Hybrid Dynamic Programming for Simultaneous Coalition Structure Generation and Assignment

Fredrik Präntare[0000−0002−0367−2430] and Fredrik Heintz[0000−0002−9595−2471]

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

Abstract. We present, analyze and benchmark two algorithms for simultaneous coalition structure generation and assignment: one based entirely on dynamic programming, and one anytime hybrid approach that uses branch-and-bound together with dynamic programming. To evaluate the algorithms' performance, we benchmark them against both CPLEX (an industry-grade solver) and the state-of-the-art using difficult randomized data sets of varying distribution and complexity. Our results show that our hybrid algorithm greatly outperforms CPLEX, pure dynamic programming and the current state-of-the-art in all of our benchmarks. For example, when solving one of the most difficult problem sets, our hybrid approach finds optimum in roughly 0.1% of the time that the current best method needs, and it generates 98% efficient interim solutions in milliseconds in all of our anytime benchmarks; a considerable improvement over what previous methods can achieve.

Keywords: Combinatorial assignment · Dynamic programming · Coalition formation · Coalition structure generation · Games with alternatives.

1 Introduction

Forming teams of agents and coordinating them is central to many applications in both artificial intelligence and operations research. In cooperative game theory, this is known as *coalition formation*—the process by which heterogeneous agents group together to achieve some goal. Central to this endeavor is: i) optimally partitioning the set of agents into disjoint groups—an optimization problem known as *coalition structure generation* (CSG) [13,11]; and ii) deciding on the teams' individual goals, which can be modelled as a *linear assignment* problem [2,4]. Combining these problems, and solving them simultaneously, can potentially both reduce a problem's computational complexity and increase the agents' aggregated potential utility and performance [6]. This combined CSG and linear assignment problem is a general case of utilitarian combinatorial assignment, and it is known as *simultaneous coalition structure generation and assignment* (SCSGA) in the multi-agent research community.

Technically, from a game theoretic perspective, SCSGA is a CSG problem for *games with alternatives* [1]. In this game type, there is a set of agents $A = \{a_1, ..., a_n\}$, and several alternatives $t_1, ..., t_m$, of which each agent must choose exactly one, with $C_i \subseteq \{a_1, ..., a_n\}$ defined to be the set of agents who choose alternative t_i. The vector

$\langle C_1, ..., C_m \rangle$ thus constitutes an ordered coalition structure over A. In SCSGA, the goal is to find an ordered coalition structure that maximizes welfare in such contexts.

Moreover, SCSGA algorithms have a range of potential different applications in many domains. They can for example be used to deploy personnel to different locations and/or allocate alternatives to agents (examples include utilitarian course allocation and winner determination in combinatorial auctions). SCSGA is also the only CSG paradigm in the literature that has been demonstrated for use in a real-world commercial application to improve agents' coordination capabilities, wherein it has been used to optimally form and deploy teams of agents to different geospatial regions [5]. However, the state-of-the-art algorithm can only solve problems with severely limited inputs with up to roughly 20 agents in reasonable time. Although this algorithm performs fairly well in practice and greatly outperforms the industry-grade solver CPLEX, it suffers from there being no proven guarantee that it can find an optimum without first evaluating all the m^n possible feasible solutions. [7]

To address these issues, we develop an algorithm with a proven worst-case time complexity better (lower) than $\mathcal{O}(m^n)$, and devise a second algorithm that finds both optimal and anytime (interim) solutions faster than the state-of-the-art. More specifically, we focus on the paradigm *dynamic programming* to accomplish this, and investigate how dynamic programming can be combined with branch-and-bound to obtain the best features of both. Against this background, our two main contributions that advances the state-of-the-art are the following:

- We develop, present and benchmark *DP*—a simple, easy-to-implement dynamic programming algorithm for SCSGA. We also analyze it, and prove its correctness and worst-case time/memory complexity, consequently showing that it has the lowest worst-case time complexity proven in the literature.
- We develop and present *HY*—a hybrid optimal anytime SCSGA algorithm that uses dynamic programming together with branch-and-bound. Subsequently, we empirically show that our hybrid algorithm greatly outperforms both current state-of-the-art and the industry-grade solver CPLEX in all of our benchmarks. We also provide empirical data that shows that the hybrid algorithm is more robust to the distribution of values compared to the state-of-the-art.

The remainder of this paper is structured as follows. We begin by presenting related work in Section 2. Then, in Section 3, we define the basic concepts that we use throughout this report. In Section 4, we describe our pure dynamic programming algorithm, and in Section 5 we show how we combine dynamic programming techniques with branch-and-bound. In Section 6, we present our experiments. Finally, in Section 7, we conclude with a summary.

2 Related Work

The only optimal algorithm in the literature that has been developed for the SCSGA problem is the aforementioned branch-and-bound algorithm. (We improve on this work by combining it with dynamic programming to construct a stronger hybrid algorithm.) Apart from this, a plethora of different optimal algorithms have been developed for

the closely related *characteristic function game* CSG problem. The first algorithm presented for it used dynamic programming [14], which [8] then improved upon by finding ways to guarantee optimality while making fewer evaluations. These algorithms both run in $\mathcal{O}(3^n)$ for n agents, and have the disadvantage that they produce no interim solutions—i.e., they generate no solution at all if they are terminated before completion. Subsequently, [12] presented an anytime tree search algorithm based on branch-and-bound that circumvented this issue, but at the cost of a much worse worst-case time complexity of $\mathcal{O}(n^n)$. In addition to these algorithms, several hybrid algorithms have been proposed. They fuse earlier methods in an attempt to obtain the best features of their constituent parts [9,3,10].

However, all of these CSG algorithms were specifically designed for problems without alternatives. Consequently, they: a) only consider *unordered* coalition structures, while we need to consider all permutations of them; b) do not allow empty coalitions in solutions—in SCSGA, an empty coalition corresponds to no agents choosing an alternative, while in CSG, empty coalitions have no clear purpose or practical interpretation; and c) evaluate coalition structures of any size (we are only interested in size-m ordered coalition structures, where m is the number of alternatives). These properties arguably renders it difficult (or impossible) to use them for SCSGA in a straightforward fashion without greatly sacrificing computational performance.

3 Basic Concepts and Notation

The SCSGA problem is defined as follows:

Input: a set of agents $A = \{a_1, ..., a_n\}$, a vector of alternatives $T = \langle t_1, ..., t_m \rangle$, and a function $\boldsymbol{v} : 2^A \times T \mapsto \mathbb{R}$ 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. *$\langle C_1, ..., C_m \rangle$ is an ordered coalition structure over A if $C_i \cap C_j = \emptyset$ for all $i \neq j$, and $\bigcup_{i=1}^{m} C_i = A$. We omit the notion "over A" for brevity.*

As is common practice, we use:

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

to denote the value of an ordered coalition structure $S = \langle C_1, ..., C_m \rangle$; the conventions $n = |A|$ and $m = |T|$ when it improves readability; and the terms *solution* and *ordered coalition structure* interchangeably. For a multiset X, we use $\boldsymbol{\mathcal{P}}(X)$ to denote its powerset. We use Π_A for the set of all ordered coalition structures over A, and define:

$$\Pi_A^m = \{S \in \Pi_A : |S| = m\}.$$

Finally, we say that a solution S^* is *optimal* if and only if:

$$\boldsymbol{V}(S^*) = \max_{S \in \Pi_A^m} \boldsymbol{V}(S).$$

4 The Dynamic Programming Algorithm

The DP algorithm is straightforwardly based on computing the following recurrence:

$$\boldsymbol{w}(U,k) = \begin{cases} \boldsymbol{v}(U,t_k) & \text{if } k=1 \\ \max_{C \in \boldsymbol{\mathcal{P}}(U)} \boldsymbol{v}(C,t_k) + \boldsymbol{w}(U \setminus C, k-1) & \text{if } k=2,...,m \end{cases} \tag{1}$$

where $U \subseteq A$. As shown in Theorem 1, this recurrence's value is equal to the value of the highest-valued k-sized ordered coalition structure over $U \subseteq A$.

Theorem 1. *If $U \subseteq A$ and $k \in \{1,...,m\}$, then:*

$$\boldsymbol{w}(U,k) = \max_{S \in \Pi_U^k} \boldsymbol{V}(S).$$

Proof. By straightforward induction. This holds for $k=1$, since $\langle U \rangle$ is the only 1-sized ordered coalition structure over U that exists, and consequently:

$$\max_{S \in \Pi_U^1} \boldsymbol{V}(S) = \boldsymbol{V}(\langle U \rangle) = \boldsymbol{v}(U,t_1) = \boldsymbol{w}(U,1). \tag{2}$$

We now show for $j = 2,...,m$, that if our theorem holds for $k = j-1$, then it also holds for $k=j$. First, note that:

$$\max_{S \in \Pi_U^k} \boldsymbol{V}(S) = \max_{C \in \boldsymbol{\mathcal{P}}(U)} \left\{ \boldsymbol{v}(C,t_k) + \max_{S \in \Pi_{U \setminus C}^{k-1}} \boldsymbol{V}(S) \right\} \tag{3}$$

holds for $k = 2,...,m$ and $U \subseteq A$. Now, for some $j \in \{2,...,m\}$, let our inductive hypothesis be:

$$\boldsymbol{w}(U,j-1) = \max_{S \in \Pi_U^{j-1}} \boldsymbol{V}(S)$$

for all $U \subseteq A$. This in conjunction with (1) gives:

$$\boldsymbol{w}(U,j) = \max_{C \in \boldsymbol{\mathcal{P}}(U)} \left\{ \boldsymbol{v}(C,t_j) + \max_{S \in \Pi_{U \setminus C}^{j-1}} \boldsymbol{V}(S) \right\}.$$

Consequently, together with (3), we have: $\boldsymbol{w}(U,j) = \max_{S \in \Pi_U^j} \boldsymbol{V}(S)$, which together with (2) proves the theorem. □

Value	**Prerequisite values**
$\boldsymbol{w}(\{a_1, a_2\}, 3)$	$\boldsymbol{w}(\emptyset, 2),\ \boldsymbol{w}(\{a_1\}, 2),\ \boldsymbol{w}(\{a_2\}, 2),\ \boldsymbol{w}(\{a_1, a_2\}, 2)$
$\boldsymbol{w}(\emptyset, 2)$	$\boldsymbol{w}(\emptyset, 1)$
$\boldsymbol{w}(\{a_1\}, 2)$	$\boldsymbol{w}(\emptyset, 1),\ \boldsymbol{w}(\{a_1\}, 1)$
$\boldsymbol{w}(\{a_2\}, 2)$	$\boldsymbol{w}(\emptyset, 1),\ \boldsymbol{w}(\{a_2\}, 1)$
$\boldsymbol{w}(\{a_1, a_2\}, 2)$	$\boldsymbol{w}(\emptyset, 1),\ \boldsymbol{w}(\{a_1\}, 1),\ \boldsymbol{w}(\{a_2\}, 1),\ \boldsymbol{w}(\{a_1, a_2\}, 1)$
$\boldsymbol{w}(\emptyset, 1)$	-
$\boldsymbol{w}(\{a_1\}, 1)$	-
$\boldsymbol{w}(\{a_2\}, 1)$	-
$\boldsymbol{w}(\{a_1, a_2\}, 1)$	-

Fig. 1: The prerequisite values needed to compute $\boldsymbol{w}(A, m)$ for $A = \{a_1, a_2\}$ and $m = 3$. The symbol "-" represents that no prerequisite values have to be computed.

Importantly for DP, the equality $\boldsymbol{w}(A, m) = \max_{S \in \Pi_A^m} \boldsymbol{V}(S)$ follows as a special case of Theorem 1. Consequently, a solution $S^* \in \Pi_A^m$ is optimal if and only if $\boldsymbol{V}(S^*) = \boldsymbol{w}(A, m)$. The DP algorithm works by computing $\boldsymbol{w}(A, m)$, while simultaneously constructing two tables that are subsequently used to generate an optimal solution that corresponds to the process by which this value is computed. However, computing $\boldsymbol{w}(A, m)$ recursively in a naïve fashion has the consequence that identical function calls have to be computed multiple times, as illustrated in Figure 1.

In light of this, we introduce two different approaches for computing $\boldsymbol{w}(A, m)$ that do not introduce such redundancy: Algorithm 1, which uses memoization to store intermediary results, so that a function call never has to be computed more than once; and Algorithm 2, which uses tabulation so that a value is only evaluated once all its prerequisite values have been computed.

Algorithm 1 : `DPMemoization`$(U = A, k = m)$
Based on Theorem 1, this algorithm recursively computes $\boldsymbol{w}(A, m)$, while simultaneously generating the two tables Γ_w and Γ_c.

1: **if** $\Gamma_w[U, k] \neq null$ **then**
2: **return** $\Gamma_w[U, k]$
3: **if** $k = 1$ **then**
4: $\Gamma_w[U, k] \leftarrow \boldsymbol{v}(U, t_k);\ \ \Gamma_c[U, k] \leftarrow U$ //Base case.
5: **return** $\boldsymbol{v}(U, t_k)$
6: **for all** $C \in \boldsymbol{\mathcal{P}}(U)$ **do**
7: $w \leftarrow \boldsymbol{v}(C, t_k) +$ `DPMemoization`$(U \setminus C, k - 1)$
8: **if** $\Gamma_w[U, k] = null$, or $w > \Gamma_w[U, k]$ **then**
9: $\Gamma_w[U, k] \leftarrow w;\ \ \Gamma_c[U, k] \leftarrow C$
10: **return** $\Gamma_w[U, k]$

Algorithm 2 : `DPTabulation()`
Based on Theorem 1, this algorithm iteratively computes $\boldsymbol{w}(A,m)$, while simultaneously generating the two tables $\mathit{\Gamma}_w$ and $\mathit{\Gamma}_c$.

```
1: for all C ∈ 𝓟(A) do
2:     Γ_w[C,1] ← v(C,t_1);  Γ_c[C,1] ← C                    //Base case.
3: for k = 2,...,m do
4:     for all U ∈ 𝓟(A) do
5:         for all C ∈ 𝓟(U) do
6:             w ← v(C,t_k) + Γ_w[U \ C, k−1]
7:             if Γ_w[U,k] = null, or w > Γ_w[U,k] then
8:                 Γ_w[U,k] ← w;  Γ_c[U,k] ← C
9: return Γ_w[A,m]
```

For both approaches, the tables $\mathit{\Gamma}_c$ and $\mathit{\Gamma}_w$ are used to store the following coalitions and values:

- $\mathit{\Gamma}_c[U,k] \leftarrow U$,
- $\mathit{\Gamma}_w[U,k] \leftarrow \boldsymbol{v}(U,t_k)$

for every $U \subseteq A$ and $k=1$; and

- $\mathit{\Gamma}_c[U,k] \leftarrow \arg\max_{C \in \boldsymbol{\mathcal{P}}(U)} \boldsymbol{v}(C,t_k) + \mathit{\Gamma}_w[U \setminus C, k-1]$,
- $\mathit{\Gamma}_w[U,k] \leftarrow \max_{C \in \boldsymbol{\mathcal{P}}(U)} \boldsymbol{v}(C,t_k) + \mathit{\Gamma}_w[U \setminus C, k-1]$

for every $U \subseteq A$ and $k = 2,...,m$. If each coalition is represented in constant size using e.g., a fixed-size binary string defined by its *binary-coalition encoding* (Definition 2), these tables require $\mathcal{O}(m|\boldsymbol{\mathcal{P}}(A)|) = \mathcal{O}(m2^n)$ space.

Definition 2. *The* binary coalition-encoding *of* $C \subseteq A$ *over* $A = \langle a_1,...,a_{|A|}\rangle$ *is the binary string* $j = b_{|A|}...b_1$ *with:*

$$b_i = \begin{cases} 1 & \text{if } a_i \in C \\ 0 & \text{otherwise} \end{cases}$$

For example, the binary coalition-encoding of $\{a_1,a_3\}$ *over* $\langle a_1,a_2,a_3\rangle$ *is equal to* 101.

To construct an optimal solution, DP uses Algorithm 3 together with the table $\mathit{\Gamma}_c$ (that has been generated using either Algorithm 1 or Algorithm 2), with which DP's worst-case time complexity is $\mathcal{O}(m3^n)$, as shown in Theorem 2.

Algorithm 3 : `DPConstruct()`
Uses the table $\mathit{\Gamma}_c$ (generated by e.g., Algorithm 1 or Algorithm 2) to construct an optimal ordered coalition structure.

```
1: U ← A;  S* ← ∅_m
2: for i = m,...,1 do
3:     S*[i] ← Γ_c[U,i];  U ← U \ Γ_c[U,i]
4: return S*
```

Theorem 2. *DP's worst-case time complexity is* $\mathcal{O}(m3^n)$.

Proof. `DPTabulation` (Algorithm 2) makes $\mathcal{O}(2^n)$ elementary operations on lines 1-2, and then proceeds to perform a total of $\mathcal{O}(mQ_n)$ operations on lines 3-8 for some $Q_n \in \mathbb{N}^+$. `DPConstruct` runs in $\mathcal{O}(m)$. Therefore, DP's worst-case time complexity is equal to:

$$\mathcal{O}(2^n) + \mathcal{O}(mQ_n) + \mathcal{O}(m) = \mathcal{O}(2^n + mQ_n). \tag{4}$$

Now recall that, for a set with n elements, there exists exactly $\binom{n}{k}$ possible k-sized subsets. Consequently, we have:

$$Q_n = \sum_{i=0}^{n} \binom{n}{i} 2^i, \tag{5}$$

since we iterate over $(m-1)2^i$ i-sized subsets for $i = 0, ..., n$ on lines 4-5. Also, as a consequence of the *binomial theorem*, the following holds:

$$(1+2)^n = \sum_{i=0}^{n} \binom{n}{i} 1^{n-i} 2^i = \sum_{i=0}^{n} \binom{n}{i} 2^i.$$

From this and (5), it follows that $Q_n = (1+2)^n = 3^n$. This together with (4) proves the theorem. □

5 The Hybrid Algorithm

Our hybrid algorithm (HY) is designed to combine the redundancy-eliminating capabilities of dynamic programming with the pruning abilities and anytime characteristics of branch-and-bound. In more detail, it incorporates these techniques with the search space presentation based on multiset permutations of integer partitions proposed by [6]. In their search space representation, each multiset permutation (ordered arrangement) of a size-m *zero-inclusive integer partition* of n (see Definition 3) corresponds to a set of solutions. More formally, if $P = \langle p_1, ..., p_m \rangle$ is such an ordered arrangement, it *represents* all solutions $\langle C_1, ..., C_m \rangle$ with $|C_i| = p_i$ for $i = 1, ..., m$.

Definition 3. *The multiset of non-negative integers* $\{x_1, ..., x_k\}$ *is a* zero-inclusive integer partition *of* $y \in \mathbb{N}$ *if:*

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

For example, the multiset $\{0, 1, 1, 2, 3\}$ *is a zero-inclusive integer partition of* 7*, since* $0 + 1 + 1 + 2 + 3 = 7$.

For brevity and convenience, we define the *subspace* $\mathcal{S}_P$ represented by P as the following set of solutions:

$$\mathcal{S}_P = \{\langle C_1, ..., C_m \rangle \in \Pi_A^m : |C_i| = p_i \text{ for } i = 1, ..., m\}.$$

For example, for a SCSGA problem instance with the set of agents $\{a_1, a_2, a_3\}$ and vector of alternatives $\langle t_1, t_2 \rangle$ as input, $\langle 1, 2 \rangle$ represents the following three solutions (and the subspace that constitutes them):

$$\langle \{a_1\}, \{a_2, a_3\} \rangle, \quad \langle \{a_2\}, \{a_1, a_3\} \rangle, \quad \langle \{a_3\}, \{a_1, a_2\} \rangle.$$

As shown by [6], it is possible to compute a lower and an upper bound for the value of the best solution in such a subspace without having to evaluate any ordered coalition structures. To accomplish this, first define:

$$\mathcal{K}_p = \{C \subseteq A : |C| = p\}.$$

Then, with the purpose to compute a lower bound, let:

$$\boldsymbol{A}(p, t) = \frac{1}{|\mathcal{K}_p|} \sum_{C \in \mathcal{K}_p} \boldsymbol{v}(C, t);$$

and to compute an upper bound, define:

$$\boldsymbol{M}(p, t) = \max_{C \in \mathcal{K}_p} \boldsymbol{v}(C, t).$$

A lower bound and an upper bound for all solutions represented by $P = \langle p_1, ..., p_m \rangle$ (if $\mathcal{S}_P \neq \emptyset$) can now be computed as $l_P = \sum_{i=1}^{m} \boldsymbol{A}(p_i, t_i)$ and $u_P = \sum_{i=1}^{m} \boldsymbol{M}(p_i, t_i)$, respectively. See [6] for proofs.

In light of these observations, we now propose a new algorithm (Algorithm 4) for searching such subspaces: *ADP* (for anytime DP). Technically, ADP uses depth-first branch-and-bound combined with an alteration of the dynamic programming techniques used in Algorithm 1. By using branch-and-bound, ADP only generates solutions that are better than the best solution that has already been found, and discards (prunes) branches of the recursion tree when they are deemed sufficiently bad. To accomplish this, ADP introduces the following variables:

- v^* : denotes the value of the best solution found so far; this is a globally kept variable initialized to $-\infty$, and it is not reinitialized when a subspace search is initiated.
- α : equals the sum of the values of all antecedent "fixed" coalition-to-alternative assignments (at shallower recursion depths).
- β : equals the most α can possibly increase at subsequent recursion steps deeper down in the recursion tree; it is initialized to u_P through a straightforward evaluation of the value function.

Consequently, since the sum $\alpha + \beta$ constitutes an upper bound on the recursion branch, the recursion can be discarded if $\alpha + \beta \leq v^*$ (see line 3 in Algorithm 4) without forfeiting optimality. Furthermore, ADP uses the tables Γ_c and Γ_w in the same fashion as DP uses them (i.e., to prevent evaluating the same function call again), with the difference that ADP only stores the values that are needed for generating the best solution for the subspace that is being investigated. The specific entries that are computed thus depends

Algorithm 4 : $\texttt{ADP}(P = \langle p_1, ..., p_m \rangle, U = A, k = m, \alpha = 0, \beta = u_P)$
Computes $\max_{S \in \mathcal{S}_P} \boldsymbol{V}(S)$ using dynamic programming together with depth-first branch-and-bound, while simultaneously generating entries for the tables $\mathit{\Gamma}_c$ and $\mathit{\Gamma}_w$.

```
1: if k = 1 then                                         //Base case.
2:     return v(U, t_k)
3: if α + β ≤ v* then
4:     return −∞                          //Cannot yield a better solution.
5: if Γ_w[U, k] exists then               //Has this call been evaluated?
6:     return Γ_w[U, k]
7: v ← −∞;  C ← ∅
8: for all C' ∈ 𝓟(U) ∩ 𝒦_{p_k} do
9:     if computation budget is exhausted then
10:        break
11:    α' ← α + v(C', t_k);  β' ← β − M(p_k, t_k)
12:    v' ← v(C', t_k) + ADP(P, U \ C', k − 1, α', β')
13:    if v' > v then
14:        v ← v';  C ← C'                          //Found a better choice.
15: if v ≠ −∞ then
16:    Γ_w[U, k] ← v;  Γ_c[U, k] ← C                   //Cache best choice.
17:    if α + v > v* then
18:        v* ← α + v                          //We found a better solution.
19: return v
```

both on the subspace's representation, and the distribution of values—therefore, it is not clear beforehand how many entries that need to be computed (we investigate this further in Section 6.1). Finally, if Algorithm 4 returns a value larger than $-\infty$, then $\mathit{\Gamma}_c$ can be used to construct a solution $\arg\max_{S \in \mathcal{S}_P} \boldsymbol{V}(S)$ in a similar fashion as DP does. If $-\infty$ is returned, then $\max_{S \in \mathcal{S}_P} \boldsymbol{V}(S) \leq v^*$.

To summarize, the complete hybrid algorithm (HY) works by continuously generating integer partitions, evaluating them, and then computing the aforementioned bounds of their multiset permutations with the aim to prune large portions of the search space. It thus generates the search space representation in a similar fashion as the state-of-the-art does. Then, when a subspace is to be searched, HY uses ADP to search it.

6 Benchmarks and Experiments

In accordance with the state-of-the-art for benchmarking SCSGA algorithms [6], we use *UPD*, *NPD* and *NDCS* for generating difficult problem instances:

- **UPD:** $\boldsymbol{v}(C, t) \sim \boldsymbol{\mathcal{U}}(0, 1)$;
- **NPD:** $\boldsymbol{v}(C, t) \sim \boldsymbol{\mathcal{N}}(1, 0.01)$; and
- **NDCS:** $\boldsymbol{v}(C, t) \sim \boldsymbol{\mathcal{N}}(|C|, \max(|C|, 10^{-9}))$;

for all $C \subseteq A$ and $t \in T$, where $\boldsymbol{\mathcal{U}}(a, b)$ and $\boldsymbol{\mathcal{N}}(\mu, \sigma^2)$ are the uniform and normal distributions, respectively. In our benchmarks, we store these values in an array, and we treat $\boldsymbol{v}$ as a black-box function that can be queried in $\mathcal{O}(1)$.

The result of each experiment was produced by calculating 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*. All tests were conducted with an Intel 7700K CPU and 16GB memory.

6.1 Optimality Benchmarks

We plot the execution time to find optimum when solving problems with 8 alternatives and different numbers of agents in Figure 2. The results show that HY is not as affected by the value distribution as the state-of-the-art algorithm (abbreviated MP) is, and that HY is considerably faster (by many orders of magnitude) compared to all other algorithms in these benchmarks. For example, for 18 agents and NPD, our algorithm finds optima in $\approx 1\%$ of the time that CPLEX (abbreviated CP) and MP needs.

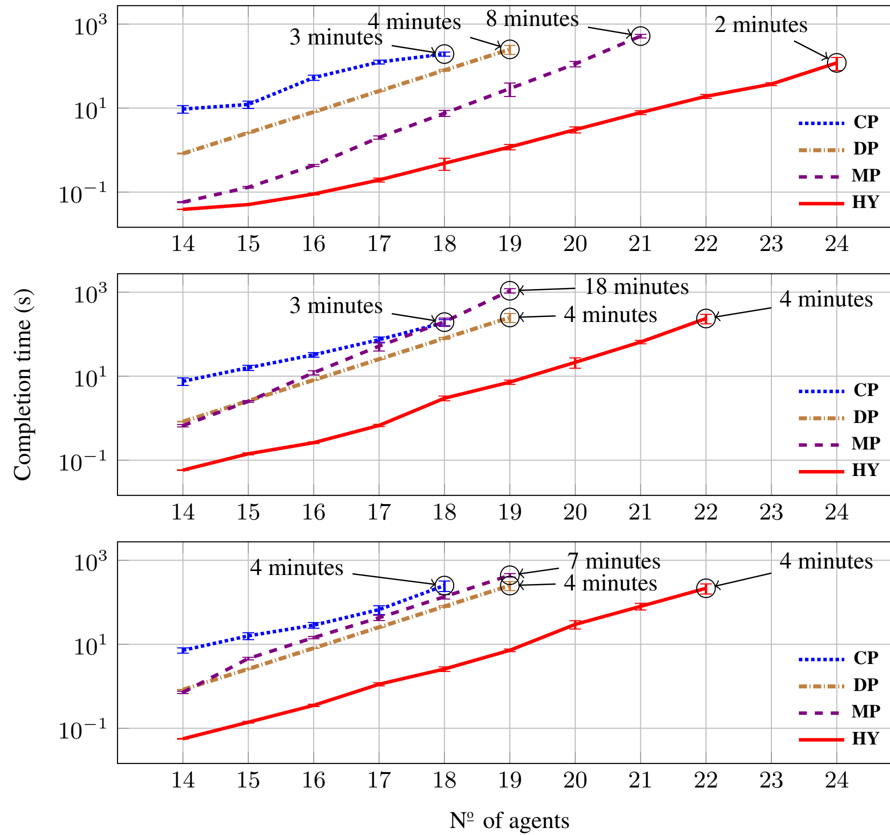

Fig. 2: The completion time (log-scale) for optimally solving problems with 8 alternatives and values generated with **UPD** (top), **NPD** (middle) and **NDCS** (bottom).

Algorithm	$t_{min}(s)$	$t_{max}(s)$	$t_{mean}(s)$	$t_{var}(s^2)$
MP (UPD)	0.02976	0.54829	0.10479	0.00572
HY (UPD)	0.02977	0.05833	0.03606	0.00003
DP (UPD)	0.85720	1.01132	0.86806	0.00017
MP (NPD)	0.04638	3.26065	0.77145	0.49433
HY (NPD)	0.03156	0.22626	0.06060	0.00092
DP (NPD)	0.85622	0.93133	0.86586	0.00013
MP (NDCS)	0.23562	2.11546	0.89745	0.00075
HY (NDCS)	0.04602	0.11447	0.06813	0.00009
DP (NDCS)	0.85946	0.89658	0.86906	0.00007

Fig. 3: Data from optimally solving problem sets with 14 agents and 8 alternatives.

When we ran our optimality benchmarks, we noticed that MP sometimes spent a considerable amount of time searching. To investigate this further, we ran 100 experiments per problem instance and algorithm with $n = 14$ and $m = 8$. We then computed the minimum t_{min}, maximum t_{max}, mean t_{mean} and variance t_{var} for the algorithms' different completion times. The results of these experiments are shown in Figure 3. As expected, they show that HY's execution time varies very little compared to MP's, and that MP's worst-case execution time is much worse than HY's.

Since HY requires additional memory due to using memoization, we investigate how its memoization table grows in the number of agents, and how it is affected by different value distributions. We tested this by keeping track of the aggregated number of entries ($= |\mathit{\Gamma}_w| = |\mathit{\Gamma}_c|$) in the algorithms' memoization tables during runtime. Our results from these experiments are plotted in Figure 4, and show: i) that, at worst-case, HY approximately requires $\approx 10\%$ of the number of entries that DP needs; and ii) that this number indeed depends on the distribution of values and not only on the problem instance's input size—for example, for UPD, HY typically only requires storing 5% of the number of entries that DP needs. These numbers are indicative to HY's ability to discard recursion branches.

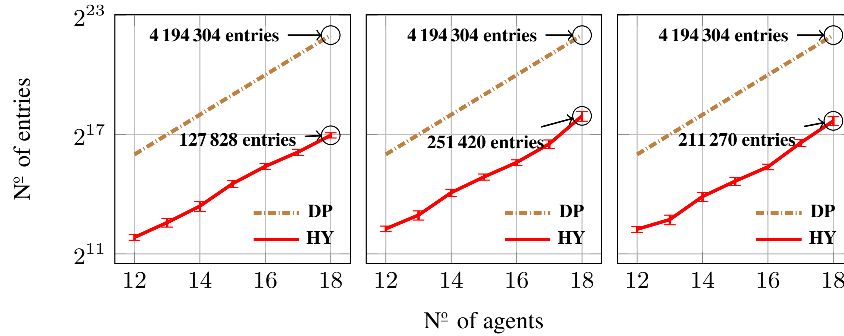

Fig. 4: The total number of entries (log-scale) stored in the memoization tables for problems with 8 alternatives and values generated with **UPD** (left), **NPD** (middle) and **NDCS** (right).

6.2 Anytime Benchmarks

In our next benchmarks, we investigate the quality of the anytime solutions generated by HY (DP is not included, since it is not anytime). To this end, we also benchmark against two simple and easy-to-implement non-optimal algorithms, which results' we use as a worst-case baseline:

- A *random sampling* (RS) algorithm. RS works by randomly (uniformly) assigning every agent to an alternative. Then, when all agents have been assigned, it evaluates the resulting solution's value. It continuously runs this procedure until the time limit has been exceeded, at which point RS returns the best result it has found so far.
- A simple *greedy* (AG) algorithm. AG generates a solution by sequentially assigning agents to alternatives in a greedy fashion.

Moreover, we used 13 agents and 14 tasks for these benchmarks, resulting in a total number of $14^{13} \approx 8 \times 10^{14}$ possible solutions per problem instance. Our results from these experiments are presented in Figure 5, with the execution time 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 "worst" solution—in other words, $\boldsymbol{V}(S^*) = \max_{S \in \Pi_A^m} \boldsymbol{V}(S)$ and $\boldsymbol{V}(S_*) = \min_{S \in \Pi_A^m} \boldsymbol{V}(S)$. Also, note that in these tests, RS generated and evaluated approximately 4.4 million solutions per second; and that for the execution time in these graphs, CPLEX fails to find any feasible (interim) solutions.

As shown by the graphs in Figure 5, HY generates at least 95%-efficient solutions in less than 10 milliseconds for all problem sets with 13 agents and 14 tasks. Moreover, HY found near-optimal 99%-efficient solutions very rapidly for all distributions and benchmarks (e.g., at worst case for NPD, this takes roughly 900 milliseconds). Moreover, compared to MP, it always finds better solutions for the same execution time. Our

anytime benchmarks thus clearly show that HY is extremely fast at finding near-optimal solutions, and that it greatly outperforms the state-of-the-art in generating high-quality interim solutions.

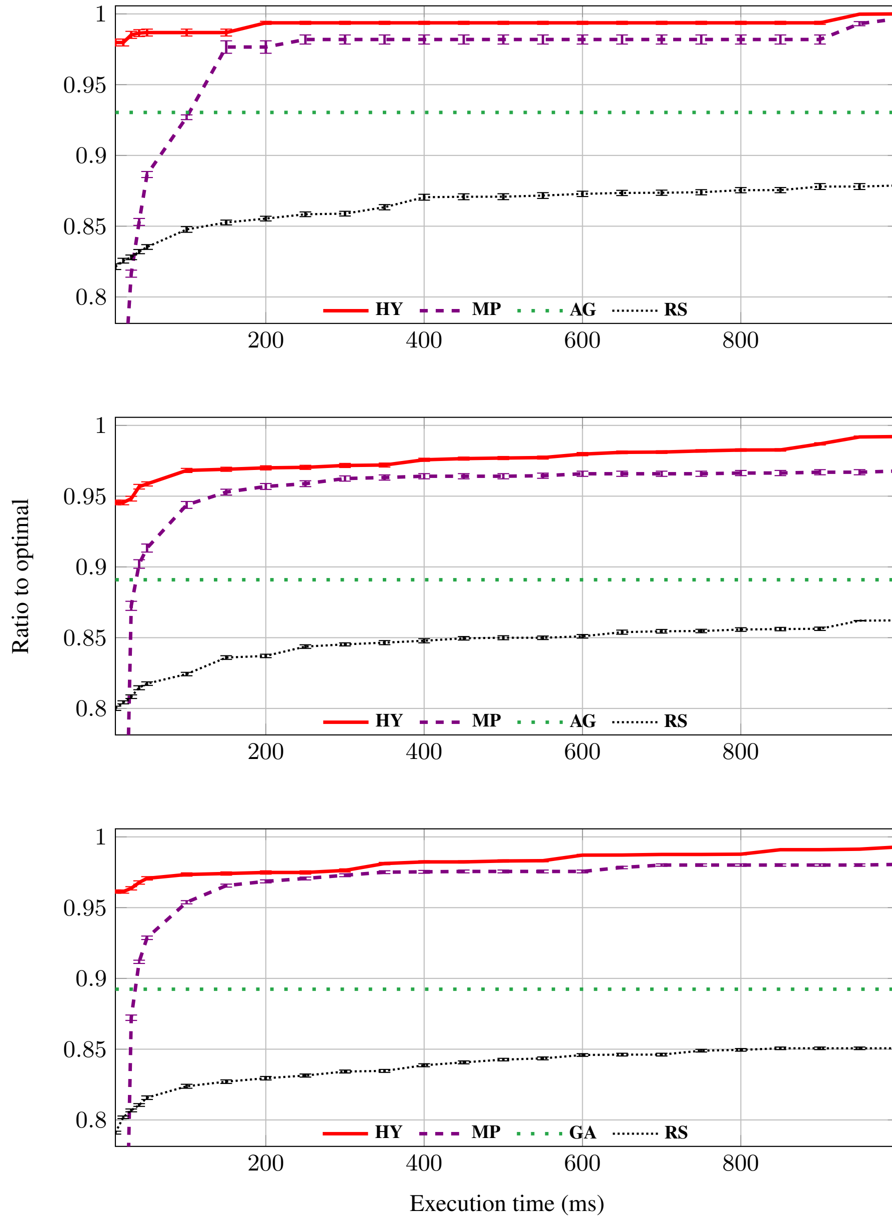

Fig. 5: The normalized ratio to optimal obtained by the different algorithms for problem sets generated using **UPD** (top), **NPD** (middle) and **NDCS** (bottom) with 13 agents and 14 tasks.

7 Conclusions

We presented two different algorithms that use dynamic programming to optimally solve the simultaneous coalition structure generation and assignment problem: one based purely on dynamic programming, and a second hybrid approach that uses dynamic programming together with branch-and-bound. We benchmarked them against the state-of-the-art, and our results show that our hybrid approach greatly outperforms all other methods in all of our experiments (often by many orders of magnitude). For example, for 18 agents, 8 alternatives, and normally distributed values, our algorithm finds an optimum in roughly 3 seconds, while this takes both the industry-grade solver CPLEX and previous state-of-the-art approximately 3 minutes. For future work, we hope to investigate if metaheuristic algorithms, probabilistic search and/or machine learning can be applied to solve large-scale problems with many agents.

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

1. Bolger, E.M.: A value for games with n players and r alternatives. International Journal of Game Theory **22**(4), 319–334 (1993)
2. Kuhn, H.W.: The hungarian method for the assignment problem. Naval Research Logistics (NRL) **2**(1-2), 83–97 (1955)
3. Michalak, T.P., Dowell, A.J., McBurney, P., Wooldridge, M.J.: Optimal coalition structure generation in partition function games. In: European Conference on Artificial Intelligence. pp. 388–392 (2008)
4. Pentico, D.W.: Assignment problems: A golden anniversary survey. European Journal of Operational Research **176**(2), 774–793 (2007)
5. Präntare, F.: Simultaneous coalition formation and task assignment in a real-time strategy game. In: Master thesis (2017)
6. Präntare, F., Heintz, F.: An anytime algorithm for simultaneous coalition structure generation and assignment. In: International Conference on Principles and Practice of Multi-Agent Systems. pp. 158–174 (2018)
7. Präntare, F., Heintz, F.: An anytime algorithm for optimal simultaneous coalition structure generation and assignment. Autonomous Agents and Multi-Agent Systems **34**(1), 1–31 (2020)
8. 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. pp. 1417–1420 (2008)
9. Rahwan, T., Jennings, N.: Coalition structure generation: Dynamic programming meets anytime optimisation. In: Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (2008)
10. Rahwan, T., Michalak, T.P., Jennings, N.R.: A hybrid algorithm for coalition structure generation. In: Twenty-Sixth AAAI Conference on Artificial Intelligence. pp. 1443–1449 (2012)

11. Rahwan, T., Michalak, T.P., Wooldridge, M., Jennings, N.R.: Coalition structure generation: A survey. Artificial Intelligence **229**, 139–174 (2015)
12. 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)
13. 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)
14. Yeh, D.Y.: A dynamic programming approach to the complete set partitioning problem. BIT Numerical Mathematics **26**(4), 467–474 (1986)