Analysis of voting systems: title here

Adrian Salamon

Kungsholmens gymnasium

Senior thesis

Supervised by:
Maja Kankaanranta

April 2, 2017

Abstract

The abstract text goes here.

Contents

1 Introduction

1.1 Background

Taking a collective decision as a population is difficult. To solve this issue, voting systems with defined rules are used. They are used to show common preferences within a population, for example what politician a population wants to see elected. Several types of systems have been designed and there are a myriad of variations of those systems. They range from simple methods such as “most votes win” to complex processes that can only be practically carried out by a computer. However, practically all voting systems are algorithmic in nature, which makes them interesting to study for both Computer Scientists and Mathematicians. When deciding on what systems that shall be used, it is useful to know what differences and similarities they have.

1.2 Purpose

The purpose of this essay is to evaluate differences and similarities in terms of election results in three different voting systems: Single Transferable Vote (STV), First-Past-The-Post (FPTP) and the Schulze Method. This paper will only be examining and discussing differences and similarities in the results of the methods, not touching on how practically feasible the methods might be in actual election.

2 Theory

Theory will be here some day

3 Methodology

All voting methods have been implemented in Typescript – mostly due to the author having previous experience with JavaScript. Typescript is a superset of JavaScript with multiple extra features. Most importantly Typescript has optional static type checking. Sample code in typescript can be seen in code example 1.

Code example 1: Basic Typescript syntax

```
let helloWorld: string = 'Hello World' // This is a comment
console.log(helloWorld)
```

3.1 Modeling test data

In order to test the methods and their implementations, test data is needed. There are cases where extensive election data is published, such as in Maltese elections, but full ballot data, which is needed for the implementations has not been found. Instead, the implementations will be tested with data generated from a simple computer algorithm. The algorithm used in this paper is highly primitive, as modeling preferences within a population is far beyond this paper. The full ballot generation program can be found in the `/generator/` folder. The goal of the program is to produce ballots that rank every candidate in order of preference (see figure 4 on page 13). The program represents each voter in a population as belonging to a certain ideology. Each ideology has has a list of weighted preferences indicating how a voter for this ideology is likely to vote. The weighed preferences are generated via an abstraction. A single configuration object is used to create all ideologies.

Code example 2: Configuration object for creating ideologies

```
const ideologyConfig: Ideology[] = [{
    // The size of the ideology
    size: 0.2,
    // Spread of votes between candidates within ideology
    // Large -> steep spread, Small -> flat spread
    candidatePower: 0.2,
    // How inclined voters of this ideology are to vote for a
     ↪  candidate of this ideology
    // Higer -> more inclined
    ideologyPower: 1.2
}, {
//  ...
},
// ...
]
```

The program then works via 4 major steps:

1. It relates each candidate to an ideology based on the size of the ideology. A larger ideology has more candidates.

2. It assigns a base size for each candidate. Each ideology has a different spread of votes generated by a negative exponential function with a different constant. The size of each candidate is then multiplied with the size of their ideology and normalized. This gives a base size of all candidates.

3. Based on the base size of each candidate, a probability-list of voting for each candidate is created for every ideology. An ideology's probability-list represents how a voter aligned the ideology is expected to vote.

4. It creates a certain number of individual ballots. Each ballot is given an ideology and creates its own list preferences based via weighted random number generation according to the probability-list of the assigned ideology.

The output of the program is a set of ballots, where each ballot is an ordered set of preferences. The implementation is highly arbitrary and all constants in the program even more so. However, expanding on the generator program would be beyond the scope of this project. The program works in that it is possible to control variables to test for different election scenarios.

3.2 Constructing test scenarios

Constructing test scenarios is difficult task. Due to there being so many variables both in real elections and the simulated elections in the generator program, it makes it practically impossible to control all variables. Instead, 3 arbitrary test scenarios will be used. The election will be simulated 1000 times for each scenario and the election data will then run through the different voting methods and the results will be compared. The simulation scenarios are arbitrary but should allow for comparison of the different voting methods. All simulations will have 3 ideologies and 500 voters.

3.2.1 Scenario 1

The scenario is built around electing one candidate out of eight. The spread of votes within ideologies is flat and all ideologies are ”weak”. This means that the general spread of votes will be even. An example of the distribution of first-preference votes generated in this scenario can be seen in figure 1. The full configuration object can be seen in code example 3.

Code example 3: Configuration object for scenario 1

```
const candidates = 8
const seats = 1
const ideologies: Ideology[] = [{
    size: 0.3,
    candidatePower: 0.3,
    ideologyPower: 0.8
}, {
    size: 0.3,
    candidatePower: 0.3,
    ideologyPower: 0.7
}, {
    size: 0.4,
    candidatePower: 0.5,
    ideologyPower: 0.8,
}]
```

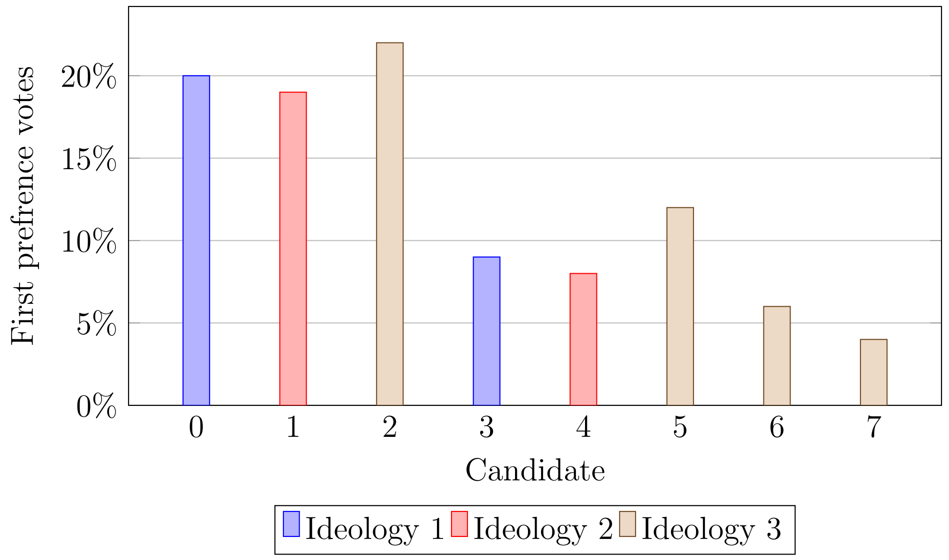

Figure 1: Example distribution of first-preference votes in scenario 1

3.2.2 Scenario 2

The second scenario elects 3 candidates out of 9 total. It features a normal spread of votes between candidates. An example distribution of first-preference votes generated in this scenario can be seen in figure 2. The full configuration can be seen in code example 4.

Code example 4: Scenario 2

```
const candidates = 9
const seats = 3
const ideologies: Ideology[] = [{
    size: 0.2,
    candidatePower: 0.5,
    ideologyPower: 1.2
}, {
    size: 0.3,
    candidatePower: 0.2,
    ideologyPower: 1
}, {
    size: 0.5,
    candidatePower: 0.3,
    ideologyPower: 0.8
}]
```

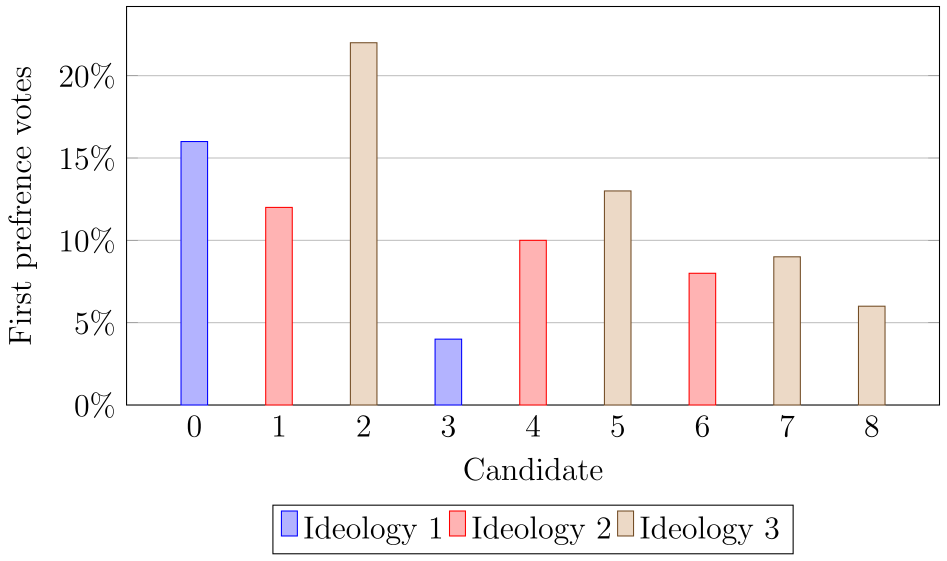

Figure 2: Example distribution of first-preference votes in scenario 2

3.2.3 Scenario 3

The second scenario elects 3 candidates out of 9 total. It features a steep spread of votes between candidates. An example distribution of first-preference votes generated in this scenario can be seen in figure 3. The full configuration can be seen in code example 5.

Code example 5: Scenario 3

```
const candidates = 9
const seats = 3
const ideologies: Ideology[] = [{
    size: 0.2,
    candidatePower: 1.8,
    ideologyPower: 0.6
}, {
    size: 0.3,
    candidatePower: 1.6,
    ideologyPower: 0.6
}, {
    size: 0.5,
    candidatePower: 2,
    ideologyPower: 0.6
}]
```

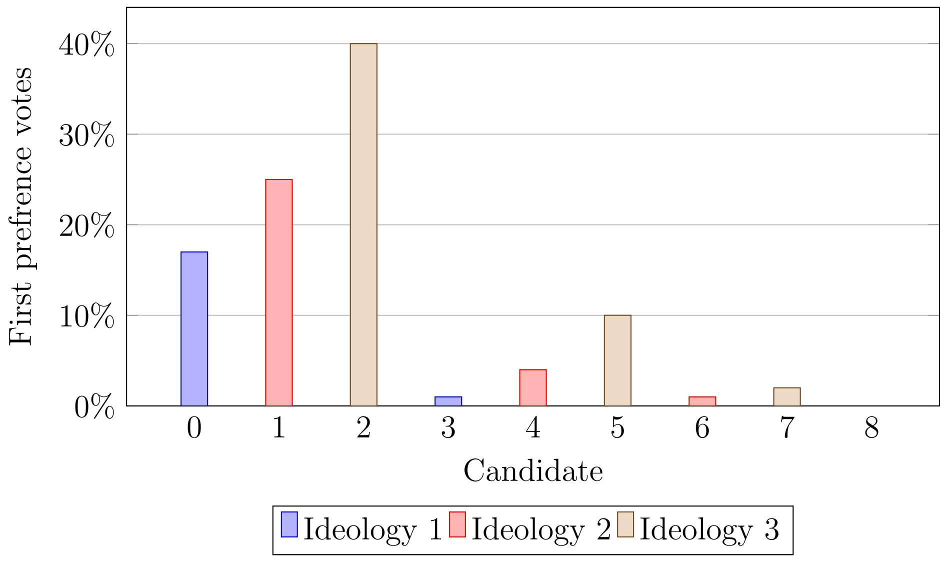

Figure 3: Example distribution of first-preference votes in scenario 3

4 Specification of voting methods

4.1 First-past-the-post

4.1.1 Description

The first-past-the-post voting method is a method designed for electing one or several candidates out of a set of candidates. Each voter has one (1) vote which may be allocated to any candidate. In a first-past-the-post election the N candidates with most votes get elected.

4.1.2 Justification

The method is simple and can easily be understood. The first-past-the-post method is widely used in for example the United Kingdom and United States. Comparisons with this method can also be easily made.

4.1.3 Pseudocode

```
let $V$ be the set of votes with $V_i$ being the number of votes for candidate $i$
let $N$ be number of candidates to be elected
sort $V$ based on $V_i$
slice $V$ between 0 and $N$
```

4.1.4 Implementation

The first-past-the-post program is implemented in a single file. The inputs of the program can be seen in code example 6.

Code example 6: Inputs for FPTP program

```
ballots: number[][] // Ex. [[0,1,2],[1,0,2],...]
seats: number // Ex. 2
```

The ballot data is first mapped to only include first preferences. Then the algorithm loops over the ballots and gets the sum of votes for each candidate. This process is shown in code example 7.

Code example 7: Processing and formatting of data

```
let results: Result[] = []
// Loops over prefrences and adds/increments result
for (var i = 0; i < firstPrefrences.length; i++) {
    // Finds index in results array of candidate at i
    let index = results.findIndex(obj => obj.cand ===
     ↪  firstPrefrences[i])
    // If candidate is not added to results
    if (index === -1) {
        results.push({ cand: firstPrefrences[i], votes: 1 })
    } else {
        // Increment count if already added in results
        results[index].votes += 1;
    }
}
```

The results array is sorted and sliced to only include the N candidates with most votes as seen in code example 8.

Code example 8: Finding and returning winning candidates

```
// Sorts array by number of votes
results.sort((a, b) => {
    if (a.votes > b.votes) {
        return -1
    } else if (a.votes < b.votes) {
        return 1
    }
    return 0
})
// Gets the first elements in the sorted array i.e the winners
let winners = results.slice(0, seats)
// Returns array of winners
return winners.map(cand => cand.cand)
```

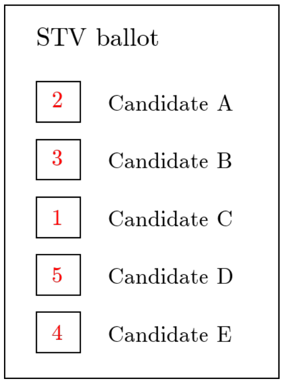

Figure 4: Sample STV ballot

4.2 Single transferable vote

4.2.1 Description

The Single Transferable Vote method is a more complicated algorithm than the first-past-the-post method. Voters are given ballots where they are supposed to rank candidates in order. The specific rules for if you need to rank all candidates or rank several candidates at the same value not can vary. For the sake of simplicity, in this implementation, all candidates must be ranked at unique and linear values. An example of a ballot can be seen in figure 4. This means that there is more data available for determining the result of the election which the method can take advantage of. Votes are, if needed, transferred between choices within a ballot, hence the name Single Transferable Vote. The method relies on the concept of a quota, the number of votes you need to be elected. The Droop-Quota, defined as $(\frac{\text{total valid poll}}{\text{seats+1}}) + 1$, is often used. If a candidate receives equal or more votes than the quota, he/she is elected. If the number of votes exceed the quota, a fraction of the votes are transferred to the next choice on the ballots that voted for the elected candidate. This process is repeated until there are no candidates with more votes than the quota. If there still are seats yet to be filled, the candidate with fewest votes is eliminated and his/her votes are transferred to the next choice on those ballots. This process is repeated until all seats are filled. The process can become quite complex with many ballots and transferring fractional votes. A computer is therefore essential in order to resolve large scale elections.

4.2.2 Justification

STV is an established voting method in use in several countries. It is used for parliamentary elections in Ireland, Malta and Australia as well as being used in local and regional elections across the world.

4.2.3 Pseudocode

```
let $quota$ be the quota
let $seats$ be the number of seats
let $winners$ be the list of elected candidates
let $votes$ be the set of votes with $votes_i$ signifying votes for candidate $i$
while $|winners| < seats$
    if any candidate $i \notin winners$ and $votes_i \geq quota$
        add $i$ to $winners$
        continue
    end if
    if any candidate $i$ where $votes_i > quota$
        multiply $votes_i$ with $(1 - \frac{\text{surplus votes}_i}{votes_i})$
        multiply next preferences for $votes_i$ with $\frac{\text{surplus votes}_i}{votes_i}$ and
        distribute into $votes$
        continue
    end if
    if $|votes| = seats$
        add all $votes \notin winners$ to $winners$
        continue
    end if
    $k :=$ index of smallest $votes_i$
    eliminate $votes_k$ and distribute all next-preference votes for candidate
$k$      into $votes$
end while
```

4.2.4 Implementation

The single transferable vote algorithm uses a tree structure to represent the state of the election. Every candidate is a top-level node with an attribute showing the number of current votes for the candidate. Each node has child nodes representing the next preferences of the voters for that particular candidate, see figure 5. It is easy to manipulate branches via recursion. Merging

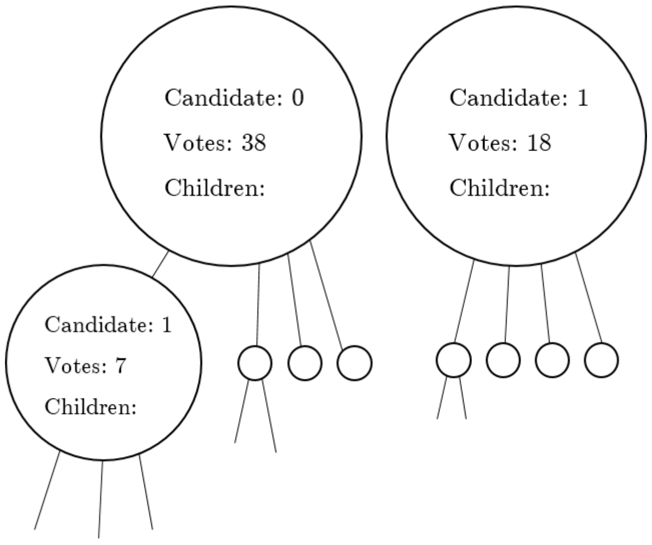

Figure 5: Simplified and shortened version of the tree structure. Each candidate is a top-level node with children, signifying the preferences of the voters for that candidate.

and multiplying branches is used when transferring votes from one candidate node to another. A candidate node is a Typescript class. It can be found in the `stv/candidate.ts` file and has the following properties and functions:

Code example 9: The CandidateNode class

```
class CandidateNode {
  public cand: number
  public count: number
  public children: CandidateNode[]
  // Add this and another branch together. Returns a new
   ↪  branch.
  add(other: CandidateNode): CandidateNode {
    //...
  }
  // Multiplies the count of this node and all its children
   ↪  and returns the new branch
  multiply(factor: number): CandidateNode {
    //...
    }
}
```

With the `add()` and `multiply()` functions it is possible to recursively modify branches. There are two different manipulations that need to be preformed on the tree: transferring surplus votes and eliminating candidates. Both functions rely on the `distribute()` function which is used to distribute a set of nodes onto the tree.

Code example 10: The distribute function

```
// Distributes a list of nodes onto the tree. Returns a new
↪ tree.
function distribute(nodeList: CandidateNode[], tree:
↪ CandidateNode[], winners: number[]): CandidateNode[] {
    // Loops through each candidate node in the list
    for (let i = 0; i < nodeList.length; i++) {
        let cand = nodeList[i];
        // Find if candidate is in the tree and/or is a winner
        let itExistsInTree = tree.some(obj => obj.cand ===
        ↪ cand.cand)
        let isWinner = winners.includes(cand.cand)

        // If it doesn't exist in tree (eliminated or no
        ↪ votes) or is a winner, distribute its children
        ↪ instead (recursive).
        if (!itExistsInTree || isWinner) {
            tree = distribute(cand.children, tree, winners)
        } else {
            // Find the index of candidate in tree
            let treeIndex = tree.findIndex(item => item.cand
            ↪ === cand.cand)
            // Add the trees/branches together
            tree[treeIndex] = tree[treeIndex].add(nodeList[i])
        }
    }
    return tree
}
```

In order to eliminate and transfer surplus votes from any candidate node A, the children of A are passed into the distribute function. In the case of transferring surplus votes, the votes for each child node is first multiplied by $\frac{\text{surplus votes}}{\text{total votes}}$ as outlined in the pseudocode in section 4.2.3. The full source code for functions that manipulate the tree structure can be found in `/stv/tree.ts`.

The main process of the program is found in the `stv/election.ts` file and runs the process defined in the pseudocode in section 4.2.3 on page 14.

4.3 Schulze

4.3.1 Description

The Schulze Method is a method used mainly to determine results in a single-winner election but can also be used to provide rankings and elect multiple candidates in a single election (Schulze 2011). Ballots are identical to those of the single transferable vote, see figure 4 on page 13. The method is compliant with the Condorcet Criterion, which means that if there is a candidate who the majority prefers in a pairwise comparison with every other candidate, that candidate wins. As mentioned, this method relies on comparing the preferences of every candidate to one another as shown in table 1. If there is a Condorcet winner the process is simple: declare the Condorcet winner a winner and run another iteration without that candidate. Repeat this process until all seats are filled. However, there can be Condorcet ties, where there is no Condorcet winner. There are multiple ways of resolving the tie, one of them being the Schulze Method.

The Schulze Method resolves ties by investigating possible paths between candidates. For example, if out of a total of 50 ballots 30 ballots prefer $B > A$, 28 ballots prefer $A > C$ and 27 ballots prefer $C > B$, a path from A to B can be created by going $A \to C \to B$. The path is said to have the *strength* of the weakest link in the path. In this example, the path strength between A and B is 27 due to the link between C and B having a strength of 27. There may be several paths between two candidates and the goal of the process is to find the strongest path between any candidate A and B, written as $p[A, B]$. By finding the strongest path between all nodes a result can be obtained where $p[X, Y] \geq p[Y, X]$ which means that candidate X wins. The Schulze Method also provides a linear ranking between all candidates, for example that $E > B > A > C > D$. By selecting the N top candidates you

Table 1: Pairwise comparison matrix used in the Schulze method. For example, 17 ballots prefer $A > B$ and 33 ballots prefer $B > A$.

	A	B	C	D	E
A		17	14	35	30
B	33		24	47	36
C	36	26		40	42
D	15	3	10		18
E	20	14	8	32	

can use the method for a multi-seat election. As the process can become very complex as the number of candidates grow, a computer is needed to resolve large elections.

The difficult problem in this method is finding the strongest path between candidates. The problem is called the widest path problem in graph theory. An efficient and relatively simple way to compute this problem is via the Floyd-Warshall algorithm which is used in the implementation. See 12 for the full algorithm.

4.3.2 Justification

The Schulze method is used by multiple organizations around the world. It has been used by various software organizations such as The Wikimedia Foundation, The Debian Project and Ubuntu. It is also used by political parties such as the Pirate Party in Sweden and various other countries. The process also differs greatly in both method and implementation from the two other voting methods used in this paper.

4.3.3 Pseudocode

```
let $d[i, j]$ be the number of ballots that prefer candidate $i$ to candidate $j$
let $p[i, j]$ be the strength of the strongest path from candidate $i$ to candidate $j$
for $i$ from 1 to $C$
    for $j$ from 1 to $C$
        if $(i \neq j)$
            if $(d[i, j] > d[j, i])$
                $p[i, j] := d[i, j]$
```

```
                        else
                                $p[i, j] := 0$
                        end if
                end if
        end for
end for
for $i$ from 1 to $C$
        for $j$ from 1 to $C$
                if $(i \neq j)$
                        for $k$ from 1 to $C$
                                if $(i \neq k$ and $j \neq k)$
                                        $p[j, k] := \max\ (p[j, k],\ \min\ (p[j, k], p[i, k]))$
                                end if
                        end for
                end if
        end for
end for
```

4.3.4 Implementation

The Schulze Method is implemented within the file `schulze/index.ts`. It takes two inputs: a list of ballots and the number of seats to be elected. A ballot is an ordered list of candidates. The output of the program is a list of winners. The program first creates the matrices for both the preferences and paths between candidates.

Code example 11: Data representation

```
// d[i][j] gives the number of ballots prefering i > j
let d: PairMap = getPairs(input)
// o[i][j] stores the strongest path between i and j
let p: PairMap = {}
// Number of candidates
let c = input[0].length
```

The program then runs the algorithm outlined in the pseudocode in section 4.3.3.

Code example 12: Main algorithm

```
// Computing strongest path strength. Variant of the
 ↪  Floyd-Warshall algorithm
for (var i = 0; i < c; i++) {
    for (var j = 0; j < c; j++) {
        if (i !== j) {
            if (d[i][j] > d[j][i]) {
                p[i][j] = d[i][j]
            } else {
                p[i][j] = 0
            }
        }
    }
}

for (var i = 0; i < c; i++) {
    for (var j = 0; j < c; j++) {
        if (i !== j) {
            for (var k = 0; k < c; k++) {
                if (i !== k && j !== k) {
                    p[j][k] = Math.max(p[j][k],
                     ↪  Math.min(p[j][i], p[i][k]))
                }
            }
        }
    }
}
```

5 Evaluation of results

5.1 Scenario 1

Scenario 1 featured 8 candidates where one was to be elected. Vote spread was flat, meaning that votes tend to be diverse in their preferences. This means that there is not an obvious winner present. This can be seen from

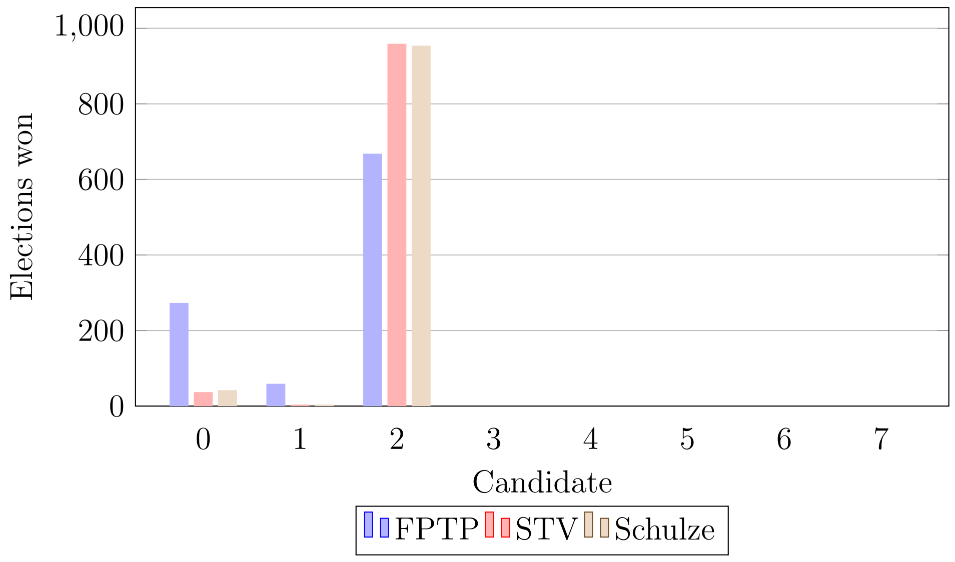

Figure 6: Results in scenario 1

Table 2: Key metrics for result in scenario 1

Method	Standard deviation	Misrepresentation	Most likely winners
FPTP	223	67%	2
STV	315	7%	2
Schulze	313	9%	2

the first-preference votes in the scenario seen in figure 1. The full result in this scenario can be seen in figure 6. The outlier in this scenario seems to be the fptp method, electing candidate 0 more often than both other methods. Key metrics for the results in this scenario can be found in table 2. Although the misrepresentation metric is not entirely accurate, especially when few seats are elected, it shows that fptp is not entirely representative. The STV and Schulze method seem to return very similar results.

5.2 Scenario 2

Scenario 2 featured 9 candidates of which 3 were to be elected. The distribution of votes was ”normal”, and as seen in figure 2, the election is very close with many candidates having a similar percentage of first-preference

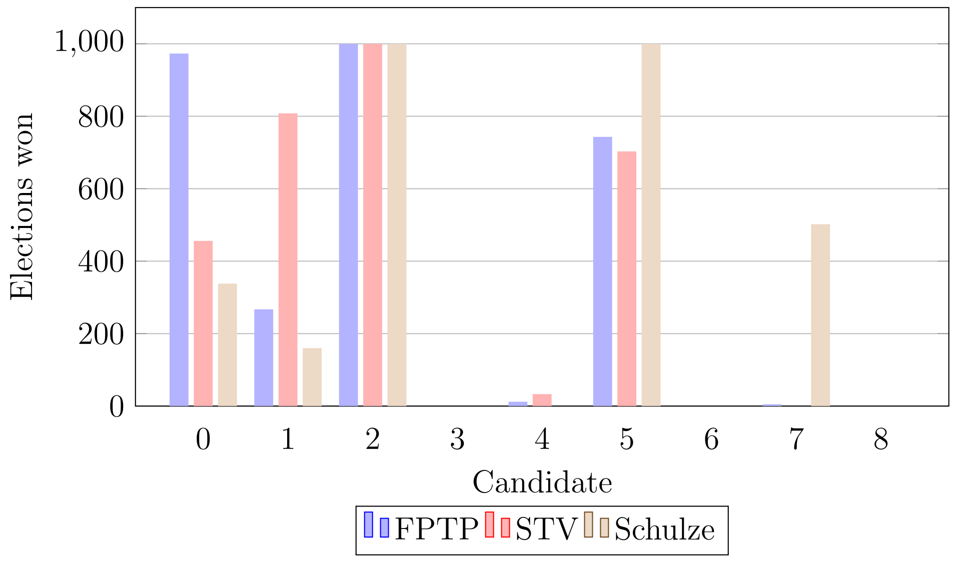

Figure 7: Results in scenario 2

Table 3: Key metrics for result in scenario 2

Method	Standard deviation	Misrepresentation	Most likely winners
FPTP	418	42%	0, 2 & 5
STV	388	14%	1, 2 & 5
Schulze	393	67%	2, 5 & 7

votes. The results obtained in scenario 2 are very interesting. FPTP tended to underrepresent candidate 1, and elected a candidate from ideology 2 only in 28% of elections, despite the ideology theoretically having 30% of popular support. The Schulze method also tended to underrepresent ideology 2. It instead overrepresented ideology 2, electing 3 candidates of ideology 2 in 50% of elections. This indicates that the Schulze method is not a representative method in a multi-seat election. The STV result showed less misrepresentation than the FPTP result, which was to be expected since the method has access to more data of any voter's preferences. The results in scenario 2 can be observed in figure 7 and the key metrics can be seen in table 3.

5.3 Scenario 3

Scenario 3 featured 9 candidates of which 3 were to be elected. The distribution of votes was steep. This can be seen by the first-preference votes in figure 3. The results obtained with each method in this scenario are practically identical. From this, it can be inferred that the results of the methods converge as the preferences in a population become more clear. The results obtained in this scenario can be seen in figure 8. The key metrics can be found in table 4.

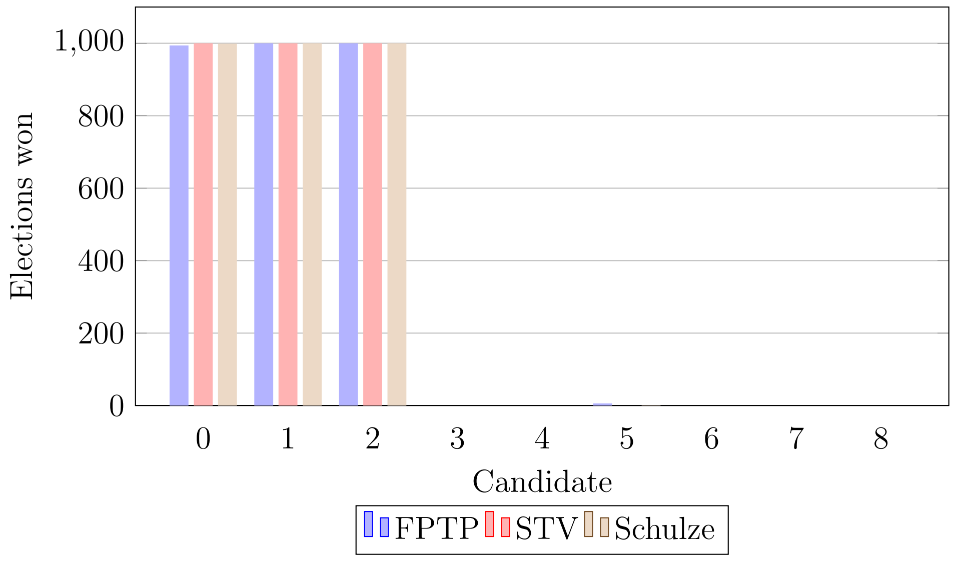

Figure 8: Results in scenario 3

Table 4: Key metrics for result in scenario 3

Method	Standard deviation	Misrepresentation	Most likely winners
FPTP	470	33%	0, 1 & 2
STV	471	33%	0, 1 & 2
Schulze	471	33%	0, 1 & 2

6 Conclusion

The results obtained in the three different scenarios provide some insight into the varying properties of the different voting systems. Each of the scenarios showed different behaviors for the different methods. For instance, the methods provided practically the same result in scenario 3 but wildly varying results in scenario 2. While the testing method used in this paper may not have been the most robust way of comparing the voting methods, several conclusions can be drawn from the data. I outline them in 4 points below.

1. Schulze did not seem like a proportional voting method in multi-seat elections. ...

2. In single-seat elections, Schulze and STV (and IRV in extension) seemed to provide similar results. ...

3. FPTP did not seem like a very proportional method in close elections. ...

4. STV seems to be the most proportional method out of the 3 tested, but is not without its flaws. ...

Summary: There is no perfect voting system. Generally, if you want proportional representation (which in a democracy you want), STV will likely be the best method to use out of the methods covered in this paper. Modeling of data needs more work, as well as including other methods (such as ...). More research is needed.

Todo:

- Write theory
- Add references where needed
- Explain metrics used to evaluate results
- Cut down on explanation of fptp
- Write abstract
- Finish summary

References

Schulze, Markus (2011). "A new monotonic, clone-independent, reversal symmetric, and condorcet-consistent single-winner election method". In: *Social Choice and Welfare* 36.2, pp. 267–303. ISSN: 1432-217X. DOI: `10.1007/s00355-010-0475-4`. URL: `http://dx.doi.org/10.1007/s00355-010-0475-4`.