Basic Voting Theory

Jan van Eijck and Floor Sietsma

Abstract

Basic concepts of voting theory, implemented in literate Haskell style, with some (hopefully) new results.

1 Preliminaries

Voting is the process of selecting an item or a set of items from a finite set A of alternatives, on the basis of the stated preferences of a set of voters.

The set of m alternatives $A = \{a_0, \ldots, a_{m-1}\}$ is represented as $\{0, \ldots, m - 1\}$, for $m = 2, 3, 4, \ldots$.

A ballot is a linear ordering of A. We assume that the preferences of a voter are represented by a ballot. A profile is a vector of ballots, one for each voter. We assume voter anonymity, so it does not matter which voter has which ballot. The only thing that matters is the number of voters holding a certain ballot. Under this assumption voting profiles can be represented as mappings from ballots to non-negative integers.

We will use $\mathbf{P}, \mathbf{Q}$ to range over (quantified) profiles, and $\mathbf{b}, \mathbf{b}'$ to range over ballots.

This paper is a literate program [Knu92] written in Haskell [Jon03, HT], with the programming code appearing in boxed typescript.

```
module Voting

where

import List
import Ratio
```

Type declarations for readability:

```
type Alternative = Int
type Ballot = [Alternative]
type Profile = [Int]
```

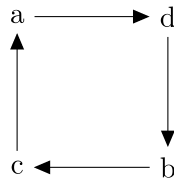
2 Cycles

Definition 1 *A permutation of alternatives* π *on* $A = \{a_0, \ldots, a_{m-1}\}$ *is a* full cycle *if* π *can be given as* $a_0 = \pi^0(a_0) \mapsto \pi(a_0) \mapsto \pi^2(a_0) \mapsto \cdots \mapsto \pi^{m-1}(a_0)$*, with the* $\pi^i(a_0)$ *all different.*

Any full cycle on A can be considered as a linear ordering on A with a_0 as least element, and vice versa. Thus, there are $(m-1)!$ full cycles on $\{a_0, \ldots, a_{m-1}\}$. Customary notation for full cycles π on a list of m elements is to give the list:

$$(a_0, \pi(a_0), \pi^2(a_0), \cdots, \pi^{m-1}(a_0)).$$

For example, the full cycle in the following picture can be given as $(adbc)$.



```
cycles :: Int -> [[Int]]
cycles n = map (0:) (perms [1..n-1])

listCycles :: [a] -> [[a]]
listCycles xs = let
   n = length xs
   lst = cycle xs
   cls ys = take n ys : cls (drop (n+1) ys)
  in take n (cls lst)
```

All permutations of a list are given by:

```
perms :: [a] -> [[a]]
perms [] = [[]]
perms (x:xs) = concat (map (insrt x) (perms xs))
  where
    insrt :: a -> [a] -> [[a]]
    insrt x [] = [[x]]
    insrt x (y:ys) = (x:y:ys) : map (y:) (insrt x ys)
```

The $(m-1)!$ cycles in a profile for m alternatives are found by:

```
getCycle :: Int -> Int -> [[Int]]
getCycle m k = listCycles ((cycles m) !! k)
```

Or with characters:

```
getCycle' :: Int -> Int -> [[Char]]
getCycle' m k = map (map int2chr) (getCycle m k)
```

```
int2chr :: Int -> Char
int2chr n = toEnum (n + 97)
```

This gives, e.g.:

```
*Voting> getCycle' 4 0
["abcd","bcda","cdab","dabc"]
*Voting> getCycle' 4 1
["acbd","cbda","bdac","dacb"]
*Voting> getCycle' 4 2
["acdb","cdba","dbac","bacd"]
*Voting> getCycle' 4 3
["abdc","bdca","dcab","cabd"]
*Voting> getCycle' 4 4
["adbc","dbca","bcad","cadb"]
*Voting> getCycle' 4 5
["adcb","dcba","cbad","badc"]
```

It is easy to generate the list of all ballots, given the number of alternatives.

```
fac :: Int -> Int
fac m = product [1..m]
```

```
genBallots :: Int -> [Ballot]
genBallots m = let
    k = fac (m-1) - 1
  in concat [ getCycle m i | i <- [0..k] ]
```

Or if you wish to see the ballots as strings "abcd" and so on:

```
genBallots' :: Int -> [String]
genBallots' = map (map int2chr) . genBallots
```

Here are some examples:

```
*Voting> genBallots' 2
["ab","ba"]
*Voting> genBallots' 3
["abc","bca","cab","acb","cba","bac"]
*Voting> genBallots' 4
["abcd","bcda","cdab","dabc","acbd","cbda","bdac","dacb","acdb",
"cdba","dbac","bacd","abdc","bdca","dcab","cabd","adbc","dbca",
"bcad","cadb","adcb","dcba","cbad","badc"]
*Voting> genBallots' 5
["abcde","bcdea","cdeab","deabc","eabcd","acbde","cbdea","bdeac",
"deacb","eacbd","acdbe","cdbea","dbeac","beacd","eacdb","acdeb",
"cdeba","debac","ebacd","bacde","abdce","bdcea","dceab","ceabd",
"eabdc","adbce","dbcea","bcead","ceadb","eadbc","adcbe","dcbea",
"cbead","beadc","eadcb","adceb","dceba","cebad","ebadc","badce",
"abdec","bdeca","decab","ecabd","cabde","adbec","dbeca","becad",
"ecadb","cadbe","adebc","debca","ebcad","bcade","cadeb","adecb",
"decba","ecbad","cbade","badec","abced","bceda","cedab","edabc",
"dabce","acbed","cbeda","bedac","edacb","dacbe","acebd","cebda",
"ebdac","bdace","daceb","acedb","cedba","edbac","dbace","baced",
"abecd","becda","ecdab","cdabe","dabec","aebcd","ebcda","bcdae",
"cdaeb","daebc","aecbd","ecbda","cbdae","bdaec","daecb","aecdb",
"ecdba","cdbae","dbaec","baecd","abedc","bedca","edcab","dcabe",
"cabed","aebdc","ebdca","bdcae","dcaeb","caebd","aedbc","edbca",
"dbcae","bcaed","caedb","aedcb","edcba","dcbae","cbaed","baedc"]
```

Notice that the number of possible ballots grows very rapidly with the number of alternatives. For m alternatives there are $m!$ possible ballots.

Profiles are represented as lists of non-negative integers, where the length of the list equals $m!$, with m the number of alternatives. The size of a profile is equal to the length of its ballots. If a profile $\mathbf{P}$ has size m, this means that its alternative set A has $|A| = m$.

```
size :: Profile -> Int
size xs = length $ fst $ findBallot xs 0
```

The integer at position p in the list indicates the number of voters with the ballot at position p in the list of all possible ballots:

```
findBallot :: Profile -> Int -> (Ballot,Int)
findBallot xs n = let
    m = length xs
    factorials = [ fac k | k <- [2..] ]
    candidates = takeWhile (\x -> x <= m) factorials
    lst        = last candidates
    k          = length candidates + 1
  in
    if m == lst then (genBallots k!!n,xs!!n)
    else error "incorrect profile length"
```

Or in terms of strings:

```
findBallot' :: Profile -> Int -> (String,Int)
findBallot' xs n = let
    m = length xs
    factorials = [ fac k | k <- [2..] ]
    candidates = takeWhile (\x -> x <= m) factorials
    lst        = last candidates
    k          = length candidates + 1
  in
    if m == lst then (genBallots' k!!n,xs!!n)
    else error "incorrect profile length"
```

It is sometimes useful to expand a profile to a list of (ballot,int) pairs:

```
expand :: Profile -> [(Ballot,Int)]
expand xs = map (findBallot xs) [0..length xs-1]
```

Or in terms of strings:

```
expand' :: Profile -> [(String,Int)]
expand' xs = map (findBallot' xs) [0..length xs-1]
```

If **b** is a ballot and **P** a profile, we use **P**(**b**) for the number of voters with ballot **b** in **P**. Here is the implementation:

```
votes :: Profile -> Ballot -> Int
votes profile ballot = let
    eprofile = expand profile
    Just k = lookup ballot eprofile
  in
    k
```

The vote size (total number of voters) in a profile:

```
voteSize :: Profile -> Int
voteSize = sum
```

3 Profile Normalization

Profiles can be normalized by dividing with the gcd of the list of all nonzero vote numbers.

```
norm :: Profile -> Profile
norm profile =
  let
    xs = filter (/= 0) profile
    k  = if null xs then 1 else foldl1 gcd xs
    f  = flip div k
  in map f profile
```

If **P** is a profile, we use $\mathbf{P}^\circ$ for the normalized form of the profile.

If we are prepared to use fractions, we can normalize still further:

```
type Nprofile = [Ratio Int]
```

Converting to a normalized profile:

```
nrm :: Profile -> Nprofile
nrm profile = let
   total = sum profile
 in
   map (\ k -> (k % total)) profile
```

This gives:

```
*Voting> norm [2,0,0,2,6,4]
[1,0,0,1,3,2]
*Voting> nrm [2,0,0,2,6,4]
[1 % 7,0 % 1,0 % 1,1 % 7,3 % 7,2 % 7]
```

4 Some Special Profiles

The empty m-profile:

```
nullprofile :: Int -> Profile
nullprofile m = take (fac m) (repeat 0)
```

Unit m-profiles:

```
unit :: Int -> Int -> Profile
unit m i = let
   f = \ (x,y) -> if y == i then (x+1) else x
 in
   map f (zip (nullprofile m) [0..])
```

This gives:

```
*Voting> unit 3 0
[1,0,0,0,0,0]
*Voting> unit 3 1
[0,1,0,0,0,0]
*Voting> unit 4 2
[0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
```

5 Voting Rules, Resolute Voting Rules, Tie Breaking

Voting rules are functions from profiles to non-empty sets of alternatives (sets represented as lists), or from profiles to non-empty sets of characters. If V is a voting rule and $\mathbf{P}$ a profile, we call $V(\mathbf{P})$ the V-winners for $\mathbf{P}$.

```
type VotingRule = Profile -> [Alternative]
type VotingRule' = Profile -> [Char]
```

Resolute voting rules are functions from profiles to alternatives (or characters). If V is a resolute voting rule and $\mathbf{P}$ a profile, we call $V(\mathbf{P})$ the V-winner for $\mathbf{P}$.

```
type ResVotingRule  = Profile -> Alternative
type ResVotingRule' = Profile -> Char
```

Tie breaking can be viewed as a method for mapping voting rules to resolute voting rules: in case the result of the voting rule is not a singleton, apply the tie break order to select a unique winner.

```
tieBreak :: [Alternative] -> VotingRule -> ResVotingRule
tieBreak tiebreaklist f profile = let
    results = f profile
    m       = size profile
    order   = list2ordering (take m tiebreaklist)
  in
    if null results then error "no winners selected"
    else if length results == 1 then head results
    else head (sortBy order results)
```

Note: the reason for cutting off the tie break list after the n-th element is to enable use of the infinite list [0..] as a generic tie break argument.

From a tie break list to an ordering. Head of the list is top (most prominent), which corresponds to lowest in the ordering. We are going to use the ordering for sorting.

```
list2ordering :: Eq a => [a] -> a -> a -> Ordering
list2ordering xs x y | x == y  = EQ
                     | elem x (dropWhile (/= y) xs) = GT
                     | otherwise = LT
```

If $\mathbf{P}$ is a profile for A, and π is a permutation of A, then $\mathbf{P}^\pi$ is the result of replacing x by $\pi(x)$ everywhere in $\mathbf{P}$. If $B \subseteq A$, then $\pi(B) = \{\pi(x) \mid x \in B\}$.

Definition 2 *A voting rule V is* neutral *if for every profile* $\mathbf{P}$ *and for every*

permutation π of the set A of alternatives,

$$V(\mathbf{P}^{\pi}) = \pi(V(\mathbf{P})).$$

Recall that $\mathbf{P}^{\circ}$ is the normalized form of profile $\mathbf{P}$.

Definition 3 *A voting rule V is* normal *if it holds for every profile $\mathbf{P}$ that $V(\mathbf{P}) = V(\mathbf{P}^{\circ})$.*

Proposition 4 *There are anonomymous and neutral voting rules that are not normal.*

Proof. Let V_k be given by $x \in V_k(\mathbf{P})$ if at least k voters have x at the top of their ballots. Then V_k is anonymous and neutral, but V_k is not normal. □

To Do 1 *Find out the minimal properties of voting rules that guarantee normality. The previous fact indicates that anonymity and neutrality are not enough.*

6 Scoring

A score for an m-profile is a list of m integers, one for each alternative. The score lists the number of points that each alternative gets. The convention is that the alternative with the highest score wins.

```
type Score = [Int]
```

A profile is tallied with a voting rule by calculating the score of each alternative.

The following function finds the alternatives with the maximum score:

```
findMaxValues :: Score -> [(Int,Int)]
findMaxValues values =
   filter (\ (_,x) -> x == maximum values)
          (zip [0..] values)
```

The alternatives with the maximum values are the winners, so `findMaxValues` can be used to map a score to a set of winners:

```
winners :: Score -> [Alternative]
winners = map fst . findMaxValues
```

Find the alternative(s) with the minimum score:

```
findMinValues :: Score -> [(Int,Int)]
findMinValues values =
   filter (\ (_,x) -> x == minimum values)
          (zip [0..] values)
```

Compute the loser(s):

```
losers :: Score -> [Alternative]
losers = map fst . findMinValues
```

A type for scoring functions:

```
type SF = Profile -> Score
```

From a scoring function to a voting rule:

```
sf2votingrule :: SF -> VotingRule
sf2votingrule sf = winners . sf
```

7 Voting Rules With Positional Scoring

A scoring vector for ballots of size m is a list of non-negative integers $(w_0, \ldots, w_{m-1})$ satisfying $w_i \geq w_{i+1}$. The number w_i indicates the weight of position i in the ballot.

Plurality or majority has scoring vector $(1, 0, \cdots, 0)$. Anti-plurality or veto has scoring vector $(1, \cdots, 1, 0)$. Borda has scoring vector $(m-1, m-2, \cdots, 1, 0)$ [Bor81].

We implement functions that construct scoring vectors from numbers of alternatives. So we define the types of scoring vectors and scoring vector functions as follows:

```
type ScoringVector = [Int]
type ScoringVF = Int -> ScoringVector
```

Example scoring vectors (in fact, vector constructing functions):

```
pluralityVector :: ScoringVF
pluralityVector n = 1 : (take (n-1) $ repeat 0)

antipluralityVector ::  ScoringVF
antipluralityVector n = (take (n-1) $ repeat 1) ++ [0]

bordaVector :: ScoringVF
bordaVector n = reverse [0..n-1]
```

For every scoring vector there is an equivalent scoring vector with $w_{m-1} = 0$, so scoring vectors can be normalized by subtracting a positive c from every w_i. Furthermore, common factors can be divided out. This gives:

```
normSV :: ScoringVector -> ScoringVector
normSV ws = let
     w = last ws
     xs = zipWith (-) ws (repeat w)
     ys = init xs
     k  = if null ys then 1 else foldl1 gcd ys
     f  = flip div k
  in
     map f xs
```

This gives:

```
*Voting> normSV [5,3,1]
[2,1,0]
*Voting> normSV [8,5,5,2]
[2,1,1,0]
```

Proposition 5 *Scoring vector normalization does not affect the set of winners.*

Proof. Let $(w_1, \ldots, w_{m-1})$ be a scoring vector. If x is a winner under this vector for profile $\mathbf{P}$, this means that the score N of x for $\mathbf{P}$ is maximal among the scores, i.e., greater than or equal to the score M of any alternative $y \neq x$. Scoring for the vector $(w_1 - w_{m-1}, \ldots, w_{m-2} - w_{m-1}, 0)$ give scores $N - kmw_{m-1}$ and $M - kmw_{m-1}$, so the score of x is still maximal. In the other direction, the scores change by adding a constant, so winners are also preserved.

Next, compare $(w_1, \ldots, w_{m-1})$ and $(w_1K, \ldots, w_{m-1}K)$, with $K > 1$. Scores M and N for x and y under $(w_1, \ldots, w_{m-1})$ change into MK and NK. Since $M > N$ iff $MK > NK$, winners are not affected in either direction. □

From a scoring vector (function) to a scoring function:

```
vector2sf :: ScoringVF -> SF
vector2sf vectorfct profile =
  let
    m          = size profile
    vector     = vectorfct m
    eprofile   = expand profile
    vmult k b  = [ (x, k*(vector!!n))
                          | (x,n) <- zip b [0..] ]
    poscounts  = concat $
                 map (\(ys,k) -> vmult k ys) eprofile
    f x ys     = map snd (filter (\ (y,_) -> y == x) ys)
  in
   [ sum (f x poscounts) | x <- [0..m-1] ]
```

From a scoring vector function to a voting rule:

```
svf2votingrule :: ScoringVF -> VotingRule
svf2votingrule = sf2votingrule . vector2sf
```

The trivial voting rule that maps any profile to A arises from the trivial scoring vector $(0, \ldots, 0)$.

```
bordaSC, plurSC, vetoSC :: SF
bordaSC = vector2sf bordaVector
plurSC  = vector2sf pluralityVector
vetoSC  = vector2sf antipluralityVector
```

```
borda, plur, veto :: VotingRule
borda = svf2votingrule bordaVector
plur = svf2votingrule pluralityVector
veto = svf2votingrule antipluralityVector
```

And with tie breaking:

```
bordaR, plurR, vetoR :: ResVotingRule
bordaR = tieBreak [0..] borda
plurR =  tieBreak [0..] plur
vetoR =  tieBreak [0..] veto
```

Versions that display the outcomes as character lists:

```
borda', plur', veto' :: VotingRule'
borda' = map int2chr . borda
plur'  = map int2chr . plur
veto'  = map int2chr . veto

bordaR', plurR', vetoR' :: ResVotingRule'
bordaR' = int2chr . bordaR
plurR'  = int2chr . plurR
vetoR'  = int2chr . vetoR
```

8 The Majority, Unanimity, Near-Unanimity Voting Rules

Majority is the voting rule that selects an alternative with more than 50 % of the votes as winner, and returns the whole set of alternatives otherwise. This is not the same as plurality, which selects an alternative that has the maximum number of votes as winner, regardless of whether more than half of the voters voted like this or not.

```
majority :: VotingRule
majority profile = let
   m       = size profile
   score   = vector2sf pluralityVector profile
   results = findMaxValues score
   total   = sum profile
   (winner,votes) = head results
 in
   if (fromIntegral votes) / (fromIntegral total) > 0.5
     then [winner]
     else [0..(m-1)]
```

Unanimity: if all voters have an alternative a at the top of their ballots then a is the winner, otherwise all alternatives tie for a win.

```
unanimity :: VotingRule
unanimity profile = let
   m       = size profile
   xs      = vector2sf pluralityVector profile
   results = findMaxValues xs
   total   = sum profile
   (winner,votes) = head results
 in
   if votes == total && total > 0
     then [winner]
     else [0..(m-1)]
```

Near-unanimity: if all but at most one of the voters have an alternative a at the top of their ballots then a is the winner, otherwise all alternatives tie for a win.

```
nearUnanimity :: VotingRule
nearUnanimity profile = let
   m       = size profile
   xs      = vector2sf pluralityVector profile
   results = findMaxValues xs
   total   = sum profile
   (winner,votes) = head results
 in
   if votes+1 >= total && total > 0
     then [winner]
     else [0..(m-1)]
```

```
majority', unanimity', nearUnanimity' :: VotingRule'
majority'      = map int2chr . majority
unanimity'     = map int2chr . unanimity
nearUnanimity' = map int2chr . nearUnanimity
```

9 Profile Restriction

Profile restriction is computing a new profile for a subset of the alternative set of the original profile. The relative preferences of the voters in the new profile should remain unchanged.

```
restrict :: [Alternative] -> Profile -> Profile
restrict xs profile = let
    m = length profile
    eprofile = expand profile
    f = filter (flip elem xs)
    table = zip xs [0..]
    g = map (\ x -> let Just y = lookup x table in y)
    pprofile = map (\ (xs,k) -> (g (f xs), k)) eprofile
  in
    makeProfile pprofile
```

```
example0 = [1,2,0,3,0,2] :: Profile
```

This gives:

```
*Voting> restrict [0,1] example0
[4,4]
*Voting> restrict [0,2] example0
[6,2]
*Voting> restrict [1,2] example0
[5,3]
```

What this means is that out of 8 voters, 4 prefer 0 to 1, 6 prefer 0 to 2, and 5 prefer 1 to 2.

If $B \subseteq A$, we use $\mathbf{P}^B$ for the result of restricting $\mathbf{P}$ to B.

Definition 6 *A voting rule V is safe for restriction if it holds for every $B \subseteq A$ and every profile $\mathbf{P}$ that*

$$V(\mathbf{P}) \cap B \subseteq V(\mathbf{P}^B).$$

What this means is that winners are preserved under restriction.

Here is an obvious question;

Question 7 *Characterize the voting rules that are safe for restriction.*

Notice that restriction destroys information. If there are m alternatives and k voters then there are $m!$ possible ballots. The number of integer solutions for

$$x_1 + \cdots + x_n = k$$

under the condition that $x_i \geq 0$ for all $i = 1, \ldots, n$ is $\binom{n+k-1}{k}$ [Juk11, Proposition 1.5].

Thus, for m alternatives and k voters there are $\binom{m!+k-1}{k}$ possible profiles. There are m ways to restrict away one alternative. After pruning, there are $(m-1)!$ possible ballots, which leaves $\binom{(m-1)!+k-1}{k}$ profiles. All in all this gives $m\binom{(m-1)!+k-1}{k}$ possibilities.

Let's calculate these outcomes for some m and k.

```
fact :: Int -> Int -> Int
fact n k = product [n-k+1..n]

binom :: Int -> Int -> Int
binom n k = div (fact n k) (fac k)
```

For 4 alternatives and 10 voters, we get:

```
*Voting> binom ((fac 4) + 10 - 1) 10
92561040
*Voting> 4 * binom ((fac 3) + 10 - 1) 10
12012
```

To see that the information destruction is vast, consider the case where the pruning process leaves only pairs. m alternatives give $m(m-1)$ pairs, so after pair pruning there are only $m(m-1)(k+1)$ possibilities left, since there are $k+1$ ways to split k into non-negative integers k_1, k_2 with $k_1 + k_2 = k$. For 4 alternatives and 10 voters, this reduces the number of possibilities from 92561040 to 132.

An alternative x beats another alternative y in a one-to-one contest if more than half of the voters prefer x to y. Such a one-to-one contest between alternatives can be defined in terms of pruning by means of:

```
beats :: Profile -> Alternative -> Alternative -> Bool
beats p x y = let
   p' = restrict [x,y] p
 in majority p' == [0]
```

10 Pairwise Contests and the Condorcet Winner

A *majority graph* [Las97] is a directed graph on the set of alternatives as nodes, with an edge $x \to y$ indicating that x beats y in a majority contest.

```
type MajorityGraph = Alternative -> Alternative -> Int
```

Every profile gives rise to a majority graph, as follows:

```
p2mg :: Profile -> MajorityGraph
p2mg p x y = if beats p x y then 1 else 0
```

The pair score gives the number of pairwise contests that each alternative wins.

```
pairscore :: Profile -> [Int]
pairscore profile = let
    m  = size profile
    as = [0..m-1]
    mg = p2mg profile
  in
    [ sum [ mg x y | y <- as \\ [x] ] | x <- as ]
```

A Condorcet winner is an alternative that beats every other alternative in pairwise contests.

```
cW :: Profile -> Alternative -> Bool
cW profile x = let
    m  = size profile
  in
    pairscore profile !! x == m-1
```

The Condorcet voting rule (proposed in 1785 by the marquis of Condorcet in [Con85]) selects the Condorcet winner if it exists, and the set of all alternatives otherwise.

```
condorcet :: VotingRule
condorcet profile = let
    m  = size profile
    as = [0..m-1]
    f [] = as
    f (x:xs) = if cW profile x then [x] else f xs
  in
    f as
```

```
condorcet' :: VotingRule'
condorcet' = map int2chr . condorcet
```

11 The Copeland Rule

The Copeland voting rule [Cop51] selects the alternative that maximizes the difference between the number of won and lost pairwise majority contests. To compute it we need an expanded version of `pairscore`:

```
pairscore' :: Profile -> [(Int,Int)]
pairscore' profile = let
    m  = size profile
    as = [0..m-1]
    mg = p2mg profile
  in
    [ (sum [ mg x y | y <- as \\ [x] ],
       sum [ mg y x | y <- as \\ [x] ])  | x <- as ]
```

The Copeland score function is the difference between these two scores:

```
copelandScore :: Profile -> Score
copelandScore profile =
   map (\(x,y) -> x-y) (pairscore' profile)
```

The Copeland rule:

```
copeland :: VotingRule
copeland = winners . copelandScore
```

```
copeland' :: VotingRule'
copeland' = map int2chr . copeland
```

12 Single Transferable Vote, or the Hare Rule

The voting rule of single transferable vote, also known as the Hare rule (described by John Stuart Mill, with an attribution to Thomas Hare, in [Mil61]), works as follows.

If one of the candidates gets an absolute majority, that candidate wins. Otherwise prune the candidate(s) who is/are ranked first by the fewest number of voters from the profile, and repeat.

This is called single transferable vote because one way to think about it is that each voter has one vote aimed at getting a majority for some alternative. If this fails, the vote is not wasted, but transfered to the next candidate on the ballot list that is still in the running. And so on.

In the implementation of the `hare` rule, the repeat construct (the "and so on") is represented by a recursive call to `hare`. Note that it may happen that at some stage in the process, there is a tie between all candidates: they all get the same number of preference votes. In that case this set is returned as the set of Hare winners.

```
hare :: VotingRule
hare profile = let
    m  = size profile
    ps = plurSC profile
    maxs = findMaxValues ps
    mins = findMinValues ps
    (winner,votes) = head maxs
    total  = sum profile
    proportion = (fromIntegral votes) / (fromIntegral total)
    as = [0..m-1]
    ws = as \\ (losers ps)
    profile' = restrict ws profile
    g = \ n -> ws !! n
  in
    if proportion > 0.5 then [winner]
    else if ws == [] then as
    else map g (hare profile')
```

```
hare' :: Profile -> String
hare' = map int2chr . hare
```

To Do 2 *Analyze how much of the ballot information is used by this rule. Compare with the Borda rule.*

To Do 3 *Implement the version where a number of candidates needs to be elected, and where the votes for eliminated candidates get transferred.*

13 Plurality with Run-Off

Use a first round to select the (two) top candidates by plurality voting. Next, have a second round using the plurality rule with the (two) top candidates.

What to do if in the first round there is a tie between more than two candidates? In the implemenentation, the second round is between all of these. But ties in the second round are broken.

```
plurRO :: ResVotingRule
plurRO profile = let
    m        = size profile
    score    = plurSC profile
    max1     = findMaxValues score
    f        = \ (n,k) ->  if elem (n,k) max1 then 0 else k
    max2     = findMaxValues (map f (zip [0..] score))
    max3     = if length max1 > 1
                 then max1 else max1 ++ max2
    ws       = map fst max3
    profile' = restrict ws profile
    g        = \ n -> ws !! n
  in
    if m == 2 then plurR profile
    else g (plurR profile')
```

```
plurRO' :: Profile -> Char
plurRO' = int2chr . plurRO
```

As the implementation shows, plurality with runoff is very similar to the Hare rule. In the case of an election with three candidates, they boil down to (almost) the same thing.

14 Profile Addition, Subtraction and Multiplication

Intuitively, we can merge two elections into a single election, by adding the numbers of votes for the various ballots.

```
addP :: Profile -> Profile -> Profile
addP = zipWith (+)
```

Call this operation $\oplus$. Note that the two operand profiles have to be of the same size.

Definition 8 *A voting rule V is* additive *if it holds for all m-profiles* $\mathbf{P}$ *and* $\mathbf{Q}$ *that* $V(\mathbf{P}) \cap V(\mathbf{Q}) \subseteq V(\mathbf{P} \oplus \mathbf{Q})$. *Or in words: V is additive if winners of two separate elections concerning the same set of alternatives remain winners if the elections are merged.*

The following definition is from [You75].

Definition 9 *A voting rule V is* consistent *if it holds for all m-profiles* $\mathbf{P}$ *and* $\mathbf{Q}$ *that* $V(\mathbf{P}) \cap V(\mathbf{Q}) \neq \emptyset$ *implies* $V(\mathbf{P}) \cap V(\mathbf{Q}) = V(\mathbf{P} \oplus \mathbf{Q})$.

The property of additivity is weaker than the property of consistency: see Fact 14 below.

The requirement of additivity seems entirely reasonable, and the following fact is perhaps surprising.

Proposition 10 *The Condorcet rule is not additive.*

Proof. Consider the following two profiles $\mathbf{P}$ and $\mathbf{Q}$:

$$(abc, 3), (bca, 0), (cab, 0), (acb, 0), (cba, 0), (bac, 2),$$

$$(abc, 1), (bca, 1), (cab, 1), (acb, 3), (cba, 3), (bac, 3).$$

The first of these has Condorcet winner a, the second has no Condorcet winner. So $V(\mathbf{P}) = \{a\}$ and $V(\mathbf{Q}) = \{a, b, c\}$, and therefore $V(\mathbf{P}) \cap V(\mathbf{Q}) = \{a\}$. Their sum is:

$$(abc, 4), (bca, 1), (cab, 1), (acb, 3), (cba, 3), (bac, 5).$$

The Condorcet winner of this sum is b. □

Here is the demonstration with the implementation:

```
*Voting> addP [3,0,0,0,0,2] [1,1,1,3,3,3]
[4,1,1,3,3,5]
*Voting> condorcet' [3,0,0,0,0,2]
"a"
*Voting> condorcet' [1,1,1,3,3,3]
"abc"
*Voting> condorcet' [4,1,1,3,3,5]
"b"
```

A voting rule satisfies the *Condorcet Criterion* if it always elects the Condorcet winner if there is one. The above fact should worry anyone who thinks of the Condorcet criterion as a benchmark for voting rule quality.

To prove the following fact, we need two more example profiles:

```
profile1, profile2 :: Profile
profile1 = makeProfile'
  [("abcd",5),("bacd",6),("cabd",2),("dabc",10)]
profile2 = makeProfile'
  [("abcd",4),("bacd",4),("cabd",8),("dabc",2)]
```

Proposition 11 *The Hare rule is not additive.*

Proof.

```
*Voting> hare' profile1
"a"
*Voting> hare' profile2
```

```
"a"
*Voting> hare' (addP profile1 profile2)
"b"
```

□

Proposition 12 *The majority, unanimity and near-unanymity rules are additive.*

Proof. Suppose $\mathbf{P}$ and $\mathbf{Q}$ are m-profiles, V is the majority rule, and $a \in V(\mathbf{P}) \cap V(\mathbf{Q})$. Let $\mathbf{P}$ have N voters and $\mathbf{Q}$ have M voters. Then either no $x \in A$ has an absolute majority, or more than $N/2$ ballots in $\mathbf{P}$ have a in first position. Similarly, either no $x \in A$ has an absolute majority in $\mathbf{Q}$, or more than $M/2$ ballots in $\mathbf{Q}$ have a in first position. It follows that either no $x \in A$ has an absolute majority in $\mathbf{P} \oplus \mathbf{Q}$, in which case $a \in V(\mathbf{P} \oplus \mathbf{Q}) = A$, or $(N+M)/2$ ballots in $\mathbf{P} \oplus \mathbf{Q}$ have a in first position, i.e., a is the majority winner in $\mathbf{P} \oplus \mathbf{Q}$.

Same reasoning for the unanimity and near-unanymity rule. □

Proposition 13 *The near-unanymity rule is not consistent.*

Proof. Let V be the near-unanimity rule and let $\mathbf{P}$ be the following profile:

$$(ab, 2), (ba, 1).$$

Then $V(\mathbf{P}) = \{a\}$ and $V(\mathbf{P} \oplus \mathbf{P}) = \{a, b\}$. This shows that V is not consistent. □

Proposition 14 *Additivity does not imply consistency.*

Proof. Immediate from Facts 12 and 13. □

Theorem 15 *Every positional voting rule is additive.*

Proof. Let V be a positional voting rule, and let $\mathbf{P}$, $\mathbf{Q}$ be a pair of m-profiles, for some m. Suppose $a \in V(\mathbf{P}) \cap V(\mathbf{Q})$. We have to show that $a \in V(\mathbf{P} \oplus \mathbf{Q})$. But this is immediate from the fact that if the score of a is maximal in $\mathbf{P}$ and $\mathbf{Q}$, it is also maximal in $\mathbf{P} \oplus \mathbf{Q}$. □

Question 16 *Can we prove an if and only if for additivity?*

Profile subtraction subtracts the result of one election from that of another election (by separating the sets of voters):

```
subtrP :: Profile -> Profile -> Profile
subtrP p1 p2 = let
    p = zipWith (-) p1 p2
  in
    if any (<0) p then error "negative number of voters"
                  else p
```

Call this operation $\ominus$. Note that the two operand profiles have to be of the same size.

Finally, we can multiply a profile by a positive integer.

```
multP :: Int -> Profile -> Profile
multP k profile = if k < 1
                  then error "wrong multiplication factor"
                  else map (k*) profile
```

Call this the k-fold product of $\mathbf{P}$, with notation $k\mathbf{P}$.

15 Profile Reduction

The following definition is from Saari [Saa95].

Definition 17 *A profile is called* reduced *if each cycle in the profile contains a ballot with no voters.*

The profile

$$(abc, 3), (bca, 1), (cab, 0), (acb, 2), (cba, 0), (bac, 2)$$

is reduced.

Definition 18 *A profile is called* balanced *if each cycle in the profile is such that each ballot in the cycle has the same number of voters. Use* $\mathbf{B}$ *for balanced profiles.*

The profile

$$(abc, 1), (bca, 1), (cab, 1), (acb, 3), (cba, 3), (bac, 3)$$

is balanced.

Proposition 19 *For every profile* $\mathbf{P}$ *there exist a reduced* $\mathbf{Q}$ *and a balanced* $\mathbf{B}$ *such that* $\mathbf{P} = \mathbf{Q} \oplus \mathbf{B}$.

Definition 20 *If* $\mathbf{P} = \mathbf{Q} \oplus \mathbf{B}$, *as in Fact 19, then call* $\mathbf{B}$ *the* surplus *of* $\mathbf{P}$ *and* $\mathbf{Q}$ *the reduced form of* $\mathbf{P}$. *Use* $\mathbf{P}^r$ *for the reduced form of* $\mathbf{P}$.

Proposition 21 *A profile* $\mathbf{P}$ *is both balanced and reduced iff* $\mathbf{P}$ *has no voters.*

Definition 22 *Call the operation of subtracting a balanced profile from* $\mathbf{P}$ reduction. *Call the operation of adding a balanced profile to* $\mathbf{P}$ dilution.

> Here is an obvious **algorithm** for putting a profile $\mathbf{P}$ in reduced form:
>
> For each cycle π of $\mathbf{P}$, let the minimum of the vote numbers in that cycle be k. Subtract k from every vote number in the cycle.

The surplus of a profile indicates by how much the profile can be reduced:

```
surplus :: Profile -> Profile
surplus profile = let
   m = size profile
   eprofile = expand profile
   n = fac (m - 1)
   cls = [ getCycle m k | k <- [0..n-1] ]
   vals = [ [ nr | (xs,nr) <- eprofile, ys <- c, xs == ys ]
              | c <- cls ]
   mins = [ minimum list | list <- vals ]
 in
   [ k | xs <- genBallots m,
         (l,k) <- zip [0..] mins,
         elem xs (getCycle m l) ]
```

The surplus of the profile

$$(abc, 4), (bca, 2), (cab, 1), (acb, 3), (cba, 3), (bac, 6)$$

is the profile

$$(abc, 1), (bca, 1), (cab, 1), (acb, 3), (cba, 3), (bac, 3).$$

Use this for putting a profile in reduced form:

```
reduce :: Profile -> Profile
reduce profile = subtrP profile (surplus profile)
```

The reduced form of the profile

$$(abc, 4), (bca, 2), (cab, 1), (acb, 3), (cba, 3), (bac, 6)$$

is the profile

$$(abc, 3), (bca, 1), (cab, 0), (acb, 0), (cba, 0), (bac, 3).$$

Theorem 23 *Any anonymous and neutral voting rule maps a balanced profile to the set of all alternatives.*

Proof. Let $\mathbf{P}$ be a balanced profile for A. Let V be an anonymous and neutral voting rule. We must prove that $V(\mathbf{P}) = A$.

Suppose not, i.e., suppose there is some $b \notin V(\mathbf{P})$. There also is some $a \in V(\mathbf{P})$, for $V(\mathbf{P}) \neq \emptyset$.

Let σ be any permutation of A that satisfies $\sigma(a) = b$.

Observe that each cycle will remain a cycle under the permutation σ. Therefore, because of anonymity and the fact that $\mathbf{P}$ is balanced: $\mathbf{P}^\sigma = \mathbf{P}$. Because of neutrality $V(\mathbf{P}^\sigma) = \sigma(V(\mathbf{P}))$, and therefore $b = \sigma(a) \in V(\mathbf{P}^\sigma) = V(\mathbf{P})$, and contradiction. □

Theorem 24 *If $|A| = m$ then the number of voters in any balanced profile for A is a multiple of m.*

Proof. Each cycle in an m-profile has m elements. There are $(m-1)!$ cycles. Let cycle i have k_i voters. Then all in all we have $m \sum_{i=1}^{(m-1)!} k_i$ voters. □

Definition 25 *A voting rule V* is safe for dilution *if it holds for all profiles $\mathbf{P}$ and balanced profiles $\mathbf{B}$ that $V(\mathbf{P}) \supseteq V(\mathbf{P} \oplus \mathbf{B})$.*

Safety for dilution means that dilution does not introduce new winners.

Definition 26 *A voting rule V* is safe for reduction *if it holds for all profiles $\mathbf{P}$ and balanced profiles $\mathbf{B}$ that $V(\mathbf{P}) \subseteq V(\mathbf{P} \oplus \mathbf{B})$.*

Safety for reduction means that reduction does not introduce new winners.

Theorem 27 *Any anonymous, neutral and additive voting rule is safe for reduction.*

Proof. Assume V is anonymous and neutral. Then $V(\mathbf{B})$ equals the set of all alternatives. By additivity we have:

$$V(\mathbf{P}) = V(\mathbf{P}) \cap V(\mathbf{B}) \subseteq V(\mathbf{P} \oplus \mathbf{B}).$$

□

Proposition 28 *The Condorcet rule is neither safe for reduction nor safe for dilution.*

Proof. Consider the profile:

$$(abc, 1), (bac, 3), (bca, 1), (acb, 5), (cab, 4), (cba, 3).$$

The Condorcet winner for this profile is a. The reduced form of this is:

$$(abc, 0), (bac, 0), (bca, 0), (acb, 2), (cab, 3), (cba, 0).$$

The Condorcet winner for the reduced profile is c. □

Proposition 29 *The majority rule is safe for reduction, but not safe for dilution.*

Proof. The example from Fact 28 works here as well. In the reduced profile

$$(abc, 0), (bac, 0), (bca, 0), (acb, 2), (cab, 3), (cba, 0)$$

there is a majority for c. Dilute this profile with

$$(abc, 1), (bac, 1), (bca, 1), (acb, 3), (cab, 3), (cba, 3).$$

There is no majority in the diluted profile

$$(abc, 1), (bac, 3), (bca, 1), (acb, 5), (cab, 4), (cba, 3).$$

□

Theorem 30 *Any voting rule V with positional scoring will assign to every alternative in a balanced profile $\mathbf{B}$ the same score.*

Proof. Let $\mathbf{B}$ be a balanced m-profile. Then there are $(m-1)!$ cycles, and there are k_i voters in each ballot in the i-th cycle. Let V be a positional voting rule with $(x_0, \cdots, x_{m-1})$ as its scoring vector. Let π_i be an arbitrary cycle of $\mathbf{P}$, let a be an arbitrary alternative, and let j be an arbitrary position (i.e., $0 \leq j < m$). Then the score for a for this position in the cycle under the voting rule is given by $k_i x_j$, for a occurs in this position exactly once in the cycle. Summing over the cycles, we get that a collects the following score in $\mathbf{B}$:

$$\sum_{i=1}^{(m-1)!} k_i x_j.$$

Summing over the positions, we see that a collects the score:

$$\sum_{j=0}^{m-1} \sum_{i=1}^{(m-1)!} k_i x_j.$$

Since a was arbitrary, every alternative collects this same score. □

Theorem 31 *Any voting rule V with positional scoring is safe for reduction and safe for dilution.*

Proof. Let $\mathbf{P}$ be an m-profile, and let $\mathbf{B}$ be a balanced m-profile.

Since $\mathbf{B}$ is balanced, it follows from the previous Theorem that the scores for the alternatives under V for $\mathbf{P}$ can be computed from those for $\mathbf{P} \oplus \mathbf{B}$ by subtracting a constant c from each score, and vice versa, by adding a constant c to each score. These subtractions and additions do not affect the outcome of V. □

Question 32 *Does the converse hold as well? If a voting rule is safe in both directions then it is positional?*

For all voting rules V that are not invariant under reduction, the derived voting rule V^r defined by $V^r(\mathbf{P}) = V(\mathbf{P}^r)$ is different from V.

Theorem 33 *There are voting rules V with the property that V is not positional, but V^r is.*

Proof. Let V be the voting rule stating that x is a single winner if x is at the top position in a majority of the ballots, and moreover these are more than $\frac{1}{m!-m}$ of all ballots. Otherwise all alternatives tie for a win. Observe that V is not positional. But V^r is the plurality rule, for the condition that a single plurality winner has more than $\frac{1}{m!-m}$ of the votes is always fulfilled in a reduced profile. □

In view of Theorem 33 it makes sense to ask the following question:

Question 34 *Can we characterize the voting rules V with the property that V^r is positional?*

Here is the implementation of the function that maps V to V^r:

```
red :: VotingRule -> VotingRule
red f p = f (reduce p)
```

Definition 35 *If $v \subseteq A$, then a v-cycle on A is a permutation on A that is a full cycle on v and the identity on $A - v$. For $v \subseteq A$ with $|v| \geq 2$, a profile is v-balanced if every v-cycle in it has the same number of voters for each ballot in the cycle.*

For example, the profile

$$(abc, 2), (bac, 2), (acb, 1), (bca, 1)$$

is $\{a, b\}$-balanced.

Theorem 36 *Any anonymous and neutral voting rule maps a v-balanced profile for A either to a superset of v or to a subset of $A - v$.*

Proof. Let $\mathbf{P}$ be v-balanced, and let V be an anonymous and neutral voting rule. Assume $v \not\subseteq V(\mathbf{P})$ and $V(\mathbf{P}) \cap v \neq \emptyset$. Let $a \in V(\mathbf{P}) \cap v$ and $b \in v - V(\mathbf{P})$. We derive a contradiction.

Since both $a \in v$ and $b \in v$, and π is a v-cycle, there is some k with $\pi^k(a) = b$. Apply $\mathbf{P} = \mathbf{P}^\pi$ k times to get $\mathbf{P} = \mathbf{P}^{\pi^k}$. Therefore $V(\mathbf{P}) = V(\mathbf{P}^{\pi^k})$. By the fact that V is neutral, we get from this that $V(\mathbf{P}) = \pi^k(V(\mathbf{P}))$. From this it follows that $b \in V(\mathbf{P})$, and contradiction. □

To Do 4 *Analyze and implement v-cycles.*

16 Casting Ballots

Casting k identical ballots can be viewed as a function from profiles to profiles, as follows:

```
cast :: Int -> Ballot -> Profile -> Profile
cast k ballot profile = let
    m = size profile
    p = position ballot (genBallots m)
    f = \ (x,n) -> if x == p then n+k else n
  in
    map f (zip [0..] profile)
```

Finding the position of an item in a list of items, where it is assumed that the item occurs in the list:

```
position :: Eq a => a -> [a] -> Int
position x xs = let
    Just p = lookup x (zip xs [0..])
  in p
```

Casting a single ballot is a function from profiles to profiles:

```
cast1 :: Ballot -> Profile -> Profile
cast1 = cast 1
```

Constructing a profile from a list of (ballot,int) pairs:

```
makeProfile :: [(Ballot,Int)] -> Profile
makeProfile [] = error "no ballots"
makeProfile [(x,k)] = let
    m = length x
  in
    cast k x (nullprofile m)
makeProfile ((x,k):xs) = cast k x (makeProfile xs)
```

Constructing a profile from a list of ballots:

```
makeProfile1 :: [Ballot] -> Profile
makeProfile1 xs = makeProfile (zip xs (repeat 1))
```

Versions with strings for the ballots:

```
makeProfile' :: [(String,Int)] -> Profile
makeProfile' = makeProfile . map f where
  f (xs,k) = (map chr2int xs,k)

makeProfile1' :: [String] -> Profile
makeProfile1' = makeProfile1 . map (map chr2int)
```

```
chr2int :: Char -> Int
chr2int c = fromEnum c - 97
```

Example (stolen from [End10]): the simplified and reconstructed Florida 2000 US presidential elections profile. Use a for Bush, b for Gore, c for Nader.

```
florida :: Profile
florida = makeProfile'
   [("abc",49),("bca",20),("bac",20),("cba",11)]
```

This gives:

```
*Voting> plur' florida
"a"
```

```
*Voting> majority' florida
"abc"
*Voting> condorcet' florida
"b"
*Voting> borda' florida
"b"
*Voting> hare' florida
"b"
*Voting> plurRO' florida
'b'
```

17 Ballot Withdrawal

If k voters with the same ballot decide to withdraw their vote from a profile, then the result is given by:

```
withdraw :: Int -> Ballot -> Profile -> Profile
withdraw k ballot profile = let
    m = size profile
    p = position ballot (genBallots m)
    f = \ (x,n) -> if x == p && n < k then
                       error "negative ballot number"
                   else if x == p then n-k
                   else n
  in
    map f (zip [0..] profile)
```

Withdrawing a single ballot:

```
withdraw1 :: Ballot -> Profile -> Profile
withdraw1 = withdraw 1
```

The following example is again from [End10]:

```
noshowExample :: Profile
noshowExample = makeProfile'
   [("abc",25),("cab",46), ("bca",24)]

noshow :: Profile
noshow = withdraw 2 [0,1,2] noshowExample
```

This illustrates what some voting theorists call the ‘no show paradox’: the fact that it can be more advantageous to abstain from voting than to cast one’s true ballot.

```
*Voting> plurRO' noshowExample
'c'
*Voting> plurRO' noshow
'b'
```

Proposition 37 *Plurality with run-off is not safe for reduction.*

Proof.

```
*Voting> plurRO' noshow
'b'
*Voting> plurRO' (reduce noshow)
'c'
```

□

18 Changing Ballots

If k voters change from ballot $\mathbf{b}$ to ballot $\mathbf{b}'$, this can be described as a withdrawal step followed by a new casting step:

```
change :: Int -> Ballot -> Ballot -> Profile -> Profile
change k b b' = cast k b' . withdraw k b
```

Ballot change by a single voter:

```
change1 :: Ballot -> Ballot -> Profile -> Profile
change1 = change 1
```

```
change1' :: String -> String -> Profile -> Profile
change1' b b' = change 1 (map chr2int b) (map chr2int b')
```

19 Strategizing

Strategizing is replacing a ballot **b** by a different one, $\mathbf{b}'$, in the hope or expectation to get a better outcome (where better is "closer to **b**" in some sense).

As is explained in [Tay05], there are many ways to interpret 'better'. One way is that X is better than Y if X weakly dominates Y, that is if every $x \in X$ is at least as good as every $y \in Y$ and some $x \in X$ is better than some $y \in Y$. Here is its implementation:

```
better1 :: Ballot -> [Alternative] ->  [Alternative] -> Bool
better1 ballot outcome1 outcome2 = let
    order = list2ordering ballot
  in
    and [ order x y /= GT | x <- outcome1, y <- outcome2 ]
    &&
    or  [ order x y == LT | x <- outcome1, y <- outcome2 ]
```

The results of strategic change to a different ballot by a group of k voters. The voters are identified by their current ballot. The output of the function is the list of all alternative ballots that would give these voters a better outcome:

```
stratChange :: Int
               -> Ballot
               -> VotingRule
               -> Profile -> [Ballot]
stratChange k b rule profile = let
   m    = size profile
   alts = genBallots m \\ [b]
   x    = rule profile
   f y  = rule (change k b y profile)
 in
   [ alt | alt <- alts, better1 b (f alt) x ]
```

Strategic change by a single voter:

```
stratChange1 :: Ballot
                -> VotingRule
                -> Profile -> [Ballot]
stratChange1 = stratChange 1
```

The following example is from Taylor [Tay05, p. 45]:

```
example1 = makeProfile1' ["abcd", "bdca", "dcab","cabd"]
```

```
bordaManip = map (map int2chr) $
             stratChange1 (genBallots 4!!0) borda example1

plurManip = map (map int2chr) $
             stratChange1 (genBallots 4!!0) plur example1
```

`bordaManip` gives the two ballot changes that are advantageous for the voter with ballot *abcd*, given this profile, and given the fact that the Borda rule is applied.

```
*Voting> example1
[1,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,0,0,0,0,0,0,0,0]
Voting> borda' example1
"c"
*Voting> bordaManip
["bacd","badc"]
*Voting> borda' (change1' "abcd" "bacd" example1)
"bc"
*Voting> borda' (change1' "abcd" "badc" example1)
"b"
```

20 Voting Games

To define voting games it looks like we need to drop the assumption of anonymity, because we have to be able look at the voting situation from the perspective of individual voters (or agents). A bit of further reflection makes clear, however, that all that really matters for an individual agent is *how many* of the other agents hold certain ballots. After all, we do not drop the assumption that the voting rule that defines the game is anonymous.

What is relevant for determining the move of an individual player (abstain from the vote or not?, strategize or not?) is (i) a profile representing the current ballots (true or not) of the other voters, and (ii) the true ballot of the player, as a yardstick for the quality of an outcome of the vote.

Define a payoff function on the basis of the `better1` relation: the payoff of an outcome equals the number of possible outcomes that are worse than that outcome, given a particular ballot.

For generating possible outcomes we need:

```
powerlist :: [a] -> [[a]]
powerlist [] = [[]]
powerlist (x:xs) = powerlist xs ++ map (x:) (powerlist xs)
```

The payoff function is defined in terms of this, as follows:

```
payoff :: Int -> Ballot -> [Alternative] -> Int
payoff _ b [] = error "no winners selected"
payoff m b ws = let
    outcomes = powerlist [0..m-1] \\ [[]]
  in
    length [ vs | vs <- outcomes, better1 b ws vs ]
```

Using this we can compute the value for a voter with true ballot **b** of abstaining from the vote:

```
abstain :: VotingRule -> Profile -> Ballot -> Int
abstain r p b = let
    m = size p
    n = length b
  in
    if m /= n then error "wrong ballot size"
    else payoff m b (r p)
```

Now compare the following amusing quote from Littlewood:

> If a man abstains from voting in a General Election on the ground that the chance of his vote's mattering is negligible, it is common to rebuke him by saying 'suppose everyone acted so.' The unpleasant truth that the rebuke is fallacious in principle is perhaps fortunately hidden from the majority of the human race. Consider, however, the magnitudes involved, where the election and the constituency are reasonably open. The chance that his vote will elect his member by a majority of 1 is of the order of 1 in 5000; there is a further chance of the order of 1 in 50 that this result will cause a change of Government. The total chance of this is no worse than 1 in 250,000. Since there are 30,000,000 voters with similar opportunities it would appear that there is

> something wrong; the explanation is that when the event happens to one man, 20,000 or so[1] other voters in his constituency are in the same position.
>
> J.E. Littlewood, *A Mathematician's Miscellany* [Lit53]

The snag is, of course, that in order to make the decision that it is safe to abstain from voting, a voter has to know the outcome of the election, and has to assume that she is the only one abstaining from the vote.

The value for a group of voters with true ballot $\mathbf{b}$ of voting with ballot $\mathbf{b}'$, given a profile and a voting rule:

```
play :: Int -> VotingRule -> Profile
            -> Ballot -> Ballot -> Int
play k r p b b' = let
     m = size p
     n = length b
     n' = length b'
     p' = cast k b' p
   in
     if m /=n || n /= n' then error "wrong ballot size"
     else payoff m b (r p')
```

And for a single voter:

```
play1 :: VotingRule -> Profile
         -> Ballot -> Ballot -> Int
play1 = play 1
```

Making a game out of a list of ballots. First a type declaration for readability:

```
type Agent = Int
```

[1] Half 70 per cent, of 80,000.

Check whether a profile is an m-profiles for i voters:

```
checkProfile :: Int -> Int -> Profile -> Bool
checkProfile m i p = size p == m && voteSize p == i
```

Creating a game given a voting rule and a list of ballots. The game computes for each agent and each possible profile for the other players a payoff function. The ballot list that is the first argument gives the true ballots of all the players.

```
makeGame :: [Ballot] -> Agent -> VotingRule
            -> Profile -> Ballot -> Int
makeGame bs k r p castb = let
    trueb = bs !! k
    m = length trueb
    n = length castb
    i = length bs - 1
    ok = checkProfile m i p
  in
    if m /= n then error "wrong ballot size"
    else if not ok then error "wrong profile"
    else play1 r p trueb castb
```

Using strings for ballots:

```
makeGame' :: [String] -> Agent -> VotingRule
              -> Profile -> String -> Int
makeGame' bs k r p castb = let
    bs' = map (map chr2int) bs
    castb' = map chr2int castb
  in
    makeGame bs' k r p castb'
```

21 Approval Voting

Ballots for approval voting (see [Ott], [BF78] and [Bra08]) are subsets of the set of alternatives that the voter approves of. If there are m alternatives there are 2^m subsets.

Approval voting was designed to remedy the following defect of “one man, one vote”:

> The whole thing can be stated in a dispassionate way, without reference to “good” or to “sides”: if, out of three or more candidates, two are similar, and even if a majority of voters prefers either one of these, yet the votes of that majority are split between them, with the result that another candidate is likely to win, though not wanted by the majority.
>
> The primary reason why this seems wrong is that it makes the result of the vote depend more on the distribution of the candidates than on the distribution of the voters' wishes. Secondly, it is the opposite of the way it should be in that candidates ought to be encouraged, not discouraged, from adding their names to the competition; each new candidate may be an improvement on the others; at any rate the voters have a wider choice, and the statistical chance of electing a good candidate is higher.
>
> Thirdly, you have only to think of the dilemma you are placed in if you happen to be one of the voters supporting side B, especially BE [i.e., B “extreme”] and especially if it is a relatively small splinter. If there had been only two candidates, you would have voted for the one you considered better. To them is added another whom you consider better still, but he has less chance to win. If you do vote for him, you have in effect given your vote to the candidate you consider worst. On the other hand you feel that if you and others like you do not vote for the one you believe in, his cause will never have a chance to grow.
>
> All this is well known. It is “a fact of political life”; it is “the voters dilemma.”
>
> [Ott]

```
type Aballot = [Alternative]
```

A profile is again a mapping from ballots to non-negative integers.

```
genAballots :: Int -> [Aballot]
genAballots m = sublists [0..m-1]

sublists :: [a] -> [[a]]
sublists [] = [[]]
sublists (x:xs) = map (x:) (sublists xs) ++ sublists xs

complement :: Int -> [Alternative] -> [Alternative]
complement m ys = [0..m-1] \\ ys
```

```
genAballots' :: Int -> [String]
genAballots' = map (map int2chr) . genAballots
```

Profiles $\mathbf{P}$ for approval voting are reduced if $\mathbf{P}(S) > 0$ implies $\mathbf{P}(A - S) = 0$ for all $S \subseteq A$.

Profiles for approval voting are balanced if $\mathbf{P}(S) = k$ implies $\mathbf{P}(A - S) = k$ for all $S \subseteq A$.

Profiles for approval voting can be put in reduced form by subtraction of a surplus balanced profile.

```
asize :: Profile -> Int
asize profile = let
   m = fromIntegral (length profile)
 in
   round (logBase 2 m)
```

The expansion of a profile for approval voting:

```
aexpand :: Profile -> [(Aballot,Int)]
aexpand profile = let
    m = asize profile
    g = \ (n,k) -> (genAballots m !! n, k)
 in
    map g (zip [0..] profile)
```

```
avotes :: Profile -> Ballot -> Int
avotes profile ballot = let
    eprofile = aexpand profile
    Just k = lookup ballot eprofile
  in
    k
```

The surplus of a profile for approval voting:

```
asurplus :: Profile -> Profile
asurplus profile = let
   m        = asize profile
 in
   [ min (avotes profile ballot) (avotes profile cballot)
      | ballot  <- genAballots m,
        cballot <- [complement m ballot]  ]
```

Reducing a ballot for approval voting:

```
areduce :: Profile -> Profile
areduce profile = subtrP profile (asurplus profile)
```

Computing a score for approval voting from a ballot for approval voting:

```
approvalScore :: Profile -> Score
approvalScore profile = let
    m = asize profile
    eprofile = aexpand profile
    count = \ n -> sum [ k | (ballot,k) <- eprofile,
                             elem n ballot        ]
  in
    [ count n | n  <- [0..m-1] ]
```

Approval voting rule:

```
approval :: VotingRule
approval = winners . approvalScore
```

```
approval' :: VotingRule'
approval' = map int2chr . approval
```

22 Knowledge

Suppose the outcome of an election (or the outcome of a poll) gets announced. Then the following relation defines what the voters learn from this:

```
announce :: VotingRule
            -> Profile -> Profile -> Bool
announce rule p1 p2 = rule p1 == rule p2
```

23 Election Matrices

As was explained above, voting rules can be mapped to scoring functions. These scoring functions can be viewed as ways to determine a matrix for a set of linear equations, as follows.

```
type Matrix = [Row]
type Row    = [Ratio Int]

rows, cols :: Matrix -> Int
rows m = length m
cols m | m == []   = 0
       | otherwise = length (head m)
```

From a scoring function to an election matrix, given a profile size m:

```
sf2matrix :: Int -> SF -> Matrix
sf2matrix m f = let
    k = fac m
  in
    [ nrm $ f (unit m i) | i <- [0..k-1] ]
```

Examples:

```
example2 = sf2matrix 3 (vector2sf pluralityVector)

example3 = sf2matrix 3 (vector2sf antipluralityVector)

example4 = sf2matrix 3 bordaSC
```

This gives:

```
Voting> example2
[[1 % 1,0 % 1,0 % 1],[0 % 1,1 % 1,0 % 1],[0 % 1,0 % 1,1 % 1],
 [1 % 1,0 % 1,0 % 1],[0 % 1,0 % 1,1 % 1],[0 % 1,1 % 1,0 % 1]]
*Voting> example3
[[1 % 2,1 % 2,0 % 1],[0 % 1,1 % 2,1 % 2],[1 % 2,0 % 1,1 % 2],
 [1 % 2,0 % 1,1 % 2],[0 % 1,1 % 2,1 % 2],[1 % 2,1 % 2,0 % 1]]
*Voting> example4
[[2 % 3,1 % 3,0 % 1],[0 % 1,2 % 3,1 % 3],[1 % 3,0 % 1,2 % 3],
 [2 % 3,0 % 1,1 % 3],[0 % 1,1 % 3,2 % 3],[1 % 3,2 % 3,0 % 1]]
```

24 Further Work

This is work in progress. Our intention is to extend this implementation to an epistemic model checker for voting under uncertainty about the profile.

References

[BF78] S.J. Brams and P.C. Fishburn. Approval voting. *The American Political Science Review*, 72(3):831–847, 1978.

[Bor81] J.-C. de Borda. *Mémoire sur les élections au scrutin.* Histoire de l'Académie Royale des Sciences, Paris, 1781.

[Bra08] Steven Brams. *Mathematics and Democracy:Designing Better Voting and Fair Division Procedures.* Princeton University Press, 2008.

[Con85] M. le Marquis de Condorcet. *Essai sur l'application de l'analyse à la probabilité des décisions rendues à la pluralité des voix.* Imprimerie Royale, Paris, 1785.

[Cop51] A.H. Copeland. A "reasonable" social welfare function. Seminar on Mathematics in Social Sciences, 1951.

[End10] U. Endriss. Tutorial on voting theory. AAAI-2010 Slides, 2010.

[HT] The Haskell Team. The Haskell homepage. `http://www.haskell.org`.

[Jon03] S. Peyton Jones, editor. *Haskell 98 Language and Libraries; The Revised Report.* Cambridge University Press, 2003.

[Juk11] Stasys Jukna. *Extremal Combinatorics, with Applications in Computer Science — Second Edition.* Texts in Theoretical Computer Science. Springer, 2011.

[Knu92] D.E. Knuth. *Literate Programming.* CSLI Lecture Notes, no. 27. CSLI, Stanford, 1992.

[Las97] J.F. Laslier. *Tournament Solutions and Majority Voting.* Springer, 1997.

[Lit53] J.E. Littlewood. *A Mathematician's Miscellany.* Methuen, London, 1953.

[Mil61] John Stuart Mill. *Considerations of a Representative Government.* Parker, Son, and Bourn, London, 1861. Electronically available from Project Gutenberg.

[Ott] Guy Ottewell. The arithmetic of voting. Available online at `http://www.universalworkshop.com/ARVOfull.htm`.

[Saa95] D.G. Saari. *Basic Geometry of Voting.* Springer, 1995.

[Tay05] Alan D. Taylor. *Social Choice and the Mathematics of Manipulation.* Cambridge University Press, 2005.

[You75] H.P. Young. Social choice scoring functions. *SIAM Journal on Applied Mathematics*, 28(4):824–836, 1975.