Vote Manipulation under Partial Knowledge

Faculty of humanities
Department of Philosophy
Cognitive Artificial Intelligence

Thesis for the Degree of Bachelor of Science

7.5 ECTS

Author:
Rutger Hommes

Supervisor:
Prof. dr. Jan van Eijck

April 2012

Contents

Abstract

Vote manipulation is an important topic in social choice theory. Usually, voters are assumed to have perfect knowledge of the preferences of all the other voters involved in a vote, and they manipulate using this knowledge. In this thesis we consider a different manipulation scenario, in which voters do not have such perfect knowledge. We will write and demonstrate a computer program in which the concepts from social choice theory are implemented, including our own extension which deals with manipulation under partial knowledge.

1 Introduction

1.1 Social choice theory

Social choice theory is the scientific field that deals with the description and analysis of the way that the preferences of individual members of a group are aggregated into a decision of the group as a whole. The field is also called *voting theory*, a revealing name that suggests a connection with politics. Indeed, an important field of political science, called *democratic theory*, is based on the premise that the resolution of a matter of social policy, group choice, or collective action should be based on the preferences of the individuals in the society, group, or collective. How to make such a resolution based on the preferences of individuals, is an import issue in the design of democratic elections. Voting theory deals with issues such as this one, and it does so in a very precise, mathematical way, using logic as its formal instrument.

1.2 Vote manipulation

One difficulty in designing rules for the preference aggregation in a vote, is the fact that voters can sometimes achieve a preferred election result by casting a ballot that misrepresents their actual preferences. Voting theory has been concerned with finding properties of voting rules that make it impossible for voters to manipulate the vote. This led to the surprising result that, for an important class of voting rules, manipulability is avoidable only if the vote is a *dictatorship*.

1.3 Manipulation and knowledge

In the formal study of manipulation done by voting theory so far, manipulators have always been assumed to have *complete knowledge*, that is, all the knowledge they need to make perfect decisions in their manipulation. In reality, however, voters often have incomplete or *partial* knowledge. In this thesis, we will extend the field of voting theory with a new concept called *profile knowledge*, which will be our key element in reasoning about and implementing manipulation under partial knowledge.

1.4 Haskell

For our implementation of concepts from voting theory, we will use the programming language *Haskell*. This language has a number of distinctive properties that make it suitable for our purpose.

First of all, it is a *functional programming language*, which means that the statements of the language are *functions*. These functions can be seen as little factories, which take an input, manipulate the input in a way defined by the programmer and then produce an output.

Second, Haskell is a *strongly typed language.* This means that both the input and the output of a function have to be of a specified type. If a function is given a parameter of the wrong type, it will not produce an output, and instead give back an error. For example, we could have a function `plus` that takes two *integers* (that is, numbers without decimals, like 1, 2, 3 and so forth) and computes the sum of those integers. In Haskell, this function would look like:

```
plus :: Int -> Int -> Int
plus x y = x + y
```

Before we move on, let's take a brief walk through both lines of the function, starting with the second line, which is the *function definition.* It consists of three parts: the function *name*, which is `plus`, the function *parameters*, which are `x` and `y`, and the function *expression*, which is `x + y`. In factory terms, the parameters serve as inputs for the function and the expression defines what to do with the input, in order to generate a certain output (in technical terms: how to *evaluate* the function). Note that, in this function definition, we have not captured all the constraints that we wanted to put on our function yet: there is no information in the definition about the fact that we want the function to take two integers, and compute a new integer. This is where the first line of our code comes in. In natural language, it says that the function called `plus`, which takes two arguments, can have only integers as arguments, and must give back an integer result.

Now that we have a basic idea of how functions and types in Haskell work, we can move on to a third property of the language: its function evaluation is *lazy*, which is the opposite of *strict* evaluation. Let's explore the difference between these two types of evaluation by looking at an example. Suppose we have a function called `generateNaturals`, which has no parameters and just creates a list of all natural numbers, starting from 0. You can already guess that this function, once called, will never stop evaluating, since there are infinitely many natural numbers. Now let's look at another function, called `head`, which takes a list and returns the first element of that list. Suppose we want to use the result of `generateNaturals` as a parameter in the function `head`. We can do this by simply taking the function `generateNaturals`, and passing it on to `head` as its parameter. What will happen if we call `head` now? If Haskell were a strictly evaluating language, the obvious would happen: the function `generateNaturals` would have to be evaluated completely before its result could be passed down to `head`, and so the final result would never be evaluated. In the real, lazy Haskell, though, this is not the case (a fact that comes as a surprise to many who are faced with it for the first time). Instead, only that part of the result of the function `generateNaturals` *that is needed* by the higher function `head`, is evaluated. Because of this property, it is possible to work with infinite objects in Haskell as if they are finite, a feature that is useful for programmers who are working with concepts from logic.

1.5 Literate programming

The focus of this thesis will be on implementing the concepts from voting theory that we introduce. Because of this focus on implementation, the thesis will have the form of a *literate program*, that is, a computer program embedded in plain text. This way, all the code we develop can be explicitely presented to the reader, yet we do not expect the reader to fully understand every line of it. Instead, each chunk of code will be explained to the reader in natural language, which will also be the case for every logical and mathematical definition we present.

1.6 Relevance

The two main questions of Artificial Intelligence are: what is the nature of intelligence, and how can we model it? One way of looking at intelligence, is by studying intelligent behavior. Group behavior, especially in the context of knowledge that the individuals of a group possess, is such a type of behavior that is of interest to Articial Intelligence. Implementing models for this kind of behavior, like we do in this thesis, can help us gain a deeper insight into the overall nature of intelligence.

1.7 Structure

In section 2 we introduce the field of social choice theory and add the first lines of code to our implementation. Next, we discuss vote manipulation in section 3 and profile knowledge in section 4. In section 5 we put the pieces of code that we have written together. We then demonstrate and test the full program in section 6. Finally, we conclude the thesis in section 7, where we also give ideas for further research.

2 Basic concepts from social choice theory

Let's start with a simple example. Suppose we have a company that wants to hire a new employee. The company has received ten applications for the job and now has to choose which applicant to hire. Suppose now that not one, but three department heads have to make this decision together. Needless to say, the different department heads disagree on the ranking of the ten, and what is needed is some procedure for passing from the preferences of the individual heads to a collective resolution.

2.1 Voters and alternatives

In the example given above, we can identify the key players in voting theory: the *voters*, the department heads in this case, and a number of *alternatives* to vote on, represented by the applicants. Formally, the voters are denoted as $N = \{i_1, ..., i_n\}$, where $|N| \geq 2$, which means that the set of voters N needs to contain at least 2 members. An obvious restriction, since any reasonable vote with one voter would be a dictatorship. For the set of alternatives that voters can choose from, denoted as $X = \{x_1, x_2, x_3 ...\}$, there is also an obvious constraint, i.e. that the set cannot be empty. In formal language, this is expressed by: $|N| > 0$ or $N \neq \emptyset$, where $\emptyset$ is the empty set.

Let's move on to the *representation* of the objects that we just introduced in a Haskell program. It is important that a representation of an object captures all the *necessary* information about that object. But who knows what kind of information about our object we are going to need in the future? For now, we will keep it simple by just making sure that we can distinguish one object from another. If we want to add information to the objects later on, we can always create a data structure in which the old information about the objects is mapped onto some new information. We will simply number our objects to make them distinguishable from one another, and since this is the only information we need, we let our objects be represented by their numbers. In Haskell, we implement this by creating so called *type synonyms* for our objects. Such type synonyms are extra names that we give to data types in our program (in the case of our objects, to the integer type, written as `Int`), with the general purpose of improving the *readability* of the program code. Let's illustrate this with an example. Suppose we have some function of type `Int -> ...` , where we actually want the integer to represent a voter.

The idea of type synonyms now allows us to create a new type, called `Voter`, which we can use as a synonym for the type `Int`. This makes it possible for us to change the type `Int -> ...` of our function into `Voter -> ...` , without making any changes to the functionality of the program.

We write the first lines of our program now.

```
module VotingTheory where

import Data.List
import Data.Function
```

To this, we add type synonyms for our first objects.

```
type Voter = Int
type Alternative = Int
```

2.2 Ballots

In a vote, each individual in N is *endowed* with, and is asked to *express*, a preference over the alternatives in X. Such a preference is called a *ballot*. We call a certain ballot over a set of alternatives X an X-*ballot*. There are several ways to look at ballots, one of which is assumed by us and has the following properties:

- A ballot is a *linear ordering* of the set of alternatives X.
- We cannot compare preferences across voters. For example, our model cannot express that voter 1 likes x more than voter 2 likes y.
- There is no notion of preference intensity: we cannot model that, for example, voter 1's preference of x is more intense than voter 2's preference of x.
- We assume that every voter has the cognitive capacity to rank any two alternatives.

Let's look at what it means for a ballot to be a linear ordering of X. Suppose $X = \{A, B, C\}$ and we have a ballot $D = \langle B, C, A \rangle$. The fact that the elements of D are surrounded by the symbols $\langle$ and $\rangle$, means that the elements are *ordered* in some way (as opposed to the elements in X, which are unordered). We can also observe that D contains all the elements of X and vica versa. This covers the “ordering” part of the term “linear ordering”: all the elements from a set are ordered in some way. Now, let’s look at the “linear” part, which is the way in which the elements of the set are ordered. This way of ordering is determined by a so called *binary relation* between the elements in the set, which relates any two elements of the set to each other in some way. Suppose we have a binary relation R on a set A. If we have two elements x and y from A, we denote the R-relation between x and y as xRy. Now let’s look at linear orderings again. The binary relations R that result in a linear ordering all have some specific properties:

transitivity For all $w, v, u \in A$: if wRv and vRu, then wRu.

completeness For all $w, v \in A$: either wRv or vRw.

antisymmetry For all $w, v \in A$: if wRv and vRw, then $w = v$.

In the context of ballots, we let xRy mean: “x is preferred over y”. Since we have defined the preference relation to be a linear ordering, we can now say certain things about ballots. For example, the property of completeness enforces that for any pair of alternatives $\langle x, y\rangle$, either x is preferred over y, or y is preferred over x.

In Haskell, we will let ballots be represented by lists of alternatives. We do this by giving a type synonym declaration for ballots.

```
type Ballot = [Alternative]
```

The brackets in the definition, surrounding the word `Alternative`, indicate that we are talking about a *list* of alternatives. In general, `[a]` denotes a list of type `a`.

We also add two functions that manipulate ballots in a certain way. The first is a function that takes a list of integers as its input, and returns this input under the synonym `Ballot` only if it passes two validity checks: the ballot should be nonempty (the constraint we gave to the ballot type earlier) and should not contain duplicate values (this, because a linear order is based from a set, and sets do not contain duplicate values). A function like this looks like something we could call a *constructor* function, except that our function does not actually construct a new type. Instead, it just labels correct input with a different name (the type synonym).

```
toBallot :: [Int] -> Ballot
toBallot l = let sl = nub l
             in if length sl > 0 && sl == l
                then l
                else error "Invalid ballot"
```

We present some examples of how to use this function:

```
*VotingTheory> toBallot [1,2,3]
[1,2,3]                           Passes the test

*VotingTheory> toBallot [1,2,3,3]
*** Exception: Invalid ballot     Does not pass the test

*VotingTheory> toBallot []
*** Exception: Invalid ballot     Does not pass the test
```

The second function we give, filters a number of alternatives from a ballot, without changing the preference order of the alternatives. We will use this function later on.

```
filterFromBallot :: [Alternative] -> Ballot -> Ballot
filterFromBallot a b = toBallot $ filter ((flip elem) a) b
```

Some examples of usage:

```
*VotingTheory> filterFromBallot [3,4,1] [1,2,3,4]
[1,3,4]

*VotingTheory> filterFromBallot [] [1,2,3,4]
*** Exception: Invalid ballot

*VotingTheory> filterFromBallot [5] [1,2,3,4]
*** Exception: Invalid ballot
```

Let $L(X)$ denote the set of all possible ballots over X. For example: over $X = \{1, 2, 3\}$, these are: $\langle 1, 2, 3\rangle$, $\langle 2, 1, 3\rangle$, $\langle 3, 2, 1\rangle$, $\langle 2, 3, 1\rangle$, $\langle 3, 1, 2\rangle$ and $\langle 1, 3, 2\rangle$. Basically, computing $L(X)$ means generating all possible ways in which you can arrange the elements of X in an order, where every element of X occurs precisely once in the order. We also say that $L(X)$ gives all the *permutations* of the set X.

The size of $L(X)$ can be easily calculated, because it is a function of X. This function is the so called $factorial$ function. We say that the number of permutations on a set of size n is *n factorial*, also written as $n!$, which is equal to $n * (n-1) * (n-2) * ... * 1$. For example: 4! equals $4 * 3 * 2 * 1$ equals 24.

Because we work a lot with ballot permutations in voting theory, we implement the factorial function here.

```
fac :: Int -> Int
fac n = foldr (*) 1 [1..n]
```

2.3 Profiles

A $profile$ $R = \{R_1, ..., R_n\} \in L(X)^N$ is an n-tuple of ballots, where R_i is the ballot supplied by voter i. Here, $L(X)^N$ is the set of all possible profiles with N voters and a set of alternatives X. $R \in L(X)^N$, means that R is one of these possible profiles.

There are several ways for us to represent profiles in our Haskell program. Which of these to choose, depends, as before, on the information about the objects that we want to capture. We consider two possibilities of how to represent profiles:

voter oriented As a list of tuples that map voters onto the ballots they support.

ballot oriented As a list of tuples that map ballots onto the number of voters that support them.

In this thesis, we will actually use both representations, because they both have their specific advantages: the ballot oriented representation takes up less space, while the voter oriented representation contains more information. Note that, in both profile representation types, we did not keep strictly to the formal definition of a profile (in which the n-tuple contains only ballots, and no information about the voters). For us, it is more convenient to have one data structure in which we capture all the information we need about voters and their ballot choices.

We will proceed by implementing these representations.

```
type Profile = [(Voter,Ballot)]      -- Voter oriented
type Profile' = [(Ballot,Int)]       -- Ballot oriented
```

Next, we add a constructor function for the voter oriented profile type. This function may seem complicated, but basically, all it does is checking whether the profile contains two or more voters (a constraint we gave earlier), contains no duplicate voters and maps all voters onto valid ballots.

```
toProfile :: [(Int,[Int])] -> Profile
toProfile p =
 let a = map fst p
     a' = nub a
     b = nub $ map (length.snd) p
     c = nub $ map (sort.snd) p
 in if length a' > 1 && a == a'
                     && length b == 1
                     && length c == 1
    then p
    else error "Invalid profile"
```

Some examples:

```
*VotingTheory> toProfile [(1,[1,2]),(2,[2,1]),(3,[1,2])]
 [(1,[1,2]),(2,[2,1]),(3,[1,2])]

*VotingTheory> toProfile [(1,[1,2])]
 *** Exception: Invalid profile

*VotingTheory> toProfile [(1,[1,2]),(2,[2,1]),(3,[1,2,3])]
 *** Exception: Invalid profile
```

We also add a filter function for the voter oriented profile type. This function merely applies the ballot filter function to all ballots in the profile.

```
filterFromProfile :: [Alternative] -> Profile -> Profile
filterFromProfile a p =
 toProfile $ map (\(i,b) -> (i,filterFromBallot a b)) p
```

```
*VotingTheory> filterFromProfile [1,2] [(1,[1,2,3]),(2,[2,1,3])]
 [(1,[1,2]),(2,[2,1])]
```

We proceed by giving a function that converts the voter oriented representation to the ballot oriented representation. Note that conversion the other way around is impossible, since the voter oriented representation contains more information (to wit, which voter supports which ballot) than the ballot oriented respresentation.

```
toProfile' :: Profile -> Profile'
toProfile' p =
 let ballots = map (\b -> (head b,length b)) $ group
                                              $ sort
                                              $ map snd p
     perms = permutations $ snd $ head p
 in map (\b -> case lookup b ballots of
                              (Just n) -> (b,n)
                              Nothing -> (b,0)
        ) perms
```

We have actually added a small twist to the ballot oriented representation: we let it map all ballot permutations of the set of alternatives X to the number of voters that support them, because this will be useful later on. For example, we could have the following function call, which takes the voter oriented profile $\{\langle 1, \langle 1, 2, 3\rangle\rangle, \langle 2, \langle 1, 3, 2\rangle\rangle\}$ and converts it to its ballot oriented counterpart:

```
*VotingTheory> toProfile' [(1,[1,2,3]),(2,[1,3,2])]
```

This would evaluate to:

```
[([1,2,3],1),([2,1,3],0),([3,2,1],0),([2,3,1],0),
 ([3,1,2],0),([1,3,2],1)]
```

In our function input, we have a voter oriented profile of size 2, where only the two ballots $\langle 1, 2, 3\rangle$ and $\langle 1, 3, 2\rangle$ occur. Yet, in the result, we have a ballot oriented profile of size 6, in which all permutations of $\{1, 2, 3\}$ occur (remember that the number of permutations obtained from a set of size n is $n!$, so in our case, this is 3! equals $3 * 2 * 1$, which indeed equals 6). As we can see in the function result, the two ballots from the function input each are supported by 1 voter, while the other ballots are supported by 0 voters.

Finally, we add a filter function for the ballot oriented profile type. This filter function is a bit more complex than the previous one, since filtering a number of alternatives from two different ballots, can result in the same ballot. In fact, this filter function reduces an input of size $n!$, where n is the size of the set of alternatives on which the ballots in the input are based, to an output of size $(n - m)!$, where $m = n - x$ and x is the size of the set of alternatives to filter.

```
filterFromProfile’ :: [Alternative] -> Profile’ -> Profile’
filterFromProfile’ a p’ =
 map (\g -> ((fst.head) g,sum $ map snd g))
             $ groupBy ((==) ‘on‘ fst) $ sort
             $ map (\(b,n) -> (filterFromBallot a b,n)) p’
```

An example:

```
*VotingTheory> filterFromProfile’ [2,1]
 [([1,2,3],1),([2,1,3],0),([3,2,1],0),([2,3,1],0),
 ([3,1,2],0),([1,3,2],1)]

[([1,2],2),([2,1],0)]
```

2.4 Preference aggregation

We move on to another aspect of voting: the mechanisms of *preference aggregation*. There are several preference aggregation mechanisms, one of which, the *social choice function* (SCF), we will treat in more detail.

SCF's are functions $F : L(X)^N \rightarrow 2^X \backslash \{\emptyset\}$ that map profiles onto nonempty sets of alternatives. Let's look at the different parts of the definition. The $L(X)^N$ part before the arrow is the input of the SCF function, which is a profile. Then we have the part on the other side of the arrow. 2^X denotes the so called *powerset* of X, which is the set containing all *subsets* of X. For example: $2^{\{1,2,3\}}$ equals $\{\emptyset, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}$. Note that this set contains the empty set $\emptyset$, which, of course, should not be in the set of possible preference aggregation results, since we don't want to end up with no winner at all. This is where the $\backslash\{\emptyset\}$ part in our definition comes in, to ensure that the function result can, indeed, not be the empty set.

A well-known example of an SCF is the *Borda rule*, proposed by the French engineer and political scientist Jean-Charles de Borda (1733-1799). In this rule, an alternative receives $|X| - 1$ points from every voter who ranks her first, $|X| - 2$ points from every voter who ranks her second, and so forth. This means that if we have the three ballots $\langle 1, 2, 3 \rangle$, $\langle 2, 3, 1 \rangle$ and $\langle 3, 1, 2 \rangle$, each of the alternatives receives three points. A problem arises here: the result {1,2,3} is a *tie*!

Let's define a Haskell type synonym with which we can express ties, again using lists to represent sets.

```
type UnresolvedResult = [Alternative]
```

With this new structure defined, we are ready to give a type synonym for SCF's.

```
type SCF = Profile' -> UnresolvedResult
```

At this point, we are still stuck with ties. Luckely for us, there is a solution to this problem: the so called *tie breaking rule* (TBR). In the case of our Borda rule tie, we could make a rule that simply takes the first element from the list of alternatives that is the result of the preference aggregation.

In Haskell, TBR's can be given the following type:

```
type TBR = UnresolvedResult -> Result
type Result = Alternative
```

Implementing our simple tie breaking rule is now very easy.

```
simpleTBR :: TBR
simpleTBR = head
```

```
*VotingTheory> simpleTBR [1,2,3]
 1
```

Let's turn to the implementation of some well-known voting rules in Haskell. We start with the Borda rule that we have already discussed.

```
bordaRule :: SCF
bordaRule p =
 let points = transpose $ map (\(b,n) -> sort
       $ zip b (map (*n) (reverse [0..(length b - 1)]))) p
     count = map (\ps@((a,_):_) ->
                          (((sum.(map snd)) ps),a)) points
 in map snd $ last $ (groupBy ((==) `on` fst)) $ sort count
```

```
*VotingTheory> bordaRule (toProfile' [(1,[1,2,3]),(2,[1,3,2])])
 [1]

*VotingTheory> bordaRule (toProfile' [(1,[1,2,3]),(2,[2,1,3])])
 [1,2]

*VotingTheory> bordaRule (toProfile' [(1,[1,2,3]),(2,[2,3,1]),
 (3,[3,1,2])])

  [1,2,3]
```

Next, we turn to a particularly undemocratic “rule”: the *dictator rule*. We can implement this rule in several ways. One way would be to *randomly* pick a voter from the profile and let that voter’s most preferred alternative, be the election winner. In our actual implementation, we let the winning alternative be the alternative that is most preferred by the first voter in the list that respresents the profile.

```
dictatorRule :: SCF
dictatorRule p = [(head.fst.head) p]
```

```
*VotingTheory> dictatorRule (toProfile’ [(1,[1,2,3]),(2,[3,2,1])])
 [1]
```

Note that this rule, though it is very simple, does capture the essence of a dictatorship: in the election results, the preference of only one voter is reflected. Also worth noting, is that this rule never results in a tie.

A third important voting rule is the so called *majority* or *plurality rule*. This rule takes the favourite alternative of every voter, and then simply counts which alternative has, or which alternatives have, been ranked first the most.

```
majorityRule :: SCF
majorityRule p =
 let ballots = concatMap (\(b,n) -> map snd
                         $ zip [0..n-1] $ cycle [b]) p
     top = group $ sort
                 $ concatMap (\l -> if length l <= 1
                                    then []
                                    else [head l]) ballots
     win = sortBy (compare ‘on‘ fst)
                    $ zip (map length top) (map head top)
 in map snd $ last $ groupBy ((==) ‘on‘ fst) win
```

```
*VotingTheory> majorityRule (toProfile’ [(1,[1,2]),(2,[1,2]),
 (2,[2,1])])

  [1]

*VotingTheory> majorityRule (toProfile' [(1,[1,2]),(2,[2,1])])
 [1,2]
```

Let's now define a function that puts an SCF and a TBR together, and returns the final result directly.

```
vote :: SCF -> TBR -> Profile' -> Result
vote swf tbr = tbr.swf
```

```
*VotingTheory> vote bordaRule simpleTBR
 (toProfile' [(1,[1,2,3]),(2,[2,3,1]),(2,[3,1,2])])

  1
```

This function captures the process of a simple vote that always ends with a tie break.

We can make the process more interesting by considering votes with multiple rounds. Such votes allow the voters to break the tie themselves, possibly by means of manipulation.

3 Voting strategies and manipulation

In the study of manipulation of voting systems, there are two rather distinct approaches:

1. One begins with an explicitly given aggregation procedure and attempts to find the ways in which a voter can secure a more favorable election outcome by changing his or her ballot.
2. One starts with an explicit notion of what it means for a voter to prefer one outcome to another and attempts to find all the aggregation procedures (of a certain kind) that are manipulable in this sense.

Generally, the first approach is felt to be much easier than the second approach. We gave an example of a result from the second approach in the beginning of this thesis (that for an important class of voting rules, only dictatorships are unmanipulable), but it is the first approach that we will take on in the rest of this thesis.

3.1 Choosing a ballot to manipulate with

First, we have to define what we mean by manipulation.

Definition 1 *When we say that a certain voter i, involved in a vote v, manipulates v, this means that i changes her ballot in order to achieve an optimal election outcome,* while the ballots of the other voters remain fixed.

The concept of group manipulation, also called *coalitional manipulability*, is beyond the scope of this thesis.

Before we define what an "optimal election outcome" is, let's look at what it means for one election outcome to be *favorable* over another for a voter. Suppose we have a voter i who supports some ballot B, and we have two voting results r_1 and r_2. We define that r_1 is a *better* result for i than r_2 if and only if the set r_1 *weakly dominates* the set r_2, which means that everything in r_1 is at least as good as everything in r_2, and something in r_1 is better than something in r_2. At the level of alternatives, it is trivial that for two alternatives A and B, A is better for voter i than B if i prefers A over B.

We can now define a general *payoff function* for voting results, that is, a function which gives a score to a result, where a result with a higher score is more favorable for a voter than a result with a lower score. We will need such a payoff function if we want to reason about manipulation, because the success of an act of manipulation depends on the results given back by such a function.

Definition 2 *Suppose we have a vote v, in which a manipulator m is involved. Let R be the set of voting results that can possibly be the outcome of v if m takes on any ballot. The* payoff *for i of a voting result $r \in R$, then, is the number of voting results from R that are weakly dominated by r.*

We can also define the *optimal* result now, as the result with the highest payoff.

Let's implement our new definitions as Haskell functions. In doing so, we will make some small changes to our payoff function: instead of giving it a list of voting results (containing one or more optimal results) as its input, we give it a list of tuples that map ballots onto voting results. For example: suppose we have a voter i, a set of alternatives $X = \{1, 2, 3\}$, a ballot $\langle 1, 2, 3\rangle$ and a set of voting results $R = \{r_1, r_2, r_3, ...\}$. The input of our payoff function will then look like: $\{\langle b_1, r_1\rangle, \langle b_2, r_2\rangle, \langle b_2, r_1\rangle, ...\}$, where $b_1, b_2, b_3, ...$ are X-ballots that voter i can choose for her manipulation, mapped onto the voting result that choosing this ballot leads to.

Using this input format will be expedient later on, where the manipulator will need to cast a new ballot and has to calculate which ballot yields the highest payoff. For this reason, we also let the output of our function be one of the ballots that, in the input, was mapped onto an optimal voting result, and we will call our function "bestBallot" instead of "payoff".

We start with a smaller auxiliary function, which checks whether one alternative is at least as good as another, with respect to a certain ballot. It does so by simply checking which alternative is ranked equally high or higher in the list that represents the ballot. Therefore, the result of the function is of type `Bool`, which can take on the value `True` or `False`.

```
isAtLeastAsGood :: Alternative -> Ballot ->
                                    Alternative -> Bool
isAtLeastAsGood a1 b a2 =
 let index1 = (\(Just d) -> d) $ elemIndex a1 b
     index2 = (\(Just d) -> d) $ elemIndex a2 b
 in index1 <= index2
```

```
*VotingTheory> isAtLeastAsGood 1 [1,2,3] 2
 True

*VotingTheory> isAtLeastAsGood 1 [1,2,3] 1
 True

*VotingTheory> isAtLeastAsGood 2 [1,2,3] 1
 False
```

The second function checks whether an alternative is stricly better than another one, again with respect to a certain ballot. This function checks which alternative is ranked strictly higher in the list that represents the ballot.

```
isBetter :: Alternative -> Ballot -> Alternative -> Bool
isBetter a1 b a2 =
 let index1 = (\(Just d) -> d) $ elemIndex a1 b
     index2 = (\(Just d) -> d) $ elemIndex a2 b
 in index1 < index2
```

```
*VotingTheory> isBetter 1 [1,2,3] 2
 True

*VotingTheory> isBetter 1 [1,2,3] 1
 False

*VotingTheory> isBetter 2 [1,2,3] 1
 False
```

The next function implements the concept of weak domination, by applying `isAtLeastAsGood` and `isBetter` to all possible pairs $\langle a_1, a_2 \rangle$ of alternatives, where a_1 is from one voting result and a_2 from another (if this is hard to understand, look back at the definition of weak domination).

```
weaklyDominates :: UnresolvedResult ->
                      UnresolvedResult -> Ballot -> Bool
weaklyDominates s1 s2 b =
 let atLeastAsGood = all (\e ->
                        all (isAtLeastAsGood e b) s2) s1
     better = any (\e -> any (isBetter e b) s2) s1
 in atLeastAsGood && better
```

```
*VotingTheory> weaklyDominates [1] [1,2] [1,2,3]
 True

*VotingTheory> weaklyDominates [1] [1] [1,2,3]
 False

*VotingTheory> weaklyDominates [1,3] [2] [1,2,3]
```

```
 False
```

We add an auxiliary function now, which pairs elements from a list, because we need this in our `bestBallot` function.

```
generalPair :: [a] -> [(a,a)]
generalPair xs = pair’ xs (length xs)
 where
  pair’ xs 0 = []
  pair’ (x : xs) n =
   map (\y -> (x,y)) xs ++ pair’ (xs ++ [x]) (n - 1)
```

```
*VotingTheory> generalPair [1,2,3]
 [(1,2),(1,3),(2,3),(2,1),(3,1),(3,2)]
```

Finally, the function that calculates the optimal ballot for a manipulator. This function first pairs all the voting results from the input using `generalPair`, then feeds them to the weak domination function, counts how many other voting results are weakly dominated by each voting result and then declares the voting result that dominates most other results, winner. We can also see this as a tournament: each player (voting result) has to compete against all other players in a game (the very simple game of who weakly dominates who). The player who has won most games at the end, wins.

```
bestBallot :: [(Ballot,UnresolvedResult)] ->
                                         Ballot -> Ballot
bestBallot br b =
 let pairs = generalPair $ map snd br
     dominations = concatMap (\(s1,s2) ->
                              if weaklyDominates s1 s2 b
                              then [s1]
                              else []) pairs
     result = if dominations /= []
              then snd $ last
                       $ sortBy (compare `on` fst)
                       $ map (\ds -> (length ds,head ds))
                       $ group $ sort dominations
              else snd $ head $ br
 in (\(Just e) -> e) $ lookup result
                               $ map (\(x,y) -> (y,x)) br
```

```
*VotingTheory> bestBallot [([1,2],[2]),([2,1],[1])] [2,1]
 [1,2]                 (Because [2] weakly dominates [1])

*VotingTheory> bestBallot [([1,2],[2]),([2,1],[1])] [1,2]
 [2,1]                 (Because [1] weakly dominates [2])
```

3.2 Yet another profile representation

With vote manipulation, we need to keep account of the difference between a voter's *actual preference order* and the ballot she supports (which may be an intended misrepresentation of her actual preference order).

```
type ActualPreferenceOrder = Ballot
```

A ballot can now either be a $truthful$ or an $untruthful$ preference order.

We define a data structure in which we can capture the difference between truthful and untruthful preference orders. We will call this data structure an *extended profile*.

```
type Profile'' = [((Voter,ActualPreferenceOrder),Ballot)]
```

An example:

```
[((1,[1,2]),[1,2]),((2,[2,1]),[1,2])]
```

To this, we add a constructor function for the new type.

```
toProfile'' :: Profile -> Profile''
toProfile'' p = zip p $ map snd p
```

```
*VotingTheory> toProfile'' [(1,[1,2,3]),(2,[3,2,1])]
 [((1,[1,2,3]),[1,2,3]),((2,[3,2,1]),[3,2,1])]
```

Next comes a function that takes an extended profile and extracts a voter oriented profile from it, in which the ballots assigned to the voters are the ballots that the voters use in their manipulation.

```
toManipulableProfile'' :: Profile'' -> Profile
toManipulableProfile'' p'' =
 let unzippedProfile = unzip p''
     p = map fst $ fst unzippedProfile
 in zip p (snd unzippedProfile)
```

```
*VotingTheory> toManipulableProfile''
 [((1,[1,2]),[2,1]),((2,[1,2]),[2,1])]

  [(1,[2,1]),(2,[2,1])]
```

We also add a function that extracts a voter oriented profile containing the voters' actual ballots.

```
toActualProfile :: Profile'' -> Profile
toActualProfile = fst.unzip
```

```
*VotingTheory> toManipulableProfile''
 [((1,[1,2]),[2,1]),((2,[1,2]),[2,1])]

  [(1,[1,2]),(2,[1,2])]
```

We proceed by adding a function that replaces the voters' old manipulation ballots with a set of new ones.

```
updateProfile'' :: Profile'' -> NewBallots -> Profile''
updateProfile'' p = zip $ map fst p

type NewBallots = [Ballot]
```

```
*VotingTheory> updateProfile''
 [((1,[1,2]),[1,2]),((2,[1,2]),[1,2])] [[2,1],[2,1]]

  [((1,[1,2]),[2,1]),((2,[1,2]),[2,1])]
```

Finally, we add a filter function for the extended profile type.

```
filterFromProfile'' :: [Alternative] -> Profile''
                                          -> Profile''
filterFromProfile'' a p'' =
 map (\((i,b),b') ->
  ((i,filterFromBallot a b),filterFromBallot a b')) p''
```

```
*VotingTheory> filterFromProfile'' [1,2]
 [((1,[1,2,3]),[1,3,2]),((2,[3,2,1]),[3,1,2])]

  [((1,[1,2]),[1,2]),((2,[2,1]),[1,2])]
```

3.3 Vote simulation

Now that we have made the necessary preperations, we can start looking at the game of manipulation itself. Indeed, it can be seen as a game, not only intuitively, but also as a concept borrowed from *game theory* (from which we also borrowed the concept of weak domination). In this context, an election can be though of as a game in which a *strategy* for a player (voter) is a choice of ballot, and the *outcome* of the game the set of winners in the election.

We already have a function `bestBallot` for determining the strategy (choice of ballot to manipulate with) of a manipulator. All we need now, is a way to calculate the possible voting results that we want to give to the `bestBallot` function as its input. Since voting results are calculated by means of voting rules and voting rules take a profile as their input, a voter needs to know the profile, so that she can simulate the vote with that profile and all possible ballots, to see which ballot is optimal for her. We make a fundamental assumption here that we will use throughout the rest of this thesis, but that will make things harder for us now: the voters do not know what the profile looks like! Of course, this does not mean that they know nothing. For example, every voter does know her own share in the profile (we assume that every voter has the cognitive capacity to know this). Also, we assume that each voter $i \in N$ participating in a vote v, where X is the set of alternatives to vote on, knows how many voters there are in v (in other words: what the size of N, $|N|$, is), and that those voters know that the real profile is a member of the set of all profiles where N is the set of voters and X the set of alternatives.

Now that we have moved our focus from vote manipulation to the knowledge of voters, it is time to branch out to the field of dynamic-epistemic logic, where we are most likely to find the tools necessary for modelling what our voters know.

4 Dynamic-epistemic logic in voting theory

The system of epistemic logic is based on the question: *what is information?* We use one of its main approaches here by modelling *uncertainty* instead of information. For instance, an individual might not know whether it is raining outside or not (she is uncertain about that). Therefore, if we ask her if it's raining outside, she will answer that she considers it possible that it's raining. If we ask her if it's *not* raining outside, she will tell us that she also considers that a possibility. Of course, individuals can be uncertain about more than just two things, and the uncertainty of many individuals can be expressed in one model. From now one, we will call the possibilities in a model *worlds*. A model has to have at least one world: the real world, which has a unique status because it is the only one of its kind. In other words: all worlds, expect the real world, are *unreal*. An individual (in our case, voter) might not be able to *distuingish* the real world from possibly many other unreal worlds.

Dynamic logic is the branch of logic that is concerned with modelling *action*. In the context of knowledge, $information$ can cause *changes in uncertainty* in an epistemic model. In other words: worlds that at first seemed plausible to an individual, can be recognized as being unreal after an information *update*. The language of dynamic-epistemic logic can be used to model these actions that involve information.

Let's explore the concepts of dynamic-epistemic logic and see how we can use them for our purpose of modelling the knowledge of voters about their profile.

4.1 Semantics

Models M for the dynamic-epistemic language are *triples* (tuples containing three elements) $(W, \{\rightarrow_i \mid i \in I\}, V)$ where W is a set of worlds, the $\rightarrow_i$ are binary accessibility (uncertainty) relations between worlds, and $V : W \rightarrow P \rightarrow \{TRUE, FALSE\}$, where P is the set of all propositions in the dynamic-epistemic language, is a valuation function that takes a propisition $p \in P$ from a world $w \in W$ and assigns a truth value to it. In the rest of the thesis, we will mostly use *pointed models*, which are tuples (M, s), where M is a model and s is the real world in that model.

Let's look at an example of a model. The model for the example at the beginning of this section would be as follows: $(\{v = \{r\}, w = \{\neg r\}\}, \{v \rightarrow_i w, w \rightarrow_i v, v \rightarrow_i v, w \rightarrow_i w\}, \{V(v, r) = TRUE; V(w, r) = FALSE\})$, where v and w are worlds, the proposition r means "it is raining outside" and "$V(v, r) = TRUE$" means: the proposition r at world v is true (likewise,

"$V(w, r) = FALSE$" means: the proposition r at world w is false, and so the proposition $\neg r$ (it is *not* raining outside) is true here).

Let's now look at the relations between the worlds. First, we have the two relations $v \to_i w$ and $w \to_i v$. Together, they basically say that individual i is uncertain about which of the worlds v and w is the real one. The relations $v \to_i v$ and $w \to_i w$, saying that i cannot distiguish worlds from themselves, are given explicitely here, but could actually be left out, because the relations in our model are actually of a special kind, in which it is enforced that all worlds are connected to themselves: they are so called *equivalence relations*, satisfying the following three conditions (one of which, transitivity, we have already seen before) for a binary relation R (the relation of uncertainty in our case) of a set of worlds W:

reflexivity For all $w \in W$: wRw.

symmetry For all $w, v \in W$: if wRv then vRw.

transitivity For all $w, v, u \in W$: if wRv and vRu, then wRu.

Here, reflexivity is the property that enforces all worlds to be connected to themselves by the relation.

One can think of these relations as partitioning the total set of worlds into a number of disjoint, maximal "zones" of worlds connected by the relation.

How can we use the notion of uncertainty in the context of vote manipulation? What we need, is to know what knowledge the voters of a vote v have about the profile of v, because we want them to be able to simulate a vote using their knowledge of the profile in order to calculate an optimal manipulation strategy. For this purpose, we can use the modelling of uncertainty done in epistemic logic to express the knowledge that voters have about their profile (which was our "missing link" in the previous section). This is exactly the concept of profile knowledge that we mentioned earlier in the thesis.

Definition 3 *The* profile knowledge *of a voter i is the uncertainty of i regarding what the profile looks like.*

4.2 Representation

For the purpose of representing profile knowledge in our program, we consider a simple example of a profile P with two voters, 1 and 2, and also two alternatives, A and B. Suppose you are voter 1. From your point of view, you cannot distinguish between the following two worlds: the world in which voter

2 has the ballot $\langle A, B \rangle$ and the world in which voter 2 has the ballot $\langle B, A \rangle$. In this simple case, voter 1 actually has the highest degree of uncertainty that is possible: she knows nothing about the preferences of any other voter. Therefore, all possible profiles have to be captured in our implementation. But how do we do this?

One approach is to let the worlds in our model contain propositions about which voter has which ballot (similar to the kind of information that our voter oriented profile representation holds). If we did this, however, and we started experimenting with bigger examples, we would soon realize that in this approach, we model a great deal of knowledge that we actually do not need (and because of that, our models get astronomically large). Does it matter whether a certain voter i or another voter j has the preference order B? No, because the identities of the voters are not at all involved in the process of preference aggregation! In fact, all we need to know, is *how many* voters support a certain ballot. This is where the ballot oriented profile representation we defined earlier comes in.

We move to the representation of a dynamic-epistemic pointed model for voting theory in Haskell. We know that pointed models contain four elements: a set of possible worlds, a set of relations, a valuation function and a real world. Thus, representing a pointed model means representing these four objects. Yet, we do not have to adhere strictly to the formal definition of a pointed model, since this definition is designed for dynamic-epistemic models in *general*, while for us, it suffices (and may even be beneficial) to work with a slightly specialized model.

The worlds in our model, for instance, actually need to contain only one proposition (to wit, a list of how many voters support each ballot). For this reason, we might as well define a world as being represented entirely by that list. This allows us to actually discard the third member in the model triple we discussed, V (the valuation function), since each world is represented by its only, true proposition.

We are left with the relations between our worlds. Since these relations are equivalence relations, we can *partition* our set of possible worlds and use these partitions for our representation. In order to achieve this, we could use a list to map each voter to the list of worlds that, from her point of view, could be the real world. If we look back at section 3, though, we discover that we can simplify our solution even further. Remember the definition of manipulation: it says specifically that only one voter changes her ballot, while the ballots of the other voters stay fixed. Since we let our voters use their knowledge only for the act of manipulation, we have to keep track of the knowledge of just one voter: the manipulator.

We can use this fact to make our implementation of epistemic-dynamic models very compact, by defining them as tuples, consisting of a list of worlds, which are the worlds considered possible by the manipulator, and the real world.

4.3 Implementation

We define a Haskell type synonym for the worlds in our models.

```
type World = Profile'
```

Next, we define a type synonym for our models.

```
type Model = ([World],RealWorld)
type RealWorld = World
```

We can now define a function that calculates the worlds that are initially considered possible by the manipulator. These are actually all possible worlds, except those in which the manipulator's own ballot is supported by zero voters.

First, we give a type synonym for the set of all possible ballot permutations, which we will use in our function that creates all possible worlds.

```
type BallotPermutations = [Ballot]
```

We proceed by giving the function that generates all possible worlds for a model. Because these worlds are actually profiles, we can also say that this function generates all *legal permutations* of a profile.

```
possibleWorlds :: Int -> BallotPermutations -> [World]
possibleWorlds ag perms =
 map (zip perms) $ gen (length perms) ag

gen m n = [xs | xs <- gen' m n [],
           sum xs == n]
gen' 1 n acc = [[xs] | xs <- [0..(n - sum acc)]]
gen' m n acc = [(x : xs) | x <- [0..(n - sum acc)],
                           xs <- gen' (m - 1) n (x : acc)]
```

```
*VotingTheory> possibleWorlds 3 (permutations [1,2])

 [[([1,2],0),([2,1],3)],
  [([1,2],1),([2,1],2)],
  [([1,2],2),([2,1],1)],
  [([1,2],3),([2,1],0)]]
```

Let's analyze this function by looking at its type. The function takes two inputs (an integer that represents the number of agents for the model, and a list of ballots that contains all permutations of a given ballot) and returns a list of worlds for the model. The inputs tell the function what kind of worlds (remember that worlds are actually possible profiles) it has to produce.

We can best illustrate how this function works by looking at the result of an example evaluation.

```
*VotingTheory> possibleWorlds 2 (permutations [1,2])

 [[([1,2],0),([2,1],2)],
  [([1,2],1),([2,1],1)],
  [([1,2],2),([2,1],0)]]
```

Here, the function has produced a list of three profiles: one in which both voters support the ballot $\langle 2, 1\rangle$, one in which both ballots are supported by one voter, and one in which both voters support the ballot $\langle 1, 2\rangle$. For $\{1, 2\}$-ballots and $N = \{1, 2\}$, these are all possible ballot oriented profiles, and thus all possible worlds in our model.

Let’s move on now and define a function that converts a profile to the list of worlds initially considered possible by the manipulator.

```
profileToWorlds :: Profile’ -> [World]
profileToWorlds p =
 possibleWorlds nAgents perms
  where
   b = (fst.head) p
   perms = permutations $ sort b
   nAgents = sum $ map snd p
```

```
*VotingTheory> profileToWorlds (toProfile’ [(1,[1,2]),(2,[2,1])])

 [[([1,2],0),([2,1],2)],
  [([1,2],1),([2,1],1)],
  [([1,2],2),([2,1],0)]]
```

To this, we add a function that, from the list of all possible worlds, deletes those worlds in which a given ballot is supported by zero voters.

```
ruleOutWorlds :: [World] -> Ballot -> [World]
ruleOutWorlds ws b = concatMap (\l ->
 let popularity = (\(Just e) -> e) $ lookup b l
 in if popularity == 0 then [] else [l]) ws
```

```
*VotingTheory> ruleOutWorlds
 [[([1,2],0),([2,1],2)],
  [([1,2],1),([2,1],1)],
  [([1,2],2),([2,1],0)]]
 [1,2]

  [[([1,2],1),([2,1],1)],
   [([1,2],2),([2,1],0)]]
```

We also add a filter function for lists of worlds.

```
filterFromWorlds :: [Alternative] -> [World] -> [World]
filterFromWorlds a ws =
 nub $ map (filterFromProfile' a) ws
```

```
*VotingTheory> filterFromWorlds [1,2]
 [[([1,2],0),([2,1],2)],
  [([1,2],1),([2,1],1)],
  [([1,2],2),([2,1],0)]]

  [[([1,2],0),([2,1],2)],
   [([1,2],1),([2,1],1)],
   [([1,2],2),([2,1],0)]]
```

Finally, we give a function that converts a profile to a full dynamic-epistemic model.

```
toModel :: Profile -> Manipulator -> Model
toModel p m =
 let p' = toProfile' p
     worlds = ruleOutWorlds (profileToWorlds p')
  ((\(Just e) -> e) $ lookup m p)
     realWorld = p'
 in (worlds,realWorld)
```

```
*VotingTheory> toModel [(1,[1,2]),(2,[2,1])] 1

 ([[([1,2],1),([2,1],1)],
   [([1,2],2),([2,1],0)]],
   [([1,2],1),([2,1],1)])
```

5 Putting the pieces together

In the previous sections we implemented some of the basic concepts from voting theory. We also explored and implemented the concept of vote manipulation and extended the field of voting theory with the notion of profile knowledge. What we are left with now, is integrating these parts.

We have the function `toModel` that creates a dynamic-epistemic model from a profile. Remember that, two sections earlier (in section 3), we actually needed this function for simulating a vote, in order for a manipulator to calculate her best move.

We can now define a function that takes a possible world and simulates a vote using this world as the profile in that vote. The only question is: what should this function return? We can answer this question by remembering that we actually want to model vote manipulation. Therefore, the output of the simulation should tell the manipulator what choice of ballot results in what outcome of the vote (note that this is exactly the input of the function `bestBallot` from section 4). Thus, we let the function take a world (from the set of worlds considered possible by the manipulator before her change of ballot) and return a list that maps a choice of ballot onto the voting results that this choice leads to.

5.1 Implementing vote simulation

First, though, we define a function that calculates the profiles resulting from a change of ballot by the manipulator in the old profile.

```
simulationWorlds :: World -> Ballot -> [World]
simulationWorlds w b =
 let perms = map fst w
     popularity = (\(Just e) -> e) $ lookup b w
     index = (\(Just i) -> i) $ elemIndex b (map fst w)
     splitWorld = splitAt (index + 1) (map snd w)
     newWorld = (init $ fst splitWorld)
         ++ [popularity - 1] ++ (snd splitWorld)
 in (map (zip perms) $ map (\i -> take i newWorld
      ++ [(!!) newWorld i + 1]
      ++ drop (i + 1) newWorld)
   [0..((length newWorld) - 1)])
      ++ [zip perms newWorld]
```

Let's demonstrate the functionality of this function with an example.

```
*VotingTheory> simulationWorlds [([1,2],1),([2,1],1)] [1,2]

 [[([1,2],1),([2,1],1)],
  [([1,2],0),([2,1],2)],
  [([1,2],0),([2,1],1)]]
```

As can be seen in the example, the function takes two arguments: a ballot oriented profile (which functions as a world in our dynamic-epistemic models) and a ballot, which is the ballot that the manipulator uses in her manipulation. The result of the function evaluation, is a list of three worlds, each of which is the result of a change (or no change) of ballot by the manipulator. In the first world, nothing changed, because the manipulator kept the same ballot. In the second world, the manipulator changed her ballot from $\langle 1, 2\rangle$ to $\langle 2, 1\rangle$. Then we have the third world, which is actually a special one, because it is the result of an extra possibility for the manipulator that we have added. It is the possibility of *no-show*. This means that the manipulator can also choose to cast no ballot at all. Of course, doing so does not mean that the manipulator is not participating in the vote anymore - she just expresses her participation by choosing to cast no ballot.

Now we can define the function that actually performs the simulation.

```
simulate :: World -> Ballot -> SCF ->
                    [(Ballot,UnresolvedResult)]
simulate w b rule =
 let perms = map fst w
     simWorlds = simulationWorlds w b
     showWorlds = init simWorlds
     noShowWorld = last simWorlds
 in (zip perms $ map rule $ showWorlds)
    ++ [([],rule noShowWorld)]
```

This function takes three parameters: a world (or, as we may also say, profile), a ballot, to wit the ballot that is supported by the manipulator, and a voting rule. It then runs our previous function `simulationWorlds` on the world in the input. After that, we have a number of worlds, a ballot and a voting rule. Each of our new worlds represents a specific ballot change by the manipulator. This fact is reflected in the output of our function, which is a list of tuples that map a ballot (chosen by the manipulator in the new world) onto a voting result. This voting result, onto which a ballot X is mapped, is exactly the result of simulating a vote with the given voting rule and the profile in which the manipulator supports ballot X. For every ballot change, such a result is calculated.

For clarity, we give one example.

```
*VotingTheory> simulate [([1,2],1),([2,1],1)] [1,2] bordaRule

 [([1,2],[1,2]),
  ([2,1],[2]),
  ([],[2])]
```

The empty list in the third tuple in the output represents the act of no-show.

5.2 Knowledge and profile update functions

Let's move on to the dynamic part of our epistemic model: updating a manipulator's knowledge with new information. This information is the result of a preference aggregation, which took a profile as its input. The manipulator can use this result to calculate which profiles (or in knowledge terms: worlds) might have led to the result.

We give a knowledge update function that deletes those worlds from the list of worlds in a model, which could not have led to the given voting result.

```
updateWorlds :: [World] -> UnresolvedResult ->
                                      SCF -> [World]
updateWorlds worlds result rule =
 concatMap (\p' -> if rule p' == result
                            then [p']
                            else []) worlds
```

```
*VotingTheory> updateWorlds
 [[([1,2],1),([2,1],1)],
 [([1,2],0),([2,1],2)],
 [([1,2],0),([2,1],1)]] [2] bordaRule

  [[([1,2],0),([2,1],2)],
   [([1,2],0),([2,1],1)]]
```

We also give a function that uses the previous function for updating an entire model.

```
modelUpdate :: Model -> UnresolvedResult -> SCF -> Model
modelUpdate (worlds,realWorld) result rule =
 let updateWs = updateWorlds worlds result rule
 in (updateWs,realWorld)
```

Remember that a model is represented by a tuple, which contains a list of worlds and the real world. The function `modelUpdate` merely runs the function `updateWorlds` on the list worlds in the model, copies the real world and put the worlds and the real worlds together in a new tuple.

We are now ready to define a function that performs the actual manipulation, using `simulate` to try all possible ballot changes and `bestBallot` to calculate which ballot change is optimal in most worlds.

```
manipulate :: [World] -> Ballot -> SCF -> Ballot
manipulate ws b r =
 let candidateBallots = map (\w ->
                         bestBallot (simulate w b r) b) ws
 in snd $ last $ sortBy (compare `on` fst) $ map (\ds ->
      (length ds,head ds)) $ group $ sort candidateBallots
```

```
*VotingTheory> manipulate
 [[([1,2],1),([2,1],1)],
  [([1,2],0),([2,1],2)],
  [([1,2],0),([2,1],1)]] [1,2] bordaRule

  [1,2]
```

Finally, we give a profile update function in which the manipulation is performed (using the function `manipulate`) and a new profile is returned, which represents the situation after the manipulation.

```
profileUpdate :: Model -> Profile''
                             -> Voter -> SCF -> Profile''
profileUpdate (worlds,realWorld) p'' manipulator rule =
 let actualProfile = toActualProfile p''
     updateBallots = map (\(i,b) ->
      if i == manipulator
      then manipulate worlds b rule
      else b) actualProfile
 in concatMap (\p@((v,a),b) ->
      if b == []
      then []
      else [p]) $ updateProfile'' p'' updateBallots
```

```
*VotingTheory> profileUpdate
 ([toProfile' [(1,[1,2,3]),(2,[2,3,1]),(3,[3,1,2])]]
  ,
  toProfile' [(1,[1,2,3]),(2,[2,3,1]),(3,[3,1,2])]
 )                                                          Model

 (toProfile'' [(1,[1,2,3]),(2,[2,3,1]),(3,[3,1,2])])   Profile''

 1                                           Voter (manipulator)

 bordaRule                                           Voting rule

  [((1,[1,2,3]),[2,1,3]),
   ((2,[2,3,1]),[2,3,1]),
   ((3,[3,1,2]),[3,1,2])]
```

5.3 Main functions

Now that we can update both profile knowledge models and profiles themselves, we are ready to incorporate this functionality in complete voting systems. We will present two systems in the form of simple user interfaces, which will allow us to walk through a voting process step by step and receive information about what is happening in the process, at the same time.

The first process that we are going to implement, is one in which all voters cast their ballots at the same time. We will call this system a *private* voting system, because no voter can see what ballot choices the other voters make. Therefore, the only source of information for the manipulator, are interim scores in the voting process, presented to the voters when the voting result is a tie.

Let's first define a function that checks whether an unresolved result is a tie or not.

```
singleton :: UnresolvedResult -> Bool
singleton r = length r == 1
```

```
*VotingTheory> singleton [1,2,3]
 False

*VotingTheory> singleton [1]
 True
```

We add a function that returns the winning alternative, when the result is not a tie.

```
toResult :: UnresolvedResult -> Result
toResult = head
```

```
*VotingTheory> toResult [1,2,3]
 1
```

We also give a (very simple) type synonym for our manipulator.

```
type Manipulator = Voter
```

What follows, is the first user interface function that we described above. Examples of this function will be given in section 6. The function has two parts: a non-recursive part that functions as a wrapper (which can be roughly described as the introduction part to the function), and a recursive part that contains the rest of the functionality.

First we give the wrapper.

```
main :: Profile -> SCF -> Manipulator -> IO()
main p r m =
 if not $ elem m $ map fst p
 then error "Invalid manipulator."
 else do putStr ("\nLet’s start our private vote with " ++
          "profile: " ++ show p ++ ".\n\n(Press q to " ++
          "quit. Press any other button to continue).\n\n")
         l <- getLine
         case l of
            "q" -> putStr "\nHave a nice day.\n\n"
            _ -> let model = toModel p m
                         result = r (toProfile’ p)
                     in process (toProfile’’ p) r model
                         result m 1
```

We add the recursive function that implements the rest of the voting system.

```
process :: Profile'' -> SCF -> Model -> UnresolvedResult
                              -> Manipulator -> Int -> IO()
process p r m r' v n =
 do putStr ("There has been an election. The outcome is: "
                                    ++ show r' ++ ".\n\n")
    if singleton r'
     then putStr ("The outcome was not a tie. Therefore "
                                    ++ show (toResult r')
                       ++ " is the election winner.\n\n")
     else do putStr ("The outcome was a tie. We go to " ++
               "round " ++ show (n + 1) ++ " of the " ++
               "election and let the voters vote " ++
               "again.\n\n(Press q to quit. Press " ++
               "any other button to continue).\n\n")
             l <- getLine
             case l of
                  "q" -> putStr "\nHave a nice day.\n\n"
                  _ -> let p' = filterFromProfile'' r' p
                          m' = (\(ws,rw) ->
                           (filterFromWorlds r' ws,rw))
                            $ modelUpdate m r' r
                          manip = concatMap (\((v',b),_) ->
                            if v' == v
                            then [(v',b)]
                            else []) p'
                          p'' = profileUpdate m' p' v r
                          result = r (toProfile'
                           (toManipulableProfile'' p''))
                          p''' = if length p'' /= length p'
                                  then p'' ++ [(head manip,
                                    snd $ head manip)]
                                  else p''
                       in do putStr ("The profile has " ++
                                "changed to: " ++ show
                                (toManipulableProfile'' p'')
                                ++ ".\n\n")
                            l <- getLine
                            case l of
                               "q" -> putStr
                                  "\nHave a nice day.\n\n"
                               _ -> process p''' r m'
                                          result v (n + 1)
```

Next, we implement what we will call a *public* voting system, in which the voters cast their ballots one by one (according to a given voting order) and where they can observe each others actions. In this system, the manipulator cannot infer profile knowledge from interim scores. Instead, she has to use the direct knowledge obtained by her observation of the other voters. Examples of how to use this function are also given in section 6.

We start by giving a type synonym for a data structure that holds the profile knowledge of those voters who haven't voted yet. Because this structure is updated after each ballot cast by a certain voter i, it holds exactly the knowledge possessed by the voters that follow i in the voting order.

```
type ProfileConstraints = Profile'
```

Our next function is the function that does the updating of the structure described above.

```
updateConstraints :: ProfileConstraints -> Ballot
                                        -> ProfileConstraints
updateConstraints pc b =
 map (\e@(b',n) -> if b == b'
                  then (b',n+1)
                  else e) pc
```

We proceed by giving a function that uses the constraint structure to filter out those worlds that cannot be the real world anymore.

```
applyConstraints :: [World] -> ProfileConstraints -> [World]
applyConstraints ws pc =
 let unzippedWorlds = map unzip ws
     constraints = map (<=) $ map snd pc
 in concatMap (\(bs,ns) -> let eqs = zip constraints ns
                               possibleWorld = and
                                  $ map (\(f,x) -> f x) eqs
                           in if possibleWorld
                              then [zip bs ns]
                              else []) unzippedWorlds
```

We give the type synonym for a structure that holds the order in which the voters participating in the vote will cast their ballots.

```
type VotingOrder = [Voter]
```

Finally, we give the second user interface function that implements a public vote. This function has three parts: a wrapper part, a part in which the voters cast their ballots one by one and a part in which model and profile updates are done. The last two parts a *mutually recursive*, which means that part 2 is followed by part 3, which is followed by part 2, which is again followed by part 3, and so forth.

First we give part 1: the wrapper function.

```
main’ :: Profile -> VotingOrder -> SCF -> Manipulator
                                                    -> IO()
main’ p o r m =
 if sort (map fst p) /= sort o || not (elem m o)
 then error "Invalid voting order or manipulator."
 else do putStr ("\nLet’s start our public vote with " ++
                 "profile: " ++ show p ++
                 ".\n\n(Press q to quit. Press any " ++
                 "other button to continue).\n\n")
         l <- getLine
         case l of
                   "q" -> putStr "\nHave a nice day.\n\n"
                   _ -> let model = toModel p m
                                perms = permutations $ sort
                                               $ snd $ head p
                            in voteInOrder (toProfile’’ p)
                                (zip perms (cycle [0])) o o
                                r model (r (toProfile’ p)) m 1
```

We add part 2, in which the voters do their public voting.

```
voteInOrder :: Profile'' -> ProfileConstraints ->
 VotingOrder -> VotingOrder -> SCF -> Model ->
 UnresolvedResult -> Manipulator -> Int -> IO()
voteInOrder p'' c [] o r m r' v n = process' p'' o r m
 (r $ toProfile' $ toManipulableProfile'' p'') v n
voteInOrder p'' c (x : xs) o r (worlds,rw) r' v n =
 let updateBallot = (\(Just e) -> e) $ lookup x
                             $ toManipulableProfile'' p''
     c' = updateConstraints c updateBallot
     updatedWorlds = if elem v xs then applyConstraints
                                    worlds c' else worlds
 in voteInOrder p'' c' xs o r (updatedWorlds,rw) r' v n
```

Last but not least: part 3 of the function.

```
process' :: Profile'' -> VotingOrder -> SCF -> Model ->
 UnresolvedResult -> Manipulator -> Int -> IO()
process' p o r m r' v n =
 do putStr ("There has been an election. The outcome is: "
        ++ show r' ++ ".\n\n")
    if singleton r'
     then putStr ("The outcome was not a tie. Therefore "
        ++ show (toResult r')
        ++ " is the election winner.\n\n")
     else do putStr ("The outcome was a tie. We go to " ++
               " round " ++ show (n + 1) ++ " of the " ++
                "election and let the voters vote again"
                ++ ".\n\n(Press q to quit. Press any " ++
                "other button to continue).\n\n")
             l <- getLine
             case l of
               "q" -> putStr "\nHave a nice day.\n\n"
               _ -> let p' = filterFromProfile'' r' p
                        m' = (\(ws,rw) -> (filterFromWorlds
                                         r' ws,rw)) m
                        manip = concatMap (\((v',b),_) ->
                                         if v' == v
                                         then [(v',b)]
                                         else []) p'
                        p'' = profileUpdate m' p' v r
                        result = r (toProfile'
                            (toManipulableProfile'' p''))
                        p''' = if length p'' /= length p'
                                then p'' ++ [(head manip,
                                    snd $ head manip)]
                                else p''
                        perms = permutations $ sort $
                            snd $ head $
                                toManipulableProfile'' p'''
                    in do putStr ("The profile has " ++
                               "changed to: " ++ show
                               (toManipulableProfile'' p'')
                               ++ ".\n\n")
                          l <- getLine
                          case l of
                                 "q" -> putStr ("\nHave " ++
                                    "a nice day.\n\n")
                                 _ -> voteInOrder p''' (zip
                                    perms (cycle [0])) o o
                                    r m' result v (n + 1)
```

6 Program correctness

In section 5 we finished our Haskell program with two user interface functions that implement private and public voting systems. What we need to do next, is show that these functions actually work as they are supposed to. We will do this by performing several tests on our functions.

Let's start with a very simple test for our private vote function `main`, with the profile $\{\langle 1, \langle 1,2\rangle\rangle, \langle 2, \langle 1,2\rangle\rangle\}$, voter 1 as the manipulator and the borda rule for preference aggregation. Running the function with these parameters should result in a vote with just one round, since, according to the Borda rule, alternative 1 receives two points and thereby defeats alternative 2, which receives zero points.

Evaluating the function gives the following output:

```
*VotingTheory > main [(1,[1,2]),(2,[1,2])] bordaRule 1

Let's start our private vote with profile:
 [(1,[1,2]),(2,[1,2])].

(Press q to quit. Press any other button to continue).

There has been an election. The outcome is: [1].

The outcome was not a tie.
 Therefore 1 is the election winner.
```

With this simple input, the function clearly works.

Let's move on to a second test that is a little trickier. Instead of letting our two voters cast the same ballot, we now give them different ones. The profile changes to $\{\langle 1, \langle 1,2\rangle\rangle, \langle 2, \langle 2,1\rangle\rangle\}$. Again, we let voter 1 be the manipulator and use the borda rule for preference aggregation.

We evaluate the function again:

```
*VotingTheory> main [(1,[1,2]),(2,[2,1])] bordaRule 1

Let's start our private vote with profile:
 [(1,[1,2]),(2,[2,1])].

(Press q to quit. Press any other button to continue).
```

```
There has been an election. The outcome is: [1,2].

The outcome was a tie.
We go to round 2 of the election and let the voters vote
 again.

(Press q to quit. Press any other button to continue).

The profile has changed to: [(1,[1,2]),(2,[2,1])].

There has been an election. The outcome is: [1,2].

The outcome was a tie.
We go to round 3 of the election and let the voters vote
 again.

(Press q to quit. Press any other button to continue).

The profile has changed to: [(1,[1,2]),(2,[2,1])].

...
```

The three dots at the end were added to emphasize that the function has entered a loop. Is this behavior correct? If we take a closer look at the calculations made in the function, we conclude that the output is indeed correct.

First, we can easily see that applying the borda rule to the input profile results in a tie, since both alternatives receive one point.

Second, by using the definitions we gave in section 3, we can also see that it is optimal for the manipulator not to change her ballot in the course of the voting process. Remember that, for a voter i, the optimal result is the result with the highest payoff, where the payoff of a voting result X for i, who supports ballot B, is the number of possible voting results that are weakly dominated by X. Therefore, in our case, the final result {1,2} should weakly dominate the result that we would have had if the manipulator changed her ballot, wich would be {2}. And it does, because everything in {1,2} is at least as good as everything in {2}, and something in {1,2} is better than something in {2} (namely, 1 is better than 2).

A logical next step in our testing procedure would be to run the examples given above on our other user interface function, the one for public votes, and experiment with different voting orders. We will not do this, though, because the results look very much like those of the private vote function, as do the arguments for showing that the function behaves correctly.

Instead, we move on to a more interesting scenario, with the profile: $\{\langle 1, \langle 1,2,3\rangle\rangle, \langle 2, \langle 2,3,1\rangle\rangle, \langle 3, \langle 3,1,2\rangle\rangle\}$. This is actually an example we already discussed in subsection 2.4. Again, we let voter 1 be the manipulator and use the borda rule for preference aggregation.

Running the private vote function gives:

```
*VotingTheory> main [(1,[1,2,3]),(2,[2,3,1]),(3,[3,1,2])]
 bordaRule 1

Let's start our private vote with profile:
 [(1,[1,2,3]),(2,[2,3,1]),(3,[3,1,2])].

(Press q to quit. Press any other button to continue).

There has been an election. The outcome is: [1,2,3].

The outcome was a tie.
We go to round 2 of the election and let the voters vote
 again.

(Press q to quit. Press any other button to continue).

The profile has changed to:
 [(1,[2,1,3]),(2,[2,3,1]),(3,[3,1,2])].

There has been an election. The outcome is: [2].

The outcome was not a tie.
Therefore 2 is the election winner.
```

Finally, some real live manipulation action! Let's take a look at why it is correct.

Voter 1 chose $\langle 2,1,3\rangle$ for her manipulation, but she could also have chosen $\langle 2,3,1\rangle$, $\langle 3,2,1\rangle$, $\langle 3,1,2\rangle$ or $\langle 1,3,2\rangle$. Also, she could have chosen to keep to her actual preference order by continuing on with the ballot $\langle 1,2,3\rangle$, or to make a no-show move (that is: cast no ballot this round). Out of all this possibilities, is $\langle 2,1,3\rangle$ really the optimal manipulation ballot?

Before we can answer this question, we need to know what ballot choice would have resulted in what voting outcome. The following table tells us that:

Ballot	Profile	Result
$\langle 2,1,3\rangle$	$\{\langle 1,\langle 2,1,3\rangle\rangle,\langle 2,\langle 2,3,1\rangle\rangle,\langle 3,\langle 3,1,2\rangle\rangle\}$	{2}
$\langle 2,3,1\rangle$	$\{\langle 1,\langle 2,3,1\rangle\rangle,\langle 2,\langle 2,3,1\rangle\rangle,\langle 3,\langle 3,1,2\rangle\rangle\}$	{2,3}
$\langle 3,2,1\rangle$	$\{\langle 1,\langle 3,2,1\rangle\rangle,\langle 2,\langle 2,3,1\rangle\rangle,\langle 3,\langle 3,1,2\rangle\rangle\}$	{3}
$\langle 3,1,2\rangle$	$\{\langle 1,\langle 3,1,2\rangle\rangle,\langle 2,\langle 2,3,1\rangle\rangle,\langle 3,\langle 3,1,2\rangle\rangle\}$	{3}
$\langle 1,3,2\rangle$	$\{\langle 1,\langle 1,3,2\rangle\rangle,\langle 2,\langle 2,3,1\rangle\rangle,\langle 3,\langle 3,1,2\rangle\rangle\}$	{3}
$\langle 1,2,3\rangle$	$\{\langle 1,\langle 1,2,3\rangle\rangle,\langle 2,\langle 2,3,1\rangle\rangle,\langle 3,\langle 3,1,2\rangle\rangle\}$	{1,2,3}
no-show	$\{\langle 2,\langle 2,3,1\rangle\rangle,\langle 3,\langle 3,1,2\rangle\rangle\}$	{3}

According to the table, there are four different possible voting results: {2}, {2,3}, {3} and {1,2,3}. The payoffs for each result, with respect to the manipulator's actual preference order $\langle 1,2,3\rangle$, are displayed in a second table:

Result	Weakly dominates	Payoff
{2}	{2,3} and {3}	2
{2,3}	{3}	1
{3}		0
{1,2,3}	{3}	1

From this table, it is clear that the result {2} has the highest payoff. If we look back at the other table, we see that {2} was the outcome resulting only from the ballot choice $\langle 2,1,3\rangle$, which is exactly the ballot that the manipulator chose in our example. This shows, again, that our program works correcly.

The thing we haven't looked at yet, is the correctness of the knowledge aspects of our program. This is where our function for public voting comes in.

Suppose we do a public vote, again with the profile $\{\langle 1,\langle 1,2,3\rangle\rangle,\langle 2,\langle 2,3,1\rangle\rangle,\langle 3,\langle 3,1,2\rangle\rangle\}$ and the borda rule for preference aggregation. If we let the voting order be $\langle 1,2,3\rangle$, we can choose the manipulator to be either voter 1, 2 or 3. As we will see, how we rank the manipulator in the voting order has a profound effect on the manipulator's success in making a good decision. This is correct behavior, since the decision depends heavily on how much profile knowledge the manipulator has, which, in turn, depends on the ranking of the manipulator in the voting order.

Let's start by letting voter 1 be the manipulator. We now have a voting order, in which the manipulator casts her ballot first. This means that the manipulator has minimal profile knowledge to base her choice of ballot on, and therefore, her decision should be poor.

We check this by executing the function:

```
*VotingTheory> main' [(1,[1,2,3]),(2,[2,3,1]),(3,[3,1,2])]
 [1,2,3] bordaRule 1

Let's start our public vote with profile:
 [(1,[1,2,3]),(2,[2,3,1]),(3,[3,1,2])].

(Press q to quit. Press any other button to continue).

There has been an election. The outcome is: [1,2,3].

The outcome was a tie.
We go to  round 2 of the election and let the voters vote
 again.

(Press q to quit. Press any other button to continue).

The profile has changed to:
 [(1,[1,2,3]),(2,[2,3,1]),(3,[3,1,2])].

There has been an election. The outcome is: [1,2,3].

The outcome was a tie.
We go to  round 3 of the election and let the voters vote
 again.

(Press q to quit. Press any other button to continue).

The profile has changed to:
 [(1,[1,2,3]),(2,[2,3,1]),(3,[3,1,2])].

...
```

As we predicted, the manipulator makes a bad choice by casting a non-optimal ballot with payoff 1.

Next, we let voter 2 be the manipulator. Her ballot choice for manipulation should be better (or at least as good) now, since she can infer a substantial piece of profile knowledge from the choice made by voter 1.

Let's check this by executing our function:

```
*VotingTheory> main' [(1,[1,2,3]),(2,[2,3,1]),(3,[3,1,2])]
 [1,2,3] bordaRule 2

Let's start our public vote with profile:
 [(1,[1,2,3]),(2,[2,3,1]),(3,[3,1,2])].

(Press q to quit. Press any other button to continue).

There has been an election. The outcome is: [1,2,3].

The outcome was a tie.
We go to  round 2 of the election and let the voters vote
 again.

(Press q to quit. Press any other button to continue).

The profile has changed to:
 [(1,[1,2,3]),(2,[2,1,3]),(3,[3,1,2])].

There has been an election. The outcome is: [1].

The outcome was not a tie.
Therefore 1 is the election winner.
```

A somewhat surprising thing happens here: the manipulator swaps the last two alternatives in her preference order. Is this a good manipulation? Let's check this by setting up two tables like the ones we gave earlier.

Ballot	Profile	Result
$\langle 2,1,3\rangle$	$\{\langle 1,\langle 1,2,3\rangle\rangle,\langle 2,\langle 2,1,3\rangle\rangle,\langle 3,\langle 3,1,2\rangle\rangle\}$	{1}
$\langle 1,2,3\rangle$	$\{\langle 1,\langle 1,2,3\rangle\rangle,\langle 2,\langle 1,2,3\rangle\rangle,\langle 3,\langle 3,1,2\rangle\rangle\}$	{1}
$\langle 1,3,2\rangle$	$\{\langle 1,\langle 1,2,3\rangle\rangle,\langle 2,\langle 1,3,2\rangle\rangle,\langle 3,\langle 3,1,2\rangle\rangle\}$	{1}
$\langle 3,1,2\rangle$	$\{\langle 1,\langle 1,2,3\rangle\rangle,\langle 2,\langle 3,1,2\rangle\rangle,\langle 3,\langle 3,1,2\rangle\rangle\}$	{1,3}
$\langle 3,2,1\rangle$	$\{\langle 1,\langle 1,2,3\rangle\rangle,\langle 2,\langle 3,2,1\rangle\rangle,\langle 3,\langle 3,1,2\rangle\rangle\}$	{3}
$\langle 2,3,1\rangle$	$\{\langle 1,\langle 1,2,3\rangle\rangle,\langle 2,\langle 2,3,1\rangle\rangle,\langle 3,\langle 3,1,2\rangle\rangle\}$	{1,2,3}
no-show	$\{\langle 1,\langle 1,2,3\rangle\rangle,\langle 3,\langle 3,1,2\rangle\rangle\}$	{1}

Result	Weakly dominates	Payoff
{1}		0
{1,3}	{1}	1
{3}	{1,3} and {1}	2
{1,2,3}	{1}	1

The result is shocking: the manipulator chose a ballot that led to the least favorable outcome! How is this possible?

The answer to this question lies in the manipulator's profile knowledge. Apparently, the manipulator considered exactly those worlds possible that led him to this poor decision. We can show that the success of the manipulator actually depends on both the ranking of the manipulator in the voting order and the ranking of the other voters. For instance, if we run the voting process with the same profile, but with voter 3 as the manipulator and $\langle 1,3,2\rangle$ or $\langle 2,3,1\rangle$ as the voting order, the manipulator *does* perform a good manipulation! We leave it up to the reader to check this for him or herself.

Last but not least, we execute our public vote function another time, again with the voting order $\langle 1,2,3\rangle$, but now with voter 3 as the manipulator. Voter 3 should be able to make the optimal manipulation decision, since she is last in the voting order and therefore, when her turn to vote has come, knows exactly what the profile looks like.

Executing the function gives:

```
*VotingTheory> main' [(1,[1,2,3]),(2,[2,3,1]),(3,[3,1,2])]
 [1,2,3] bordaRule 3

Let's start our public vote with profile:
 [(1,[1,2,3]),(2,[2,3,1]),(3,[3,1,2])].

(Press q to quit. Press any other button to continue).

There has been an election. The outcome is: [1,2,3].
```

```
The outcome was a tie.
We go to  round 2 of the election and let the voters vote
 again.

(Press q to quit. Press any other button to continue).

The profile has changed to:
 [(1,[1,2,3]),(2,[2,3,1]),(3,[1,3,2])].

There has been an election. The outcome is: [1].

The outcome was not a tie.
Therefore 1 is the election winner.
```

Using the first set of tables we gave, we can conclude that the result of the function is, indeed, the optimal result.

We could proceed now by doing some more tests (with other voting rules, for example), but they would essentially be no more than variations on the examples we already gave. We leave it up to the reader to do further experimentation with the program.

7 Conclusion

In this thesis we introduced the reader to the most important concepts from voting theory and implemented these concepts in a Haskell program. We extended the field of voting theory with the notion of profile knowledge and implemented this notion using concepts from dynamic-epistemic logic and game theory. In doing so, we struggled with the size of our dynamic-epistemic models. We discovered that we could use a specialized profile representation, the ballot oriented profile, to keep the size of our knowledge models within bounds. We presented two main vote simulation functions, on which we performed several tests. From the results of these tests we concluded that our program works correctly. After having made this conclusion, we can say that our approach to the simulation of vote manipulation under partial order was a good one.

Implications for the scientific field of Artificial Intelligence

One of the characteristic aspects of human intelligence, is that we do not need perfect knowledge of things to reason about them. In this thesis, we presented a model which captures this kind of reasoning, and we showed that it works. Since Artificial Intelligence is concerned with modelling (aspects of) human intelligence, our model is important to the field. This thesis may serve as an inspiration and reference for other research that deals with reasoning under partial knowledge, especially in the field of Agent Technology, which is an important subdomain of Artificial Intelligence.

Ideas for further research

The program we developed in this thesis can model only a very limited number of voting scenarios. We could extend the program in several ways.

More social choice functions In this thesis we implemented only three social choice functions: the borda rule, the dictator rule and the plurality rule. Yet, there are a great deal of other social choice functions we could implement.

Other kinds of voting rules Social choice functions are not the only kind of voting rules used in voting theory. For example, we could also have a type of voting rule which maps profiles not onto a nonempty set of winning alternatives, but on a "collective ballot" (this type of voting rule is actually called a *social wellfare function*).

Other kinds of ballots In this thesis, ballots were presented as linear orders over a set of alternatives. In voting theory, however, other types of ballots (weak ordered sets, for example) are also used.

Coorporation amongst voters We could imagine voting scenarios in which voters do not operate alone, but form parties. Having one manipulative party of n voters, would then mean having n manipulators, which may require a whole new modelling approach.

Inclusion of false beliefs In our thesis, we assumed that everything the manipulator beliefs, is true. Allowing our manipulator to have false beliefs could be a possible extention of the program.

References

[1] J.v. Benthem, H.v. Ditmarsch, J.v. Eijck, and J. Jaspers. *Logic in Action*. Logic Education Group, Amsterdam, 2012.

[2] S. Brams and P. Fishburn. *Voting Procedures*. C.V. Starr Center for Applied Economics, New York, 1998.

[3] K. Doets and J. van Eijck. *The Haskell Road to Logic, Maths and Programming*. Kings College Publications, London, 2004.

[4] U. Endriss. *Computational Social Choice*. Course Notes, University of Amsterdam, 2011.

[5] U. Endriss. Logic and social choice theory. *Logic and Philosophy Today*, 2011.

[6] T. Haskell Team. The haskell homepage. http://www.haskell.org.

[7] A. Taylor. *Social Choice and the Mathematics of Manipulation*. Cambridge University Press, New York, 2009.