Finding Fair Allocations for the Coalition Problem with Constraints

Evan Tick

CIS-TR-95-13
June 1995

Abstract

Fair allocation of payoffs among cooperating players who can form various coalitions of differing utilities is the classic game theoretic "coalition problem." Shapley's value is perhaps the most famous fairness criterion. In this paper, a new allocation principle is proposed based on constraints. Initially constraints are defined by the coalition payoffs, in the standard way. The algorithm proceeds by "tightening" the constraints in a fair manner until the solution space is sufficiently small. An arbitrary boundary point is then chosen as the "fair" allocation. The proposed technique was implemented in the constraint language CLP(R) and evaluated against Shapley values for various benchmark problems. We show axiomatically and empirically how our method coincides with Shapley-fairness.

DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE
UNIVERSITY OF OREGON

Contents

1 Introduction

The standard *coalition problem* in game theory is formalized by assigning each subset of group members a utility. Usually this utility is simply a real number representing the value of the coalition. Normally, the coalition comprising the entire group has the greatest utility. In this case, we assume that all players cooperate, receiving the highest payment. The problem is to divide this payment among the players in a "fair" way.

Raiffa [5] shows, for the three-party coalition problem, how to convert the utilities into a linear program. A variable in these constraints corresponds to the fair payment to be received by an individual player. An n-player game defines at most $2^n - 1$ constraints composed of n unique variables. Each constraint defines the payoff bound of a unique potential coalition. The $2^n - 1$ constraints derive from the $2^n - 1$ subsets within a set of n elements. Each subset defines a potential coalition.

One of the constraints, corresponding to the entire group, is an *equality*: we assume the players will cooperate and take this highest payment. The other constraints are *inequalities*, e.g., $X+Y \geq 33$ represents that players X and Y as a coalition of two would receive 33, so that any fair allocation of the group utility must ensure that the sum of X and Y's utilities exceeds 33. In the games we consider, no coalition payoff can be negative.

If the constraint set has no solution, i.e., is inconsistent, it indicates that the individual coalition constraints are too restrictive to allow any fair allocation. Although certain allocation techniques such as Shapley values produce a solution even if the constraints are inconsistent, in this paper we focus only on consistent problems.

If the constraint set is consistent, then it represents the convex hull of a multi-dimensional solution space to the problem. In other words, all points *within* and *on* the hull represent feasible allocations of the group payment among the cooperating players. If the constraints were given to a linear programming system, an objective function could be specified to optimize over the perimeter of the space. Unfortunately, it is highly likely that perimeter values are "unfair" in the sense that certain players feel cheated because of the gap between what they are receiving and what they could receive if they joined a smaller coalition. Such players might drop out of the union, leaving the Pareto frontier. Intuitively, fair solutions are *within* the space, and therefore not optimizations of any objective function in a linear programming sense.

Raiffa [5] and Shapley [4] suggest heuristics for finding a fair internal point. In fact, their algorithms do not involve the constraint space per se: they compute the fair solution from the original data, i.e., the coalition payoffs. Thus, as mentioned above, they can produce an allocation even for an inconsistent problem with no rational agreement point.

In this paper we show an alternative definition for a fair allocation under the condition that the problem is consistent. This is accomplished by formulating the linear constraints as previously described and implementing them in a constraint programming language, CLP(R) [2]. The constraints are then artificially tightened in an incremental, fair manner. Intuitively, we assume that each coalition payoff is worth *more* than it actually is, slowly reducing the size of the solution space. As we tighten the constraints, we eventually produce a very small solution space that contains only solutions acceptable to all players. At this time, we arbitrarily choose a solution to present as the answer.[1]

This technique is practical with constraint programming languages which allow quick manipulations of complex constraints. The advantage of the technique is that it can potentially be extended to produce solutions where Shapley values and other techniques based on average marginal utility cannot.

[1]The definition of "small space" is made indirectly with a parameter ϵ that defines the termination condition of the tightening procedure, as discussed in Section 2.

This paper is organized as follows. The proposed algorithm is formally defined in Section 2. Section 5 reviews Shapley's axioms and discusses how fair our technique is. A CLP(R) implementation is summarized in Section 4. Section 6 illustrates the technique for the Scandinavian Cement Problem from Raiffa [5]. Section 7 presents empirical benchmark analysis comparing the proposed technique to Shapley's. Conclusions and future work are summarized in Section 8. The appendix lists the source of the CLP(R) implementation of the proposed algorithm (as well as Shapley's algorithm).

2 Proposed Algorithm

In this section we describe the allocation algorithm a bit more formally. Consider n-player games where variable X_i denotes the allocated payoff to player $1 \leq i \leq n$ from the (cooperative) group payoff. We restrict $X_i \geq 0$, i.e., we don't allow negative shares.

The algorithm inputs are a set T of inequalities describing the proper-subset coalition payoffs, and an equality G describing the cooperative group payoff. As explained in the previous section, $0 \leq |T| \leq 2^n - 2$. Assume that constraints in T are normalized in the form $\sum_{k \in Z^n} X_k \geq \text{RHS}_j$, where RHS_j is some real, nonnegative constant, for $1 \leq j \leq |T|$. Here the sum encompasses the collection of players in coalition j.

We define a tightening vector $\overline{K}$ of constants, and

$$T(\overline{K}) = \{ \sum_{k \in Z^n} X_k \geq \gamma(\text{RHS}_j, K_j) \mid 1 \leq j \leq |T| \}$$

where $\gamma()$ is called the fair tightening function. We defer defining this function until Section 5. Let $S(\overline{K}) = T(\overline{K}) \cup G$.

Figure 1 gives the pseudo-code of the algorithm. The arguments of the algorithm are the parameterized constraints $S()$, the tightening vector $\overline{K}$, a bit vector $\overline{F}$, a termination granularity ϵ, and an increment Δ_{max}. Δ_{max} is chosen as a sufficiently large real number. We used $(\Delta_{max}, \epsilon) = (1000, 0.0001)$ in the tests conducted in Section 7. The output of the algorithm is a refined $\overline{K}$.

The bit vector $\overline{F}$ has a '1' to represent that the corresponding tightening parameter is nonfixed, a '0' to represent that it is fixed. The outer loop of the algorithm finds the tightest $\overline{K}$ such that $S(\overline{K})$ is satisfied. Phase I finds the largest consistent increment for all nonfixed tightening parameters. Phase II then fixes critical parameters that previously limited the tightening search. This loop (containing phases I and II) iterates until all parameters are fixed. Phase III then finds an arbitrary solution within this space by conditional conversion of inequalities to equalities.

Some clarifications about the algorithm are in order. The ratio ϵ/Δ is the percentage variation that is acceptable. Furthermore note that arbitrary solutions are at most $2\epsilon\sqrt{n}$ apart in the space, the usual Euclidean distance. Thus we can tune the search with ϵ.

Considering the time complexity of the proposed algorithm, let m be the number of legal coalitions (including the entire group). Input size m is bounded by $2^n - 1$ for an n-player game. In the worst case, we need to iterate m times, wherein each iteration only one coalition is tightened. Within each iteration, we need to make $log_2(\Delta/\epsilon)$ search steps. Each search step we need to solve m linear inequations. Let's call this cost $P(m)$ which is polynomial for some solution algorithms, and exponential worst-case for the Simplex method, but polynomial average case. Thus the oworst-case time complexity order is $O(mP(m)log_2(\Delta_{max}/\epsilon))$.

Thus the complexity is exponential in the number of players because of the explosion of coalitions that can be formed. A naive implementation of Shapley's algorithm has a complexity $O(n \cdot n!)$ because it requires computing the marginal utility for each player for each permutation of the n player coalitions. See Owen [4] for discussion of special cases wherein Shapley values can be computed more efficiently.

```
search( $S()$, $\overline{K}$, $\overline{F}$, $\epsilon$, $\Delta_{max}$ )
    initialize $\overline{K} =< 0, 0, ..., 0 >$
               $\overline{F} =< 1, 1, ...1 >$
    while $(\overline{F} \neq \overline{0})$ do {
        $\Delta = \Delta_{max}$
        while $(\Delta > \epsilon)$ do {                                   (I)
            $\overline{K} = \overline{K} + \Delta\overline{F}$
            if $\neg$sat($S(\overline{K})$) then
                $\overline{K} = \overline{K} - \Delta\overline{F}$
            $\Delta = \Delta/2$
            }
        for { $\forall j \mid 1 \leq j \leq |S|$ } do {                       (II)
            $\overline{R} = \overline{K}$
            $R_j = K_j + 2\epsilon$
            $F_j = \begin{cases} 1 & \text{if sat}(S(\overline{R})) \\ 0 & \text{otherwise} \end{cases}$
            }
        }
    for { $\forall j \mid 1 \leq j \leq |S|$ } do {                           (III)
        $R(\overline{K}) = S(\overline{K}) \setminus \sum_{k \in Z^n} X_k \geq \gamma(\text{RHS}_j, K_j)$
                        $\cup \sum_{k \in Z^n} X_k = \gamma(\text{RHS}_j, K_j)$
        if sat( $R(\overline{K})$ ) then
            $S() = R()$
        }
```

Figure 1: Constraint Tightening Algorithm

3 Review of CLP(R)

CLP(R) is constraint logic programming language over the domain of real arithmetic [1]. Programs appear in syntax to be Prolog programs, i.e., data and control structures are the same. The semantics of unification, however, are vastly different. We illustrate the language with a standard algorithm to compute the net present value of a loan, shown in Figure 2. This program is not related to the cooperative games explored in this paper, but exposes several important points about CLP(R).

Procedure `npv/6` has the following parameters: the `Start` and `End` periods of the loan, the `Principle`, the fixed loan `Rate`, `MR` (a fixed market rate), and `Value` (the net present value of the loan). The spawned equations form a recurrence. Each successive value is equal to the next value plus the discounted principle multiplied by the interest rate (`loan/5` clause 2). The final value (at the final period) also includes payback of the entire principle (`loan/5` clause 1). The final period can be fractional, requiring us to scale the loan and market rates by the remaining time.

Examples of queries to this program are instructive. The value of a three year $100 loan at 10% assuming a 5% market rate is:

```
?- npv( 1, 4, 100, 10, 5, V ).

V = -13.6162
```

Alternatively we can solve for loan rate:

```
?- npv( 1, 4, 100, R, 5, -14 ).

R = 10.14
```

```
npv( Start, End, _, _, _, 0 ) :- Start >= End.
npv( Start, End, Principle, Rate, MR, Value ) :-
   Start < End,
   Value = Principle - Payments
   loan( End-Start, Principle, Rate/100, MR/100, Payments ).

loan( Time, In, Rate, MR, Value ) :-
   Time > 0, Time <= 1,
   Out = In / ( 1 + MR * Time ),
   Value = Out * ( 1 + Rate * Time ).

loan( Time, In, Rate, MR, Value ) :-
   Time > 1,
   Out = In / ( 1 + MR ),
   Value = Next_Value + ( Out * Rate ),
   loan( Time-1, Out, Rate, MR, Next_Value ).
```

Figure 2: Loan Valuation in CLP(R)

However, we cannot solve for the market rate because the function is nonlinear in this variable. More strangely, we cannot solve for the time. For example, trying to solve for the ending period:

```
?- npv( 1, E, 100, 10, 5, -14 ).

E <= 2
1 < E
114 = _t13 * (0.1*E + 0.9)
100 = (0.05*E + 0.95) * _t13

*** (Maybe) Retry?
```

The "maybe" caveat in the result indicates that the non-linearity could not be removed and that the solution may be inconsistent.

4 Implementation

The previously described algorithm was implemented in 184 lines of CLP(R). In this section only the kernel of the implementation is described: the binary search for the tightening parameters.

Figure 3 shows the CLP(R) implementation of the search procedure. The main procedure, **search1/6** is passed the payoff data (**Id**), the tightest bounds known to keep the space solvable (**Prev**), the current increment to attempt on the bounds (**Delta**), the minimum granularity for termination (**Epsilon**), and the current tightening vector (**K0**). The final argument (**Final**) is the output tightening vector.

Clause (1) of **search1/6** terminates the search when the increment is smaller than the desired granularity. Clause (2) increments the current tightening vector and executes the new constraint set with **const/3**. If the space is still nonempty, the search recurses with half the increment. If the space is empty, the clause (3) is executed which retains the old tightening vector, but also halves the increment.

Procedure **update/3** updates the tightening vector. Any K_i is implemented by the data structure **f(Type,Value)** where **Type** is either **fix** (the parameter is fixed) or **float** (the parameter has not

```
search1( _, _, Delta, Epsilon, Final, Final ) :-
   Delta < Epsilon, !.

search1( Id, Prev, Delta, Epsilon, K0, Final ) :-
   Next = Prev + Delta,
   update( K0, Next, K1 ),
   const( Id, _, K1 ), !,
   search1( Id, Next, Delta/2, Epsilon, K1, Final ).

search1( Id, Prev, Delta, Epsilon, K, Final ) :-
   search1( Id, Prev, Delta/2, Epsilon, K, Final ).

update( [], _, [] ).
update( [ f(fix,K) | Ks ], New, [ f(fix,K) | Vs ] ) :-
   update( Ks, New, Vs ).
update( [ f(float,_) | Ks ], New, [ f(float,New) | Vs ] ) :-
   update( Ks, New, Vs ).
```

Figure 3: Binary Search for Tightening Parameters in CLP(R)

yet been fixed). Only nonfixed parameters are incremented.

All parameters are initialized as `float`, and if sufficiently iterated, all parameters eventually become `fix`. The test to convert a nonfixed to a fixed parameter is done in procedure `restrict/4` listed in the Appendix. Essentially, each nonfixed parameter is separately incremented by 2ϵ. For each such experiment where the space becomes empty, the parameter is fixed.

5 Fairness

All nonfixed parameters are increased in step during an iteration. Once the space becomes unsatisfiable, critical parameters are fixed, and another iteration is executed. However, the tightening algorithm is only as "fair" as the γ function which defines how the constraints are inflated. To further understand how fairness relates to γ, we first review the Shapley axioms [4].

Shapley stipulated that n-person coalition games must satisfy the following three criteria to be considered fair. He went on to prove that Shapley values are the only technique that can make this guarantee.

1. A *carrier* of a game must split the entire group payoff: none must go to *slackers*. For example, consider a 3-player game where $\{X\} = 0$, $\{Y\} = 5$, $\{Z\} = 5$, $\{X, Y\} = 5$, $\{X, Z\} = 5$, $\{Y, Z\} = 10$, and $\{X, Y, Z\} = 20$. Here X is the slacker and $\{Y, Z\}$ is the carrier. A fair solution will pay X nothing.

2. Fair solutions are invariant under relabeling the players. In other words, the search strategy or solution method does not depend on the order that players are considered.

3. Fair solutions are linearly composable. For example, consider two n-player games G_1 and G_2 with solutions S_1 and S_2. If we sum corresponding coalition payoffs in both games into G_3, then the fair solution to G_3 should be $S_1 + S_2$. The implications of this axiom are far reaching. It means that fair solutions are insensitive to *translation* or *scaling* of payoffs. For example given a game denominated in dollars or yen, the solution is identical. Less intuitively, if we

type	function	Shapley axioms
absolute	$\gamma(\text{RHS}, K) = \text{RHS} + f(K)$	2 & 3
relative	$\gamma(\text{RHS}, K) = \text{RHS}(1 + g(K))$	1 & 2
hybrid	$\gamma(\text{RHS}, K) = \text{RHS}(1 + g(K)) + f(K)$	2

Table 1: Families of Partially Fair γ Functions ($f(0) = g(0) = 0$)

offer an extra $100 to each player in a game, a fair solution gives each player his original cut (before the bonus) plus the $100. In other words, the bonus cannot affect a fair solution.

It is arguable if these axioms are necessary or sufficient to define fairness. Certainly there are alternatives to Shapley values that have been proposed, e.g., Raiffa's "offers that cannot readily be refused" method ([5] pg. 270). Raiffa claims:

> "It is not easy to suggest a compelling set of 'fairness principles' that deserve to be universally accepted as *the* arbitrated solution."

Without taking sides in this dispute, it is enlightening to evaluate the fairness of the tightening method with respect to these axioms. For the $\gamma(\text{RHS}, K)$ function to make sense, it must necessarily obey:

1. monotonicity in K to ensure that the bounds get tighter.
2. $\gamma(\text{RHS}, 0) = \text{RHS}$ to ensure consistency with the original problem.

In addition to these, J. Larson [3] conjectured that if γ also obeys the following two axioms in addition to those above, it will necessarily obey Shapley's axioms:

1. $\gamma(0, K) = 0$ to ensure that slackers don't get paid!
2. $\gamma(R + S, K) = \gamma(R, K) + \gamma(S, K)$.

We also hypothesize that in fact no γ can satisfy all four of the previous tightening axioms. However, we offer three families of γ functions that achieve partial success, summarized in Table 1. f and g are any monotonic functions in K such that $f(0) = g(0) = 0$.

Intuitively, absolute tightening shrinks the space without regard to its displacement from the origin. Thus slackers eventually get some payoff, making the family less fair. Relative tightening is less intuitive: it doesn't suffer from the fair carrier problem, but rather it is not linearly composable.

To visualize relative tightening, consider a rectangle in two-space defined by four inequalities such that each side is parallel to an axis, as shown in Figure 4. Each side of the rectangle will travel inwards towards the solution space, *in proportion to its distance from its parallel axis.* Thus the space will be shrinking as shown (the tightening vectors are not drawn to scale).

At some point during the binary search, two opposite sides of the rectangle will *cross*, making the space empty. The tightening parameters (corresponding to these two sides) allowing the smallest nonempty space will then be fixed. During the next iteration, the remaining two sides will continue to travel towards each other until they also cross. It is important to note that the final small square will *not* lie in the center of mass of the original rectangle. Since the sides moved in proportion to their distances from the axes, the final space will be biased towards the origin.

The relative tightening procedure strongly resembles Raiffa's "balanced increments" method ([5], pg. 243). Furthermore, the families of tightening functions can be interpreted in terms of economics.

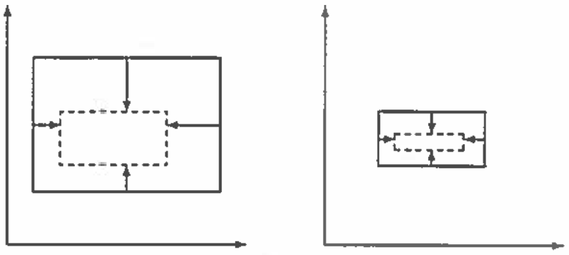

Figure 4: Illustrating Tightening Procedure

Consider the coalitions as processes that convert inputs into utility. Then $\overline{K}$ are additional inputs that we feed the system, and $\partial\gamma/\partial K$ is the rate as which inputs are converted into utility. Absolute tightening presupposes that $\partial\gamma/\partial K = \partial f(K)/\partial K$ so there is no "economy of scale" associated with the original value of a coalition. Relative tightening presupposes that $\partial\gamma/\partial K = RHS(\partial g(K)/\partial K)$. Thus utility production is amplified by the original coalition value.

The hybrid family loses both Shapley axioms 1 and 3, but offers the flexibility to perhaps moderate the tightening process through judicious selection of f and g. From an economic interpretation, coalition "production" is modeled as a combination of fixed and variable effects, as in any realistic process. Heuristic experimentation with the hybrid family is a topic of future research. In the remainder of the paper, we focus on absolute and relative tightening functions where $f(K) = g(K) = K$.

6 Example

Raiffa [5] presents a 3-player game called the Scandinavian Cement Problem. Consider players X, Y, and Z with the following coalition payoffs: $\{X\}$ gives 30, $\{Y\}$ gives 22, $\{Z\}$ gives 5, $\{X,Y\}$ gives 59, $\{X,Z\}$ gives 45, $\{Y,Z\}$ gives 39, and $\{X,Y,Z\}$ gives 77. Raiffa presents this as a linear programming problem in which he immediately eliminates one of the variables in order to plot the solution space in two dimensions. Here we plot the constraints in three dimensions because it better illustrates how the constraint tightening behaves.

The problem can be formulated in CLP(R) as follows:[2]

```
const( [ X, Y, Z ] ) :-
     X           >= 30,
     Y           >= 22,
     Z           >=  5,
     X + Y       >= 59,
     X + Z       >= 45,
     Y + Z       >= 39,
     X + Y + Z    = 77.
```

The success of `const/1` indicates that the solution space is nonempty. The procedure can be extended with tightening factors K_i (here we show relative tightening):

```
const( [ X,Y,Z ], [ K1,K2,K3,K4,K5,K6 ] ) :-
     X           >= 30*(1 + K1),
     Y           >= 22*(1 + K2),
     Z           >=  5*(1 + K3),
     X + Y       >= 59*(1 + K4),
```

[2]In reality, we do *not* generate such a procedure. Instead, there is a 3-player template defined: see the procedure `const/3` in the Appendix.

iteration	K1	K2	K3	K4	K5	K6
0	1000	1000	1000	1000	1000	1000
1	0.07689	0.07689	0.07689	0.07689	0.07689*	0.07689*
2	0.166655*	0.166655	0.166655	0.07689	0.07689*	0.07689*
3	0.166655*	0.297189*	0.297189	0.07689*	0.07689*	0.07689*
4	0.166655*	0.297189*	1.69241*	0.07689*	0.07689*	0.07689*

Table 2: Scandinavian Cement: Relative Tightening

```
X + Z      >= 45*(1 + K5),
Y + Z      >= 39*(1 + K6),
X + Y + Z  = 77.
```

In relative tightening, the factors are implemented as a percentage of the original lower bound. Thus for instance if `K1` = 0.2, the singleton X coalition is being "tightened" by 20% over its original value of 30 to 36. Table 2 shows the six parameters for successive iterations of the relative tightening algorithm. Initially parameters are set very high (1000%), but the binary search algorithm rapidly focuses the space in the first iteration to 7.6%. Parameters labeled "*" have been fixed by the search, i.e., no further tightening is possible for that constraint, given the minimal granularity ϵ. In this problem, the second iteration almost doubles parameters `K1`, `K2`, and `K3` to 17%. The third iteration doubles parameters `K2` and `K3` to 30%. Finally, the widest constraint, for coalition $\{Z\}$, is tightened down in the final iteration with `K3` = 169%.

Figure 5 (not drawn to scale) illustrates the 3-dimensional solution space of the problem. The $X + Y + Z = 77$ plane is shown, as are the projections of $X \geq 30$, $Y \geq 22$, and $Z \geq 5$. These projections on the plane define an asymmetric triangle from the larger equilateral triangle. The 2-player coalition constraints are not shown in this figure.

Figure 6 (not drawn to scale) illustrates the problem in a 2-space defined by the plane $X+Y+Z = 77$. The asymmetric triangle is intersected by the three inequality constraints representing the 2-player coalition payoffs. As the RHS bounds of 2-player coalitions are increased during tightening, these three constraints move towards the center of the solution space. As the RHS bounds of 1-player coalitions are increased during tightening, the three edges of the triangle move towards the center of the solution space.

7 Empirical Analysis

Table 3 shows eight benchmark games of 3 and 4 players used to evaluate the proposed allocation technique. The payoff data of these games is listed in the Appendix (see the `data/3` predicate). Note that the games do *not* all have the same total payoff. **cement** is the Scandinavian Cement problem. **tweak1** is a slight variant of **cement** used to illustrate some shortcomings of proposed algorithm. The next five games were randomly generated. The final game **tweak5** is a slight variant of **rand5**, again illustrating how our solutions diverge from Shapley values when the space is irregular. For each game, the Shapley values, relative and absolute tightened values are given. The mean squared errors (MSE) with respect to the Shapley values are given.

Obviously how accurately the tightening methods approximate Shapley values is highly variable. Neither relative nor absolute tightening appears to be superior. To illustrate the discrepancies with Shapley values, consider **tweak1** which takes the original Scandinavian Cement problem and modifies the payoff to coalition $\{X, Z\}$ from 45 to 53. The resulting relative MSE increases by a factor of 31 times. Notice that the effect of boosting the $\{X, Z\}$ payoff relative to all others is the same in

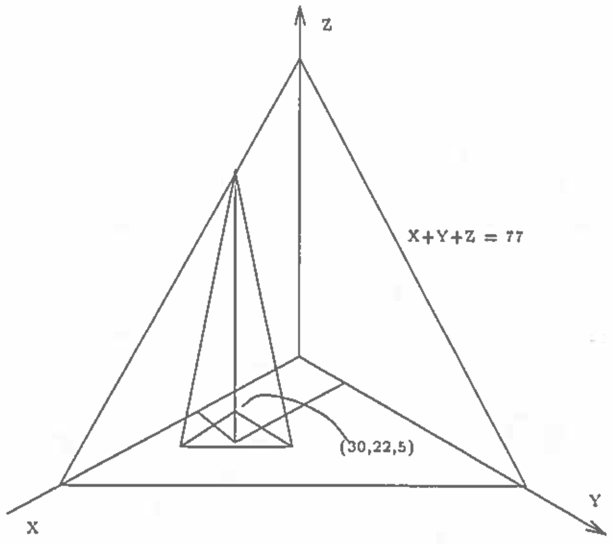

Figure 5: Scandinavian Solution Space: 3D View

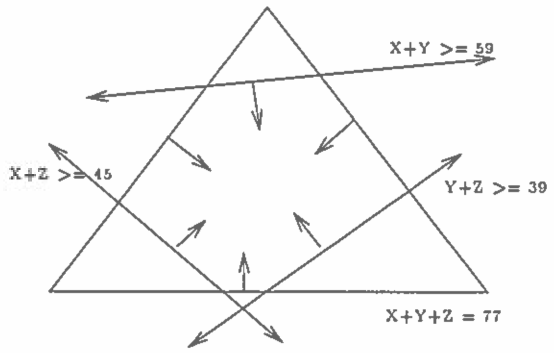

Figure 6: Shrinking the Scandinavian Solution Space

			Tightening Scheme	
game	players	Shapley Values	relative	absolute
cement		{35.5,28.5,13}	{35.0,28.5,13.5}	{34.4,28.3,14.3}
tweak1	3	{36.8,25.8,14.3}	{37.2,23.0,16.8}	{37,23,17}
rand1		{29.5,41,49.5}	{31.4,38.8,49.7}	{27.5,42,50.5}
rand2		{22.5,35,39.5}	{20.1,36.8,40.1}	{21.3,36.3,39.3}
rand3		{46.5,38.5,28.3,36.7}	{47.7,36.4,23.8,42.1}	{48,40,32,30}
rand4	4	{39.3,44.2,46.8,39.7}	{43.7,21.4,64.7,40.2}	{37.5,30.25,53.5,48.75}
rand5		{31.3,51.8,54.2,62.7}	{34.3,55.3,51.0,59.4}	{34,59,52,55}
tweak5		{37.2,46,48.3,68.5}	{49.4,35.7,28.1,86.8}	{60.5,38,31,70.5}
mean squared errors				
cement		0	0.155	1.06
tweak1	3	0	4.88	5.06
rand1		0	2.86	2.00
rand2		0	3.18	1.06
rand3		0	13.5	15.8
rand4	4	0	215	81.2
rand5		0	10.5	30.8
tweak5		0	249	228

Table 3: Benchmark Allocations: Shapley vs. Tightening

all techniques: Y's side payment decreases, subsidizing X and Z. The relative tightening scheme however more significantly penalizes Y (by 20%) than do the Shapley values, which penalizes Y by only 10%. Absolute tightening gets MSE = 5 for tweak1, and approximates Shapley slightly better than does the relative scheme. However, it also breaks down for tweak5.

8 Conclusions

This paper described a technique for determining a fair allocation for the classic n-player coalition problem. The solution exploits the ability to manipulate constraints at a meta-level in a constraint programming language such as CLP(R) used here. It has been long known how to model the coalition problem in terms of linear constraints. The only practical use of such modeling in the literature was to visualize 3-player games. In this paper, we propose a method wherein any number of (multi-dimensional) linear constraints are tightened by iterative scaling. The tightening shrinks the solution space in a "fair" manner based on a γ function. The final space is reduced arbitrarily small, from which any boundary point can be chosen as the answer.

We have demonstrated the technique for relative and absolute γ functions, and discussed the axiomatic construction of γ functions needed to obey Shapley-fairness. Our empirical results for simple γ functions show inconsistent approximations of Shapley values: some were close, some were not. Larson's Conjecture may point the way to more fair γ functions.

Interesting future research includes: 1) experimentation with the family of γ functions to better approach Shapley-fairness; 2) derive a formal relationship between our scheme are two types of Raiffa-fairness: "balanced increments" and "offers that cannot readily be refused" [5]; 3) determine how to model, in constraints, uncertain coalition payoffs and a related attribute: risk preferences among players, and 4) devise a dual method of *relaxing* an unsatisfiable set of constraints to permit a solution.

Acknowledgements

I am indebted to James Larson for explaining game theory to me and greatly contributing to the ideas presented in this paper. I thank Bob Clemen, Michael Maher, and Roland Yap for many key corrections and suggestions.

References

[1] N. C. Heintz, J. Jaffar, S. Michaylov, P. J. Stuckey, and R. H. C. Yap. The CLP(R) Programmer's Manual Version 1.2, September 1992.

[2] J. Jaffar, S. Michaylov, P. Stuckey, and R. H. C. Yap. The CLP(R) Language and System. *ACM Transactions on Programming Languages and Systems*, 14(3):339–395, 1992.

[3] J. Larson. personal communication, May 1995.

[4] G. Owen. *Game Theory.* Academic Press, Orlando, second edition, 1982.

[5] H. Raiffa. *The Art and Sciene of Negotiation.* Harvard University Press, Cambridge, 1982.

Appendix: Source Code of Allocation Algorithms

```
/*------------------------------------------------------------------------
Program:  N-Player Coalition Problem
Language: CLP(R)
Author:   E. Tick
Date:     May 7 1995

Notes:
1. Cannot solve contradictory data sets.
2. To use:
          ?- shapley( cement, Answer ).
             gives Answer to cement problem...

          ?- data( cement, _, Data ), tick( Data, 4, Answer ).
             gives Answer to cement problem using 4 tightening iterations...
------------------------------------------------------------------------*/
:- dynamic(found,1).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                            %
% find Shapley values...     %
%                            %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

shapley( Id, Answer ) :-
   alpha( Alpha ),
   data( Id, N, RawPayoffs ),
   convert( N, RawPayoffs, Alpha, Payoffs ),
   find_grand_coalition( Payoffs, Grand_Coalition ),
   findall( T, perm( Grand_Coalition, T ), Perms ),
   valuate( Perms, Payoffs, Values ),
   combine( Perms, Values, Answer ).

find_grand_coalition( Payoffs, Grand_Coalition ) :-
   rev( Payoffs, Rev ),
   Rev = [ f( Grand_Coalition, _ ) | _ ].

valuate( [], _, [] ).
valuate( [ Perm | Perms ], Payoffs, [ Value | Values ] ) :-
   value( Perm, Payoffs, Value ),
   valuate( Perms, Payoffs, Values ).

value( Perm, Payoffs, Value ) :-
   value( [], Perm, Payoffs, 0, Value ).

% compute marginal utility of adding each player to coalition...
value( _, [], _, _, [] ).
value( Coalition, [ Player | Players ], Payoffs, Current_Value,
       [ Delta | Deltas ] ) :-
   append( Coalition, [ Player ], New_Coalition ),
   member( New_Coalition, Payoffs, New_Value ),
   Delta = New_Value - Current_Value,
   value( New_Coalition, Players, Payoffs, New_Value, Deltas ).

% average marginal utility for each player to get Shapley value...
combine( Perms, Values, NewVector ) :-
```

```
   length( Perms, K ),
   combine1( Perms, Values, Vector ),
   divide_all( Vector, K, NewVector ).

divide_all( [], _, [] ).
divide_all( [ f( Player, Value ) | Vs ], K, [ New | NewVs ] ) :-
   Avg = Value / K,
   New = f( Player, Avg ),
   divide_all( Vs, K, NewVs ).

combine1( [ Perm | Perms ], [ Value | Values ], Vector ) :-
   initial( Perm, Value, List ),
   combine2( Perms, Values, List, Vector ).

combine2( [], [], List, List ).
combine2( [ Perm | Perms ], [ Value | Values ], List, Final ) :-
   add( Perm, Value, List, NewList ),
   combine2( Perms, Values, NewList, Final ).

add( [], [], List, List ).
add( [ Player | Players ], [ Value | Values ], List, NewList ) :-
   replace( f( Player, Value ), List, NextList ),
   add( Players, Values, NextList, NewList ).

replace( _, [], [] ) :- !.
replace( f(Key,Value), [ f(Key,OldValue) | Rest ], [ New | Rest ] ) :- !,
   NewValue = Value + OldValue,
   New = f( Key, NewValue ).
replace( NewItem, [ Item | Rest ], [ Item | News ] ) :-
   replace( NewItem, Rest, News ).

initial( [], [], [] ).
initial( [ Player | Players ], [ Value | Values ],
         [ f(Player,Value) | Rest ] ) :-
   initial( Players, Values, Rest ).

%----------------------------------------------------------------------

convert( N, Payoffs, Names, List ) :-
   grab( N, Names, Vars ),
   create_subsets( N, Vars, d( Subsets, [] ) ),
   shuffle( Payoffs, Subsets, List ).

shuffle( [], [], [] ).
shuffle( [ Payoff | Payoffs ], [ Subset | Subsets ], [ New | News ] ) :-
   New = f( Subset, Payoff ),
   shuffle( Payoffs, Subsets, News ).

% create a list of subsets of Vars...
create_subsets( N, Vars, Dlist ) :-
   K = pow( 2, N ),
   create_n_subset( 1, K, Vars, Dlist ).

create_n_subset( N, N, _, d( S0, S0 ) ).
create_n_subset( K, N, Vars, d( S0, S2 ) ) :- K < N,
   S0 = [ New | S1 ],
   make_subset( K, Vars, New ),
   create_n_subset( K+1, N, Vars, d( S1, S2 ) ).

make_subset( _, [], [] ).
make_subset( 0,  _, [] ).
make_subset( K, [ _ | Vars ], News ) :- K > 0,
   floor( K / 2, T ),
   T * 2 = K,
   make_subset( T, Vars, News ).

make_subset( K, [ Var | Vars ], [ Var | News ] ) :- K > 0,
   floor( K / 2, T ),
   not( T * 2 = K ),
   make_subset( T, Vars, News ).

grab( 0, Alpha, Alpha ).
grab( N, [ Alpha | Alphas ], [ Alpha | Rest ] ) :- N > 0,
   grab( N-1, Alphas, Rest ).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                            %
% find Tick values...        %
%                            %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% for 3-player games...
tick( RawPayoffs, Iterations, [ X, Y, Z ] ) :-
   init_parms( RawPayoffs, Payoffs ),
   go1( Iterations, RawPayoffs, 0.0001, 1000, Payoffs, Final ),
%  [A,B,C] and [X,Y,Z] must be explicit to make dump/3 work...
   const( RawPayoffs, [ A, B, C ], Final ),
   dump( [ A, B, C ], [ X, Y, Z ], Answer),
   choose( eval( Answer ), [] ).

% for 4-player games...
tick( RawPayoffs, Iterations, [ W, X, Y, Z ] ) :-
   init_parms( RawPayoffs, Payoffs ),
   go1( Iterations, RawPayoffs, 0.0001, 1000, Payoffs, Final ),
%  [A,B,C,D] and [W,X,Y,Z] must be explicit to make dump/3 work...
   const( RawPayoffs, [ A, B, C, D ], Final ),
   dump( [ A, B, C, D ], [ W, X, Y, Z ], Answer),
   choose( eval( Answer ), [] ).

init_parms( [ _ ], [] ) :- !.
init_parms( [ _ | Ls ], [ f( float, 1000 ) | Rest ] ) :-
   init_parms( Ls, Rest ).

% binary search for set of tightening parameters...
go1( 0, _, _, _, K, K ).
go1( N, Id, Epsilon, Delta, K0, K3 ) :- N > 0,
   search1( Id, 0, Delta, Epsilon, K0, K1 ),
   restrict( Id, K1, Epsilon, K2 ),
   go1( N-1, Id, Epsilon, Delta, K2, K3 ).

search1( _, _, Delta, Epsilon, Final, Final ) :-
   Delta < Epsilon, !.

search1( Id, Prev, Delta, Epsilon, K0, Final ) :-
   Next = Prev + Delta,
   update( K0, Next, K1 ),
   const( Id, _, K1 ), !,
   search1( Id, Next, Delta/2, Epsilon, K1, Final ).

search1( Id, Prev, Delta, Epsilon, K, Final ) :-
```

```
   search1( Id, Prev, Delta/2, Epsilon, K, Final ).

% replace all floating parameters by next choice,
% but keep all fixed parameters constant...
update( [], _, [] ).
update( [ f(fix,K) | Ks ], New, [ f(fix,K) | Vs ] ) :-
   update( Ks, New, Vs ).
update( [ f(float,_) | Ks ], New, [ f(float,New) | Vs ] ) :-
   update( Ks, New, Vs ).

% result of this procedure is to convert floating parameters
% into fixed parameters if they are critically causing the
% constraint set to fail on the next iterative deepening...

restrict( Id, In, Epsilon, Out ) :-
   restrict1( Id, In, [], Epsilon, Out ).

% return updated tightening vector...
restrict1( _, [], Out, _, Out ).

% keep fixed parameter constant...
restrict1( Id, [ In | Ins ], Partial, Epsilon, Out ) :-
   In = f( fix, _ ),!,
   append( Partial, [ In ], NewPartial ),
   restrict1( Id, Ins, NewPartial, Epsilon, Out ).

% test if floating parameter can be increased by a bit...
restrict1( Id, [ In | Ins ], Partial, Epsilon, Out ) :-
   In = f( float, Value ),
   NewValue = Value + 2*Epsilon,
   NewIn = f( float, NewValue ),
   append( Partial, [ NewIn | Ins ], Test ),
   const( Id, _, Test ), !,
   append( Partial, [ In ], NewPartial ),
   restrict1( Id, Ins, NewPartial, Epsilon, Out ).

% if floating parameter cannot be increased, fix it...
restrict1( Id, [ In | Ins ], Partial, Epsilon, Out ) :-
   In = f( float, Value ), !,
   NewIn = f( fix, Value ),
   append( Partial, [ NewIn ], NewPartial ),
   restrict1( Id, Ins, NewPartial, Epsilon, Out ).

% find an interior point in the solution space...
% iteratively replace <= constraints with =, if success,
% then include in final answer, otherwise back out...

choose( [], _ ).
choose( [ X <= Y | Constraints ], Partial ) :-
   append( Partial, [ X = Y | Constraints ], New ),
   map_call( New ), !,
   choose( Constraints, [ X = Y | Partial ] ).

choose( [ Constraint | Constraints ], Partial ) :-
   choose( Constraints, [ Constraint | Partial ] ).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                           %
% utility functions...      %
%                           %
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

alpha( [a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z] ).

var(   [_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_] ).

map_call( [] ).
map_call( [ Goal | Goals ] ) :-
   call( Goal ),
   map_call( Goals ).

append( [], X, X ).
append( [ X | Y ], Z, [ X | T ] ) :-
   append( Y, Z, T ).

rev( X, Y ) :-
   rev( X, [], Y ).

rev( [], X, X ).
rev( [ A | X ], Y, Z ) :-
   rev( X, [ A | Y ], Z ).

member( _, [], [] ).
member( Key1, [ f( Key2, Value ) | _ ], Value ) :-
   perm( Key1, Key2 ), !.
member( Key, [ _ | Rest ], Value ) :-
   member( Key, Rest, Value ).

perm( [], [] ).
perm( X, [ U | V ] ) :-
   del( U, X, Z ),
   perm( Z, V ).

del( X, [ X | Y ], Y ).
del( X, [ Y | Z ], [ Y | W ] ) :-
   del( X, Z, W ).

qsort( In, Out ) :-
   qsort( In, Out, [] ).

qsort( [], Ans, Ans ).
qsort( [ X | R ], Y, T ) :-
   partition( R, X, S, L ),
   qsort( S, Y, [ X | Y1 ] ),
   qsort( L, Y1, T ).

partition( [], _, [], [] ).
partition( [ X | Xs ], A, S, [ X | L1 ] ) :- A < X, !,
   partition( Xs, A, S, L1 ).
partition( [ X | Xs ], A, [ X | S1 ], L ) :- A >= X, !,
   partition( Xs, A, S1, L ).

length( List, Length ) :-
   length( List, 0, Length ).

length( [], K, K ).
length( [ _ | Ls ], K, Length ) :-
   length( Ls, K+1, Length ).

% from Clocksin and Mellish, 1981...
```

```
findall( X, G, _ ) :-
   asserta( found( mark ) ),
   call( G ),
   asserta( found( X ) ),
   fail.
findall( _, _, L ) :-
   collect_found( [], M ), !,
   L = M.

collect_found( S, L ) :-
   getnext( X ), !,
   collect_found( [ X | S ], L ).
collect_found( L, L ).

getnext( X ) :-
   retract( found( X ) ), !,
   not( X == mark ).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                           %
% data sets                 %
%                           %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% payoff list must be (2^K)-1 elements
% in length, where K is the # players...

%     id    players    payoffs
data( cement, 3, [30,22,59,5,45,39,77] ).
data( tweak1, 3, [30,22,59,5,53,39,77] ).
data( raiffa, 3, [0,0,118,0,84,50,121] ).    % should fail...
data( rand1,  3, [12,33,65,46,69,71,120] ).
data( rand2,  3, [18,33,39,36,45,55,97] ).
data( rand3,  4, [30,22,59,8,45,39,77,12,55,45,90,44,81,73,150]).
data( rand4,  4, [6,8,40,8,63,47,99,10,64,47,75,80,55,84,170] ).
data( rand5,  4, [8,33,17,26,51,45,43,29, 56,49,61,53,56,99,200] ).
data( tweak5, 4, [8,33,17,26,51,45,43,29,126,49,61,53,56,99,200] ).

% (absolute templates not shown: similar to those below)
% general relative template for 3-player coalition game...
const( [ A, B, C, D, E, F, G], [ X, Y, Z ],
    [ f(_,K1), f(_,K2), f(_,K3), f(_,K4), f(_,K5), f(_,K6) ] ) :-
    X          >= A*(1 + K1),
    Y          >= B*(1 + K2),
    Z          >= D*(1 + K3),
    X + Y      >= C*(1 + K4),
    X + Z      >= E*(1 + K5),
    Y + Z      >= F*(1 + K6),
    X + Y + Z  = G.

% general relative template for 4-player coalition game...
const( [ A, B, C, D, E, F, G, H, I, J, K, L, M, N, O ],
    [ W, X, Y, Z ],
    [ f(_,K1), f(_,K2), f(_,K3), f(_,K4), f(_,K5),
      f(_,K6), f(_,K7), f(_,K8), f(_,K9), f(_,K10),
      f(_,K11), f(_,K12), f(_,K13), f(_,K14) ] ) :-

    W          >= A*(1 + K1),
    X          >= B*(1 + K2),
    Y          >= D*(1 + K3),
Z         >= H*(1 + K4),
W + X     >= C*(1 + K5),
W + Y     >= E*(1 + K6),
W + Z     >= I*(1 + K7),
X + Y     >= F*(1 + K8),
X + Z     >= J*(1 + K9),
Y + Z     >= L*(1 + K10),
W + X + Y >= G*(1 + K11),
W + X + Z >= K*(1 + K12),
W + Y + Z >= M*(1 + K13),
X + Y + Z >= N*(1 + K14),
W + X + Y + Z = 0.
```