in *C++11*, and all random numbers were generated with `uniform_real_distribution` and `normal_distribution` from the *C++ Standard Library*. We used the hash map `unordered_map` for storing evaluated values and `shuffle` for shuffling agent permutations from the same library. All tests were conducted with an AMD 3950X 3.5GHz CPU and 32GB memory.

For MCTS, the cross-entropy method (*CEM*) was used to tune the hyperparameters c and d, as described by (Chaslot et al. 2008). CEM is an evolutionary algorithm that maintains distributions of promising values for each parameter. In several iterations, samples are drawn from the distributions, and a metric is used to evaluate their fitness. The *elite samples* (samples with the highest fitness values) are used to update the distributions in preparation for the next iteration. In more detail, we used two distributions $\boldsymbol{\mathcal{N}}(\mu_c, \sigma_c)$ and $\boldsymbol{\mathcal{N}}(\mu_d, \sigma_d)$ for this, initiated with $\mu_c, \sigma_c, \mu_d, \sigma_d = 50$. In each iteration, we selected 100 samples, and evaluated them using MCTS. The mean solution value returned for 25 such problems determined the fitness value of each sample. At the end of each CEM iteration, μ_c and σ_c were set to the mean value and standard deviation respectively of the values for c used in 10 elite samples, and similar for μ_d and σ_d. We then ran several iterations until CEM converged, at which point we used the values of μ_c and μ_d for c and d, respectively. This was repeated for all four distributions.

Finally, we use the following abbreviations to denote the different random-restart algorithms when they are used in Algorithm 6: *RGA* for the greedy algorithm (Algorithm 1); *RHC* for the hill climb algorithm (Algorithm 3); *RMCTS* for the MCTS algorithm (Algorithm 4); *SA* for simulated annealing (Algorithm 5); and *RHY* for the hybrid algorithm (Algorithm 1 combined with Algorithm 2).

5.1 Small-Scale Near-Optimality Benchmarks

To see how the myopic nature of our algorithms affect their generated solutions' quality, we benchmark against the state-of-the-art algorithm MP (described in Section 1). Recall that, for problems with many agents, MP is not able to generate a solution in feasible time. For this reason, we only use a few agents in these benchmarks. We also plot the average optimal value (computed with MP) in all graphs.

Figure 1 clearly shows that all of our algorithms yield near-optimal solutions very quickly. For example, after 2 seconds, at worst case for the most difficult problem set, RHC achieves solutions that are roughly 98% of the optimal. Surprisingly, RHC greatly outperforms current state-of-the-art and all other methods. This was unexpected for SUPD and SNPD; since for them, MP should be able to quickly detect parts of the search space where it can find high-quality solutions. This shows that MP's effectiveness is more sensitive to a problem's value distribution than we anticipated.

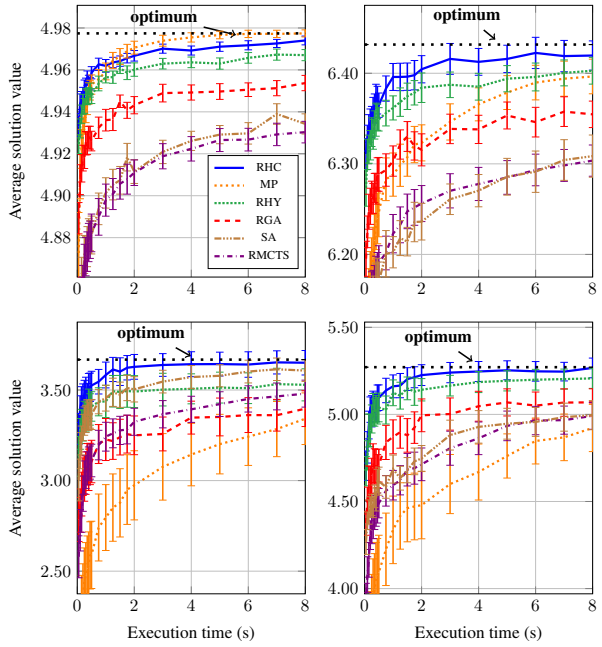

Figure 1: The average solution values obtained for **UPD** (top-left), **NPD** (top-right), **SUPD** (bottom-left) and **SNPD** (bottom-right) problem sets with 15 agents and 5 alternatives.

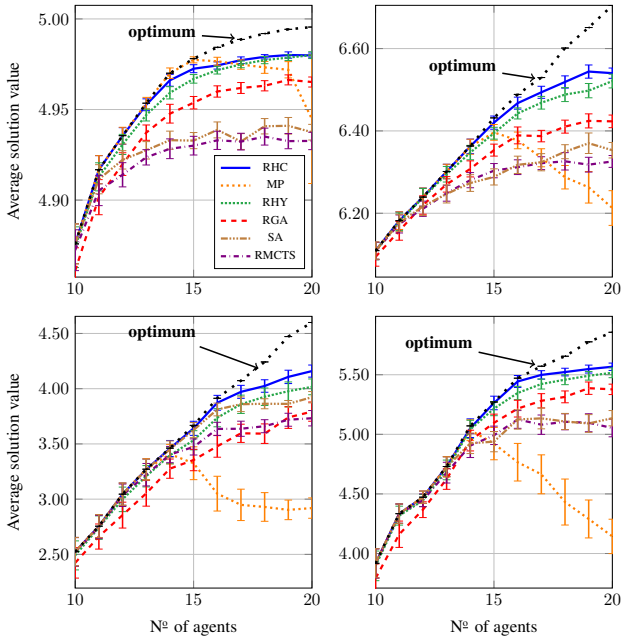

Figure 2: The average solution values obtained after 8 seconds for **UPD** (top-left), **NPD** (top-right), **SUPD** (bottom-left) and **SNPD** (bottom-right) problem sets with 5 alternatives.

Our results from varying the number of agents are shown in Figure 2, and they clearly indicate that—when the number of agents exceed a certain number—MP fails to quickly generate feasible solutions of high quality. This is because it is not able to compute bounds for the different branches of the search tree quickly enough, which leads to exponentially worse performance as the number of agents is increased. Naturally, our algorithms do not exhibit this behaviour.

5.2 Large-Scale Benchmarks

To see how our algorithms behave for large-scale problems, we plot the results from benchmarks with large inputs in Figures 4 and 3. We used a time limit of 100 seconds in these benchmarks, since after this time, the algorithms typically exhausted our memory budget (32GB) due to the memory-footprint of storing the evaluated values of the value function. Note that the problems in these large-scale benchmarks have immense search spaces with sizes up to $50^{1000} \approx 10^{1700}$, and we expect that no algorithms exist that can generate an optimal (or approximate) solution for them in feasible time. Thus, to deduce how good a solution is, we instead propose to compute a probability indicative to its quality. To illustrate this, in Figure 4, we see that e.g., RHC finds a solution of value ≈ 49.5 for UPD after 1 second. Numerical computation together with Corollary 1 gives:

$$\boldsymbol{P}(\boldsymbol{V}(S) \geq 49.5) = \int_{49.5}^{50} \boldsymbol{f}_{\mathcal{U}}^{50}(x)\, dx \approx 2 \times 10^{-80}$$

for a solution S. This value can be interpreted as the expected portion of feasible solutions with value 49.5 or greater, and is thus illustrative to RHC's (high-yield) performance. We provide an analogous result for NPD in Corollary 2 that can be used similarly. For UPD and NPD, we use these corollaries to compute and plot an approximate expected optimal value (i.e., the threshold value for which we expect there to exist exactly one solution). We do not compute a similar value for SUPD and SNPD, since it is out of this paper's scope to derive their probability density functions. Hence, in the following large-scale benchmarks, we only compute the expected optimal value for UPD/NPD.

Corollary 1. *For a SCSGA problem instance with m alternatives and UPD-distributed values, the probability density function for the distribution of $\boldsymbol{V}$ is equal to:*

$$\boldsymbol{f}_{\mathcal{U}}^{m}(x) = \frac{1}{2(m-1)!} \sum_{k=0}^{m} (-1)^k \binom{m}{k} (x-k)^{m-1} sgn(x-k).$$

Proof. This follows directly from the Irwin-Hall distribution's probability density function, see e.g., (Hall 1927). □

Corollary 2. *For a SCSGA problem instance with m alternatives and NPD-distributed values, the probability density function for the distribution of $\boldsymbol{V}$ is equal to:*

$$\boldsymbol{f}_{\mathcal{N}}^{m}(x, \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi m}} \exp\Big(-\frac{1}{2}\Big(\frac{x - m\mu}{\sigma\sqrt{m}}\Big)^2\Big).$$

Proof. This follows directly from the known probability density function of a normal distribution. □

The benchmarks in Figures 3 and 4 show that RHY, RHC and RGA are able to generate near-optimal solutions for

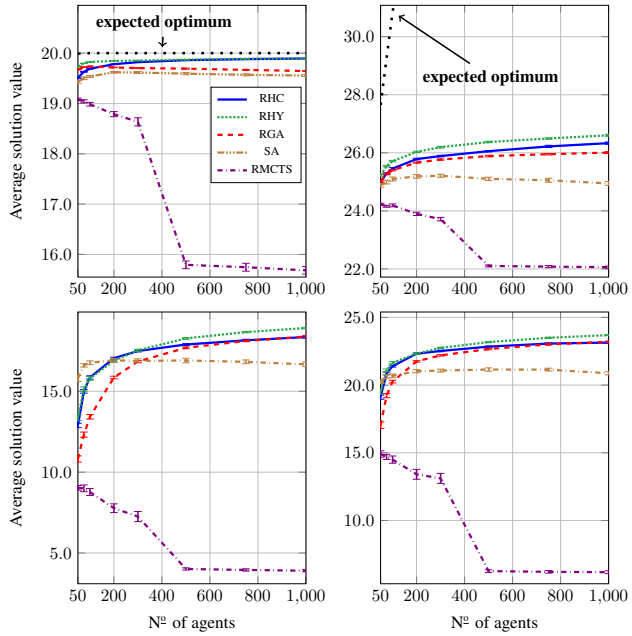

Figure 3: The solution values obtained by the algorithms after 100 seconds for **UPD** (top-left), **NPD** (top-right), **SUPD** (bottom-left) and **SNPD** (bottom-right) problem sets with 20 alternatives.

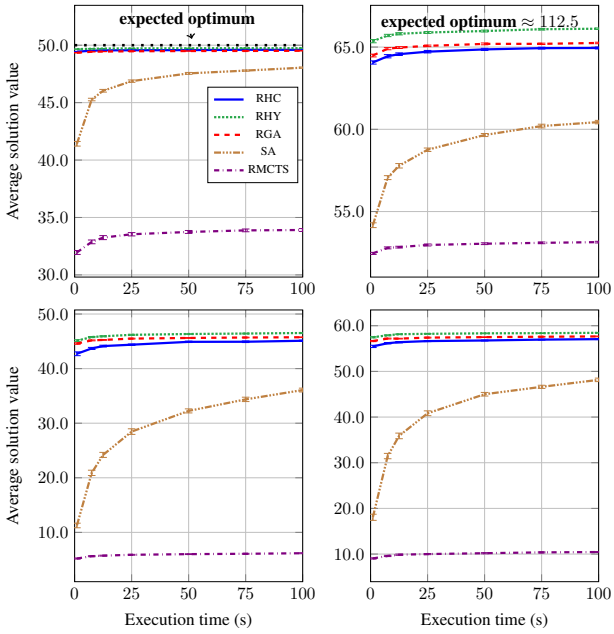

Figure 4: The solution values obtained for **UPD** (top-left), **NPD** (top-right), **SUPD** (bottom-left) and **SNPD** (bottom-right) problem sets with 1000 agents and 50 alternatives.

problems with uniformly distributed values despite an increasing number of agents. The results also show that RHY, RHC and RGA are able to generate solutions of relatively high quality very quickly for problems with immense search spaces and large input-sizes. Despite being far from the expected optimum for normally distributed problems, e.g., with 1000 agents and 50 alternatives, Corollary 2 can be used to show that the portion of solutions with a value ≥ 65 (which RHC and RGA generate after roughly 25 seconds, and RHY after a few milliseconds) is expected to be $\approx 3 \times 10^{-100}$—an incredibly small portion of the total number of solutions.

In contrast to these algorithms, the quality of solutions found by SA progressively worsens as the number of agents increases. It is notable that SA performs comparatively better on the sparse distributions. It seems reasonable that this is because the lower number of high-quality solutions favours an exploration-focused approach. SA is also the only algorithm for which increased computation time consistently gives better solutions.

The performance of RMCTS quickly declines when the number of agents exceeds 300. This is because with our random rollout strategy, the time complexity of RMCTS is quadratic in the number of agents. It is thus unable to build any tree that can reasonably guide its search before it has to "cut a level"—effectively pruning/discarding a large part of the search space at random.

6 Summary and Conclusions

In this paper, we developed and used several general-purpose heuristic and Monte Carlo methods for simultaneous coalition structure generation and assignment (SCSGA). In more detail, we presented five different centralized, heuristic SCSGA algorithms based on various paradigms (e.g., Monte Carlo tree search, simulated annealing, local search). In our benchmarks, the presented myopic algorithms (e.g., local search) are greatly superior to the state-of-the-art, simulated annealing, and a baseline implementation of Monte Carlo tree search. In other words, the myopic algorithms are superior at quickly finding high-quality solutions for problems with large input sizes and normally/uniformly distributed values. They quickly find close-to-optimal feasible solutions in many of our large-scale experiments. For example, in our benchmarks with uniformly distributed values and immense search spaces of size $50^{1000} \approx 10^{1700}$, our hill-climb hybrid finds 99% efficient solutions in less than one second. Moreover, all presented algorithms find near-optimal solutions almost instantly for small-scale problems. Finally, by providing initial theory and empirical data for this difficult optimization problem, we have established a first baseline for future work in large-scale SCSGA.

As a final note, we stress that the only real data that has been used in research is currently protected and inaccessible for legal reasons (Prantare and Heintz 2020). In light of this, we believe that the most important strand of future work to consider is publishing real-world data, and then using it for benchmarking/developing new specialized algorithms. It would also be interesting to investigate if machine learning algorithms can be used for SCSGA—we believe that both *machine learning alongside optimization algorithms* and *end-to-end methods* are interesting approaches to look into. Genetic and k-opt algorithms also remain unexplored.

Acknowledgments

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

References

Browne, C. B.; Powley, E.; Whitehouse, D.; Lucas, S. M.; Cowling, P. I.; Rohlfshagen, P.; Tavener, S.; Perez, D.; Samothrakis, S.; and Colton, S. 2012. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games* 4(1): 1–43.

Chaslot, G. M.-B.; Winands, M. H.; Szita, I.; and van den Herik, H. J. 2008. Cross-entropy for Monte-Carlo tree search. *Journal* 31(3): 145–156.

Di Mauro, N.; Basile, T. M.; Ferilli, S.; and Esposito, F. 2010. Coalition structure generation with grasp. In *International Conference on Artificial Intelligence: Methodology, Systems, and Applications*. Springer.

Farinelli, A.; Bicego, M.; Bistaffa, F.; and Ramchurn, S. D. 2017. A hierarchical clustering approach to large-scale near-optimal coalition formation with quality guarantees. *Engineering Applications of Artificial Intelligence* 59: 170–185.

Hall, P. 1927. The distribution of means for samples of size n drawn from a population in which the variate takes values between 0 and 1, all such values being equally probable. *Biometrika* 19(3/4): 240–245.

Keinänen, H. 2009. Simulated annealing for multi-agent coalition formation. In *KES International Symposium on Agent and Multi-Agent Systems: Technologies and Applications*, 30–39. Springer.

Kirkpatrick, S.; Gelatt, C. D.; and Vecchi, M. P. 1983. Optimization by simulated annealing. *science* 220(4598): 671–680.

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

Osborne, M. J.; and Rubinstein, A. 1994. *A course in game theory*. MIT press.

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

Präntare, F. 2017. Simultaneous coalition formation and task assignment in a real-time strategy game. In *Master thesis*.

Präntare, F.; and Heintz, F. 2020. An anytime algorithm for optimal simultaneous coalition structure generation and assignment. *Autonomous Agents and Multi-Agent Systems* 34(1): 1–31.

Rahwan, T.; Michalak, T. P.; Wooldridge, M.; and Jennings, N. R. 2015. Coalition structure generation: A survey. *Artificial Intelligence* 229: 139–174.

Sandholm, T.; Larson, K.; Andersson, M.; Shehory, O.; and Tohmé, F. 1999. Coalition structure generation with worst case guarantees. *Artificial Intelligence* 111(1-2): 209–238.

Sandholm, T.; Suri, S.; Gilpin, A.; and Levine, D. 2002. Winner determination in combinatorial auction generalizations. In *Proceedings of the first International Joint Conference on Autonomous Agents and Multiagent Systems*.

Schadd, M. P.; Winands, M. H.; Tak, M. J.; and Uiterwijk, J. W. 2012. Single-player Monte-Carlo tree search for SameGame. *Knowledge-Based Systems* 34: 3 – 11. A Special Issue on Artificial Intelligence in Computer Games: AICG.

Sen, S.; and Dutta, P. S. 2000. Searching for optimal coalition structures. In *Proceedings Fourth International Conference on MultiAgent Systems*, 287–292. IEEE.

Wu, F.; and Ramchurn, S. D. 2020. Monte-Carlo Tree Search for Scalable Coalition Formation. In *Proceedings of the 29th International Joint Conference on Artificial Intelligence*.

Yeh, C.; and Sugawara, T. 2016. Solving coalition structure generation problem with double-layered ant colony optimization. In *5th International Congress on Advanced Applied Informatics*. IEEE.