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). 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.
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. However, the algorithm is still $O(n^m)$, 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 $O(n^m)$. 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. Then, 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 $f_2[C]$.

Note that every coalition splitting $C$, $C'$, $C''$ 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-
ent paths, which are shown in figure 2 using dotted, dashed
nodes, one could reach one path leading to it. For example, starting from the bottom structure containing more than two coalitions has more than the previous example (which was shown in figure 1).

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 \( n \). For example, given 4 agents, the possible integer partitions are \([4] \), \([1, 3] \), \([2, 2] \), \([1, 1, 1, 1] \), 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. 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 \( \{n\} \) be the set of integer partitions of \( n \), and \( S_I \) be the sub-space that corresponds to the integer partition \( I \in \{n\} \). Then, for all \( I \in \{n\} \), it is possible to compute an upper bound \( UB_I \) on the value of the best solution in \( S_I \) as follows: \( UB_I = \sum_{I \in \{n\}} Max_s \). Similarly, a lower bound \( LB_I \) on the value of the best solution in \( S_I \) can be computed as follows: \( LB_I = \sum_{I \in \{n\}} Avg_s \). 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-

3Recall 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).
4We use the square brackets instead of the curly brackets to distinguish between an integer partition and a coalition.
5Interestingly enough, Rahwan et al. (2007b) proved that this lower bound is actually the average value of all the solutions in \( S_I \).
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).

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 splitings 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 $C_{S_1}$ to another $C_{S_2}$, IDP is actually making a movement from one integer partition $(I_1 : C_{S_1} \in S_{I_1})$ to another $(I_2 : C_{S_2} \in S_{I_2})$.

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 splitings 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 subspace 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: $\arg \max_{C_{S \in S_{I}}} \sum_{C \in S_{I}} v[C]$.
2. When IDP makes a movement from one coalition structure $C_{S_1}$ to another $C_{S_2}$, the following always holds: $\sum_{C \in S_{I}} f_2[C] = \sum_{C \in S_{I}} f_2[C].$ Moreover, when IDP reaches a coalition structure, say $C_{S_3}$, after no movement is beneficial, then the following always holds: $\sum_{C \in S_{I}} f_2[C] = \sum_{C \in S_{I}} v[C] = V(C_{S_3})$. This implies that, for every coalition structure $C_{S}$, the value of the best coalition structure that could be reached from $C_{S}$ can be computed as follows: $\sum_{C \in S_{I}} f_2[C]$. 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: $\arg \max_{C_{S \in S_{I}}} \sum_{C \in S_{I}} f_2[C]$.

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: $\arg \max_{C_{S \in S_{I}}} \sum_{C \in S_{I}} f_2[C]$). 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 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 $\bar{I}(n)$. 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 $|\bar{I}(n)|$ the number of parts within an integer partition $I$, then the set $\bar{I}(n)$ can be defined as follows:

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

The sub-spaces that need to be searched by IP would then be $S_I : I \in \bar{I}(n)$. Note that Rahwan et al.'s IP algorithm avoids searching a sub-space $S_I$ if $U \cap I$ 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

---

*This comes from the fact that $C_{S_1}$ and $C_{S_2}$ include the same coalitions, except for one coalition in $C_{S_1}$ (denoted $C_1$), which is replaced with two coalitions in $C_{S_2}$ (denoted $C_1'$ and $C_1''$). Moreover, the fact that IDP has replaced $C_1$ with $C_1'$ and $C_1''$ 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 \( S_J \) or in any other sub-space \( S_J \) such that \( J \) is reachable from \( I \). We will denote such an upper bound by \( \bar{V}B_J \). Moreover, we will denote by \( \bar{V}(CS) \) the upper bound of the value of the best coalition structure that could be reached from \( CS \). What we need, then, is to compute an upper bound on \( \max CS \in S_I \bar{V}(CS) \). Note, however, that \( \bar{V}(CS) = \sum CS \in S f_2[C] \). Based on this, if we define \( \bar{Max} \) as follows:

\[
\forall s \in \{1, \ldots, n\} : \bar{Max}_s = \max CS \subseteq A, |C| = s f_2[C]
\]

then \( \bar{Max}_s \) can be used to compute \( \bar{UB}_I \) in the same way that \( Max_s \) is used to compute \( UB_I \) (i.e., \( \bar{UB}_I = \sum s \in I \bar{Max}_s \)).

Similarly, if we define \( \bar{Avg} \) as follows:

\[
\forall s \in \{1, \ldots, n\} : \bar{Avg}_s = \frac{\sum CS \subseteq A, |C| = s f_2[C]}{s}
\]

then, a lower bound on \( \max CS \in S_I \bar{V}(CS) \), denoted \( \bar{LB}_I \), can be computed in the following way: \( \bar{LB}_I = \sum s \in I \bar{Avg}_s \).

Note that this lower bound is actually the value of \( \bar{V}(CS) \) : \( CS \in S_I \). This comes from the fact that \( \bar{LB}_I \) is the average of the values of all the coalition structures in \( S_I \) (Rahwan et al. 2007).

Note that \( \bar{Max}_s \) and \( \bar{Avg}_s \) are upper bounds on the values of the best coalition structures (which makes \( \bar{V}(CS) \) over-estimate the number of possible coalition structures). This is because step 1 of IDP-IP would involve the evaluation of the possible splitting of a coalition of size \( s \) into two coalitions of sizes \( s' \) and \( s'' \) (\( s' + s'' = s \)), it prunes every sub-space \( S_J \) \( : s', s'' \in I \), and the smaller \( s \) is, the more likely it is for an integer partition to contain two parts that sum up to \( s \).

<table>
<thead>
<tr>
<th>( m )</th>
<th>Num of evaluated splittings</th>
<th>Num of pruned subspaces</th>
<th>Percentage of pruned subspaces</th>
<th>Num of pruned solutions</th>
<th>Percentage of pruned solutions</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>16777515</td>
<td>1268</td>
<td>64.8%</td>
<td>3230863621587 314836129</td>
<td>69.7%</td>
</tr>
<tr>
<td>3</td>
<td>16784415</td>
<td>1478</td>
<td>76.5%</td>
<td>416891377777544329</td>
<td>90.3%</td>
</tr>
<tr>
<td>4</td>
<td>16879265</td>
<td>1703</td>
<td>87.0%</td>
<td>46549934469540192</td>
<td>98.4%</td>
</tr>
<tr>
<td>5</td>
<td>27669915</td>
<td>1776</td>
<td>90.7%</td>
<td>4689609309696812892</td>
<td>99.8%</td>
</tr>
<tr>
<td>6</td>
<td>27160015</td>
<td>1840</td>
<td>94.4%</td>
<td>4626586723830896699</td>
<td>99.9%</td>
</tr>
<tr>
<td>7</td>
<td>53440115</td>
<td>1874</td>
<td>95.7%</td>
<td>46378497635479692</td>
<td>99.9%</td>
</tr>
<tr>
<td>8</td>
<td>100604410</td>
<td>1900</td>
<td>97.0%</td>
<td>46284614312614992</td>
<td>99.9%</td>
</tr>
<tr>
<td>9</td>
<td>71126765</td>
<td>1915</td>
<td>97.8%</td>
<td>463856364445793142</td>
<td>99.9%</td>
</tr>
<tr>
<td>10</td>
<td>280299125</td>
<td>1937</td>
<td>98.5%</td>
<td>4628581551308540858</td>
<td>99.9%</td>
</tr>
<tr>
<td>11</td>
<td>940201375</td>
<td>1931</td>
<td>98.8%</td>
<td>463858823705171238</td>
<td>99.9%</td>
</tr>
<tr>
<td>12</td>
<td>1578033425</td>
<td>1942</td>
<td>99.1%</td>
<td>462859028602264358</td>
<td>99.9%</td>
</tr>
<tr>
<td>13</td>
<td>3888262195</td>
<td>1946</td>
<td>99.3%</td>
<td>462859029699299388</td>
<td>99.9%</td>
</tr>
<tr>
<td>14</td>
<td>7492479125</td>
<td>1959</td>
<td>99.7%</td>
<td>4628590296104948068</td>
<td>99.9%</td>
</tr>
<tr>
<td>15</td>
<td>132215499950</td>
<td>1966</td>
<td>99.9%</td>
<td>462859010779638328</td>
<td>99.9%</td>
</tr>
</tbody>
</table>

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 \( m = 2, \ldots, 15 \).

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. 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. 2007): uniform, normal, sub-additive, and super-additive. As can be seen, given small values of \( m \), IDP-IP always outperforms both IP and IDP (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.

---

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

10The proof is provided in the appendix.

11This 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).

12The 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.

13Similar 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.](image)

**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 $\left\lfloor \frac{2x^m}{3} \right\rfloor$.

**References**


**Appendix: Proof of Theorem 1**

We need to prove that any edge involving the splitting of a coalition of size $s \in \{\left\lfloor \frac{2x^m}{3} \right\rfloor + 1, \ldots, n - 1\}$ 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 $s'' \geq \left\lceil \frac{s}{2} \right\rceil$ (this comes from the facts that $s' \leq s''$ and $s' + s'' = s$). Based on this, what we need to prove is the following:

$$\forall s \in \{\left\lfloor \frac{2x^n}{3} \right\rfloor + 1, \ldots, n\}, \forall s'' \in \{\left\lceil \frac{s}{2} \right\rceil, \ldots, s - 1\} : s'' > n - s. \quad (3)$$

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:

$$\left\lfloor (\left\lfloor \frac{2x^n}{3} \right\rfloor + 1) / 2 \right\rfloor > n - (\left\lfloor \frac{2x^n}{3} \right\rfloor + 1) \quad (4)$$

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:

$$\left\lfloor (\left\lfloor \frac{2x(n+3)^3}{3} \right\rfloor + 1) / 2 \right\rfloor > (n+3) - (\left\lfloor \frac{2x(n+3)}{3} \right\rfloor + 1) \quad (5)$$

This can be done as follows:

$$\left\lfloor (\left\lfloor \frac{2x(n+3)^3}{3} \right\rfloor + 1) / 2 \right\rfloor = \left\lfloor (\left\lfloor \frac{2x(n+3)^3}{3} \right\rfloor + 2 + 1) / 2 \right\rfloor$$

$$= \left\lfloor (\left\lceil \frac{2x(n+3)}{3} \right\rceil + 1) / 2 \right\rfloor + 1$$

$$= n - (\left\lfloor \frac{2x(n+3)}{3} \right\rfloor + 1) + 1 \quad \text{(because of (4))}$$

$$= n - \left\lfloor \frac{2x(n+3)}{3} \right\rfloor \quad \text{(5)}$$

Moreover, we have:

$$(n + 3) - \left(\left\lfloor \frac{2x(n+3)}{3} \right\rfloor + 1\right) = n + 3 - (\left\lfloor \frac{2x(n+3)}{3} \right\rfloor + 1)$$

$$= n + 3 - \left(\left\lfloor \frac{2x(n+3)}{3} \right\rfloor + 2 + 1\right)$$

$$= n - \left\lfloor \frac{2x(n+3)}{3} \right\rfloor \quad \text{(5)}$$

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 $n = 3, \left\lfloor (\left\lfloor \frac{2x}{3} \right\rfloor + 1) / 2 \right\rfloor = 2$ and $n - (\left\lfloor \frac{2x}{3} \right\rfloor + 1) = 0$
- For $n = 4, \left\lfloor (\left\lfloor \frac{2x}{3} \right\rfloor + 1) / 2 \right\rfloor = 2$ and $n - (\left\lfloor \frac{2x}{3} \right\rfloor + 1) = 1$
- For $n = 5, \left\lfloor (\left\lfloor \frac{2x}{3} \right\rfloor + 1) / 2 \right\rfloor = 2$ and $n - (\left\lfloor \frac{2x}{3} \right\rfloor + 1) = 1$