Algorithms project - Preferential voting systems

Alberto Amadessi 1072049

April 2024

1 Introduction

Given n voters $x_1, ..., x_n$ and k options $a_1, ..., a_k$, assuming each voter has a preference profile, i.e. an ordering of the options, a **preferential** (or **ranked**) **voting system** is a procedure which, from individual preference profiles, constructs a profile of common preference, called **social choice**.

Different preferential voting systems can correspond to different social choices and my goal is to analyze the most popular ones in order to raise their properties both from a qualitative and algorithmic point of view.

2 History

The modern analysis of ranked voting began when Jean-Charles de **Borda** published a paper in 1781, advocating for the Borda count method that now bears his name. This methodology drew criticism from the Marquis de **Condorcet**, who developed his own methods after arguing Borda's approach did not accurately reflect group preferences. Interest in ranked voting continued throughout the 19th century. Danish pioneer Carl Andræ formulated the single transferable vote (**STV**) system, which was adopted by his native Denmark in 1855. Condorcet had previously considered the similar **instant-runoff system** before rejecting it as paradoxical. Theoretical exploration of electoral processes was initiated by a 1948 paper from Duncan Black, which was soon followed by Kenneth **Arrow**'s research on the consistency of voting criteria. This subject has continued to receive scholarly interest under **social choice theory**, a branch of welfare economics.

3 Adoptions around the world

Ranked voting first saw governmental use in the 1890s in Tasmania, deploying the STV system. Its broader adoption in Australia began in the 1910s. By the 1920s, ranked voting had expanded globally: it was used in Ireland, South Africa, and approximately 20 cities each in Canada and the United States. In more recent years, ranked choice voting has been implemented in.

In November 2016, the voters of Maine approved instant-runoff voting for all elections. This was first put to use in 2018, marking the inaugural use of a ranked choice voting system in a statewide election in the United States. Later, in November 2020, also Alaska voters approved this kind of vote, bringing ranked choice voting into effect from 2022. Nowadays, in the United States, some cities, counties, and federal primaries across 16 states, as well as 5 additional states' overseas voters for federal elections, employ ranked voting.

4 General assumptions

The goodness of a voting system is determined by the qualities it presents, so I report the main desirable characteristics of a multiple choice voting system:

1. **Anonymity**: Each vote counts equally, that is, by exchanging the votes between two voters the social choice does not change.
2. **Neutrality**: Each option is treated equally by the electoral procedure: if the social choice is a_1 and all voters swap places a_1 and a_2 in their preference profiles, then the social choice becomes a_2.
3. **Monotony**: If the social choice is a or indeterminate and one or more voters move a up their preference scale, then the social choice is a.

Theorem (May): In the case of a choice with two options, the only procedure that verifies the properties of anonymity, neutrality and monotony is absolute majority voting.

5 Arrow's theorem

In 1951 Kenneth Arrow in his doctoral thesis gave an axiomatic definition of a voting system, requiring the following properties:

1. **Universality**: The voting system must assign a collective preference to each set of individual preferences.
2. **Democraticity = Anonymity**: By exchanging the preferences of two voters the social choice does not change.
3. **Unanimity**: If all voters prefer a_1 to a_2, then society must prefer a_1 to a_2.
4. **Independence from irrelevant alternatives**: The social choice between the options a_1 and a_2 is determined only by the preferences relative to a_1 and a_2.

Let $X = \{x_1, ..., x_n\}$ be the finite set of voters and let $A = \{a_1, ..., a_k\}$ be the finite set of options. An individual preference profile on A is a one-to-one function $u : A \to \{1, ..., k\}$, i.e. an order on options. The a_i option is preferred to the a_j option in the u profile if and only if $u(a_i) > u(a_j)$.

Let U be the set of individual preference profiles, then a collective preference profile is an element $\mathbf{u} = (u_1, ..., u_n)$ of $P := U^n$. Arrow defines as a social choice a family of functions $\{f_{n,k}\}$ that can be used for any pair of finite sets (X, A).

Universality translates into the fact that such functions exist for every possible pair of integers (n, k). **Anonymity** translates into imposing that

$$f_{n,k}(u_1, ..., u_i, ..., u_j, ..., u_n) = f_{n,k}(u_1, ..., u_j, ..., u_i, ..., u_n)$$

Unanimity is formalized when

$$u_i(a_1) > u_i(a_2) \quad \forall i \quad \Rightarrow \quad f_{n,k}(\mathbf{u})(a_1) > f_{n,k}(\mathbf{u})(a_2)$$

To formalize the last property, given a preference profile $u : A \to \{1, ..., k\}$, we associate a new profile with it, $\bar{u} : \{a_1, a_2\} \to \{1, 2\}$, setting

$$\bar{u}(a_1) > \bar{u}(a_2) \Leftrightarrow u(a_1) > u(a_2).$$

so **independence from irrelevant alternatives** therefore occurs if

$$f_{n,2}(\bar{\mathbf{u}})(a_1) > f_{n,2}(\bar{\mathbf{u}})(a_2) \Leftrightarrow f_{n,k}(\mathbf{u})(a_1) > f_{n,k}(\mathbf{u})(a_2)$$

Theorem (Arrow): In the case with three or more options there are no voting procedures that satisfy the four axioms.

6 Gibbard-Satterthwaite's theorem

A **non-manipulable** social choice is defined as a decision-making process that guarantees the impossibility of influencing the outcome through fraudulent or manipulative actions, thus helping to preserve the fairness and integrity of the decision-making process. A **non-dictatorial** social choice is a decision-making process in which the outcome is determined in a fair and inclusive way, taking into account the preferences and opinions of all members of society, rather than being imposed by a single person or a small group of individuals.

Let us therefore consider a function $g : P \to A$ as a social choice function, that is, the function does not associate an individual preference profile with the collective preference profiles, but only the winning option.

Then, a social choice function $g : P \to A$ is **non-manipulable** if for every voter x_i and every pair of profiles $\mathbf{u} = (u_1, ..., u_i, ..., u_n)$, $\mathbf{u}' = (u_1, ..., u_i', ..., u_n)$ we have $u_i(g(\mathbf{u}')) \leq u_i(g(\mathbf{u}))$; instead, it is said to be **dictatorial** if there exists a voter x_i such that $u_i(g(\mathbf{u})) \geq u_i(a_j)$ for every $u \in P$ and every $a_j \in A$.

Theorem (Gibbard - Satterthwaite): A social choice function between three or more options, unanimous and non-manipulable is dictatorial.

7 Presentation of datasets

I decided to use two datasets in order to highlight the disadvantages and advantages of the various multiple election methods that I present.

The first dataset presents 19.811 observations and concerns the **Democratic Party** primary elections in the state of Alaska held on 10 April 2020, where voters expressed their first 5 preferences among the 8 Democratic candidates.

The second dataset was created artificially and presents 55 observations, which correspond to the preferences of 55 voters in 5 candidates called 'a','b','c','d' and 'e'. We call the first dataset **Df** while the second is referred to as **Df2**.

Using these two datasets I can both show what the actual characteristics of the algorithms are in a real case and show how, based on the type of voting method chosen, different social choices can be obtained in a balanced case.

```
# View the top 5 and bottom 5 observations of Df
print(pd.concat([Df.head(), Df.tail()]))
############################################################
                1          2          3        4        5
    0       Biden  Klobuchar  Buttigieg   Steyer   Warren
    1       Biden    skipped    skipped  skipped  skipped
    2       Biden    skipped    skipped  skipped  skipped
    3     Sanders      Biden    skipped  skipped  skipped
    4       Biden     Warren    skipped  skipped  skipped
    19806 Sanders      Biden     Warren  skipped  skipped
    19807   Biden    skipped    skipped  skipped  skipped
    19808   Biden    Sanders    skipped  skipped  skipped
    19809 Sanders    skipped     Warren   Steyer  Gabbard
    19810  Warren    Sanders      Biden  skipped  skipped
############################################################
```

```
# View the top 5 and bottom 5 observations of Df2
print(pd.concat([Df2.head(), Df2.tail()]))
############################################################
                      1  2  3  4  5
                0     a  d  e  c  b
                1     a  d  e  c  b
                2     a  d  e  c  b
                3     a  d  e  c  b
                4     a  d  e  c  b
                50    e  b  c  d  a
                51    e  b  c  d  a
                52    e  b  c  d  a
                53    e  b  c  d  a
                54    e  c  b  a  d
############################################################
```

8 Preprocessing of datasets

Since Df is a real dataset, it may have errors in data collection or may have data inconsistent with reality. In order to increase the quality of the dataframe, it is essential to clean the dataset in order to eliminate misleading observations.

The following code eliminates null votes, i.e. all votes in which there is not even a favorite among the available candidates (any preference coincides with 'skipped'), deletes duplicates and moves the chosen candidates in a coherent way with what are the orders of preference (for example if in the observation there are 2 candidates positioned in second and fourth place, the code transfers them to first and second place maintaining the preferential order).

```
# Data preprocessing Df

# 1 op per n loop + 1 op
for i in Df.index:
  # 7 op per n loop
  if (all(Df.loc[i,:]=='skipped')):
    # 1 op per n loop
    Df.drop(i, inplace=True)
  else:
    # 2 op per n loop
    serie=Df.loc[i,:]
    # 8 op per n loop
    serie=serie[~serie.isin(['skipped'])]
    # 11 op per n loop
    serie.drop_duplicates(inplace=True)
    # 2 op per n loop per 4 loop
    while(len(serie)<5):
      # 4 op per n loop per 4 loop
      serie=pd.concat([serie,pd.Series('skipped')])
    # 3 op per n loop
    Df.loc[i,:]=list(serie)
# 5 op
Df.index=list(range(Df.shape[0]))
```

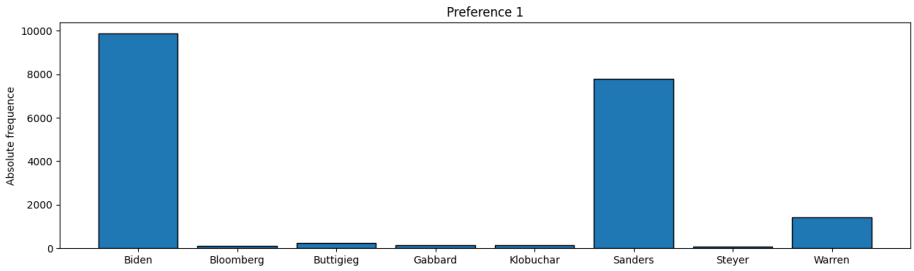
A for loop is initialized on the indices and a preliminary check is done using conditional instructions in order to eliminate null votes. In valid observations, the else condition treats the vote as a series where duplicates and skipped votes are eliminated while other are moved up. Then the missing data are filled in the dataframe with skipped votes and the function ends by updating the indices.

9 Data analysis

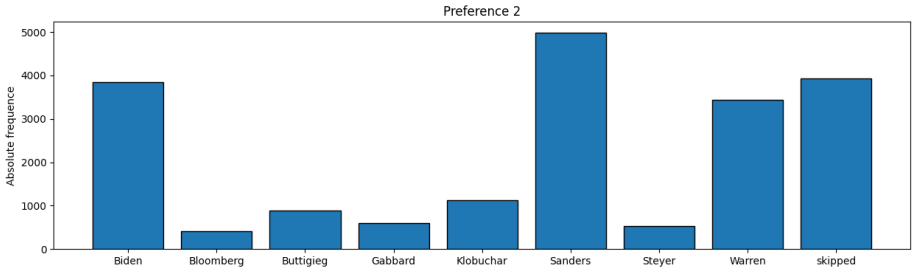
In order to understand the dataset I performed an appropriate data analysis. Through the study of histograms my aim is to analyze what are the preferences of voters in the state of Alaska.

Df analysis

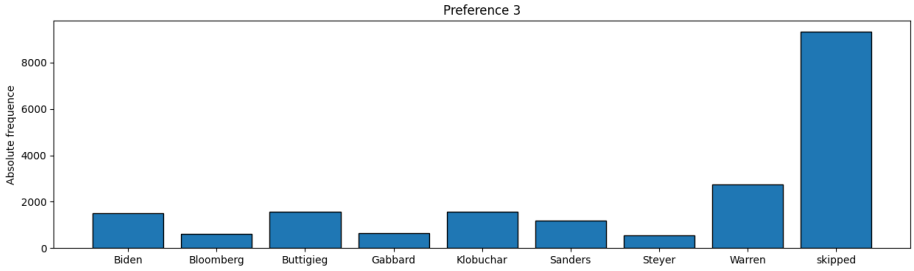
As can be seen from the histogram showing the candidates selected by voters as their first preference, it seems that the only two contenders as the preferred candidate in the social choice are Joe Biden and Bernie Sanders.



Analyzing the histogram of the second preference, we observe that Elizabeth Warren is also of great interest among voters, but the 'skipped' column suggests that many people voted only for the first choice, leaving the last 4 slots blank.

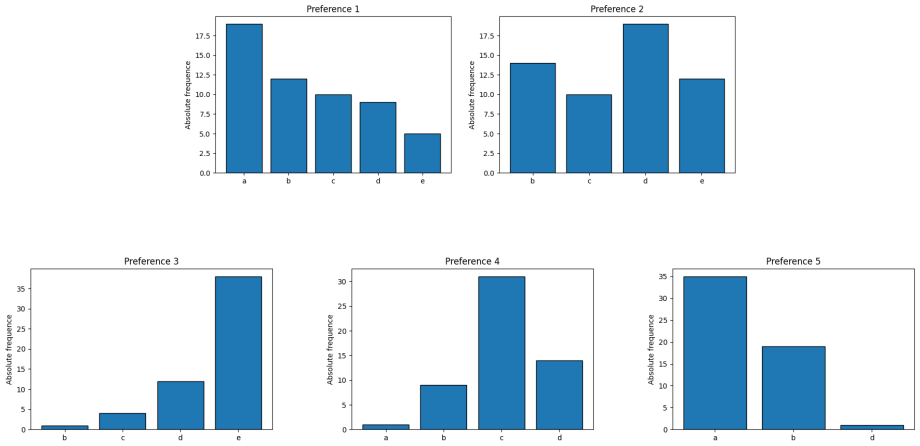


In the third choice, however, the votes seem to be distributed equally even if the majority of voters left the box blank. This also applies to the fourth and fifth choices where I decide not to report the histograms.



Df2 analysis

In the case of the artificial dataset, the votes are much more balanced and I will use this one in order to show how in case of a different voting method the results can change significantly.



10 Preparatory functions

In this section I describe the functions I used to compile the voting algorithms.

FindCandidates function

The **FindCandidates** function takes a dataset as input and returns the list of candidates that are part of it.

```
# Find candidates
def FindCandidates(Dataframe):
  # 5n+3 op
  Candidates=list(pd.unique(Dataframe.values.flatten()))
  # k+1 op
  if 'skipped' in Candidates:
    # 1 op
    Candidates.remove('skipped')
  # 1 op
  return(Candidates)
```

Using the values.flatten() function, the dataframe is flattened to an array of objects whose uniqueness is taken with the pandas library command pd.unique. The candidates are saved in a list from which the 'skipped' value is removed.

BuildDf function

The **BuildDf** function takes as input a list of candidates and returns a dataframe initialized with all zeros with row and column names that match the candidate names.

```
# Dataframe initialization
def BuildDf(CandidatesList):
  # k*k+3 op
  matrix=np.zeros((len(CandidatesList),len(CandidatesList)))
  # 4 op
  df=pd.DataFrame(matrix, columns=CandidatesList, index=
      CandidatesList)
  # 1 op
  return(df)
```

The length of the candidate list is used to create a square matrix that is transformed into a dataframe using the pandas library function pd.DataFrame.

PreferencesMatrix function

The **PreferencesMatrix** function has a dataframe as input and returns the respective preference matrix between the candidates, i.e. it reports the count of how many times candidate i was preferred over candidate j, $\forall i, j \in k$ candidates.

```
# Preferences matrix
def PreferencesMatrix(Dataframe):

  # Matrix initialization
  # 5n+k+6 op
  Candidates=FindCandidates(Dataframe)
  # k*k+8 op
  df=BuildDf(Candidates)

  # Matrix filler
  # 1 op per n loop + 2 op
  for i in range(Dataframe.shape[0]):
    # 2 op per n loop
    Worst=set(Candidates)
    # 1 op per n loop per 5 loop + 3 op per n loop
    for j in range(1,Dataframe.shape[1]+1):
      # 3 op per n loop per 5 loop
      C=Dataframe.at[i,f'{j}']
      # 2 op per n loop per 5 loop
      if (C=='skipped'): break
      # 1 op per n loop per 5 loop
      Worst.remove(C)
```

```
      # 1 op per n loop per 5 loop per k loop
      for l in Worst:
        # 3 op per n loop per 5 loop per k loop
        df.at[C,l]+=1

  # Sorting matrix
  # k*log(k) op
  df.sort_index(axis=0, inplace=True)
  # k*log(k) op
  df.sort_index(axis=1, inplace=True)

  # k*k op
  return(df.astype(int))
```

Initially the two functions seen previously are called in order to obtain a dataframe initialized to zero with rows and columns renamed with the name of the candidates. Subsequently, a for loop is executed for each observation present in the input dataframe which as a first action creates the set of candidates present starting from the list of candidate names. The for loop is then executed on the columns of the input dataframe and by crossing the values of the two loops each entry of the dataframe is reached, which contains a name of a candidate. The extracted candidate is removed from the set of candidates and with a third for loop the created dataframe is updated indicating that the extracted candidate is preferred over the candidates remaining in the candidate set. This procedure continues for each column until a 'skipped' value is extracted, in this case the conditional if statement interrupts the loop on the columns and the algorithm proceeds with the next row until the dataframe ends.

PreferencesMatrix output on Df2

To better understand the output of this function, I report the printout on Df2. The first line says that candidate a was preferred 19 times over candidates b, c and e and 20 times over candidate d.

```
# Print preferences matrix of Df
print(PreferencesMatrix(Df2))
############################################################
   a   b   c   d   e
a  0  19  19  20  19
b 36   0  16  27  22
c 36  39   0  15  19
d 35  28  40   0  28
e 36  33  36  27   0
############################################################
```

FrequencyVote function

The **FrequencyVote** function counts the frequency of preferences expressed, that is, it counts how many times each sequence of preferences has occurred.

```
# Frequency votes
def FrequencyVote(Dataframe):

  # Matrix initialization
  CountMatrix=Dataframe.groupby(list(Dataframe.columns)).
      size().reset_index(name='Count')
  Candidates=FindCandidates(Dataframe)
  matrix=np.zeros((len(Candidates)+1, 1))
  index=Candidates+list(['COUNT'])
  FrequencyVote=pd.DataFrame(matrix, columns=[''], index=
      index)

  # Matrix filler
  for i in range(CountMatrix.shape[0]):
    Values=[int(x) for x in CountMatrix.iloc[i,:][0:-1].
        index]+list([CountMatrix.iloc[i,-1]])
    Index=list(CountMatrix.iloc[i,:][0:-1])+list(['COUNT'])
    serie=pd.Series(Values, index=Index, name=f'{i+1}')
    if 'skipped' in serie.index:
      serie=serie.drop('skipped')
    FrequencyVote=pd.concat([FrequencyVote,serie],axis=1)

  # Sorting matrix
  FrequencyVote.drop('', axis=1, inplace=True)
  FrequencyVote.fillna(len(Candidates), inplace=True)

  return(FrequencyVote.astype(int))
```

The first line of the algorithm uses the groupby function to aggregate identical votes, which are counted in the 'Count' column. Subsequently, a dataframe is initialized to zero with a single column where the rows coincide with the name of the candidates with the addition of a bottom row called 'Count'. Then a for loop is initialized for each different vote that has been performed. The list containing the order of the candidates and the number of times the vote was carried out is saved inside the 'Values' variable, while the 'Index' list contains the sequence of the chosen candidates. From these two lists a series is created from which the 'skipped' values are removed and is then concatenated to the dataframe created previously. Finally, the created dataframe is filled by inserting the number of candidates in the missing data as values.

FrequencyVote output on Df2

To better understand the output of this function I report the print on Df2. The first column says that 19 times there has been a vote in which candidate a is in first place, d in second, e in third, c in fourth and b in fifth.

```
# Print frequency of votes of Df2
print(FrequencyVote(Df2))
############################################################
                          1   2   3  4  5  6
                    a     1   5   5  5  5  4
                    d     2   3   4  1  4  5
                    e     3   2   3  3  1  1
                    c     4   4   1  2  3  2
                    b     5   1   2  4  2  3
                    COUNT 19  12  10 9  4  1
############################################################
```

StrongestPaths function

The **StrongestPaths** function takes as input the matrix of preferences of candidates and returns a matrix containing the strongest paths of candidates. It corresponds to the implementation of Floyd-Warshall algorithm.

```
# Strongest paths
def StrongestPaths(DistancesMatrix):

  # Paths initialization
  # 2 op
  Candidates=DistancesMatrix.columns
  # k*k+8 op
  p=BuildDf(Candidates)

  # Paths filler
  # 1 op per k loop
  for i in Candidates:
    # 1 op per k loop per k loop
    for j in Candidates:
        # 1 op per k loop per k loop
        if (i!=j):
            # 3 op per k loop per k loop
            if (DistancesMatrix.at[i,j] > DistancesMatrix.at
                [j,i]):
                # 3 op per k loop per k loop
                p.at[i,j]=DistancesMatrix.at[i,j]
            else:
                # 2 op per k loop per k loop
```

```
                p.at[i,j]=0

  # 1 op per k loop
  for i in Candidates:
    # 1 op per k loop per k loop
    for j in Candidates:
        # 1 op per k loop per k loop
        if (i!=j):
            # 1 op per k loop per k loop per k loop
            for k in Candidates:
                # 3 op per k loop per k loop per k loop
                if (i!=k) and (j!=k):
                    # 7 op per k loop per k loop per k
                    p.at[j,k]=max(p.at[j,k], min(p.at[j,i],
                        p.at[i,k]))

  # k*k op
  return(p.astype(int))
```

The first step consists in obtaining the name of the candidates and initializing a dataframe using the BuildDf function. Subsequently, two for loops are initialized on the list of candidates and through a conditional if-else statement the new dataframe is updated by inserting 0 or the distance present in the preference matrix as inputs based on the head-to-head clash between the two candidates. Once the two cycles are finished, another 3 cycles are initialized on the candidates in which the strongest paths of the various candidates are updated taking as input the maximum value between the direct path or the minimum value obtained passing through an intermediate candidate.

StrongestPaths output on Df2

To better understand the output of this function I report the print on Df2, but the functionality of this function will be explained later.

```
# Print strongest path on Df2
print(StrongestPaths(PreferencesMatrix(Df2)))
############################################################
    a   b   c  d   e
a   0   0   0  0   0
b  36   0   0  0   0
c  36  39   0  0   0
d  36  39  40  0  28
e  36  36  36  0   0
############################################################
```

11 Criteria of an election method

Different voting methods correspond to different electoral criteria. Below I list the main properties that the main voting methods compete for:

1. **Majority**: If a candidate is selected as the first choice in a majority of the voting, then that candidate is the overall winner.
2. **Majority loser criterion**: If a majority of voters prefers every other candidate over a given candidate, then that candidate must not win.
3. **Mutual majority criterion**: If a political faction or party wins a majority of the vote, they should win the election.
4. **Condorcet winner criterion**: A voting systems satisfy the Condorcet criterion if the election is won by a Condorcet winner, that is a candidate that win any one-on-one matchup with another candidate.
5. **Condorcet loser criterion**: A voting system complying with the Condorcet loser criterion will never allow a Condorcet loser to win, that is a candidate who can be defeated in an head-to-head competition against each other candidate (This criterion implies the majority loser criterion but does not imply the Condorcet winner criterion).
6. **Independence of clones criterion**: This criterion measures an election method's robustness to strategic nomination. It states that the winner must not change due to the addition of a non-winning candidate who is similar to a candidate already present.
7. **Monotonicity**: This criterion says that voters should never have a negative effect on an election's results. In other words, increasing a winning candidate's grade should not cause them to lose.
8. **Reversal symmetry**: This criterion requires that if candidate i is the unique winner, and each voter's individual preferences are inverted, then i must not be elected.
9. **Later-no-harm criterion**: Adding a later preference to a ballot should not harm any candidate already listed.
10. **Later-no-help criterion**: The criterion is satisfied if, in any election, a voter giving an additional ranking to a less-preferred candidate can not cause a more-preferred candidate to win. Voting systems that fail the later-no-help criterion are vulnerable to the tactical voting strategy called mischief voting, which can deny victory to a Condorcet winner.

12 Schulze method

The **Schulze method** is an electoral system developed in 1997 by Markus Schulze that creates a sorted list of winners using votes that express preferences. The Schulze method is a **Condorcet method**, which means that if there is a candidate who is preferred by a majority over every other candidate in pairwise comparisons, then this candidate will be the winner by the Schulze method.

Mathematical description

We define as $d[V, W]$ the number of voters who prefer candidate V to candidate W. A path from candidate X to candidate Y is a sequence of candidates $C(1), ..., C(n)$ with the following properties:

1. $C(1) = X$ and $C(n) = Y$.
2. $\forall i = 1, ..., n-1$ we have $d[C(i), C(i+1)] > d[C(i+1), C(i)]$.

In other words each candidate in the path beats the next candidate in a pairwise comparison. We call the **strength p** of a path from candidate X to candidate Y the smallest number of voters in the sequence of comparisons:

$$d[C(i), C(i+1)] > p \quad \forall i = 1, ..., n-1$$

For a pair of candidates X and Y that are connected by at least one path, the **strength of the strongest path** $p[X, Y]$ is the maximum strength of the paths connecting them. If there is no path from candidate X to candidate Y at all, then $p[X, Y] = 0$. We have that:

- Candidate V is better than candidate W if and only if $p[V, W] > p[W, V]$.
- Candidate V is a potential winner if and only if $p[V, W] \geq p[W, V]$ for every other candidate W.
- It is guaranteed that the above definition of "better" defines a transitive relation.
- There is always at least one candidate V with $p[V, W] \geq p[W, V]$ for every other candidate W.

Pseudocode

I report the pseudocode used for the Schulze voting method with the related computational cost. As we can easily see, the complexity lies on the order of $\theta(k^2)$, where k is the number of candidates.

```
# Input: p matrix where p[i,j] is the strength of the
    strongest path from candidate i to candidate j.
# Output: Schulze candidate winner.

  # 1 op per k loop
  for i in p.columns
      # k+3 op per k loop
      comparison=(p[i,:]>=p[:,i])
      # k op per k loop
      if (all(comparison)):
        # 1 op
        return(i)
```

Algorithm implementation on Df2

The **SchulzeMethod** function takes a dataframe as input and reports the winner according to Schulze method as output.

```
#Schulze method on Df2
SchulzeMethod(Df2)

# Schulze method function
def SchulzeMethod(Dataframe):

  # PreferencesMatrix function call
  DistancesMatrix=PreferencesMatrix(Dataframe)
  # StrongestPaths function call
  Path=StrongestPaths(DistancesMatrix)

  # For loop on candidates
  for i in Path.columns:
      # Comparison of row vs column for the candidate
      Compare=(Path.loc[i,:]>=Path.loc[:,i])
      # If statement that check the Schulze condition
      if (all(Compare)):
        # Winning candidate return
        return(i)
```

The first step consists in calculating the distance matrix of candidates in the dataframe, which I have already printed previously and which I report in the form of a graph in order to visualize what happened with the 55 votes.

For each pair of candidates, only one arrow is shown which indicates how many voters preferred the candidate at the base of the arrow compared to the candidate at the tip of the arrow. The number of voters who preferred the opposite is easily identified by taking the complement to 55.

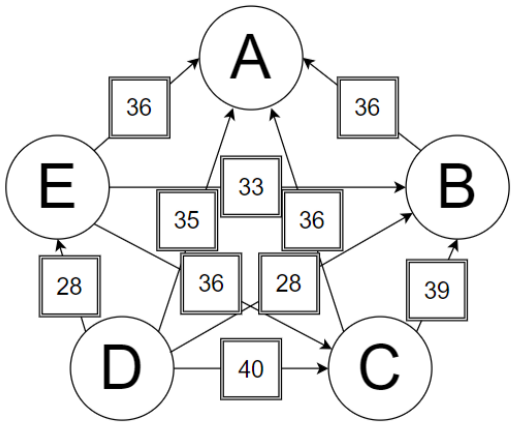


Figure 3: Preferences graph of Df2

The matrix of the strongest paths is then calculated, also in this case previously printed and viewable via graph.

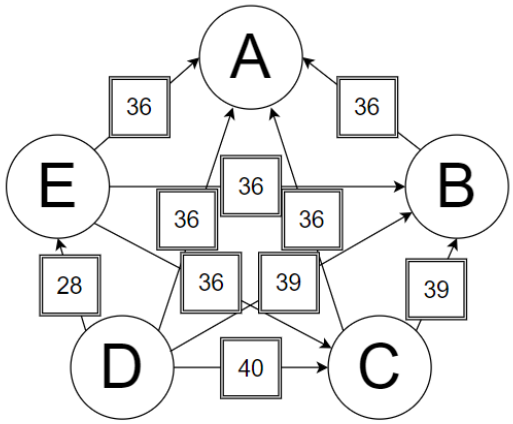


Figure 4: Graph of strenght of the strongest path of Df2

For each pair of candidates, only one arrow is reported which indicates the strength of the strongest path for the candidate at the base of the arrow to the candidate at the tip of the arrow. The reverse direction is calculated as zero.

The rest of the code simply selects the candidate that mutually wins against all other candidates in the strongest paths matrix. This is achieved when the candidate obtains a vector of non-negative values by considering the difference

between the respective candidate row and the respective transpose row. As can be seen from the graph, the only candidate that has only outgoing arrows is d which is chosen as the winner according to Schulze method.

Properties of Schulze method

Schulze method has the following properties:

- **Majority**
- **Majority loser criterion**
- **Mutual majority criterion**
- **Condorcet winner criterion**
- **Condorcet loser criterion**
- **Independence of clones criterion**
- **Monotonicity**
- **Reversal symmetry**

13 Borda method

The **Borda method** is an electoral system developed in 1770 by Jean-Charles de Borda that creates a sorted list of winners using votes of preference. The Borda count is intended to elect broadly acceptable candidates, rather than those preferred by a majority, and so is often described as a consensus-based voting system rather than a majoritarian one.

Pseudocode

I report the pseudocode used for the Borda voting method with the related computational cost. As we can easily see, the complexity lies on the order of $\theta(vk)$, where v is the number of different votes and k is the number of candidates.

```
# Input: Frequency votes dataframe.
# Output: Borda candidate winner.

  # 1 op per k loop
  for i in Frequency.index:
    # v+3 op per k loop
    score = dot(Frequency[i,:], Frequency[count,:])

  # k op
  return(score.idxmin())
```

Algorithm implementation on Df2

The **BordaMethod** function takes a dataframe as input and reports the winner according to Borda as output.

```
#Borda method on Df2
BordaMethod(Df2)

# Borda method function
def BordaMethod(Dataframe):

  # FrequencyVote function call
  Frequency=FrequencyVote(Dataframe)
  # Score initialization
  Frequency['Score']=np.zeros(Frequency.shape[0])

  # For loop on candidates
  for i in Frequency.index[0:-1]:
    # Products between positions and number of votes
    Frequency.loc[i,'Score']=
    np.dot(Frequency.loc[i,:][0:-1],
    Frequency.loc['COUNT',:][0:-1])

  # Borda winner return
  return(Frequency['Score'][0:-1].idxmin())
```

The first step is to calculate the candidate vote frequency matrix, which I printed previously for the Df2 dataframe. Next, a score column is initialized with zero values. The for loop determines the score of each candidate by multiplying the positions obtained by the respective voting frequencies. Finally, the winner with the lowest score is selected as it coincides with the candidate who best manages to satisfy the preferences of all voters. According to Borda voting method the winning candidate is e.

Properties of Borda method

Borda method has the following properties:

- **Majority loser criterion**
- **Condorcet loser criterion**
- **Monotonicity**
- **Reversal symmetry**
- **Later-no-help criterion**

14 Plurality method

Plurality voting refers to electoral systems in which a candidate in an electoral district who polls more than any other (that is, receives a plurality) is elected. Plurality voting is distinguished from majority voting, in which a winning candidate must receive an absolute majority of votes (50% of all votes).

Pseudocode

I report the pseudocode used for the plurality voting method with the related computational cost. As we can easily see, the complexity lies on the order of $\theta(vk)$, where v is the number of different votes and k is the number of candidates.

```
# Input: Frequency votes dataframe.
# Output: Plurality candidate winner.

  # 1 op per k loop
  for i in Frequency.index:
    # 1 op per k loop per v loop
    for j in Frequency.columns:
      # 2 op per k loop per v loop
      if (Frequency.at[i,j]==1):
        # 3 op per k loop per v loop
        score+=Frequency.at[count,j]

  # k op
  return(Frequency[score].idxmax())
```

Algorithm implementation on Df2

The **Plurality** function takes a dataframe as input and reports the winner according to plurality voting system as output.

```
#Plurality method on Df2
Plurality(Df2)

# Plurality function
def Plurality(Dataframe):

  # FrequencyVote function call
  Frequency=FrequencyVote(Dataframe)
  # Score initialization
  Frequency['Score']=np.zeros(Frequency.shape[0])

  # For loop on candidates
  for i in Frequency.index[0:-1]:
```

```
    # For loop on different votes
    for j in Frequency.columns:
      # Check on position 1
      if (Frequency.at[i,j]==1):
        # Score update
        Frequency.at[i,'Score']+=Frequency.at['COUNT',j]

  # Plurality winner return
  return(Frequency['Score'].idxmax())
```

The first step is to calculate the usual candidate vote frequency matrix and next a score column is initialized with zero values. The two for loops combined with the conditional if statement are used to count how many times each candidate is chosen as the favorite. The score column is used to verify the final ranking from which the candidate with the highest score is extracted. According to plurality method the winner is candidate a.

Properties of plurality method

Plurality method has the following properties:

- **Majority**
- **Monotonicity**
- **Later-no-harm criterion**
- **Later-no-help criterion**

15 Instant-runoff method

Instant-runoff voting, also known as plurality with elimination or plurality loser, is a ranked-choice voting system that modifies plurality by repeatedly eliminating the last-place winner until only one candidate is left.

Instant-runoff voting elections are a virtual (instant) variant on exhaustive elimination. In each round, voters choose a favourite candidate; the last-place finisher is eliminated and another round is held.

Pseudocode

I report the pseudocode used for the instant runoff method with the related computational cost. The complexity lies on the order of $\theta(vk^2)$, where v is the number of different votes and k is the number of candidates.

```
# Input: Frequency votes dataframe.
# Output: Instant runoff candidate winner.

  # 2 op per k-1 loop
  while (Frequency.index>2):

    # 3 op per k-1 loop + k(k+1)/2 - 1 op
    Frequency[score]=np.zeros(Frequency.index)

    # k(k+1)/2 - 1 op
    for i in Frequency.index:
      # v(k(k+1)/2 - 1) op
      for j in Frequency.columns:
        # 2v(k(k+1)/2 - 1) op
        if (Frequency.at[i,j]==1):
          # 8v(k(k+1)/2 - 1) op
          Frequency.at[i,score]+=Frequency.at[count,j]

    # 2 op per k-1 loop + k(k+1)/2 - 1 op
    eliminated=Frequency[score].idxmin()

    # k(k+1)/2 - 1 op
    for i in Frequency.index:
      # v(k(k+1)/2 - 1) op
      for j in Frequency.columns:
        # 3v(k(k+1)/2 - 1) op
        if (Frequency.at[i,j]>Frequency.at[eliminated,j]):
          # 9v(k(k+1)/2 - 1) op
          Frequency.at[i,j]-=1

    # 2 op per k-1 loop
    Frequency=Frequency.drop(eliminated)

  # 1 op
  return(Frequency.index)
```

Algorithm implementation on Df2

The **IRV** function takes a dataframe as input and reports the winner according to the instant runoff voting system as output.

```
#Instant runoff method on Df2
IRV(Df2)

# Instant runoff
def IRV(Dataframe):
```

```
# FrequencyVote function call
Frequency=FrequencyVote(Dataframe)

# While loop
while (Frequency.shape[0]>2):

  # Score initialization
  Frequency['Score']=np.zeros(Frequency.shape[0])

  # For loop on candidates
  for i in Frequency.index[0:-1]:
    # For loop on different votes
    for j in Frequency.columns:
      # Check on position 1
      if (Frequency.at[i,j]==1):
        # Score update
        Frequency.at[i,'Score']+=Frequency.at['COUNT',j]

  # Last candidate is eliminated
  Eliminated=Frequency['Score'][0:-1].idxmin()

  # For loop on candidates
  for i in Frequency.index[0:-1]:
    # For loop on different votes
    for j in Frequency.columns[0:-1]:
      # Votes updating
      if (Frequency.at[i,j]>Frequency.at[Eliminated,j]):
        Frequency.at[i,j]-=1

  # Drop eliminated candidate row
  Frequency=Frequency.drop(Eliminated)

# IRV winner return
return(Frequency.index[0])
```

The first step is to calculate the usual candidate vote frequency matrix. The while loop runs until the algorithm has not found the winning candidate and at each round it eliminates the candidate deemed worst by this method. A score column is initialized to zero for each candidate and the next two for loops together with the conditional if instruction update the column by inserting as a score how many times each candidate was defined as the first choice. Subsequently, the candidate with the lowest score is eliminated and the subsequent commands are used to update the matrix in order to scale the preferences with each elimination. The candidate selected by the algorithm is c.

Properties of instant runoff method

Instant runoff method method has the following properties:

- **Majority**
- **Majority loser criterion**
- **Mutual majority criterion**
- **Condorcet loser criterion**
- **Independence of clones criterion**
- **Later-no-harm criterion**
- **Later-no-help criterion**

Algorithm analysis

16 Considerations

I report in the following table the properties and the winners of the algorithms of the preference voting systems that I have analysed.

Criterion\Method	Schulze	Borda	Plurality	IRV
Majority	YES	NO	YES	YES
Majority loser criterion	YES	YES	NO	YES
Mutual majority criterion	YES	NO	NO	YES
Condorcet winner	YES	NO	NO	NO
Condorcet loser	YES	YES	NO	YES
Independence of clones	YES	NO	NO	YES
Monotonicity	YES	YES	YES	NO
Reversal symmetry	YES	YES	NO	NO
Later-no-harm	NO	NO	YES	YES
Later-no-help	NO	YES	YES	YES
Winning Candidate	D	E	A	C

Figure 5: Properties and winners of voting methods

The first important observation to consider is that no preference voting method can be considered better than another and there is no one that is absolutely better. The preference system to choose must be weighted based on the properties you want to satisfy. Subsequently, each voting method proposed a different winning candidate. This means that in balanced cases the choice of the algorithm can significantly influence the choice of the community.

17 Algorithms in real case scenario

Now that the algorithms have been tested on a toy model, I try to see the results on the dataframe regarding the Democratic Party primary elections.

```
# Schulze method test
start_time = time.time()
print(" \nAccording to Schulze method the winner is " +
    SchulzeMethod(d))
end_time = time.time()
print("Schulze time:", round(end_time-start_time,2), "
    seconds")

# Borda method test
start_time = time.time()
print(" \nAccording to Borda method the winner is " +
    BordaMethod(d))
end_time = time.time()
print("Borda time:", round(end_time-start_time,2), "seconds"
    )

# Plurality method test
start_time = time.time()
print(" \nAccording to plurality method the winner is " +
    Plurality(d))
end_time = time.time()
print("Plurality time:", round(end_time-start_time,2), "
    seconds")

# Instant runoff test
start_time = time.time()
print(" \nAccording to instant runoff method the winner is "
     + IRV(d))
end_time = time.time()
print("Instant runoff time:", round(end_time-start_time,2),
    "seconds")

According to Schulze method the winner is Biden
Schulze time: 17.48 seconds

According to Borda method the winner is Biden
Borda time: 13.86 seconds

According to plurality method the winner is Biden
Plurality time: 14.71 seconds

According to instant runoff method the winner is Biden
Instant runoff time: 20.32 seconds
```

As can be seen from the output, in this case the only winner is Biden and the algorithms seem to take a different running time to each other. The algorithms of the Borda and plurality methods are faster as they are simpler to implement and have a lower computational cost. The algorithms of the Schulze and instant runoff methods require a greater computational effort, but on the other hand they are able to satisfy a greater number of required properties. Ultimately, I consider these last two methods to be the best to implement for a real case, while I would exclude the first two.

18 Improvements

The optimal way to run the entire code is to call the preparatory functions only once and save them in a variable, rather than calling them every time for each algorithm used. The approach I used instead aims to study the computational cost for all the algorithms using the same input (the dataframe containing the votes). An improvement at the computational level that could be developed could therefore be to separate preparatory functions and voting methods and to give each method a different input based on how the algorithm must operate.
Instead, an improvement regarding the properties of the voting algorithm could be to switch to an approval vote. Approval voting is a voting system used in elections, in which the voter has the right to give his preference to as many candidates as he wishes. This type of voting can be considered a primitive form of scoring voting, where voters can choose between only two possibilities instead of a numerical score: accept or not accept. Approval voting is not affected by the Arrow Impossibility Theorem paradox, as it is not a system in which votes are sorted by preference.

19 Bibliography

- Matematica e sistemi elettorali (2023)
- Votazioni e sistemi di preferenze (2023)
- *Wikipedia: Ranked Voting* (2024). URL: `https://en.wikipedia.org/wiki/Ranked_voting`
- *Wikipedia: Schulze Method* (2024). URL: `https://en.wikipedia.org/wiki/Schulze_method`
- *Wikipedia: Borda Count* (2024). URL: `https://en.wikipedia.org/wiki/Borda_count`
- *Wikipedia: Plurality Voting* (2024). URL: `https://en.wikipedia.org/wiki/Plurality_voting`
- *Wikipedia: Instant-runoff Voting* (2024). URL: `https://en.wikipedia.org/wiki/Instant-runoff_voting`