Evaluation and Improvement of Laruelle-Widgrén Inverse Banzhaf Approximation

Frits de Nijs Daan Wilmer Tomas Klos

Algorithmics Group, Delft University of Technology
Delft, The Netherlands

Abstract

Voting is a popular way of reaching decisions in multi-agent systems. Weighted voting in particular allows different agents to have varying levels of influence on the decision taken: each agent’s vote carries a weight, and a proposal is accepted if the sum of the weights of the agents in favor of the proposal is at least equal to a given quota. Unfortunately, there is no clear and unambiguous relation between a player’s weight and the extent of her influence on the outcome of the decision making process. Different measures of ‘power’ have been proposed, such as the Banzhaf and the Shapley-Shubik indices.

Here we consider the ‘inverse’ problem: given a vector of desired power indices for the players, how should we set their weights and the quota such that the players’ power in the resulting game comes as close as possible to the target vector? There has been some work on this problem, both heuristic and exact, but little is known about the approximation quality of the heuristics for this problem. The goal of this paper is to empirically evaluate the heuristic algorithm for the Inverse Banzhaf Index problem by Laruelle and Widgrén. We analyze and evaluate the intuition behind this algorithm. We found that the algorithm can not handle general inputs well, and often fails to improve inputs. It is also shown to diverge after only tens of iterations. Based on our analysis, we present three alternative extensions of the algorithm that do not alter the complexity but can result in up to a factor 6.5 improvement in solution quality.

1 Introduction

In systems composed of multiple agents, voting is a popular means of aggregating the preferences of these agents to come to a joint decision. A good example of voting concerns the presidential elections of the United States of America. This follows a two-step process, which nicely illustrates two types of voting. In the first step the citizens vote in each state. Every voter has the same weight and the candidate with the most votes in a state wins that state. The second step illustrates another type of voting: weighted voting. In this step every state votes for the candidate that won in that state. However, it would not be fair if each state has the same vote: the state of California represents over 37 million citizens while little over 560,000 live in Wyoming. Therefore each state has a certain weight, represented by a number of electors. The new president is then chosen by the majority of electors.

With such weighted voting situations, and especially when they are used to elect one of the most powerful men on earth, it is the question how fair the voting is. We can measure this, for example, by using the Banzhaf power index [2] and comparing that to a fair power distribution. Instead of trying to create a fair index—which is much more a philosophical and political question, rather than an algorithmic problem—we try to find a distribution of weights of which the power index matches a target power index.

In Section 2 we give some preliminaries and a more exact definition of the problem we address, along with related work. Section 3 gives a detailed description of the algorithm of Laruelle and Widgrén, including a discussion of some of its weak points. We propose and empirically evaluate alternative methods of initializing the algorithm in Section 4. Then, we propose and evaluate some adaptations to the algorithm itself in Section 5. Section 6 concludes and gives directions for further study.

1 We thank the reviewers for their helpful comments.
2 Problem statement

A weighted voting game (WVG) consists of a set $\mathcal{N}$ of $n$ players $1, 2, \ldots, n$, each with a voting weight $w_1, w_2, \ldots, w_n$, along with a quota $q$. We write a WVG as $[q; w_1, \ldots, w_n]$. A coalition $C$ is a subset of players, and every coalition has a value $v(C) \in \{0, 1\}$ as follows: $v(C) = 1 \leftrightarrow \sum_{i \in C} w_i \geq q$. A coalition with value 1 is called winning, and a coalition with value 0 is called losing.

We are interested in the influence of players on the outcome of decisions in WVGs, i.e., in their so-called ‘voting power.’ To see that a player’s weight is not a good measure of influence, consider the WVG $[50; 49, 49, 2]$. Here, at least two players are necessary to form a winning coalition, so that in this sense, the third player can be considered to have the same amount of influence as the other two, even though her weight is much lower. A popular—though not the only—method to measure a priori power is the Banzhaf power index [2]. It measures the power of a player $i$ by dividing the number of coalitions of other players for which player $i$ is critical (meaning that the coalition is losing, and that player $i$ can make it winning by joining it), by the total number of coalitions of other players. More formally,

$$\tilde{\beta}_i = \frac{1}{2^{n-1}} \sum_{C \subseteq \mathcal{N} \setminus \{i\}} (v(C \cup \{i\}) - v(C)).$$

Often not the regular Banzhaf index is used, but the normalized version [9]. This abstraction is made when it is not interesting in how many cases players can actually exert power, but only how the power is distributed among players. Because that is what we need, we use the normalized Banzhaf index as well:

$$\beta_i = \frac{\tilde{\beta}_i}{\sum_{j=1}^{n} \tilde{\beta}_j}.$$

Just computing the power of a weighted voting game is an NP-hard problem [10]. (See [3] for a thorough survey of problems related to power indices and algorithms for solving them.) However, our goal is not to compute power indices, but to solve the ‘inverse’ problem: When we are given a desired distribution of power, we need to find a quota and a set of weights, such that the power in the resulting WVG is distributed as ‘closely’ as possible to the desired distribution (according to some distance measure).

There has been some recent work on the inverse power index problem. De Keijzer, Klos and Zhang [4] propose a method to enumerate all weighted voting games for a given number of agents, that makes use of a partial order that they prove exists on the set of WVGs. By calculating the power distribution for each enumerated game, a game with a power distribution closest to the target will certainly be found. Due to its exponential runtime of $O(2^{n^2+2n})$, however, this algorithm is not practical for larger instances—for example for computing the weights for the 27 members of the Council of the EU.

Several other heuristics than the one by Laruelle and Widgrén examined here, have also been proposed. Fatima, Wooldridge and Jennings [7] designed an $O(n^6)$ approximation algorithm, that traverses the space of WVGs by iteratively shifting parts of the weight from players that have too much power, to players that have too little power, according to a comparison of the game’s power index with the target. Their update rules have a property that makes the algorithm anytime: it can be stopped at any iteration and every iteration gives a better or equal result. Unfortunately, it is focused on the Shapley-Shubik power index, which is similar, but not equal to the Banzhaf power index.

Aziz, Paterson and Leech [1] also designed an iterative approximation algorithm. Their algorithm is used to design games that approximate some target Banzhaf index. They use generating functions to calculate the Banzhaf index, which is efficient, but only if the weights are integer. They use interpolation of the current voting power and the desired voting power to determine the next set of weights, multiply them with a certain factor and then round them to integers. The authors don’t analyze the approximation quality of their algorithm.

3 Laruelle-Widgrén

In our paper we focus on the iterative algorithm by Laruelle and Widgrén [9]. Starting from a set of initial weights (for which they use the target power index itself), the algorithm iteratively updates the weights of the players by first calculating the Banzhaf index for the given weights, then calculating, for each player,
the ratio of her Banzhaf index and her target power index, and finally dividing each player’s weight by its corresponding ratio. That way the weights are adjusted according to the error in the Banzhaf index. The algorithm stops when it is close enough to a given distance threshold, or when a maximum number of iterations has been reached.

A pseudocode version of this algorithm is given in Algorithm 1. It takes as input the valuation function \( v \), the vector ‘target’ (also called \( t \) below), the vector \( \omega_0 \) containing the initial weights, and the numbers ‘\( \text{maxDistance} \)’ (the distance threshold) and ‘\( \text{maxIterations} \)’ (the maximum number of iterations). The vectors \( \omega_0, \omega, \text{banzhafIndex}, \text{target} \) and \( \text{ratio} \) are all vectors of length equal to the number of players.

**Algorithm 1 Laruelle-Widgrén**

**Require:** The vector ‘target’ is a normalized vector of size \( n \), with \( n > 0 \).
1. For each player \( i \): weight\((i) \leftarrow \omega_0(i) \)
2. iterations \( \leftarrow 0 \)
3. repeat
4. banzhafIndex \( \leftarrow \text{calculateBanzhaf}(v, \text{weight}) \)
5. For each player \( i \): ratio\((i) \leftarrow \frac{\text{banzhafIndex}(i)}{\text{target}(i)} \)
6. For each player \( i \): weight\((i) \leftarrow \frac{\text{weight}(i)}{\text{ratio}(i)} \)
7. distance \( \leftarrow \text{distance} (\text{banzhafIndex}, \text{target}) \)
8. iterations \( \leftarrow \) iterations \( + 1 \)
9. until (distance \( < \) maxDistance) \( \lor \) (iterations \( > \) maxIterations)

**Ensure:** weight is a vector of size \( n \)

The authors do not provide any guarantees for the algorithm. In their paper it is shown to give a good approximation for some cases, but the general case is not analyzed. Here, we do focus on the general case, which is why we refrain from giving the details of the specific WVGs analyzed by Laruelle and Widgrén—those being the WVGs governing different types of decision making in the Council of the EU.

As mentioned above, the original algorithm sets the starting weights \( \omega_0 \) (the initial weight vector) to the target vector. Intuitively, this makes sense because the power and the weight distribution are related to some extent. As it turns out, however, this choice can easily lead to selecting starting weights in which at least one of the players has a Banzhaf index of zero. For example, if the highest weight is larger than the quota, then all the players with lower weights have zero power according to the Banzhaf index. Then the ratio for that player will also be zero, resulting in a divide-by-zero error in the first iteration (in line 6, also see [5]). In the next section we present two alternative methods for setting \( \omega_0 \) to alleviate this issue, and we evaluate them empirically in section 4.2.

## 4 Alternative \( \omega_0 \)

We consider three methods for setting \( \omega_0 \) in our empirical evaluation.

**Target** The initial weight distribution used by Laruelle and Widgrén.

**Centroid** A centroid is a type of center, intuitively defined as the average of all vectors in the body, or alternatively as the center of its mass. As in [4], and without loss of generality, we look only at canonical WVGs, in which the vector of weights is ordered in non-increasing order from player 1 to \( n \). Thus, all allowed weight vectors appear in what is called the ordered simplex. The centroid of a Simplex is computed as the normalized sum of its \( n \) vertices \((v_0, \ldots, v_{n-1})\). Using this method, the initial weight vector is \( \omega_0 = \frac{1}{n} \sum_{i=0}^{n-1} v_i \), where the \( v_i \) are the vertices of the (ordered) simplex. The rationale is that this vector is in a region of the Simplex where, according to Kurz [8], many Banzhaf vectors exist. Starting close to many Banzhaf vectors is desirable, because then each iteration is likely to jump to a new Banzhaf vector which in turn results in a slightly different ratio. Changing the ratio often, introduces variance in the direction of update which intuitively leads to better convergence.

**Offset Target** As the target vector \( t \) may lead to a powerless player and the associated problems, we can smooth the initial weight distribution by averaging \( t \) with the uniform weight distribution. In this setting, player \( i \)'s initial weight vector \( \omega_0(i) \) is computed as: \( \omega_0(i) = \frac{t(i)+1/n}{2} \).
4.1 Evaluation Metrics

For our evaluation we use the concept of Manhattan or Taxicab distance $d_1$, or $||.||_1$. We chose this distance norm because it relates to the results in [8] on the lower bound on the distance between $t$ and $\beta_{opt}$, and because it is cheap to compute.

In order to illustrate our evaluation metrics, Figure 1 presents an example of the distance between $t$ and a number of related Banzhaf vectors in two dimensions. In the figure, we consider the following Banzhaf vectors:

- $\beta_{opt}$, an unknown optimal answer. In this figure, it is not equal to $t$, which in general it may not be.
- $\beta_{best}$, a best known (closest) algorithm output found in a database of previously returned Banzhaf vectors.
- $\beta_{alg i}$, the algorithm output for certain parameter settings $i$.

![Figure 1: Example of $d_1$ distance between target vector $t$ and related Banzhaf vectors in 2D. All points on a diamond are equidistant to $t$, according to the $d_1$ distance metric.](image)

We thus propose two metrics of interest:

1. The relative improvement obtained by using a parameter setting $2$ compared to $1$, $\frac{d_1(t, \beta_{alg 1}) - d_1(t, \beta_{alg 2})}{d_1(t, \beta_{opt})}$.

2. The error in the output produced by a run of the algorithm, defined as $d_1(t, \beta_{alg i}) - d_1(t, \beta_{opt})$.

The relative improvement tells us something about the usefulness of a specific parameter setting. In other words, it tells us how the algorithm should be used to get the best possible results. In figure 1 we can see that the point for $\beta_{alg 2}$ lies inside the black diamond indicating $d_1(t, \beta_{alg 1})$, so it is an improvement over $\beta_{alg 1}$. In the computation for improvement we do not compare the points directly, but rather the distance to $t$, or the minimum Manhattan distance between $\beta_{alg 2}$ and a point on the diamond for $\beta_{alg 1}$, shown as ‘Improve’ in figure 1. We consider an improvement to be significant if it exceeds 0.05.

The error tells us something about the general usefulness of the algorithm. The error is the minimum Manhattan distance between two points on the green and black diamonds, indicated by the line ‘Error’ in figure 1. Since we do not know $\beta_{opt}$, the error must be estimated. An upper bound on the error is $d_1(t, \beta_{alg i})$. This corresponds to the distance between a point on the black and $t$, line ‘Upper’ in figure 1. This upper bound is tight, since it could be that $t = \beta_{opt}$, in which case $d_1(t, \beta_{opt}) = 0$. A lower bound on the error is $d_1(t, \beta_{alg i}) - d_1(t, \beta_{best})$. In figure 1 it is the line ‘Lower,’ the minimum Manhattan distance between any two points on the black and blue diamonds. This lower bound is also tight since we may have stored the optimal answer, in which case $\beta_{best} = \beta_{opt}$.

The upper bound $d_1(t, \beta_{alg i})$ is a biased estimate of the actual error, since it is in general not the case that $t = \beta_{opt}$. This is a consequence of the fact that Banzhaf vectors are composed of rational numbers. Kurz proves in [8] that there exists a lower bound on the largest $d_1(t, \beta_{opt})$ of $\frac{1}{3}$, and conjectures that this bound is actually $\frac{14}{31}$.
obtaining Banzhaf vectors. The quality of the lower bound further depends on the percentage of all Banzhaf vectors we have in our database. We generated Banzhaf vectors by running the algorithm on random samples and storing the vector computed in each iteration of the algorithm (step four) until the database remained constant for 250 consecutive samples. For 8 players this resulted in 1,094,138 Banzhaf vectors, which compared to the 2,730,164 weighted voting games that exists for 8 players [8] means that there exists at least 1 vector for every 2.5 games at this size.

The conjectured lower bound of \( \frac{14}{37} \) on the largest distance between \( t \) and \( \beta_{\text{opt}} \) can be used to define the size of a significant error for the upper bound estimate. We say that the upper bound error of an algorithm (parameter setting) is significant if the average value of the error exceeds 10\% of \( \frac{14}{37} \), or \( \frac{7}{185} \approx 0.0378 \). Further, we say a change in the error is significant if the difference exceeds 1\% of the maximum value of \( d_1 \). Then a significant error for the lower bound estimate is at least 1\% of the maximum value of \( d_1 \).

### 4.2 Experiments with methods for setting \( \omega_0 \)

For our experiments we need a number of target vectors \( t \) to apply the algorithm on. Since a target vector is a vector in the simplex, we produce sample target vectors by drawing a vector uniformly at random from the \( n \)-dimensional ordered Simplex. This can be done by drawing \( n \) samples \( w_i \) from \( U[0, 1] \), setting \( w_i \leftarrow -\ln w_i \), renormalizing, and sorting in non-increasing order [6].

In order to evaluate empirically what choice of \( \omega_0 \) produces the best results, we performed a number of experiments. For our experiments we drew 10,000 samples from the ordered \( n = 8 \)-dimensional Simplex. On each sample we applied the algorithm with all methods for setting the initial weight vector discussed in Section 4, and with \( q = 0.6 \) which resulted in the best performance in our initial experiments [5]. Each parameter combination was run for 50 iterations.

Figure 2 presents the results. The x-axis presents the initial distance \( d_1(t, \beta(\omega_0)) \) between the target and the power distribution resulting from the initial weight vector \( \omega_0 \). The y-axis shows the lower bound on the error obtained after 50 iterations, relative to the initial distance \( d_1(t, \beta(\omega_0)) \). A lower value means that the algorithm improves more, relative to the initial distance: A value of 1 on the y-axis means that the eventual error found is equal to the initial error. Each datapoint is one of the 10,000 samples.

Table 1 presents the key features (mostly averages) of the data in figure 2. The first column shows the method for selecting the initial weight vector for the algorithm. The second column shows the initial distance

<table>
<thead>
<tr>
<th>( \omega_0 ) method</th>
<th>Initial</th>
<th>Worst Error</th>
<th>Error Upper</th>
<th>Error Lower</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target</td>
<td>0.1603</td>
<td>0.7988</td>
<td>0.1179</td>
<td>0.0899</td>
</tr>
<tr>
<td>Centroid</td>
<td>0.2535</td>
<td>0.6047</td>
<td>0.0803</td>
<td>0.0522</td>
</tr>
<tr>
<td>Offset</td>
<td>0.3146</td>
<td>0.4228</td>
<td>0.0833</td>
<td>0.0552</td>
</tr>
</tbody>
</table>

Table 1: Effect of the different choices for \( \omega_0 \) on the error.
to \(t\) (averaged over the 10,000 samples), and therefore the average upper bound on the error before running the algorithm. The third column presents the worst case distance to \(t\) (across all samples) after running the algorithm. Columns four and five present the averages of the upper and lower bounds on the error, where we note the average upper bound on the error is the average distance to \(t\) after applying the algorithm.

These results show how the initial distance is affected by the choice of \(\omega_0\), with \(\omega_0\) set to the target (Figure 2(a)) having the smallest initial distance and offset (Figure 2(c)) having the largest (also see the second column in Table 1). In this sense, then, using the target as the initial vector was indeed a smart choice. However, when \(\omega_0\) is set to the target, relatively many samples are hard to improve—those with high relative errors. We expect that this is caused by samples starting in a segment with at least one very low target power index, leading it to quickly obtain 0 power, which therefore cannot be improved beyond this initial target, as we explained in section 3. Our intuitions are confirmed by the fact that we don’t find this effect when \(\omega_0\) is set by both alternative methods, which were explicitly designed to overcome this. For \(\omega_0\) set to either target or offset a higher initial distance generally results in a higher relative error, while for \(\omega_0\) set to centroid the opposite appears to happen where the higher initial distances are improved more.

The consequence of these hard-to-improve samples is that \(\omega_0\) set as offset or centroid gives a significant improvement in both the upper and lower bound error compared to \(\omega_0\) set to the target. However, the total magnitude of the error is still significant in all cases. The difference between offset and centroid is not significant, but offset does produce the lowest worst case distance to \(t\), and it can thus be seen as the most robust. Overall we can conclude that starting close to the target is less important than starting in a position where the algorithm can improve the result.

5 Algorithmic improvements

In order to factor out the premature stop due to zeros in the weight vector we now propose and evaluate three possible changes to the algorithm itself (rather than to the initialization, as in the previous section).

**Restarts** When a zero is encountered, restart on a new set of weights. This new set cannot be a set that was already encountered, because otherwise the algorithm will not find any new weights and it will stop at the same point, due to the zeros. The new set of weights that we propose is a transformation, in the form of the Offset Target procedure described above, of the best found set of weights to get weights between that point and the centroid of the simplex. Because it is essentially the same as the original algorithm until it stops on a zero, we expect this algorithm to return results that are at least as good as the original, at least beyond the standard number of iterations of the algorithm.

**Coalition** Avoid zeros in the power index by modifying the valuation function such that players without sufficient voting weight can have power. We do this by imposing a minimum coalition size—a feature also of WVGs governing some decision making in the Council of the EU. If there are coalitions with enough voting weight but not enough members, players that have zero voting weight can make it winning by joining it, and therefore have power after all.

**Scaling** Avoid zeros in the power vector by changing the calculation of the new set of weights. Or rather: change the way the ratio is calculated (see Algorithm, 1 line 5). Adding a value, which we call the scaling factor, to both the dividend and the divisor: \(\text{ratio}(i) = \frac{\text{banzhafIndex}(i) + s}{\text{targetIndex}(i) + s}\). As long as \(s > 0\) this will never be reduced to zero, so the weights for a player will always be strictly positive unless the initial weight for that player is zero.

An added effect of this scaling factor is that the steps of the iterations will be smaller, which leads to slower convergence to the target but may also limit overshooting the target.

5.1 Evaluation

Since each suggested improvement has its own parameters, we first look at the effect of these parameters in isolation. To examine what relative improvement can be obtained, we drew 10,000 samples from the ordered \(n = 8\)-dimensional simplex and compared the result with that of the standard algorithm run for 50 iterations, with parameters \(\omega_0\) set to the target, and \(q = 0.5\) since we expect this to be the best value in general [5]. For the restart improvement we increased the number of iterations performed from 10 through 80 (original is 50). For the minimum coalition improvement we varied the minimum coalition size from 1 through 8 (original is 1). And finally, for the scaling improvement we varied the scaling factor from 0
through 7 (original is 0). Other parameters are set equal to the original version. The results can be seen in figure 3, which shows relative improvement on the \( y \)-axis (see the definition of this measure in Section 4).

![Figure 3: Relative improvement for different parameter settings on the three improvements (\( q = 0.5 \) and \( \omega_0 \) is set to the target). The figure shows box-plots summarizing 10,000 samples at each parameter setting.](image)

From the figure we can see that restart should be run with the same number of iterations as the base algorithm to obtain universal improvement: If less than 50 iterations are used for the restart algorithm, it may not yet have attained the distance reached by the original algorithm in 50 iterations. For minimum coalition the ideal size appears to be 3, since here the majority of samples is improved. (Note that this is likely related to the dimension in these experiments. For higher numbers of players, we expect the optimum minimum coalition size to increase.) The scaling factor should be positive, but kept small.

To investigate the improvement of the proposed additions in terms of the error, we performed an experiment with tuned parameters for each. Every algorithm was run on 5,000 samples of \( n = 8 \)-dimensional Simplex, with parameters set to produce the best results: 50 iterations, \( q = 0.6 \) (except when using scaling which performed better with \( q = 0.5 \)), and \( \omega_0 \) set to the centroid. Table 2 shows the results. The first column lists the algorithm under consideration. The second and third columns list the percentage of samples that were improved and made worse, respectively, compared to the base. Column four shows how often an algorithm produced strictly the best result compared to all the others. The fifth and sixth columns present the average distance to the target vector and the best known power vector, respectively. The last row of these columns shows what the results would be like if we could always pick the best algorithm for a sample.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>% Improved</th>
<th>% Worse</th>
<th>% Best</th>
<th>Error upper</th>
<th>Error lower</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0.0801</td>
<td>0.0520</td>
</tr>
<tr>
<td>Restarts</td>
<td>32.9%</td>
<td>0.0%</td>
<td>3.0%</td>
<td>0.0643</td>
<td>0.0363</td>
</tr>
<tr>
<td>Coalition (min. 3)</td>
<td>23.6%</td>
<td>18.6%</td>
<td>3.6%</td>
<td>0.0761</td>
<td>0.0481</td>
</tr>
<tr>
<td>Scaling (( s = 0.4 ))</td>
<td>86.2%</td>
<td>12.0%</td>
<td>80.0%</td>
<td>0.0357</td>
<td>0.0078</td>
</tr>
<tr>
<td>Sum/Best</td>
<td>-</td>
<td>-</td>
<td>86.5%</td>
<td>0.0344</td>
<td>0.0065</td>
</tr>
</tbody>
</table>

Table 2: Effect of the improvements for ideal parameter settings.

The table shows that all additions reduce the error on average (in columns five and six, the errors for the three improvements are lower than in the first row of the table), however the magnitude of improvement can only be considered significant for scaling. (Again, significance is established when the error is more than
1% of the maximum $d_1$ in the simplex (which is 1.75 for $n = 8$) smaller than the error of the original.) Additionally, for scaling, the upper bound on the error is below 10% of the largest possible error, i.e. below $\frac{7}{100}$ (see section 4.1), and the lower bound is even below 0.5% of the maximum $d_1$ in the 8-dimensional simplex. Therefore we can say that, by our definition of significance, the error made by scaling is not significantly larger than zero. Compared to the original algorithm with ideal parameters, introducing a scaling factor improves the lower bound error by a factor of more than 6.5.

The other approaches have their own strengths, as restarting does not make the result worse, and minimum coalition actually returns the best result more often than restarting does. In a sense the approaches can be seen to complement each other. If we always take the best result, the improvement compared with scaling is still another 20% in the lower bound which indicates it could be worthwhile to find a new version of the algorithm that combines the effect of the three approaches in some way.

6 Conclusion and Future Work

The algorithm by Laruelle and Widgrén works quite well in most cases, but it has some major shortcomings. Our proposals remove the possibility of the algorithm getting stuck in a case where one or more zeroes are in the weight vector, and experiments show that our scaling factor algorithm also improves the average approximation performance. However, in some cases it performs worse than the original algorithm. Our multiple start proposal performs at least as well as the original algorithm, but improves the solution not nearly as much as the scaling factor algorithm. Further work could be done to find an algorithm that improves on these proposals: either by giving a better worst-case performance or by improving the average case approximation, or both.

Our experiments also show that the algorithm is not anytime: an iteration often improves the solution, but it could also deteriorate. Our improvements do not counter that, other than storing the best found solution. This is also something that could be researched in the future. It is also interesting to know the dependence of the improvements of the various proposed modifications on the number of players. So far, we have only evaluated on $n = 8$ players, but if we want to design voting games for larger numbers of players, we need to know whether they pose additional challenges, and whether we need different designs of our algorithm.

References


