Multiagent Systems Assignment 1: Voting Manipulation

Group 11: Maxime Jakubowski, David Robinson, Kevin Trebing, Gwen van Vlokhoven

1 INTRODUCTION

Voting is an important aspect of group decision making. Different voting schemes try to implement a fair way to represent the general preference of a group. The quality metric for a voting outcome is measured in the overall happiness of a group.

In a perfect world all voting participants will give true true preference ordering of voting options. Even though it is their preference, the preference list could be altered to not maximise the general happiness of a group but the happiness for an individual participant. This is an instance of casting a *strategic vote* to manipulate the voting outcome.

In this report, we describe our implementation of the Tactical Voting Analyst. This program takes a preference matrix and a voting scheme as input. It then analyses the situation and provides tactical voting information.

2 IMPLEMENTATION

2.1 Software description

The program is written in `Python3.6` with the `numpy` library. Python was chosen as the implementation language for its high level syntax. The `numpy` library was used for the wide range of vector and matrix manipulation functions.

2.2 Running the software

In the file `votingManipulation.py` you enter your preference matrix as a numpy array in line 86. The rows are the voters and the columns are the voters preference. It is important to note this is inverted from the example for the sake of easier implementation. In line 93 you can choose what voting scheme should be used for evaluating a winner. If multiple voting schemes are given, each one will be evaluated one after the other. The output is written to a text file named after the scheme that was used. This is done due to the fact that if the output was printed out to the terminal it could happen that one could not see the whole output due to character history limitations.

2.3 Implementation Decisions

It is required to list **all** different possible voting preferences for a voter that has an impact on him instead of showing only the best voting preference. Because a voter would only change his true preference if his first choice would not win we only look at voters who are not completely happy. Therefore we use a brute-force algorithm to first list all possible preferences and then only output those where the happiness of the voter increases. This can lead to a very large output since there are maximally $c!$ possible different preferences for one voter. Because some terminals have character limitations we wanted to make sure that the full output is able to be investigated. That is the reason we created a global variable `OUTPUT` where all prints are put in and in the end written to a file named after the voting scheme used.

Concerning the risk of strategic voting we compute the number of strategic options divided by the number of voters. Since this value can have the upper bound $\frac{vc!}{v}$ it does not give an immediate insight into what it actually wants to convey. For example: This risk does not differentiate between one out of four voters having eight possible strategic voting options or four out of four voters each having two voting options; the value would be the same. It solely states that there are multiple options reduced by the number of voters.

That is the reason we introduced a more meaningful risk analysis indicator. This indicator is calculated by dividing the number of voters who have strategic options by the total number of voters. The resulting value is an indicator of the voting scheme being susceptible to strategic voting given the preference matrix. A value of 0.4 would then indicate that for this voting scheme 40% of the voters would strategically vote.

When looking for the type of strategic voting we only consider compromising and burying. Push-over is disregarded because it is only applicable to round-based voting schemes and we do not simulate this kind of voting.

Bullet voting is a manipulation technique that ignores all voting preferences except for the first one. The voter essentially only votes for one candidate and does not give any other preference information. In a normal setting, a voting scheme can be represented by a point vector which describes the importance of an option withing a preference list by assigning point values to them. Bullet voting manipulates this concept by only giving points to the first preference. We analysed bullet voting and saw that it does not have any influence on *Vote for One* as the only option that gets a vote is the first one. With all other considered voting schemes it does matter and it has quite a big influence on *Borda* and *Veto*. This is because as the point vector has more non-zero values, bullet voting has more influence as we ignore more of the points given to the options. Our implementation omits bullet voting.

2.4 Time complexity

In this task, four different voting schemes were used. Each of these had different time complexities for finding the results of a vote. As such, the overall time complexity of the algorithm depends on which voting scheme is selected. All of the complexities will be dependant on two variables. The number of voters, v, and the number of candidates, c.

The time complexity for finding a winner in the *Voting for One* and *Voting for Two* schemes is $O(v)$. These voting schemes only have to consider the first or first two preferences of each voter, so a constant number of operations will be performed for each vote, resulting in a time complexity of $O(v)$.

Proc. of the 18th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2019), N. Agmon, M. E. Taylor, E. Elkind, M. Veloso (eds.), May 2019, Montreal, Canada

https://doi.org/doi

In this implementation of veto, each candidate in a voter's preferences are considered except the last one. This means O(c) operations are performed for each voter, resulting in an overall time complexity of O(cv). It should be noted that the winner of a *Veto* election can be calculated in O(v) time. This could be achieved by counting only the last candidate in each voter's preferences and taking the winner as the one that appears the least. This approach was not used in this project as it does not have much effect on run-time for the size of preference matrix expected for this task and the overall complexity is dominated by the $c!$ term as discussed below. However, if this would be applied to a large scale vote, it could make a noticeable difference. One downside of this more efficient approach is it cannot handle bullet voting, which is not accounted for in this project, but may appear in a larger vote.

For *Borda* voting, every candidate must be considered for each voter as they all are assigned some amount of votes dependant on their position in the preference vector. This gives a time complexity of O(cv).

In this project, every possible ordering of candidates is considered for each voter. Each voter in this instance has complete knowledge and assumes all other voters will vote honestly. This means for each voter, O($c!$) preference orderings are tested to see if they change the result of an election to improve that voter's happiness. This means the overall time complexity will be O($c! \times vt$) where t is the time complexity of finding the winner using the selected voting scheme. This means the overall time complexity for each voting scheme is as follows:

- Voting for One: O($c! \times v^2$)
- Voting for Two: O($c! \times v^2$)
- Anti-plurality voting (veto): O($c! \times v^2 c$)
- Borda voting: O($c! \times v^2 c$)

3 EXPERIMENTS

We defined a preference matrix for every voting scheme and calculated by hand the winners and compared this with the calculated winners. This way we made sure that the calculations are correct. What we noticed was that when using the default matrix and using *Vote for Two* as voting scheme it is possible for Voter 2 to increase his own and the overall happiness. This could be due to this voting scheme not aiming at maximising the overall happiness. This lead us to investigate further which voting schemes try to maximise the overall happiness.

3.1 Overall happiness

It is interesting to see which voting schemes give the highest overall happiness for a given voting scheme. Preference matrices were generated for certain combinations of voter numbers (v) and number of voting options (c). Using these matrices we look at the happiness that the different voting schemes achieved and see which gave the highest happiness for each matrix. The percentage of matrices for which each scheme gave the highest happiness is shown in table 1. The average total happiness across the generated preference matrices is given for each voting scheme in table 2. Note that *Borda* also performs best in average happiness.

3.2 Overall risk

Another interesting thing to look into is how susceptible the different voting schemes are to voting manipulation. Here we will use the previously described function again and look at the risk statistics.

4 FINDINGS

4.1 Overall happiness experiment

Looking at table 1 we can see that the voting scheme *Borda* produced the highest amount of overall happiness for every matrix. Additionally we can see that the *Veto* voting scheme produces less overall happiness as the number of candidates increases. In the case that there are three candidates available, *Vote for Two* and *Veto* perform the same. Also visible in this table is the happiness in *Vote for Two* when only two candidates are available. This makes sense, considering every voter voters for every candidate, meaning it defaults to the alphabetical tie-breaking rule.

v	c	# matrices	VfO	VfT	Veto	Borda
5	4	98280	74.8%	76.9%	60.2%	100.0%
4	4	17550	77.0%	79.4%	61.0%	100.0%
4	3	126	88.1%	73.8%	73.8%	100.0%
3	4	2600	77.9%	74.0%	58.5%	100.0%
3	3	56	92.9%	71.4%	71.4%	100.0%
3	2	4	100.0%	50.0%	100.0%	100.0%
2	3	21	85.7%	85.7%	85.7%	100.0%
2	2	3	100.0%	66.7%	100.0%	100.0%

Table 1: Percentage of matrices for which voting schemes produced the highest overall happiness.

v	c	# matrices	max h	VfO	VfT	Veto	Borda
5	4	98280	15	10.2	10.3	9.6	10.7
4	4	17550	12	8.4	8.5	7.9	8.8
4	3	126	8	5.9	5.6	5.6	6.1
3	4	2600	9	6.5	6.5	6.0	6.9
3	3	56	6	4.6	4.3	4.3	4.7
3	2	4	3	2.5	1.5	2.5	2.5
2	3	21	4	3.1	3.0	3.0	3.3
2	2	3	2	1.7	1.0	1.7	1.7

Table 2: Average total happiness of voting schemes. Highest possible happiness (max h) is calculated by $v * (c - 1)$.

4.2 Overall risk experiment

The percentages denote the amount of matrices that were susceptible to manipulation by voters (so the risk of voting manipulation for these matrices is greater than 0). From table 3 we can derive that the strategy *Vote for One* has the least amount of matrices that can be manipulated by one voter. We can also see that *Veto* has the highest amount of matrices where manipulation is possible.

v	c	# matrices	VfO	VfT	Veto	Borda
5	4	98280	32.1%	41.1%	54.2%	46.8%
4	4	17550	28.4%	41.5%	54.4%	47.1%
4	3	126	11.9%	30.2%	30.2%	21.4%
3	4	2600	26.1%	39.5%	51.4%	51.0%
3	3	56	10.7%	30.4%	30.4%	23.2%
3	2	4	0%	0%	0%	0%
2	3	21	9.5%	28.6%	28.6%	33.3%
2	2	3	0%	0%	0%	0%

Table 3: Possibility of voter manipulation per scheme for different sizes of voters and candidates. The voting scheme *Vote for One* is the most resilient scheme to voting manipulation.

5 FUTURE WORK

Possible extensions to this project could be to extend the domain to include voter collusion or counter-strategic voting. These could be considered in the current environment or in one where voters do not have perfect information.

When considering voter collusion, two scenarios can be considered. One is which agents form coalitions before voting occurs, and the other is where all possible coalitions are considered. A voting strategy will be considered if it increases the happiness of at least one agent in the coalition without decreasing the happiness of any other agent that it is colluding with. In the first case, the program will have time complexity $O(\frac{v}{w} \times (c!)^w \times t)$ where w is the size of the largest coalition and the other variables are as previously defined in section 2.4. For the second scenario, the time complexity is shown in equation 1:

$$complexity = O(\sum_{x=1}^{v} \binom{v}{x} (c!)^x t) = O(t \times (c! + 1)^v) \quad (1)$$

Implementing a model to include voter collusion would be feasible in terms of difficulty but the time complexity is very high, so it may not be feasible to run using the same computational resources.

To consider counter-strategic voting, each voter must consider every possible combination of strategic votes that the other voters may make. This would mean considering $O((c!)^{v-1})$ different preference combinations which would take $O((c!t)^{v-1})$ time to generate, and the voter will then have to consider $O(c!)$ voting schemes for itself. The overall time complexity for this would then be:

$$complexity = O(v \times ((c!t)^{v-1} + (c!)^v \times t)) = O(v \times (c!t)^v) \quad (2)$$

This extension would be difficult to implement and would also require a lot of computational power to run due to it's large time complexity.

One other extension would be to have the program consider both coalitions and counter-strategic voting. For this, every voter must not consider only its own strategic voting options but also the voting options for every other voter. There are $O(v^v)$ possible sets of coalitions. For each of these there is a search space that is $O((c!)^v)$. This results in an overall time complexity:

$$complexity = O(v^v \times (c!)^v \times t) \quad (3)$$

This is harder to implement than just counter-strategic voting on its own, but has a similar time complexity.

In a domain where the voting agents do not know all the honest preferences of other voters, they must consider every possible preference order for each of the voters. In all cases, this will be more difficult to implement than the version with perfect information. In the case where both voter collusion and counter-strategic voting are considered, this is already done, so the time complexity will not change. In the case where only voter collusion is considered, the assumption will be made that voters in a coalition know each others' true preferences. This will result in a complexity of:

$$complexity = O(\sum_{x=1}^{v} \binom{v}{x} (c!)^v t) = O(2^v \times (c!)^v \times t) \quad (4)$$

Where counter-strategic voting is considered with incomplete information, the voter would have to consider the entire search space of all possible preference matrices. This will result in a time complexity:

$$complexity = O(v \times (c!t)^v) \quad (5)$$

This is the same time complexity as the case with complete information.