Iterative Voting for Committee Selection with MAB

Harshitha Puttaswamy

December 18, 2024

1 Introduction

Iterative voting is a dynamic process in which voters adapt their preferences over multiple rounds of voting. All the voters can change their preferences in each round simultaneously. Initially, the voters start with their true preferences and change their ballots such that most preferred candidates are present in the winning committee hence maximizing their utility. Iterative voting presents special opportunities and challenges in the context of committee selection since voters can strategically modify their preferences to affect the chosen committee's conclusion. This study formulates the committee selection problem as a Multi-Armed Bandit (MAB) problem and investigates the use of iterative voting in this context. The purpose of the study is to examine how different committee selection procedures behave within this iterative framework to shed light on their effectiveness and voter satisfaction.

The process of voting consists of three main components, the set of voters $V = \{1, .., n\}$ set of candidates $C = \{c_1, .., c_m\}$, and the list of candidate preferences $P = \{p_1, .., p_n\}$ which is also known as preference profile. For further discussions we will refer to committee size as k, number of candidates as m, and number of voters as n. The list of candidates would be $[c_1, c_2, ..., c_m]$. The list of voters would be $[v_1, v_2, ..., v_n]$.

1.1 Contributions

Significant contributions of the study include analysis of iterative voting in committee selection in aspects of voter behavior, utility optimization, and the performance of voting rules. The contributions include:

Experimental Framework for Iterative Voting. Developed a systematic framework for simulating iterative voting scenarios. Implemented mechanisms to record shifting voter preferences and dynamic behavior over iterations.

Formulation as a Multi-Armed Bandit (MAB) Problem. Modeled voters as agents in an MAB setup, where candidates represent arms. Designed reward structures that align with voter utilities and preference satisfaction, enabling effective comparisons of strategies.

Implementation of Voting Rules. Implemented positional scoring rules (e.g., Plurality, Approval, Bloc) and Condorcet-consistent rules (e.g., Borda, Monroe). Incorporated Integer Linear Programming (ILP) formulations for complex rules like Proportional Approval Voting (PAV) and Chamberlin-Courant, ensuring optimal committee selection.

Extensive Experimental Analysis. Conducted experiments with diverse voter profiles, candidate sets, and voting rules. Analyzed results using key metrics such as average Borda utility scores, Kendall-Tau distance for preference stability, and cost of strategy for assessing strategic behavior.

1.2 Related Work

Iterative voting has been extensively studied in the context of single-winner elections. Iterative voting for single winner setting with different rules like Plurality, Borda are examined in [1]. In the experiments described, only one voter was allowed to change their ballot in the iterative process. The paper discusses results on borda scores and condorcet efficiency of these rules. Voting rules using Fully connected graph neural networks (GNN) and set transformers are studied in [2]. The convergence of various voting rules when different tie-breaking rules are used in iterative setting are analysed in [3]. All the above works focus on single winner setting. There are few papers on multi winner setting using iterative voting for plurality rule [4] but not much literature on other committee selection rules using iterative voting setting which is the major focus of this paper.

2 Preliminaries

The voting problem can be represented as a MAB problem where the voters are the agents and the candidates of an election are the arms that the agents will play. In the process of iterative voting, the voters first select their desired candidate and a winner is announced, later in each iteration a single voter changes their vote and the winner is announced. In the experiments discussed below the voters start to cast their votes based on their true preferences. Each voter's utility is calculated based on the voter's ballot and the winner is computed at the end of the iteration. The agents i.e., the voters try to maximize their utilities. The goal of the experiments is to learn how each voting rule behaves in the iterative voting setting and produce a comparative study of these rules.

2.1 Preferences

In an election, preferences serve as the primary means of expressing a voter's priorities and choices. They have a direct impact on the voting outcome and reflect the extent to which voters regard certain candidates. Two methods are commonly used to model preferences:

2.1.1 Ranked Preferences

In ranked preferences, each voter provides an ordered list of candidates, arranged from the most to the least preferred. This ranking reflects the relative importance of each candidate to the voter. For example, a voter might rank candidates as $[c_2, c_4, c_1, c_3, c_5]$, where c_2 is the most preferred candidate, and c_5 is the least preferred.

2.1.2 Approval Preferences

Approval preferences involve voters indicating which candidates they approve of, without specifying a rank order. Voters assign a binary decision (approve or disapprove) to each candidate. For instance, a voter might approve $[c_1, c_2]$ and disapprove of $[c_3, c_4, c_5]$. Approved candidates are treated equally, making this method simpler but less expressive than ranked preferences.

2.2 Committee selection rules.

Each voter's ballot is the rankings assigned to candidates based on their preference and is defined by a list of integers $R = \{1, .., m\}$. A candidate's ranking is influenced by the utilities i.e., a voter assigns a higher rank to a candidate with higher utility, the vector of voter utilities is defined as utility profile $u = \{u_1, .., u_n\}$. The voting rule is a function that maps preference profiles to candidates f : P $\leftrightarrow$ A.

Voting rules can be divided into two categories: Scoring based and Condorcet voting rules. In score-based voting rules like Plurality, and Approval, the candidates are assigned a score and the candidate with the highest score is selected as the winner whereas in Condorcet rules like Monroe, and Chamberlin Courant rules, the candidates are compared in pairs and the candidate who beats every other candidate wins the election.

2.2.1 Positional scoring rules

In a positional scoring rule, each voter ranks the candidates from most to least preferred. The rule assigns a score to each candidate based on their position in each voter's ranking. The total score of a candidate across all voters determines their collective ranking, and the candidate with the highest score is typically the winner.

Plurality rule. The Plurality score of candidate c is the number of votes where c is ranked first. The winning committee would be the top k candidates with the highest plurality score for the given voting iteration [6].

Anti-Plurality Rule. Anti-plurality voting describes an electoral system in which each voter votes against a single candidate, and the candidate with the fewest votes against wins [8]. Anti-plurality voting is an example of a positional voting method. The welfare function used is the same as the Plurality rule.

Approval Rule. In approval voting, the voters approve of a certain number of candidates and disapprove of the others. In the experiment, all winning committee combinations are computed with all the m candidates. For the given ballot in an iteration, if there is any approved candidate in the committee then that committee gets a score of one. In the end, the committee with the highest number of points is declared the winning committee [7].

Bloc Rule. In the Bloc Rule, each voter names his or her k favorite candidates (k approved candidates), and the winning committee consists of candidates with the highest k-approval scores [6].

Proportional Approval Voting Rule. The Proportional Approval Voting (PAV) rule is a voting system designed for multi-winner elections, where the goal is to select a group of candidates that maximizes voter satisfaction while ensuring proportional representation [7]. Proportional representation seeks to distribute representation fairly across different voter groups. Approval dictionary is computed based on the approvals of voters which contains binary values. This is passed to an ILP that is used to compute the winning committee if one exists.

Theorem 1 (ILP for PAV). *Given the committee size constarint, the voter assignment constraint, and the balance constraint, the ILP gives the committee with maximum Borda score.*

The proof of Theorem 1 is in the Appendix A.5.

The variables, constraints and objective of ILP are defined below.

Let x_i be the dictionary variable for candidate selection. S_{il} is defined as score dictionary.

$Variables:$

$$x_i \in \begin{cases} 1 & \text{if candidate } i \text{ is selected,} \\ 0 & \text{otherwise.} \end{cases}$$

$$\alpha_{ij} = \begin{cases} 1 & \text{if } i \text{ approves of } j, \\ 0 & \text{otherwise.} \end{cases}$$

$$a_i = \sum_j \alpha_{ij} \cdot x_j$$

$$s_{il} = \begin{cases} 1 & \text{if } a_i \geq l, \\ 0 & \text{if } a_i < l. \end{cases}$$

$$l \in \{1, 2, \ldots, k\}$$

$$\text{score}_i = \sum_{l=i}^{k} \frac{s_{il}}{l}$$

$Constraints:$

$$\sum_{i=1}^{m} x_i = k$$

$$\frac{a_i}{l} > s_{il} \geq \frac{a_i - l + 1}{k}$$

$Objective:$

$$\max \sum_i \text{score}_i$$

Single Transferable Vote Rule (STV). STV proceeds in rounds until k candidates are elected. A single round proceeds as follows: The voters cast their ballot (ranked preference of candidates). The most preferred candidate gets a score of 1, which is computed over all voters. The candidate with the lowest score is removed from the voting process for the current round, this step is reaped until k candidates are remaining hence forming the winning committee [6].

2.2.2 Condorcet consistent rules

A Condorcet winner is a candidate who, when compared in a head-to-head contest against every other candidate, is preferred by a majority of voters in each pairwise comparison. The Condorcet rule selects this candidate as the winner, if such a candidate exists.

Borda Rule. Each candidate is assigned a number of points from each ballot equal to the number of candidates to whom he or she is preferred, so that with n candidates, each one receives n–1 points for a first preference, n–2 for a second, and so on. This is called the Borda score of a candidate c [6]. The winner is the candidate with the highest Borda score. We consider candidates with the top k highest Borda score to get the winning committee.

Borda Chamberlin-Courant Rule ($\beta - CC$). In Borda Chamberlin-Courant Rule the score that a committee receives from a voter is the Borda score of the committee member that the voter ranks highest (among all the committee members) [7]. One possible interpretation is that each voter chooses a representative from the committee (clearly, a voter chooses the candidate that he or she likes the most) and gives the committee the Borda score of his or her representative.

An integer linear program is implemented to accomplish the task of finding a winning committee if one exists. A problem is defined in the ILP, it tries to output a winning committee within the given constraints.

Theorem 2 (**ILP for** $\beta - CC$)**.** *Given the committee size constarint and the voter assignment constraint, the ILP gives the committee with maximum Borda score.*

The proof of Theorem 2 is in the Appendix A.7.

The ILP is formulated below:
Let x_i be the variable for candidate selection. r_{ij} is defined as voter assignment.

$Variables:$

$$x_i \in \begin{cases} 1 & \text{if candidate } i \text{ is selected,} \\ 0 & \text{otherwise.} \end{cases}$$

$$r_{ij} \in \begin{cases} 1 & \text{if } i \text{ is highest ranked selected candidate,} \\ 0 & \text{otherwise.} \end{cases}$$

$Constraints:$

$$\sum_{i=1}^{m} x_i = k$$

$$\forall j \in n, \sum_{i \in m} r_{ij} = 1$$

$Objective:$

$$\max \left(\sum_{j=1}^{n} \sum_{i=1}^{m} (m - \sigma_{ij}) \cdot r_{ij} \right)$$

Monroe Rule. Monroe's voting rule is very similar to Chamberlin-Courant with an additional balance constraint for fairness [5]. ILP is used to compute the winning committee within the given constraints. All the constraints are the same as the Chamberlin-Courant with an additional balance constraint as defined below. The objective of the ILP is the same as Chamberlin-Courant.

Theorem 3 (**ILP for Monroe**)**.** *Given the committee size constarint, the voter assignment constraint, and the balance constraint, the ILP gives the committee with maximum Borda score.*

The proof of Theorem 3 is in the Appendix A.8.

Balance constraint:

$$\left\lfloor \frac{n}{k} \right\rfloor \leq \forall j \in n \sum_{i=1}^{m} r_{ij} \leq \left\lceil \frac{n}{k} \right\rceil$$

3 Methods

3.1 Multi-Arm Bandit model

Multi-armed bandits (MAB) are a class of decision-making problems where an agent must choose between multiple options (or ”arms”) over a series of rounds to maximize cumulative rewards. Each arm provides stochastic rewards, and the agent faces the trade-off between:
Exploration: Trying out different arms to gather information about their reward distributions.
Exploitation: Choosing the arm is believed to give the best reward based on past observations.
This trade-off is central to algorithms like epsilon-greedy, where the agent randomly explores with a probability ε and exploits the best-known arm otherwise.

3.2 Model Description

In the Multi-arm Bandit (MAB) setup we have Agents $N = \{1, .., n\}$ and Arms(preference profiles) $A = \{a_1, .., a_m\}$ where the agent repetitively pulls a desired arm and gets a reward for each play. An epsilon-greedy algorithm is implemented for the learning process. In each run, the training loop runs for the specified number of iterations. In each iteration the model is called to get votes, arms are pulled based on the exploration-exploitation criterion, ballots are returned to the calling function and based on the voting rule specified, the winning committee is computed using that voting rule. The winning committee is used to compute the welfare of voters, the welfare function depends upon the voting rule being used. This welfare score is updated for each voter as their reward and is averaged over iterations. The rewards are summed up and averaged over all voters to get the average utility score.

Algorithm 1 Epsilon-Greedy Algorithm

```
0: **Input:** $\epsilon$, Number of iterations $T$, Number of actions $K$
0: **Initialize:** Action-value estimates $Q(a) \leftarrow 0 \; \forall a \in \{1, \ldots, K\}$
0: Action counts $N(a) \leftarrow 0 \; \forall a \in \{1, \ldots, K\}$
0: for $t = 1$ to $T$ do
0:     Generate a random number $r \sim U(0, 1)$
0:     if $r < \epsilon$ then {Explore}
0:         Choose an action $a_t$ randomly with equal probability
0:     else{Exploit}
0:         Choose action $a_t = \arg\max_a Q(a)$
0:     end if
0:     Observe reward $R_t$ for action $a_t$
0:     Update counts: $N(a_t) \leftarrow N(a_t) + 1$
0:     Update action-value estimate:
           $$Q(a_t) \leftarrow Q(a_t) + \frac{1}{N(a_t)}\big(R_t - Q(a_t)\big)$$
0: end for
0: **Output:** Estimated action-values $Q(a)$ =0
```

3.3 Action Space

The action space in MAB is the available set of arms (options) that can be chosen, essentially representing the different choices the agent can make. In the voting process, the different choices are the types of ballots a voter can cast, these are discussed below.

Plurality ballots - The options available are individual candidates, the voter's ballot consists of a single candidate. This is used as action space for $Plurality$ and *Anti-Plurality* voting rules.

Approval ballots - Each voter can select (approve) one or more candidates or options without ranking them. We assign 1 to a candidate who is approved by the voter and 0 otherwise. The choices available are all combinations of 0's and 1's with a specified approval count value which limits the number of candidates that a voter can approve. $Approval$, $Bloc$, PAV voting uses Approval ballots as it's action space.

Ranked ballots - The voter ranks candidates or options in order of preference, instead of just selecting one or multiple options. The action space consists of permutations of all candidates. The voting rules using this are $Borda$, $Monore$, $\beta - CC$, and STV.

3.4 Reward space

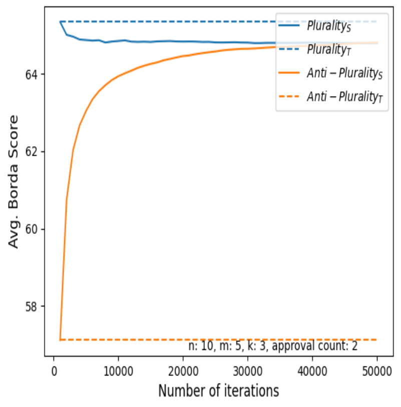
The reward space is the range of possible rewards received after selecting an arm. It is a numerical value representing the outcome of that action. In the iterative voting problem, the reward that the voters experience is the Borda score of the winning committee. The winning committee is computed by looking at the voter ballots in that iteration of the process. The cumulative Borda score of the candidates in the winning committee is computed w.r.t. the true preferences of the voters.

4 Results

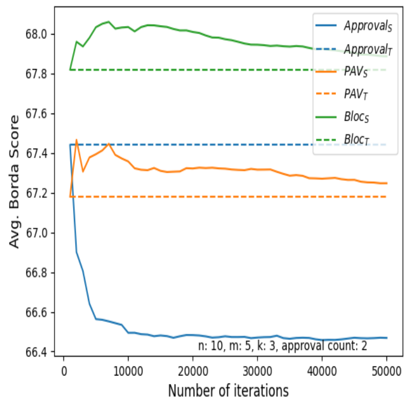
Different metrics are discussed in this section. The report talks about the utility scores for different voting rules, Kendall-Tau distance at convergence, and the cost of strategy of voting rules which are discussed in detail below. The experiments are averaged over 50 runs for 50000 iterations in each run. All the experiments are run with 10 voters, 5 candidates, an approval count of 2, and a committee size of 3.

4.1 Average Borda utility score

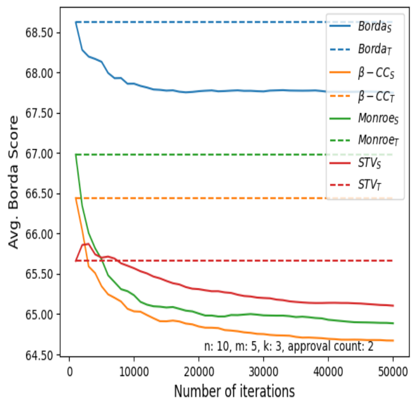
The average Borda utility score is the average reward that the voters get from their ballot cast at every iteration. The winning committee and welfare function (Borda score) decides the rewards that the voters experience, based on these rewards the voters learn how to maximize their utility and change their ballot as the training occurs. At convergence, we check if the voter has learned to vote for the candidates that give them the highest reward. The results for different voting rules are attached below.



(a) Rules based on Plurality ballot



(b) Rules based on approval ballot



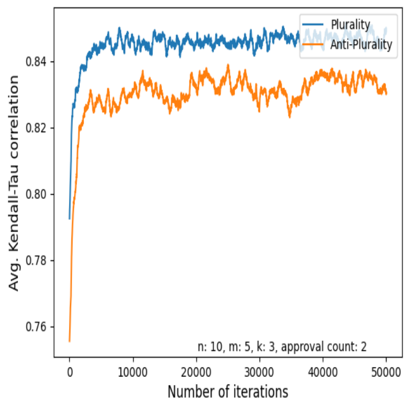
(c) Rules based on Ranked ballot

Figure 1: Average Borda utility score

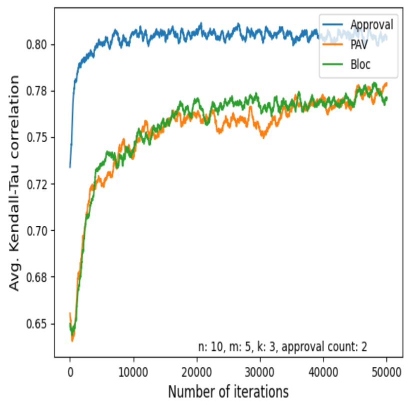
4.2 Kendall-Tau distance

The Kendall tau rank distance is a metric (distance function) that counts the number of pairwise disagreements between two ranking lists. The larger the distance, the more dissimilar the two lists are.

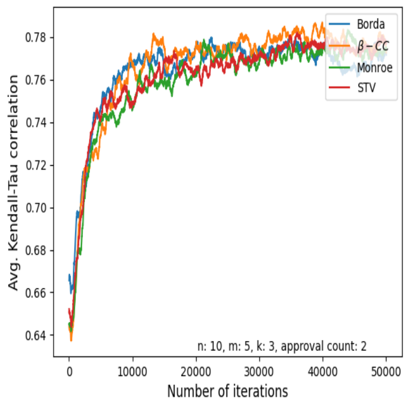
The distance between the winning committee at iteration t and at iterations $[t-10, t-9, ..., t-1]$ are computed, and their average is calculated. In the experiments, the Kendall-tau distance can range between 0, 1. If the value is closer to 1, it implies a strong positive ordinal correlation, 0 implies a weak ordinal correlation, and -1 implies a strong negative ordinal correlation. This gives valuable information regarding the changing of ballots by the voters, if the KT distance is closer to 1 then it implies that the voters are satisfied with their ballot.



(a) Rules based on Plurality ballot



(b) Rules based on approval ballot

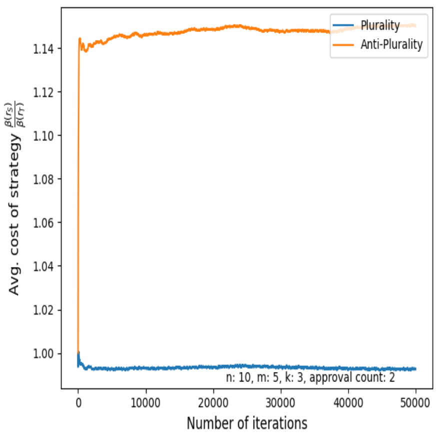


(c) Rules based on Ranked ballot

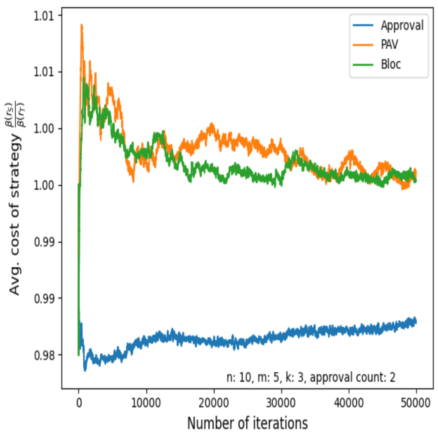
Figure 2: Average Kendall-Tau distance

4.3 Cost of strategy

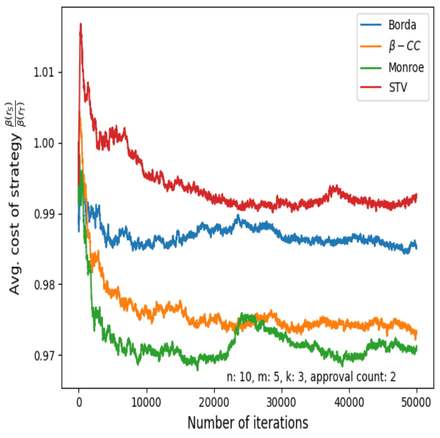
The cost of strategy is computed by taking the ratio between the utility score at nth and $0th$. If the ratio is greater than one then it would mean that the voters have achieved higher utility with strategic behavior when compared to their true preference. If the cost of the strategy is less than one then it would mean that at convergence the average utility score of the voters was lesser than the utility score with their true preference.



(a) Rules based on Plurality ballot



(b) Rules based on approval ballot



(c) Rules based on Ranked ballot

Figure 3: Average cost of strategy $\frac{\beta(r_S)}{\beta(r_T)}$

References

[1] Airiau, Stéphane, Umberto Grandi, and Filipo Studzinski Perotto. ”Learning agents for iterative voting.” Algorithmic Decision Theory: 5th International Conference, ADT 2017, Luxembourg, Luxembourg, October 25–27, 2017, Proceedings 5. Springer International Publishing, 2017.

[2] Anil, Cem, and Xuchan Bao. ”Learning to elect.” Advances in Neural Information Processing Systems 34 (2021): 8006-8017.

[3] Lev, Omer, and Jeffrey S. Rosenschein. ”Convergence of iterative voting.” Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems-Volume 2. 2012.

[4] Sultan, Saira, et al. ”Convergence of Iterative Voting Under Restrictive Dynamics for Plurality Rule: A Review.” 2018 IEEE 21st International Multi-Topic Conference (INMIC). IEEE, 2018.

[5] Lackner, Martin, and Piotr Skowron. ”Approval-Based Multi-Winner Rules and Strategic Voting.” IJCAI. 2018.

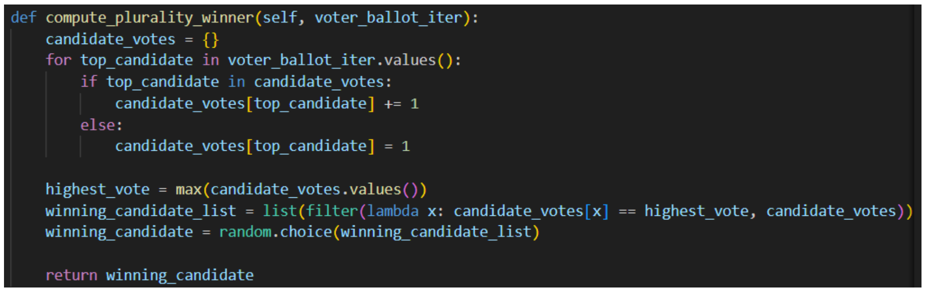
[6] Elkind, Edith, et al. ”Properties of multiwinner voting rules.” Social Choice and Welfare 48 (2017): 599-632.

[7] Endriss, Ulle. Trends in computational social choice. Lulu. com, 2017.

[8] Diss, Mostapha, Eric Kamwa, and Abdelmonaim Tlidi. ”On some k-scoring rules for committee elections: agreement and Condorcet Principle.” Revue d’économie politique 130.5 (2020): 699-725.

A Implementation of Voting Rules

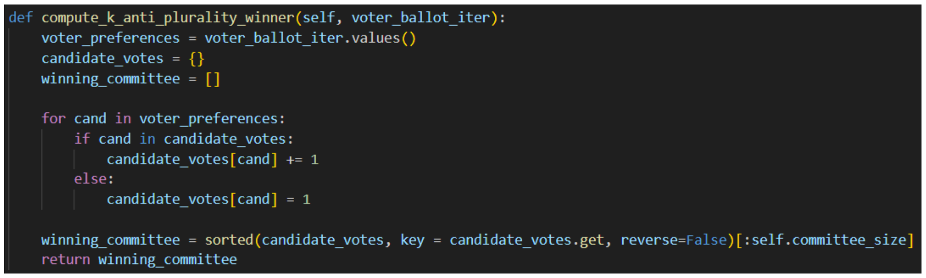
A.1 Plurality rule



```
def compute_plurality_winner(self, voter_ballot_iter):
    candidate_votes = {}
    for top_candidate in voter_ballot_iter.values():
        if top_candidate in candidate_votes:
            candidate_votes[top_candidate] += 1
        else:
            candidate_votes[top_candidate] = 1

    highest_vote = max(candidate_votes.values())
    winning_candidate_list = list(filter(lambda x: candidate_votes[x] == highest_vote, candidate_votes))
    winning_candidate = random.choice(winning_candidate_list)

    return winning_candidate
```

Figure 4: Compute winning committee with plurality rule

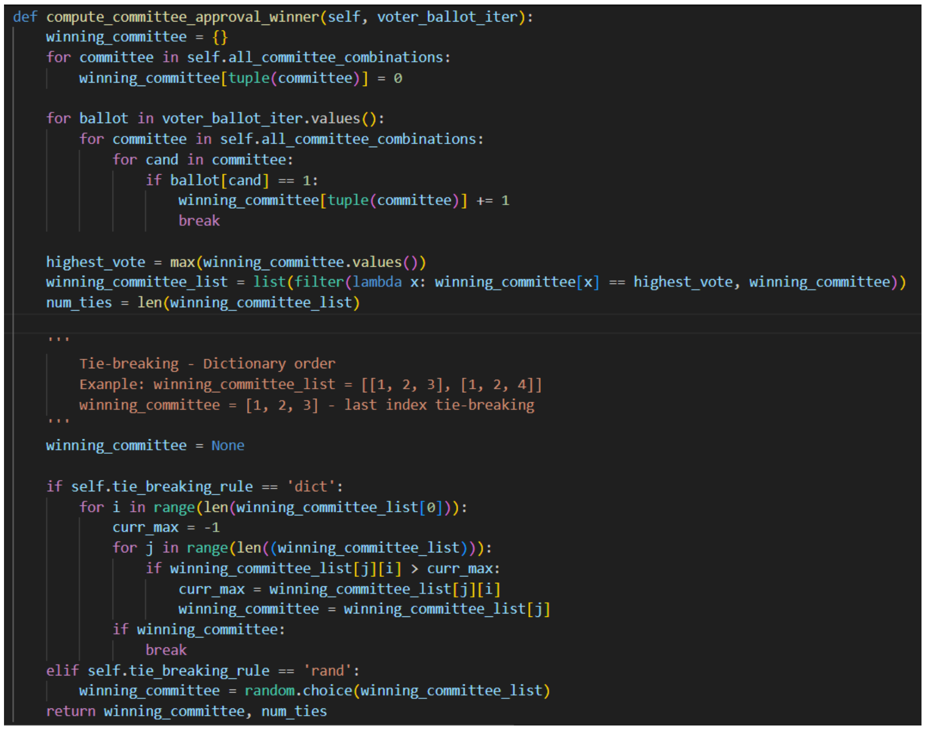
A.2 Anti-Plurality rule



```python
def compute_k_anti_plurality_winner(self, voter_ballot_iter):
    voter_preferences = voter_ballot_iter.values()
    candidate_votes = {}
    winning_committee = []

    for cand in voter_preferences:
        if cand in candidate_votes:
            candidate_votes[cand] += 1
        else:
            candidate_votes[cand] = 1

    winning_committee = sorted(candidate_votes, key = candidate_votes.get, reverse=False)[:self.committee_size]
    return winning_committee
```

Figure 5: Compute winning committee using anti-plurality rule

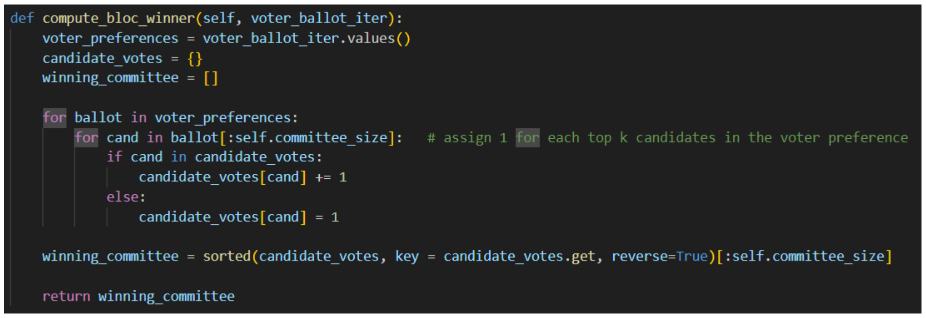
A.3 Approval rule



```python
def compute_committee_approval_winner(self, voter_ballot_iter):
    winning_committee = {}
    for committee in self.all_committee_combinations:
        winning_committee[tuple(committee)] = 0

    for ballot in voter_ballot_iter.values():
        for committee in self.all_committee_combinations:
            for cand in committee:
                if ballot[cand] == 1:
                    winning_committee[tuple(committee)] += 1
                    break

    highest_vote = max(winning_committee.values())
    winning_committee_list = list(filter(lambda x: winning_committee[x] == highest_vote, winning_committee))
    num_ties = len(winning_committee_list)

    '''
        Tie-breaking - Dictionary order
        Exanple: winning_committee_list = [[1, 2, 3], [1, 2, 4]]
        winning_committee = [1, 2, 3] - last index tie-breaking
    '''
    winning_committee = None

    if self.tie_breaking_rule == 'dict':
        for i in range(len(winning_committee_list[0])):
            curr_max = -1
            for j in range(len((winning_committee_list))):
                if winning_committee_list[j][i] > curr_max:
                    curr_max = winning_committee_list[j][i]
                    winning_committee = winning_committee_list[j]
            if winning_committee:
                break
    elif self.tie_breaking_rule == 'rand':
        winning_committee = random.choice(winning_committee_list)
    return winning_committee, num_ties
```

Figure 6: Compute winning committee using Approval voting Rule

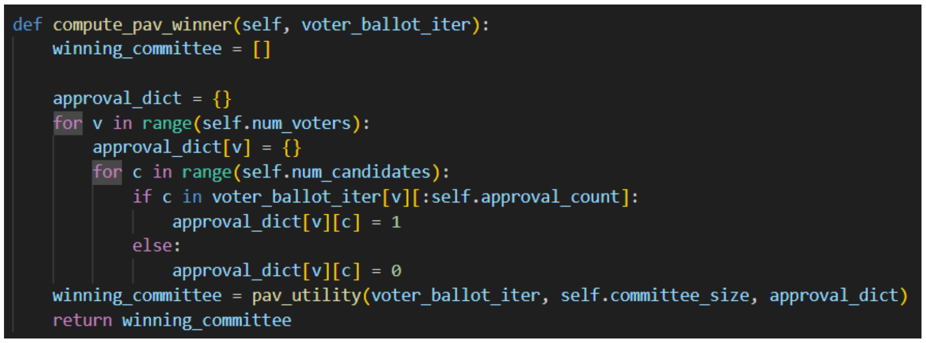
A.4 Bloc Rule



```
def compute_bloc_winner(self, voter_ballot_iter):
    voter_preferences = voter_ballot_iter.values()
    candidate_votes = {}
    winning_committee = []

    for ballot in voter_preferences:
        for cand in ballot[:self.committee_size]:   # assign 1 for each top k candidates in the voter preference
            if cand in candidate_votes:
                candidate_votes[cand] += 1
            else:
                candidate_votes[cand] = 1

    winning_committee = sorted(candidate_votes, key = candidate_votes.get, reverse=True)[:self.committee_size]

    return winning_committee
```

Figure 7: Compute winning committee using Bloc rule

A.5 Proportional Approval Voting Rule



```
def compute_pav_winner(self, voter_ballot_iter):
    winning_committee = []

    approval_dict = {}
    for v in range(self.num_voters):
        approval_dict[v] = {}
        for c in range(self.num_candidates):
            if c in voter_ballot_iter[v][:self.approval_count]:
                approval_dict[v][c] = 1
            else:
                approval_dict[v][c] = 0
    winning_committee = pav_utility(voter_ballot_iter, self.committee_size, approval_dict)
    return winning_committee
```

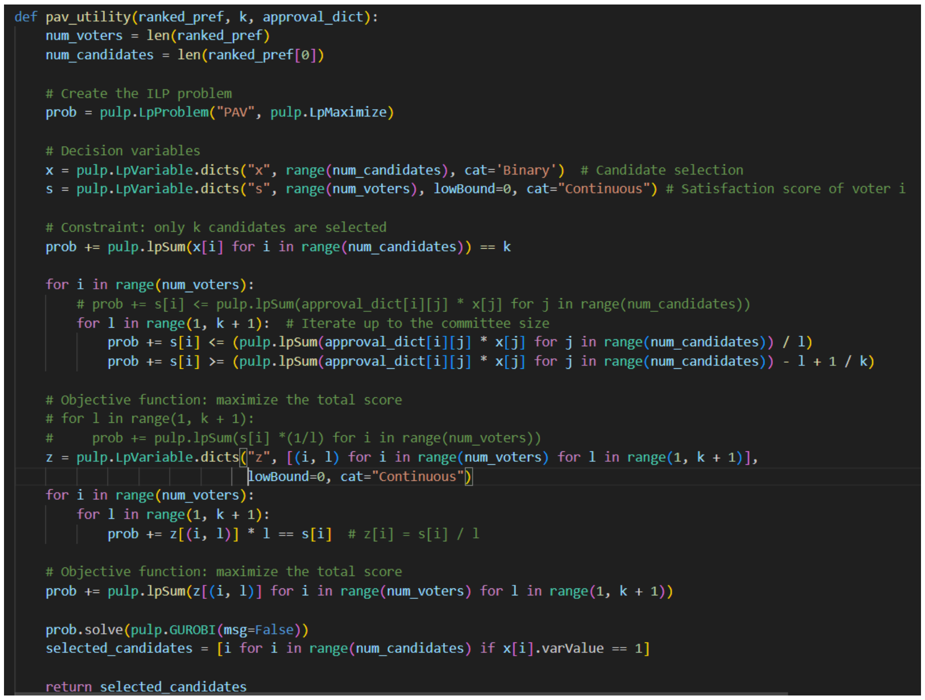
Figure 8: Compute winning committee using PAV

Theorem 1 (ILP for PAV). *Given the committee size constarint, the voter assignment constraint, and the balance constraint, the ILP gives the committee with maximum Borda score.*

Proof. We proceed by contradiction.
Assumption: Let's assume that it is possible to achieve a higher objective value than the one given by the current solution. This would imply that for some candidate i, the Borda score can be increased. The objective function depends on the scores s_{il}, which are defined as: $s_{il} = \begin{cases} 1 & \text{if } a_i \geq l, \\ 0 & \text{if } a_i < l. \end{cases}$
For a particular value of l if a candidate i must have a higher score then the total approval count a_i must increase.
This can be achieved by selecting candidate j such that $\alpha_{ij} = 1$, i.e., voter i approved of candidate j. However, the committee size constraint limits how many candidates can be selected which in turn limits the maximum possible value of a_i. If we try to

increase a_i too much, it could force us to select more than k candidates or violate the approval structure.
Hence the assumption that a committee with higher Borda score can be achieved by painting all the constraints in incorrect.

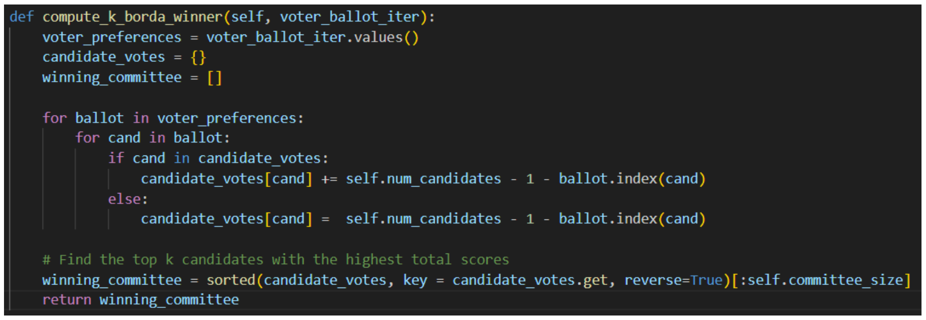
□



```python
def pav_utility(ranked_pref, k, approval_dict):
    num_voters = len(ranked_pref)
    num_candidates = len(ranked_pref[0])

    # Create the ILP problem
    prob = pulp.LpProblem("PAV", pulp.LpMaximize)

    # Decision variables
    x = pulp.LpVariable.dicts("x", range(num_candidates), cat='Binary')  # Candidate selection
    s = pulp.LpVariable.dicts("s", range(num_voters), lowBound=0, cat="Continuous") # Satisfaction score of voter i

    # Constraint: only k candidates are selected
    prob += pulp.lpSum(x[i] for i in range(num_candidates)) == k

    for i in range(num_voters):
        # prob += s[i] <= pulp.lpSum(approval_dict[i][j] * x[j] for j in range(num_candidates))
        for l in range(1, k + 1):  # Iterate up to the committee size
            prob += s[i] <= (pulp.lpSum(approval_dict[i][j] * x[j] for j in range(num_candidates)) / l)
            prob += s[i] >= (pulp.lpSum(approval_dict[i][j] * x[j] for j in range(num_candidates)) - l + 1 / k)

    # Objective function: maximize the total score
    # for l in range(1, k + 1):
    #     prob += pulp.lpSum(s[i] *(1/l) for i in range(num_voters))
    z = pulp.LpVariable.dicts("z", [(i, l) for i in range(num_voters) for l in range(1, k + 1)],
                              lowBound=0, cat="Continuous")
    for i in range(num_voters):
        for l in range(1, k + 1):
            prob += z[(i, l)] * l == s[i]  # z[i] = s[i] / l

    # Objective function: maximize the total score
    prob += pulp.lpSum(z[(i, l)] for i in range(num_voters) for l in range(1, k + 1))

    prob.solve(pulp.GUROBI(msg=False))
    selected_candidates = [i for i in range(num_candidates) if x[i].varValue == 1]

    return selected_candidates
```

Figure 9: Integer linear program to compute PAV winning committee

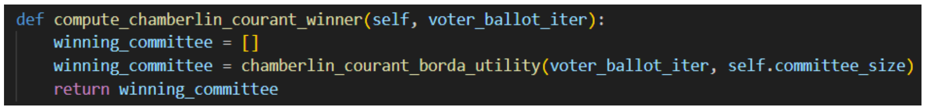
A.6 Borda rule



```python
def compute_k_borda_winner(self, voter_ballot_iter):
    voter_preferences = voter_ballot_iter.values()
    candidate_votes = {}
    winning_committee = []

    for ballot in voter_preferences:
        for cand in ballot:
            if cand in candidate_votes:
                candidate_votes[cand] += self.num_candidates - 1 - ballot.index(cand)
            else:
                candidate_votes[cand] =  self.num_candidates - 1 - ballot.index(cand)

    # Find the top k candidates with the highest total scores
    winning_committee = sorted(candidate_votes, key = candidate_votes.get, reverse=True)[:self.committee_size]
    return winning_committee
```

Figure 10: Compute winning committee using Borda

A.7 Borda Chamberlin-Courant Rule

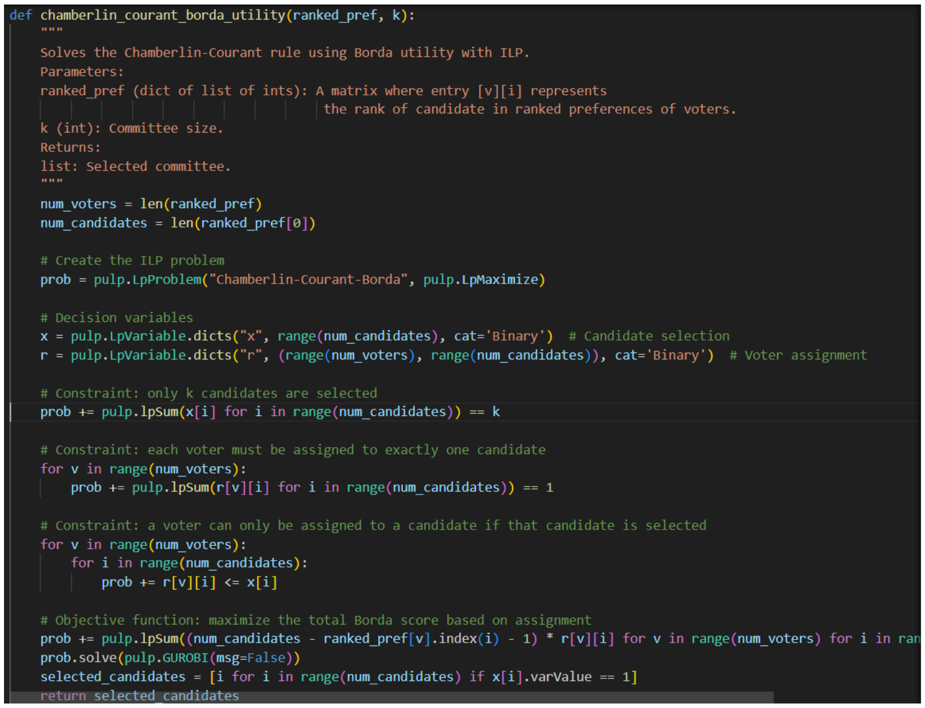


```python
def compute_chamberlin_courant_winner(self, voter_ballot_iter):
    winning_committee = []
    winning_committee = chamberlin_courant_borda_utility(voter_ballot_iter, self.committee_size)
    return winning_committee
```

Figure 11: Compute winning committee using Chamberlin-Courant Rule

Theorem 2 (**ILP for** $\beta - CC$)**.** *Given the committee size constarint and the voter assignment constraint, the ILP gives the committee with maximum Borda score.*

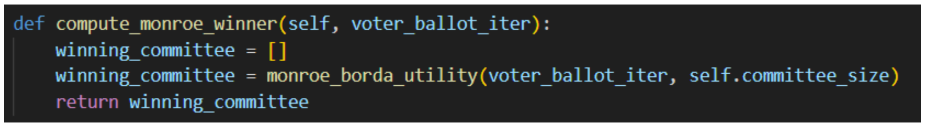
Proof. We proceed by contradiction.
Assumption: Assume that we can achieve a higher Borda score while satisfying the committee size constraint $\sum_{i=1}^{m} x_i = k$.
To maximize the Borda score, we want to select candidates that are ranked highly by the voters, i.e., those with the least σ_{ij} value. This means we want the highest-ranked candidates (in terms of the voters' preferences) to be selected, and each voter should assign $r_{ij} = 1$ to one of the selected candidates.
Hence if the Borda score needs to be increased then more than k candidates must be selected. This contradicts our assumption that a higher Borda score can be achieved while satisfying the committee size constraint. □



```python
def chamberlin_courant_borda_utility(ranked_pref, k):
    """
    Solves the Chamberlin-Courant rule using Borda utility with ILP.
    Parameters:
    ranked_pref (dict of list of ints): A matrix where entry [v][i] represents
                                        the rank of candidate in ranked preferences of voters.
    k (int): Committee size.
    Returns:
    list: Selected committee.
    """
    num_voters = len(ranked_pref)
    num_candidates = len(ranked_pref[0])

    # Create the ILP problem
    prob = pulp.LpProblem("Chamberlin-Courant-Borda", pulp.LpMaximize)

    # Decision variables
    x = pulp.LpVariable.dicts("x", range(num_candidates), cat='Binary')  # Candidate selection
    r = pulp.LpVariable.dicts("r", (range(num_voters), range(num_candidates)), cat='Binary')  # Voter assignment

    # Constraint: only k candidates are selected
    prob += pulp.lpSum(x[i] for i in range(num_candidates)) == k

    # Constraint: each voter must be assigned to exactly one candidate
    for v in range(num_voters):
        prob += pulp.lpSum(r[v][i] for i in range(num_candidates)) == 1

    # Constraint: a voter can only be assigned to a candidate if that candidate is selected
    for v in range(num_voters):
        for i in range(num_candidates):
            prob += r[v][i] <= x[i]

    # Objective function: maximize the total Borda score based on assignment
    prob += pulp.lpSum((num_candidates - ranked_pref[v].index(i) - 1) * r[v][i] for v in range(num_voters) for i in ran
    prob.solve(pulp.GUROBI(msg=False))
    selected_candidates = [i for i in range(num_candidates) if x[i].varValue == 1]
    return selected_candidates
```

Figure 12: Integer linear program to compute Chamberlin-Courant winning committee

A.8 Monroe Rule



```python
def compute_monroe_winner(self, voter_ballot_iter):
    winning_committee = []
    winning_committee = monroe_borda_utility(voter_ballot_iter, self.committee_size)
    return winning_committee
```

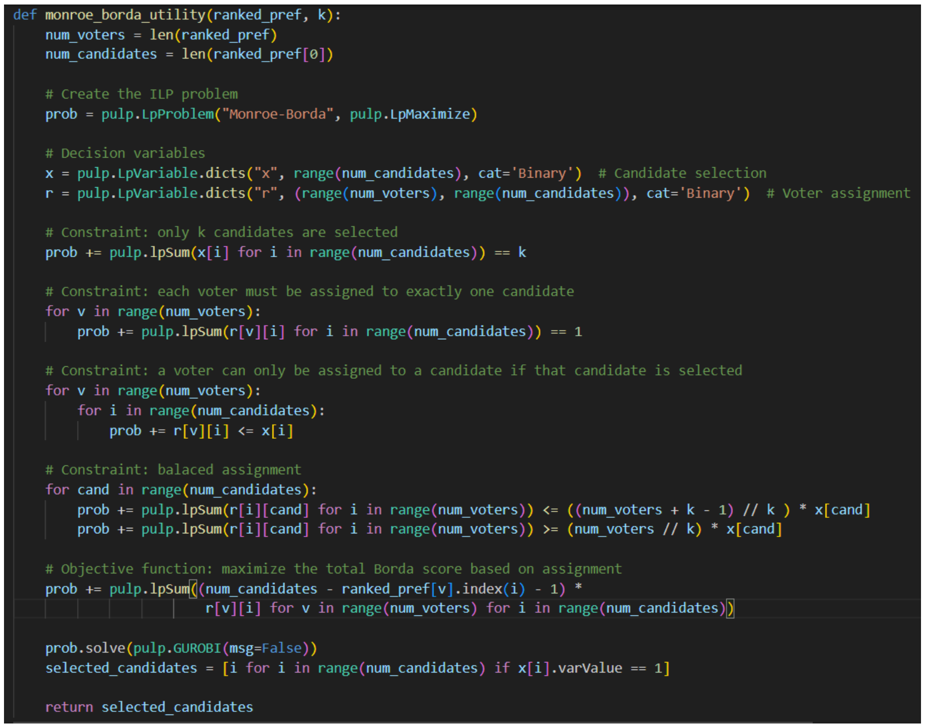
Figure 13: Compute winning committee using Monroe

Theorem 3 (ILP for Monroe). *Given the committee size constarint, the voter assignment constraint, and the balance constraint, the ILP gives the committee with maximum Borda score.*

Proof. The proof is similar to that of $\beta - CC$. We proceed by contradiction.
Assumption: Assume that we can achieve a higher Borda score while satisfying the committee size constraint $\sum_{i=1}^{m} x_i = k$ and balance constraint $\left\lfloor \frac{n}{k} \right\rfloor \leq \forall j \in n \sum_{i=1}^{m} r_{ij} \leq \left\lceil \frac{n}{k} \right\rceil$.

If the alternative solution seeks to increase the score by allocating considerably more voters to specific top-ranked candidates, the balance constraint would be violated. Assigning more than $\left\lfloor \frac{n}{k} \right\rfloor$ voters to a candidate would increase their score but break the balance constraint. Hence the assumption is incorrect.

□



```python
def monroe_borda_utility(ranked_pref, k):
    num_voters = len(ranked_pref)
    num_candidates = len(ranked_pref[0])

    # Create the ILP problem
    prob = pulp.LpProblem("Monroe-Borda", pulp.LpMaximize)

    # Decision variables
    x = pulp.LpVariable.dicts("x", range(num_candidates), cat='Binary')  # Candidate selection
    r = pulp.LpVariable.dicts("r", (range(num_voters), range(num_candidates)), cat='Binary')  # Voter assignment

    # Constraint: only k candidates are selected
    prob += pulp.lpSum(x[i] for i in range(num_candidates)) == k

    # Constraint: each voter must be assigned to exactly one candidate
    for v in range(num_voters):
        prob += pulp.lpSum(r[v][i] for i in range(num_candidates)) == 1

    # Constraint: a voter can only be assigned to a candidate if that candidate is selected
    for v in range(num_voters):
        for i in range(num_candidates):
            prob += r[v][i] <= x[i]

    # Constraint: balaced assignment
    for cand in range(num_candidates):
        prob += pulp.lpSum(r[i][cand] for i in range(num_voters)) <= ((num_voters + k - 1) // k ) * x[cand]
        prob += pulp.lpSum(r[i][cand] for i in range(num_voters)) >= (num_voters // k) * x[cand]

    # Objective function: maximize the total Borda score based on assignment
    prob += pulp.lpSum((num_candidates - ranked_pref[v].index(i) - 1) *
                        r[v][i] for v in range(num_voters) for i in range(num_candidates))

    prob.solve(pulp.GUROBI(msg=False))
    selected_candidates = [i for i in range(num_candidates) if x[i].varValue == 1]

    return selected_candidates
```

Figure 14: Integer linear program to compute Monroe winning committee

A.9 Single Transferable Vote Rule

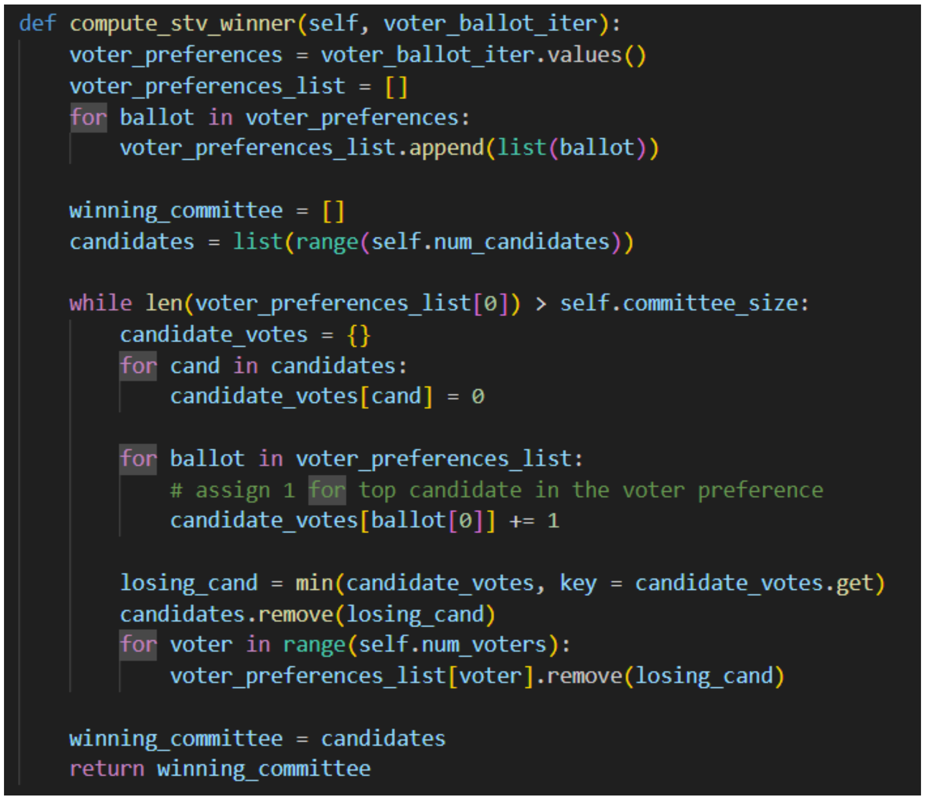
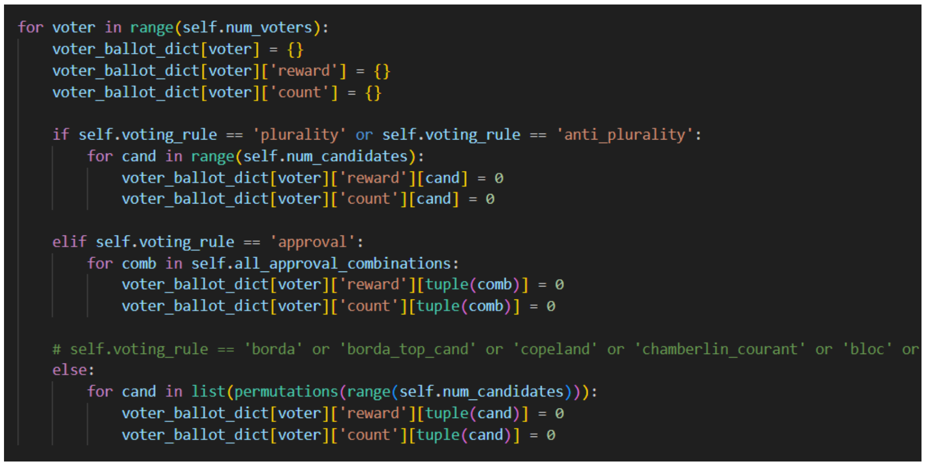

```python
def compute_stv_winner(self, voter_ballot_iter):
    voter_preferences = voter_ballot_iter.values()
    voter_preferences_list = []
    for ballot in voter_preferences:
        voter_preferences_list.append(list(ballot))

    winning_committee = []
    candidates = list(range(self.num_candidates))

    while len(voter_preferences_list[0]) > self.committee_size:
        candidate_votes = {}
        for cand in candidates:
            candidate_votes[cand] = 0

        for ballot in voter_preferences_list:
            # assign 1 for top candidate in the voter preference
            candidate_votes[ballot[0]] += 1

        losing_cand = min(candidate_votes, key = candidate_votes.get)
        candidates.remove(losing_cand)
        for voter in range(self.num_voters):
            voter_preferences_list[voter].remove(losing_cand)

    winning_committee = candidates
    return winning_committee
```

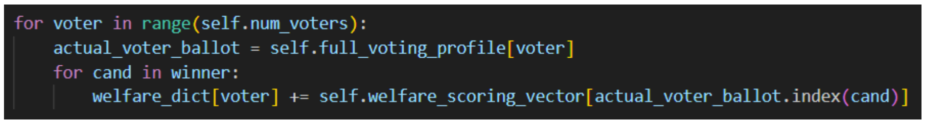
Figure 15: Compute winning committee using STV

A.10 Action and Reward Space



```
for voter in range(self.num_voters):
    voter_ballot_dict[voter] = {}
    voter_ballot_dict[voter]['reward'] = {}
    voter_ballot_dict[voter]['count'] = {}

    if self.voting_rule == 'plurality' or self.voting_rule == 'anti_plurality':
        for cand in range(self.num_candidates):
            voter_ballot_dict[voter]['reward'][cand] = 0
            voter_ballot_dict[voter]['count'][cand] = 0

    elif self.voting_rule == 'approval':
        for comb in self.all_approval_combinations:
            voter_ballot_dict[voter]['reward'][tuple(comb)] = 0
            voter_ballot_dict[voter]['count'][tuple(comb)] = 0

    # self.voting_rule == 'borda' or 'borda_top_cand' or 'copeland' or 'chamberlin_courant' or 'bloc' or
    else:
        for cand in list(permutations(range(self.num_candidates))):
            voter_ballot_dict[voter]['reward'][tuple(cand)] = 0
            voter_ballot_dict[voter]['count'][tuple(cand)] = 0
```

Figure 16: Voter ballot types (action space) for voting rules

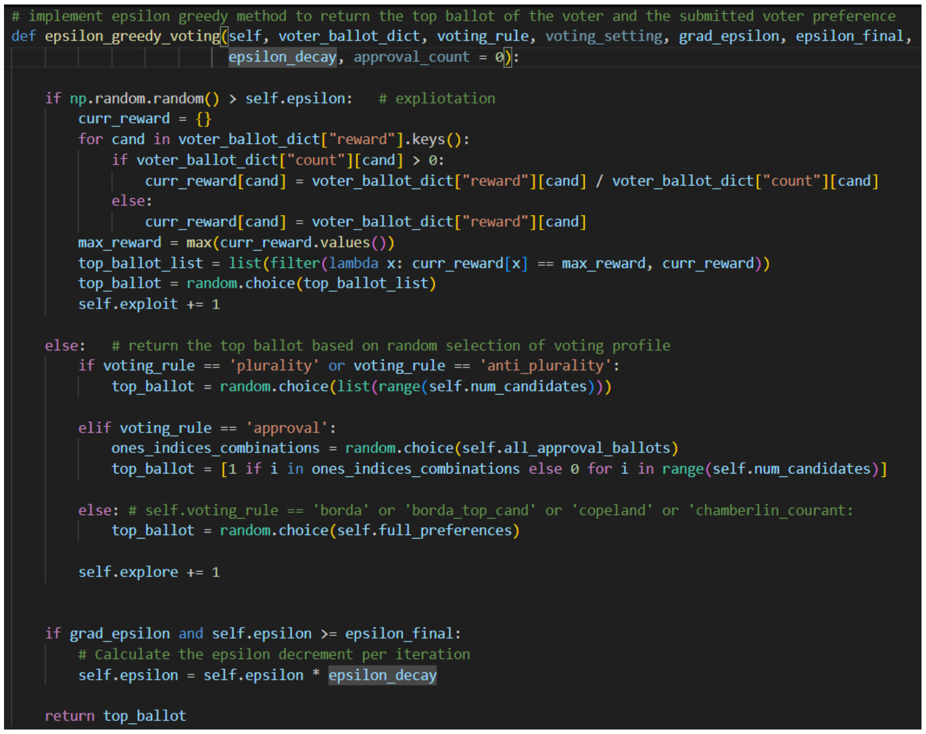


```
for voter in range(self.num_voters):
    actual_voter_ballot = self.full_voting_profile[voter]
    for cand in winner:
        welfare_dict[voter] += self.welfare_scoring_vector[actual_voter_ballot.index(cand)]
```

Figure 17: Reward function for voters

A.11 MAB Model Inputs

$voter\_ballot\_dict$: Contains data about rewards and counts for candidates:
$voter\_ballot\_dict[''reward'']$: Tracks cumulative rewards for each candidate.
$voter\_ballot\_dict[''count'']$: Tracks the number of times each candidate was selected.
$voting\_rule$: Specifies the voting method (e.g., plurality, approval).
$voting\_setting$: Context or additional settings for the voting process.
$grad\_epsilon$, $epsilon\_final$, $epsilon\_decay$: Parameters controlling epsilon's decay over iterations.
$approval\_count$: Optional parameter (default = 0), number of candidates that can be approved.



```python
# implement epsilon greedy method to return the top ballot of the voter and the submitted voter preference
def epsilon_greedy_voting(self, voter_ballot_dict, voting_rule, voting_setting, grad_epsilon, epsilon_final,
                          epsilon_decay, approval_count = 0):

    if np.random.random() > self.epsilon:   # expliotation
        curr_reward = {}
        for cand in voter_ballot_dict["reward"].keys():
            if voter_ballot_dict["count"][cand] > 0:
                curr_reward[cand] = voter_ballot_dict["reward"][cand] / voter_ballot_dict["count"][cand]
            else:
                curr_reward[cand] = voter_ballot_dict["reward"][cand]
        max_reward = max(curr_reward.values())
        top_ballot_list = list(filter(lambda x: curr_reward[x] == max_reward, curr_reward))
        top_ballot = random.choice(top_ballot_list)
        self.exploit += 1

    else:   # return the top ballot based on random selection of voting profile
        if voting_rule == 'plurality' or voting_rule == 'anti_plurality':
            top_ballot = random.choice(list(range(self.num_candidates)))

        elif voting_rule == 'approval':
            ones_indices_combinations = random.choice(self.all_approval_ballots)
            top_ballot = [1 if i in ones_indices_combinations else 0 for i in range(self.num_candidates)]

        else: # self.voting_rule == 'borda' or 'borda_top_cand' or 'copeland' or 'chamberlin_courant:
            top_ballot = random.choice(self.full_preferences)

        self.explore += 1


    if grad_epsilon and self.epsilon >= epsilon_final:
        # Calculate the epsilon decrement per iteration
        self.epsilon = self.epsilon * epsilon_decay

    return top_ballot
```

Figure 18: MAB model function