Master 1 internship

Ecole normale suprieure
Computer science department

Algorithms and Systems for Computational Social Choice: Incorporating Context into Preference Aggregation

Final report

Theo Delemazure
theo.delemazure@ens.fr

March-August 2019

Advisor:
Prof. Julia Stoyanovich

New York University
Center for Data Science
60, 5th avenue
New York

Introduction

In the context of my first year of Master's degree, I did a 5-months internship at the *New York University* on the *DBCOMSOC* project. As *DBCOMSOC* stands for *Databases meet computational social choice*, the main aim of this project is to develop a unifying framework that brings together computational social choice ([6]) and database query languages. The question of "*how to aggregate preferences of individuals in order to reach a consensus?*" is one of the most important questions from computational social choice, and a lot of recent research is focused on the determination of actual and potential winners in a poll or an election. In the past decades, the data management community investigated methods to efficiently query large preferences databases, and it is not rare to have incomplete preferences, for instance on restaurant recommendation databases. The main goals of my internship, per internship project proposal, were the following:

- To understand the **theoretical and systems challenges** inherent in preference aggregation, in the presence of incompleteness, and with database context. These notions are introduced in [10].
- To **implement the necessary winner, possible winner, necessary answer, and possible answer** primitives for a subset of positional scoring rules.
- To analyze performance bottlenecks experimentally, and **develop performance optimization** for the implemented methods.

These goals were the guidelines of my internship, but we also looked at **the synthetic generation** of realistic data and at **collecting real data** to test our algorithms.

The remainder of this document is organized as follows. **Section 1** formally introduces the concepts of *incomplete preferences* and *voting rules*. Synthetic data generation and collection of real datasets of posets are detailed in **Section 2**. The work on implementing and optimizing the *Necessary winner* algorithm, done jointly with *Kunal Relia*, a PhD student at NYU advised by Prof. Julia Stoyanovich, is explained in **Section 3**. **Section 4** details the work on possible winners, done jointly with *Vishal Chakraboty*, a PhD student at the University of California Santa Cruz, advised by Prof. Phokion Kolaitis. While he was working on reducing the possible winner algorithm using *Integer Linear Programming (ILP)* to solve the *Possible Winner* problem, I was working on an *pruning* method that takes less time to run, and leads to an over-all improvement for many workloads. These three sections will be part of the paper we worked on during the internship and that we will submit to *AAAI-2020*. Finally, **Section 5** details how I implemented the first API for *Possible and Necessary answer* primitives.

1 Preliminaries

1.1 Definitions

Let $\mathcal{C}$ be a set of m elements, called *candidates*. A *total order* (or *ranking*), $\rho = (\rho_1, .., \rho_m)$ on $\mathcal{C}$, is a permutation of $\mathcal{C}$. It also defines a total ordering on $\mathcal{C}$ and we write $a \succ_\rho b$ if a is ranked better than b. We have $\rho_1 \succ_\rho \ldots \succ_\rho \rho_m$.

A *partially ordered set* (or ***poset***) P on $\mathcal{C}$ is a set of preference pairs (a, b) associated to a binary relation $\succ_P$ defined on $\mathcal{C}$. We write $\text{TC}^+(P)$ to denote the transitive closure of P (the smallest set of preference pairs that contains P and is transitive). We then have $a \succ_P b$ if $(a, b) \in \text{TC}^+(P)$. We suppose that the poset is acyclic.

A *transitively implied pair* on P is a pair (a, b) such that $\exists o_1, \ldots, o_k$, with $k > 2, a = o_1, b = o_k$ and $\forall i \geq 2, (o_{i-1}, o_i) \in P$. We denote by $\text{TC}^-(P)$ a poset P' that does not contains any transitively implied pairs, such that $\text{TC}^+(P) = \text{TC}^+(P')$. A poset is said to be *without transitive closure* if $P = \text{TC}^-(P)$.

Proposition 1. *For a poset P, $\text{TC}^+(P)$ and $\text{TC}^-(P)$ exist, and are unique. Moreover, the posets on $\mathcal{C}$ implied by P, $\text{TC}^+(P)$ and $\text{TC}^-(P)$ are the same.*

We can represent a poset as a preference graph. A *preference graph* is a directed acyclic graph (DAG), in which nodes are candidates from $\mathcal{C}$ and there is a directed edge from a candidate a to a candidate b in the graph if $(a, b) \in P$. To avoid any confusion, we consider that the preference graph is without transitive closure (i.e. the graph represents $\text{TC}^-(P)$).

A *(complete) voting profile* is a set $PW = (T^1, \dots, T^n)$ of total orders on C. Similarly, a *partial voting profile* is a set $\mathcal{P} = (P_1, \dots, P_n)$ of posets on C. A *completion* of a partial voting profile $\mathcal{P} = (P_1, \dots, P_n)$ is a complete voting profile $PW = (T^1, \dots, T^n)$ such that each T^k is a completion of the partial order P_k (i.e. T^k is a total order that extends P_k). We denote $\mathcal{PW}(\mathcal{P})$ the set of completions of a partial voting profile $\mathcal{P}$. Note that, in general, the set $\mathcal{PW}(\mathcal{P})$ have exponential size.

1.2 Families of posets

There are some interesting families of posets that are well-studied and for which **we can optimize** the running time of some of our algorithms.

- A *linear forest* poset is a poset in which every candidate has at most one direct parent and one direct child in the preference graph. Each candidate can be uniquely identified by the sub-ranking to which it belongs and its relative position on it (See *Figure 1a*).
- In *partitioned* preferences, we have $(S_1, \dots, S_k)$, k sets of candidates such that $\forall i, j, i < j, \forall a \in S_i, b \in S_j, a \succ_P b$ and $\forall i \in [1, k], \forall a, b \in S_k, a \not\succ_v b \wedge a \not\prec_v b$ (See *Figure 1b*).
- *Top-k* is a particular kind of partitioned preferences with $k+1$ sets of candidates $(C_1, \dots, C_{k+1})$ and $\forall i \in [1, k], |C_i| = 1$ and $|C_{k-1}| = m - k$ (See *Figure 1c*).
- A *tree* is a poset in which there is at most one path from one node to another in the preference graph without transitive closure (See *Figure 1d*).

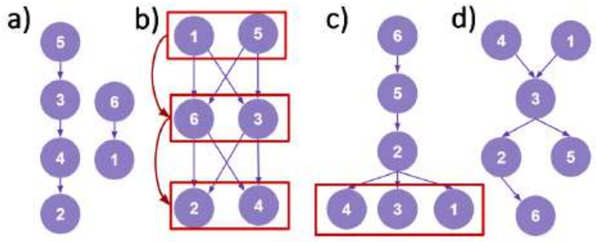

Figure 1: Different types of posets: **(a)** *Linear* **(b)** *Partitioned* **(c)** *Top-k* and **(d)** *Tree.*

1.3 Positional scoring rules

We assume a set $\mathcal{C}$ of m candidates and a set $\mathcal{V}$ of n voters. Each voter gives its preferences over m candidates in the form of a full ranking ρ. Then, there are multiple voting rules that can be used to select a winner. A common family of rules is the *positional scoring rules* family.

Let $\vec{s} \in \mathbb{N}^m$ be a vector of a positional scoring rule for m candidates, such that $\forall i, j, i < j \Rightarrow s_i \geq s_j$. For a total order $\rho = (\rho_1 \succ_\rho \dots \succ_\rho \rho_n)$, $\forall i \in [1, m]$, the candidate in position i, ρ_i will get s_i points. *The winner of the election* is the candidate with the most points when we sum scores across all the voters.

Here are some examples of positional scoring rules:

- *Borda*: $\forall i \in [1, m], s_i = m - i$.
- *k-approval*: $\forall i \leq k, s_i = 1$ and $\forall i > k, s_i = 0$. More specifically, 1-approval is called *plurality* and $(m-1)$-approval is called *veto*.
- *Formula 1*: The ranking used in Formula 1 races is $\vec{s} = (25, 18, 15, 12, 10, 8, 6, 4, 2, 1, 0, \ldots, 0)$.

1.4 Necessary winners and Possible winners

Let r be a voting rule and $\mathcal{P}$ a partial voting profile.

- A candidate c is a *necessary winner* with respect to r and $\mathcal{P}$ if it is a winner in every completion of $\mathcal{P}$ for the voting rule r.
- A candidate c is a *possible winner* with respect to r and $\mathcal{P}$ if it is a winner in at least one completion of $\mathcal{P}$ for the voting rule r.

The notions of *necessary unique winners* and *possible unique winners* are defined in analogous manner. We do a deeper analysis of the state-of-the-art of these problems in **Sections 3 and 4**.

2 Data generation

To test the efficiency of our algorithms, we needed datasets with incomplete preferences. We had two possibilities to obtain such datasets: Generate synthetic datasets, or use real datasets. **Section 2.1** presents the methods we used for synthetic data generation. Real datasets with posets are not easily available online. Hence, we collected real datasets into two ways: (i) transform preference datasets available online into partial preference datasets, and (ii) collect our own dataset of partial preferences. **Section 2.2** reviews our data collection and transformation methods.

2.1 Synthetic datasets

For our synthetic datasets, we used various parameters, which are summarized in Table 1. I will describe the necessary terminology next.

# Mallows	1,3 or 9 with $\phi = 0.5$
#candidates m	$5, 10, 15, 20, 25, 50, 100, 150$ or 200
#voters n	10^x with x from 1 to 4
Methods	*Drop candidates*, *RSM* and *Top-k*

Table 1: Parameters for the synthetic datasets.

2.1.1 Preliminaries

Currently, there are a lot of models used to generate total orders. One of the most commonly used is the one introduced by *Mallows* in 1957. In a Mallows ϕ-model, with a reference ordering σ and a distribution parameter ϕ, the probability to generate a total order ρ^0 is given by:

$$P(\rho^0|\sigma, \phi) = \frac{\phi^{d_{\text{KT}}(\sigma, \rho^0)}}{\sum_\rho \phi^{d_{\text{KT}}(\sigma, \rho)}} \tag{1}$$

where d_{KT} is the *Kendall-Tau distance*, defined as the number of inversions between the two rankings:

$$\begin{aligned} d_{\text{KT}}(\sigma, \rho) = |\{(i,j)|i < j \ \wedge \ &((i \succ_\sigma j \wedge i \prec_\rho j) \\ &\vee (i \prec_\sigma j \wedge i \succ_\rho j))\}| \end{aligned}$$

When $\phi = 1$ we obtain the uniform distribution over the set of all possible rankings. When $\phi \to 0$, rankings sampled tend to be closer to σ.

While the Mallows model has been known for a long time, there were no efficient methods for sampling from that model until recently. Almost 50 years later, the *Repeated Insertion Model (RIM)* [8] was introduced, and proposed a simple way to sample total orders from the Mallows model. RIM is also more general than Mallows and can simulate a large family of models. However, RIM does not immediately lend itself to sampling posets. This is the reason we introduce the *Random Selection Model (RSM)*.

We first developed RSM, and later found a 1986 paper by [9] that also focused on generating posets. That paper proposed and compared several methods, and comparing the density of the generated poset datasets. We compare the methods of [9] with RSM in **Section 2.1.4**.

2.1.2 Drop candidates

Perhaps the most natural method to generate posets from rankings is the *drop candidates* method. This method is very natural: Once the full ranking is sampled, we randomly drop between 0 and $(m-2)$ candidates to obtain a *linear subranking.* This kind of a dataset appears frequently, for instance when users rank movies: If a user never saw a movie, she will omit it in the ranking. However, the datasets obtained with this method are really sparse and we want to be able to generate more diverse posets.

I then started searching for new methods. In one of them, we select exactly one comparison pair $a \succ b$ for each candidate a. In another one, the probability to add a pair is given by the distance between the two candidates in the total order. All these rules contained **two steps**: (1) generate a total order from a Mallows model and (2) generate a poset from this total order. The reason why I introduced RSM was that we wanted a method that could easily **merge these two steps into one step**.

2.1.3 The Random Selection Model

Similar to the Repeated Insertion Model, the *Random Selection Model* is also a process model that can generate any possible ranking distribution. But, while in RIM we **"insert"** every candidate one by one in the output ranking, in RSM we **"select"** every candidate one by one to build our output ranking. When we **"insert"**, the candidate is given and we are randomly choosing its position in the temporary ranking, and when we **"select"**, the position on the ranking is given and we select which candidate will take this position.

For instance, the RSM algorithm selects which candidate will be the first one in the output ranking, then the second one, etc. The process continue until every candidate have been selected. At the last step, the last remaining candidate takes the last position.

Formally, we have two inputs: a reference ordering σ and a probability matrix Π. When we select the i^{th} candidate at *step* i, we rank the remaining candidates in the order they are on σ and we obtain a ranking σ^i with $m-(i-1)$ candidates. Then, we select the j^{th} candidate of σ^i with probability $\Pi_{i,j}$.

With the above process, we obtain a full ranking. With $\forall i, \forall j \leq m-(i-1), \Pi_{i,j} = \frac{\phi^{j-1}}{\sum_{k=1}^{m-(i-1)} \phi^{k-1}}$, we obtain a Mallows model with parameters (σ, ϕ).

Next, to enable us to obtain posets from this, we use **a third parameter**: the *probability vector* $p = (p_1, \dots, p_m)$. We initialize the poset as an empty list of preference pairs P. At *step* i, when we select the i^{th} candidate c_i, we know that every candidate that is not yet selected will be less well ranked than c_i. Consequently, for every remaining candidate c', we independently add the preference pair (c_i, c') to P with probability p_i.

With particular settings for p, we obtain known "poset" models. For instance, with a uniform p, we simulate *the random graph model* from [7], which is also the *Method 1* from Gehrlein [9].

With $p = (1, \dots, 1, 0, \dots, 0)$, we will generate *top-k* posets (with k = number of 1) and with $p = (0, \dots, 0, 1, \dots, 1)$, *linear* posets (of size k = the number of 1).

For our experiments, we used the *RSM method* with a random probability vector p for every voter, and the *Top-k method* using RSM with $p = (1, \dots, 1, 0, \dots, 0)$ and a random number of 1 for each poset.

2.1.4 RSM and other methods

After two months using RSM to sample datasets, another researcher on the project sent us a paper from 1986 by *Gehrlein* [9] describing 3 methods to generate posets. Like for the *drop candidate* method, the process for these methods is separated in two: generate the total order, then the poset. The *Method 1* is basically RSM with a uniform p. In the *Method 2*, the probability to add a preference pair is based on the distance between the two candidates in the full ranking: the further they are, the higher the probability. In his paper, *Gehrlein* studied the link between the parameter of the methods and the density of the generated dataset. We did the same for RSM and obtained the Figure 2.

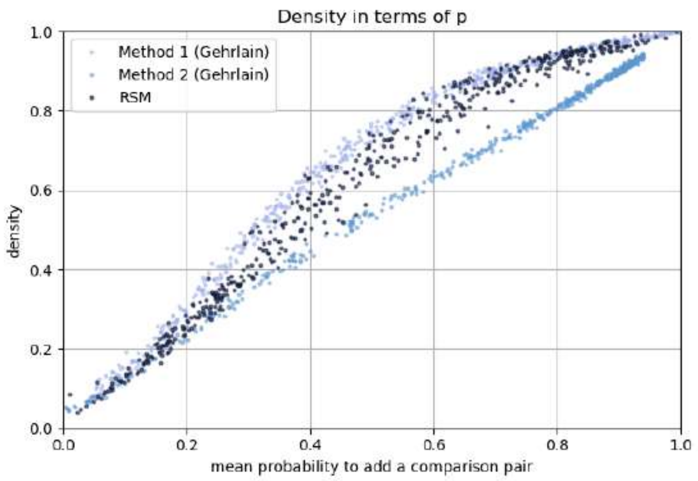

Figure 2: Comparison of the link between the mean probability to add a comparison pair and the density of the dataset for different methods. **Each dot represent a partial voting profile** with 25 voters with 10 candidates.

As a reminder, *the density* is defined as the number of preference pairs in the poset P, divided by the maximum number of pairs. With m candidates, we have:

$$d = \frac{2|\textsc{tc}^{+}(P)|}{m(m-1)}$$

As we can see on Figure 2, the *Random Selection Model* cover a higher range of density for one particular mean probability. It can be explained by the fact that RSM is more general, because it **uses an m-dimensional parameter** as compared to *Gehrlein*'s methods having 1-dimensional parameters.

2.2 Real datasets

Once we had tested our algorithms on synthetic datasets, we wanted to test it on real datasets too. Moreover, an interesting observation would be to see whether RSM fits with real datasets or not. At the end, we had *3 different real datasets*, which are summarized in Table 2.

Name	Method	#candidates	#voters	density
Desserts	Collect	8	229	0.5 $\sim$ 0.9 (variable)
Travels	Transform	24	5456	0.91 (dense)
Books	Transform	241	1288	0.02 (sparse)

Table 2: Our real datasets.

2.2.1 Collect a real dataset

After unsuccessfully searching *poset datasets* online for a long time, the first thing we tried to obtain a real dataset was **to create our own experiment**. The *8 candidates* of our election were American desserts. I thought a lot about how to collect interesting posets, and the first idea was to show two options to the subject and ask him to choose between three answers: *Option 1*, *Option 2* and *Undecided*. I asked some friend to fill the form, but when the first results came, almost nobody used the *Undecided* answer, so we only obtained total orders, which are not very interesting for us.

The problem disappeared with the second version of the form. In this new version, the respondent has **to move a slider on his screen**. The more the slider is on the left side, the more the respondent prefers the option 1, and conversely for the right side. If the slider is in the middle of the screen, that means the respondent is undecided (see Figure 3).

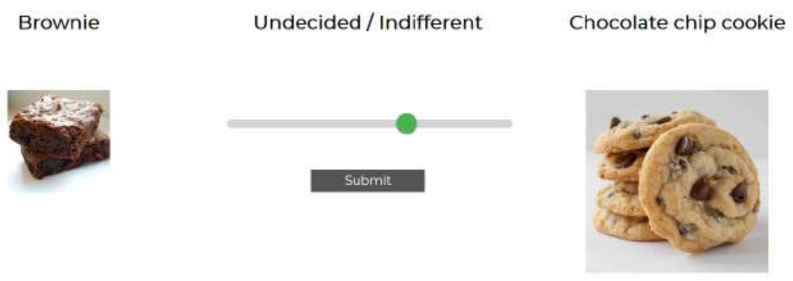

Figure 3: Screenshot of the *DessertPref* experiment showing the slider used by the participant to select between the two desserts; right position of slider denotes Cookie is preferred over Brownie, center position denotes undecided state.

After a lot of internal testing and waiting for approval of the experiment by the *Institutional Review Board (IRB)*, we finally sent our experiment to students and friends and collected 229 answers. The form is available on [1].

Thanks to the slider, we can actually obtain dozens of datasets from this Dessert dataset. Indeed, by changing the confidence threshold (i.e. the proportion of the slider that is considered as *undecided*), we can obtain very different datasets. We called this threshold the *Indecision Margin (IM)* with $IM = 0$ if no response is considered as undecided and $IM = 50$ if we only take into account the extreme positions of the slider. As we can see on Figure 4, there are some unusable entries in the datasets, but that's still interesting to study. For instance, the number of **empty posets** is increasing with the IM, because we drop more pairs when the IM is high, and the number of **posets with cycles** is higher for low IM, because we take into account undecided answer, which are more likely to be inconsistent with the rest of the poset.

We extensively studied this dataset, and particularly studied the *confidence* of the voters. We called *confidence* the average distance between the slider and the middle of the screen for one particular voter. We tried to see if there was a link between the confidence and other parameters, like the time taken to fill the form. We found that there is a strong correlation between confidence and mother tongue (see Figure 5). For instance, an interesting observation is that the median confidence of French speakers is less (60%) than that of Hindi speakers (73%). One hypothesis is

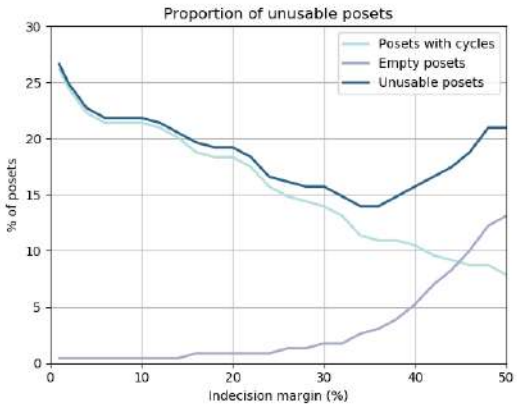

Figure 4: Proportion of **unusable posets** (empty or with cycle) in terms of the *indecision margin*.

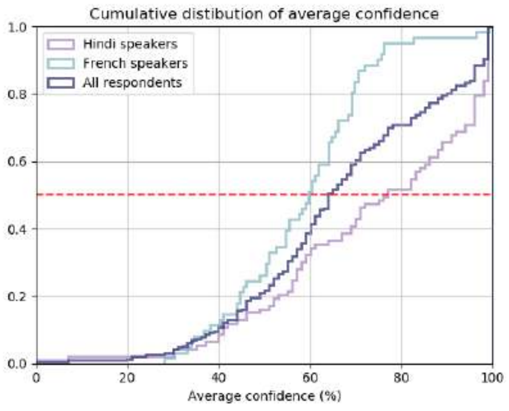

Figure 5: Cumulative distribution of **the confidence** among the *Dessert Dataset*. The red line shows the median.

that the latter group **just don't know all the Desserts from the form**, which are standard American desserts.

We verified if this dataset follows a Mallows distribution, or at least a mixture of mallows. With a mixture of 2 Mallows, the model seems to fit pretty well. We also verified if the structures of the posets follow **a mixture of Random Selection Model**. It is harder to fit RSM than Mallows, because for Mallows, the ϕ parameter is a pretty good indicator of success, but there is no such parameter for RSM. The indicator we used to check if the dataset verified a RSM model was the distribution of the number of pairs on each posets. As we can see on Figure 6, we can approach with really good precision the *Dessert dataset* with $IM = 10$, but it becomes less good for $IM = 30$.

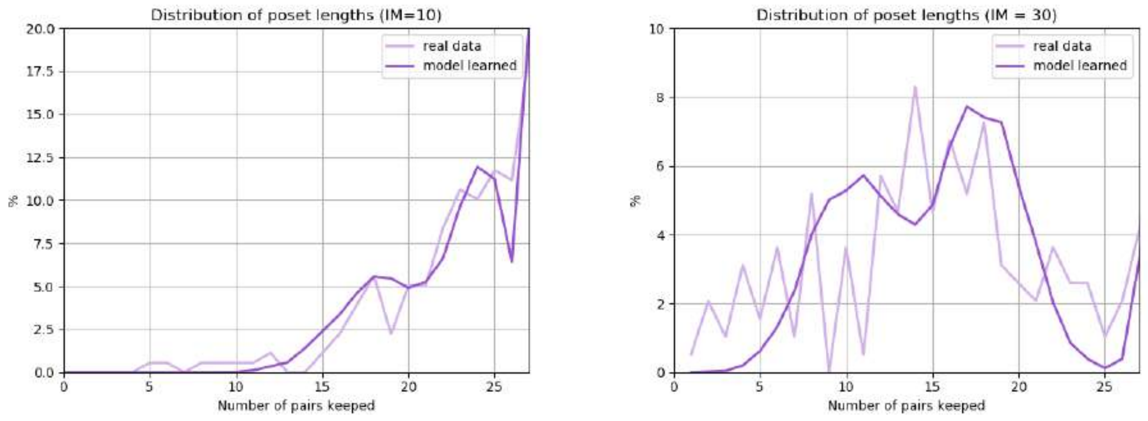

Figure 6: Best fit of the RSM model over the distribution of poset lengths among the *Dessert dataset* with an *Indecision Margin* of 10 and 30.

2.2.2 Transform real datasets

We also wanted **larger datasets** to test our algorithms. For that, we used large datasets from review website like *TripAdvisor*, *Amazon* or *Yelp*. However, these datasets do not contain preference pairs but **ratings**. The task of converting ratings into preference pairs was done by two high school students who were part of the *New York University ARISE program* and helped us in the project during the summer.

They converted ratings into posets by adding a comparison pair only if the two candidates have an **absolute rating difference** greater than some threshold δt. With this method, we obtained two datasets: one from *Google Travel Ratings* and the second from *BookCrossing book reviews*. These datasets and their parameters are described in **Table 2**.

3 Necessary winners

The first election problem we wanted to solve was the *Necessary Winner problem*. *Xia and Conitzer* studied this problem for several voting rules and found a dichotomy between rules for which we can solve the problem in **polynomial time** and rules for which the problem is **coNP-Complete** [12].

They showed algorithms for voting rules for which the problem is PTIME, notably for positional scoring rules (defined Section 1.3). As we did not find any implementation of that algorithm online, **we implemented and optimized** the algorithm they proposed to identify necessary winners in elections with positional scoring rules and incomplete preferences.

3.1 Preliminaries: *Xia and Conitzer*'s algorithm

Before explaining the algorithm, we need to introduce **some notations**:

Definition 1. *For a poset P and a candidate $c \in \mathcal{C}$, we define $\textsc{Up}_P(c)$ and $\textsc{Down}_P(c)$ as follows:*

$$\textsc{Up}_P(c) = \{c' \in \mathcal{C} | c' \succeq_P c\}$$
$$\textsc{Down}_P(c) = \{c' \in \mathcal{C} | c' \preceq_P c\}$$

Definition 2. *For a poset P and two candidates $c_1, c_2 \in \mathcal{C}$ such that $c_1 \succ_P c_2$, we define $\textsc{Block}_P(c_1, c_2)$ as the set of candidates contained between c_1 and c_2:*

$$\textsc{Block}_P(c_1, c_2) = \{c' \in \mathcal{C} | c_1 \succeq_P c' \succeq_P c_2\} = \textsc{Down}_P(c_1) \cap \textsc{Up}_P(c_2)$$

We assume that we have a partial voting profile $\mathcal{P}$ of n posets *without transitive closure* over m candidates from $\mathcal{C}$. We denote the score of the candidate c in a total order T with $S_T(c)$, and the score of the candidate c in a completion $PW \in \mathcal{PW}(\mathcal{P})$ with $S_{PW}(c) = \sum_{T \in PW} S_T(c)$.

We also note that

$$\min_{PW \in \mathcal{PW}(\mathcal{P})} (S_{PW}(c) - S_{PW}(w)) = \min_{PW \in \mathcal{PW}(\mathcal{P})} \sum_{T_i \in PW} (S_{T_i}(c) - S_{T_i}(w)) = \sum_{P_i \in \mathcal{P}} \min_{T_i \in \mathcal{T}(P_i)} (S_{T_i}(c) - S_{T_i}(w)) \quad (2)$$

The algorithm from *Xia and Conitzer*'s paper [12] for a positional scoring rule $\vec{s}$ is described as follows:

1. For each poset $P \in \mathcal{P}$ and each candidate $c \in \mathcal{C}$, compute $\textsc{Up}_P(c)$ and $\textsc{Down}_P(c)$.
2. For each candidate $w \neq c$, minimize the score difference between c and w over all the *completions of $\mathcal{P}$*. Because of equation (2), we just need to minimize the score difference for each poset P:
   - If $c \not\succ_P w$, the lowest possible position for c is the $(m - (|\textsc{Down}_P(c)| - 1))^{th}$ position, and the highest possible position for w is the $|\textsc{Up}_P(w)|^{th}$ position.
   - If $c \succ_P w$, then, to find the minimum score difference, we need to slide the block $\textsc{Block}_P(c, w)$ between its two extreme positions and only keep the minimum score difference. This has a complexity $O(m)$ in the worst case.
3. If, for at least one $w \neq c$, we obtained $\min_{PW \in \mathcal{PW}(\mathcal{P})} S_{PW}(c) - S_{pw}(w) < 0$, then there is a completion in which c have a lower score than w, and consequently c is not a NW. Otherwise, c is a NW.

To find *necessary unique winners*, we replace the < 0 with ≤ 0 in the *step 3*.

3.2 Optimizations

The above algorithm is not really precise, and particularly the first step (computation of $\textsc{Up}_P(c)$ and $\textsc{Down}_P(c)$). If we don't take care of it, the running time of the algorithm can increase really quickly. With *Kunal Relia*, we tried **to improve every step of the algorithm**. The optimizations we've implemented and the results obtained are detailed in this section.

3.2.1 Computation of Up and Down

In [12], *Xia and Conitzer* did not explain how they compute $\textsc{Up}_P(c)$ and $\textsc{Down}_P(c)$. The most efficient way we found to compute it is to use a *BFS-like algorithm.* For reasons of space, I will not explain the algorithm here. The complexity of this algorithm for one poset is $O(m^2)$ in the worst case.

For some families of posets, we can **reduce this complexity** significantly:

- For *linear posets* (Fig. 1a), the relative rank of a candidate in the subranking to which it belongs to gives us enough information for the next steps of the algorithm. The complexity of this algorithm **decreases to** $O(m)$ for these posets.
- In the case of *partitioned preferences* (Fig. 1b), the relative rank of the set S_i to which each candidate belongs to and the length of each set are sufficient for the next steps of the algorithm. The complexity of this algorithm also **decreases to** $O(m)$ for these posets.
- The improvement for *trees* (Fig. 1d) is less important: Instead of using a **set** for $\textsc{Up}_P$ in step 3, a **list** is a sufficient. Consequently, we can get rid of some operations with $O(m^2)$ cost, but the algorithm **still has complexity** $O(m^2)$.

Moreover, if there is **a consensus in the population**, it is likely that a lot of voters will have the same preferences, and it is also likely that we will do the same computation for $\textsc{Up}$ and $\textsc{Down}$ several times. One way to avoid that is to keep in a dictionary the $\textsc{Up}_P$ and $\textsc{Down}_P$ of each poset P we've seen. Before computing these sets, we check if they are not already in the dictionary. This last optimization worked very well in case of a strong consensus, but in most cases it adds too much time maintaining and querying the dictionary, **so we didn't keep it for our experiments**.

3.2.2 Number of competitions

We call the step 2 of the algorithm shown in Section 3.1 a *competition* between the candidate c we are testing and an *opponent* w. In average, the number of competitions is $O(m^2)$ (if we try every couple of candidate (c, w)) but thanks to optimizations and heuristics, we can reduce significantly the number of competitions.

The first simple trick is that if we test c against w and $\min_{PW \in \mathcal{PW}(\mathcal{P})}(S_{PW}(c) - S_{PW}(w)) > 0$, we don't need to test w anymore because it is **definitely not a necessary winner** (w always lose against c). Thanks to this optimization, the average number of competition is reduced to $O(m)$.

Moreover, during the computation of $\textsc{Up}$, we can compute at the same time the **maximal score** of every candidate. Indeed, the best position a candidate $c \in \mathcal{C}$ can have in a poset P is $|\textsc{Up}_P(c)|$. Then, the candidates with the highest maximal score are the only ones that can be necessary winners, because in the completion of $\mathcal{P}$ in which they achieve their maximal score, every other candidate have a lower score, and consequently is not a necessary winner.

Moreover, it is more likely for a candidate we are testing to lose against an opponent with a high maximal score. A useful heuristic is then to **choose opponents in the decreasing order of their maximal scores**. Thanks to these two improvements, we can decrease the number of competitions in the general case to $O(1)$ when there is no NW and $O(m)$ otherwise.

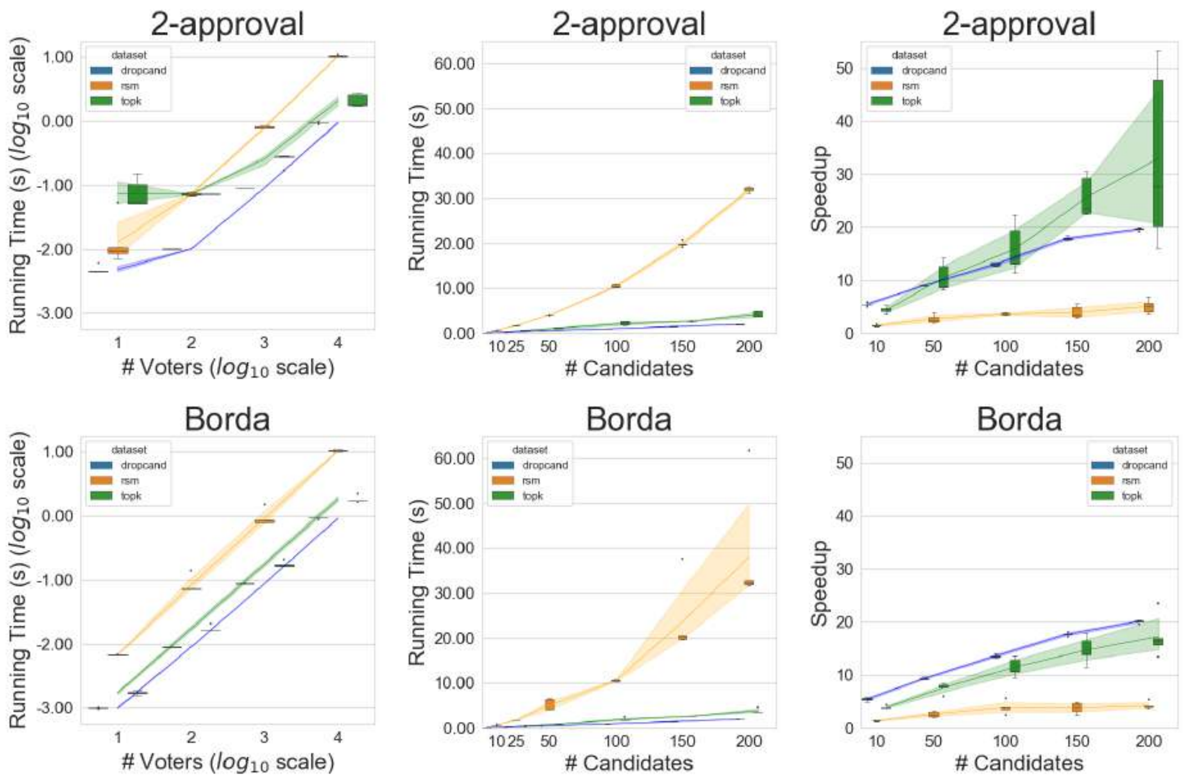

Figure 7: *left:* Dependency of the running time of the optimized NW algorithm on the number of voters $\#n$ with 100 candidates. *center:* Dependency of the running time of optimized NW algorithm on the number of candidates $\#m$ with $10,000$ voters. *right:* Speedup ($\frac{\text{Non-optimized}}{\text{Optimized}}$) of the algorithm by varying m. Figure on top row are with 2-*approval* rule and figure on bottom row with *Borda* rule. Each line correspond to one particular dataset.

3.2.3 Complexity of one competition

This part of the algorithm (the second step) is already described on *Xia and Conitzer*'s algorithm and has a complexity $O(m)$ per voter in the worst case. However, we can improve the efficiency of this step in many ways.

As explained in section 3.2.1, for *linear poset* and *partitioned preferences*, we can replace UP and DOWN by other values, which need less running time to use on a competition. For these particular kind of partial voting profile, the complexity in the worst case is $O(1)$ per voter.

For other partial voting profile, we can still improve the complexity. Indeed, in the second step, in the case where $c \succ_P w$, the best position of the block to minimize the difference of score only depends on 3 values: the **highest and lowest positions** of the block and the **size of the block** between w and c. Instead of computing the same thing several times, we compute the best position for the block for every possible case during a preprocessing step using *dynamic programming*. The preprocessing step has complexity $O(m^3)$ and the complexity of a competition is reduced to $O(1)$ per voter, that's why we use this optimization only if $m^2 < n$.

For *Borda* rule, we don't even need to find the best position of the block, because the minimal difference of score only depends on the length of the block between the two candidates. For k-*Approval* rule, we don't need the preprocessing step either because we can compute the minimal score difference in constant time. For these two rules, we can reduce the complexity to $O(1)$ per voter.

3.2.4 Multithreading

Next, we studied **the impact of parallelization** on the computation time of the necessary winner algorithm [12]. We parallelized the first step of the algorithm (computing Up and Down), and we specifically used 3 different system setups to check the reproducibility of the parallelization. For each setup, we found parallelized implementation **to be faster than non-parallelized implementation** and up to 2.5 times faster. As the parallelization plateaued after a certain number of threads across the different setups, we invested a lot of time trying to improve the parallelization, but the running time was always slowed down by the output of the parallelized function (i.e. the computation of Up and Down), which is apparently too big for the `multithreading` python library to work well.

3.3 Results

We did a lot of experiments with the *Necessary Winner* algorithm (without multithreading) for the paper we want to submit to *AAAI-20*. Figure 7 summarizes the results for *Borda* and *2-approval* scoring rules. The results for *plurality* are very similar to the ones for *2-approval*.

The first column of Figure 7 highlights the linearity of the algorithm in the number of voters. Thanks to the second column, we can check the quadratic evolution of the running time for *rsm* datasets and linear evolution for *drop candidates* and *top-k*. Finally, the last column shows the **speedup** of our optimized version (i.e. how many time our algorithm is faster than the non-optimized algorithm). The running time is up to 4 or 5 time faster for *RSM* datasets. With *Borda* rule and *top-k* and *drop candidate* datasets, the algorithm is up to **20 times faster**. With *2-approval*, the speedup is really variable for *top-k* (and up to **50 times faster**), it is due to the fact that there is often a necessary winner in *top-k* datasets, which is rare for *RSM* and impossible for *drop candidate*.

4 Possible winners

The second main problem in elections with incomplete preferences is the *Possible Winner problem*. In [5], *Betzler and Dorn* showed an almost perfect dichotomy between positional scoring rules for this problem. Indeed, they described a **PTIME algorithm** for *Plurality* and *Veto* and proved that this problem is **NP-Complete** for other positional scoring rules (including *Borda* and *k-approval*). This dichotomy was completed later in [4] and extended to more voting rules in [12].

I implemented the algorithms described by *Betlzer and Dorn* for *Plurality* and *Veto* and we tried to solve this problem for other scoring rules in the **most efficient way possible**.

4.1 Plurality and Veto

The algorithm presented in [5] reduce the PW problem to the *maximum flow problem*, which is known to be **PTIME**. For instance, to solve PW for *plurality*, we need to build a graph with one node for each candidate, one node for each poset and an edge between a candidate c and a poset P if c **can be ranked in first position** in the poset P. I implemented this algorithm using the `pymaxflow` python library and significantly reduced the running time with a few heuristics:

- First, we can **gather nodes of posets** which have same output nodes (i.e. the same set of candidate that can be ranked first in the poset) to significantly reduce the size of the graph.
- We can also eliminate **obvious winners and losers**: If a candidate can be ranked first in more than half of the posets, it is an *obvious winner* and if it can be ranked first in less than $1/m$ of the posets, it is an *obvious loser*. This can be done in time $O(1)$ and in many cases, a lot of candidates are obvious winners or losers (see Table 3).

- Then, we can try to **build a completion** of the partial voting profile in time $O(mn)$ in which the candidate we are testing is a winner before using the maximum flow algorithm. This helps to solve the problem more quickly for some candidates (see Table 3).

Dataset	Obvious W/L	Completion	Maxflow
Drop candidate	70%	18% (total 88%)	12% (total 100%)
Top-k	82%	13% (total 95%)	5% (total 100%)
RSM	48%	51% (total 99%)	< 1% (total 100%)

Table 3: Percentage of candidates for which we conclude at each step of the Possible Winner algorithm for *plurality* (**1.** Obvious winners/losers, **2.** Completion of the partial voting profile, **3.** Maxflow algorithm)

Finally, the running times of the algorithm I implemented are shown on Table 4. The running time is really low for both *top-k* and *RSM* datasets, and is linear in both n and m. It is a bit higher for *drop candidates* datasets, for which it seems quadratic in m, because all candidates on these datasets are possible winners, so we need to test more candidates (as detailed on Table 3).

Dataset	Total time (s)	Time/candidate (s)	Evolution with m
RSM	1.06	0.005	Linear
Top-k	0.44	0.002	Linear
drop-cand	31.27	0.154	Quadratic

Table 4: Running time of the PW algorithm for plurality on the 3 datasets. The total time is taken for $m = 200$ and $n = 10000$.

4.2 Other positional scoring rules

To solve the *Possible Winner problem* for other positional scoring rules (mainly *Borda* and *k-approval*), we worked with two different methods: while a collaborator on the project (*Prof. Phokion Kolaitis* and his PhD student *Vishal Chakraborty*) was figuring out how to build an *Integer Linear Programming (ILP)* solver for this problem by using the GUROBI solver, I was working on a way to **approximate** the set of possible winners with a higher bound set and a lower bound set. It finally appears that the two methods could be **combined together**.

4.2.1 Using a solver

Since I did not work on this part, I'll explain really quickly how it works with the example of the *k-approval* scoring rule. In that case, the linear system we want to solve is actually quite simple, it contains $m \times n$ binary variables $x_{i,j}$ and $O(nm)$ rules:

- **Respect to the rule:** For each poset $P_i \in \mathcal{P}$, $\sum_{j=1}^{m} x_{i,j} = k$.
- **Respect to the preferences:** For each poset $P_i \in \mathcal{P}$, if $c_{j_1} \succ_{P_i} c_{j_2}$ then $x_{i,j_1} \geq x_{i,j_1}$
- w **is a PW:** For each candidate $c \neq w$, $\sum_{i=1}^{n} x_{i,c} \leq \sum_{i=1}^{n} x_{i,w}$

Then, there is a solution to this system if and only if w is a possible winner in this election for *k-approval* rule. The system for the *Borda* rule is a bit more complicated, it contains $m^2 \times n$ binary variables and takes much more time (see Figure 8).

4.2.2 Using a pruning algorithm

Even if it always found **the entire set of possible winners**, the ILP solver can take a lot of time, so I proposed **an augmented version** of the algorithm. I will explain in this section how this augmented algorithm enables us to solve the PW problem **really faster**.

Obvious winners and losers. First of all, using the first step of the NW algorithm (computation of UP and DOWN), we can compute the maximum score of every candidate c, noted $S_{max}(c)$. An **obvious possible winner** is the candidate with the highest maximum score. Moreover, candidates with a maximum score that is more than half the total number of points are **possible winners for sure** and candidates with a maximum score that is less than $\frac{1}{m}$ of the total number of points are **not possible winners for sure**. This part is same as that for the *plurality* rule.

Using Necessary Winner algorithm. I also observed that we could use the second step of *Xia and Conitzer*'s algorithm for NW (the competitions) to eliminate potential possible winners. Indeed, when we do a **competition** between c and w in which we minimize the score difference $\min_{PW \in \mathcal{PW}(\mathcal{P})}(S_{PW}(c) - S_{PW}(w))$, if this difference is > 0, then w is beaten by c **in every completion of** $\mathcal{P}$ and consequently, w is **not a possible winner**. With this step, we eliminate almost every candidate that is not a possible winner (see Table 5).

Moreover, if c is the candidate with the highest maximum score and we have the following, then w is **also a possible winner for sure** (Unfortunately, I have not enough space for the proof here).

$$\forall w' \neq c, s_{max}(w') > s_{max}(w) \Rightarrow \min_{PW \in \mathcal{PW}(\mathcal{P})}(S_{PW}(c) - S_{PW}(w')) > 0 \tag{3}$$

$$\min_{PW \in \mathcal{PW}(\mathcal{P})}(S_{PW}(c) - S_{PW}(w)) \leq 0 \tag{4}$$

So far, the augmented algorithm found **at least two possible winners for sure** (on a condition that there are two or more possible winners) and have eliminated some candidates that are not possible winners for sure. We now want to test the remaining candidates.

Build a completion. The simplest solution to test if a particular candidate c is a possible winner is **to build completion** of $\mathcal{P}$ in which c is a winner. However, if we build this completion randomly, we have a really small probability to find that c is a winner. Hence, we need to use some **heuristics**.

For each poset $P \in \mathcal{P}$, we create a total order T that is a completion of P. Let's see how we can compute this total order **wisely**.

- The first thing that seems obvious is to put the candidate we are testing c **at its best possible rank**. However, sometimes it is **not a good idea**. For instance, let's say that for some poset P, only two candidates c' and c" are preferred over c. With *2-approval* rule, if c is ranked third then c' get a point from P for sure. Then, if at the end c' wins the completion with **only one more point** than c, the algorithm cannot conclude if c is a possible winner or not. By decreasing the rank of c in T, we can also decrease the score difference between c and c' and find that c is a possible winner. That's why the first heuristic can be summarized as **putting c at the worst possible rank in which it achieves its best possible score.**

- Once c's rank is chosen, we need to set the rank of the other candidates in the poset. To do so, we use the preferences from P and if we have several possibilities for one position in T, we give more points to **the candidate with the least points so far**. Thus, we need to keep track of the score of each candidate while building the completion.

- Another useful heuristic is to **try several completions**. Indeed, depending on the order in which we see the posets from $\mathcal{P}$, the completion obtained will not be the same and in some cases, this has a huge impact on whether or not we found a possible winner. I added a parameter *shuffle* to the function so the user can precisely mention the number of different shuffles of the partial voting profile the algorithm should try.

- The last heuristic is maybe the most useful one. In most cases, the above heuristics are not enough to significantly **reduce the score of the "dominant" candidates** (i.e. the ones with the highest maximal score). That's why we can give a list of *dangerous candidates* when calling the function. Before building any completion, the list is empty. Each time we build a completion, there are **3 possibilities**:

  1. The candidate we are testing **is a winner**, then the function returns that it is a possible winner.
  2. The winner is **another candidate, which is not yet in the list of dangerous candidates**, and hence, we add the winner to the list and try again with the updated list.
  3. The winner is *a candidate from the list of dangerous candidates.* That means it won even if its score was minimized, so we stop trying to find a good completion for the candidate currently being tested, and leave it to **the ILP solver**.

  We can also add a parameter for the **maximal size of the list** of dangerous candidates.

4.2.3 Results

I ran the algorithm on all the datasets and looked at the proportion of candidate for which **we can conclude if it is a possible winner or not** after each step:

1. **NW-like algo:** Contains the obvious winners and losers found thanks to their maximum score and the not possible winners found thanks to *Xia and Conitzer*'s algorithm and competitions with other candidates.

2. **Build a completion** is the step during which we try to build a completion in which the candidate is a possible winner. To obtain these results, I tried 2 shuffles for each candidate and limited the size of the list of dangerous candidate to 4.

3. The **ILP solver** gets the remaining candidates.

The results are shown in Table 5. As we can see, for *Drop candidates* and *Top-k* datasets, the first two steps are **sufficient in almost every case**. This is probably due to the fact that the partial voting profiles are not diverse. For *RSM* datasets, we need to use the ILP solver for between 1% and 4% of the candidates. However, for more than 72% of the datasets, the first two steps are sufficient to determine precisely the set of possible winners (82% for *Borda* and 64% for *2-approval*).

Dataset	Rule	NW-like algo	Build a completion	ILP Solver
drop candidate	*Borda*	18.3%	81.7% (total 100%)	
	2-approval	22.9%	77% (total 99.9%)	0.1% (total 100%)
top-k	*Borda*	71.6%	28.3% (total 99.9%)	0.1% (total 100%)
	2-approval	96.4%	3.6% (total 100%)	
RSM	*Borda*	59.4%	39.4% (total 98.8%)	1.2% (total 100%)
	2-approval	63.4%	33.1% (total 96.5%)	3.5% (total 100%)

Table 5: Proportion of candidates for which we can conclude after each step of the algorithm.

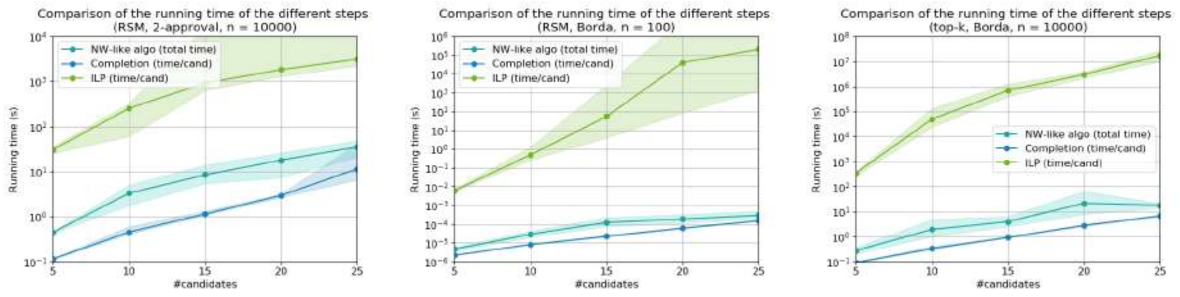

Figure 8: Comparison of the total time of the **NW-like** step and the time per candidate of **the two other steps** for different datasets parameters and voting rules.

The Figure 8 shows a comparison of the running time of each step of the augmented algorithm. For *RSM* dataset and *Borda* rule, with $m = 25$ and $n = 100$, the augmented version is **in average 1961 times faster** (when it does not need the ILP solver), and this number increases exponentially with m. The speedup is less impressive for the *2-approval* rule, but still important (in average **2.5 times faster**, all datasets included).

5 Beyond winners : necessary and possible queries

The **original aim** of this project is to develop a framework bringing together preferences, voting rules, election outcomes, contextual information and database query languages, as detailed on the paper at the origin of the DBCOMSOC project [10]. We want to be able to answer queries such as "*Is it necessary that a winner is older than 65?*" or more complicated queries such as "*Is it possible to have two winners who disagree on at least one social issue?*". These problems are studied in many recent papers [10, 11, 3].

Most of my time at the internship was actually spent on possible and necessary winners because we did a lot of work with optimizing these problems and working on data generation, but I also worked a little bit on my side on implementing a small API for **possible and necessary answers**. This work is detailed here.

5.1 Preliminaries

5.1.1 Kind of queries

An *election query* is a query of the form:

$$q(x_1, \ldots, x_k) = \text{WINNER}(c_1), \ldots, \text{WINNER}(c_l), q'(x_1, \ldots, x_k, c_1, \ldots, c_l)$$

where q' is a *conjunctive query* on the database. In my work, I only studied *Boolean election queries* (i.e $k = 0$).

We can divide election queries into different families. First of all, we separate *long* and *short* queries: We say that a query is *short* if $l = 1$ and *long* if $l > 1$. Most importantly, we distinguish queries in which all winner atoms are *disconnected* from those which have at least two winner atoms that are *connected*.

Let's define what are *connected* atoms. For that, **we build an undirected graph** in which nodes are variables appearing on atoms of q. We add an edge between two variables x and y if they appear in the same relation $R_i \in q'$. We call this graph *The Gaifman graph* of q [11]. The Figure 9 gives an example of some *Gaifman graphs* (I took them from [11]).

We say that two winner atoms $\text{WINNER}(c_i)$ and $\text{WINNER}(c_j)$ are connected if there a path from c_i to c_j on the *Gaifman graph* of q, and disconnected otherwise.

Definition 3. *$q \in \mathcal{C}_D \Leftrightarrow$ Every pair of winner atoms of q is disconnected.*

$q_2() = \textsc{Winner}(x), \textsc{Winner}(y),$
$\quad R_1(x, \texttt{true}), R_1(y, \texttt{false});$
$q^2_{wr} = \textsc{Winner}(x_1), \textsc{Winner}(x_2), R(x_1, x_2);$
$q_3() = \textsc{Winner}(x), \textsc{Winner}(y),$
$\quad R_2(x, z), R_2(y, z);$

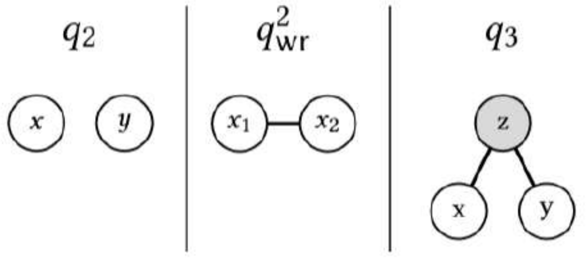

Figure 9: The Gaifman graphs of some generic queries. All the variables are Winner variables, except for the shaded z.

The complementary of $\mathcal{C}_D$ is noted $\mathcal{C}_C$. For instance, the query q_2 from Figure 9 is in $\mathcal{C}_D$, and the queries q^2_{wr} and q_3 are in $\mathcal{C}_C$.

5.1.2 Theoretical complexity

In [10], *Kimelfeld, Kolaitis, and Stoyanovich* have proved the following theorem:

Theorem 1. *For* plurality *rule or* veto *rule and q a Boolean election query:*

- *If $q \in \mathcal{C}_D$, then* NECESSITY*(q) is decidable in polynomial time.*
- *If $q \in \mathcal{C}_C$, then* NECESSITY*(q) is CoNP-Complete.*

We can easily reduce the complement of the possible unique winner problem to the necessary answer problem. We know that the possible winner problem is NP-Complete for every positional scoring rule other than plurality and veto, so NECESSITY(q) for these rules is a **CoNP-Complete problem**.

It is also easy to see that the POSSIBILITY(q) problem and the possible winner problem are equivalent. Consequently, POSSIBILITY(q) is **decidable in polynomial time** for *Plurality* and *Veto* and remains **NP-Complete** for other positional scoring rules.

The **Table 6** summarize the complexities of these problems.

		`possibility`(q)	`necessity`(q)
Plurality/Veto	$q \in \mathcal{C}_D$	P	P
	$q \in \mathcal{C}_C$	P	coNP-complete
Other rules		NP-complete	coNP-complete

Table 6: Complexity of *possible and necessary answers* problems for various scoring rules

5.2 Possible answers

Let's first see how to solve the *possible answer* problem for **short queries**. In that case, we only have one WINNER atom on the query, so we just need to check if there is one possible winner of the election that verifies the conjunctive query q'. We can use the algorithms seen in Section 4 for different scoring rules (*Plurality*, *Borda*, *k-approval*, etc.).

For long queries, the problem is a little bit harder, because we need to check if a set of candidates is a *possible winner set*. A *possible winner set* is a set of candidates such that there is a completion of the partial voting profile in which **all candidates from the set are winners**. For *plurality*, I found a way to adapt the PW algorithm for sets of size more than 2, **the hardest part** being the computation of the maximum score of the set in polynomial time.

However, for *Borda* and *k-approval*, we cannot use the *"build a completion"* algorithm for *possible winner set*. Indeed, it is very unlikely that all the candidates of the set obtain the same score at the end, which is required in our case. Hence, we have no choice but to **use the ILP solver** for sets of size 2 or more.

Finally, **the strategy to minimize the use of the ILP solver**, which takes a lot of time, is to begin the possible winner tests with sets of size 1, then 2, and so on.

5.3 Necessary answers

For necessary answer, as explained in [10], the problem is harder when we deal with *connected queries* as compared to *disconnected queries.*

Let's first see how to solve the problem for **short queries**. In that case, we can compute easily the list of candidates $\mathcal{C}_{true}$ that verify the conjunctive query q'. If we write $\mathcal{C}_{false}$ the complementary of $\mathcal{C}_{true}$, then the query is necessary if $\mathcal{C}_{false}$ does not contains any possible winner (without candidates from $\mathcal{C}_{true}$ as co-winners). We can use algorithms described in Section 4 to solve that.

By definition, we can re-write a **disconnected election query** into independent subqueries with only one WINNER atom in each subquery. Consequently, the query is necessary if and only if each subquery is necessary. We can then use the algorithm for **short queries** on each subquery.

It becomes a bit harder for **connected queries**. The first step is to use the *Gaifman graph* to divide the query into shorter independent subqueries. For subqueries with one WINNER atom, we use the algorithm for short queries. Let's see now how to find if an indivisible subquery with at least two WINNER atoms is necessary or not.

For a query with $l > 2$ WINNER atoms, the results of the conjunctive query q' are sets of candidates of size at most l. We note it $\mathcal{P}(\mathcal{C})_{true}$ and $\mathcal{P}(\mathcal{C})_{false} = \mathcal{P}(\mathcal{C}) - \mathcal{P}(\mathcal{C})_{true}$. The query is necessary if and only if no set from $\mathcal{P}(\mathcal{C})_{false}$ is a *possible unique winner set.*

A *possible unique winner set* is a set S such that there is a completion in which the set of winner is exactly S. To test if one set is a *possible unique winner set*, we use the same algorithm I described for **possible answers problem on long queries**.

Now, we want to know in which order we should test the sets from $\mathcal{P}(\mathcal{C})_{false}$. For that, I used *an a-priori* algorithm in which we test sets of small size first. It is describes below:

1. Initialize a list $L = []$. The first sets we are testing are $\mathcal{T}(\mathcal{C})^1 = \{\{c\}|c \in \mathcal{P}(\mathcal{C})_{false}\}$. For each candidate in $\mathcal{T}(\mathcal{C})^1$, there are **three possibilities:**

   - If it is **not a possible winner**, then do nothing.
   - If it is **a possible unique winner**, then **return** that NECESSITY(q) is `false`.
   - If it is a **possible winner but not possible unique winner**, then add it to L.

2. The sets of candidates for the next step are sets of size 2 with candidates from L, and that are not on $\mathcal{P}(\mathcal{C})_{true}$. Repeat (1) with these new sets.

3. More generally, let's say the algorithm did not return anything after trying sets of size k, and we have a list L of sets of size k, which are *possible winner sets* but not *possible unique winner sets.* We denote $\mathcal{P}^{k+1}(L)$ the set that contains **the sets of candidates of size $k+1$ such that every subset of size k is contained in L**. The sets of size $k+1$ we need to test are computed with:

   $$\mathcal{T}(\mathcal{C})^{k+1} = \mathcal{P}^{k+1}(L) \cap \mathcal{P}(\mathcal{C})_{false}$$

   Then, repeat (1) and (2) with these sets.

4. When we reach step m, or if L is empty after (1), then **return** that NECESSITY(q) is `true`.

5.4 Results

I implemented a parser for *boolean election queries* with the RPLY library, and used our PYCOMSOC library for possible winner algorithms. I also used the MYSQL library to execute conjunctive queries. The *Dessert database* I used in the experiments is described on Table 7.

Name	**Columns**	**Description**
DESSERTS	INT id, VARCHAR name, BOOL contains_chocolate	Contains the 8 desserts of the dataset and an information about whether they contain chocolate or not.
BAKERIES	INT id, VARCHAR name, ENUM location	Contains 8 bakeries of New York and their location (*"Downtown"*, *"Midtown"* or *"Uptown"*).
SELL	INT bakery_id, INT dessert_id, FLOAT price	Contains information about which bakeries sell which desserts. Each bakery sells 4 desserts in average and the prices are randomly chosen between \$2.5 and \$4.5.
BALLOTS	INT id_voter, INT id_dessert_1, INT id_dessert_2	Contains the preference pairs of each voter of the *partial voting profile* $\mathcal{P}$.

Table 7: Schema of the Dessert database I used on my experiments.

Then, I tried the algorithm on 3 different queries:

1. Short query: *Does a winner contain chocolate?*
2. Disconnected query: *Does a winner contain chocolate, and is a winner sold in Midtown?*
3. Connected query: *Are two winners sold by the same bakery in* Midtown *but one contains chocolate and the other does not contain chocolate?*

I first tested it on the real *Dessert dataset*, but there were not enough data points to obtain interesting results. I then created a *Fake Dessert dataset* with 10,000 data points using *the random selection model.* The running times are reported in Table 8. As expected, we have $time(Q_1) < time(Q_2) < time(Q_3)$. The only "surprising" value is for POSSIBILITY(q), *Borda* and the third query, for which the algorithm took almost 100 seconds to return. This huge running time is only due to the use of the ILP solver, which was not used in the other cases.

	NECESSITY(q)			POSSIBILITY(q)		
	Plurality	*Borda*	*3-Approval*	*Plurality*	*Borda*	*3-Approval*
Query 1	$0.03s$	$1.4s$	$1.5s$	$0.03s$	$1.4s$	$1.5s$
Query 2	$0.03s$	$1.4s$	$1.5s$	$0.05s$	$3.5s$	$3.5s$
Query 3	$0.03s$	$4.3s$	$4.3s$	$0.07s$	$97.9s$	$7.4s$

Table 8: Running time of NECESSITY(q) and POSSIBILITY(q) for 3 different queries and different voting rules.

Conclusion

Finally, after these five months of internship, we are able to implement **solvers for Possible and Necessary winners problem**, some algorithms taking more time than other. However, we tried to **optimize them** the best we can for various families of partial voting profile and voting rules. I also had a first look on **data science** with new methods of synthetic data generation and real data gathering. Finally, I implemented a small API for **necessary and possible answers**, which will probably lead to future research with new and better algorithms. These five months of works are gathered on my GitHub repository *DBCOMSOC* [2]. It contains the `pycomsoc` python library we built, which is probably one of the most complete for elections with partial voting profile. As explained, this internship also taught me what is to **prepare a paper for an international conference**, and hopefully I will continue to work on these topics with *Prof. Stoyanovich*'s team over the next months.

Acknowledgments

I learnt a lot during this internship, and I want to especially thank my advisor, *Prof. Julia Stoyanovich*, for giving me this great opportunity and for guiding me during these amazing five months. I also want to thank *Kunal Relia*, with whom working on this project was really great, and for helping me proofreading this report.

References

[1] Experiment *DessertPref*: https://dataresponsibly.hpc.nyu.edu/desertpref.php.

[2] Github of the project: https://github.com/theodlmz/dbcomsoc/.

[3] Amarilli, A., Ba, M. L., Deutch, D., and Senellart, P. Computing possible and certain answers over order-incomplete data. *Theoretical Computer Science* (2019).

[4] Baumeister, D., and Rothe, J. Taking the final step to a full dichotomy of the possible winner problem in pure scoring rules. *CoRR abs/1108.4436* (2011).

[5] Betzler, N., and Dorn, B. Towards a dichotomy for the possible winner problem in elections based on scoring rules. *Journal of Computer and System Sciences 76*, 8 (2010), 812 – 836.

[6] Brandt, F., Conitzer, V., Endriss, U., Lang, J., and Procaccia, A. D. *Handbook of Computational Social Choice*, 1st ed. Cambridge University Press, New York, NY, USA, 2016.

[7] Brightwell, G. Surveys in combinatorics, 1993. Cambridge University Press, New York, NY, USA, 1993, ch. Models of Random Partial Orders, pp. 53–83.

[8] Doignon, J.-P., Peke, A., and Regenwetter, M. The repeated insertion model for rankings: Missing link between two subset choice models. *Psychometrika 69* (03 2004), 33–54.

[9] Gehrlein, W. V. On methods for generating random partial orders. *Oper. Res. Lett. 5*, 6 (Dec. 1986), 285–291.

[10] Kimelfeld, B., Kolaitis, P. G., and Stoyanovich, J. Computational social choice meets databases. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence* (2018), IJCAI'18, AAAI Press, pp. 317–323.

[11] Kimelfeld, B., Kolaitis, P. G., and Tibi, M. Query evaluation in election databases. In *Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (New York, NY, USA, 2019), PODS '19, ACM, pp. 32–46.

[12] Xia, L., and Conitzer, V. Determining possible and necessary winners under common voting rules given partial orders. vol. 41, pp. 196–201.