1 Voting Rules and Manipulability [70 marks]

This question deals with different voting mechanisms and how they fare against each other in terms of the average rank of the winner and the manipulability of the voting rule.

1.1 Voting Rules

You have to implement *four* different voting mechanisms, which were discussed during class in detail. **For each rule, if there is a tie, then the candidate with the lowest index amongst the tied candidates is declared the winner.** Suppose there are n voters and m candidates.

1. **Plurality** - Choose the candidate that is preferred by the most number of voters as the winner. Your code should run in $\mathcal{O}(m+n)$ time complexity.
2. **Borda's rule** - The i$^{\text{th}}$ preference of a voter is given a Borda score of $m-i$ (where $1 \leqslant i \leqslant m$), and these scores are summed up for each candidate across all voters. The candidate with the highest Borda score wins. Your code should run in $\mathcal{O}(mn)$ time complexity.
3. **Single Transferable Vote (STV)** - This mechanism proceeds in $m-1$ rounds, such that exactly one candidate is eliminated from all the preference profiles in each round. The candidate with the lowest plurality score, i.e., who is the most preferred candidate of the least number of voters, is chosen as the eliminating candidate. If there is a tie among the candidates with the lowest plurality score, then eliminate the one with a smaller index. Your code should run in $\mathcal{O}(m^2n)$ time complexity.
4. **Copeland's method** - This rule entails finding the pairwise winners among every 2 candidates. In other words, if a and b are two candidates, such that a is better than b in x elections and b is better than a in y elections $(x+y=n)$, then we say that a wins against b if $x>y$, and b wins against a if $y>x$, and the winner adds 1 point to their Copeland score. Also, if $x=y$, then both candidates get 0.5 points each. The candidate with the maximum Copeland score is declared the winner. Your code should run in $\mathcal{O}(m^2n)$ time complexity.

The suggested time complexities are not a hard rule that has to be followed, but will help ensure that your submission doesn't time out, as we will check the functions on multiple (and possibly larger) profiles. After implementing each strategy, you also need to find the average rank of the winner, defined as follows:

Suppose the winner for some voting rule is candidate c, and candidate c has rank r_i in voter i's preference list $(1 \leqslant i \leqslant n)$. Here, r_i is an integer from 1 to m, with 1 being the rank of the most preferred candidate for voter i. Then the average rank of the winner is given by $\frac{1}{n}\sum_{i=1}^{n} r_i$.

1.2 Manipulability of voting rules

Your next task is to check if a voting rule is manipulable for the given preferences of each voter. You can check the definition of manipulability in the class notes. For this task, it is guaranteed that there will be at most 8 candidates when checking if a preference profile is manipulable or not, and that there will be more voters than candidates.

1.3 Tasks

You have the following tasks to complete in `q1.py`:

1. In `find_winner(profiles, voting_rule)`, return the index of the winning candidate (1-indexed) for the given voting rule. `profiles` is a 2D NumPy array where row i denotes the strict preference order of voter i, and `voting_rule` can be one of `plurality`, `borda`, `stv`, or `copeland` (marks are assigned in this order). [8 + 12 + 15 + 15 = 50 marks]

2. In `find_winner_average_rank(profiles, voting_rule)`, find the average rank of the winner as per the given voting rule. [5 marks]

3. In `check_manipulable(profiles, voting_rule)`, return a boolean representing whether the given voting rule is manipulable for the preference orders in `profiles`. [15 marks]

We also plot the average rank of the winning candidate across all voters, and the fraction of manipulable profiles for randomly generated preference profiles. The expected plots are given in Figures 1 and 2.

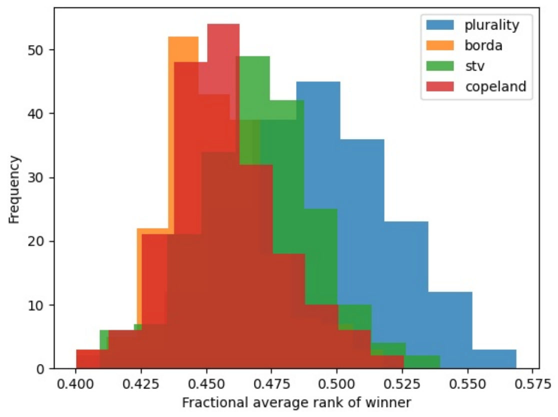


Figure 1: A histogram for the average rank of the winner (as a fraction of the number of candidates) across 200 randomly generated preference profiles. A lower rank is better as it shows that on average more voters were satisfied. As we can see from this graph, plurality performs the worst on this metric, while Copeland and Borda's rules are the best.

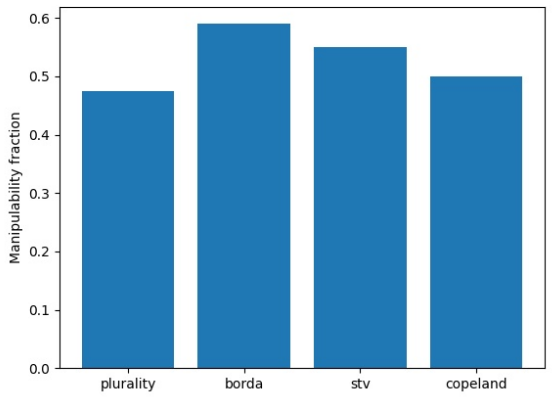


Figure 2: A bar plot of the fraction of times a randomly generated preference profile was manipulable. Note that Borda's rule tends to be the most manipulable voting strategy. You can change the seeds and check if this still holds.

2 Gale Shapley Algorithm [30 marks]

In the field of mechanism design, the matching problem refers to the task of pairing a set of suitors with a set of reviewers based on their preferences. This scenario considers an equal number of suitors and reviewers. The goal is to find a stable matching, where no suitor and reviewer prefer each other over their current partners. The Gale-Shapley algorithm guarantees to find a stable matching for any instance of the problem.

To learn more about the Gale-Shapley algorithm, you can visit its Wikipedia page here. Suitors are Proposers and reviewers are Acceptors according to our definition.

Every suitor has a strict preference profile over the reviewers given by a dictionary variable `suitor_prefs`. Similarly, every reviewer has a strict preference profile over the reviewers given by variable `reviewer_prefs`. These preferences are read from `data_n.txt` by the helper function `get_preferences`. It is provided that the suitors are denoted by lowercase alphabets while the reviewers are denoted by uppercase alphabets.

We will consider this running example of the preferences to explain the other functions.

```
suitor_prefs = { ‘A’: [‘a’, ‘b’, ‘c’], ‘B’: [‘c’, ‘b’, ‘a’], ‘C’: [‘c’, ‘a’, ‘b’] }
reviewer_prefs = { ‘a’: [‘A’, ‘C’, ‘B’], ‘b’: [‘B’, ‘A’, ‘C’], ‘c’: [‘B’, ‘A’, ‘C’] }
```

2.1 Tasks

You have the following tasks to complete in `q2.py`:

1. **Gale Shapley Algorithm** [20 marks]
   Implement the function `Gale_Shapley(suitor_prefs, reviewer_prefs)` which returns a mapping from suitors to reviewers. Specifically for the above example, the returned matching takes the following form -

   ```
   matching = { ‘A’: ‘a’, ‘B’: ‘c’, ‘C’: ‘b’ }
   ```

2. **Average Ranking** [10 marks]
   For every suitor, we can assign a ranking score based on the matched reviewer in his preference profile. Since, `A` is matched to its first choice `a`, it receives a ranking score of 1. Similarly, `C` is matched to its third choice `b`, it receives a ranking score of 3.

   ```
   suitor_ranking = { ‘A’: 1, ‘B’: 1, ‘C’: 3 }
   ```

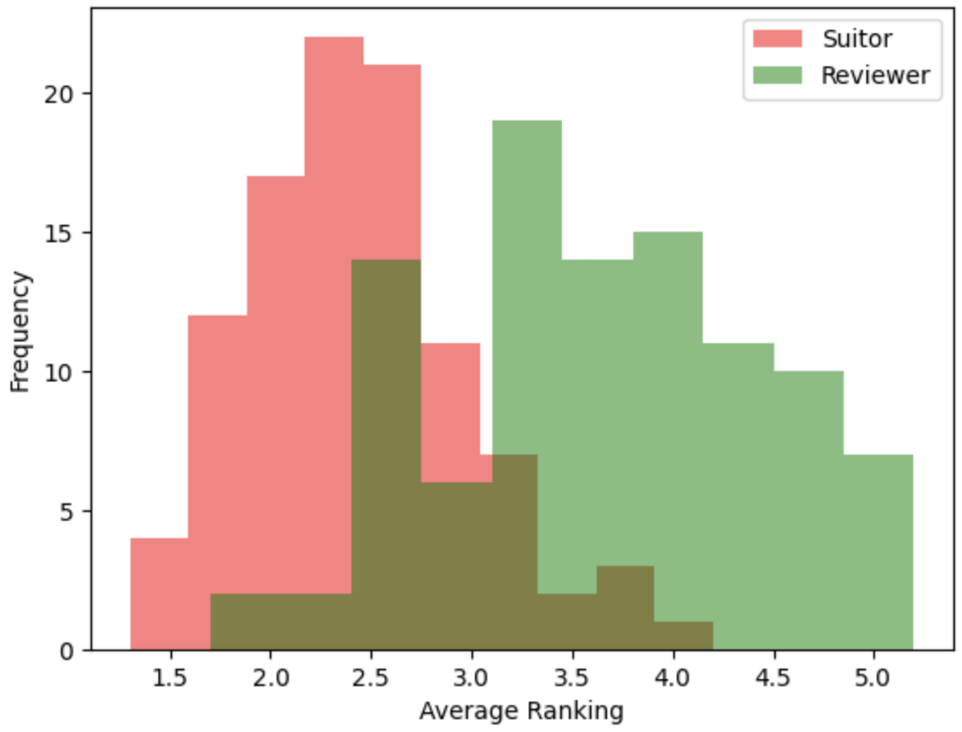
   Implement the function `avg_suitor_ranking(suitor_prefs, matching)` which takes in the suitor preferences and a matching between suitors and reviewers and outputs the average suitor ranking. For the given example, we can find out average suitor ranking as

   ```
   avg_suitor_ranking = (1 + 1 + 3) / 3 = 5/3
   ```

   Similarly, implement the function `avg_reviewer_ranking(reviewer_prefs, matching)`.

2.2 Analysis of Average Ranking [Reading Exercise]

We employ the Gale-Shapley algorithm to determine stable matchings corresponding to 100 randomized preference profiles of 10 suitors and 10 reviewers. Subsequently, we generate a histogram illustrating the average rankings for both suitors and reviewers. Please refer to the plot saved in the file `q2.png`. Why do you think suitors tend to receive lower rankings compared to reviewers? You might find this resource helpful.


Suitor
Reviewer
Frequency
Average Ranking