Efficient Algorithm for Designing Weighted Voting Games

Haris Aziz
Computer Science Department, University of Warwick
Coventry, UK, CV4 7AL
haris.aziz@wawick.ac.uk

Mike Paterson
Computer Science Department, University of Warwick
Coventry, UK, CV4 7AL
msp@dcs.wawick.ac.uk

Dennis Leech
Economics Department,University of Warwick
Coventry UK, CV4 7AL
d.leech@warwick.ac.uk

20 May, 2007

Abstract

Weighted voting games are mathematical models, used to analyse situations where voters with variable voting weight vote in favour of or against a decision. They have been applied in various political and economic organizations. Similar combinatorial models are also encountered in neuro-science and distributed systems.

The calculation of voting powers of players in a weighted voting game has been extensively researched in the last few years. However, the inverse problem of designing a weighted voting game with a desirable distribution of power has received less attention. We present an elegant algorithm which uses generating functions and interpolation to compute an integer weight vector for target *Banzhaf power indices*. This algorithm has better performance than any other known to us. It can also be used to design egalitarian two-tier weighted voting games and the representative weighted voting game for a multiple weighted voting game.

MSC2000 Classification: 05A05, 68Q25, 91B12.

Keywords: generating functions, weighted voting games, Banzhaf indices, algorithms and complexity.

1 Introduction

1.1 Motivation

Weighted voting games are mathematical models which are used to analyze voting bodies in which the voters have different number of votes. In weighted voting games, each voter is assigned a non-negative weight and makes a vote in favour of or against a bill. The bill is passed if and only if the total weight of those voting in favour of the bill is equal to or greater than the quota. Weighted voting games have been applied in various political and economic organizations for structural or constitutional purposes. Prominent applications include the United Nations Security Council, the Electoral College of the United States and the International Monetary Fund ([18], [3]). The distribution of voting power in the European Union Council of Ministers has received special attention in [1], [5], [17], [15] and [10]. Voting power is also used in joint stock companies where each shareholder gets votes in proportion to the ownership of a stock ([4], [11]). Weighted voting games are also encountered in reliability theory, neuroscience and logical computing devices [27]. A power index attempts to measures the ability of a player to determine the outcome of the vote.

The calculation of voting powers of the voters which is NP hard in all well known cases [23], has been extensively researched in the last few years and has interested computer scientists [14]. However, the inverse problem of designing a weighted voting system with a desirable distribution of power has received less attention. In this paper, we present an efficient algorithm to compute a corresponding integer vector for a given vector of Banzhaf Indices. This is a natural extension of the work on the method of generating functions to compute voting power indices. The algorithm is designed as a ready-made tool to be used by economists and political scientists in their analysis of weighted voting games. The tool is planned to be accessible from [21]. This algorithm has better performance than any other known to us. We have looked at designing multiple weighted voting games and also proposed further directions for research. Experiments with variations of the algorithm also promise to give better insight into the nature of the relationship between voting weights and corresponding voting powers.

1.2 Outline

Section 2 gives the definitions and notations of key terms and concepts covering voting power and weighted voting games. It also outlines how generating functions can be used to compute Banzhaf indices of players in a weighted voting game. Section 3 provides a survey of approaches to designing voting games and weighted voting games in particular. It also includes our main algorithm to design weighted voting games and its analysis. Section 4 highlights an application of our algorithm which is to use the Penrose Law to design 'egalitarian' weighted voting games. Similarly, Section 5 shows how our algorithm can be used to find the 'representative' single weighted voting game for a multiple weighted voting game. The final section presents conclusions.

2 Preliminaries

2.1 Voting Games

In this section we give definitions and notations of key terms. The set of voters is $N = \{1, ..., n\}$.

Definition 2.1. *A simple voting game is a pair (N, v) where $v : 2^N \rightarrow \{0, 1\}$ where $v(\emptyset) = 0$, $v(N) = 1$ and $v(S) \leq v(T)$ whenever $S \subseteq T$. A coalition $S \subseteq N$ is winning if $v(S) = 1$ and losing if $v(S) = 0$. A simple voting game can alternatively be defined as (N, W) where W is the set of winning coalitions.*

Definition 2.2. *The simple voting game (N, v) where $W = \{X \subseteq N, \sum_{x \in X} w_x \geq q\}$ is called a weighted voting game. A weighted voting game is denoted by $[q; w_1, w_2, ..., w_n]$ where w_i is the voting weight of player i. Generally $w_i \geq w_j$ if $i < j$.*

2.2 Voting Power Indices

Definition 2.3. *A player i is critical in a coalition S when $S \in W$ and $S \setminus i \notin W$. For each $i \in N$, we denote the number of coalitions in which i is critical in game v by $\eta_i(v)$. The Banzhaf Index of player i in weighted voting game v is $\beta_i = \frac{\eta_i(v)}{\sum_{i \in N} \eta_i(v)}$. The Probabilistic Banzhaf Index, β_i' of player i in game v is equal to $\eta_i(v)/2^{n-1}$.*

The worst case running time of a naive algorithm for computing the Banzhaf indices is in $O(n2^n)$.

2.3 Generating functions to compute Banzhaf Power

Algaba et al. [1] point out that Brams and Affuso [7] obtained generating functions for computing Banzhaf indices. Algaba et al. observed that for a weighted voting game $v = [q; w_1, ..., w_n]$, the number of coalitions in which a player is critical is $b_i = |\{S \subset N : v(S) = 0, v(S \cup \{i\}) = 1\}| = \sum_{k=q-w_i}^{q-1} b_k^i$ where b_k^i is the number of coalitions which do not include i and with total weight k.

Proposition 2.4. *(Brams-Affuso). Let $v = [q; w_1, ..., w_n]$ be a weighted voting game where $W = \sum_{1 \leq i \leq n} w_i$. Then the generating functions of numbers $\{b_k^i\}$ are given by $B_i(x) = \prod_{j=1, j \neq i}^{n} (1 + x^{w_j}) = 1 + b_1^i x + b_2^i x^2 + ... + b_{W-w_i}^i x$.*

Example 2.5. *Let $v = [6; 5, 4, 1]$ be a weighted voting game.*

- $B_1(x) = (1 + x^4)(1 + x^1) = 1 + x + x^4 + x^5$

  The coalitions in which player, 1 is critical are $\{1, 2\}$, $\{1, 3\}$, $\{1, 2, 3\}$. Therefore $\eta_1 = 3$

- $B_2(x) = (1 + x^5)(1 + x^1) = 1 + x + x^5 + x^6$ *The coalition in which player, 2 is critical is $\{1, 2\}$. Therefore $\eta_2 = 1$*

- $B_3(x) = (1 + x^5)(1 + x^4) = 1 + x^4 + x^5 + x^9$

  The coalition in which player, 3 is critical is $\{1, 3\}$. Therefore $\eta_3 = 1$

Consequently $\beta_1 = 3/5$, $\beta_2 = 1/5$ and $\beta_3 = 1/5$

The generating function method provides an efficient way of computing Banzhaf indices if the voting weights are integers [22].

3 Designing Weighted Voting Games

3.1 Outline and Survey

The problem of designing weighted voting games can be defined formally as follows:

Definition 3.1. *ComputeRealWeightsforGivenPowers: Given a real number vector of Banzhaf Indices, $P = (p_1, ..., p_n)$ for the n players, some appropriate Error function and ϵ, compute the corresponding real approximate weights $w = (w_1, ...w_n)$ such that Error$(P, P(w)) < \epsilon$.*

The problem of designing weighted voting games was first discussed in [**?**] and [20]. Holler et all. [20] proposed an iterative procedure with a stopping criterion to approximate to a game which has a voting power vector almost equal to the target. The method was to choose an initial weight vector w^0 and use successive iterations to get a better approximation: $w^{r+1} = w^r + \lambda(d - P(w^r))$ where λ is a scalar and $P(w^r)$ is the power vector of w^r. The approach has been used to analyse the Council of European Union and the International Monetary Fund Board of Governors.

Not every power distribution is feasible and might not have corresponding weights for it. There are some unexplored questions concerning the convergence of the vector, such as whether the iteration always converges to the right region. It is also critical to design systems with desirable properties.

Carreras [8] points out factors considered in designing simple games. The focus is different from the computation of powers and weights. By focusing on the protectionist tendency found in the design of voting games, the role of blocking coalitions is analysed in a simple game. Similarly, complexity results in designing simple games are provided in [26]. They show that it is NP-Complete to verify the 'stability' of a simple game.

3.2 Algorithm to design weighted voting games

We provide a more effective *hill climbing* approach than the previous proposed algorithms. Our algorithm tackled a variation of the problem *ComputeRealWeightsforGivenPowers*. The reason we are computing integer voting weights is that we want to utilize the generating function method in each iteration. The constraint of having integer weights is a reasonable assumption. Firstly many weighted voting games naturally have integer weights. Secondly some policy makers feel more comfortable dealing with integers. Thirdly our algorithm is giving results to high degree of accuracy even without using real or rational weights.

Definition 3.2. *ComputeIntegerWeightsforGivenPowers: Given a real number vector of Banzhaf Indices, $P = (p_1, ..., p_n)$ for the n players, some appropriate Error function and ϵ, compute the corresponding integer approximate weights $w = (w_1, ...w_n)$ such that $Error(P, P(w)) < \epsilon$.*

Proposition 3.3. *The power indices of players in weighted voting game $v = [q; w_1, ..., w_n]$ are the same as the power indices in the weighted voting game $\lambda v = [\lambda q; \lambda w_1, ..., \lambda w_n]$*

Proof. The proof is trivial. We notice that the set of coalitions for which

player i is critical is the same for both games v and λv. This means that the Banzhaf indices of players in both games are the same. □

Algorithm 1 ComputeIntegerWeightsforGivenPowers

Input: Target Vector of Powers, T

Output: Corresponding vector of voting weights

1: Use Normal Distribution Approximation to get an initial estimate of the weights
2: Multiply the real voting weights by a suitable real number λ which minimises the error while rounding to get new integer voting weights.
3: Use the Generating Function Method to compute new vector of voting powers
4: Interpolate by using a best fit quadratic curve to get the new real voting weights
5: Repeat Step 2 until the sum of squares of differences between the powers and the target powers is less that the maximum error.

3.3 Algorithmic and Technical Issues

The main aim of the algorithm is to use the generating function method to compute voting powers of estimated voting weights in a limited number of iterations. The generating function method is an elegant and efficient combinatorial method to compute Banzhaf indices of players with integer votes. Bilbao et al. [6] prove that the computational complexity of computing Banzhaf indices by generating functions is $O(n^2C)$ where C is the number of nonzero coefficients in $\prod_{1\leq i\leq n}(1+x^{w_j})$. Therefore in order to limit the computational complexity, we have tried to find moderate voting weights with values under 1000 in each iteration.

Since the generating function method can only be applied on integer votes, Algorithm 1 rounds off interpolated values to integer values. This rounding off can lead to varying errors if different potential multiples of the same real voting weight vector are used. After every interpolation step, we find a real λ which is multiplied with the voting weight vector and minimizes the total error on rounding. While finding the appropriate λ to multiply with the voting weights vector we minimize the sum of squares of the difference between new real voting weights and the rounded new voting weights. So for example if $w_1, w_2..., w_n$ are reals and m_i =Round$(\lambda w_i)\forall i \in N$, we want to minimize $\sum_{i\in N}(\frac{m_i}{M}-\frac{w_i}{W})^2$ where $M=\sum_{i\in N} m_i$ and $W=\sum_{i\in N} w_i$. All multiples of the same voting weight vector are equivalent (Proposition 3.3). It also appears reasonable to minimize the sum of the differences between the normalized voting weights and their corresponding rounded normalized voting weights. During interpolation, there is also the technical issue of having two possible voting weights with the same voting power. To avoid any errors in Mathematica, the Union function is used to delete pairs where the voting power is repeated.

The normal distribution approximation has been used to get an initial estimate of the voting weights. Leech [19] also uses the multi-linear extension approach in approximation of voting powers.

One extra degree of freedom which we have ignored is the variation in the quota. The same voting weights profile results in different Banzhaf indices according to the quota. The exact effects on the Banzhaf indices of changing the quota has various open problems.

One concern is the extra error induced when the interpolated weights are rounded off. From a relative movement point of view, it will be a bigger problem if most of the weights are rounded down and only a few weights are rounded up. Ideally, we will want the positive and negative differences in rounding to be balanced. The likelihood of this balance increases as we

use more players. However if the above mentioned problem arises, we would rather have those few weights which have been rounded up to also be rounded down. Some insight from the apportionment literature promises to shed light on how to deal with this technical issue. We are also looking for fresh computational approaches to apportionment. An overview of apportionment is available at [13] and [12].

3.4 Performance and Computational Complexity

As mentioned previously, the computational complexity of computing Banzhaf indices by generating functions is $O(n^2C)$ where C is the number of nonzero coefficients in $\prod_{1\leq i\leq n}(1+x^{w_j})$. We can approximate the number of iterations required based on the number of significant figures required in our final solution. However for practical purposes, our algorithm is giving an error of less than 10^{-8} for 30 players after only 3 iterations.

4 Designing Egalitarian Voting Games

We have the data of the latest EU Council members and we are running some experiments which use our algorithm to design integer votes for EU members if new countries are given membership. Felsenthal and Machover [9] have obtained the following result for a two-tier voting system based on Penrose' seminal paper [24].

Theorem 4.1. *(Felsenthal and Machover) Let v be a 2-tier voting system in which m delegates of m different states with populations $\{n_1, ..., n_m\}$ vote in a weighted voting game as the representative of their states. The probabilistic Banzhaf indices $\beta_i^{'}$ are equal for all the population voters if and only if the probabilistic Banzhaf indices $\beta_i^{'}$ of the delegates are proportional to the respective $\sqrt{n_i}$*

This is an approximation as n_i tends to infinity. The assumption of the Penrose square root law is that 'yes' and 'no' are equiprobable and the voters are totally independent. Based on this result, we can devise an algorithm to compute voting weights of countries so that every member of any country has approximately equal voting power.

Algorithm 2 ComputeFairIntegerWeightsforGivenPopulations

Input: Vector of State Populations, $p = \{n_1, ..., n_m\}$

Output: Corresponding vector of voting weights $w = \{w_1, ..., w_n\}$

1: Let $\beta' = \{\sqrt{n_1}, ..., \sqrt{n_m}\}$ and $B = \sum_{1 \leq i \leq m} \sqrt{n_i}$
2: Target powers, $T = \{\sqrt{n_1}/B, ..., \sqrt{n_m}/B\}$
3: Run Algorithm 1 with input T and return the output

W Slomczynski and K. Zyczkowski [25] have proposed that giving each nation a vote proportional to the square root of its population and establishing a quota rule equal to around 62% of the population makes the voting rule almost egalitarian. Although this voting method appears to be elegant and transparent, Algorithm 2 provides an alternative in which we can change the quota to accomodate various levels of efficiency to make a decision. We used Algorithm 2 with simple majority quota to compute a sample vector of egalitarian voting weights for the EU countries. The populations data is available from [16]. We use our algorithm to compute 'egalitarian' weights in case Turkey joins EU:

State	Population	$\sqrt{(population)}$	Target Banzhaf Indices	Voting Weights
Belgium	10570500	3251.230536	0.030050536	30
Bulgaria	7666500	2768.844524	0.02559193	26
Czech Republic	10288900	3207.631525	0.029647559	30
Denmark	5445700	2333.602365	0.021569065	22
Germany	82311700	9072.579567	0.083856213	81
Estonia	1339900	1157.540496	0.010698938	11
Ireland	4326700	2080.072114	0.019225731	20
Greece	11169100	3342.020347	0.03088969	31
Spain	44484300	6669.655163	0.061646417	61
France	63336300	7958.410645	0.073558151	72
Italy	58933800	7676.835285	0.070955601	70
Cyprus	776000	880.9086218	0.008142079	8
Latvia	2280500	1510.132445	0.013957881	14
Lithuania	3385700	1840.027174	0.017007039	17
Luxembourg	464400	681.4690015	0.006298695	6
Hungary	10057900	3171.419241	0.029312855	30
Malta	407700	638.5138996	0.005901669	6
Netherlands	16346200	4043.043408	0.03736912	38
Austria	8295900	2880.260405	0.026621726	27
Poland	38101800	6172.665551	0.057052832	57
Portugal	10609000	3257.14599	0.030105212	30
Romania	21570600	4644.416002	0.042927498	43
Slovenia	2010300	1417.850486	0.013104936	13
Slovakia	5391600	2321.981912	0.021461659	22
Finland	5277100	2297.19394	0.021232548	22
Sweden	9119800	3019.900661	0.027912396	28
United Kingdom	60707100	7791.476112	0.072015206	71
Croatia	4439.8	2107.083292	0.01947539	20
R Macedonia	2042200	1429.055632	0.013208503	13
Turkey	73430000	8569.130644	0.079202926	77
TOTAL	574587000	108192.097	1	996

5 Multiple weighted voting games

A weighted m-majority game is the simple game $(N, v_1 \wedge ... \wedge v_m)$ where the games (N, v_t) are the weighted majority games $[q^t; w_1^t, ...w_n^t]$ for $1 \leq t \leq m$. Then $v = v_1 \wedge ... \wedge v_m$ is defined as:

$$(v_1 \wedge ... \wedge v_m)(S) = \begin{cases} 1, & \text{if } \sum_{i \in S} {w^t}_i \geq q^t, \text{ for all } t,\ 1 \leq t \leq m \\ 0, & \text{otherwise} \end{cases}$$

The treaty of Nice makes the overall voting games of the EU countries a triple majority weighed voting game with certain additional constraints. Algaba et al. [2] outline a generating function method to find the Banzhaf indices of players in a multiple weighted majority game. Their algorithm m-banzhafPower computes the Banzhaf index of the players in O(max(m, n^2c)) time where c is the number of terms of $B(x_1, ..., x_m) = \prod_{j=1}^{n}(1 + x_1^{w_j^1}...x_m^{w_j^m})$

Our Algorithm 1 can be used to produce an approximate single majority weighted game as a representative for a double majority weighted voting game.

SingleWeightedVotingGameForMultipleGames:

Algorithm 3 SingleWeightedVotingGameForMultipleGames

Input: Multiple weighted voting game $(N, v_1 \wedge ... \wedge v_m)$

Ouput: Corresponding weighted voting game

1: Use Algorithm m-banzhafPower to compute vector of Banzhaf indices, $\{\beta_1, ..., \beta_n\}$
2: Run Algorithm 1 to compute the corresponding weighted voting game v
3: Return v

The questions we are interested in exploring are: Is there a way of directly transforming a multiple majority weighted game into a weighted voting game with the same voting powers. Is there any loss of information in the transformation? Is it possible to identify and remove redundant weighted games from the multiple majority weighted voting game?

6 Conclusion

This paper provides an algorithm which will be useful for practitioners in the voting power field. Moreover we analyse computational considerations

which will be of interest to computer scientists. We notice that our algorithm can be used to design egalitarian two-tier weighted voting games and also find the 'representative weighted voting game' for multiple weighted voting games.

The computational complexity of Algorithm 1 remains an open problem. Moreover the voting rules in the EU Council comprise of multiple weighted voting games. It is an interesting problem to analyse multiple voting games as a function of its constituent single weighted voting games.

References

[1] E. Algaba, J. M. Bilbao, and J. Fernandez. The distribution of power in the *European Constitution*. *European Journal of Operational Research*, 176(3):1752–1755, 2007.

[2] E. Algaba, J. M. Bilbao, J. R. Fernandez Garcia, and J. J. Lopez. Computing power indices in weighted multiple majority games. *Mathematical Social Sciences*, 46(1):63–80, 2003. available at http://ideas.repec.org/a/eee/matsoc/v46y2003i1p63-80.html.

[3] J. Alonso-Meijide. Generating functions for coalition power indices: An application to the *IMF*. *Annals of Operations Research*, 137:21–44, 2005.

[4] G. Arcaini and G. Gambarelli. Algorithm for automatic computation of the power variations in share tradings. *Calcolo*, 23(1):13–19, January 1986.

[5] J. Bilbao, J. Fernandez, N. Jimenez, and J. Lopez. Voting power in the *European Union Enlargement*. *European Journal of Operational Research*, 143(1):181–196, 2002.

[6] J. M. Bilbao, J. R. Fernandez, A. J. Losada, and J. J. Lopez. Generating functions for computing power indices efficiently. *citeseer.ist.psu.edu/452269.html*, 2000.

[7] S. F. Brams and P. J. Affuso. Power and size: A new paradox. *Theory and Decision*, 7:29–56, 1976.

[8] F. Carreras. On the design of voting games. *Mathematical Methods of Operations Research*, 59(1):503–515, 2004.

[9] D. Felsenthal and M. Machover. *The Measurement of Voting Power*. Edward Elgar Publishing, Cheltenham, UK, 1998.

[10] D. S. Felsenthal and M. Machover. Analysis of *QM* rules in the draft constitution for *Europe* proposed by the *European Convention*, 2003. *Social Choice and Welfare*, 23(1):1–20, 08 2004. available at http://ideas.repec.org/a/spr/sochwe/v23y2004i1p1-20.html.

[11] G. Gambarelli. Power indices for political and financial decision making: A review. *Annals of Operations Research*, 51:1572–9338, 1994.

[12] E. Hemaspaandra and L. Hemaspaandra". Computational politics: electoral systems. *In Mathematical foundations of computer science*, 2000. available at http://citeseer.comp.nus.edu.sg/hemaspaandra00computational.html.

[13] L. A. Hemaspaandra, K. S. Rajasethupathy, P. Sethupathy, and M. Zimand. Power balance and apportionment algorithms for the *United States Congress*. *ACM Journal of Experimental Algorithms*, 3:1, 1998.

[14] D. Johnson. Challenges for theoretical computer science. *http://www.research.att.com/ dsj/nsflist.html*, 2000.

[15] J.-E. Lane and R. Maeland. Constitutional analysis: The power index approach. *European Journal of Political Research*, 37:31–56, 2000.

[16] G. Lanzeiri. Population and social conditions. *Statistics in focus, Eurostat, European Commission*, 41, 2007.

[17] A. Laruelle and M. Widgren. Is the allocation of voting power among *EU* states fair? *Public Choice*, 94(3-4):317–39, March 1998. available at http://ideas.repec.org/a/kap/pubcho/v94y1998i3-4p317-39.html.

[18] D. Leech. Voting power in the governance of the international monetary fund. *Annals of Operations Research*, 109(1):375–397, 2002.

[19] D. Leech. Computing power indices for large voting games. *Management Science*, 49(6):831–837, 2003.

[20] D. Leech. Power indices as an aid to institutional design: the generalised apportionment problem. *Yearbook on New Political Economy, edited by M. Holler, H.Kliemt, D. Schmidtchen and M. Streit.*, 2003.

[21] D. Leech. Voting power algorithms website. *http://www.warwick.ac.uk/ ecaae/*, 2007.

[22] T. Matsui and Y. Matsui. A survey of algorithms for calculating power indices of weighted majority games. *Journal of the Operations Research Society of Japan*, 43(7186), 2000. available at http://citeseer.ist.psu.edu/matsui00survey.html.

[23] Y. Matsui and T. Matsui. NP-completeness for calculating power indices of weighted majority games. *Theoretical Computer Science*, 263(1-2):305–310, 2001. available at citeseer.ist.psu.edu/matsui98npcompletenes.html.

[24] L. S. Penrose. The elementary statistics of majority voting. *Journal of Royal Statistical Society*, 109:40–63, 1946.

[25] W. Slomczynski and K. Zyczkowski. From a toy model to the double square root voting system. *ArXiv Physics e-prints*, Jan. 2007.

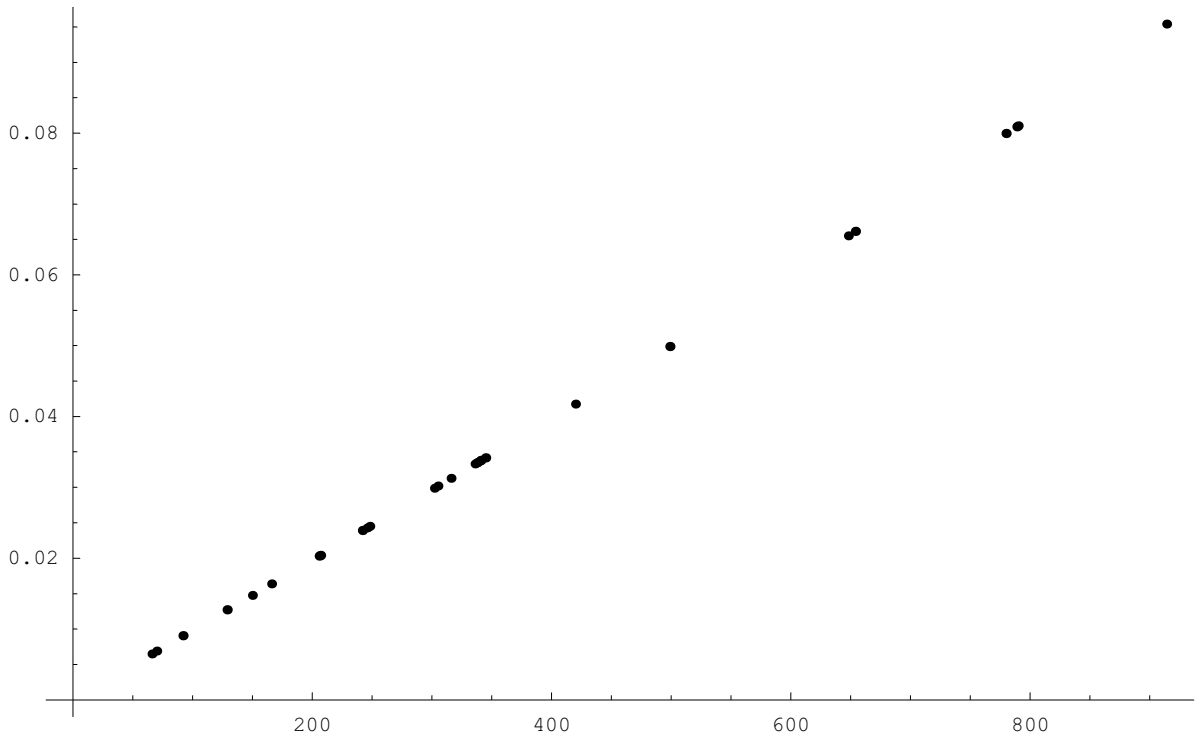
[26] K. Takamiya and A. Tanaka. Computational complexity in the design of voting games. *The Institute of Social and Economic Research, Osaka University*, 653, 2006.

[27] A. Taylor and W. Zwicker. *Simple Games: Desirability Relations, Trading, Pseudoweightings*. Princeton University Press, New Jersey, first edition edition, 1999.

Appendix

```
(*:Mathematica Version:5.2*)
(*:Package Version:1.10*)
(*:Name:Compute_Banzhaf _Indices _of _MWVG *)
(*:Title: Compute Integer Weights for given Powers*)
(*:Authors:
   Mike Paterson (msp@dcs.warwick.ac.uk) and Haris Aziz (haris.aziz@warwick.ac.uk) *)
(*:Copyright:*)
(* The first part should be familiar. The rest has some experiments and
  graphs and then shows how to semiautomate the finding of good "lambdas". *)

Off[InterpolatingFunction::"dmval", General::"spell1", NumberForm::"sigz"];
Clear[P, RW, V];
g[v_] := Apply[Times, Map[(1 + x^#) &, v]];
s[r_] := Normal[Series[1 / (1 - x), {x, 0, r - 1}]];
h[r_, v_] := Expand[g[v] / (1 + x^r)];
p[r_, v_] := Coefficient[s[r] h[r, v], x^Round[Total[v] / 2 - 1]];
(* strict majority rule *)
Ind[v_] := Map[p[#, v] &, v]; (* raw index *)
NBI[v_] := (temp = Ind[v]; temp / N[Total[temp]]); (* normalized index *)
FirstEqual[u_, v_] := u[[1]] == v[[1]];
F[r_, RW_] := Total[(FractionalPart[RW r + 1 / 2] - 1 / 2)^2];

T = {0.0954, 0.0810, 0.0809, 0.0799, 0.0661, 0.0655, 0.0499, 0.0418, 0.0342,
   0.0338, 0.0337, 0.0335, 0.0333, 0.0313, 0.0302, 0.0299, 0.0245, 0.0243,
   0.0239, 0.0204, 0.0203, 0.0164, 0.0148, 0.0127, 0.0091, 0.0069, 0.0065};
P[n_] := P[n] = NBI[V[n]] 1.0002; (* correcting for target excess *)
DT[n_] := P[n] - T;
Err[n_] := Total[DT[n]^2];
RW[n_] :=
  RW[n] = Map[Interpolation[Union[Transpose[{P[n - 1], V[n - 1]}], SameTest → FirstEqual],
     InterpolationOrder → 2], T];
Go[n_] := Print["Error = ", NumberForm[Err[n], 4], ";
Diff = ", NumberForm[DT[n], 3], "; Next raw weights = ", RW[n + 1]];
V[0] = {914, 790, 789, 780, 654, 648, 499, 420, 345, 341, 340, 338,
  336, 316, 305, 302, 248, 246, 242, 207, 206, 166, 150, 129, 92, 70, 66}

{914, 790, 789, 780, 654, 648, 499, 420, 345, 341, 340, 338,
 336, 316, 305, 302, 248, 246, 242, 207, 206, 166, 150, 129, 92, 70, 66}

Go[0]

Error = 1.232×10^-8;  Diff =
 {0.0000215, 0.0000443, 0.0000328, 0.0000266, 0.000038, -4.82×10^-7, -0.000016, -0.0000269,
  -1.56×10^-6, -3.73×10^-6, -3.24×10^-6, -2.74×10^-6, -2.32×10^-6, -0.0000197, -0.0000122,
  -9.89×10^-6, -7.67×10^-6, -4.69×10^-6, 2.55×10^-7, 0.0000242, 0.0000251, -0.0000358,
  -0.0000332, 0.0000124, -0.0000324, -0.0000107, -2.04×10^-6}; Next raw weights =
 {913.821, 789.603, 788.706, 779.763, 653.643, 648.005, 499.155, 420.265, 345.015,
  341.037, 340.032, 338.027, 336.023, 316.194, 305.123, 302.1, 248.078, 246.048,
  241.997, 206.755, 205.747, 166.362, 150.336, 128.874, 92.327, 70.109, 66.0209}
```

```
Inter[n_] := ListPlot[Transpose[{V[n], P[n]}]];
Inter[0]
```



- Graphics -

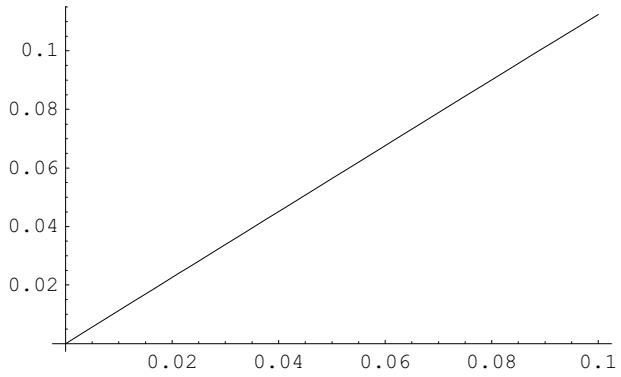
```
F0 = Interpolation[
  Union[Transpose[{P[0], V[0]}], SameTest → FirstEqual], InterpolationOrder → 2]
```

```
InterpolatingFunction[{{0.00649796, 0.0954215}}, <>]
```

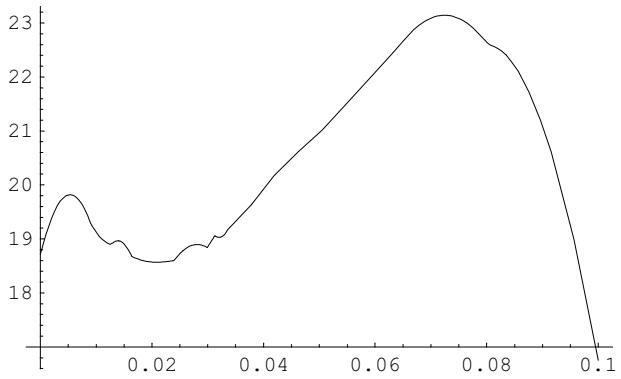
```
Total[V[0]]
```

```
9939
```

```
Plot[Erf[x], {x, 0, 0.1}];
```




```
(* We look at how well the Erf
  approximation does compared with quite a good solution *)

Plot[F0[x] - 8500 Erf[x] + 8500 (x - 0.05)^2, {x, 0, 0.1}, PlotRange → All];
```




```
V[1] = Round[RW[1] 1.01556]
```

```
{928, 802, 801, 792, 664, 658, 507, 427, 350, 346, 345, 343,
 341, 321, 310, 307, 252, 250, 246, 210, 209, 169, 153, 131, 94, 71, 67}
```

```
Go[1]
```

```
Error = 8.091×10^-8;  Diff =
 {0.000166, 0.0000803, -0.0000355, 0.0000913, 8.27×10^-6, -0.0000123, 0.0000443, -0.0000714,
  0.0000391, 0.0000457, -0.000047, -0.0000426, -0.0000359, -7.36×10^-6, -0.0000693,
  0.0000383, -0.0000241, -0.0000172, -2.43×10^-6, 0.0000106, -0.000082, -0.0000471,
  -0.0000128, -5.85×10^-6, -0.0000578, 0.0000142, 0.0000328}; Next raw weights =
 {468.301, 404.627, 404.164, 399.586, 334.96, 332.06, 255.781, 215.36, 176.804,
  174.762, 174.243, 173.217, 172.187, 162.037, 156.36, 154.801, 127.122, 126.089,
  124.013, 105.945, 105.426, 85.2431, 77.066, 66.0313, 47.2986, 35.9257, 33.8277}
```

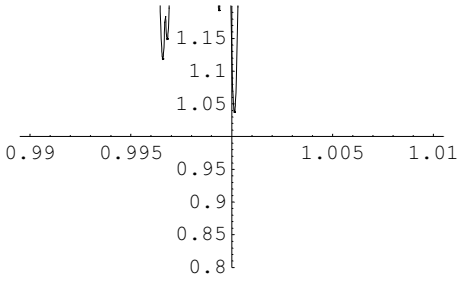
```
V[2] = Round[RW[2] 1.9818]
```

```
{928, 802, 801, 792, 664, 658, 507, 427, 350, 346, 345, 343,
 341, 321, 310, 307, 252, 250, 246, 210, 209, 169, 153, 131, 94, 71, 67}
```

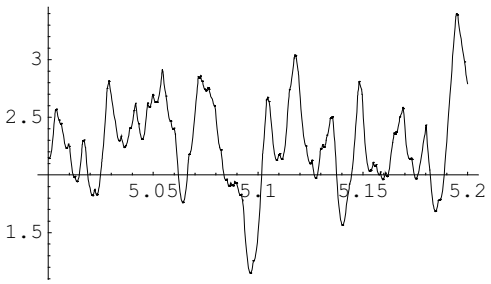
Go[2]

```
Error = 9.605×10^-9;  Diff = {-5.29×10^-6, 0.000012, 1.81×10^-6, 0.0000107, 0.0000191,
  -0.0000103, 4.55×10^-6, 0.0000191, -0.0000386, -0.000034, -0.0000325, -0.000029,
  -0.0000261, -0.0000117, 0.0000125, 0.0000183, 6.31×10^-6, 0.0000118, 0.0000227, 2.35×10^-6,
  4.87×10^-6, 4.05×10^-6, 0.0000299, 0.0000108, 0.0000211, -0.0000193, -5.21×10^-6}
 ; Next raw weights = {928.045, 801.891, 800.984, 791.903, 663.818, 658.099, 506.955,
  426.809, 350.39, 346.345, 345.33, 343.296, 341.265, 321.117, 309.872, 306.816, 251.935,
  249.879, 245.768, 209.976, 208.95, 168.959, 152.693, 130.888, 93.7852, 71.2, 67.054}
```

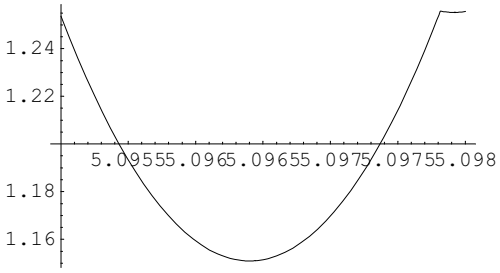
Plot[F[r, RW[1]], {r, 0.99, 1.01}, PlotPoints → 500, PlotRange → {0.8, 1.2}];



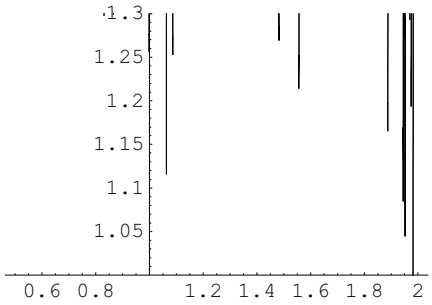
Plot[F[r, RW[1]], {r, 5, 5.2}, PlotPoints → 500];



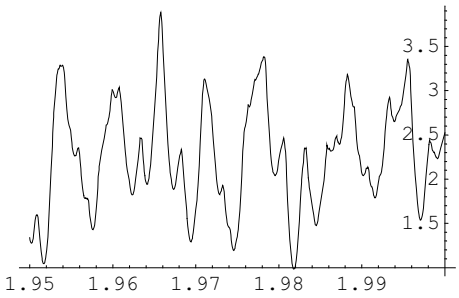
Plot[F[r, RW[1]], {r, 5.095, 5.098}, PlotPoints → 50];



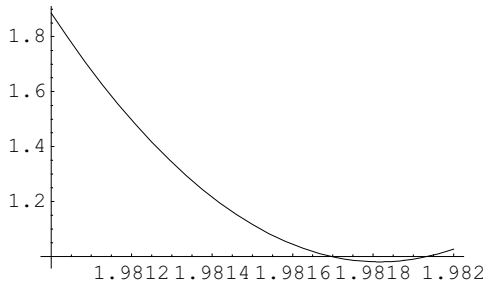
```
Plot[F[r, RW[2]], {r, 0.5, 2}, PlotPoints → 500, PlotRange → {1, 1.3}];
```




```
Plot[F[r, RW[2]], {r, 1.95, 2}];
```




```
Plot[F[r, RW[2]], {r, 1.981, 1.982}];
```




```
(* Ordering[X,1] gives the position of the minimum element in list X *)

Ordering[Table[F[r / 1000, RW[1]], {r, 501, 6000}], 1] + 500
```

```
{5096}
```

```
(* We add 500 because we are starting the list at 501 *)

Ordering[Table[F[r / 1000, RW[2]], {r, 501, 3000}], 1] + 500
```

```
{1982}
```

```
(* Below we are focussing on a small interval around the
  minimum and looking for the minimum of the smooth function there *)

(m = (Ordering[Table[F[r / 1000, RW[2]], {r, 501, 3000}], 1] + 500)[[1]] / 1000;
 Minimize[{F[r, RW[2]], r < m + 0.001, r > m - 0.001}, r])
```

```
{0.980609, {r → 1.98182}}
```

```
(m = (Ordering[Table[F[r / 1000, RW[1]], {r, 1001, 6000}], 1] + 1000)[[1]] / 1000;
 Minimize[{F[r, RW[1]], r < m + 0.001, r > m - 0.001}, r])
```

```
{1.15091, {r → 5.0964}}
```

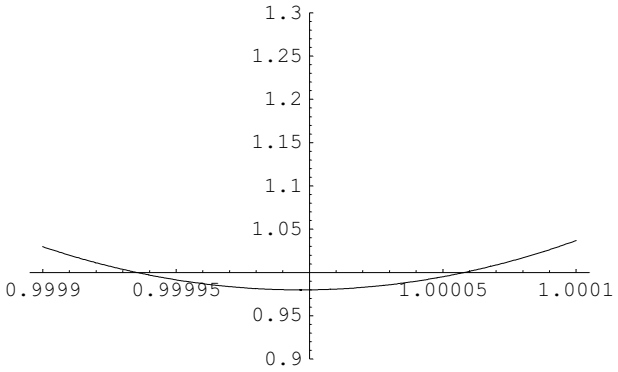
```
RW = {928.0446756879035`, 801.890975741139`, 800.9835681353658`,
  791.9032510340901`, 663.8181327829079`, 658.0987278456653`, 506.955311026178`,
  426.80883803717495`, 350.38999175795306`, 346.34502009479246`, 345.3303576423212`,
  343.29569580991506`, 341.26489700865017`, 321.1169046165965`, 309.87198782161033`,
  306.8160124180501`, 251.93514864574425`, 249.87850342581248`, 245.76838926170723`,
  209.97587499725208`, 208.95006944496586`, 168.9592161438174`, 152.69292442323265`,
  130.88808284765457`, 93.7851998285877`, 71.20003053704546`, 67.05404214163751`}
```

```
{928.045, 801.891, 800.984, 791.903, 663.818, 658.099, 506.955, 426.809, 350.39,
 346.345, 345.33, 343.296, 341.265, 321.117, 309.872, 306.816, 251.935, 249.879,
 245.768, 209.976, 208.95, 168.959, 152.693, 130.888, 93.7852, 71.2, 67.054}
```

```
Ordering[Table[F[r / 1000, RW], {r, 501, 2000}], 1] + 500
```

```
{1000}
```

```
Plot[F[r, RW], {r, 0.9999, 1.0001}, PlotPoints → 500, PlotRange → {0.9, 1.3}];
```




```
Sums[{}] = {0};
Sums[v_] := Sums[v] = Union[Sums[Drop[v, 1]], v[[1]] + Sums[Drop[v, 1]]];
```

```
A20 = Sums[{928.0446756879035`, 801.890975741139`, 800.9835681353658`,
    791.9032510340901`, 663.8181327829079`, 658.0987278456653`, 506.955311026178`,
    426.80883803717495`, 350.38999175795306`, 346.34502009479246`,
    345.3303576423212`, 343.29569580991506`, 341.26489700865017`, 321.1169046165965`,
    309.87198782161033`, 306.8160124180501`, 251.93514864574425`,
    249.87850342581248`, 245.76838926170723`, 209.97587499725208`}];
```

```
RW
```

```
{928.045, 801.891, 800.984, 791.903, 663.818, 658.099, 506.955, 426.809, 350.39,
 346.345, 345.33, 343.296, 341.265, 321.117, 309.872, 306.816, 251.935, 249.879,
 245.768, 209.976, 208.95, 168.959, 152.693, 130.888, 93.7852, 71.2, 67.054}
```

```
{928.0446756879035`, 801.890975741139`, 800.9835681353658`, 791.9032510340901`,
 663.8181327829079`, 658.0987278456653`, 506.955311026178`, 426.80883803717495`,
 350.38999175795306`, 346.34502009479246`, 345.3303576423212`, 343.29569580991506`}
```

```
Length[A20] - 2^20
```

```
0
```

```
A20[[1]]
```

```
0
```

```
A20[[3]]
```

```
245.768
```

```
D1 = Drop[A20, 1] - Drop[A20, -1];
```

```
Min[D1]
```

2.0912×10^{-6}

```
P[2]
```