Coalition Structure Generation: Dynamic Programming Meets Anytime Optimization

Talal Rahwan and Nicholas R. Jennings
School of Electronics and Computer Science, University of Southampton, Southampton, UK
{tr,nrj}@ecs.soton.ac.uk

Abstract

Coalition structure generation involves partitioning a set of agents into exhaustive and disjoint coalitions so as to maximize the social welfare. What makes this such a challenging problem is that the number of possible solutions grows exponentially as the number of agents increases. To date, two main approaches have been developed to solve this problem, each with its own strengths and weaknesses. The state of the art in the first approach is the Improved Dynamic Programming (IDP) algorithm, due to Rahwan and Jennings, that is guaranteed to find an optimal solution in $O(3^n)$, but which cannot generate a solution until it has completed its entire execution. The state of the art in the second approach is an anytime algorithm called IP, due to Rahwan et al., that provides worst-case guarantees on the quality of the best solution found so far, but which is $O(n^n)$. In this paper, we develop a novel algorithm that combines both IDP and IP, resulting in a hybrid performance that exploits the strength of both algorithms and, at the same, avoids their main weaknesses. Our approach is also significantly faster (e.g. given 25 agents, it takes only 28% of the time required by IP, and 0.3% of the time required by IDP).

Introduction

A key organizational paradigm in multi-agent systems is the coalition-based organization in which members of the same coalition coordinate their activities to achieve the collective's goal(s). When effective, such coalitions can improve the performance of the individual agents and/or the system as a whole, especially when tasks cannot be performed by a single agent or when a group of agents perform the tasks more efficiently. Applications of coalition formation include distributed vehicle routing (Sandholm & Lesser 1997) and multi-sensor networks (Dang *et al.* 2006), among others.

To date, a number of coalition formation algorithms have been developed to determine which of the many potential coalitions should actually be formed. To do so, they typically calculate a value for each coalition, known as the *coalition value*, which provides an indication of the expected outcome that could be derived if that coalition was formed. Then, having computed all the coalition values, the decision about the optimal coalition(s) to form can be taken. One of the main bottlenecks that arise in this formation process is that of *coalition structure generation (CSG)*. Specifically, given the value of every possible coalition, the CSG problem involves partitioning the set of agents into exhaustive and disjoint coalitions so as to maximize the social welfare. Such a partition is called a *coalition structure*. In this context, it is usually assumed that the value of a coalition does not depend on the actions of non-members. Such settings are known as *characteristic function games (CFGs)*. Many (but clearly not all) real-world multi-agent problems happen to be CFGs (Sandholm *et al.* 1999; Sandholm & Lesser 1997). However, finding the optimal coalition structure in a CFG setting has been proved to be NP-complete (Sandholm *et al.* 1999). To combat this complexity, a number of algorithms have been developed in the past few years. These can be classified as follows (Rahwan & Jennings 2008):

1. **Dynamic programming:** Algorithms in this class use dynamic programming techniques to avoid the evaluation of every possible coalition structure. The state of the art in this class of algorithms is the IDP algorithm (Rahwan & Jennings 2008), which improves on the widely-used dynamic programming (DP) algorithm (Yeh 1986).[1] The main advantage of IDP is that, for n agents, it guarantees to find an optimal solution in $O(3^n)$ time. However, the downside is that it only produces a solution when it has completed its entire execution. This means that no partial or interim solutions are available, which is undesirable, especially given large numbers of agents, since the time required to return the optimal solution might be longer than the time available to the agents.
2. **Heuristics:** A number of heuristics have been developed to solve the CSG problem. Shehory and Kraus (1998), for example, proposed a greedy algorithm that puts constraints on the size of the coalitions that are taken into consideration. Sen and Dutta (2000), on the other hand, use an order-based genetic algorithm. Although these algorithms return solutions relatively quickly, they do not provide any guarantees on finding the optimal. In fact, their solutions can always be arbitrarily worse than the optimal, and even if these algorithms happen to find an optimal solution, it is not possible to verify this fact.
3. **Anytime optimal algorithms:** The rationale behind this line of research is that, if the search space is too large to be fully searched, then the alternative does not necessarily have to be applying heuristics that return *"good"* solutions

[1] More details can be found in the next section.

with absolutely no guarantees on their quality. Between these two extremes lies a class of anytime algorithms that generate initial solutions that are guaranteed to be within a bound from the optimal. These algorithms can then improve on the quality of their solutions, and establish progressively better bounds, as they search more of the search space, until an optimal solution is found. Being anytime is important since the search space grows exponentially with the number of agents involved, meaning that the agents might not always have sufficient time to run the algorithm to completion. Moreover, being anytime makes the algorithm more robust against failure; if the execution is stopped before the algorithm would normally have terminated, then it can still provide the agents with a solution that is better than the initial, or any other intermediate, one. The state of the art in this class of algorithms is the IP algorithm (Rahwan *et al.* 2007b), which uses a novel representation of the search space, and uses branch-and-bound techniques that allow the pruning of huge portions of the search space.[2] However, the algorithm is still $O(n^n)$, meaning that it might, in the worst case, end up searching the entire space.

As can be seen, these different classes have their relative strengths and weaknesses. Given this, there is scope to combine methods to try and obtain the best features of the constituent parts. In particular, we focus here on the classes that return optimal solutions or, at least, provide guarantees on the quality of their solutions (i.e. classes 1 and 3). Specifically, we focus on the state of the art in both of these classes (namely IDP and IP). By comparing these two algorithms, we find that IP's advantage over IDP is that it returns solutions anytime, and that it returns an optimal solution much quicker. On the other hand, IDP's advantage over IP lies in the worst-case computational complexity, which is $O(3^n)$ instead of $O(n^n)$. Now, as discussed earlier, each of these properties (i.e. being anytime, returning an optimal solution quickly, and having low complexity) is highly desirable. However, none of the algorithms that are available in the literature has *all* of these properties. In other words, given the current state of the art, the agents have no choice but to give up one of these properties for another. What would be desirable, then, is to develop an algorithm that has all of these properties, even if this meant that the *"quality"* of some of these properties will be slightly affected. For example, compared to IP, it might be more desirable to use an anytime algorithm that performs a pre-processing stage to reduce the number of solutions that could (in the worst case) be examined, even if this meant that the agents would have to wait a little longer before the algorithm can return an initial solution. Moreover, what would be even more desirable is to have full control over this trade-off in quality between different properties.

Against this background, our contribution to the state of the art lies in developing a novel algorithm (called IDP-IP) that does exactly that. As the name suggests, the main idea is to combine IDP with IP, resulting in a hybrid approach in which the agents can control the performance as required, depending on the problem domain. Specifically, this is done by simply adjusting a single parameter (denoted m) to one of the following values: $1, 2, \ldots, \lfloor (2 \times n)/3 \rfloor$; setting m to 1 causes the performance to be identical to IP, and setting it to $\lfloor (2 \times n)/3 \rfloor$ causes the performance to be identical to IDP. More importantly, setting m anywhere between these two extremes determines how much of IP's properties, and how much of IDP's properties, to have in our hybrid approach. In such a case, we expected that the time that IDP-IP requires to return an optimal solution would be somewhere between IDP and IP. However, we were surprised to find that IDP-IP, given a relatively small value of m, is significantly faster than both IDP and IP!

The remainder of this paper is structured as follows. In the next section, we analyse how IDP and IP work. We then detail the IDP-IP hybrid and evaluate its performance.

Analysing IDP and IP

In this section, we start by explaining how DP works, and how IDP improves on DP.[3] After that, we explain how IP works. First, however, we provide some basic notations. In particular, we will denote by A the set of agents, n the number of agents, v the input table (i.e. $v[C]$ is the value of coalition C), and $V(CS)$ the value of coalition structure CS. Moreover, the terms *"coalition structure"* and *"solution"* will be used interchangeably.

The DP algorithm

The way this algorithm works is by computing two tables, namely f_1 and f_2, each with an entry for every possible coalition. In more detail, for every coalition $C \subseteq A$, the algorithm computes $f_1[C]$ and $f_2[C]$ as follows. First, it evaluates all the possible ways of splitting C in two, and compares the highest evaluation with $v[C]$ to determine whether it is beneficial to split C. If so, then the best splitting (i.e. the one with the highest evaluation) is stored in $f_1[C]$, and its evaluation is stored in $f_2[C]$. Otherwise, the coalition itself is stored in $f_1[C]$ and its value is stored in $f_2[C]$.[4] Note that every splitting $\{C', C''\}$ is evaluated as follows: $f_2[C'] + f_2[C'']$. This implies that, in order to evaluate the possible splittings of a coalition C, the algorithm first needs to compute f_2 for the possible subsets of C. Based on this, the algorithm does not evaluate the splittings of the coalitions of size s until it has finished computing f_2 for the coalitions of sizes $1, \ldots, s-1$. Figure 1 shows an example of how f_1 and f_2 are computed given $A = \{1, 2, 3, 4\}$.

Once f_1 and f_2 are computed for every coalition, the optimal coalition structure, denoted CS^*, can be computed recursively. In our example, this is done by first setting $CS^* = \{1, 2, 3, 4\}$. Then, by looking at $f_1[\{1, 2, 3, 4\}]$, we find that it is more beneficial to split $\{1, 2, 3, 4\}$ into $\{1, 2\}$ and $\{3, 4\}$. Similarly, by looking at $f_1[\{1, 2\}]$, we find that it is more beneficial to split $\{1, 2\}$ into $\{1\}$ and $\{2\}$, and by looking at $f_1[\{3, 4\}]$, we find that it is more beneficial to keep $\{3, 4\}$ as it is. The optimal coalition structure is, therefore, $\{\{1\}, \{2\}, \{3, 4\}\}$.

The IDP algorithm

In order to improve on DP, Rahwan and Jennings discovered a link between DP and the *coalition structure graph*, which con-

[2] Again see the next section for more details.

[3] Understanding how IDP improves on DP is crucial since some of the techniques that are used to gain this improvement are similar to those that we use in our hybrid algorithm.

[4] The special case in which C contains only one agent is dealt with separately; in this case, we always have: $f_1[C] = C$ and $f_2[C] = v[C]$.

size	Coalition	The evaluations that are performed before setting f_1 and f_2	f_1	f_2
1	{1}	v[{1}]=30	{1}	30
	{2}	v[{2}]=40	{2}	40
	{3}	v[{3}]=25	{3}	25
	{4}	v[{4}]=45	{4}	45
2	{1,2}	v[{1,2}]=50 f2[{1}]+f2[{2}]=70	{1} {2}	70
	{1,3}	v[{1,3}]=60 f2[{1}]+f2[{3}]=55	{1,3}	60
	{1,4}	v[{1,4}]=80 f2[{1}]+f2[{4}]=75	{1,4}	80
	{2,3}	v[{2,3}]=55 f2[{2}]+f2[{3}]=65	{2} {3}	65
	{2,4}	v[{2,4}]=70 f2[{2}]+f2[{4}]=85	{2} {4}	85
	{3,4}	v[{3,4}]=80 f2[{3}]+f2[{4}]=70	{3,4}	80
3	{1,2,3}	v[{1,2,3}]=90 f2[{1}]+f2[{2,3}]=95 f2[{2}]+f2[{1,3}]=100 f2[{3}]+f2[{1,2}]=95	{2} {1,3}	100
	{1,2,4}	v[{1,2,4}]=120 f2[{1}]+f2[{2,4}]=115 f2[{2}]+f2[{1,4}]=110 f2[{4}]+f2[{1,2}]=115	{1,2,4}	120
	{1,3,4}	v[{1,3,4}]=100 f2[{1}]+f2[{3,4}]=110 f2[{3}]+f2[{1,4}]=105 f2[{4}]+f2[{1,3}]=105	{1} {3,4}	110
	{2,3,4}	v[{2,3,4}]=115 f2[{2}]+f2[{3,4}]=120 f2[{3}]+f2[{2,4}]=110 f2[{4}]+f2[{2,3}]=110	{2} {3,4}	120
4	{1,2,3,4}	v[{1,2,3,4}]=140 f2[{1}]+f2[{2,3,4}]=150 f2[{2}]+f2[{1,3,4}]=150 f2[{3}]+f2[{1,2,4}]=145 f2[{4}]+f2[{1,2,3}]=145 f2[{1,2}]+f2[{3,4}]=150 f2[{1,3}]+f2[{2,4}]=145 f2[{1,4}]+f2[{2,3}]=145	{1,2} {3,4}	150

Figure 1: Example of how the DP algorithm computes f_1 and f_2, given $A = \{1, 2, 3, 4\}$.

sists of a number of nodes and a number of edges connecting these nodes. Specifically, every node in the graph represents a coalition structure, and every edge represents the merging of two coalitions into one (when followed downwards) and the splitting of one coalition into two (when followed upwards). The nodes are categorized, based on the number of coalitions in each node, into levels $L_i : i \in \{1, 2, \ldots, n\}$ such that L_i contains the nodes (i.e. coalition structures) that contain i coalitions. Figure 2 shows an example of 4 agents.

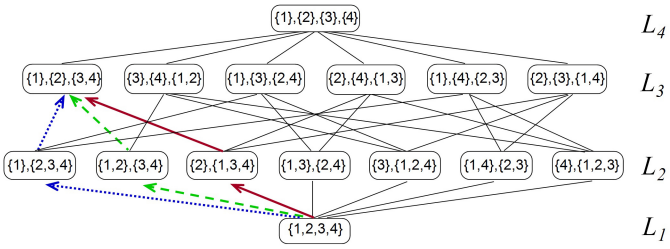

Figure 2: The coalition structure graph of 4 agents.

Looking at this graph, it was possible to visualize how DP works. Basically, given a coalition structure CS, the splitting of a coalition $C \in CS$ into $\{C', C''\}$ can be seen as a movement from the node that contains CS to the node that contains $(CS \backslash C) \cup \{C', C''\}$. The way DP works is by evaluating every possible movement in the graph, and then, starting at the bottom node and moving upwards through a series of connected nodes (which is called a *"path"*) until an optimal node is reached, after which no movement is beneficial. The dashed path in figure 2 shows how DP found the optimal in the previous example (which was shown in figure 1).

From this visualization, it is apparent that, any coalition structure containing more than two coalitions has more than one path leading to it. For example, starting from the bottom node, one could reach $\{\{1\}, \{2\}, \{3, 4\}\}$ through three different paths, which are shown in figure 2 using dotted, dashed, and bold lines. Given this insight, Rahwan and Jennings then showed that, as long as there exists a path leading to the optimal node, DP will be able to find it. Based on this, the basic idea behind IDP was to avoid the evaluation of as many splittings as possible, which corresponds to the removal of as many edges from the graph as possible, while still maintaining at least one path to every node in the graph. In more detail, let $E^{s's''}$ be the set of all the edges that involve the splitting of a coalition of size $(s' + s'')$ into two coalitions of sizes s' and s'', where $s' \leq s''$. Moreover, let E^* be a subset of edges that is defined as follows:

$$E^* = (\bigcup_{s'' \leq n-(s'+s'')} E^{s's''}) \cup (\bigcup_{s'+s''=n} E^{s's''}) \qquad (1)$$

Rahwan and Jennings prove that the edges in E^* are sufficient for every node to have a path leading to it. Based on this, IDP only evaluates those edges, and, in so doing, performs considerably fewer operations than DP. Another improvement of IDP over DP is to significantly reduce the memory requirements (see Rahwan & Jennings 2008).

The IP algorithm

This algorithm uses a novel representation of the search space, which partitions the coalition structures into sub-spaces based on the sizes of the coalitions they contain. In more detail, a sub-space is represented by an integer partition of n.[5] For example, given 4 agents, the possible integer partitions are $[4]$, $[1, 3]$, $[2, 2]$, $[1, 1, 2]$, $[1, 1, 1, 1]$, and each of these represents a sub-space containing all the coalition structures within which the coalition sizes match the parts of the integer partition.[6] For example, $[1, 1, 2]$ represents the sub-space of all the coalition structures within which two coalitions are of size 1, and one coalition is of size 2.

What is interesting about this representation is that, for every sub-space, it is possible to compute upper and lower bounds on the value of the best solution that can be found in it. To this end, let Max_s and Avg_s be the maximum, and the average, value of the coalitions of size s respectively. Also, let $\mathcal{I}(n)$ be the set of integer partitions of n, and S_I be the sub-space that corresponds to the integer partition $I \in \mathcal{I}(n)$. Then, for all $I \in \mathcal{I}(n)$, it is possible to compute an upper bound UB_I on the value of the best solution in S_I as follows: $UB_I = \sum_{s \in I} Max_s$. Similarly, a lower bound LB_I on the value of the best solution in S_I can be computed as follows: $LB_I = \sum_{s \in I} Avg_s$.[7] These bounds are then used to establish worst-case guarantees on the quality of the best solution found so far, and to prune any sub-space that has no potential of containing a solution better than the current best one. As for the remaining sub-spaces, IP searches them one at a time, unless a value is found that is higher than the upper bound of another sub-space, in which case, that sub-space no longer needs to be searched. Searching a sub-space is done using an efficient process that applies branch-and-

[5]Recall that an integer partition of a positive integer number x consists of a set of positive integers (called *"parts"*) that add up to exactly x (Skiena 1998).

[6]We use the square brackets instead of the curly brackets to distinguish between an integer partition and a coalition.

[7]Interestingly enough, Rahwan et al. (2007b) proved that this lower bound is actually the average value of all the solutions in S_I.

bound techniques to avoid examining every solution in it. For more details on the IP algorithm, see (Rahwan *et al.* 2007a; 2007b).

Combining IDP and IP

In this section, we present our hybrid algorithm (IDP-IP) and then evaluate its performance.

The hybrid algorithm

Given the differences between IDP and IP, in terms of the space representation as well as the search techniques that are being used, it is non-trivial to determine how these two algorithms can be combined. We first developed what we call an *integer partition graph*, which is similar to a coalition structure graph, except that every node in L_s now represents an integer partition containing s parts, and every edge now represents a replacement of two parts (i', i'') with one part $(i = i' + i'')$ when followed downwards, and a replacement of one part (i) with two parts $(i', i'' : i' + i'' = i)$ when followed upwards. An example is shown in figure 3(A).

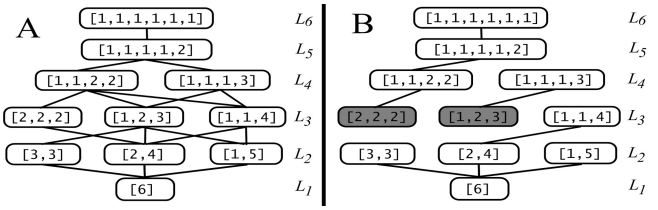

Figure 3: The integer partition graph of 6 agents; (A) shows all the edges in the graph, while (B) shows only those that involve replacing 2 with $1, 1$, or replacing 6 with $1, 5$, or $2, 4$, or $3, 3$.

Looking at this graph enables us to visualize how IDP searches the sub-spaces that are represented by different integer partitions. Basically, by evaluating the possible splittings of every coalition of size s into two coalitions of sizes s' and s'', IDP is actually evaluating the movements (in the integer partition graph) that involve replacing s with s', s''. Similarly, by making a movement from one coalition structure CS_1 to another CS_2, IDP is actually making a movement from one integer partition $(I_1 : CS_1 \in S_{I_1})$ to another $(I_2 : CS_2 \in S_{I_2})$.

Given this link between IDP and IP, we now show how the two algorithms can be combined. Basically, instead of setting IDP to evaluate the possible splittings of the coalitions of size $s \in \{1, 2, \ldots, n\}$, we can set it to evaluate only those of the coalitions of size $s \in \{1, 2, \ldots, m-1, m, n\}$, where $m < n$. As for the coalitions of the remaining sizes, we simply set $f_2[C] = v[C]$. This corresponds to the removal of the edges (in the integer partition graph) that involve replacing an integer $i : m < i < n$ with two smaller ones. As a result, some nodes will no longer have an edge leading to them. Figure 3(B), for example, shows the case where $m = 2$. As can be seen, the nodes that are shown in gray no longer have an edge leading to them. Now if the optimal solution happened to be in a sub-space that is represented by one of those nodes, or by a node that can only be reached through one of those nodes, then IDP will no longer be able to find it. However, by combining IDP with IP, the optimal solution can still be found. This is based on the following two observations:

1. When IP searches through a sub-space S_I, it finds the coalition structure that satisfies: $\arg\max_{CS \in S_I} \sum_{C \in CS} v[C]$.
2. When IDP makes a movement from one coalition structure CS_1 to another CS_2, the following always holds: $\sum_{C \in CS_1} f_2[C] = \sum_{C \in CS_2} f_2[C]$.[8] Moreover, when IDP reaches a coalition structure, say CS_3, after which no movement is beneficial, then the following always holds: $\sum_{C \in CS_3} f_2[C] = \sum_{C \in CS_3} v[C] = V(CS_3)$. This implies that, for every coalition structure CS, the value of the best coalition structure that could be reached from CS can be computed as follows: $\sum_{C \in CS} f_2[C]$. Based on this, the best movement that could be made from a sub-space S_I is the one made from the coalition structure that satisfies: $\arg\max_{CS \in S_I} \sum_{C \in CS} f_2[C]$.

Based on these two observations, if we replace $v[C]$ with $f_2[C]$, then, whenever IP searches through a sub-space S_I, it would find the coalition structure in S_I from which the best movement can be made (i.e. the one that satisfies: $\arg\max_{CS \in S_I} \sum_{C \in CS} f_2[C]$). It is, then, possible to make the required movements from that particular coalition structure using IDP. The result, in this case, will be the best of all the coalition structures that belong to S_I or to any other sub-space S_J (where J is an integer that is reachable from I). Based on this, our hybrid approach consists of the following two main steps:

1. IDP is initially used to compute f_1 and f_2 for the coalitions of size $s \in \{1, \ldots, m, n\}$ and to make the best movements from the bottom node in the integer partition graph (while only taking into consideration the nodes that are still connected to the bottom one).
2. IP and IDP are then iteratively used to search through the remaining sub-spaces. Specifically, for every sub-space S_I such that I has no edge leading to it, IP is used to find the coalition structure in S_I from which the best movements can be taken, and IDP is then used to make those movements.

Applying these two steps ensures that every sub-space is taken into consideration, and that the optimal coalition structure is found. However, in order to apply the second step, we need to first identify the set of nodes that have no edges leading to them, denoted $\tilde{\mathcal{I}}(n)$. Recall that an edge represents the replacement of two integers i', i'' with $i = i' + i''$ when followed downwards. Also recall that such an edge is removed if $m < i' + i'' < n$. Based on this, if we denote by $|I|$ the number of parts within an integer partition I, then the set $\tilde{\mathcal{I}}(n)$ can be defined as follows:

$$\tilde{\mathcal{I}}(n) = \{I \in \mathcal{I}(n) : |I| > 1 \, and \, \forall i', i'' \in I, m < i' + i'' < n\}$$

The sub-spaces that need to be searched by IP would then be $S_I : I \in \tilde{\mathcal{I}}(n)$. Note that Rahwan et al.'s IP algorithm avoids searching a sub-space S_I if UB_I is smaller than the value of the best solution found so far. In our hybrid approach, however, it might still be worthwhile to search S_I since there

[8] This comes from the fact that CS_1 and CS_2 include the same coalitions, except for one coalition in CS_1 (denoted C_1), which is replaced with two coalitions in CS_2 (denoted C_1' and C_1''). Moreover, the fact that IDP has replaced C_1 with C_1', C_1'' implies that it had previously set $f_1[C_1]$ to either C_1' or C_1''. This, in turn, implies that: $f_2[C_1] = f_2[C_1'] + f_2[C_1'']$.

could be a coalition structure in it from which IDP can make particular movements such that another coalition structure is reached that is better than the current best. In other words, the pruning, in our case, needs to be based on an upper bound on the value of the best coalition structure that could be found in S_I or in any other sub-space S_J such that J is reachable from I. We will denote such an upper bound by $\widehat{UB}_I$. Moreover, we will denote by $\widehat{V}(CS)$ the value of the best coalition structure that could be reached from CS. What we need, then, is to compute an upper bound on $\max_{CS \in S_I} \widehat{V}(CS)$. Note, however, that $\widehat{V}(CS) = \sum_{C \in CS} f_2[C]$.[9] Based on this, if we define $\widehat{Max}_s$ as follows:

$$\forall s \in \{1, \ldots, n\} \ : \ \widehat{Max}_s = \max_{C \subseteq A, |C|=s} \ f_2[C]$$

then $\widehat{Max}_s$ can be used to compute $\widehat{UB}_I$ in the same way that Max_s is used to compute UB_I (i.e., $\widehat{UB}_I = \sum_{s \in I} \widehat{Max}_s$). Similarly, if we define $\widehat{Avg}_s$ as follows:

$$\forall s \in \{1, \ldots, n\} \ : \ \widehat{Avg}_s = avg_{C \subseteq A, |C|=s} \ f_2[C]$$

then, a lower bound on $\max_{CS \in S_I} \widehat{V}(CS)$, denoted $\widehat{LB}_I$, can be computed in the following way: $\widehat{LB}_I = \sum_{s \in I} \widehat{Avg}_s$. Note that this lower bound is actually the average of $\widehat{V}(CS) : CS \in S_I$. This comes from the fact that LB_I is the average of the values of all the coalition structures in S_I (Rahwan *et al.* 2007b), which means that the following holds:

$$\sum_{s \in I} avg_{C \subseteq A, |C|=s} v[C] \ = \ avg_{CS \in S_I} \sum_{C \in CS} v[CS] \quad (2)$$

Based on this, if we replace every v in (2) with f_2, we find that: $\widehat{LB}_I = avg_{CS \in S_I} \widehat{V}(CS)$.

What is interesting about IDP-IP is that the performance can be controlled by simply adjusting m. In this context, although the maximum value that m can take is $n-1$, the following theorem implies that it practically makes no difference whether m is set to $\lfloor \frac{2 \times n}{3} \rfloor$ or to any other value that is greater than $\lfloor \frac{2 \times n}{3} \rfloor$.

Theorem 1. IDP does not evaluate any of the possible splittings of a coalition of size $s \in \{\lfloor \frac{2 \times n}{3} \rfloor + 1, \ldots, n-1\}$.[10]

Note that, if we set m to $\lfloor \frac{2 \times n}{3} \rfloor$, then the performance of IDP-IP becomes identical to IDP.[11] On the other hand, if we set m to 1, then the performance becomes identical to IP. More importantly, by setting m anywhere between these two extremes, we can determine how much of IDP, and how much of IP, to use in our hybrid approach.

Performance evaluation

Given 25 agents, and given different values of m, table 1 shows the number of splittings that are evaluated by IDP, and the number of sub-spaces that are pruned (i.e. that no longer need to be searched by IP). As can be seen, given small values of m, IDP evaluates a small number of splittings, and, in return, prunes substantial portions of the search space. For example, by only evaluating 17669915 splittings, which takes 0.2 seconds on our PC, it prunes 1776 sub-spaces, and these contain over 4.6×10^{18} possible coalition structures (which makes 99.6% of the search space).[12] However, as the value of m increases, the number of evaluated splittings increases significantly, and the number of pruned solutions decreases significantly. For example, increasing m from 14 to 15 causes IDP to evaluate an additional 4.7×10^{10} splittings (which takes 2.3 hours) only to prune an additional 0.0000009% of the search space! This is because, when IDP evaluates the possible splittings of every coalition of size s into two coalitions of sizes s' and s'' ($s' + s'' = s$), it prunes every sub-space $S_I : s', s'' \in I$, and the smaller s is, the more likely it is for an integer partition to contain two parts that sum up to s.

m	Num of evaluated splittings	Num of pruned subspaces	Percentage of pruned subspace	Num of pruned solutions	Percentage of pruned solutions
2	16777515	1268	64.8%	3230863621973843192	69.7%
3	16784415	1478	75.5%	4189923777773485292	90.3%
4	16872965	1703	87.0%	4564993844653940392	98.4%
5	17669915	1776	90.7%	4619890309966821892	99.6%
6	23160015	1843	94.1%	4635658723330786692	99.9%
7	53444115	1874	95.7%	4637889492635473692	>99.9%
8	190804140	1900	97.0%	4638465143123614992	>99.9%
9	711762765	1915	97.8%	4638563634445793142	>99.9%
10	2382099125	1927	98.4%	4638581351908188038	>99.9%
11	6942019325	1935	98.8%	4638588823705171238	>99.9%
12	17587033425	1942	99.1%	4638590208602264538	>99.9%
13	38882261925	1948	99.5%	4638590229863691088	>99.9%
14	74924798325	1953	99.7%	4638590265094980688	>99.9%
15	122135499005	1956	99.9%	4638590307397638228	>99.9%
16	158653677130	1958	100%	4638590332229999353	100%

Table 1. Given 25 agents, the figure shows the number of evaluated splittings, as well as the number of pruned coalition structures, when running IDP up to $m = 2, \ldots, 16$.

Note that we are the first to provide such an understanding of how the dynamic programming approach works. This is because this approach has only been used in the literature to find the optimal coalition structure, in which case the main concern is usually the total number of operations that it requires to run to completion. In contrast, we show that this approach can also be used to prune parts of the search space, and that, even if the algorithm is not run to completion, it can still be extremely useful. In fact, when setting m to a relatively small value, IDP becomes considerably more efficient than IP at pruning the search space. This fact is reflected in figure 4, which shows (on a log scale) the time required to run IDP-IP given 25 agents.[13] The figure also shows the time required by IP (when $m = 1$) and IDP (when $m = 16$). Here, the algorithm was tested against the four value distributions that are widely used in the CSG literature (Larson & Sandholm 2000; Rahwan *et al.* 2007b): normal, uniform, sub-additive, and super-additive. As can be seen, given small values of m, IDP-IP always outperforms both IDP and IP (e.g., given a normal distribution, IDP-IP takes only 28% of the time required by IP, and 0.3% of that required by IDP, and that is when $m = 7$). Given other value distributions that reduce IP's capability to prune the space, we believe that IDP-IP could outperform IP

[9] This comes from our second observation which was mentioned earlier in this section.

[10] The proof is provided in the appendix.

[11] This is because step 1 of IDP-IP would involve the evaluation of every splitting in E^*. As a result, every node in the integer partition graph will have an edge leading to it (i.e. IP will not be used).

[12] The PC on which we ran our simulations had 4 processors (each is an Intel(R) Xeon(R) CPU @ 2.66 GHz), with 3GB of RAM.

[13] Similar results were observed given different numbers of agents.

by orders of magnitude, and that is mainly because the improvements that are obtained by initially running IDP (i.e. the portions of the space that are pruned) remain the same given any value distribution.

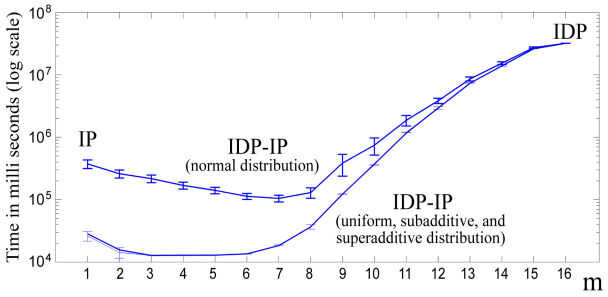

Figure 4: Given 25 agents, and given different values of m, the figure shows (on a log scale) the time required for IDP-IP to return the optimal solution.

Conclusions

In this paper, we combine the state of the art in two of the main approaches to the CSG problem, namely IDP and IP, resulting in a hybrid approach that significantly outperforms both of them. Specifically, this hybrid initially uses dynamic programming to prune the search space, and then uses an efficient anytime search procedure to focus on the most promising regions of the space. Moreover, depending on the preferences of the user, as well as the context of the search, the relative time spent on each of the two components (i.e. IDP and IP) can be adjusted to give the best possible performance. For future work, there are two main avenues of further work. First, we would like IDP-IP to automatically tune its m parameter so that it can determine when to switch from IDP to IP at runtime. Second, we would like to examine the worst-case complexity of IDP-IP. In particular, we would like to determine how this varies between $O(n^n)$ and $O(3^n)$ as the value of m increases from 1 to $\lfloor \frac{2\times n}{3} \rfloor$.

References

Dang, V. D.; Dash, R. K.; Rogers, A.; and Jennings, N. R. 2006. Overlapping coalition formation for efficient data fusion in multi-sensor networks. In *AAAI-06*, 635–640.

Larson, K., and Sandholm, T. 2000. Anytime coalition structure generation: an average case study. *J. Exp. and Theor. Artif. Intell.* 12(1):23–42.

Rahwan, T., and Jennings, N. R. 2008. An improved dynamic programming algorithm for coalition structure generation. In *AAMAS-08*.

Rahwan, T.; Ramchurn, S. D.; Dang, V. D.; and Jennings, N. R. 2007a. Near-optimal anytime coalition structure generation. In *IJCAI-07*, 2365–2371.

Rahwan, T.; Ramchurn, S. D.; Giovannucci, A.; Dang, V. D.; and Jennings, N. R. 2007b. Anytime optimal coalition structure generation. In *AAAI-07*, 1184–1190.

Sandholm, T. W., and Lesser, V. R. 1997. Coalitions among computationally bounded agents. *Artificial Intelligence* 94(1):99–137.

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

Sen, S., and Dutta, P. 2000. Searching for optimal coalition structures. In *Proceedings of the Fourth International Conference on Multiagent Systems*, 286–292.

Shehory, O., and Kraus, S. 1998. Methods for task allocation via agent coalition formation. *Artificial Intelligence* 101(1–2):165–200.

Skiena, S. S. 1998. *The Algorithm Design Manual*. New York, USA: Springer.

Yeh, D. Y. 1986. A dynamic programming approach to the complete set partitioning problem. *BIT Numerical Mathematics* 26(4):467–474.

Appendix: Proof of Theorem 1

We need to prove that any edge involving the splitting of a coalition of size $s \in \{\lfloor \frac{2\times n}{3} \rfloor + 1, \ldots, n-1\}$ into two coalitions of sizes s', s'' (where $s' \leq s''$) does not belong to E^*. From (1), we can see that it is sufficient to prove that $s'' > n - (s' + s'')$ for all s', s''. Note that $s' + s'' = s$, and that $s'' \geq \lceil \frac{s}{2} \rceil$ (this comes from the facts that $s' \leq s''$ and $s' + s'' = s$). Based on this, what we need to prove is the following:

$$\forall s \in \{\lfloor \tfrac{2\times n}{3} \rfloor + 1, .., n\}, \forall s'' \in \{\lceil \tfrac{s}{2} \rceil, .., s-1\} \; : \; s'' > n - s. \quad (3)$$

Note that, if the inequality in (3) holds for the smallest possible value of s'' (which minimizes the left hand side of the inequality), then it must hold for every possible value of s''. Similarly, if the inequality holds for the smallest possible value of s (which maximizes the right hand side) then it must hold for every value of s. Therefore, it is sufficient to prove that the inequality in (3) holds for *the smallest* possible values of s, s''. In other words, it is sufficient to prove that:

$$\lceil (\lfloor \tfrac{2\times n}{3} \rfloor + 1)/2 \rceil > n - (\lfloor \tfrac{2\times n}{3} \rfloor + 1) \quad (4)$$

This can be proved by mathematical induction. In more detail, we will prove that, if equation (4) is true for n, then it is also true for $(n+3)$. After that, we will show that equation (4) is true for $n = 3$, $n = 4$, and $n = 5$. Specifically, assuming that equation (4) is true for n, we will prove that:

$$\lceil (\lfloor \tfrac{2\times (n+3)}{3} \rfloor + 1)/2 \rceil > (n+3) - (\lfloor \tfrac{2\times (n+3)}{3} \rfloor + 1) \quad (5)$$

This can be done as follows:

$$\begin{aligned}
\lceil (\lfloor \tfrac{2\times (n+3)}{3} \rfloor + 1)/2 \rceil &= \lceil (\lfloor \tfrac{2\times n}{3} + 2 \rfloor + 1)/2 \rceil \\
&= \lceil (\lfloor \tfrac{2\times n}{3} \rfloor + 2 + 1)/2 \rceil \\
&= \lceil (\lfloor \tfrac{2\times n}{3} \rfloor + 1)/2 \rceil + 1 \\
&> n - (\lfloor \tfrac{2\times n}{3} \rfloor + 1) + 1 \quad \text{(because of (4))} \\
&= n - \lfloor \tfrac{2\times n}{3} \rfloor
\end{aligned}$$

Moreover, we have:

$$\begin{aligned}
(n+3) - (\lfloor \tfrac{2\times (n+3)}{3} \rfloor + 1) &= n + 3 - (\lfloor \tfrac{2\times n}{3} + 2 \rfloor + 1) \\
&= n + 3 - (\lfloor \tfrac{2\times n}{3} \rfloor + 2 + 1) \\
&= n - \lfloor \tfrac{2\times n}{3} \rfloor
\end{aligned}$$

This means that (5) is true whenever (4) is true. Now, all we need to show is that (4) is true for $n = 3$, $n = 4$, and $n = 5$:

- For $n = 3$, $\lceil (\lfloor \frac{2\times n}{3} \rfloor + 1)/2 \rceil = 2$ and $n - (\lfloor \frac{2\times n}{3} \rfloor + 1) = 0$
- For $n = 4$, $\lceil (\lfloor \frac{2\times n}{3} \rfloor + 1)/2 \rceil = 2$ and $n - (\lfloor \frac{2\times n}{3} \rfloor + 1) = 1$
- For $n = 5$, $\lceil (\lfloor \frac{2\times n}{3} \rfloor + 1)/2 \rceil = 2$ and $n - (\lfloor \frac{2\times n}{3} \rfloor + 1) = 1$