SVAMP: Simulator of Various Voting Algorithms in Manipulating Populations

François Durand
Inria
francois.durand@inria.fr

Fabien Mathieu
Alcatel-Lucent Bell Labs
France
fabien.mathieu@alcatel-lucent.com

Ludovic Noirie
Alcatel-Lucent Bell Labs
France
ludovic.noirie@alcatel-lucent.com

ABSTRACT

We present SVAMP, a Python package dedicated to the study of voting systems with an emphasis on manipulation analysis. Voters' preferences can be imported from external files or generated by a variety of probabilistic models. SVAMP currently implements more than 20 voting systems, and its object-oriented design facilitates the implementation of new voting systems. Special attention has been paid to Coalitional Manipulability (CM) and its variants. Algorithms for Condorcet efficiency, Individual Manipulability (IM) and Independence of Irrelevant Alternatives (IIA) are also implemented.

Categories and Subject Descriptors

J.4 [**Social and behavioral sciences**]: Economics

General Terms

Design, Documentation, Economics

Keywords

Voting, Manipulation, Simulation

1. INTRODUCTION

History of voting theory has been marked by the discovery of several paradoxes. Among the most famous are the Condorcet paradox or "paradox of voting" on majority pairwise comparisons [13], Arrow paradox, whose key features are transitivity and Independence of Irrelevant Alternatives (IIA) [3], and Gibbard–Satterthwaite impossibility theorem on manipulation [18, 37].

Since no reasonable voting system can avoid these paradoxes totally (e.g. only dictatorship escapes the Gibbard–Satterthwaite paradox), their likeliness of occurrence under various probability assumptions or in real-life elections has been studied at length [9, 10, 31, 26, 24, 2, 38, 28, 15, 34, 1, 36, 23, 35, 22, 41]. However, there remain open questions in the domain, especially about the relative performance of various voting systems according to different criteria and under different sets of assumptions on the preferences of the voters.

Recently, interesting results were published about algorithmic issues linked to voting systems and their manipulation: complexity results (mainly NP-hardness, following the line initiated by [5]) and explicit algorithms [44, 40, 46, 45, 17].

However, to the best of our knowledge, there was no publicly available software building on these existing techniques, in particular for the study of manipulability.

This observation led us to develop SVAMP (*Simulator of Various Voting Algorithms in Manipulating Populations*), a Python package designed to study voting systems and their manipulability. It is easy to use, supports a large choice of models for the preferences of the voters (or importation of real-life data) and a variety of voting systems. It is simple to extend SVAMP by adding new models of populations and new voting systems.

1.1 Availability

SVAMP is a free software, under GNU General Public License version 3. Its documentation includes installation procedure, tutorials, reference guide and instructions for new contributors. It is available at:

`https://svvamp.readthedocs.org`.

After providing a basic example to give a brief overview SVAMP's usage, we present the tools for the Population class, representing a set of voters with preferences over a set of candidates (section 2). Then we present the tools for the Election class, which is an abstract class whose all voting systems inherit (section 3). Finally, we illustrate the implementation choices made in SVAMP by focusing on the example of Coalitional Manipulability, whose study is one of the main purpose of the software (section 4).

1.2 Toy example

Let us give a brief overview of SVAMP's usage through a basic example.

After loading SVAMP, we define a random population of $V = 5$ voters with preferences over $C = 4$ candidates by using the *Spheroid* model, which is an extension of the usual *Impartial Culture* (see 2.3).

```
>>> import svvamp
>>> pop = svvamp.PopulationSpheroid(V=5, C=4)
```


TARK '15 Pittsburgh, Pennsylvania USA

This creates an object `pop` of class Population. We can then sort the voters by lexicographic order on their preference rankings and print these rankings.

```
>>> pop.ensure_voters_sorted_by_rk()
>>> print(pop.preferences_rk)
[[0 2 1 3]
 [1 3 2 0]
 [1 3 2 0]
 [2 0 1 3]
 [3 2 0 1]]
```

Each row of the matrix attribute `preferences_rk` represents the preference ranking of a voter. For example, the first voter prefers candidate 0, then 2, then 1, then 3. Note that following Python conventions, numbering of candidates and voters start from 0.

We can create an election with this population of voters, using the voting system *Plurality*[1], and determine the *sincere winner*, i.e. the candidate who wins the election when voters vote sincerely. This sincere winner, denoted w in this paper and in SVAMP as well, is candidate 1 in our example.

```
>>> election = svvamp.Plurality(pop)
>>> print(election.w)
1
```

We say that an election is *Coalitionaly Manipulable* (CM) if and only if (iff) there exists a candidate $c \neq w$ such that the voters who prefer c to w, by choosing appropriate strategies, can make c win the election (while other voters still vote sincerely). SVAMP can tell that the election is CM and give details.

```
>>> is_CM, log_CM, candidates_CM = election.
    CM_with_candidates()
>>> print(is_CM)
True
>>> print(candidates_CM)
[ 1.  0.  1.  0.]
```

Candidates 0 or 2 could win through CM, but not candidate 3 (candidate 1, i.e. w, is excluded by convention).

2. POPULATION

In this section, we present how to study the properties of a population by continuing our first basic example. Then, we show that SVAMP is not limited to preference rankings but can also exploit utilities and weak orders of preference. Finally, we present a variety of tools to define populations.

2.1 Get Information About a Population

An object of class Population has several functions and attributes that allow us to study its properties.

The *Borda scores*[2] given by voters to candidates are directly implemented as an attribute in the class Population, because they are used for several voting systems.

```
>>> print(pop.preferences_borda_rk)
[[3 1 2 0]
 [0 3 1 2]
 [0 3 1 2]
 [2 1 3 0]
 [1 0 2 3]]
```

[1]For the sake of conciseness, we will not give the definition of each voting system mentioned in this paper: see SVAMP's documentation for these definitions.

[2]The *Borda score* given by a voter v to a candidate c is the number of candidates d such that v ranks c before d.

The *total Borda score*[3] and the *Plurality score*[4] of each candidate are accessed the same way, using attributes `borda_score_c_rk` and `plurality_scores_rk` respectively.

Other attributes provide the *matrix of duels*[5], the *matrix of victories*[6] and the identity of the *Condorcet winner*[7].

```
>>> print(pop.matrix_duels_rk)
[[0 3 1 2]
 [2 0 2 4]
 [4 3 0 2]
 [3 1 3 0]]
>>> print(pop.matrix_victories_rk)
[[ 0.  1.  0.  0.]
 [ 0.  0.  0.  1.]
 [ 1.  1.  0.  0.]
 [ 1.  0.  1.  0.]]
>>> print(pop.condorcet_winner_rk)
nan
```

In this example, the last return value is `nan` (not a number), which is a convention meaning that there is no Condorcet winner.

The matrix of duels, the matrix of victories and the Condorcet winner have several variants implemented in SVAMP, depending on tie-breaking choices. The interested reader can refer to SVAMP's documentation for more details.

2.2 Work with Utilities

Most of the studies on the likeliness of manipulation, cited in the introduction, deal only with *ordinal* voting systems, which are based on strict or weak orders of preference. However, some interesting voting systems are *cardinal*, i.e. depending on grades. A particular case is *Approval voting*, which is an important object of research ([8, 16, 25] and many others). It is also used in real-life elections; for example, it has been chosen by social choice theorists for actual elections in the Society of Social Choice and Welfare.

One of SVAMP's objectives is to allow the comparison of ordinal and cardinal voting systems on the same populations of voters. For this purpose, voters do not have only preferences rankings, i.e. strict preference orders, but also utilities over the candidates. Hence we can define a population by providing utilities.

```
>>> pop1 = svvamp.Population(preferences_ut=
                             [[1,  1,  0],
                              [2,  0, -1]])
```

Utilities extend the set of possible preferences, compared to rankings. They provide a measure of intensity, and they allow a voter to be indifferent between several candidates, giving a convenient way to deal with weak order of preferences. In `pop1`, if candidate 0 is the sincere winner of an

[3]The *total Borda score* for a candidate c is the sum of her Borda scores given by all voters.

[4]The *Plurality score* of a candidate c is the number of voters who rank c first.

[5]For any pair of candidates (c, d), `matrix_duels_rk[c, d]` is the number of voters who rank c before d.

[6]For any pair of distinct candidates (c, d), `matrix_victories_rk[c, d]` is:

- 1 iff `matrix_duels_rk[c, d]` $> V/2$,
- 0.5 iff `matrix_duels_rk[c, d]` $= V/2$,
- 0 iff `matrix_duels_rk[c, d]` $< V/2$.

By convention, diagonal coefficients are equal to 0.

[7]A candidate c is *Condorcet winner* iff for any other candidate d, `matrix_victories_rk[c, d]` $= 1$.

election, then the first voter is not interested in a manipulation for candidate 1, and vice versa.

Note that for some voting systems, voters are compelled to provide a strict order of preference: in that case, each voter must break the ties in her own preferences. For this reason, if a population has been defined by its utilities only, then rankings are derived for each voters by breaking her personal ties randomly, once and for all, the first time the attribute `preferences_rk` is called.

```
>>> print(pop1.preferences_rk)
[[1 0 2]
 [0 1 2]]
```

The purpose of this implementation choice is to compare ranking-based voting systems on the sole basis of their specific behavior, without introducing a bias due to the fact that sincere voters may give different rankings for different voting systems.

If a population is defined by rankings only, SVAMP can populate utilities. Choosing utilities that are coherent with the rankings is necessarily arbitrary, so SVAMP sets by convention the utilities to the corresponding Borda scores.

```
>>> pop2 = svvamp.Population(preferences_rk=
                             [[0, 1, 2, 3],
                              [2, 0, 1, 3]])
>>> print(pop2.preferences_ut)
[[3 2 1 0]
 [2 1 3 0]]
```

It is also possible to provide utilities and rankings simultaneously, the latter indicating how each voter breaks her personal ties in her sincere ballot when a ranking-based voting system is used.

```
>>> pop3 = svvamp.Population(
            preferences_ut=[[1, 1,  0],
                            [2, 0, -1]],
            preferences_rk=[[0, 1, 2],
                            [0, 1, 2]])
```

Rankings must be coherent with utilities, in the sense that if a voter v put a candidate c before candidate d in her ranking, then her utility for c must be at least equal to her utility for d. If it is not the case, SVAMP raises an error.

Some attributes seen in section 2.1 differ if we consider voters' utilities or rankings. For example, `preferences_borda_ut` relies on utilities[8] and treat indifference as such. In contrast, the attribute `preferences_borda_rk` relies on rankings.

```
>>> print(pop3.preferences_borda_ut)
[[ 1.5  1.5  0. ]
 [ 2.   1.   0. ]]
>>> print(pop3.preferences_borda_rk)
[[2 1 0]
 [2 1 0]]
```

Similar variants with suffixes `_rk` (based on rankings) and `_ut` (based on utilities) are implemented for the matrix of duels, the matrix of victory and the Condorcet winner.

[8] For each voter v and candidate c, `preferences_borda_ut[v, c]` is the sum of:

- 1 point for each candidate d such that v has a strictly greater utility for c than for d;
- And 0.5 point for each candidate d such that v has the same utility for c and for d.

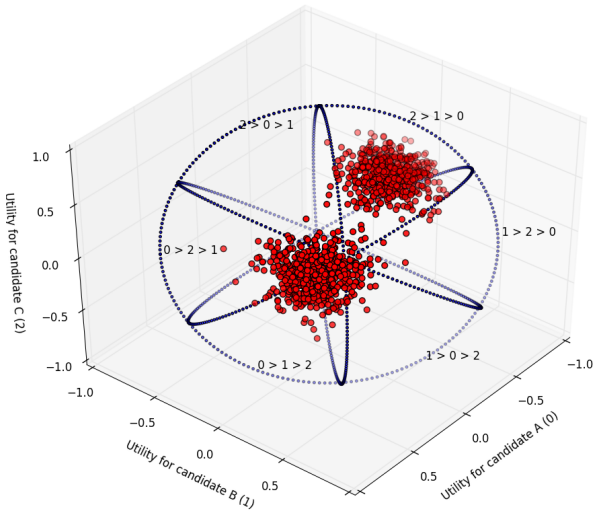

Figure 1: A population using Von Mises–Fisher model. $V = 1000$ voters, $C = 3$ candidates, two random poles of equal concentration 50.

2.3 Create a Population Object

Until now, toy example apart, we have defined populations manually, using the initialization function from class Population. This provides an easy way to define simple examples. In order to perform large scale studies, SVAMP also allows us to import a population from an external file, or to use a variety of probabilistic models to generate a random population.

Importing a population from an external file is straightforward: SVAMP can read simple CSV files containing the utilities of the population or files using the PrefLib format [30].

To generate artificial random populations, SVAMP implements a variety of probabilistic models (*cultures*).

Spheroid and *Cubic Uniform*: these two models extend the *Impartial Culture* (see for example [31]) to utilities. For rankings, both are equivalent to the *Impartial Culture*.

Ladder: like in *Impartial Culture*, voters are independent and the model is neutral, but a voter can be indifferent between several candidates.

Gaussian Well and *Euclidean box*: these two models attribute random positions to voters and candidates in a Euclidean space called the *political spectrum*. The utility of a voter for a candidate is a decreasing function of the distance between them. If the dimension of the political spectrum is 1, the preferences are necessarily *single-peaked* [7].

Von Mises–Fisher: this model is similar to Mallows' model [29], but it is adapted for populations defined by their utilities. There is a special point, the *pole*, where the density of probability is maximal, and a concentration parameter that allows SVAMP to control the shape of the distribution [42]. It is possible to instantiate a population constituted of several sub-groups using different poles with different concentrations. An example of such population is given in figure 1, generated by SVAMP. Each red dot represents the utility vector of a voter. Von-Mises Fisher profiles are generated with Ulrich's algorithm [39] modified by Wood [43].

3. ELECTIONS

In SVAMP, Election is an abstract class. An end-user always handles one of its subclasses implementing elections with a specific voting system (Plurality, Approval, etc.), inheriting the attributes and methods of class Election.

In this section, we will see how SVAMP allows us to study the result of an election, different notions of manipulation and Independence of Irrelevant Alternatives (IIA).

3.1 Result of the Election

Several attributes provide ways to study the result of the election when voters are sincere.

```
>>> pop = svvamp.PopulationSpheroid(V=3, C=5)
>>> print(pop.preferences_rk)
[[0 2 3 4 1]
 [0 2 4 3 1]
 [2 1 3 4 0]]
>>> election = svvamp.Plurality(pop)
>>> print(election.ballots)
[0 0 2]
>>> print(election.scores)
[2 0 1 0 0]
>>> print(election.w)
0
```

We get the sincere ballot of each voter, the score of each candidate and the winner of the election. Tie-breaking issues will be discussed in section 4.3. SVAMP also provides the attributes `candidates_by_scores_best_to_worst` and `scores_best_to_worst` to sort the candidates by their result in the election.

The type of attributes `ballots`, `scores`, and `scores_best_to_worst` depend on the voting system. For example, for Approval voting, `ballots` is a $V \times C$ matrix of booleans, and `ballots[v, c]` is True iff voter v approves candidate c.

See the documentation for an exhaustive list of attributes and more details on each voting system.

3.2 Coalitional Manipulation (CM)

To illustrate Coalitional Manipulability, consider the following example.

```
>>> pop = svvamp.PopulationSpheroid(V=1000,
                                    C=10)
>>> election = svvamp.IRV(pop)
```

IRV stands for *Instant-Runoff Voting*, also known as *Single Transferable Vote*, *Alternative vote* or *Hare method*[9]. Let us check the Coalitional Manipulability of this election.

```
>>> is_CM, log_CM = election.CM()
>>> print(is_CM)
nan
```

For each function, SVAMP uses by default its most precise algorithm among those running in polynomial time. For IRV, deciding CM is NP-complete [5], so this polynomial algorithm is not exact. For that reason, `is_CM` can be a boolean (whether the election is manipulable or not), or the conventional value `nan` meaning that the algorithm was not able to decide.

[9] *Single Transferable Vote* can lead to confusion with its version with multiple winners, whereas to the best of our knowledge, the terminology *Instant-Runoff Voting* is only used for the version with a single winner. This is why the latter has been preferred in SVAMP.

We can choose an exact algorithm instead by modifying the attribute `CM_option`.

```
>>> election.CM_option = 'exact'
>>> is_CM, log_CM = election.CM()
>>> print(is_CM)
True
```

For voting systems where there is an exact algorithm running in polynomial time, `'exact'` is the only available option.

As seen in the introduction, SVAMP can precise for which candidates the election is manipulable.

```
>>> is_CM, log_CM, candidates_CM = election.
    CM_with_candidates()
>>> print(candidates_CM)
[ 0.  0.  0.  1.  0.  0.  0.  0.  0.  0.]
```

3.3 Variants of Coalitional Manipulation

SVAMP also implements three additional variants of CM.

We say that an election is *Ignorant-Coalition Manipulable* (ICM) in favor of candidate $c \neq w$ iff voters who prefer c to the sincere winner w can use strategies such that, whatever strategies are chosen by the other voters, candidate c is declared the winner.

We say that an election is *Unison-Manipulable* (UM) in favor of candidate c iff there exists a strategy such that, when all voters who strictly prefer c to the sincere winner w use this same one strategy (while other voters still vote sincerely), candidate c is declared the winner. The terminology "unison" is borrowed from [40].

Lastly, we say that an election is *Trivially Manipulable* (TM) in favor of candidate $c \neq w$ iff, when all voters who strictly prefer c to w use their *trivial strategies* (while other voters still vote sincerely), candidate c is declared the winner.

What we call *trivial strategy* for voter v in favor of candidate c against candidate w depends on the ballot type. For ordinal voting systems, it is the ballot where c is ranked first ("compromising"), w is ranked last ("burying"), and the other candidates are kept in v's sincere binary relation of preference. For grade-based voting systems, it is the ballot where v attributes the highest allowed grade to c and the lowest allowed grade to other candidates.

Trivial strategies are natural when voter v wants to make c win and when she knows that w is a strong opponent, without any clue about the odds of other candidates. They require little coordination between manipulators. As such, when CM is possible, TM is an indicator of whether manipulation is easy or not, in terms of exchange of information between the manipulators. The other obvious advantage of this notion is that it can be computed in polynomial time (provided computing the winner of an election is polynomial, which is generally the case, except for voting systems like *Kemeny*).

These three variants of CM — ICM, UM and TM — use same syntax than CM. For example, UM can be decided with functions `UM` and `UM_with_candidates` and its algorithm can be chosen with `UM_option`. Like for CM, their default option is to use the most precise polynomial algorithm available.

3.4 Individual Manipulation

We say that an election is *Individually Manipulable* (IM) iff a voter v, by casting an insincere ballot, can secure an outcome c that she strictly prefers to the sincere winner w (while other voters still vote sincerely).

The function providing most information about IM is `IM_full`.

```
>>> pop = svvamp.PopulationSpheroid(V=4, C=3)
>>> election = svvamp.IRV(pop)
>>> election.IM_option = 'exact'
>>> is_IM, log_IM, candidates_IM, voters_IM,
    v_IM_for_c = election.IM_full()
>>> print(is_IM)
True
>>> print(candidates_IM)
[ 1.  0.  0.]
>>> print(voters_IM)
[ 0.  0.  1.  1.]
>>> print(v_IM_for_c)
[[ 0.  0.  0.]
 [ 0.  0.  0.]
 [ 1.  0.  0.]
 [ 1.  0.  0.]]
```

In this example, the election is IM. More precisely, only the first candidate can benefit from IM. This manipulation can be performed by any of the two last voters, whereas the two first ones cannot or are not interested.

If only a subset of these informations is required, dedicated functions are provided. They run only the necessary computations, which makes them faster than `IM_full`.

3.5 Independence of Irrelevant Alternatives

The election is *Independent of Irrelevant Alternatives* (IIA) iff when running the election with the same voters and with any subset of candidates including the sincere winner w, she remains the winner.

Here we study IIA for a specific instance of election with a given population, not IIA as a property for a voting system in general.

```
>>> not_IIA, log_IIA, example_subset_IIA,
    example_winner_IIA = election.not_IIA_full
    ()
>>> print(not_IIA)
True
>>> print(example_subset_IIA)
[ True  True False]
>>> print(example_winner_IIA)
0
```

`not_IIA` is True, which means that the election violates IIA (by coherence with the manipulation methods like CM, True stands for the "'bad" behavior). When it is the case, `example_subset_IIA` provides a subset of candidates violating IIA: here, the subset with candidates 0 and 1. `example_winner_IIA` provides the corresponding winner, candidate 0 in this example.

If the election does not violate IIA, the counter-example variables return the conventional value `nan`.

3.6 Properties of a Voting System

Voting systems come with special attributes that represent a variety of properties. The end-user can access these attributes via any Election object, but: they concern the voting system itself, not a specific instance of election; they are mostly useful for developers of new voting systems, who can fill some attributes to help SVAMP speed up computations (cf section 4.4).

For example, the following attribute means that IRV is based on strict rankings, not on utilities.

```
>>> election = svvamp.IRV(pop)
>>> print(election.is_based_on_rk)
True
```

The next attribute means that the voting system is based on utilities and that, for a single manipulator or a coalition of manipulators for candidate c, it is an optimal strategy to act as if they have utility 1 for c and -1 for all other candidates. For example, it is the case for Approval voting or Range voting (in their default configuration). Of course, this attribute is False for IRV.

```
>>> print(election.is_based_on_ut_minus1_1)
False
```

The four next criteria are closely related together and are used extensively for the computation of coalitional manipulation and its variants.

We say that a voting system meets the *Condorcet criterion* (**Cond**) iff, when a candidate w is a Condorcet winner and voters cast sincere ballots, then w is elected.

We say that a candidate w is a *majority favorite* iff more than $V/2$ voters prefer w to all other candidates. We say that a voting system meets the *majority favorite criterion* (**MajFav**) iff, when a candidate w appears as a majority favorite, then she is elected.

We say that a voting system meets the *ignorant majority coalition criterion* (**IgnMC**) iff for any candidate c, any subset of candidates with more than $V/2$ voters can choose strategies so that c is elected, whatever the other voters do.

We say that a voting system meets the *informed majority coalition criterion* (**InfMC**) iff for any candidate c, any subset of candidate with more than $V/2$ voters, knowing the strategies of other voters, can choose strategies so that c is elected.

```
>>> print(election.meets_Condorcet_c_rk)
False
>>> print(election.meets_majority_favorite_c_rk
    )
True
>>> print(election.meets_IgnMC_c)
True
>>> print(election.meets_InfMC_c)
True
```

Several variants of these criteria exist, depending on the usual distinction between rankings and utilities, and on notions of tie-breaking between the candidates. See the documentation for more details.

It is easy to prove that **Cond** implies **MajFav**, which implies **IgnMC**, which implies **InfMC**. SVAMP manages the implications between these criteria: for example, a developer can only inform SVAMP that a voting system meets a specific variant of **MajFav**. SVAMP knows what other variants of **MajFav** and what declinations of **IgnMC** and **InfMC** it implies.

4. IMPLEMENTATION OF CM

In this section, we will focus on the implementation of CM, which illustrates best the general techniques used in SVAMP.

4.1 Minimizing computation

When computing CM and variants, if the election is not manipulable, it is generally necessary to loop on all candidates to prove it. But if the election is manipulable, it is sufficient to prove manipulation in favor of one candidate.

For this reason, it is interesting to guess for which candidates a manipulation is more likely, and to test these candidates first. In the generic methods of the class Election, we use a simple heuristic: candidates are examined in decreasing order of the number of corresponding manipulators. This behavior can easily be overridden when implementing a specific voting system.

Generally speaking, SVAMP aims at being as lazy as possible. This means for instance: 1. not doing useless computation and 2. not doing the same computation twice.

This can be illustrated over the following example.

```
>>> is_CM, log_CM = election.CM()
>>> print(is_CM)
True
>>> is_CM, log_CM, candidates_CM = election.
    CM_with_candidates()
>>> print(candidates_CM)
[0. 0. 1. 0. 1.]
```

1. When `CM` is called, a possible execution is: SVAMP examines candidate 0 first, because she has more manipulators, but manipulation is proven impossible. Then SVAMP examines candidate 2, and finds a manipulation. SVAMP stops computation and decides `is_CM` to True.

2. When `CM_with_candidates` is called, SVAMP reminds the computation for candidates 0 and 2, and only runs the computation for the other candidates.

4.2 Number of Manipulators

When computing UM and TM for a given candidate c, a fixed set of manipulators is used: the voters who actually prefer c to w. However, CM and ICM work differently. In fact, to study CM or ICM in favor of candidate c, SVAMP deals with the following question. Given the sub-population of sincere voters, i.e those who do not strictly prefer c to w, what minimal number of manipulators must be added to perform manipulation[10]? Let us note this integer x_c.

During the computation, SVAMP maintains a lower bound and an upper bound of x_c. As soon as the lower bound becomes greater than the number of manipulators, manipulation is proven impossible. Similarly, as soon as the upper bound becomes lower or equal to the number of manipulators, manipulation for c is proven.

This mechanism allows SVAMP to combine different algorithms and to accelerate computation, even when `CM_option` is `'exact'`. For example, using IRV in exact mode, SVAMP starts by using an approximate polynomial algorithm to diminish the interval between the bounds. If it suffices to decide CM, computation stops. Otherwise, the exact non-polynomial algorithm is used, which exploits the bounds already known to accelerate computation.

4.3 Anonymity and Resoluteness

The mechanism of adding manipulators we have just seen explains an important choice in SVAMP: voting systems must be *anonymous*, in the sense that they treat all voters equally. This choice allows SVAMP to work on the number of manipulators, not on their identity. Would voters have different weights, many problems would become difficult: for example, even for a fixed number of 3 candidates, deciding CM with weighted voters for Borda, Maximin or IRV is NP-complete [11], whereas with unweighted voters and a fixed number of candidates, it is in P.

We also assume that voting systems are deterministic and *resolute*: they elect a single candidate. Otherwise, defining manipulation is an issue by itself, because we need to tell how preferences over candidates extend to preferences over sets of candidates (or probability distributions over the candidates), which can be done in a variety of ways [20, 19, 14]. Resolute voting systems allow SVAMP to decide CM and variants by looping on the candidates who may benefit from manipulation, considering for each one the set of voters interested by the candidate under study. In contrast, if the voting system returns a subset of candidates (resp. a probability distribution over the candidates), then it is not reasonable (resp. not possible) to loop over all potential outcomes.

Since the voting systems are anonymous and resolute, they are necessarily not neutral: indeed, they need to return a single winner even in totally symmetric situations. In SVAMP, the usual tie-breaking rule is that candidates with lower index are favored. For example, if there is a tie between candidates 0 and 1, then candidate 0 is declared the winner. We chose this rule for its simplicity: with more intricate tie-breaking rules, manipulation problems usually in P can become NP-complete [33, 32, 4]. However, there is no architectural limitation in SVAMP preventing to implement voting systems with other tie-breaking rules.

4.4 CM sub-functions

SVAMP is written in a modular way. For instance, the function `CM` (or the detailed version `CM_with_candidates`) defined in class Election actually calls a set of specific sub-functions. Each of these sub-functions can be overridden in the subclass implementing a specific voting system, while keeping the others.

Broadly speaking, the sub-functions belong to one of three categories: general preliminary checks, candidate-specific preliminary checks, core computation.

The general preliminary checks perform a series of tests aiming at deciding CM immediately.

- If the voting system meets **MajFav** (cf 3.6) and if w is a majority favorite[11], then CM is False.
- If the voting system meets **Cond** and if w is *resistant-Condorcet winner*[12], then CM is False.
- If the voting system meets **InfMC** and w is not *Condorcet-admissible*[13], then CM is True.

[10] This is called the *Constructive Coalitional Unweighted Optimization* problem [46], which allows SVAMP to decide CM for all voting systems currently implemented.

[11] For all preliminary checks, there is a variety of variants depending on notions of tie-breaking. For the sake of conciseness and clarity, we do not develop these variants in this paper. For more details, see the code of class Election.

[12] A candidate w is a *resistant Condorcet winner* iff for any other candidate c, voters who strictly prefer c to w (in the sense of utilities) cannot prevent w from appearing as a Condorcet winner. Cf. SVAMP's documentation for more details.

[13] A candidate w is *Condorcet-admissible* iff for any candidate c, there is not a strict majority of voters who strictly prefer c to w (in the sense of utilities). When voters have no ties in their preferences, this notion is equivalent to the

- If either TM, UM or ICM is True, then CM is True.

If necessary, SVAMP performs preliminary checks that are specifically dedicated to the manipulation in favor of a given candidate c.

- If the voting system meets **InfMC**, then it is sufficient to have more manipulators than sincere voters.
- If the voting system meets **MajFav**, then it is necessary to have enough manipulators to prevent the sincere winner w from appearing as a majority favorite.
- If the voting system meets **Cond**, then it is necessary to have enough manipulators to prevent w from appearing as a Condorcet winner.
- If TM or UM is possible for c, then CM is possible.
- The sufficient number of manipulators for ICM is also a sufficient number of manipulators for CM.

At that point, if `CM_option` is set to `'lazy'`, the algorithm ends whether the solution has been found or not. In the latter case, it returns the conventional `nan`.

Otherwise, in the default settings of Election class, it will use an exact algorithm.

If the attribute `is_based_on_rk` is True, the default exact algorithm of the generic Election class uses brute force: manipulators try all the possible ballots. For candidate c, denoting n_c the number of manipulators for c against w, this default algorithm has a time complexity of order $(C!)^{n_c}$ (multiplied by a polynomial and by the time needed to computethe winner of an election), which makes it only usable for small populations.

If the attribute `is_based_on_ut_minus1_1` is True, the default exact algorithm is straightforward: manipulators just need to test the sincere ballot corresponding to an utility 1 for c and an utility -1 for all other candidates. This runs in polynomial time (multiplied by the time needed to compute the winner of an election).

In other cases, this method raises a NotImplemented error.

In the subclass implementing a specific voting system, the sub-function implementing the exact algorithm is frequently overridden to the benefit or a more efficient algorithm.

4.5 Performance of the generic polynomial algorithm

Using the `'lazy'` option, the generic algorithm for CM runs in polynomial time but may not decide. Figure 2 gives its efficiency measured on voting systems where no specific algorithm is implemented in SVAMP. For these voting systems, only the rule to compute the winner is provided to SVAMP, as well as the basic properties of the voting system (`is_based_on_rk`, `meets_Condorcet_c`, etc.). Therefore, the generic `'lazy'` algorithm relies only on the preliminary checks exposed in 4.4, the most important of them being TM.

To get Figure 2, for each value of C, 10 000 random populations were generated by the Spheroid model with $V = 33$. The value on y-axis is the ratio of population where the function `CM` returned `nan`, meaning that it was not able to prove CM or ¬CM.

usual notion of *weak Condorcet winner*. Cf. SVAMP's documentation for more details.

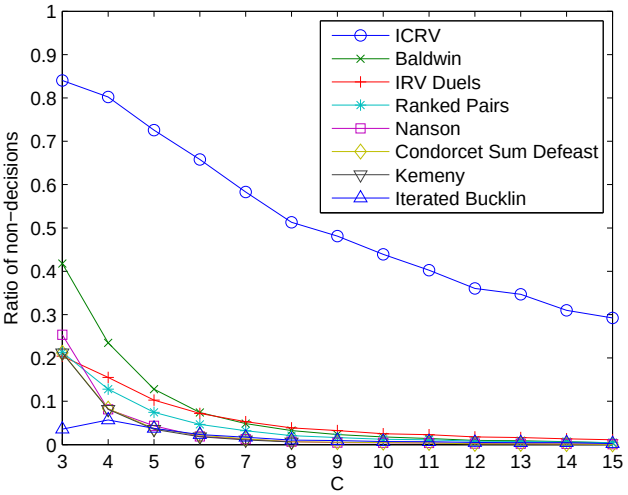

Figure 2: Performances of the default polynomial algorithm for CM. Spheroid model, $V = 33$.

We see that the generic lazy algorithm is quite efficient to decide CM, except for the voting system *ICRV* (one of the Condorcet variants of IRV). Empirically, when comparing with the exact algorithm, we noticed that the lazy one is efficient to prove CM, but bad to prove ¬CM. In the simulations leading to this figure, except for ICRV, the frequencies of CM were high, which explains the good performances for other voting systems.

4.6 Algorithms for specific voting systems

For some voting systems, specific algorithms are implemented. In this section, we give an overview of these algorithms.

For Range Voting, Majority Judgment, Approval voting, Bucklin rule, Plurality, Two-Round System and Veto, CM is computed exactly in polynomial time. For the Bucklin rule, we use an algorithm from [44].

For the Schulze Method, we use an algorithm from [17], running in polynomial time. The original article proves that this algorithm is exact for the multiple winner problem, or the single winner problem. In SVAMP, it has a window of error of 1 manipulator due to the tie-breaking rule.

For the Borda Rule, deciding CM is NP-complete, even when there are only 2 manipulators [6, 12]. We use an approximation algorithm from [46], which has a window of error of 1 manipulator.

For Maximin, deciding CM is NP-complete, even when there are only 2 manipulators [44]. We use an approximation algorithm from [45], which has a multiplicative factor of error of $\frac{5}{3}$ on the number of manipulators needed.

For the Coombs Method, there are two possible options for CM, `'fast'` and `'exact'`. In exact mode, our algorithm is similar to the one used in [40] for IRV. Its time complexity is of order $C!$. In fast mode, we use an original heuristic (out of the scope of this article).

IRV was one of the first voting systems for which deciding IM (and *a fortiori* CM) was proven NP-complete [5]. Furthermore, it seems to be especially resilient against manipulation [10, 5, 27, 28, 21, 22]. For these reasons, we paid a special attention to IRV methods and implemented specific algorithms whose performance is briefly described below.

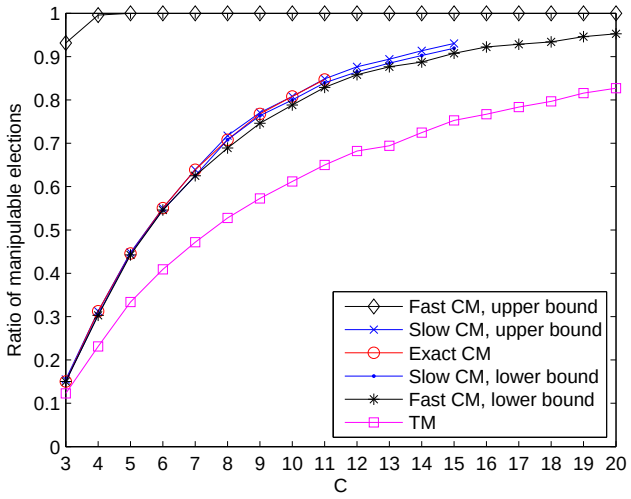

Figure 3: Precision of algorithms for IRV. Spheroid model (Impartial Culture), $V = 33$.

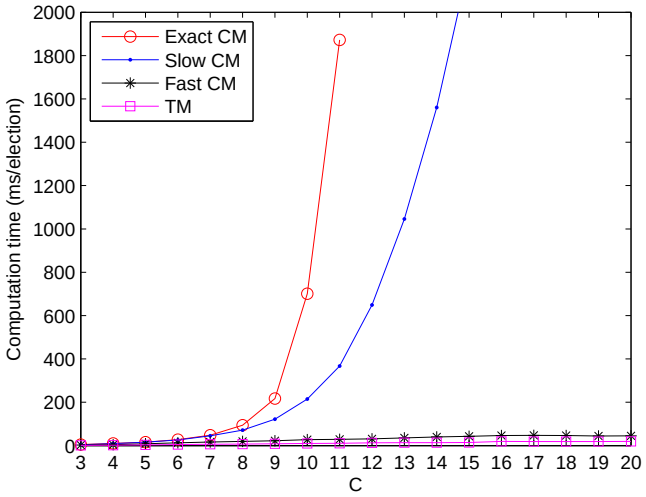

Figure 4: Computation time of algorithms for IRV. Spheroid model (Impartial Culture), $V = 33$.

4.7 Performance of IRV algorithms

In SVAMP, the implementation of CM for IRV is closely linked to Exhaustive Ballot (a voting system similar to IRV but with actual rounds) and Condorcet-IRV (one of the Condorcet variants of IRV). In addition to an `'exact'` algorithm, which uses non-polynomial algorithms adapted from [40], we propose two original heuristics accessed through `'fast'` and `'slow'` options.

Describing how `'fast'` and `'slow'` are implemented under the hood is out of the scope of this article, but their main features are that: `'fast'` is a polynomial algorithm designed to be efficient at finding manipulation if they exist; `'slow'` is a non-polynomial algorithm that aims at deciding CM in most cases while running significantly faster than the exact algorithm derived from [40].

This is illustrated by Figures 3 and 4, which study how the number of candidates impact the performance. For each figure, the results are obtained by averaging 10,000 random instances. Figure 3 indicates the accuracy of non-exact algorithms by giving their lower bound for manipulability (the algorithms decided CM) and their upper bound (the algorithm decided CM or stopped before decision). For parameters where the `'exact'` algorithm can run, we see that the `'fast'` option manages to detect most of the manipulable elections (Fast CM, lower bound). To compare with, simply running the trivial manipulation (TM) fails to detect a significant part of these. The main drawback of the `'fast'` option is that it has great difficulty in proving the absence of manipulation (Fast CM, upper bound): even if empirically most of the cases where it does not decide corresponds to non-manipulable elections, no guarantee is provided. This justified the introduction of the `'slow'` option, which decides CM with high accuracy (Slow CM bounds in the Figure) at the price of an increased time complexity.

The computation time of the algorithms is given by Figure 4. It has been measured on a regular personal laptop used for benchmarking. We see that `'fast'` has a running time greater than TM but stays very efficient with less than 50ms for all the considered parameters. On the other hand, the non-polynomial property of `'slow'` and `'exact'` is clearly confirmed, but we can observe that `'slow'` is significantly faster than `'exact'`, allowing us to explore a larger set of parameters for the same time budget.

5. CONCLUSION

We have presented SVAMP, a flexible simulator dedicated to the study of voting systems and their manipulation, under various assumptions on the preferences of voters. Populations of voters can be imported from external files or generated via several random models. They can be tested on elections with more than 20 voting systems, with a variety of manipulation criteria.

While state-of-the-art algorithms are implemented for many of these systems, the generic methods defined in SVAMP allow developers to quickly define a new voting system, only by its rule, and already benefit from generic manipulation algorithms, which makes SVAMP easily extensible. We hope that it will be useful to researchers, teachers and students interested in voting theory.

6. REFERENCES

[1] F. Aleskerov, D. Karabekyan, R. Sanver, and V. Yakuba. Computing the degree of manipulability in the case of multiple choice. *on Computational Social Choice (COMSOC-2008)*, page 27, 2008.

[2] F. Aleskerov and E. Kurbanov. Degree of manipulability of social choice procedures. In S. for the Advancement of Economic Theory. International Meeting, A. Alkan, C. Aliprantis, and N. Yannelis, editors, *Current trends in economics: theory and applications*, Studies in economic theory, pages 13–27. Springer, 1999.

[3] K. J. Arrow. A difficulty in the concept of social welfare. *The Journal of Political Economy*, 58(4):pp. 328–346, 1950.

[4] H. Aziz, S. Gaspers, N. Mattei, N. Narodytska, and T. Walsh. Ties matter: Complexity of manipulation when tie-breaking with a random vote. In *AAAI*, 2013.

[5] J. J. Bartholdi and J. B. Orlin. Single transferable vote resists strategic voting. *Social Choice and Welfare*, 8:341–354, 1991.

[6] N. Betzler, R. Niedermeier, and G. Woeginger. Unweighted coalitional manipulation under the borda rule is np-hard. In *Proceedings of the 22th International Joint Conference on Artificial Intelligence (IJCAI '11)*, 2011.

[7] D. Black. *The theory of committees and elections*. University Press, 1958.

[8] S. Brams and P. Fishburn. Approval voting. *American Political Science Review*, 72:831–847, 1978.

[9] C. D. Campbell and G. Tullock. A measure of the importance of cyclical majorities. *The Economic Journal*, 75(300):pp. 853–857, 1965.

[10] J. R. Chamberlin, J. L. Cohen, and C. H. Coombs. Social choice observed: Five presidential elections of the american psychological association. *The Journal of Politics*, 46:479–502, 1984.

[11] V. Conitzer and T. Sandholm. Complexity of manipulating elections with few candidates. In *Eighteenth national conference on Artificial intelligence*, pages 314–319. American Association for Artificial Intelligence, 2002.

[12] J. Davies, G. Katsirelos, N. Narodytska, and T. Walsh. Complexity of and algorithms for borda manipulation. *CoRR*, abs/1105.5667, 2011.

[13] J. de Caritat marquis de Condorcet. *Essai sur l'application de l'analyse à la probabilité des décisions rendues à la pluralité des voix*. Landmarks of science. De l'Imprimerie royale, 1785.

[14] J. Duggan and T. Schwartz. Strategic manipulability without resoluteness or shared beliefs: Gibbard-satterthwaite generalized. *Social Choice and Welfare*, 17:85–93, 2000.

[15] P. Favardin and D. Lepelley. Some further results on the manipulability of social choice rules. 26:485–509, 2006.

[16] P. C. Fishburn and S. J. Brams. Efficacy, power and equity under approval voting. *Public Choice*, 37(3):425–434, 1981.

[17] S. Gaspers, T. Kalinowski, N. Narodytska, and T. Walsh. Coalitional manipulation for schulze's rule. In *Proceedings of the 2013 international conference on Autonomous agents and multi-agent systems*, pages 431–438. International Foundation for Autonomous Agents and Multiagent Systems, 2013.

[18] A. Gibbard. Manipulation of voting schemes: A general result. *Econometrica*, 41(4):pp. 587–601, 1973.

[19] A. Gibbard. Straightforwardness of game forms with lotteries as outcomes. *Econometrica*, 46(3):pp. 595–614, 1978.

[20] P. Gärdenfors. Manipulation of social choice functions. *Journal of Economic Theory*, 13:217–228, 1976.

[21] J. Green-Armytage. Four condorcet-hare hybrid methods for single-winner elections. *Voting matters*, 29:1–14, 2011.

[22] J. Green-Armytage. Strategic voting and nomination. *Social Choice and Welfare*, 42(1):111–138, 2014.

[23] A. Jennings. *Monotonicity and Manipulability of Ordinal and Cardinal Social Choice Functions*. BiblioBazaar, 2011.

[24] J. S. Kelly. Almost all social choice rules are highly manipulable, but a few aren't. *Social Choice and Welfare*, 10:161–175, 1993.

[25] J.-F. Laslier. The leader rule: A model of strategic approval voting in a large electorate. *Journal of Theoretical Politics*, 21(1):113–136, 2009.

[26] D. Lepelley and B. Mbih. The proportion of coalitionally unstable situations under the plurality rule. *Economics Letters*, 24(4):311 – 315, 1987.

[27] D. Lepelley and B. Mbih. The vulnerability of four social choice functions to coalitional manipulation of preferences. *Social Choice and Welfare*, 11:253–265, 1994.

[28] D. Lepelley and F. Valognes. Voting rules, manipulability and social homogeneity. *Public Choice*, 116:165–184, 2003.

[29] C. L. Mallows. Non-null ranking models. i. *Biometrika*, pages 114–130, 1957.

[30] N. Mattei and T. Walsh. Preflib: A library of preference data. In *Proceedings of Third International Conference on Algorithmic Decision Theory (ADT 2013)*, Lecture Notes in Artificial Intelligence. Springer, 2013.

[31] S. Nitzan. The vulnerability of point-voting schemes to preference variation and strategic manipulation. *Public Choice*, 47:349–370, 1985.

[32] S. Obraztsova and E. Elkind. On the complexity of voting manipulation under randomized tie-breaking. *COMSOC 2012*, page 347, 2012.

[33] S. Obraztsova, E. Elkind, and N. Hazon. Ties matter: Complexity of voting manipulation revisited. In *The 10th International Conference on Autonomous Agents and Multiagent Systems-Volume 1*, pages 71–78. International Foundation for Autonomous Agents and Multiagent Systems, 2011.

[34] G. Pritchard and M. Wilson. Exact results on manipulability of positional voting rules. *Social Choice and Welfare*, 29:487–513, 2007.

[35] R. Reyhani. *Strategic manipulation in voting systems*. PhD thesis, 2013.

[36] R. Reyhani, G. Pritchard, and M. Wilson. A new measure of the diffculty of manipulation of voting rules, 2009.

[37] M. A. Satterthwaite. Strategy-proofness and arrow's conditions: Existence and correspondence theorems for voting procedures and social welfare functions. *Journal of Economic Theory*, 10(2):187 – 217, 1975.

[38] D. A. Smith. Manipulability measures of common social choice functions. *Social Choice and Welfare*, 16:639–661, 1999.

[39] G. Ulrich. Computer generation of distributions on the m-sphere. *Applied Statistics*, pages 158–163, 1984.

[40] T. Walsh. Manipulability of single transferable vote. In F. Brandt, V. Conitzer, L. A. Hemaspaandra, J.-F. Laslier, and W. S. Zwicker, editors, *Computational Foundations of Social Choice*, number 10101 in Dagstuhl Seminar Proceedings. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2010.

[41] T. Wang, P. Cuff, and S. Kulkarni. Condorcet methods are less susceptible to strategic voting. 2014.

[42] G. S. Watson and E. J. Williams. On the construction of significance tests on the circle and the sphere. *Biometrika*, 43(3/4):pp. 344–352, 1956.

[43] A. T. Wood. Simulation of the von mises fisher distribution. *Communications in Statistics-Simulation and Computation*, 23(1):157–164, 1994.

[44] L. Xia, M. Zuckerman, A. D. Procaccia, V. Conitzer, and J. S. Rosenschein. Complexity of unweighted coalitional manipulation under some common voting rules. In *International Joint Conference on Artificial Intelligence*, pages 348–353, 2009.

[45] M. Zuckerman, O. Lev, and J. S. Rosenschein. An algorithm for the coalitional manipulation problem under maximin. In *The 10th International Conference on Autonomous Agents and Multiagent Systems - Volume 2*, AAMAS '11, pages 845–852. International Foundation for Autonomous Agents and Multiagent Systems, 2011.

[46] M. Zuckerman, A. Procaccia, and J. Rosenschein. Algorithms for the coalitional manipulation problem. *Artificial Intelligence*, 173(2):392–412, feb 2009.