Institut für Informationssicherheit

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Erweiterung des E-Voting-Systems Ordinos um weitere Wahlverfahren

Fabian Hertel

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr. Ralf Küsters
Betreuer/in:	M.Sc. Julian Liedtke
Beginn am:	14. Februar 2020
Beendet am:	9. Oktober 2020

Kurzfassung

In der aktuellen Zeit mit COVID-19 ändert sich einiges, was nicht erwartet wurde. Das könnte auch neuen Schwung in das Thema elektronische Wahlen bringen. Die Akzeptanz für solche Systeme ist nicht in der gesamten Gesellschaft vorhanden. Doch aktuell ist viel Umdenken zu verspüren.

Ordinos beschäftigt sich als E-Voting-System speziell mit der Verschlüsselung von Daten in geheimen Wahlen. Mit diesem sollen so wenig Informationen über das Ergebnis der Wahl preisgegeben werden wie nötig. So können Wahlblamagen und Rückschlüsse auf Stimmabgaben der Wähler verhindert werden. Für Ordinos existiert bereits eine Implementierung. Nun sollen das neue Wahlverfahren zum System hinzugefügt werden. Diese Arbeit untersucht Borda-Verfahren und unterschiedliche Condorcet-Methoden auf ihre Varianten und Implementierung. Nach der theoretischen Vorarbeit folgte auch die Umsetzung. Diese Ausarbeitung beschreibt die Algorithmen und untersucht sie auf Performanz und Geheimhaltung. Neben der Implementierung von Wahlverfahren soll sich das System weiterentwickeln, um später einen Einsatz in der Realität finden zu können. Auch hiervon findet sich eine Dokumentation zu allen größeren Änderungen in der Software.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Algorithmen

1 Einleitung

Unsere Gesellschaft bleibt nicht stehen. Besonders in Zeiten von COVID-19 gibt es Änderungen, von denen die wenigsten gedacht hätten, dass sie möglich sind. Wer hätte im Jahr 2019 noch geglaubt, dass in Deutschland Menschen in der Öffentlichkeit Mund-Nasen-Schutz tragen? Wer hätte gedacht, dass Schulen einen Monat stillstehen und Unterricht Monate lang ausfällt? Aufgrund eines Virus suchen Verantwortliche in allen Bereichen nach Möglichkeiten, persönlichen Kontakt zu vermeiden. Veranstaltungen und Treffen finden online statt. Ämter richten Portale ein, mit welchen sie ihre Dienste im Internet zur Verfügung stellen. Größere Wahlen fanden in Deutschland bis September nicht statt. An den Kommunalwahlen in Nordrein-Westfahlen hat sich beim Verfahren nichts grundlegend verändert, jedoch ist der Anteil an Briefwahlen gestiegen. Da nach über einem halben Jahr mit COVID-19 viele Maßnahmen noch nicht wegzudenken sind, lässt sich vermuten, dass wir noch eine Weile damit leben müssen und diese Pandemie eine Brandmarkung für die weite Zukunft sein wird.

Wir werden nicht noch jahrelang im Zustand von Mitte 2020 leben müssen, aber es wird noch lange Maßnahmen geben, welche eine neue Pandemie verhindern wollen. Da in den letzten Monaten so manche unerwartete Veränderung Richtung Digitalisierung spontan akzeptiert und umgesetzt wurde, ist nicht ausgeschlossen, dass sich auch Wahlen noch verändern können. Laut dem National Democratic Institute wurde schon im Jahr 2013 in 14 Ländern Internet Voting verwendet oder getestet (siehe [20a]). Internet Voting ist die Weiterführung von elektronischen Wahlen, im weiteren Verlauf auch E-Voting genannt, in welchem die Stimmabgabe online vom heimischen Computer geschehen kann.

E-Voting hat aber noch mehr Potenzial als das Wählen von daheim. Von Computern ausgewertete Stimmzettel können nach beliebigen Algorithmen ausgewertet werden. Die meisten existierenden E-Voting-Systeme setzen eine automatische anonyme Auszählung der Stimmzettel um. Es ist nicht bekannt, welche Stimme zu welchem Wähler gehört, jedoch ist bekannt, wie oft welcher Kandidat gewählt wurde. Zu detaillierte Informationen können aber zu Peinlichkeiten führen oder im Widerspruch zum Wahlgeheimnis stehen. Besonders bei kleinen Wahlen oder bei Informationen über einen zu kleinen Teil der Wählerschaft können Rückschlüsse auf einzelne Wähler gezogen werden. Besonders wenn sich die Menschen gegenseitig kennen, können sie beispielsweise bei einer einzelnen Stimme für einen Kandidaten oder eine Partei vielleicht vermuten, von wem diese gekommen sein könnte. Für Kandidaten kann es peinlich werden, wenn sie besonders wenig Stimmen erhalten. Auch dieser Effekt wirkt sich stark in kleinen Wahlen aus, in denen sich Wähler und Kandidaten vielleicht sogar gegenseitig kennen. Hier ist die Klassensprecherwahl ein gutes Beispiel.

Um diese Effekte zu verhindern, wurde Ordinos entwickelt. Ordinos ist ein E-Voting-System, welches mehr als nur das anonyme Auszählen von Stimmzetteln umsetzt. Es nutzt die Stärke von Verschlüsselungen, welche durch die elektronische Form möglich sind. Verschlüsselungen können mehr als nur die Identität des Wählers verheimlichen. Mit Encryption-Schemes, wie sie in Ordinos verwendet werden, lässt sich Tally-Hiding verwirklichen. Tally-Hiding bezeichnet das Verheimlichen

von Informationen, welche nicht relevant für das Ergebnis sind. Es ist möglich, allein den Sieger einer Wahl zu evaluieren, ohne eine weitere Information preiszugeben. Dabei kann die Reihenfolge aller anderen Kandidaten und die Anzahl der Wählerstimmen für jeden Kandidaten geheimgehalten werden. Verständlicherweise will man oft auch mehr Informationen. Mit Ordinos lassen sich Daten nach Wunsch veröffentlichen. Es müssen sich nur genug vertrauenswürdige Komponenten einig sein, was bekannt werden soll. Alle anderen Daten sind nicht einmal den Computern bekannt, welche das Ergebnis berechnen.

Die Encryption-Schemes wurden in anderen Veröffentlichungen untersucht. Nun gilt es das System zu implementieren. Erste Wahlverfahren waren bereits umgesetzt und nun sollen neue hinzugefügt werden. Dazu gehört das Borda-Verfahren, welches von Jean-Charles de Borda vorgeschlagen wurde und unterschiedliche Condorcet-Methoden. Condorcet-Methoden sind Wahlverfahren, welche einem Kriterium von Marie Jean Antoine Nicolas Caritat, Marquis de Condorcet entsprechen. So wird der Funktionsumfang der vorhandenen Software durch neue Algorithmen erweitert. Außerdem soll auch die Architektur der Implementierung angepasst werden, um einem realistischen System näherzukommen.

Die Arbeit beginnt mit einer Erklärung zur Theorie hinter Ordinos. Diese Informationen stammen aus einer Beschreibung von Ordinos (siehe [KLM+19]). Es folgt eine Vorlage für die Beschreibung von Algorithmen. Die neuen Wahlverfahren erforderten einige neue Wahlevaluationen, welche zu vergleichen sind. Die Vorlage beschreibt, wie Algorithmen beschrieben und bewertet werden. Häufiger verwendete Bestandteile werden hier erklärt.

Danach folgen die Dokumentation zum Code und Ideen zu zukünftigen Implementierungen. Zunächst werden die wahlunabhängigen Änderungen präsentiert. Eine theoretische Einführung zu Borda beschreibt die Möglichkeiten des Verfahrens. Ein Modell der Konfiguration beschreibt, wie die Varianten in Zukunft zu unterscheiden sein könnten. Es folgt die Dokumentation der umgesetzten Algorithmen und eine kurze Analyse zur Performanz und dem Tally-Hiding. Auch das Kapitel zu Condorcet beschreibt seine Varianten, welche davon umgesetzt und wie diese implementiert sind. Danach folgt ein direkter Vergleich der Performanz und des Tally-Hidings aller Algorithmen.

Da alle Berechnungen auf verschlüsselten Daten ausgeführt werden, kann die Laufzeit der Algorithmen sehr groß sein. Einzelne Vergleichsoperationen können einige Sekunden Berechnungszeit benötigen, weshalb diese nur so selten wie möglich verwendet werden sollten. Datenstrukturen wie Graphen können beispielsweise nicht objektorientiert dargestellt werden. Es ist entscheidend, die richtigen Algorithmen für Ordinos zu finden und zu verwenden. Die Definitionen der Wahlverfahren reichen meist aus, um hier eigene Algorithmen zu entwickeln.

Nicht betrachtet werden in dieser Arbeit die Sicherheit des Encryption-Schemes und der Komponenten im System. Es wird davon ausgegangen, dass die Methoden, welche in der *Grundlage der Ordinos-Implementierung* vorgestellt werden, sicher verwendet werden können. Zum Thema Sicherheit und Tally-Hiding wird nur untersucht, was der Aufruf dieser Methoden veröffentlicht. Die Vorteile von Ordinos gegenüber anderen E-Voting-Systemen werden nur geringfügig angesprochen. Es ist auch nicht Ziel dieser Arbeit ein fertiges Produktivsystem umzusetzen. Dazu fehlt die Infrastruktur und die Software-Architektur ist noch zu weit davon entfernt. Stattdessen sollen Evaluationalgorithmen, welche während der Auswertung ablaufen könnten, entwickelt und mit der aktuellen Software getestet werden.

2 Ordinos-Theorie und Implementierung

Um die Basis und Vorraussetzungen dieser Arbeit zu zeigen, wird zunächst die Theorie hinter Ordinos knapp geschildert und der Stand der Software zu Beginn dieser Implementierung beschrieben. Mit der Theorie hinter Ordinos werden Fachbegriffe erläutert, welche in der folgendenen Ausarbeitung auftauchen. Dabei wird inhaltlich vorgegangen und somit immer im Kontext erklärt. Um schnell die Erklärung einer einzelnen Bezeichnung zu finden, ist jeder Begriff an genau einer Stelle fettgedruckt. Im entsprechenden Absatz ist die Bedeutung auszulesen. Bei Verwendung werden die Begriffe im Folgenden kursiv gedruckt. Aber nicht jedes kursiv gedruckte Wort ist hier nachzulesen, da oft im Kontext des jeweiligen Kapitels spezielle Ausdrücke, wie zum Beispiel Variablennamen, verwendet werden. Diese Definitionen gelten im Rahmen von Ordinos und speziell dieser Arbeit. Meist haben die Bezeichnungen aber dieselbe oder eine ähnliche Bedeutung im Kontext anderer Software und anderer E-Voting Systeme. Grundlage ist dabei die Veröffentlichung von Prof. Dr. Ralf Küsters (siehe [KLM+19]). Im zweiten Teil dieses Kapitels ist der Stand der Software zu Beginn der Arbeit festgehalten, um die Basis der neuen Implementierung darzustellen.

2.1 Begriffe zur Ordinostheorie

Es werden nur die Komponenten und Vorgänge von Ordinos betrachtet, welche im Kontext der aktuellen Implementierung interessant sind. In anderen Veröffentlichungen zu Ordinos gibt es mehr Komponenten, die in der aktuellen Umsetzung jedoch noch keine Rolle spielen. Manche Bezeichnungen werden für diese Arbeit neu eingeführt und tauchen in anderen Veröffentlichungen zu Ordinos nicht auf. Der folgende Abschnitt ist eine Mischung aus Beschreibung der Abbildung 2.1 und einem Stichwortregister. Jeder Fachbegriff wird in einem Abschnitt definiert, wobei auch mehrere Begriffe pro Abschnitt möglich sind. Im Gesamten stellt der Text den Ablauf einer Wahl dar. Möglich ist, dass Bezeichnungen verwendet werden, deren Erklärung erst noch folgt.

Die Erläuterungen wurden aufgeteilt in die unterschiedlichen Phasen einer Ordinoswahl: Setup-Phase, Voting-Phase und Tallying-Phase. Die Setup-Phase ist zur Initialisierung aller Komponenten da, sodass diese wissen, wie sie sich zu verhalten haben. Während der Voting-Phase geben die Wähler ihre Stimme ab. In der Tallying-Phase wird das Ergebnis evaluiert und veröffentlicht.
Die Abbildung auf der nächsten Seite beschreibt die grobe Architektur eines Ordinos-Systems und dessen Ablauf.

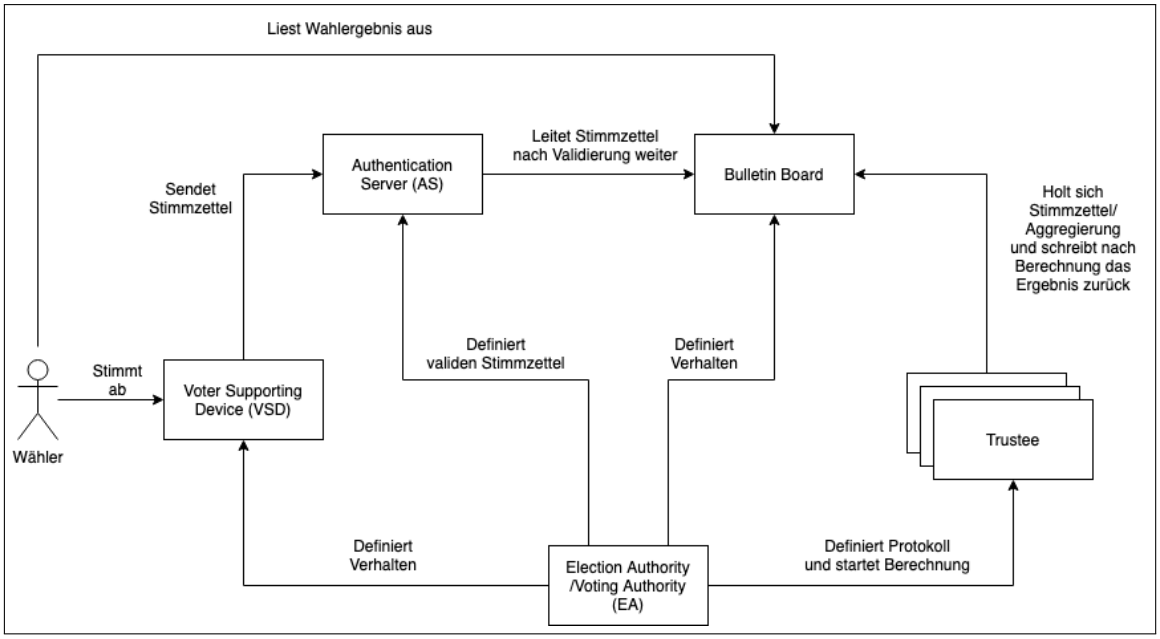

Abbildung 2.1: Modell der Ordinos Komponenten beim Ablauf einer Wahl

Setup-Phase

Zu Beginn bestimmt die **Election Authority** die Wahlparameter. Sie legt fest, welche Wahl mit welcher Konfiguration durchgeführt wird. Die **Election Properties** sind ein Objekt, welche diese Informationen enthalten. Den anderen Komponenten gibt die *Election Authority* Anweisungen, wie sie sich zu verhalten haben.

In der Setup-Phase einer Ordinos-Wahl werden Schlüssel generiert. Der *AS* und die *Trustees* generieren zusammen einen **Public Key (PK)** und die **Shared Keys (SKs)**. Die Erklärungen zu *AS* und *Trustee* folgen später. Mit dem *PK* kann verschlüsselt werden. Die *SKs* werden benötigt zum Entschlüsseln. Ein einzelner *SK* reicht dazu aber meist nicht aus. Bei der Generierung von Schlüsseln wird ein **Threshold** definiert, welcher angibt, wie viele *SKs* benötigt werden, um Daten zu entschlüsseln. Jeder *SK* ist nur einem einzigen *Trustee* bekannt und jeder *Trustee* kennt nur einen *SK*. Somit definiert der *Threshold*, wie viele *Trustees* für die Wahl korrekt funktionieren und vertrauenswürdig sein müssen. Um eine höhere Ausfallsicherheit zu haben, ist dieser meist nicht gleich der Anzahl der *Trustees* sondern kleiner. Jedoch leidet die Sicherheit des Systems unter einem zu kleinen *Threshold*. Er gibt an, wie viele von Angreifern kontrollierte *Trustees* ausreichen, um Wahlgeheimnisse zu lüften.

Es sind unterschiedliche **Encryption Schemes** für Ordinos möglich. Diese beschreiben, wie **Ciphertexte** und deren Operationen aussehen. Operationen sind meist arithmetisch oder logisch. *Encryption Schemes* sollten für Ordinos zum Beispiel Addition, Multiplikation und Vergleichsoperatoren wie "größer" oder "gleich" unterstützen. *Ciphertexte* verraten nichts über deren Klartext. Das Ergebnis solcher Operationen ist wieder ein *Ciphertext* und auch während der Berechnung dürfen keine Information über die Werte der Operanden bekannt werden. Die Operationen werden sinngemäß

auf den Klartexten ausgeführt. *Ciphertexte* können mit *SKs* wieder entschlüsselt werden. Je nach *Encryption Scheme* sind auch für manche arithmetische und logische Operationen *SKs* nötig, aber auch hier sind Rückschlüsse auf Klartexte unmöglich.

Voting-Phase

Ein Wähler gibt seine Entscheidung auf seinem **VSD** ein. Dieses sollte dafür sorgen, dass der Stimmzettel das richtige Format hat und verschlüsselt wird. Verschlüsselt wird mit dem *Public Key (PK)*, der vor der Wahl geniert wurde und frei zugänglich ist. Außerdem generiert es ein Zero-Knowledge Proof (ZKP), welches versichert, dass bestimmte Regeln für den Inhalt des Stimmzettels gelten. Samt der ID des Wählers schickt das *VSD* die verschlüsselten Daten und das ZKP and den *AS*.

Der **Authentication Server (AS)** ist im Gegensatz zum *VSD* und zum Wähler vertrauenswürdig. Dieser prüft mit dem ZKP, ob der Stimmzettel korrekt ist, ohne dabei den Inhalt auszulesen. Gegenüber dem *AS* authentifiziert sich der Wähler mit seiner ID. Es wird geprüft, dass nicht mit einer Wähler-ID doppelt abgestimmt wird und ob die ID korrekt ist. Nach dem Validieren leitet der *AS* den Stimmzettel and das *Bulletin Board* weiter.

Das **Bulletin Board** enthält alle öffentlichen Informationen und kann von jedem eingesehen werden. Da jedes Wahlgeheimnis verschlüsselt ist, wird nichts verraten durch das Anzeigen aller abgegeben Stimmen. Nach *Evaluation* der Wahl ist auch das Wahlergebnis hier zu finden.

Bei vielen Wahlen ist es möglich Vorberechnungen durchzuführen, um Aufwand bei der *Evaluation* zu sparen. Dabei werden Stimmzettel zu einem Objekt zusammengefasst, wie es das Wahlverfahren erlaubt. Dieser Vorgang und das dabei entstehende Objekt heißen **Aggregierung**. Zum Beispiel können Punkte oft zusammenaddiert werden, sodass für jeden Kandidaten die aktuelle Gesamtpunktzahl gespeichert wird. Bedingung hierfür ist, dass beim Addieren sowohl die Summanden als auch die Summe geheim bleiben. Während der gesamten Berechnung bleiben alle Wahlgeheimnisse verschlüsselt und der Ciphertext des Ergebnisses ermöglicht keine Rückschlüsse auf die Eingaben. Dies ist für die möglichen Verschlüsselungen für Ordinos gegeben. Oft sind diese Vorberechnungen sogar ohne *SKs* und *Trustees* möglich.

Tallying-Phase

Wenn alle Wähler ihre Stimmen abgegeben haben oder die Zeit abgelaufen ist, findet die **Evaluation** durch *Trustees* statt. Dazu holen sich diese die *Aggregierung* vom *Bulletin Board*. Nun wird ein Protokoll ausgeführt, welches die geforderten Informationen berechnet. Die geforderten Informationen könnten zum Beispiel eine Liste der besten drei Kandidaten sein. Das Verstecken aller anderen Informationen nennt sich Tally-Hiding. Das Ergebnis und alles, was beim Berechnen öffentlich wurde, wird wird an das *Bulletin Board* zurückgeschickt. Im Optimalfall wird nur das gesuchte Wahlergebnis, beispielsweise nur der Sieger, öffentlich.

Um Informationen aus den Ciphertexten zu ziehen, müssen die **Trustees** gemeinsam arbeiten. Jede besitzt einen *Shared Key (SK)*, der geheim für die Öffentlichkeit und alle anderen *Trustees* ist. Ein einzelner *SK* kann keine Informationen aus einem Ciphertext auslesen. Abhängig vom Encryption Scheme sind auch für manche arithmetische und vergleichende Operationen *SKs* notwendig. Keine

Operation darf jedoch Informationen über Klartexte verraten. Vergleichsoperationen können prüfen, welche von zwei Zahlen die größere ist oder ob diese denselben Wert haben. Durch Kommunikation der *Trustees* wird mit diesen nun das Wahlergebnis evaluiert. Abhängig vom *Threshold* der Keys muss immer eine Mindestzahl von *Trustees* vertrauenswürdig und aktiv bleiben.

2.2 Grundlage der Ordinos-Implementierung

Zu Beginn dieser Arbeit waren bereits fertige Wahlsysteme mit Python implementiert. Somit mussten nur die vorhandenen Komponenten für neue Wahlen neu zusammengesetzt werden. Hierzu gehört die Ciphertext-Klasse, welche eine Verschlüsselung mit Paillier umsetzt. Diese verschlüsselt Integer-Zahlen zu Ciphertexten, welche ohne *SKs* nichts über ihren ursprünglichen Wert verraten. Einfache mathematische Operationen sind mit den Ciphertexten möglich, auch ohne den Schlüssel zu besitzen. Hierzu gehören zum Beispiel das Addieren und Subtrahieren von zwei verschlüsselten Werten oder Multiplizieren mit einer unverschlüsselten Zahl.

Für die Protokolle *Greater-Than*, *Equals*, *Multiplicate* und *Decrypt* werden jedoch *Trustees* benötigt. **Greater-Than** entspricht einer "größer-gleich-Abfrage". Mit **Equals** kann geprüft werden, ob zwei Werte gleich groß sind. **Multiplicate** erlaubt die Multiplikation von zwei verschlüsselten Werten. Das **Decrypt** entschlüsselt Ciphertexte. Dabei kommunizieren die *Trustees*, um gemeinsam mit ausreichend SKs ein Ergebnis zu berechnen, ohne mehr preiszugeben als gefordert. Bei *Greater-Than* und *Equals* muss beim Aufruf abgeschätzt werden, wie groß die verglichenen Zahlen sein können. Mit einer Bitzahl wird definiert, wie groß der Zahlenraum ist. Werte, welche verglichen werden, müssen in diesem Bereich liegen. Ansonsten sind falsche Ausgaben möglich. Bei einem Vergleich der Punktzahlen x und y, muss bei *Equals* und *Greater-Than* eine Bitzahl l angegeben werden für welche gilt: $2^l > x \wedge 2^l > y$. Bei $l = 16$ wäre der Bereich von 0 bis $2^{16} = 65535$ abgedeckt. Je kleiner der Zahlenraum ist, desto schneller ist die Berechnung. *Greater-Than* ruft sich rekursiv mit halbierter Bitzahl auf und auch Equals ist mit größerer Bitzahl aufwändiger. Durch die Unterschiede in der Laufzeit wird später bei der Performanz-Analyse die Größenordnung der Parameter betrachtet. Parallel zu dieser Arbeit entstand die *Arithmetic Black Box (ABB)*. Mit dieser können alle genannten Operationen auf Ciphertexten durchgeführt werden, ohne sich Gedanken darüber machen zu müssen, ob dazu *Trustees* benötigt werden oder nicht.

Auch die Generierung von öffentlichen und privaten Schlüsseln war implementiert. Der *PK* wird im System gespeichert, um Zahlen zu verschlüsseln, und die *SKs* werden zur Erstellung der *Trustees* generiert. *Trustees* sind als Objekte implementiert, welche zu Beginn einen *SK* überreicht bekommen. Über einen *Connector* verbinden sich diese und kommunizieren miteinander. Mit einem Protokoll können sie gestartet werden. Wenn genug *Trustees* parallel dasselbe Protokoll mit denselben Eingabeparametern berechnen, liefern sie nach der Berechnung das Ergebnis des Protokolls. So ist die Evaluation von Wahlen möglich. Evaluationsprotokolle verwenden die Basisprotokolle *Greater-Than*, *Equals*, *Decrypt* und *Multiplicate*. Diese sind die Grundbausteine einer Wahlevaluation.

Durch ein relatives Mehrheitswahlsystem, IRV-Voting und die deutsche Bundestagswahl war bereits die vollständige Umsetzung erster Wahlverfahren geschehen. Neben der Evaluation wurden auch die Generierung von zufälligen Stimmzetteln und eine *Aggregierung* umgesetzt. Diese Implementierungen dienten als Basis und Vorlage für diese Arbeit.

3 Erklärungen zu Algorithmen und Schnellanalyse

Die Algorithmen, welche durch Ordinos implementiert werden, müssen auf eine eigene Art und Weise ausgewählt und untersucht werden. Nur ganz bestimmte Funktionsaufrufe werden die Laufzeit bestimmen. Deshalb wird besonders darauf geachtet, dass diese möglichst selten vorkommen. Es ist sehr sinnvoll, durch Vorberechnungen die Stimmzettel in ein bestimmtes Format zu bringen, welches Zeit bei der Evaluation spart. Bei einer Paillier-Verschlüsselung hängt die Laufzeit hauptsächlich von der Anzahl von Vergleichsoperationen ab. Additionen hingegen spielen für die Laufzeit so gut wie keine Rolle. Durch die großen Unterschiede entsteht die Aufgabe, einen optimierten Algorithmus für dieses System zu finden. Das Konzept und die Entwicklung der Algorithmen waren Teil dieser Arbeit, welche allein aus der Definition der Wahlverfahren resultierten. Nur im Falle der Schulze-Methode wurde ein Algorithmus aus einer Quelle übernommen. Deshalb werden sie als Pseudocode definiert und nach Performanz und Tally-Hiding untersucht werden. Der Pseudocode könnte zum Beispiel folgendermaßen aussehen:

Algorithmus 3.1 Example

```
input   points ← list of encrypted points per candidate
output  1 if product of candidates with even index is bigger than product of candidates
        with odd index or equal else 0
procedure OddVsEven(points)
    evenProduct ← enc(1)
    oddProduct ← enc(1)
    for all cand in 0..points.length − 1 do
        if cand % 2 is 0 then
            evenProduct ← multiplicate(evenProduct, points[cand])
        else
            oddProduct ← multiplicate(oddProduct, points[cand])
        end if
    end for
    return decrypt(greaterThan(evenProduct, oddProduct))
end procedure
```

Dieser Algorithmus wurde willkürlich gewählt, um ein Beispiel anzugeben. Er spielt im folgenden keine Rolle mehr.

Die Funktion *enc* gibt immer einen Ciphertext einer unverschlüsselten Zahl aus. Mit EmptyList() kann eine neue Liste der Länge null erstellt werden. Mit der Funktion *append* werden Elemente zur Liste hinzufügt. EmptyList(n) erstellt eine Liste der Länge n in der jeder Eintrag 0 ist. EmptySquareMatrix(n) erstellt eine quadratische Matrix. Hier können keine Elemente hinzugefügt,

sondern nur überschrieben werden. Listen und quadratische Matrizen besitzen das Attribut *length*. Die Protokolle *Greater-Than*, *Equals*, *Multiplicate* und *Decrypt*, wie sie in Kapitel 2 angesprochen wurden, werden durch die Funktionen *greaterThan*, *equals*, *multiplicate* und *decrypt* verwendet.

Die Schnellanalyse besteht aus farblichen Markierungen im Pseudocode und einer Tabelle. Sie betrifft sowohl die Performanz als auch das Tally-Hiding.

3.1 Performanz-Schnellanalyse

Da bis jetzt nur Paillier implementiert ist, gehen wir davon aus, dass *Greater-Than*, *Equals* und *Multiplicate* für die Laufzeit ausschlaggebend sind. Deshalb sind diese im Pseudocode blau markiert. Der Vollständigkeit halber wird aber auch *Decrypt* berücksichtigt. Dieses benötigt deutlich mehr Zeit als beispielsweise eine Addition oder Subtraktion, im Vergleich zu den anderen drei ist der Einfluss jedoch sehr gering. Diese vier Operationen werden nach ihrer Häufigkeit untersucht. Alle anderen werden vernachlässigt, da sie mit einer sicheren Verschlüsselung einen verschwindend geringen Einfluss auf die Laufzeit haben. Die genannten verschlüsselten Operationen werden abhängig von n, die Anzahl der Kandidaten, und anderen Parametern gezählt. n beschreibt im gesamten Dokument die Anzahl von Kandidaten. Tabellen wie Tabelle 3.1 geben die einzelnen Werte an und zeigen Beispiele für Laufzeiten (Lfz.). Die Berechnungsdauer wird mit fünf und mit 20 Kandidaten berechnet. Wenn der Algorithmus weitere Parameter besitzt, existieren auch Beispiele für deren unterschiedliche Werte. Für die Condorcet-Methoden existiert eine Vorberechnung, welche in den Laufzeiten immer einberechnet ist. Es wird von einem Computer aus dem SEC-Pool der Universität Stuttgart ausgegangen, der 3 *Trustees* simuliert mit einem *Threshold* von 2. Die Laufzeitwerte für diese Umgebung sind in Abbildung 4.6 zu finden. Die Hardware entspricht nicht einer realistischen Umsetzung, jedoch können mit konkreten Werten Verhältnismäßigkeiten abgeschätzt werden. Diese Tabelle würde den oberen Code beschreiben:

	Greater-Than	Equals	Multiplicate	Decrypt
Anzahl	1[Wähler]	0	n	1
Lfz. (5 Kandidaten)	00:00:58 h	0	00:00:06 h	00:00:00 h
Lfz. (20 Kandidaten)	00:00:58 h	0	00:00:26 h	00:00:00 h

Tabelle 3.1: Zeitkritische Operationen für Beispiel

Die Anzahl gibt an, wie oft diese vier Operationen jeweils aufgerufen werden. Oft sind diese in Abhängigkeit von Parametern angegeben. Meist reicht n als Parameter aus. Darunter sind die Laufzeiten für die genannte Umgebung angegeben, aufgeteilt in die vier Funktionen. Wenn nun die Gesamtlaufzeit abgeschätzt werden soll, muss eine Zeile mit ihren Werten aufaddiert werden. So würde hier die Laufzeit insgesamt circa eine Stunde und 24 Minuten bei 20 Kandidaten betragen.

Equals und *Greater-Than* sind meist hauptverantwortlich für die Laufzeit. Diese brauchen deutlich länger als *Multiplicate* und *Decrypt*. Außerdem werden bei *Equals* und *Greater-Than* Bitzahlen unterschieden. Die Laufzeit dieser Funktionen hängt davon ab, wie groß der benötigte Zahlenraum ist. Mehr Informationen sind unter Abschnitt 2.2 zu finden. Meist werden Gesamtpunktzahlen verglichen, welche durch $h \cdot n_{voter}$ begrenzt werden können. Hierbei ist n_{voter} die Anzahl an Wählern und h eine Konstante des Wahlverfahrens. Das Produkt kann sehr groß werden. Manchmal

vergleicht der Algorithmus aber auch nur um eine verschlüsselte Anzahl von Kandidaten mit einer der beiden Funktionen. Die Anzahl von Kandidaten ist oft nicht größer als 16. In diesem Fall würden 4-Bit Operationen ausreichen, welche deutlich schneller sind als beispielsweise 16 oder 32 Bit. In eckigen Klammern ist angegeben, ob abhängig von der Wählerzahl oder der Kandidaten geprüft wird. Um dies zu unterscheiden, werden die Stichwörter *Wähler* oder *Kandidaten* verwendet. Die Zahlen können für diese sehr unterschiedlich groß sein abhängig vom Wahlverfahren und der Zahl von Wählern und Kandidaten. Um Beispiellaufzeiten angeben zu können, werden pauschale Bitzahlen für beide Kategorien gewählt. Bei der Berechnung von Laufzeiten werden bei Gesamtpunktzahlen 16-Bit und bei Anzahl von Kandidaten 4-Bit Operationen betrachtet, auch wenn zum Beispiel die 20 Kandidaten betrachtet werden. Im Code wird die Bitzahl automatisch an die richtige Größe angepasst. Hier begrenzen wir uns der Einfachheit halber auf diese beiden Fälle. Die Dauer der Berechnung einer einzelnen verschlüsselten Operation ist in Abbildung 4.6 auszulesen. Die Summe dieser vier Laufzeiten entspricht circa der Laufzeit des gesamten Algorithmus. Wenn die Summe jedoch null oder beinahe null ist, spielen die anderen Operationen im Verhältnis eine größere Rolle. Dann ist die Laufzeit sehr klein im Vergleich zu anderen Algorithmen.

3.2 Tally-Hiding-Schnellanalyse

Eine weiteren Zeile der Tabelle und farbliche Markierungen veranschaulichen, wo und wie viel Tally-Hiding der Algorithmus umsetzen kann. Im Pseudocode sind alle Variablen, welche verschlüsselte Daten enthalten, grün markiert. Bei verschlüsselten Listen oder ähnlichen Datentypen kann die Länge ausgelesen werden. Dies sollte immer eine bekannte Größe sein, wie zum Beispiel die Anzahl der teilnehmenden Kandidaten.

Die Funktionen *Greater-Than*, *Equals* und *Multiplicate* benötigen zwar *SKs*, jedoch ist ihr Ergebnis weiterhin verschlüsselt. Bei den Vergleichsoperationen ist dies eine verschlüsselte Eins oder Null. Solange kein *Decrypt* verwendet wird im Algorithmus, werden keine Wahlgeheimnisse öffentlich. Da aber gewisse Informationen bekannt werden sollen, wird das *Decrpyt* meist am Ende dann auf solch einem verschlüsselten booleschen Wert ausgeführt, der angibt, ob ein Kandidat zu den Siegern gehört oder nicht. Das Ergebnis eines *Greater-Than* kann nicht direkt als Bedingung einer If-Verzweigung verwendet werden. Eine Pfadentscheidung im Code ist kein Geheimnis. In manchen Fällen kann die Pfadentscheidung aber umgangen werden (siehe IfThenElse in Abschnitt 3.4). Da nur *Decrpyt* Informationen aus verschlüsselten Daten auslesen kann, ist dieses mit rot hervorgehoben. Es wird untersucht, welches Geheimnis es an dieser Stelle lüftet. Gemeinsam in der Tabelle mit den Laufzeitanalyse ist in einer Kurzform beschrieben, was entschlüsselt wird. Für das Pseudocodebeispiel könnte die zusätzliche Zeile so aussehen:

Entschlüsselung	$\prod_{cand \in Gerade} Gesamtpunkte[cand] \geq \prod_{cand \in Ungerade} Gesamtpunkte[cand]$

Tabelle 3.2: Entschlüsselung für Beispiel

Diese Zeile versucht möglichst kurz die veröffentlichten Informationen zu beschreiben. Hier steht die Menge *Gerade* für alle Kandidaten mit geradem Index und *Ungerade* für alle Kandidaten mit ungeradem Index. Dabei kommen neben individuellen Parametern folgende Stichwörter und Kürzel vor:

Position und *Gesamtpunkte* sind Listen, welche für jeden Kandidaten die entsprechende Information enthält. *Max* definiert das Maximum einer Menge von Zahlen. Ein $\forall(Kandidat) : \ldots$ deutet darauf hin, dass die folgende Information für jeden Kandidaten entschlüsselt wird. $\prod$ bildet das Produkt einer Menge aus Zahlen, wie es auch der sonstigen Verwendung entspricht. Es wird immer nur die Gesamtaussage der Zeile öffentlich. In diesem Fall wird nur die Aussage des Vergleichsoperators bekannt und nicht das Produkt. Ein "—" Eintrag bedeutet, dass alles geheim bleibt. Wenn eine Aussage nur unter einer bestimmten Konfiguration entschlüsselt wird, ist dies mit $\Rightarrow$ gekennzeichnet.

Die Erklärung, wie es durch den Algorithmus zu der entschlüsselten Information kommt, ist vor oder nach dem Algorithmus zu finden. Ein Vergleich aller Algorithmen im Bezug auf ihr Tally-Hiding findet in Kapitel 8 statt.

3.2.1 Wichtiger Hinweis zu verschlüsselten Werten

In Algorithmen gibt es immer wieder pseudo-verschlüsselte Werte. Ein Wert, der verschlüsselt ist, muss deshalb noch lange nicht geheim sein. Es wird davon ausgegangen, dass Trustees jeden Schritt, jede Pfadentscheidung und jede Zufallsgenerierung öffentlich durchführen. Es ist ihnen nicht möglich, während der Evaluation ein Geheimnis zu generieren. Wenn während eines Algorithmus ein Wert verschlüsselt werden soll, ist jedem Beobachter der Plaintext bekannt. Erst wenn dieser Wert mit einem sicher verschlüsselten Wert verrechnet wird, erhält das Ergebnis für den Beobachter einen nicht nachvollziehbaren Wert. Sicher verschlüsselte Werte haben alle ihren Ursprung auf den Geräten der Wähler. Der dort generierte Zufall, der zur Verschlüsselung verwendet wurde, darf niemals öffentlich werden. Solange mindestens ein sicher verschlüsselter Wert in ein Ergebnis eingerechnet wurde, ist auch dieses sicher. In den Algorithmen entstehen solche verschlüsselte aber nicht geheime Werte durch die *enc*-Methode. Es ist kein Sicherheitsrisiko verschlüsselte bekannte Werte zu verwenden, da jede Operationen mit einer geheimen Information automatisch ein geheimes Ergebnis liefert. Es ist somit weiterhin nur möglich Wahlgeheimnisse zu lüften, indem die *Decrpyt*-Methode verwendet wird.

3.3 Verwendete Datenstrukturen

Um die Algorithmen besser zu verstehen, werden hier die gängigen Datenstrukturen erklärt. Diese werden häufiger in den verschiedenen Wahlmethoden verwendet.

Punktelisten

Meist tragen diese Listen den Variablennamen *points*. Es sind Listen, welche für jeden Kandidaten eine Punktzahl enthalten. Die Kandidaten werden im System als Indizes gesehen. Durch diese Indizes ist eindeutig, welche Punktzahl welchem Kandidaten zugeordnet ist.

Duell-Matrix

Das Ziel dieser Tabelle ist es, Priorisierungen zwischen Kandidaten darzustellen. Sie enthält für jeden Kandidaten jeweils eine Spalte und eine Zeile. Eine Null in einer Zelle bedeutet, dass der Zeilenkandidaten weniger Priorität hat als der Spaltenkandidat. Wenn der Zeilenkandidat höhere Priorität als der Spaltenkandidat erhält, wird die Zelle mit einer Eins befüllt. Wenn nicht genauer spezifiziert gehen wir von einem $\geq_p$-Vergleich bei den Prioritäten aus. Falls also zwei Kandidaten dieselbe Priorität haben, wird in den Zellen dieses Duells eine Eins eingetragen. Bei einer Variablen, welche mit dem Wort *strict* gekennzeichnet ist, handelt es sich um einen $>_p$-Vergleich. Hier ist bei Gleichstand jeweils eine Null eingetragen.
Abhängig vom Wahlsystem muss sich aus der Priorisierung keine eindeutige Hierarchie bilden. Es gilt nicht unbedingt eine Transitivität, womit Zirkelbezüge in der Priorisierung durchaus möglich sind. In den beiden Tabellen hätte Kandidat A höhere Priorität als B und C. Die Kandidaten B und C liegen auf derselben Ebene.
Im Algorithmus tragen Duell-Matrizen oft den Namen *duelMatrix*. Diese Darstellung entspricht auch einer Adjazenzmatrix in einem gerichteten Graphen, in dem die Pfeile den Priorisierungen entsprechen.

	A	B	C
A	1	1	1
B	0	1	1
C	0	1	1

Tabelle 3.3: Beispiel *Duell-Matrix mit* $\geq_p$

	A	B	C
A	0	1	1
B	0	0	0
C	0	0	0

Tabelle 3.4: Beispiel *Duell-Matrix mit* $>_p$

Ergebnislisten

In der Variablen *winners* werden in jeder Evaluation die Sieger der Wahl aufgelistet. Diese Liste ist nicht verschlüsselt und wird dann als Wahlergebnis zurückgegeben. Die Wahlergebnisse enthalten im aktuellen System immer nur eine Liste von Siegern. Es werden keine Punkte oder Rangfolgen ausgegeben. Die Kandidaten werden hier immer als Indizes mit Werten von 0 bis n_{cand} repräsentiert. Bei mehreren Kandidaten entspricht die Reihenfolge in der Liste nicht dem Wahlergebnis. Alle Kandidaten in dieser Liste sind Sieger und die Reihenfolge darin ist willkürlich.

3.4 Basisalgorithmen

If-Then-Else

Diese Methode wird benötigt für eine cryptische Wertzuweisung, welche von einem booleschen Wert abhängt. Dabei können alle Parameter und das Ergebnis verschlüsselt sein. Niemand kann nachverfolgen, welcher von zwei verschlüsselten Werten zurückgegeben wird.

Der boolesche Wert ist einen verschlüsselte Null oder Eins. Dieser wird mathematisch mit dem Wert, der im Fall $condition = 1$ zurückgegeben wird, multipliziert. Die Invertierung $1-condition$ wird mit dem anderen Wert multipliziert. Durch die Addition dieser Ergebnisse erhält man nur den gewollten Wert.

Da alle mathematischen Operationen sicher sind im Bezug auf die Verschlüsselung, ist auch der Wert des Ergebnisses nicht nachvollziehbar. Diese Methode hat nur ihren Sinn, wenn die *condition* auch wirklich geheim ist. Wenn einer der beiden anderen Wert nicht geheim ist, sollte er auch unverschlüsselt übergeben werden, um Zeit zu sparen. Intern wird dann dann nicht die *Multiplicate*-Methode mit Kommunikation der Trustees verwendet, sondern nur eine einfache mathematische Berechnung auf dem Ciphertext. In Abschnitt 2.2 wurde bereits erwähnt, dass die Multiplikationen mit einem unverschlüsselten Wert ohne einen Schlüssel möglich sind und somit keine Kommunikation nötig ist. So wird die Laufzeit verringert. Wenn die Bedingung bekannt ist, kann ein normales If verwendet werden und muss nicht auf diese Funktion zugegriffen werden.

Algorithmus 3.2 If-Then-Else

input condition ← cipher text with value 0 or 1
valueTrue ← cipher whit value which is returned when dec(condition) == 1
valueFalse ← cipher whit value which is returned when dec(condition) == 0
output cipher text with valueTrue if condition has value 1, else cipher text with valueFalse
procedure IfThenElse(condition, valueTrue, valueFalse)
    return multiplicate(condition, valueTrue) + multiplicate(($enc(1)$ − condition), valueFalse)
end procedure

	Greater-Than	Equals	Multiplicate	Decrypt
Anzahl	0	0	0/1/2	0
Laufzeit	0	0	0-00:00:03 h	0

Entschlüsselung	—

Tabelle 3.5: Zeitkritische Operationen und Entschlüsselungen für die IfThenElse-Methode

Get-Maximum

Bei diesem Algorithmus wird die höchste Punktzahl aus einer Liste gesucht. Dabei wird zunächst davon ausgegangen, dass der Kandidat mit Index 0 der beste ist. Dessen Wert wird in *maximum* geschrieben. Dann wird über die Punktzahl aller anderen iteriert und geprüft, ob diese besser sind. Wenn ja, wird der Wert von *maximum* überschrieben. So sollte nach Abschluss der Schleife der größte gefunden sein, welcher anschließend zurückgegeben wird.

Die Schleife wird $n-1$ Mal ausgeführt und verwendet ein *Greater-Than* und zwei *Multiplicate* durch die IfThenElse-Methode. Die Bitzahl für den *Greater-Than*-Vergleich hängt von der Größe der Werte in *points* ab. Wir gehen davon aus, dass die maximale Punktzahl später von der Wählerzahl abhängt. n ist hier die Länge der Liste *points*. Da weder einer der Vergleiche noch eine Punktzahl entschlüsselt wird, wird in der GetMaximum-Methode nichts verraten. Das Ergebnis ist nur der verschlüsselte Wert des Maximums.

Algorithmus 3.3 Get-Maximum

```
input     points ← list of encrypted numbers, which are examined for its maximum
output    encrypted maximum
procedure GetMaximum(points)
    maximum ← points[0]
    for all cand in 1..points.length − 1 do
        gtEq ← greaterThan(points[cand], maximum)
        maximum ← IfThenElse(gtEq, points[cand], maximum)
    end for
    return maximum
end procedure
```

	Greater-Than	Equals	Multiplicate	Decrypt
Anzahl	$n-1$[Wähler]	0	$2\cdot(n-1)$	0
Laufzeit ($n=5$)	00:03:52 h	0	00:00:10 h	0
Laufzeit ($n=20$)	00:18:22 h	0	00:00:49 h	0

Entschlüsselung	—

Tabelle 3.6: Zeitkritische Operationen und Entschlüsselungen für die GetMaximum-Methode

Get-Minimum

Diese Algorithmus ist äquivalent zu GetMaximum, nur dass hier das Minimum statt dem Maximum gesucht wird. Deshalb wird hier auf eine weitere Beschreibung verzichtet.

Algorithmus 3.4 Get-Minimum

```
input     points ← list of encrypted numbers, which are examined for its minimum
output    encrypted minimum
procedure GetMinimum(points)
    minimum ← points[0]
    for all cand in 1..points.length − 1 do
        gtEq ← greaterThan(points[cand], minimum)
        minimum ← IfThenElse(gtEq, points[cand], minimum)
    end for
    return minimum
end procedure
```

	Greater-Than	Equals	Multiplicate	Decrypt
Anzahl	$n-1$[Wähler]	0	$2\cdot(n-1)$	0
Laufzeit ($n=5$)	00:03:52 h	0	00:00:10 h	0
Laufzeit ($n=20$)	00:18:22 h	0	00:00:49 h	0

Entschlüsselung	—

Tabelle 3.7: Zeitkritische Operationen und Entschlüsselungen für die GetMinimum-Methode

Match-Points

Die MatchPoints-Methode hat zwei Parameter: Eine Liste mit verschlüsselten Werten und ein Wert nach dem gesucht wird. Es wird über die Liste iteriert und geprüft, ob der Wert der Liste dem gesuchten Wert entspricht. In der Liste *matchCandIndicators* geben verschlüsselte Einsen und Nullen an, ob der entsprechende Kandidat die gesuchte Punktzahl besitzt oder nicht. Mit matchCount wird verschlüsselt mitgezählt, wie viele Kandidaten diesen Wert haben. Solange einer der beiden Parameter geheime Werte enthält ist auch das Ergebnis geheim.

Pro Schleifendurchlauf wird eine *Equals*-Methode benötigt. Die Bitzahl für den *Equals*-Vergleich hängt erneut von der Größe der Werte in *points* ab. Dabei wird von Gesamtpunkten ausgegangen, deren Maximum von der Wählerzahl abhängen, weshalb hier *Wähler* als Indikator für die Bitgröße angegeben ist. n ist auch hier die Länge der Liste und in der Verwendung immer die Anzahl der Kandidaten.

Algorithmus 3.5 Match-Points

input points ← list of encrypted numbers, which are rummaged for a value
pointsToSearch ← encrypted number which should be found
output count of matches and list of match indicators
procedure MatchPoints(points, pointsToSearch)
matchCandIndicators ← $EmptyList()$
matchCount ← $enc(0)$
for all cand **in** $0..$points$.length - 1$ **do**
matchIndicator ← equals(points, pointsToSearch)
matchCount ← matchCount + matchIndicator
matchCandIndicators.append(matchIndicator)
end for
return matchCount, matchCandIndicators
end procedure

	Greater-Than	Equals	Multiplicate	Decrypt
Anzahl	0	n[Wähler]	0	0
Laufzeit ($n = 5$)	0	00:01:09 h	0	0
Laufzeit ($n = 20$)	0	00:04:37 h	0	0

Entschlüsselung	—

Tabelle 3.8: Zeitkritische Operationen und Entschlüsselungen für die MatchPoints-Methode

4 Wahlunabhängige Softwareänderungen

Vor und während der Implementierung von Borda und Condorcet entstanden Änderungen im Code, welche alle Wahlverfahren betreffen. Zum Beispiel wurde der generische Wahlzettel eingeführt, welcher den Wahlinput der unterschiedlichen Verfahren vereinheitlicht. Da nun die Zahl von implementierten Wahlverfahren größer wurde, ist es wichtig auf eine gemeinsame Struktur zu achten. Durch Änderungen in der Softwarearchitektur sollten Codeduplikate verhindert werden. Außerdem sollten diese das Hinzfügen weiterer Wahlsysteme erleichtern und den Code verständlicher machen. Die Änderungen in der Architektur sind an die Architektur des Ordinos-Systems aus der Theorie angelehnt. Somit waren diese auch ein Schritt Richtung Realität und Umsetzung des finalen Systems. Die Architektur erleichterte ebenso das letzte Thema dieses Kapitels. Durch die Einheitlichkeit und gemeinsam genutzten Klassen wurde es einfacher ein kollektives Logging umzusetzen, welches auch die Laufzeiten untersucht.

Diese Änderungen sind Grundlagen für die Implementierungen von Borda und Condorcet. Deshalb sind die verfahrensunabhängigen Softwareänderungen vor den Kapiteln der konkreten Wahlverfahren beschrieben.

4.1 Generischer Stimmzettel

Um die Implementierung und Vergleichbarkeit von Wahlverfahren zu erleichtern, wurde der generische Stimmzettel eingeführt. Das Ziel war es, Stimmzettel generieren zu können, welche in unterschiedlichen Wahlen eingesetzt werden können. Nach Wunsch sollten dieselbe Liste von Stimmzetteln in einem Instant-Runoff-Votinig, einem relativen Mehrheitswahlsystem oder anderem als Input dienen können. Dann kann verglichen werden, in welchen Fällen diese dasselbe Ergebnis liefern und in welchen sie sich unterscheiden. Durch Differenzen könnten durch das Vergleichen sogar Fehler im Code zu Tage kommen. Neben der Vergleichbarkeit von Ergebnissen ermöglicht der generische Stimmzettel auch mehr Vergleichbarkeit im Code. Dadurch, dass alle Wahlsysteme denselben Input bekommen, werden diese auch zu einheitlicher innerer Struktur gezwungen. Die Schnittstellen sind nun überall die gleichen und somit sieht auch der Zugriff von außerhalb, die Funktion der Stimmabgabe, nun einheitlich aus. Code kann übertragen werden, sodass theoretisch die UnitTests für alle Wahlsysteme bis auf das Wahlergebnis gleich aussehen können. Damit muss weniger Aufwand betrieben werden, UnitTests zu schreiben. Hinzu kommt, dass der einheitliche Aufruf den Code verständlicher macht, da überall dieselbe Schreibweise verwendet wird. Zuvor dienten in Wahlsystemen zum Beispiel Matrizen als Input, die nur im Kontext dieser Wahl verstanden werden konnten. Ein weiterer Vorteil ist das Verschwinden von Code zur zufälligen Stimmgenerierung bei Testwahlen. Nun kann ein Zufallsalgorithmus viele Wahlsysteme versorgen, womit redundanter Code bei ähnlichen Wahlsystemen entfällt.

Dabei wurde der Wahlzettel abstrahiert mit möglichst wenig Informationen, sodass die Entscheidung des Wählers aber noch eindeutig ist. Durch die Reduktion dürfen keine Informationen verloren gehen. Redundanzen und wahlspezifische Informationen sind aber zu verhindern. In allen bisher implementierten Wahlen reicht bisher eine Rangfolge der Kandidaten mit einer Sonderregel aus. Man spricht von Rang-Wahlen (siehe [11]). Umgesetzt wird dieser Ansatz in der python-Klasse *PositionVote*. Wenn alle Kandidaten in einer Prioritätenliste angegeben sind, kann dies bis jetzt in jedem Wahlsystem verwertet werden. Das Instant-Runoff-Voting benötigt zum Beispiel die gesamte Rangfolge, ein relatives Mehrheitswahlsystem interessiert sich nur für den Kandidat, welcher an Position eins ist. Dieser bekommt dann im angepassten Wahlzettel im Gegensatz zu allen anderen Kandidaten ein Kreuz beziehungsweise eine Eins zugewiesen. Borda wird aus dieser Reihenfolge eine Liste der zugehörigen Punkte generieren. Condorcet-Wahlen leben von Prioritäten und sind deshalb für diesen Stimmzettel auch gut geeignet. Die angesprochene Sonderregel betrifft das Ignorieren von Kandidaten. Zum Beispiel für Condorcet ist es in manchen Wahlen möglich, Kandidaten zu ignorieren. Diese haben dann keine Priorität. Sowohl mit dem Ersten als auch mit dem Letzten befinden sich diese im Gleichstand.

Es ist aber nicht möglich, in jedem System mit jedem Stimmzettel umzugehen. Die Sonderregel ergibt nicht in jeder Wahl einen Sinn. Neben dem Ignorieren könne aber auch noch andere Probleme auftreten. So erlauben manche Wahlen Gleichstand zwischen Kandidaten auf einem Stimmzettel und andere nicht. Jedoch gibt es Systeme, in welchen deutlich weniger Informationen ausreichen. In einem relativen Mehrheitswahlsystem wird nur eine einzige Stimme abgegeben. Dann wird aus der Rangfolge nur der beste betrachtet. Grundsätzlich sind mit der *PositionVote* auch Gleichstand zwischen Kandidaten erlaubt. Borda-Systeme können aber zum Beispiel verlangen, dass eine Mindestzahl von Kandidaten in eine Reihenfolge gebracht werden können, um sinnvoll Punkte zu verteilen. Nur durch Zufall könnten Kandidaten mit gleicher Priorität geordnet werden, was aber der Verfälschung einer Wahl entspricht.

Deshalb werden beim Transformieren der generischen Wahlzettel die Eigenschaften des Stimmzettels geprüft. Falls der Stimmzettel nicht geeignet ist, wird ein Fehler geworfen und diese Stimmabgabe ignoriert. Unterschiedliche Eigenschaften können durch Methoden der *PositionVote* abgefragt werden:

- Anzahl der ignorierten Kandidaten
- Gibt es einen Gleichstand zwischen zwei oder mehr Kandidaten?
- Die niedrigste Platzierung, welche doppelt belegt ist

Neben der *PositionVote* gibt es eine noch mächtigere Klasse. Die *PointVote* speichert sich zusätzlich noch Punkte für jeden Kandidaten. Dies wäre zum Beispiel bei einer Bewertungswahl nötig (siehe [15]). Hier können für jeden Kandidaten Punkte oder Schulnoten vergeben werden. Dabei reicht allein die Reihenfolge zwischen den Kandidaten nicht aus. Derjenige mit dem besten Durchschnitt gewinnt die Wahl. Nötig ist die *PointVote* für die existierenden Wahlverfahren noch nicht, jedoch ist sie für manchen Zufall besser geeignet. Wenn der Zufall Kandidaten auf denselben Rang setzen soll, geschieht dies einfacher über zufällig generierte Punkte anstatt zufällig generierter Reihenfolgen.

Dann ist es sinnvoll die *PointVote* zu verwenden, da diese einfach daraus generiert werden kann und noch mächtiger als die PositionVote ist. Der Zufallsalgorithmus der *PositionVote* eignet sich für ein absolutes Ranking ohne Gleichstand.

Wenn nun neue Wahlverfahren dem System hinzugefügt werden, müssen diese einen Adapter implementieren, welche einen generischen Stimmzettel interpretiert und für das System anpasst. Es müssen sich aber keine Gedanken mehr gemacht werden, wie der Input und der Zufall bei automatisch generierten Wahlen aussehen soll.

4.2 Architektur

Neben der Implementierung von Borda und Condorcet war es auch die Aufgabe, sich mit der Software dem Zielsystem anzunähern. Der Python-Code enthält nun Klassen für die *Election Authority*, das *Bulletin Board* und die *Election Properties*. Mit diesen Klassen als Basis eines Wahlprozesses, gilt folgende grobe Architektur:

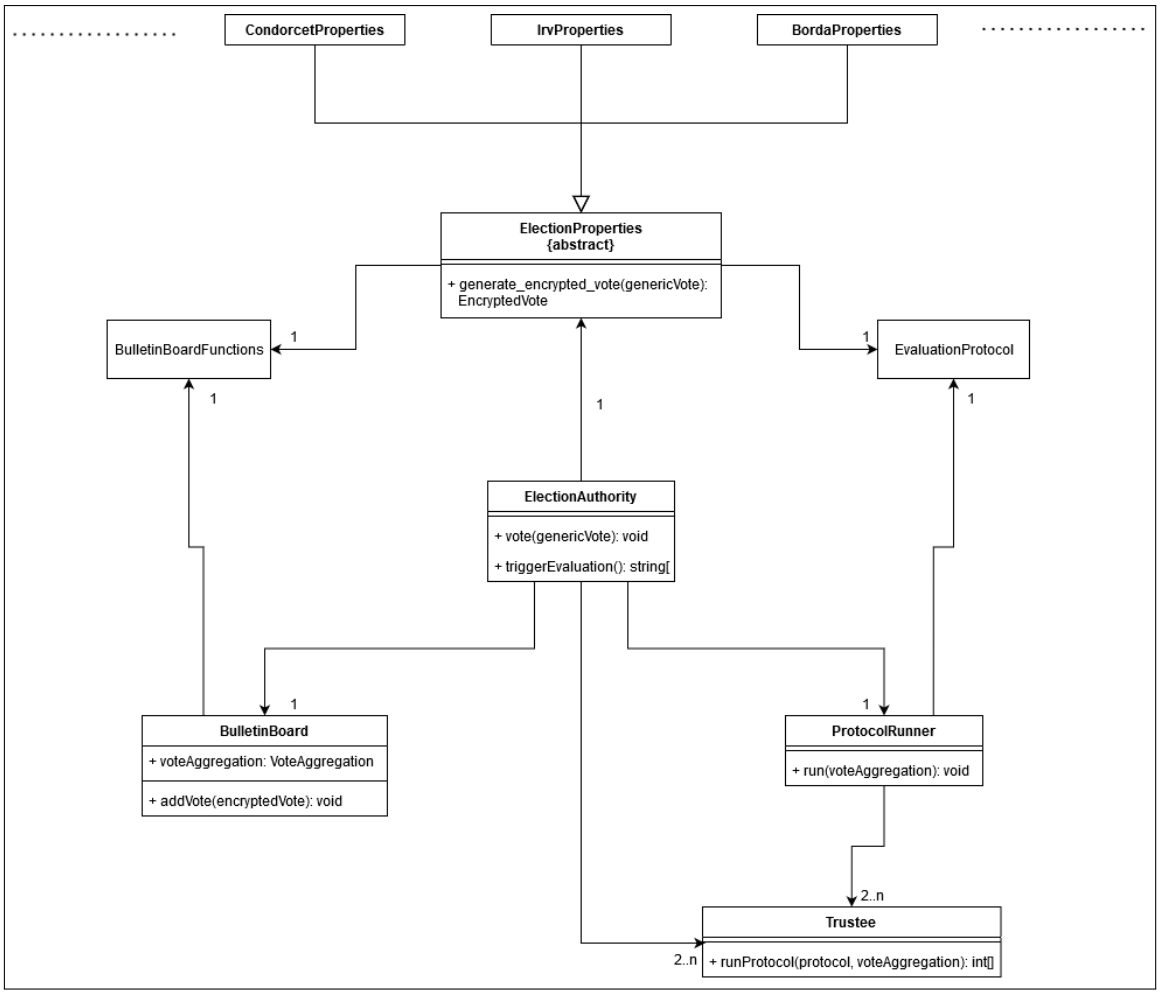

Abbildung 4.1: Grobe Softwarearchitektur

Die *Election Authority* ist das Kontrollelement des Systems. Diese ist für den Ablauf einer Wahl zuständig, indem sie die anderen Komponenten erstellt und mit ihnen zu gegebener Zeit kommuniziert. Um den Ablauf der Wahl festzulegen, müssen zur Initialisierung der *Election Authority* die *Election Properties* mitgegeben werden. Diese sind teilweise vom Wahlverfahren abhängig. In Abschnitt 5.1 und Abschnitt 6.1 werden die Konfigurationen für Borda und Condorcet beschrieben. Ein Beispiel für eine Konfigurationsmöglichkeit ist die Liste der Namen aller Kandidaten.

Neben statischen Werten enthalten die *Election Properties* aber auch Funktionen und Klassen, welche den weiteren Wahlverlauf definieren. Eine Funktion definiert, wie ein generischer Stimmzettel umgewandelt wird in einen verschlüsselten Stimmzettel, welcher für dieses System geeignet ist. Außerdem enthält sie Klassen, welche das Verhalten vom *Bulletin Board* und den evaluierenden *Trustees* beschreibt.

Die verschlüsselten Stimmzettel werden bei Stimmabgabe auf dem *Bulletin Board* gesammelt. In den meisten Fällen findet eine Vorberechnung auf dem *Bulletin Board* statt, um bei der Evaluierung Zeit zu sparen. Zum Beispiel könnte dies das Aufsummieren von Punkten sein. Hier sprechen wir von *Aggregierung* und das entsprechende Objekt ist die *aggregation*, wie sie auch im Code später auftaucht. Da die Stimmzettel und die Vorverarbeitung für jedes Wahlverfahren individuell sind, wird die *Aggregierung* mit den *Election Properties* festgelegt. Den Algorithmus hierfür enthalten die *BulletinBoardFunctions*, welche dem *Bulletin Board* beim Initialisieren von der *Election Authority* mitgegeben werden.

Auch der Algorithmus für die Evaluation ist in den *Election Properties* referenziert. Beim Initialisieren des *ProtocolRunners*, welcher die Evaluation durchführt, wird von der *Election Authority* das entsprechende Protokoll übergeben. Somit wird später die Evaluation allein mit der *aggregation* des *Bulletin Boards* als Input gestartet. Der *ProtocolRunner* erhält außerdem zu Beginn eine Referenz auf die von der *Election Authority* generierten *Trustees*. Der Initialisierungsvorgang sieht im Sequenzdiagramm so aus:

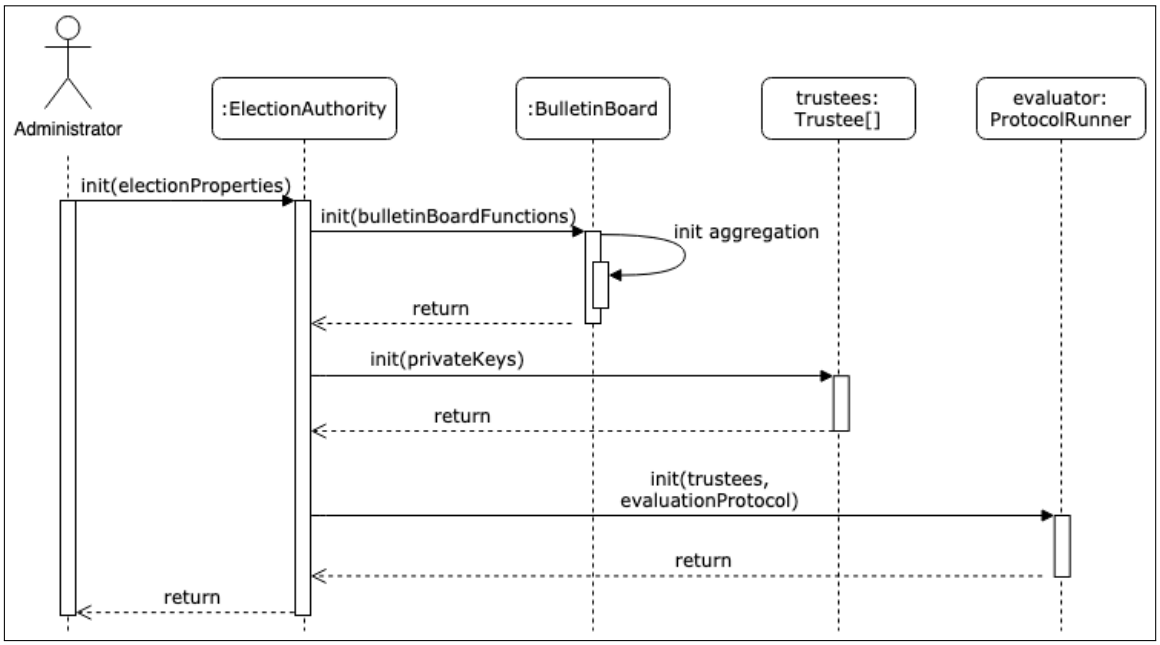

Abbildung 4.2: Sequenzdiagramm Systeminitialisierung

Die Wählerstimmen können von Methoden aus der Klasse der generischen Stimmzettel zufällig generiert werden. Den *Voter* wie im folgenden Schaubild gibt es noch nicht in der Implementierung. Eine Stimme wird dann über die *Election Authority* abgegeben. Im realen System wird die Entscheidung des Wählers auf dem *VSD* verschlüsselt und in das passende Format gebracht. Da dieses momentan noch nicht existiert, wird die Methode *vote* der *Election Authority* mit einer unverschlüsselten Stimme aufgerufen. Die *Election Authority* verwendet nun die Methode der *Election Properties*, um diese für das System anzupassen. Nach der Transformation ruft die *Election Authority* die *addVote*-Methode des *Bulletin Boards* auf. Dank der *BulletionBoardFunctions* kann dieses die Stimme verarbeiten. Somit verhält sich das System bei einer Stimmabgabe folgendermaßen:

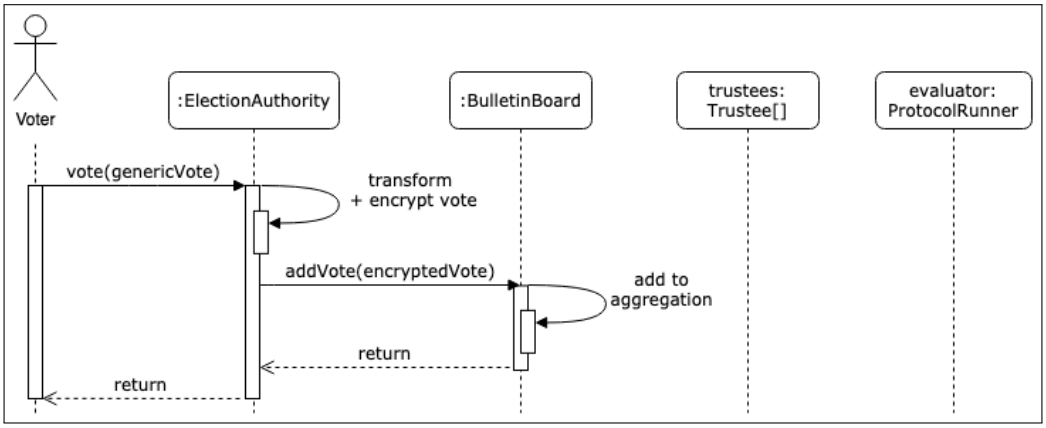

Abbildung 4.3: Sequenzdiagramm Stimmabgabe

Wenn nun die Wahl evaluiert werden soll, kann die Funktion *triggerEvaluation* der *Election Authority* aufgerufen werden. Die *Election Authority* holt sich nun die aktuelle *aggregation* vom *Bulletin Board*. Dann ruft sie den *ProtocolRunner* mit *run(aggregation)* auf. Dieser leitet den Input weiter an die *Trustees* mit dem Protokoll, welches ihm durch die Initialisierung bekannt ist. Auf allen *Trustees* läuft das Protokoll nun parallel ab. Somit ist der Aufruf der *Trustees* im folgenden Sequenzdiagramm (Abbildung 4.4) asynchron. Nachdem genug *Trustees* ein übereinstimmendes Ergebnis an den *ProtocolRunner* zurückgegeben haben, liefert dieser das Ergebnis an die *Election Authority*. Während der *Aggregierung* und der Evaluation arbeitet das System nur mit Indizes für die Kandidaten. Wenn zu Beginn Namen definiert wurden, ordnet die *Election Authority* die Indizes wieder den Namen zu und gibt dann die entsprechende Liste aus. In Zukunft sollte alles, was durch die Berechnung bekannt wurde, auf dem *Bulletin Board* veröffentlicht werden.

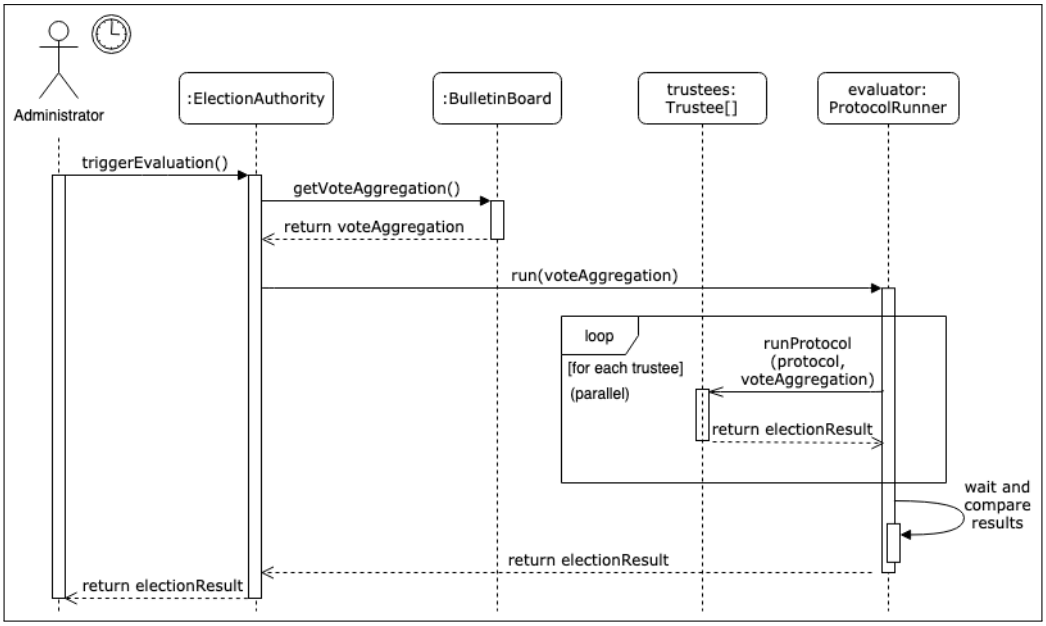

Abbildung 4.4: Sequenzdiagramm Evaluation

4.3 Laufzeiten und Logging

Entscheidend für die Laufzeit eines Ordinos-Wahlsystems ist die Anzahl kryptographischer Operationen. Ausschlaggebend sind Methoden, bei welchen die Trustees zusammen arbeiteten müssen, um ein Ergebnis zu erhalten. Für das Verfahren mit Paillier betrifft das die Operationen *Greater-Than*, *Equals*, *Multiplicate* und *Decrypt*. Hierbei stechen vor allem die ersten beiden heraus mit ihrer Laufzeit. Im sicheren System kann eine einzelne *Greater-Than*-Operation einige Sekunden dauern.

Da die Anzahl an Durchführungen der vier Operationen maßgeblich über die Gesamtlaufzeit entscheidet, werden diese vom System mitgezählt. Nach der Wahlevaluation werden sie mit dem Wahlergebnis ausgegeben. Im Test kann mit kleineren Schlüsseln alles deutlich schneller durchgeführt werden. Danach kennt man die Anzahl an kritischen Operationen. Wenn nun deren einzelne Laufzeit bei realistischen Bedingungen mit sicherem Schlüssel bekannt ist, kann die Dauer der gesamten Berechnung abgeschätzt werden. Die Laufzeitmessungen der Einzeloperationen sind in Abbildung 4.5 auszulesen. Die Zeiten wurden auf einem Pool-Rechner des SEC-Pools der Universität Stuttgart gemessen. Dabei wurden 3 Trustees auf einem Rechner simuliert und der *Threshold* betrug 2.

Wie in Abschnitt 3.1 bereits angesprochen, wird bei *Equals* und *Greater-Than* zwischen Bitzahlen unterschieden. Für beide Funktionen werden deshalb vier Datenreihen angezeigt. Es werden jeweils alle Zweierpotenzen von vier bis 32 als Bitzahlen der Operationen betrachtet. Die Kurve mit Beschriftung *EQ-32* entspräche den Laufzeiten für *Equals* mit Zahlen im Bereich von 0 bis 2^{32} = 4.294.967.296, welche in fast allen Wahlsystemen alle Werte verarbeiten könnte.

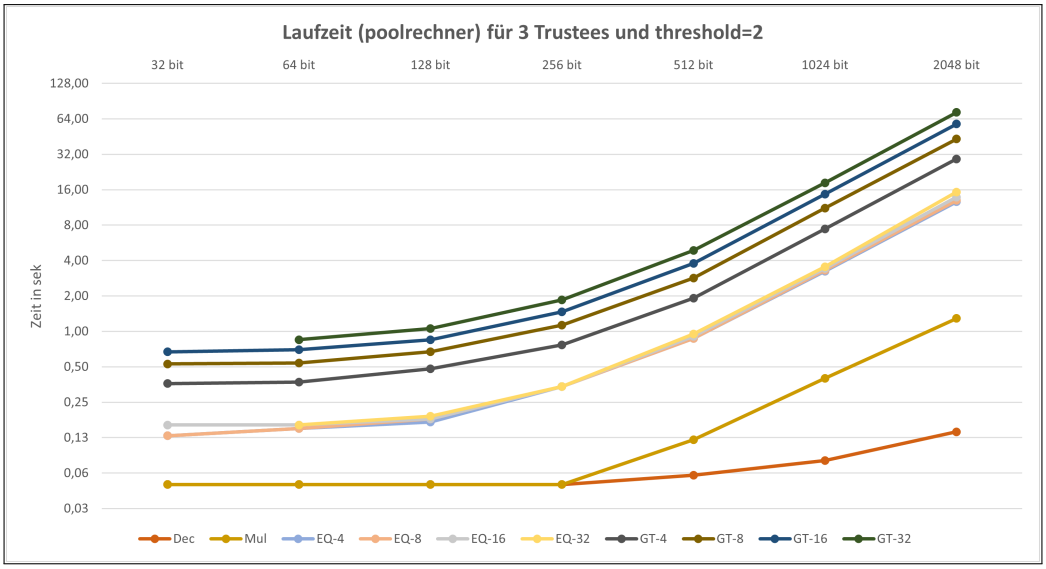

Abbildung 4.5: Laufzeit der Einzeloperationen auf SEC-Poolrechner (Diagramm)

Die x-Achse beschreibt unterschiedliche Schlüsselgrößen. Je größer der Schlüssel ist, desto sicherer ist das System. Für die Bitzahl der Schlüsselgrößen kommen nur Zweierpotenzen in Frage. Da diese im Diagramm den immer denselben Abstand zueinander haben, ist diese Skala logarithmisch. Erst ab 2048 Bit spricht man von einem sicheren Schlüssel. Kleinere Schlüssel bieten sich für Testumgebungen an. Das Untersuchen vieler Schlüssellängen brachte auch Vorteile für das Testen von Ordinos. So konnte eine Unregelmäßigkeit bei 128-Bit Schlüsseln entdeckt werden, welche aber mitlerweile aus dem System herausgearbeitet wurde. Im Diagramm ist zu erkennen, dass beim *Equals* die Bitzahl bei dieser Verschlüsselung nur einen geringen Unterschied macht. Bei *Greater-Than* ist der Unterschied recht groß. So benötigen 16-Bit-Operationen knapp doppelt so lange wie bei vier Bit. Auch die y-Achse mit der Laufzeit ist logarithmisch aufgetragen. Somit würde ein linearer Verlauf auch einem linearen Verhältnis zwischen Laufzeit und Bitgröße entsprechen. Die Operationen wurden zehn Mal auf dem Poolrechner durchgeführt und dann der Mittelwert darüber gebildet. Die genauen Zeiten können aus der Tabelle ausgelesen werden:

Bit_key	32 bit	64 bit	128 bit	256 bit	512 bit	1024 bit	2048 bit
Dec	0,05	0,05	0,05	0,05	0,06	0,08	0,14
Mul	0,05	0,05	0,05	0,05	0,12	0,40	1,29
EQ-4	0,13	0,15	0,17	0,34	0,87	3,25	12,66
EQ-8	0,13	0,15	0,18	0,34	0,87	3,31	13,00
EQ-16	0,16	0,16	0,18	0,34	0,91	3,42	13,86
EQ-32		0,16	0,19	0,34	0,95	3,55	15,31
GT-4	0,36	0,37	0,48	0,77	1,92	7,41	29,21
GT-8	0,53	0,54	0,67	1,13	2,84	11,14	43,21
GT-16	0,67	0,70	0,85	1,47	3,78	14,64	57,98
GT-32		0,85	1,06	1,85	4,87	18,25	72,69

Abbildung 4.6: Laufzeit der Einzeloperationen in Sekunden auf SEC-Poolrechner (Tabelle)

32-Bit-Vergleiche sind bei einem 32-Bit-Schlüssel nicht möglich. Die Untersuchungen mit mehr oder weniger *Trustees* und anderem *Threshold* ergab ein ähnliches Bild. In der Skalierung der y-Achse gab es Unterschiede, da mehr Rechenlast nötig ist bei größerer Zahl von *Trustees*. Solange alle Trustees auf einem einzigen Rechner laufen, ist die Zahl der laufenden Trustees ausschlaggebend. Wenn jeder Trustees auf einer eigenen Maschine läuft, sollte der *Threshold* eine größere Rolle spielen, da keiner den anderen ausbremst und der *Threshold* angibt, von wie vielen *Trustees* Rückmeldung erhalten werden muss. Jedoch wurde im Rahmen dieser Arbeit nur auf einer Maschine getestet. Dabei ergibt sich folgendes Bild für die Laufzeit unterschiedlicher Trustee-Zahlen und Threshold-Werten:

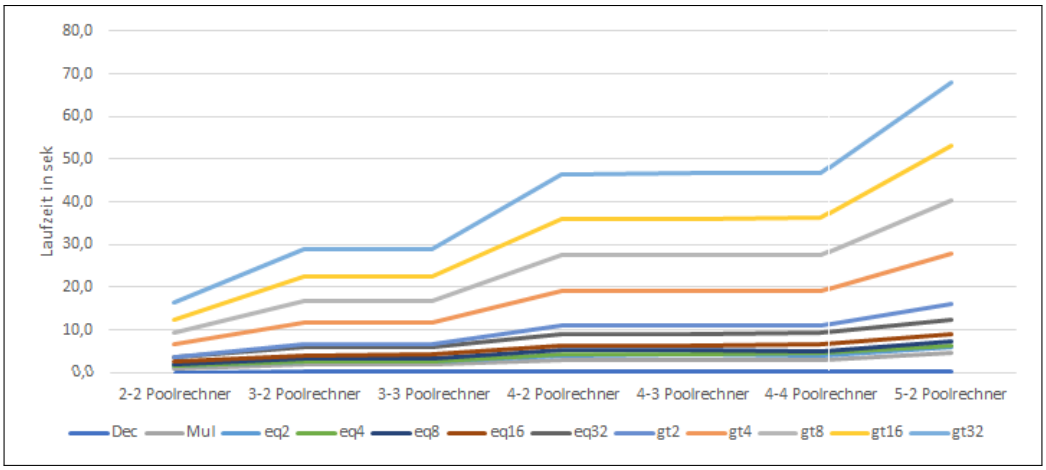

Abbildung 4.7: Laufzeit abhängig von *Trustees* und *Threshold* auf SEC-Poolrechner

Die y-Achse beschreibt wieder die Laufzeit, die x-Achse die Umgebung. Die Beschriftung “3-2 Poolrechner” bedeutet, dass die Operationen auf dem SEC-Poolrechner mit 3 *Trustees* und *Threshold* 2 durchgeführt wurden. Seit dieser Untersuchung hat sich etwas bei der Verschlüsselung verändert, somit passen die Zahlen nicht ganz zu den Daten aus der Tabelle. Die neuen Daten haben nur eine andere Skalierung der y-Achse zur Folge.

5 Borda

Borda ist ein sehr intuitives Wahlverfahren, weshalb es gerne im Alltag verwendet wird. Die Logik von Wahlverfahren wird nicht nur in echten Wahlen verwendet, sondern lässt sich auf viele Anwendungen übertragen, in denen ein Sieger gesucht wird. Bei Gesellschaftsspielen oder kleinen Sportwettkämpfen springen oft Platzierung als Ergebnis heraus. Wenn man nun mehrere Spiele oder Wettkämpfe durchführt, will man oft einen Gesamtsieger kühren. Der intuitive Ansatz ist oft das Addieren der Platzierungen und derjenige, welcher die geringste Summe besitzt, gewinnt die Gesamtwertung. Alternativ könnte man bei beispielsweise auch zehn Personen der ersten neun Punkte geben, der zweiten acht, der dritten sieben und so weiter, wobei dann der mit den meisten Gesamtpunkten gewinnt. Die zweite Möglichkeit wäre die originale Version von Borda. Die erste Möglichkeit ist vom Ergebnis äquivalent zur zweiten und somit auch eine Borda-Variante. Borda lässt sich auch noch weiter variieren. In manchen Fällen ist es sinnvoll, individuelle Punkte für jeden Platz festzulegen. Der Punkteabstand zwischen Platzierungen muss nicht immer eins betragen.

Borda ist nicht nur im Alltag zu finden, sondern auch beim Eurovision Song Contest oder bei offiziellen Sportwettkämpfen. Hierbei wird ein einzelner Wettkampf als Stimmzettel verstanden. So ist zum Beispiel der FIS-Weltcup eine Variante dieses Wahlverfahrens. FIS organisiert den Weltcup für Wintersportarten wie z.B. Ski-alpin, Langlauf, Biathlon oder Skispringen. Dabei erhalten die besten 30 Sportler Punkte abhängig von ihrer Platzierung in einzelnen Wettkämpfen. So bekommt zum Beispiel der Erste 100 und der Zweite 80 (siehe [20b]). Für den Gesamtweltcup werden alle Punkte der Saison addiert und der Athlet mit der größten Summe ist Gesamtweltcupsieger. Für die Formel-1-Weltmeisterschaft sieht es ähnlich aus. Auch der Medaillenspiegel bei den Olympischen Spielen kann als Borda verstanden werden (siehe [ES20]). Bei N Wettkämpfen erhält der Erste N^2 Punkte, der Zweite N und der Dritte einen Punkt. So kann eine einzelne Goldmedaille von beliebig vielen Silbermedaillen nicht aufgeholt werden. Die Nation mit den meisten Goldmedaillen gewinnt, bei Gleichstand zählen die Silbermedaillen und mit letzter Priorität Bronze.

Der Eurovision Song Contest (ESC) ist vermutlich das bekannteste Beispiel für Borda und ist im Gegensatz zu den anderen Anwendungen eine echte Wahl. Hierbei gibt jedes Land eine Wahl der Jury und eine Wahl durch Televoting ab. Beide dürfen an zehn Nationen Punkte verteilen. Der Erste erhält jeweils zwölf Punkte, der Zweite zehn, der Dritte acht. Dahinter gibt es immer einen Punkt weniger pro Platzierung. Zum Schluss werden all diese Punkte aufaddiert und die Nation mit den meisten erhaltenen Punkten gewinnt.

Die Sportwettkämpfe sind passende Beispiele für Borda-Verfahren, jedoch spielen sie keine Rolle für Ordinos. Ordinos wurde entwickelt für Tally-Hiding, also dem Geheimhalten von nicht relevanten Informationen. Das Ergebnis der Wahlevaluation wird auf das Geringste reduziert. Bei Sportwettkämpfen ist jedoch jeder Wettkampf öffentlich. Somit kann der Punktestand jedes Sportlers nachvollzogen werden und nichts ist geheim. Verschlüsselung kann keine Informationen mehr verstecken, weshalb einfache Algorithmen ohne jegliche Verschlüsselung ausreichen. Das macht die Verwendung von Ordinos überflüssig. Trotzdem werden die Beispiele im Folgenden weiterhin

betrachtet, da sie die Möglichkeiten und Varianten von Borda aufzeigen. Es könnte Wahlen geben, welche genau gleich wie die Auswertung im Sport umgesetzt sind, indem man sich das Ergebnis eines einzelnen Wettkampfes als Stimmzettel eines einzelnen Wählers vorstellt.

Abhängig von der Anwendung gelten für die Stimmzettel unterschiedliche Regeln, da zum Beispiel bei einem Biathlon-Weltcup andere Punkte vergeben werden als beim ESC. Die genannten Beispiele werden in folgenden Abschnitt abstrahiert und in ihren Gemeinsamkeiten zusammengefasst. Die unterschiedlichen Varianten, welche Borda ermöglicht, werden durch Konfigurationen beschrieben.

5.1 Varianten

Im Original wurde das Borda Verfahren von Jean-Charles de Borda definiert und es funktioniert folgendermaßen: “Alle Kandidaten werden von jedem Wähler in eine Rangreihenfolge gebracht. Die schlechteste Alternative erhält 0 Punkte, die nächstbessere 1 Punkt, die nächste 2 Punkte und so weiter. Gewählt ist, wer die höchste Anzahl Gesamtpunkte auf sich vereinigen konnte.” (siehe [06])

Die Gemeinsamkeit aller Borda Verfahren mit seinen Varianten ist das Punkteverteilen abhängig von Platzierungen und das Aufaddieren der Gesamtpunkte. Es werden nur die Systeme betrachtet, in welchen die höchste Gesamtpunktzahl gewinnt. Jedes Borda Verfahren, in dem wenig Punkte gut sind, lässt sich in ein äquivalentes umgekehrtes Verfahren übertragen. Durch eine Liste von Punkten wird vor der Wahl oder dem Wettkampf für jeden Rang festgelegt, welche Punktzahl hierfür vergeben wird. Für den Eurovision Song Contest wäre es die Liste 12, 10, 8, 7, 6, 5, 4, 3, 2, 1. Es wird davon ausgegangen, dass diese Liste immer monoton fallend ist. Theoretisch würde das System aber auch mit beliebigen Punktzahlen funktionieren.

Neben individuellen Punktzahlen gibt es noch einige andere Varianten. Abhängig von den geltenden Regeln für das Ranking, tauchen neue Komplikationen auf. Für dieses stellen sich folgende Fragen:

- Ist es erlaubt zwei Kandidaten auf dieselbe Priorität zu setzen?
- Wenn ja: Welche Punktzahl erhalten diese?
- Ist es erlaubt weniger Kandidaten in eine Reihenfolge zu bringen als die Punkteverteilung hergibt?
- Wenn ja: Erhält der Erste die höchste Punktzahl oder der Letzte die niedrigste?

Dieselbe Priorität kommt im Sport häufiger vor, da dies einem Gleichstand in einem Wettkampf entspricht. Auch die dritte Frage wird im Sport oft mit ja beantwortet. Meist handelt es sich um Ausnahmen, die jedoch immer wieder eintreten. Zum Beispiel beim Skispringen erhalten nur die 30 Teilnehmer aus dem Finalspringen FIS-Weltcuppunkte. Wenn jedoch einer der Athleten im Finaldurchgang disqualifiziert wird aufgrund eines zu weiten Anzuges, sammeln an diesem Tag nur 29 Sportler Punkte.

Für echte Wahlen kann entschieden werden, was erlaubt sein soll und wie Punkte verteilt werden. Im Eurovision Song Contest sind weder Gleichstand noch weniger als zehn Kandidaten im Ranking erlaubt. Somit erübrigen sich hier die beiden anderen Fragen.

Äquivalent zum Vorgehen der Platzierungsvergabe bei Gleichstand im Sport wird auch Frage zwei meist behandelt. Bei Gleichstand zweier Kandidaten auf dem ersten Rang sind beide erster und keiner zweiter. So wird auch in den meisten Fällen bei Borda für beide Kandidaten die bessere der beiden Punkte vergeben die schlechtere für niemanden.

Durch Tally-Hiding entstehen bei Borda zusätzliche Varianten. Tally-Hiding verhindert das Veröffentlichen von Informationen. Abhängig davon, wie viel man vom Ergebnis wissen will, kann man nun auch hier unterschiedliche Versionen durchführen. Zunächst stellt sich die Frage, welche Kandidaten das Ergebnis enthalten soll. Hierzu gibt es unter anderem diese Varianten:

- Alle Kandidaten zwischen Rang a und b
- Alle Kandidaten zwischen s und t Punkten
- Keine Kandidaten, sondern nur Metadaten
- Kombination aus letzten drei Varianten

Hierbei sind a und b beliebige Zahlen zwischen eins und der Anzahl an Kandidaten. Oft ist nur der Sieger relevant, dann gilt $a = b = 1$. In der zweiten Variante gibt es die Schranken s und t für Punkte. Durch Veröffentlichung nur der Kandidaten, welche eine Mindestpunktzahl erreicht haben, könnten Blamagen vermieden werden. Möglich ist auch ein Ergebnis ohne Kandidaten zu nennen. Es könnte eventuell interessant sein, wie deutlich gewonnen wurde ohne den Sieger zu nennen, um Spannung aufzubauen. Vor allem in Kombination mit den anderen wird diese Möglichkeit interessant. Man könnte zum Beispiel nur die besten 3 Kandidaten nennen und nur den Abstand zum vierten verkünden, ohne den Namen zu nennen.

Bei einer Liste von Kandidaten kann nun noch entschieden werden, was denn über diese bekannt werden soll. Folgende Möglichkeiten kommen in Frage:

- Nichts, also nur die Menge der Kandidaten mit ihren Namen
- Nur die Reihenfolge zwischen diesen
- Deren Gesamtpunkte
- Kombination aus letzten drei Varianten

Auch hier sind Kombinationen und damit sind sehr individuelle Wahlergebnisse möglich. Ein Wahlergebnis könnte folgende Informationen veröffentlichen:

- Namen der besten zehn
- Die Reihenfolge unter den ersten drei
- Die Punktzahl des Ersten

Durch die unterschiedlichen Kombinationen ist die Auswahl an möglichem Tally-Hiding groß. Deshalb sind bis jetzt nur wenige implementiert worden. Hier müsste man sich mit sinnvoll definierten Konfigurationen und deren einheitlichem Umgang im System beschäftigen. In der aktuellen Software werden nur die Kandidaten gelistet, ohne deren Punktzahl oder Reihenfolge zu nennen. Da die Metadaten sehr individuell sind, fehlen auch diese in der Implementierung.

Nun wird ein Konzept vorgestellt, welches die genannten Konfigurationsmöglichkeiten zusammenfasst. Dieses ist nicht implementiert, sondern nur ein Vorschlag. Das Modell dieses Konfigurationsobjektes würde wie folgt aussehen:

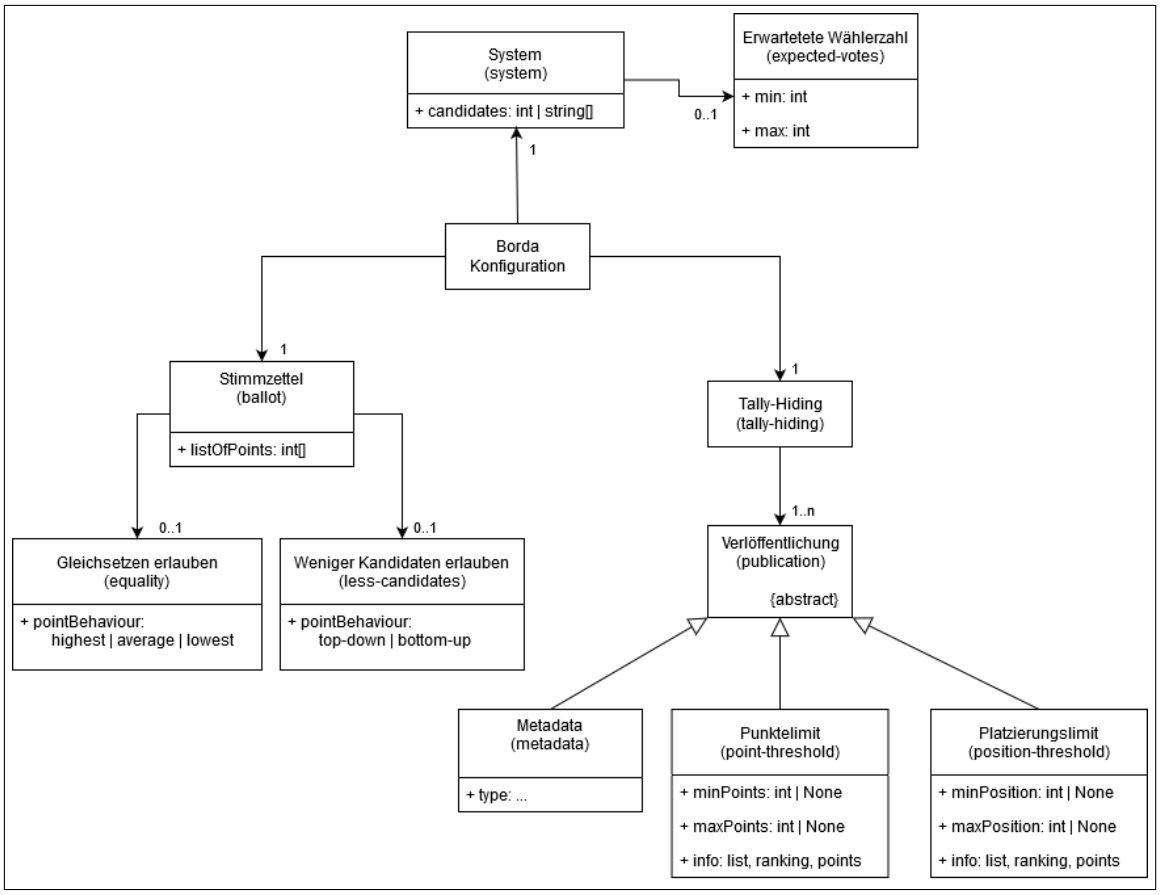

Abbildung 5.1: Modell einer Borda-Konfiguration

Mit der *listOfPoints* in *ballot* können individuelle Punkte für das Borda Verfahren definiert werden. *Equality* oder *less-candidates* sind optional und nicht vorhanden, wenn es jeweils verboten ist. Wenn das Gleichsetzen von Kandidaten möglich ist, muss mit *pointBehaviour* die Punktevergabe definiert werden. Sollen Kandidaten auf demselben Rang die beste, schlechteste oder durchschnittliche Punktzahl erhalten. Wenn weniger Kandidaten gelistet werden dürfen als Punkte bekommen würden, muss auch für *less-candidates* ein Verhalten festgelegt werden. Bei *top-down* erhält der Erste weiterhin dieselben Punkte und die kleinen Punkte werden nicht vergeben. Alternativ kann man auch mit *bottom-up* unten mit der Punktevergabe beginnen.

Beim Veröffentlichen kann man mit einer Liste mehrere unabhängige Informationen anfordern. Hier gibt es die Möglichkeiten über Platzierungen und über Punkte zu gehen. Dafür gibt man den gewünschten Bereich durch *minPoints* und *maxPoints* oder *minPosition* und *maxPosition* an. Desweiteren muss entschieden werden, was über betroffene Kandidaten angegeben werden soll. Dies kann eine reine Auflistung, eine Rangfolge oder sogar die genauen Gesamtpunkte der jeweiligen Kandidaten sein. Die Metadaten sind sehr individuell und vielfältig, jedoch sind sie nicht sehr häufig gefragt. Deshalb fehlt hier eine genauere Ausformulierung.

Die Einstellungsmöglichkeiten über *system* sind unabhängig von Borda. Sie sollten in jedem Wahlverfahren gesetzt werden können. Für jede Wahl wird vorausgesetzt, dass beim Initialisieren die Namen der Kandidaten oder zumindest deren Anzahl bekannt ist. Es ist möglich, im System eine Ober- und Untergrenze von Stimmen zu definieren. Dann wird nicht evaluiert, falls zu wenig Stimmen abgegeben wurden und Stimmen werden nicht akzeptiert, falls das obere Limit schon erreicht ist.

Um dieses Modell verständlicher zu machen, folgt ein Beispiel für den FIS-Weltcup in Form einer JSON. Es wird angenommen, dieser besteht aus 28 Wettkämpfen und die Liste der *candidates* entspricht der Menge von Athleten, welche im Laufe der Saison mindestens einen Weltcuppunkt erhielten. Wenn nach dem Weltcupfinale die Namen aller Sportler mit mindestens 200 Punkten, die Reihenfolge mit Namen der besten drei und die Punkte des Ersten bekannt gegeben werden sollen, könnte die Konfiguration folgendermaßen aussehen:

```
{
    system: {
        candidates : [Maria Musterfrau, Max Mustermann, ...],
        expected-votes: {
            min: 28,
            max: 28
        }
    },
    ballot : {
        listOfPoints : [100, 80, 60, 50, 45, 40, 36, 32, 29, 26, 24, 22, 20, 18, 16, 15, 14,
                        13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1],
        equality : {
            pointBehaviour: "highest"
        },
        less-candidates: {
            pointBehaviour: "top-down"
        }
    },
    tally-hiding: {
        publications : [
            {
                minPoints: 200,
                info: " list "
            }, {
                maxPosition: 3,
                info: "ranking"
            }, {
                maxPosition: 1,
                info: "points"
            }
        ]
    }
}
```

Diese Konfiguration wäre sehr mächtig, jedoch ist sie noch nicht implementiert. Beim Erstellen eines Borda-Systems können momentan folgende Parameter gesetzt werden: *list-of-points*, *allow-equality*

(wahr oder falsch), *allow-less-candidates* (wahr oder falsch), *begin-with-last* (entspricht *pointBehaviour* von *less-candidates*), *num-winners* (äquivalent zu *maxPosition* eines Platzierungslimits), *point-limit* (äquivalent zu *minPoints* eines Punktelimits) und *num-expected-votes* (entspricht *max* von *expected-votes*).

5.2 Implementierung

Um ein Wahlverfahren in Ordinos umzusetzen, müssen die *ElectionProperties* mit seinen Komponenten implementiert werden (siehe Abschnitt 4.2). Die Klasse *BordaElectionProperties* setzt die statischen Konfigurationsmöglichkeiten aus Abschnitt 5.1 als Attribute um. Außerdem muss die Funktion implementiert werden, welche einen generischen Stimmzettel in einen passenden verschlüsselten Stimmzettel für Borda transformiert. Dieser Algorithmus muss die Reihenfolge der Kandidaten aus dem generischen Stimmzettel auslesen und diese in Borda-Punkte umwandeln. Zum Abschluss werden die Daten verschlüsselt. Da dieser Algorithmus nicht mit verschlüsselten Daten rechnet, handelt sich hierbei nicht um ein Ordinos-spezifisches Problem. Deshalb wird er hier nicht genauer untersucht.

Interessanter wird es bei den Klassen, welche durch die *ElectionProperties* der *ElectionAuthority* darüber hinaus mitgegeben werden. Die *BulletinBoardFunctions*-Klasse beschreibt, wie die Stimmzettel aggregiert werden, um bei der Evaluation Zeit zu sparen. Durch die *EvaluationProtocol*-Klasse wird definiert, wie aus der *Aggregierung* der Sieger ermittelt wird. Es folgen nun die *Aggregierung* und unterschiedliche Methoden der Evaluation für Borda.

5.2.1 Aggregierung

Die *BulletinBoardFunctions* sind im Rahmen von Borda intuitiv. Die Punktzahlen werden für jeden Kandidaten zu einer Gesamtpunktzahl aufaddiert. Die *aggregation* hat dasselbe Format wie die *points*, die Stimmabgabe des Wählers. Mit dem Paillier-Encryption-Scheme, welches momentan in der Ordinos-Implementierung verwendet wird, können verschlüsselte Zahlen aufaddiert werden ohne einen privaten Schlüssel zu verwenden. Somit sieht der Pseudocode für das Hinzufügen einer *points* zur *aggregation* folgendermaßen aus.

Algorithmus 5.1 Borda Aggregierung

```
input     aggregation ← list of encrypted sum of points per candidate
          points_ballot ← list of new encrypted points per candidate
output    new aggregation (the sum of old aggregation and pointsBallot)
procedure BordaAggregate(aggregation, points_ballot)
    for all cand in 0..aggregation.length − 1 do
        aggregation[cand] ← aggregation[cand] + points_ballot[cand]
    end for
    return aggregation
end procedure
```

	Greater-Than	Equals	Multiplicate	Decrypt
Anzahl	0	0	0	0
Lfz. (5 Kandidaten)	0	0	0	0
Lfz. (20 Kandidaten)	0	0	0	0
Entschlüsselung	—			

Tabelle 5.1: Zeitkritische Operationen und Entschlüsselungen für Borda *Aggregierung*

Wie die Tabelle zu interpretieren ist, steht in Abschnitt 3.1 geschrieben. Es werden in diesem Algorithmus keine für die Performanz des Gesamtsystems relevanten Funktionen aufgerufen.

Es wird bei der *Aggregierung* auch noch nichts verraten. Beide Parameter sind grün markiert, enthalten also verschlüsselte Daten. Dann wird ein Schleifendurchlauf für jeden Kandidaten durchgeführt. Hier ist immer bekannt welcher Kandidat betrachtet wird. Es wird für jeden Kandidaten jedoch nur eine verschlüsselte Zahl auf eine andere verschlüsselte Zahl addiert.

5.2.2 Evaluation - Punktelimit

Für das Evaluieren müssen Operationen verwendet werden, welche *Trustees* mit privatem Schlüssel berechnen. Wie in Abschnitt 5.1 beschrieben wird zwischen dem Platzierungslimit und dem Punktelimit unterschieden. Wenn alle Kandidaten ausgegeben werden sollen, welche eine Mindestpunktzahl erreicht haben, wird die PointThresholdEvaluation verwendet. Diese ist die kürzeste und performanteste aller Borda-Evaluationen. Der *pointThreshold* darf dabei verschlüsselt sein, womit niemand das Punktelimit erfahren würde. Dies kann bei zufälligen Zahlen oder Werten, welche Informationen über Punktzahlen der Kandidaten enthalten, eine Rolle spielen. Eine öffentliche Mindestpunktzahl kann mit dem *PK* jederzeit verschlüsselt werden, um sie dem Algorithmus anzupassen.

Algorithmus 5.2 Punktelimit Evaluation

```
input     aggregation ← list of encrypted sum of points per candidate
          pointThreshold ← defines which candidates should be found as decrypted number; f.E. 100 if
          candidates with points >= 100 are requested
output    candidates (as indices) which reached pointThreshold
procedure PointThresholdEvaluation(aggregation, pointThreshold)
    winners ← EmptyList()
    for all cand in aggregation.length − 1 do
        gt ← greaterThan(aggregation[cand], pointThreshold)
        if decrypt(greaterThan) is 1 then
            winners.append(cand)
        end if
    end for
    return winners
end procedure
```

	Greater-Than	Equals	Multiplicate	Decrypt
Anzahl	n[Wähler]	0	0	n
Lfz. (5 Kandidaten)	00:04:50 h	0	0	00:00:01 h
Lfz. (20 Kandidaten)	00:19:20 h	0	0	00:00:03 h

Entschlüsselung	∀(Kandidat): Gesamtpunkte[Kandidat]≥Punktelimit

Tabelle 5.2: Zeitkritische Operationen und Entschlüsselungen für Punktelimit

Bei der Evaluation mit Punktelimit sind beide Parameter verschlüsselt. Das geforderte Punktelimit darf wie angesprochen auch geheim bleiben. In den meisten Fällen ist dies jedoch bekannt. In der öffentlichen Liste *winners* werden die Sieger gesammelt. Für jeden Kandidaten wird geprüft, ob seine Gesamtpunktzahl größer ist als *pointThreshold*. Nur *gt* wird entschlüsselt, welches die Werte null oder eins annehmen kann. Wenn der Kandidat genug Punkte erreicht hat, wird er zur *winner*-Liste hinzugefügt. Pro Kandidat wird ein *Greater-Than* und ein *Decrypt* ausgeführt.

5.2.3 Evaluation - Positionslimit

Die PositionThresholdEvaluation greift dann ein, wenn beispielsweise die besten drei Kandidaten gelistet werden sollen. Der *positionThreshold* legt die Anzahl der gesuchten Kandidaten fest. Bei Gleichstand ist eine Überschreitung des Limits möglich. Bei *positionThreshold* = 5 werden alle Kandidaten mit Position <= 5 gelistet. Im Ergebnis ist nicht mehr zu unterscheiden, wer welche Position einnimmt. Der *positionThreshold* darf sogar verschlüsselt sein, sodass nicht bekannt ist, nach wie vielen Kandidaten gefragt wurde. Diese Evaluation war bereits für andere Wahlverfahren vor dieser Arbeit implementiert. Aus Gründen der Vollständigkeit und der Vergleichbarkeit, wird diese hier erneut beschrieben und bewertet.

Beschreibung und Performanz

Zunächst vergleicht der Algorithmus jedes Duell zwischen zwei Kandidaten. Die verschlüsselten Punktzahlen werden dabei mit der *Greater-Than*-Methode geprüft, sodass auch das Ergebnis dieser Abfrage verschlüsselt bleibt. In Kombination mit der *Equals*-Methode wird die *duelMatrix* (siehe Abschnitt 3.3) gefüllt mit verschlüsselten Nullen und Einsen. Es wird $\frac{n\cdot(n-1)}{2}$-Mal verglichen, wobei n weiterhin die Anzahl von Kandidaten ist. In der FindWinnerCandidates-Methode werden aus der fertigen Matrix werden jeweils alle Zellen einer Zeile aufaddiert. Jede dieser Summen entspricht der Anzahl von Siegen des Zeilenkandidaten in direkten Duellen. Bei Borda bildet die *duelMatrix* eine Hierarchie. Deshalb ist aus der Summe der Duellsiege direkt die Platzierung abzuleiten. Die Daten sind jedoch weiterhin verschlüsselt. Für die Laufzeit und das Tally-Hiding wird es in dieser Methode erst beim Prüfen, ob ausreichend Duelle gewonnen wurde, interessant. Hier wird pro Kandidat ein *Decrypt* und ein *Greater-Than* aufgerufen. Die Werte hierfür liegen aber im Bereich der Anzahl von Kandidaten, weshalb wenig Bits für diese Operation benötigt werden.
Dieser Algorithmus liegt in der Aufwandsklasse $O(n^2)$.

Algorithmus 5.3 Positionslimit Evaluation

```
input     aggregation ← list of encrypted sum of points per candidate
          positionThreshold ← defines how many candidates should be found as decrypted number; f.E. 3 if
          candidates with position <= 3 are requested
output    candidates (as indices) which meet condition of positionThreshold
procedure PositionThresholdEvaluation(aggregation, positionThreshold)
    duelMatrix ← CreateDuelMatrix(aggregation)
    winners ← FindWinnerCandidates(duelMatrix, positionThreshold)
    return winners
end procedure
procedure CreateDuelMatrix(points)
    duelMatrix ← EmptySquareMatrix(points.length)
    for all cand_a in 0..points.length − 1 do
        for all cand_b in 0..points.length − 1 do
            if cand_a == cand_b then
                duelMatrix[cand_a][cand_b] ← enc(0)
            else if cand_a > cand_b then
                gtEq ← greaterThan(points[cand_a], points[cand_b])
                eq ← equals(points[cand_a], points[cand_b])
                duelMatrix[cand_a][cand_b] ← gtEq
                duelMatrix[cand_b][cand_a] ← enc(1) − gtEq + eq
            end if
        end for
    end for
    return duelMatrix
end procedure
procedure FindWinnerCandidates(duelMatrix, positionThreshold)
    neededWins ← enc(duelMatrix.length) − positionThreshold
    winners ← EmptyList()
    for all cand_a in 0..duelMatrix.length − 1 do
        winCount ← enc(0)
        for all cand_b in 0..duelMatrix.length − 1 do
            winCount ← winCount + duelMatrix[cand_a][cand_b]
        end for
        if decrypt(greaterThan(winCount, neededWins)) is 1 then
            winners.append(cand_a)
        end if
    end for
    return winners
end procedure
```

	Greater-Than	Equals	Multiplicate	Decrypt
Anzahl	$\frac{n\cdot(n-1)}{2}$[Wähler] n[Kandidaten]	$\frac{n\cdot(n-1)}{2}$[Wähler]	0	n
Lfz. (5 Kandidaten)	00:12:06 h	00:02:19 h	0	00:00:01 h
Lfz. (20 Kandidaten)	03:13:20 h	00:43:53 h	0	00:00:03 h

Entschlüsselung	∀(Kandidat): Position[Kandidat]≤Positionslimit

Tabelle 5.3: Zeitkritische Operationen und Entschlüsselungen für Borda Positionslimit

Tally-Hiding

Zu Beginn wird die verschlüsselte *duelMatrix* erstellt. Hierbei wird kein *Decrypt* verwendet und somit können nicht mehr Informationen verraten werden, als schon aus den Inputparametern auszulesen ist. Beim Aufruf vom Decrypt definiert die duelMatrix für jeden Kandidaten wie viele Kandidaten weniger Punkte haben als dieser. Über die Anzahl von Kandidaten und dem *positionThreshold* wird die Anzahl von Duellsiegen berechnet, welche ausreicht, um die geforderte Platzierung zu erreichen. Es wird über alle Kandidaten iteriert und dann für die Summe der Siege geprüft, ob *neededWins* erreicht wurde oder nicht. Nur das Ergebnis des *Greater-than*-Tests wird entschlüsselt. Es entspricht der Information, ob ein Kandidat eine bestimmte Position erreicht hat oder nicht.

5.2.4 Evaluation - Sieger

In vielen Fällen wird nur nach dem Sieger gefragt. Wenn der *positionThreshold* unverschlüsselt ist und *positionThreshold* = 1 gilt, kann eine Optimierung genutzt werden, welche nur die Aufwandsklasse $O(n)$ anstatt $O(n^2)$ besitzt. Diese ist nun nach folgender Vorlage implementiert:

Algorithmus 5.4 Siegerevaluation

```
input     aggregation ← list of encrypted sum of points per candidate
output    list of candidates with most points as indices
procedure SingleWinnerPointEvaluation(aggregation)
    maximum ← GetMaximum(aggregation)
    winCount, winningIndicator = MatchPoints(aggregation, maximum)
    winners ← EmptyList()
    for all cand in 0..aggregation.length − 1 do
        if decrypt(winningIndicator[cand]) is 1 then
            winners.append(cand)
        end if
    end for
    return winners
end procedure
```

	Greater-Than	Equals	Multiplicate	Decrypt
Anzahl	$n-1$[Wähler]	n[Wähler]	$2 \cdot (n-1)$	n
Lfz. (5 Kandidaten)	00:03:52 h	00:01:09 h	00:00:10 h	00:00:01 h
Lfz. (20 Kandidaten)	00:18:22 h	00:04:37 h	00:00:49 h	00:00:03 h

Entschlüsselung	∀(Kandidat): Gesamtpunkte[Kandidat]=Max(Gesamtpunkte)

Tabelle 5.4: Zeitkritische Operationen und Entschlüsselungen für Borda Optimierung

Beschreibung

Bei diesem Algorithmus wird zuerst die höchste Punktzahl gesucht. Dafür wird die Methode GetMaximum verwendet. Danach wird in der Liste geprüft, welche Kandidaten diese Punktzahl haben. Nur bei einem Gleichstand auf Position eins sollten mehrere Kandidaten gefunden werden.

Performanz

n ist die Anzahl der Kandidaten, also die Länge der Liste *points*. In der GetMaximum-Methode wird $n-1$ mal *Greater-Than* und $2 \cdot (n-1)$ mal *Multiplicate* aufgerufen. MatchPoints verwendet n mal *Equals* und danach sind n *Decrypt*-Aufrufe nötig. Alle Operationen liegen in $O(n)$ und somit ist auch der gesamte Algorithmus in Aufwandsklasse $O(n)$.

Tally-Hiding

Bei der Evaluation eines einzelnen Siegers wird beim Bestimmen der maximalen Punktzahl nichts verraten. Es wird in der Funktion GetMaximum kein *Decrypt* verwendet. Danach wird für jeden Kandidaten geprüft, ob seine Gesamtpunkte dem *maximum* entsprechen. Nur die Frage, ob die Punktzahl identisch ist mit der größten Punktzahl, wird öffentlich beantwortet. Wie groß die höchste Punktzahl ist bleibt geheim.

6 Condorcet

Condorcet alleine kann nicht bei jedem Wahlergebnis einen Sieger finden. Es ist vielmehr eine Vorlage und ein Kriterium für Wahlverfahren als ein eigenständiges Wahlsystem. Es existieren einige unterschiedliche Condorcet-Methoden, welche sich gegenseitig widersprechende Ergebnisse erzeugen können. Am häufigsten ist davon die Schulze-Methode in Verwendung. Sie wird unter anderem von der deutschen Piratenpartei und Organisationen der Freien Software, wie zum Beispiel Ubuntu und Debian (siehe [Adl17]), benutzt. Neben der Schulze-Methode finden zum Beispiel auch Minimax und Ranked-Pairs immer wieder Anwendung. Mehr Informationen sind in Abschnitt 6.1 zu finden.

Bei Condorcet werden auf einem Stimmzettel Präferenzen angegeben (siehe [05]). Meist geschieht dies durch Angabe einer Rangfolge. Für die Auswertung spielt aber nur der paarweise Vergleich aller Kandidaten eine Rolle. So wird in jedem möglichen Kandidatenduell betrachtet, welcher von beiden die höhere Priorität besitzt. Unentschieden sind grundsätzlich erlaubt. Bei der Evaluation wird für jedes Duell gezählt, wie oft welcher Kandidat besser war. Wenn Kandidat A öfter gegen Kandidat B gewonnen hat als B gegen A, so wird auch im Endergebnis A gegenüber B präferiert. Ein Kandidat, welcher gegenüber jedem anderen präferiert wird, ist der Condorcet-Sieger. Dieser muss aber nicht existieren. Im Endergebnis sind Unentschieden und sogar zyklische Präferenzen möglich. Hier spricht man vom Condorcet-Paradoxon. Das Condorcet-Kriterium verlangt, dass ein Condorcet-Sieger gewinnen muss, falls dieser existiert. Ein Wahlverfahren, welches das Condorcet-Kriterium erfüllt, nennt sich Condorcet-Methode oder Condorcet-Verfahren. Wie es sich ohne Condorcet-Sieger verhält und ob zwischen den Kandidaten eine Rangfolge im Ergebnis gibt, bleibt der konkreten Umsetzung offen.

6.1 Varianten

Auch bei Condorcet kann es unterschiedliche Regeln für Stimmzettel geben. Im Normalfall gehen wir davon aus, dass die Stimmabgabe durch Angabe einer Rangfolge geschieht. Dabei können wie bei Borda in manchen Wahlsystemen Kandidaten auf dieselbe Priorität gesetzt werden und in anderen nicht. Daneben kommt bei Condorcet noch eine weitere Möglichkeit hinzu. Da hier keine Punkte pro Platzierung vergeben werden, sondern jedes Kandidatenduell für sich betrachtet wird, sind ungewöhnlichere Stimmabgaben umsetzbar. Hierzu gehört das “Ignorieren” von Kandidaten, welches weder eine positive noch eine negative Bewertung ist. Falls ein Kandidat “ignoriert” wurde, so ist er gleichgesetzt mit jedem anderen Kandidaten. Sowohl der Kandidat mit der höchsten Priorität, als auch der mit der niedrigsten, befindet sich im Gleichstand mit dem “ignorierten”. Dies findet dann eine Anwendung, wenn dem Wähler offen gelassen werden soll, Kandidaten zu bewerten. Theoretisch kann ein Condorcet-System jeden Stimmzettel akzeptieren, der für jedes Kandidatenduell eine Präferenz angibt. Die Beziehung, welche die Kandidaten zueinander haben, muss auch keine Rangfolge ergeben, sondern könnte auch zyklisch sein. Somit wäre auch eine Auswertung des

Spiels Schere-Stein-Papier mit beliebig vielen Personen in mehreren Runden möglich. In unserem Fall betrachten wir jedoch nur Rangfolgen mit der Ausnahme des "Ignorierens", da dies meist im Kontext von echten Wahlen ausreichend ist. Wir kommen damit zu zwei unabhängigen Fragen zur Stimmzettelkonfiguration:

- Dürfen mehrere Kandidaten auf dieselbe Priorität gesetzt werden?
- Dürfen Kandidaten "ignoriert" werden?

Im Gegensatz zu Borda gibt es bei Condorcet unterschiedliche Möglichkeiten zu evaluieren. Da Condorcet nur ein Kriterium für eine Wahl ist, gibt es viele unterschiedliche Implementierungen. Da die *Stimmabgabe* und die *Aggregierung* bei allen Condorcet-Systemen identisch abläuft, werden alle Condorcet-Wahlsysteme als Konfigurationsoption des einen Condorcet-Systems gesehen. Hier gibt es zum Beispiel diese Implementierungen:

- Copeland
- Minimax (Margins, Winning-Votes)
- Schulze-Methode
- Smith-Systeme
- Schwaches Condorcet
- Ranked-Pairs

Wie diese bereits implementierten Evaluation funktionieren, wird in Abschnitt 6.2 geschildert. Bis auf Ranked-Pairs wurden alle Methoden implementiert. Ranked-Pairs ist eine Methode, welche auch ab und zu in der Realität eingesetzt wird. Doch für Ordinos eignet sich dieses nicht gut, da aus Gründen des Tally-Hidings der Algorithmus nur schwer umzusetzen ist.

Bei Condorcet wird davon ausgegangen, dass nur nach dem Sieger gesucht wird. Zwar bieten manche Methoden auch Rangfolgen an, was aber im Rahmen dieser Arbeit nicht weiter untersucht wird. Es wird oft nicht mit Punkten gerechnet, weshalb einige zusätzliche Information, welche bei Borda interessant sein könnten, hier nicht grundsätzlich möglich sind. Das reine Condorcet und viele konkrete Implementierungen sehen nur einen einzelnen Sieger vor. Um zu entscheiden, welche Informationen zusätzlich veröffentlicht werden sollen, müsste auf jede Condorcet-Methode konkret eingegangen werden. Die Konfigurationsmöglichkeiten für gewollte weitere Informationen wird nicht weiter betrachtet. Jedoch kann es aus Gründen der Performanz manchmal vorteilhaft sein, Informationen zu entschlüsseln. Wenn aus allen Stimmzettel auch Borda-Punkte generiert werden, können zum Beispiel einige Kandidaten über die Borda-Gesamtpunkte als Condorcet-Sieger ausgeschlossen werden. Nur wer die mindestens die durchschnittlich Borda-Gesamtpunktzahl erreicht, kann Condorcet-Sieger werden (siehe [06]). Durch öffentliche Evaluation dieser Kandidaten müssen bei der Condorcet-Evaluation weniger Kandidaten betrachtet werden. Dies spart vor allem bei einer großen Zahl von Kandidaten Zeit, jedoch wird mehr über das Ergebnis verraten als gefragt. Eine andere Optimierungsmöglichkeit ist zu evaluieren, ob ein Condorcet-Sieger existiert. Bei diesem ist dann ersichtlich, ob er Condorcet-Sieger oder nur Sieger der jeweiligen Condorcet-Methode ist, jedoch kann dieser deutlich schneller gefunden werden als beispielsweise der Sieger von Minmax. Für die Copeland- und die Smith-Methode existieren weitere Laufzeitoptimierungen, welche Informationen entschlüsseln. Ob diese Varianten gewählt werden sollen, geben die Optionen

leakMinCopeland beziehungsweise *leakMaxCopeland* an. Sie sind nur in den entsprechenden Methoden verfügbar. Genauer wird dies in Abschnitt 6.2 untersucht. Wir gehen deshalb von dieser Konfiguration aus:

- Borda-Veröffentlichung (berechnet, welche Kandidaten die durchschnittliche Borda-Punktzahl erreicht haben)
- Condorcet-Sieger-Berechnung (prüft zunächst, ob ein Condorcet-Sieger existiert)
- *leakMinCopeland* (für Smith: minimale Copeland-Punktzahl eines Kandidaten im Smith-Set)
- *leakMaxCopeland* (für Copeland: Copeland-Punkte des Siegers)

Äquivalent zu Borda gibt es auch hierfür eine Idee für ein Modell der Konfiguration:

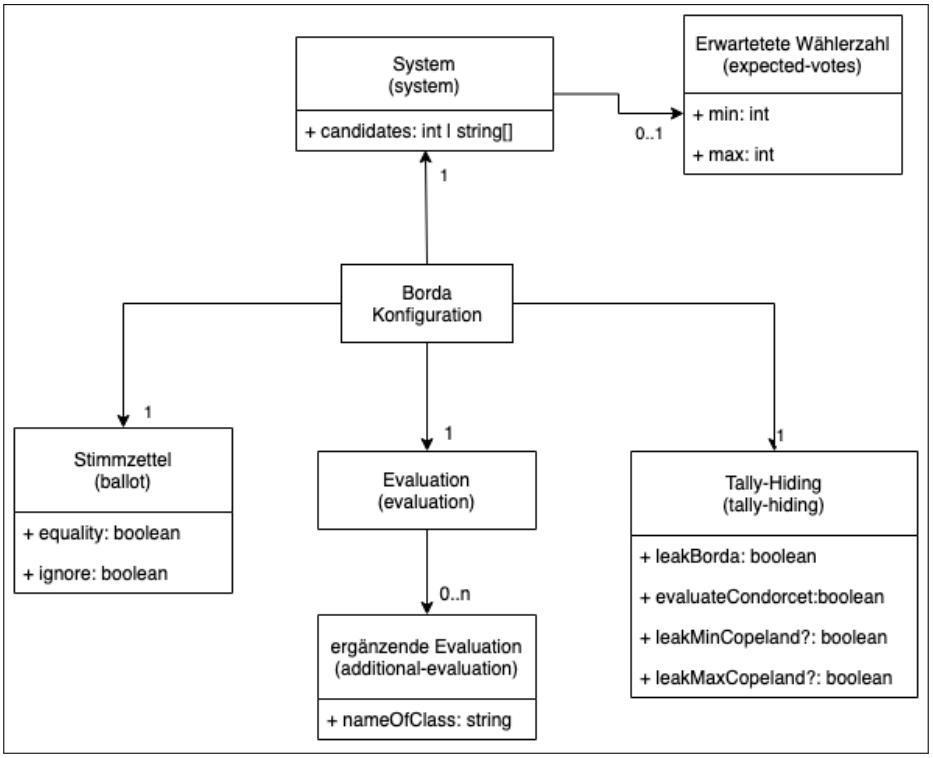

Abbildung 6.1: Modell einer Condorcet-Konfiguration

Da der Bereich *system* verfahrensunabhängig ist, gibt es hier keine Unterschiede zu Borda. Für *ballot* ist bei *equality* keine zusätzliche Konfiguration nötig, weshalb hier ein boolean statt einem Objekt ausreicht. Hinzu kommt das Attribut *ignore* um zu entscheiden, ob Kandidaten "ignoriert" werden dürfen. Beim Tally-Hiding wird hier nur darüber entschieden, ob eine performantere Variante mit Veröffentlichung von Informationen verwendet werden soll. Die *evaluation* taucht bei Borda noch nicht auf, da es nur eine Möglichkeit zur Evaluierung gibt. Die unterschiedlichen Borda-Evaluationen unterscheiden sich nur im Tally-Hiding. Hier kann jedoch durch eine Liste angegeben werden, welche Varianten gewählt werden sollen. Es kann mit allen Varianten zu einem Gleichstand kommen und somit mehrere Sieger geben. Dann kann durch eine zweite zusätzliche Varianten

versucht werden zu unterscheiden. Relevant ist dies vor allem bei Smith, da dieses immer eine Menge von Kandidaten liefert, sofern es keinen Condorcet-Sieger gibt. Mehr zu diesem Thema ist in Abschnitt 6.2 zu finden.

Da die Verwendungen in der Realität sehr speziell sind, reicht diese Konfiguration hier nicht aus, um eine existierende Wahl abzubilden. Deshalb ist hier kein Beispiel durch eine JSON angegeben.

6.2 Implementierung

Auch bei Condorcet mussten alle Komponenten der Election Properties implementiert werden. Die Wahlzettel haben bei Ordinos die Form einer *Duell-Matrix*. Für jedes Kandidatenpaar ist definiert, welcher der beiden höher priorisiert ist. Dabei müsste der Stimmzettel theoretisch nicht einmal einer Hierarchie zwischen den Kandidaten entsprechen. Im generischen Stimmzettel sind jedoch nur eindeutige Reihenfolgen umgesetzt. Mit der Implementierung kann daraus direkt eine *Duell-Matrix* generiert werden. Somit muss der Stimmzettel nur noch verschlüsselt werden um ihn für das Condorcet-System zu verwenden.

Auch hier werden nun die Algorithmen, welche auf verschlüsselten Daten arbeiten, genauer untersucht. Dazu gehören die *Aggregierung* aus der *BulletinBoardFunctions*-Klasse und die unterschiedlichen Evaluatoren.

Der Bezug zwischen den unterschiedlichen Varianten ist ein anderer als bei Borda. Bei Borda definiert die *Aggregierung* schon eindeutig das Wahlergebnis. Der mit den meisten Punkten gewinnt, der mit den wenigsten verliert. Bei Borda unterscheiden sich die unterschiedlichen Varianten nur im Tally-Hiding, grundsätzlich spiegeln sie aber dasselbe Ergebnis wider. Bei Condorcet ist die *Aggregierung*, die Grundlage aller Wahlevaluationen, nicht eindeutig zu interpretieren. Unterschiedliche Methoden können widersprüchliche Ergebnisse erzeugen. So kann bei Varianten eins Kandidat A gewinnen und bei Varianten zwei Kandidaten B und C. Nur unter bestimmten Bedingungen muss bei allen Condorcet-Methoden ein bestimmter Kandidat gewinnen, welcher Condorcet-Sieger ist.

Bei den Condorcet-Methoden geht es oft nur um den Sieger der Wahl und nicht um die gesamte Rangfolge oder Punkteabstände. Plain Condorcet unterstützt zum Beispiel gar keine Rangfolge. Alle Condorcet-Methoden haben eine Gemeinsamkeit: Wenn ein Condorcet-Sieger existiert, muss dieser der alleinige Sieger der Wahl sein. Deshalb ist es möglich, bei jeder Variante zunächst Plain Condorcet auszuführen, welches genau diesen evaluiert. Auch aus Sicht der Performanz hat es sich als sinnvoll herausgestellt, zuerst nach einem möglichen Condorcet-Sieger zu suchen. Deshalb wird in jedem Condorcet-System erst Plain Condorcet durchgeführt. Dabei kann aber mit *evaluateCondorcet* entschieden werden, ob der Condorcet-Sieger evaluiert werden soll oder nicht.

Dem System können mehrere zusätzliche Evaluatoren mitgegeben werden. Es ist möglich, das System nach Smith und dann nach Minimax evaluieren zu lassen. Plain Condorcet zählt dabei als erster Evaluator, wenn evaluateCondorcet aktiviert ist. Dabei gibt es nach jeder Evaluation die vier Fälle:

1. Kein Sieger
2. Alle Kandidaten sind Sieger
3. Genau ein Sieger
4. Eine Teilmenge der Kandidaten ist Sieger

Die Fälle 1. und 2. werden gleich behandelt. In beiden findet die folgende Evaluation mit allen Kandidaten statt. Im dritten Fall werden alle folgenden Evaluatoren ignoriert und dieser eine Sieger ausgegeben. Dies ist zum Beispiel immer der Fall, wenn ein Condorcet-Sieger existiert und evaluateCondorcet aktiviert ist. Im letzten Fall wird nur noch diese Liste von Kandidaten betrachtet. Relevant ist dies häufig bei der Verwendung von Smith. Es ist möglich die Smith-Evaluation zu nutzen, um die Kandidaten auf eine Teilmenge zu begrenzen. Daraus kann ein Performanzgewinn resultieren.

6.2.1 Aggregierung

Für die *aggregation* muss eine Datenstruktur gefunden werden, welche alle nötigen Informationen möglichst passend für die Evaluation enthält. Diese Datenstruktur sollte für Condorcet für jedes Kandidatenduell darstellen, wie oft welcher Kandidat gewonnen hat. Hierzu bietet sich die *Duell-Matrix* an, welche anstatt Nullen und Einsen die Gesamtzahl der Siege im Duell enthält. Die *Aggregierung* entspricht dann einer einer elementweise Addition aller Wahlzettel. Nach dieser Idee wurde der folgende Algorithmus umgesetzt.

Auch hier werden die verschlüsselten Werte nur addiert, weshalb der Rechenaufwand zu vernachlässigen ist. Da kein *Decrypt* ausgeführt wird, wird auch nichts verraten.

Algorithmus 6.1 Condorcet Aggregierung

input aggregation ← current sum of duel matrices
duelMatrix$^{weak}_{ballot}$ ← look at Abschnitt 3.3
output new aggregation (the sum of aggregation and duelMatrix$^{weak}_{ballot}$)
procedure CondorcetAggregation(aggregation, duelMatrix$^{weak}_{ballot}$)
  for all cand$_a$ **in** 0..aggregation.$length - 1$ **do**
    for all cand$_b$ **in** 0..aggregation.$length - 1$ **do**
      aggregation[cand$_a$][cand$_b$] ← duelMatrix$^{weak}_{ballot}$[cand$_a$][cand$_b$]
    end for
  end for
  return aggregation
end procedure

	Greater-Than	Equals	Multiplicate	Decrypt
Anzahl	0	0	0	0
Lfz. (5 Kandidaten)	0	0	0	0
Lfz. (20 Kandidaten)	0	0	0	0
Entschlüsselung	—			

Tabelle 6.1: Zeitkritische Operationen und Entschlüsselungen für Condorcet-Aggregierung

6.2.2 Evaluation - Plain Condorcet

Wie in Abschnitt 6.2 angesprochen, ist dieser Algorithmus Einstiegspunkt aller Condorcet-Methoden. In den Algorithmen ist hier nicht festgehalten, dass dem System beliebig viele Condorcet-Evaluatoren mitgegeben werden können. Kommt ein Verfahren zu einem eindeutigen Ergebnis, das heißt ein einzelner Sieger wurde gefunden, werden keine weiteren Evaluatoren ausgeführt. Plain Condorcet kann als erster Evaluator oder als Vorberechnung verwendet werden. Wenn gewollt, sucht dieser Algorithmus direkt nach dem Condorcet-Sieger. Existiert dieser, wird er dann sofort als Sieger der Wahl zurückgegeben. Existiert er nicht oder sollte nicht nach ihm gesucht werden, schafft Plain Condorcet die Grundlage aller weiterführenden Algorithmen. Beim Condorcet-Sieger sprechen wir im weiteren Verlauf auch vom echten Condorcet-Sieger.

Beschreibung

Ist die Option *evaluateCondorcet* auf *false* gesetzt oder wurde kein Condorcet-Sieger gefunden, werden die Objekte *resultDuelMatrix*, $gtSums_{Strict}$ und $gtSums_{Weak}$ zurückgegeben. Die Variable *resultDuelMatrix* enthält eine *Duell-Matrix*, welche im Gegensatz zur *Aggregierung* nur Nullen und Einsen besitzt. Die Zellen definieren für jedes Duell, welcher Kandidat im Gesamten häufiger über dem anderen priorisiert wurde oder ob es einen Gleichstand gibt. Die $gtSums_{Weak}$ zählt für jeden Kandidaten mit, gegen wie viele Kandidaten er das direkte Duell gewonnen hat plus die Anzahl der Unentschieden. Bei $gtSums_{Strict}$ handelt es sich um dieselbe Liste, nur dass hier die Unentschieden nicht mitdazuzählen. Letztere beiden sind nicht notwendig, da sie nur Informationen der *resultDuelMatrix* zusammenfassen. Jedoch ersparen diese Datenstruktur Redundanzen im Code der noch folgenden Evaluatoren.

Neben der Option *evaluateCondorcet* ist das Objekt der *Aggregierung* der einzige Input. Zu Beginn des Algorithmus werden diese Variablen mit Nullen oder leer initialisiert. Nun wird in zwei verschachtelten Schleifen über jeden Kandidaten iteriert. Da jedes Duell nur ein Mal betrachtet werden muss, durchläuft die innere Schleife nur die Kandidaten mit höherem Index als der des Kandidaten cand_a. Für das Duell cand_a und cand_b werden die Werte aus der *Aggregierung* ausgelesen und verglichen. Die Frage ist: Welcher der beiden hat mehr direkte Duelle auf den Stimmzetteln für sich entschieden? Durch die Operationen *Greater-Than* und *Equals* werden drei Fälle unterschieden: cand_a gewinnt mehr, cand_b gewinnt mehr oder ein Unentschieden ist eingetreten. Gemäß der Definitionen der drei Variablen wird das Ergebnis des Vergleiches gespeichert.

Wenn evaluateCondorcet aktiviert ist, wird bei jedem Durchlauf der äußeren Schleife geprüft, ob der aktuelle Kandidat cand_a Condorcet-Sieger ist. Er ist genau dann Sieger, wenn die Summe seiner Siege der Anzahl der Kandidaten minus eins entspricht.

Algorithmus 6.2 Plain Condorcet

input aggregation ← current sum of duel matrices
evaluateCondorcet ← false if this algorithm is used only as precalculation, else true
output the Condorcet winner or all calculated encrypted results
procedure PLAINCONDORCET(aggregation, evaluateCondorcet)
$\text{duelMatrix}^{\text{strict}}_{\text{result}} \leftarrow EmptySquareMatrix(\text{aggregation}.length)$
$\text{gtEqSums} \leftarrow EmptyList(\text{aggregation}.length)$
$\text{gtSums}_{\text{strict}} \leftarrow EmptyList(\text{aggregation}.length)$

for all cand_a **in** $0..\text{aggregation}.length - 1$ **do**
for all cand_b **in** $\text{cand}_a + 1..\text{aggregation}.length - 1$ **do**
$\text{gtEq} \leftarrow \text{greaterThan}(\text{aggregation}[\text{cand}_a][\text{cand}_b], \text{aggregation}[\text{cand}_b][\text{cand}_a])$
$\text{eq} \leftarrow \text{equals}(\text{aggregation}[\text{cand}_a][\text{cand}_b], \text{aggregation}[\text{cand}_b][\text{cand}_a])$
$\text{gt}_{\text{strict}} \leftarrow \text{gtEq} - \text{eq}$

$\text{duelMatrix}^{\text{strict}}_{\text{result}}[\text{cand}_a][\text{cand}_b] \leftarrow \text{gtEq} - \text{eq}$
$\text{duelMatrix}^{\text{strict}}_{\text{result}}[\text{cand}_b][\text{cand}_a] \leftarrow enc(1) - \text{gtEq}$
$\text{gtEqSums}[\text{cand}_a] \leftarrow \text{gtEq}$
$\text{gtEqSums}[\text{cand}_b] \leftarrow enc(1) - \text{gt}_{\text{strict}}$
$\text{gtSums}_{\text{strict}}[\text{cand}_a] \leftarrow \text{gt}_{\text{strict}}$
$\text{gtSums}_{\text{strict}}[\text{cand}_b] \leftarrow enc(1) - \text{gtEq}$
end for
if evaluateCondorcet **then**
if $\text{decrypt}(\text{equals}(\text{gtSums}_{\text{strict}}[\text{cand}_a], enc(\text{aggregation}.length - 1)))$ **is** 1 **then**
return cand_a
end if
end if
end for
return $\text{duelMatrix}^{\text{strict}}_{\text{result}}, \text{gtEqSums}, \text{gtSums}_{\text{strict}}$
end procedure

	Greater-Than	Equals	Multiplicate	Decrypt
Anzahl				
Bei evaluateCondorcet=true	$\frac{2 \cdot k \cdot n - k^2 - k}{2}$[Wähler]	$\frac{2 \cdot k \cdot n - k^2 - k}{2}$[Wähler] k[Kandidaten]	0	k
Bei evaluateCondorcet=false	$\frac{n^2 - n}{2}$[Wähler]	$\frac{n^2 - n}{2}$[Wähler]	0	0
evaluateCondorcet=true				
Lfz. (5 Kandidaten, $k = 2$)	00:06:46 h	00:02:02 h	0	00:00:00 h
Lfz. (5 Kandidaten, $k = 5$)	00:09:40 h	00:03:22 h	0	00:00:01 h
Lfz. (20 Kandidaten, $k = 2$)	00:35:45 h	00:08:58 h	0	00:00:00 h
Lfz. (20 Kandidaten, $k = 20$)	03:03:36 h	00:48:07 h	0	00:00:03 h
evaluateCondorcet=false				
Lfz. (5 Kandidaten)	00:09:40 h	00:02:19 h	0	0
Lfz. (20 Kandidaten)	03:03:36 h	00:43:53 h	0	0
Entschlüsselung	evaluateCondorcet ⇒ Condorcet-Sieger; leere Liste wenn er nicht existiert !evaluateCondorcet ⇒ —			

Tabelle 6.2: Zeitkritische Operationen und Entschlüsselungen für Plain Condorcet

Performanz

Die Laufzeit dieses Algorithmus hängt von der Anzahl von Iterationen ab. Diese betrachten wir als k. Wenn *evaluateCondorcet* nicht genutzt wird oder kein Condorcet-Sieger existiert, gilt $k = n$. Ansonsten entspricht k dem Index des Condorcet-Siegers. Dabei wird nicht die Zählweise aus der Informatik beginnend mit der null, sondern die beginnend mit der eins genommen. Ist der Condorcet-Sieger der erste in der Liste der Kandidaten, nimmt k den Wert eins an. Durch die innere Schleife finden immer $n - i$ Vergleiche in der i-ten Iteration statt. Ein Vergleich besteht aus einem *Equals* und einem *Greater-Than*. Diese arbeiten auf Werten von 0 bis $\mathsf{n}_{\mathsf{votes}}$. Mit der Gaußschen Summenformel (GS) kommt man auf folgende Zahl von Vergleichen nach k Iterationen:

$$\sum_{i=1}^{k} n - i = \sum_{i=1}^{k} n \; - \sum_{i=1}^{k} i \underset{\text{GS}}{=} k \cdot n - \frac{k^2 + k}{2} = \frac{2 \cdot k \cdot n}{2} - \frac{k^2 + k}{2} = \frac{2 \cdot k \cdot n - k^2 - k}{2}$$

Im Fall $k = n$ gilt:

$$\frac{2 \cdot k \cdot n - k^2 - k}{2} \underset{\text{k=n}}{=} \frac{2 \cdot n \cdot n - n^2 - n}{2} = \frac{n^2 - n}{2}$$

Wenn *evaluateCondorcet* auf true gesetzt ist, folgt noch ein *Equals* und ein *Decrypt* in der äußeren Schleife pro Iteration. Dieses *Equals* prüft die Anzahl von Duellsiegen, welche nicht größer sein kann als die Anzahl der Kandidaten. Deshalb gehen wir hier von kleineren Bit-Werten als beim vorherigen Vergleich aus. Mit der Option *evaluateCondorcet* liegt ein kleiner Mehraufwand vor, wenn kein Sieger gefunden wird. Wenn er jedoch existiert, können mit dieser einige Vergleiche und alle weiteren Evaluationen erspart werden. Da es sich immer um Condorcet-Methoden handelt, müssten diese auch den Condorcet-Sieger als alleinigen Wahlsieger finden. In den Laufzeiten der folgenden Condorcet-Methoden wird die Vorberechnung immer dazugezählt, da ohne diese der Algorithmus nicht möglich wäre. Es wird davon ausgegangen, dass die Option *evaluateCondorcet* dabei deaktiviert war.

Tally-Hiding

Die Vergleiche laufen verschlüsselt ab und können nicht mitgelesen werden. Erst nachdem in der äußeren Schleife ein Kandidat vollständig abgehandelt wurde, wird, wenn der Condorcet-Sieger gesucht werden soll, eine Information über diesen Kandidaten entschlüsselt: Ist er der Condorcet-Sieger oder nicht? Dies ist genau dann der Fall, wenn er in jedem direkten Duell gewinnt und somit die $gtSums_{Strict}$ für ihn den Wert $n - 1$ enthält. Da es nur einen Condorcet-Sieger geben kann, kann nach dem ersten Zutreffen der Bedingung die gesamte Evaluation abgeschlossen werden mit diesem Kandidaten als Ergebnis. Wird dieser Algorithmus nur als Vorberechnung genutzt, wird nichts entschlüsselt.

Im Abschnitt 6.1 wurde die Optimierung mit *leakBorda* angesprochen. Sie ist zwar im System implementiert, wurde jedoch in der schriftlichen Ausarbeitung im Algorithmus ignoriert. *leakBorda* macht es kompliziert und unübersichtlich. Da fängt es schon bei der Punkteverteilung auf den Stimmzetteln an. Wie viele Borda-Punkte sollen die Kandidaten bekommen, wenn auf den Stimmzetteln Gleichstand erlaubt ist? Es muss parallel zur Condorcet-Aggregierung noch eine Borda-Aggregierung durchgeführt werden. Im Plain Condorcet können dann über die PointThresholdEvaluation

alle Kandidaten ausgeschlossen werden, welche weniger Borda-Punkte als die durchschnittliche Borda-Punktzahl besitzen. Plain Condorcet würde dann nur mit circa der Hälfte der Kandidaten ablaufen, wodurch einige Vergleiche erspart bleiben. Wenn kein Condorcet-Sieger existiert, müssen die ersparten Operationen jedoch für die weiteren Evaluationen nachgeholt werden. Die Regel mit den Borda-Punkten gilt nur für den Condorcet-Sieger und nicht für alle Condorcet-Methoden.

6.2.3 Evaluation - Weak Condorcet

Um Weak-Condorcet-Sieger zu sein, muss ein Kandidat nicht gegen jeden anderen eindeutig gewinnen (siehe [05]). Er darf nur gegen keinen anderen verlieren. Wenn es zwei Kandidaten gibt, welche gegen alle anderen im direkten Duell gewinnen, zwischen diesen aber ein Gleichstand herrscht, so sind diese beiden Weak-Condorcet-Sieger.

Algorithmus 6.3 Weak Condorcet

```
input     gtEqSums ← for each candidate the number of gained duels in the result aggregation
                     plus the number of draws
          possibleWinners ← list of regarded candidates, candidates represented as indices
output    the Weak Condorcet winners
procedure WeakCondorcet(gtEqSums, possibleWinners)
    winners ← EmptyList()
    for all cand_a in possibleWinners do
        if decrypt(equals(gtEqSums[cand_a], enc(gtEqSums.length − 1))) is 1 then
            winners.append(cand_a)
        end if
    end for
    return winners
end procedure
```

	Greater-Than	Equals	Multiplicate	Decrypt
Anzahl (Plain Condorcet)	$\frac{n^2-n}{2}$[Wähler]	$\frac{n^2-n}{2}$[Wähler]	0	0
Bei evaluateCondorcet=true	0	n[Wähler]	0	n
Anzahl (Weak Condorcet)	0	n[Kandidaten]	0	n
Lfz. (5 Kandidaten)[1]	00:09:40 h	00:03:22 h	0	00:00:01 h
Lfz. (20 Kandidaten)	03:03:36 h	00:48:07 h	0	00:00:03 h

Entschlüsselung	Liste der Weak-Condorcet-Sieger evaluateCondorcet ⇒ Unterscheidung zu Condorcet-Sieger

Tabelle 6.3: Zeitkritische Operationen und Entschlüsselungen für Weak Condorcet

[1] Wie bei Plain Condorcet angesprochen, wird in der Laufzeit von Condorcet-Methoden die Vorberechnung dazugezählt. Es wird dabei der Fall *evaluatenCondorcet=false* betrachtet. Dies gilt auch für alle weiteren Laufzeiten der Condorcet-Verfahren.

Beschreibung

Die Vorberechnung hat den größten Teil für diese Evaluation bereits erledigt. In der Variablen *gtEqSums* wurden die Siege mit den Unentschieden für jeden Kandidaten addiert. Es muss nur noch ausgewertet werden, welche Kandidaten hier den Wert $n - 1$ besitzen. Falls andere Condorcet-Evaluationen schon zuvor angewendet wurden und diese eine Teilmenge von Kandidaten geliefert haben, werden nur die Kandidaten aus *possibleWinners* geprüft (siehe Abschnitt 6.2). Dies wird auch durch die folgenden Algorithmen unterstützt. Für die Performanzuntersuchung wird davon ausgegangen, dass alle Kandidaten weiterhin in der Liste enthalten sind.

Performanz

Zur Vorberechnung kommen nun hier pro Kandidat der *possibleWinners* ein *Equals*-Vergleich mit Entschlüsselung hinzu. Diese Condorcet-Methode ist die einzige Evaluation, für welche in der Vorberechnung unnötige Operationen durchgeführt werden. Die *Equals*-Vergleiche der Vorberechnung sind für Weak-Condorcet überflüssig. Da jedoch die *Greater-Than*-Vergleiche einen größeren Einfluss auf die Rechenzeit haben, wird der Einfachheit halber von einer Optimierung abgesehen.

Tally-Hiding

Falls ein echter Condorcet-Sieger existiert und *evaluateCondorcet* aktiviert ist, wurde dieser schon in Plain Condorcet berechnet und kann somit von einem Weak-Condorcet-Sieger unterschieden werden. Dies gilt aber für alle Condorcet-Methoden. Bei mehreren Siegern wäre jedoch klar, dass es Weak-Condorcet-Sieger sein müssen. Entschlüsselt wird in diesem Algorithmus für jeden Kandidaten nur, ob er die Bedingung für einen Weak-Condorcet-Sieger erfüllt oder nicht.

6.2.4 Evaluation - Copeland-Methode

Diese ist die einfachste aller implementierten Condorcet-Methoden, welche definitiv mindestens einen Sieger liefert. Für Plain Condorcet und Weak-Condorcet ist dies nicht garantiert. Bei Copeland werden für jeden Kandidaten die Niederlagen von seinen Siegen abgezogen (siehe [FG76]). Der oder die Kandidaten mit der größten positiven Differenz gewinnt. Um negative verschlüsselte Werte zu vermeiden, wird anstatt der Differenz die Summe aus *gtStrictSums* und *gtEqSums* gebildet. Diese beiden Summen sind Ergebnis der Vorberechnung. Die Gesamtsumme entspricht der Verschiebung der angesprochenen Differenz in den positiven Bereich. Im weiteren Verlauf wird diese Summe Copeland-Punkte genannt.

Algorithmus 6.4 Copeland-Methode

```
input    gtSums_strict ← for each candidate the number of gained duels in the result aggregation
         gtEqSums ← for each candidate the number of gained duels in the result aggregation
                    plus the number of draws
         possibleWinners ← list of regarded candidates, candidates represented as indices
         leakMaxCopeland ← true if maximum of Copeland points could be decrypted, else false
output   the Copeland winners
procedure COPELAND(gtSums_strict, gtEqSums, possibleWinners, leakMaxCopeland)
    copelandPoints ← EmptyList(gtSums_strict.length)
    for all cand in possibleWinners do
        copelandPoints[cand] ← gtSums_strict[cand] + gtEqSums[cand]
    end for
    if leakMaxCopeland then
        maxPossibleCopelandPoints ← 2 · gtSums_strict.length − 3
        for all pointsToSearch in maxPossibleCopelandPoints..0 do        // iterate from max down to 0
            winners ← EmptyList()
            for all cand in possibleWinners do
                if decrypt(equals(enc(pointsToSearch), copelandPoints[cand])) is 1 then
                    winners.append(cand)
                end if
            end for
            if winners.length > 0 then
                return winners
            end if
        end for
    else
        return SINGLEWINNERPOINTEVALUATION(copelandPoints)
    end if
end procedure
```

Beschreibung

Zunächst werden im Algorithmus die Copeland-Punkte berechnet und in *copelandPoints* abgelegt. Nun wird zwischen zwei Varianten unterschieden: Eine performantere, welche vor allem *Equals* verwendet und eine welche auf *Greater-Than* aufbaut. Zunächst wird die performantere beschrieben, welche bei leakMaxCopeland = $true$ benutzt wird. Ein Condorcet-Sieger hätte $n-1$ Punkte aus *gtStrictSums* und $n-1$ Punkte aus *gtEqSums*. Somit hätte dieser $2 \cdot (n-1)$ Copeland-Punkte. In der Schleife wird nun im ersten Durchlauf geprüft, ob einer oder mehrere Kandidaten Copeland-Punkte eines Condorcet-Siegers erreichen. Gibt es keinen Condorcet-Sieger, wird nun die gesuchte Copeland-Punktzahl um eins verringert und wieder werden alle Kandidaten geprüft. Dies würde alle Kandidaten betreffen, welche alle Duelle für sich entscheiden bis auf einem Unentschieden. Wenn es auch solch einen Kandidat nicht gibt wird die Schleife weiter ausgeführt, bis ein oder mehrere Kandidaten für eine Punktzahl gefunden werden. Falls mindestens einer die gesuchte Punktzahl besitzt wird der Algorithmus abgeschlossen. Da jeder Kandidat null oder mehr Punkte besitzt, wird definitiv ein Sieger gefunden. Als Alternative kann einfach die SINGLEWINNERPOINTEVALUATION mit den Copeland-Punkten aufgerufen werden.

	Greater-Than	Equals	Multiplicate	Decrypt
Anzahl (Plain Condorcet)	$\frac{n^2-n}{2}$[Wähler]	$\frac{n^2-n}{2}$[Wähler]	0	0
Bei evaluateCondorcet=true	0	n[Wähler]	0	n
Anzahl (Copeland)				
Bei leakMaxCopeland=true	0	$c \cdot n$[Kandidaten]	0	$c \cdot n$
Bei leakMaxCopeland=false	$n-1$[Wähler]	n[Wähler]	$2 \cdot (n-1)$	n
leakMaxCopeland=true				
Lfz. (5 Kandidaten, $c = 3$)	00:09:40 h	00:05:28 h	0	00:00:02 h
Lfz. (20 Kandidaten, $c = 3$)	03:03:36 h	00:56:33 h	0	00:00:08 h
leakMaxCopeland=false				
Lfz. (5 Kandidaten)	00:13:32 h	00:03:28 h	00:00:10 h	00:00:01 h
Lfz. (20 Kandidaten)	03:21:58 h	00:48:31 h	00:00:49 h	00:00:03 h
Entschlüsselung	Liste der Sieger nach Copeland-Methode leakMaxCopeland ⇒ Max(Copeland-Punkte) evaluateCondorcet ⇒ Unterscheidung zu Condorcet-Sieger			

Tabelle 6.4: Zeitkritische Operationen und Entschlüsselungen für die Copeland-Methode

Performanz

In der ersten Schleife wird nur addiert, weshalb diese nicht für die Laufzeit betrachtet wird. Dann kommt die Unterscheidung der Varianten. Die Performanz der SingleWinnerPointEvaluation kann unter Unterabschnitt 5.2.4 betrachtet werden. Hier wird nur noch auf die andere Variante eingegangen. Deren Schleife wird solange ausgeführt, bis ein Kandidat mit gesuchter Punktzahl gefunden wird. Es wird mit der Siegzahl eines möglichen Condorcet-Siegers begonnen, also $2 \cdot (n-1)$. Ein Copeland-Sieger besitzt mindestens $n-1$ Copeland-Punkte. Somit kann die Schleife ein bis $2 \cdot (n-1)-(n-1)+1 = n$ Mal durchgeführt werden. Der Fall n ist sehr unwahrscheinlich, da dieser einem Unentschieden zwischen allen Kandidaten entspricht. Wir gehen in den Beispielen für Laufzeiten konstant von drei Iterationen aus. In der Tabelle steht die Variable c für die Anzahl von Iterationen des Copeland-Algorithmus. Pro Schleifendurchlauf wird für jeden Kandidaten geprüft, ob er genau diese Punktzahl erreicht hat. Da die Copeland-Punkte durch die Anzahl von Kandidaten begrenzt werden und nicht von Wählern, sind diese auch der Anhaltspunkt für die Bitgröße des Vergleiches. Diese Varianten kann n^2-Aufrufe von *Equals* benötigen. Da *Equals* aber noch deutlich schneller als *Greater-Than* ist, ist diese Variante meist schneller als mit SingleWinnerPointEvaluation. In Betracht der Gesamtrechenzeit ist jedoch vor allem die Vorberechnung entscheidend.

Tally-Hiding

Wie in Unterabschnitt 5.2.4 beschrieben wird in der SingleWinnerPointEvaluation nur die Liste der Kandidaten mit der höchsten Punktzahl entschlüsselt. Mehr wird dabei nicht bekannt. Die andere Variante, welche ihre Vorteile in der Performanz hat, hat beim Tally-Hiding ihre Nachteile. Sie verrät mehr als von ihr gefordert wurde. Entschlüsselt wird in jeder Schleifeniteration, ob mindestens ein Kandidat die Punktzahl erreicht hat und damit Sieger ist. Wenn dies der Fall ist, wird auch verraten, um welche Kandidaten es sich handelt. Wenn kein Kandidat gefunden wurde, ändern

sich die Kriterien für den Wahlsieg. Nun wird nach Kandidaten mit einem Punkt weniger gesucht. Somit ist nachher bekannt, wie viele Copeland-Punkte der Sieger hat. Dies sagt etwas über die Struktur des Wahlergebnisses aus. Werden sechs Iterationen benötigt, so ist unabhängig von der Anzahl der Siegerkandidaten für jeden dieser bekannt, dass er mindestens drei Duelle nicht für sich entscheiden konnte. Es könnten zwei Niederlagen und ein Unentschieden, eine Niederlage und drei Unentschieden oder fünf Unentschieden sein. Aus der Anzahl teilnehmender und siegender Kandidaten können weitere Rückschlüsse gezogen werden.

6.2.5 Evaluation - Minimax

Minimax ist weniger direkt an Plain Condorcet angelehnt als die letzten beiden Methoden. Diese waren einfach zu berechnende Erweiterungen, um Sieger mit schwächeren Kriterien direkt aus dem Ergebnis der Vorberechnung zu finden. Minimax ist jedoch ein eigenes Konzept, welches nur in manchen Varianten das Condorcet-Kriterium erfüllt (siehe [KAAR14]). Minimax-Margins und Minimax-Winning-Votes sind Varianten, welche dieses erfüllen und implementiert wurden. Minimax ist auch als Simpson-Kramer-Methode bekannt.

Wie bei allen Condorcet-Methoden werden auch die direkten Duelle betrachtet. Für das Ergebnis ist entscheidend, wie oft in den Duellen der eine oder der andere gewinnt. Da die beiden implementierten Methoden dem Condorcet-Kriterium entsprechen, darf zunächst auch hier Plain Condorcet ausgeführt werden. Plain Condorcet ist deutlich performanter als Minimax und die Berechnungen, welche für Plain Condorcet getätigt werden, werden auch für Minimax benötigt. Deshalb ist es auch in Betracht der Performanz sinnvoll, zunächst Plain Condorcet durchzuführen.

Minimax bewertet für jeden Kandidaten alle seine Duelle. Die Metrik, um diese Duelle zu bewerten, hängt von der Minimaxvariante ab. Für jeden Kandidaten wird das schlechteste Ergebnis gesucht und betrachtet. Unter den schlechtesten Ergebnissen wird wiederum nach dem besten gesucht, um den Sieger der Wahl zu finden.

Beschreibung

In zwei verschachtelten Schleifen wird über die Kandidaten iteriert. Für jeden Kandidaten werden alle Duelle betrachtet. Mit einer Metrik wird das Duell gemessen. Hierbei muss gelten: Je größer der Wert, desto besser für cand_a und die Werte von verlorenen Duellen müssen im Bereich 0 bis *nVotes* liegen. Betrachtet werden nur die Niederlagen der Kandidaten, da es sonst unter Umständen nicht konform zu Condorcet ist. Durch die Methode IfThenElse (siehe Abschnitt 3.4) wird wenn cand_a das Duell gewinnt und somit $resultDuelMatrix[\mathsf{cand}_a][\mathsf{cand}_b]$ den Wert eins hat, immer der maximale Wert n_{votes} verwendet, da die Deutlichkeit der Siege keine Rolle spielen darf. Wenn nicht, wird das Ergebnis der Metrik in die Liste *values* geschrieben.

Nach dem Durchlauf der inneren Schleife werden die *values* nach ihrem Minimum untersucht. Dazu wird die Methode GetMinimum (siehe Abschnitt 3.4) verwendet. Danach ist für jeden Kandidaten das für ihn schlechteste Duell in der Variablen *worstResults* gespeichert. Von diesen wird nun der beste Kandidat gesucht. Das entspricht dem Suchen nach der größten Zahl und des jeweiligen Kandidaten. Zusammengefasst sind diese beiden Algorithmen schon in der Borda-Evaluation SingleWinnerPointEvaluation.

Algorithmus 6.5 Minimax

```
input     aggregation ← original aggregation object (see Unterabschnitt 6.2.1)
          n_votes ← number of valid votes
          duelMatrix^strict_result ← defines for duels which candidate wins more duels
          possibleWinners ← list of regarded candidates, candidates represented as indices
output    the Minimax winners
procedure MINIMAX(aggregation, n_votes, duelMatrix^strict_result, possibleWinners)
    worstResults ← EmptyList(aggregation.length)
    for all cand_a in possibleWinners do
        values ← EmptyList()
        for all cand_b in possibleWinners do
            if cand_a != cand_b then                          // ignore comparison with itself
                value ← METRIC(n_votes, aggregation[cand_a][cand_b], aggregation[cand_b][cand_a])
                values.append(IFTHENELSE(duelMatrix^strict_result[cand_a][cand_b], n_votes, value))
            end if
        end for
        worstResults[cand_a] ← GETMINIMUM(values)
    end for
    return SINGLEWINNERPOINTEVALUATION(worstResults)
end procedure
```

	Greater-Than	Equals	Multiplicate	Decrypt
Anzahl (Plain Condorcet)	$\frac{n^2-n}{2}$[Wähler]	$\frac{n^2-n}{2}$[Wähler]	0	0
Bei evaluateCondorcet=true	0	n[Wähler]	0	n
Anzahl (Minimax)	$n^2 - n - 1$[Wähler]	n[Wähler]	$3 \cdot n^2 - 3 \cdot n - 2$	n
Lfz. (5 Kandidaten)	00:28:01 h	00:03:28 h	00:01:15 h	00:00:01 h
Lfz. (20 Kandidaten)	09:09:51 h	00:48:31 h	00:24:28 h	00:00:03 h

Entschlüsselung	Liste der Minimax-Sieger evaluateCondorcet ⇒ Unterscheidung zu Condorcet-Sieger

Tabelle 6.5: Zeitkritische Operationen und Entschlüsselungen für Minimax

Performanz

Die innere Schleife wird $n^2 - n$ Mal ausgeführt. Das $-n$ kommt durch das Überspringen im Falle $\text{cand}_a = \text{cand}_b$. In der inneren Schleife wird das Messen der Metrik ausgeführt. Dies ist jedoch nur ein Auslesen und eventuell ein Addieren oder Subtrahieren von bereits berechneten Werten. Deshalb spielt die Metrik für die Performanz keine Rolle. Da beim IFTHENELSE-Aufruf n_{votes} nicht verschlüsselt ist, ist für die Performanz nur ein *Multiplicate* statt zwei zu betrachten (siehe Abschnitt 3.4). Nach dem Beenden der inneren Schleife besitzt *values* $n - 1$ Werte. Somit wird die GETMINIMUM-Methode mit Listenlänge $n - 1$ verwendet. Dies geschieht pro Kandidat ein Mal. Die Anzahl der Operationen in den Untermethoden wird in den folgenden Berechnungen berücksichtigt. Zum Schluss wird ein Mal die SINGLEWINNERPOINTEVALUATION mit Listenlänge n aufgerufen. Alle durchgeführten Vergleiche arbeiten mit Zahlen von 0 bis *nVotes*. Damit sind diese abhängig von der Anzahl von Wählern und werden für die Anzahl der Bits mit dem Stichwort *Wähler* gekennzeichnet.

Man kommt auf folgende Zahl von Durchführungen:

Greater-Than:	$n \cdot \underbrace{(n-1-1)}_{\text{GetMinimum}}[Wghler] + \underbrace{(n-1)}_{\text{SingleWinnerPointEvaluation}}[Wghler]$ $= n^2 - 2 \cdot n + n - 1 = n^2 - n - 1[Wghler]$
Equals:	$\underbrace{n}_{\text{SingleWinnerPointEvaluation}}[Wghler]$
Multiplicate:	$\underbrace{n^2 - n}_{\text{innere Schleife}} + \underbrace{(n-1-1) \cdot 2 \cdot n}_{\text{GetMinimum}} + \underbrace{(n-1) \cdot 2}_{\text{SingleWinnerPointEvaluation}}$ $= (n^2 - n) + ((n-2) \cdot n + (n-1)) \cdot 2$ $= n^2 - n + (n^2 - 2 \cdot n + n - 1) \cdot 2$ $= n^2 - n + 2 \cdot n^2 - 4 \cdot n + 2 \cdot n - 2$ $= 3 \cdot n^2 - 3 \cdot n - 2$
Decrypt:	$\underbrace{n}_{\text{SingleWinnerPointEvaluation}}$

Tally-Hiding

Bis zum Aufruf von SingleWinnerPointEvaluation wird in diesem Algorithmus nichts öffentlich. Die GetMinimum- und IfThenElse-Methoden sind beide sicher wie im Kapitel Abschnitt 3.4 gezeigt. Es kann das Minimum aller Kandidaten gefunden werden, ohne etwas über einen der Werte zu verraten. SingleWinnerPointEvaluation verrät schlussendlich nur, welche der Kandidaten die beste Punktzahl erreichen und somit bei dieser Minimaxvariante gewinnen. Selbst der Wert der besten Punktzahl bleibt dabei geheim.

Die beiden implementierten Varianten heißen Minimax-Margins und Minimax-Winning-Votes. Minimax-Margins betrachtet immer die Differenz aus den Siegzahlen (siehe [KAAR14]). Wenn im Duell cand_a gegen cand_b auf 20 Stimmzetteln gilt $\text{cand}_a \geq \text{cand}_b$ und 30 Mal $\text{cand}_b \geq \text{cand}_a$, so ist in diesem Duell der Wert der Metrik für cand_a -10 und für cand_b $+10$. Um in einen Wertebereich von 0 bis *nVotes* für die verlorenen Duelle zu kommen, werden die Werte mit *nVotes* addiert. Somit lautet der Pseudocode der Metrik für Minimax-Margins folgendermaßen:

Algorithmus 6.6 Minimax-Margins Metrik

```
procedure Metric(n_votes, winCount, looseCount)
    return enc(winCount) − looseCount + n_votes
end procedure
```

Bei Minimax-Winning-Votes spielt für einen Kandidaten nur die Anzahl der Niederlagen eines einzelnen Duells eine Rolle. Somit sind Unentschieden auf einzelnen Wahlzetteln weniger schlimm für einen Kandidaten (siehe [KAAR14]). Die Metrik sieht hier so aus:

Algorithmus 6.7 Minimax-Winning-Votes Metrik

```
procedure Metric(n_votes, winCount, looseCount)
    return enc(n_votes) − looseCount
end procedure
```

6.2.6 Evaluation - Smith

Smith ist wie Condorcet ein Wahlkriterium (siehe [Sch11; Str80]). Es ist stärker als das Condorcet-Kriterium, da jede Smith-Methode automatisch eine Condorcet-Methode ist. Das Kriterium definiert für jede Wahl eine Teilmenge von Kandidaten, aus welcher der Sieger kommen muss. Dieses Smith-Set ist definiert als die kleinste Menge von Kandidaten, welche gegenüber allen anderen in jedem Duell priorisiert werden. Die Definition ist ähnlich zu der des Condorcet-Siegers, nur können hier mehrere Kandidaten enthalten sein. In der Literatur ist das Smith-Set im englischen oft als *top cycle* zu finden. Gibt es einen Condorcet-Sieger, so enthält das Smith-Set nur diesen einen Kandidaten. Im Gegensatz zum Condorcet-Sieger existiert das Smith-Set immer und ist niemals leer. Es kann aber alle Kandidaten der Wahl enthalten.

Smith wird selten als vollständiges Wahlsystem verwendet, da nur im Falle eines Condorcet-Siegers ein eindeutiges Ergebnis eintritt. Es kann aber verwendet werden, um andere Evaluationen konform zum Smith-Kriterium zu machen. Minimax ist zum Beispiel sinnvoll in Kombination mit Smith zu verwenden. Dazu müssen nur die beiden Evaluatoren hintereinander ausgeführt werden. Außerdem kann es aus Gründen der Performanz sinnvoll sein, diesen Algorithmus zuerst auszuführen, wenn ein System das Smith-Kriterium erfüllt. So werden einige Kandidaten direkt ausgeschlossen und müssen nicht mehr evaluiert werden. Nachteil ist aber, dass es öffentlich ist, welche Kandidaten zum Smith-Set gehören und welche nicht.

Dieser Algorithmus beschränkt sich nicht auf die Liste von *possibleWinners*. Erst am Ende nach der Smith-Evaluation werden nur die Kandidaten in die Rückgabeliste hinzugefügt, welche auch in der *possibleWinners*-Liste zu finden sind. Damit ist es konform zum Sinn der *possibleWinners*, aber eine Performanzverbesserung durch vorangehenden Ausschluss von Kandidaten ist somit nicht möglich.

Algorithmus 6.8 Smith

```
input    gtSums_strict ← for each candidate the number of gained duels in the result aggregation
         gtEqSums ← for each candidate the number of gained duels in the result aggregation
                    plus the number of draws
         possibleWinners ← list of regarded candidates, candidates represented as indices
         leakMinCopeland ← true if fast variant should be used, else false
output   the Smith-Set
procedure SMITH(gtSums_strict, gtEqSums, possibleWinners, leakMinCopeland)
    n_cand ← gtSums_strict.length
    copelandPoints ← EmptyList(n_cand)
    for all cand in 0..n_cand − 1 do
        copelandPoints[cand] ← gtSums_strict[cand] + gtEqSums[cand]
    end for
    candCount ← enc(0)
    pointSum ← enc(0)
    finished ← enc(0)
    smithSetIndicator ← EmptyList(n_cand)          // in this case: initialised with encrypted zeros
    pointsToSearch ← 2 · (n_cand − 1)
    while true do
        matchCount, matchCandIndicators ← MATCHPOINTS(copelandPoints, enc(pointsToSearch))
        if leakMinCopeland then
            candCount ← candCount + matchCount
            pointSum ← pointSum + matchCount · pointsToSearch
        else
            candCount ← candCount + multiplicate(matchCount, 1 − finished)
            pointSum ← pointSum + multiplicate(matchCount · pointsToSearch, 1 − finished)
        end if
        for all cand in 0..n_cand − 1 do
                    // every candidate could match only one time, so indicator is never bigger than 1
            smithSetIndicator[cand] ← smithSetIndicator[cand] + matchCandIndicators[cand]
        end for
        foundOne ← enc(1) − equals(candCount, enc(0))
        smithSetCondition ← equals(multiplicate(candCount, enc(2 · n_cand − 1) − candCount), pointSum)
        finished ← multiplicate(foundOne, smithSetCondition)
        if leakMinCopeland then
            if decrypt(finished) is 1 then
                break
            end if
        else if pointsToSearch is 1 then          // candidate with 0 points couldn't be part of Smith-Set
            break
        end if
        pointsToSearch ← pointsToSearch − 1
    end while
    smithSet ← EmptyList()
    for all cand in possibleWinners do
        if decrypt(smithSetIndicator[cand]) is 1 then
            smithSet.append(cand)
        end if
    end for
    return smithSet
end procedure
```

	Greater-Than	Equals	Multiplicate	Decrypt
Anzahl (Plain Condorcet)	$\frac{n^2-n}{2}$[Wähler]	$\frac{n^2-n}{2}$[Wähler]	0	0
Bei evaluateCondorcet=true	0	n[Wähler]	0	n
Anzahl (Smith)				
Bei leakMinCopeland=true	0	$s \cdot (n+1)$[Kandidaten] s[Wähler]	$s \cdot 2$	$s+n$
Bei leakMinCopeland=false	0	$2 \cdot (n^2-1)$[Kandidaten] $2 \cdot (n-1)$[Wähler]	$8 \cdot (n-1)$	n
leakMinCopeland=true				
Lfz. (5 Kandidaten, $s = 4$)[2]	00:09:40 h	00:08:18 h	00:00:10 h	00:00:01 h
Lfz. (5 Kandidaten, $s = 7$)[3]	00:09:40 h	00:12:47 h	00:00:18 h	00:00:02 h
Lfz. (20 Kandidaten, $s = 4$)	03:03:36 h	01:02:32 h	00:00:10 h	00:00:03 h
Lfz. (20 Kandidaten, $s = 37$)[4]	03:03:36 h	03:36:23 h	00:01:35 h	00:00:08 h
leakMinCopeland=false				
Lfz. (5 Kandidaten)	00:09:40 h	00:14:17 h	00:00:41 h	00:00:01 h
Lfz. (20 Kandidaten)	03:03:36 h	03:41:03 h	00:03:16 h	00:00:03 h
Entschlüsselung	Smith-Set leakMinCopeland ⇒ minimale Copeland-Punkten innerhalb dieses evaluateCondorcet ⇒ Unterscheidung zu Condorcet-Sieger			

Tabelle 6.6: Zeitkritische Operationen und Entschlüsselungen für das Smith-Verfahren

Beschreibung

Dieser Algorithmus arbeitet mit Copeland-Punkten. Einfacher zu verstehen ist er, wenn man zunächst die *leakMinCopeland*-Variante betrachtet. Dann verhält sich der Algorithmus folgendermaßen. Wie in der Copeland-Evaluation werden erst einmal die *gtEqSums* mit den *gtStrictSums* aufaddiert zu *copelandPoints*. In der Schleife wird von der höchst möglichen Copeland-Punktezahl abwärts iteriert, bis eine bestimmte Bedingung eintritt. In jedem Schleifendurchlauf wird nach allen Kandidaten mit entsprechender Copeland-Punktezahl gesucht. Um dies zu ermöglichen, werden zuvor einige Variablen initialisiert. *candCount* gibt an, wie viele Kandidaten bereits gefunden wurden. *pointSum* ist die Summe der Copeland-Punkte aller gefunden Kandidaten. *finished* indiziert verschlüsselt, ob das Smith-Set komplett ist. *smithSetIndicator* gibt durch verschlüsselte Nullen und Einsen für jeden Kandidaten an, ob der Kandidat sicher Teil des Smith-Sets ist oder nicht. Es wird bei der Initialisierung davon ausgegangen, dass keiner der Kandidaten im Smith-Set ist. *pointsToSearch* ist die Copeland-Punktezahl, nach welcher gesucht wird. Diese beginnt bei der maximalen möglichen Copeland-Punktzahl $2 \cdot (\mathsf{n}_{\mathsf{cand}} - 1)$. Zu Beginn der Schleife werden mit der Methode *matchPoints* alle Kandidaten mit der Copeland-Punktezahl *pointsToSearch* gesucht. *matchCandIndicators* enthält in verschlüsselter Form, welche Kandidaten diese Punktzahl haben und *matchCount* ist die Anzahl der Kandidaten. Danach werden die Variablen *candCount*, *pointSum* und *smithSetIndicator* entsprechend

[2] $s = 4$ entspricht 3 oder 4 Kandidaten im Smithset

[3] Bei $s = 7$ sind alle 5 Kandidaten im Smithset, (entspricht dem Worst Case für die Performanz)

[4] Bei $s = 37$ sind alle 20 Kandidaten im Smithset, (entspricht dem Worst Case für die Performanz)

ihrer Definition auf den neuesten Stand gebracht. Ein Smith-Set ist genau dann gefunden, wenn für die Summe der Punkte und die Anzahl der Kandidaten die Gleichung

$$pointSum = candCount \cdot (2 \cdot n_{cand} - 1 - candCount)$$

gilt. Diese Bedingung garantiert, dass alle gefunden Kandidaten gegen alle andere in jedem direkten Duell gewinnen. Solch eine Gruppe von Kandidaten wird dominierende Menge genannt. Am Ende des Abschitts für Smith folgt eine Intuition dazu, warum diese Bedingung für dominierende Mengen gilt und warum sie nur für diese zutrifft. Ob die Bedingung zutrifft, wird in *smithSetCondition* geschrieben. Da die Abbruchbedingung der Schleife aber auch bei null Kandidaten gilt, muss auch geprüft werden, ob bereits mindestens einer gefunden wurde. Beide boolschen Werte sind verschlüsselt. Durch ein *Multiplicate* wird ein logisches Und umgesetzt. Da von der höchsten Punktzahl abwärts iteriert wird und beim ersten Erfüllen der Bedingung abgebrochen wird, entspricht die gefundene Menge der kleinsten dominierenden Menge. Das entspricht der Definition eines Smith-Sets. Die Liste der Indikatoren muss nach Abschluss der Schleife noch entschlüsselt werden. Daraus ist auszulesen, welche Kandidaten im Smith-Set enthalten sind.

Wenn *leakMinCopeland* nicht gewollt ist, wird die Variable *finished* nie entschlüsselt. Die Schleife kann nicht abgebrochen werden. Es wird nur verhindert, dass Zuweisungen zu Beginn der Schleife die Werte verändern, da sie das Smith-Set bereits enthalten. Da die Operationen mathematisch verschlüsselt ablaufen, kann nicht nachvollzogen werden, in welchem Schleifendurchlauf das Smith-Set vollständig ist. Die Ciphertexte ändern sich weiterhin, nur deren Werte bleiben konstant. Die Schleife iteriert dann bis zur Copeland-Punkzahl eins.

Performanz

Die Definition des Smith-Sets lässt einen rechenintensiven Algorithmus vermuten. Durch Nutzung der Copeland-Punkte hält sich der Berechnungsaufwand jedoch in Grenzen. Alle Operationen können durch Komplexitätsklasse $O(n)$ abgeschätzt werden. Ohne *leakMinCopeland* sind es immer $2 \cdot (n_{cand} - 1)$ Iterationen der Schleife. Mit dieser Option hingegen sind es nur ein bis $2 \cdot (n_{cand} - 1)$. Die Wahrscheinlichkeit, dass alle Kandidaten im Smith-Set sind, ist bei vielen Kandidaten und zufälligen Wahlen nicht so unwahrscheinlich. Ansonsten sind kleine Smith-Sets und somit weniger Iterationen wahrscheinlicher als große. Die Zahl der Iterationen wird mit Variable s betrachtet. Bei jedem Schleifendurchlauf wird zu Beginn die *matchPoints*-Methode verwendet. In den folgenden Berechnungen finden bei *leakMinCopeland*= $false$ zwei Multiplikationen statt. Für die Abbruchbedingung sind zwei *Equals*, zwei *Multiplicate* und bei *leakMinCopeland* noch ein *Decrypt* nötig. Der erste Test ist, ob überhaupt ein Kandidat gefunden wurde. Für dieses *Equals* sind Werte von 0 bis n_{cand} möglich. In der zweiten Gleichheitsprüfung wird die Summe der Copeland-Punkte aller bereits gefundenen Kandidaten verglichen. Diese kann Werte bis $n_{cand} \cdot (n_{cand} - 1)$ annehmen. Somit wird hier für die Bitzahl der Operation die Markierung [Kandidaten^2] verwendet. Es wird hier wie bei [Wähler] pauschal mit einer Bitzahl von 16 gerechnet. Nach Abschluss der Schleife folgt ein weiterer Aufruf von *Decrypt* pro Kandidat. Folgende Berechnungen fassen alle relevanten Funktionsaufrufe zusammen:

leakMinCopeland=true

Greater-Than: 0	
Equals:	$s \cdot (\underbrace{n}_{\text{MatchPoints}}[\text{Kandidaten}] + \underbrace{1[\text{Kandidaten}] + 1\text{Kandidaten}^2]}_{Abbruchbedingung})$
	$= s \cdot (n+1)[\text{Kandidaten}] + s[\text{Kandidaten}^2]$
Multiplicate: $s \cdot 2$	
Decrypt: $s + n$	

leakMinCopeland=false

Greater-Than: 0	
Equals:	$2 \cdot (n-1) \cdot (\underbrace{n}_{\text{MatchPoints}}[\text{Kandidaten}] + \underbrace{1[\text{Kandidaten}] + 1[\text{Kandidaten}^2]}_{Abbruchbedingung})$
	$= 2 \cdot (n-1) \cdot (n+1)[\text{Kandidaten}] + 2 \cdot (n-1)[\text{Kandidaten}^2]$
	$= 2 \cdot (n^2-1)[\text{Kandidaten}] + 2 \cdot (n-1)[\text{Kandidaten}^2]$
Multiplicate: $2 \cdot (n-1) \cdot 4 = 8 \cdot (n-1)$	
Decrypt: n	

Tally-Hiding

Bis zum abschließenden Entschlüsseln sind alle Daten geheim. In der Schleife ist weder welche noch wie viele Kandidaten gefundenen wurden bekannt. Nur die Anzahl der Schleifendurchläufe wird bei *leakMinCopeland* nicht verschleiert. Hierdurch kann in diesem Fall nachvollzogen werden, wie groß die kleinste Copeland-Punktzahl innerhalb des Smith-Sets ist. Die minimale Copeland-Punktzahl ist die einzige Konsequenz und Information, welche durch das *Decrypt* in der Schleife im Falle von *leakMinCopeland* entschlüsselt wird. Wenn diese Option nicht gewählt wurde, kann in der gesamten Schleife keine Information über das Wahlergebnisses gewonnen werden. Das *Decrypt* am Ende der Evaluation verrät genau das geforderte Ergebnis: Für alle Kandidaten aus *possibleWinners* wird beantwortet, ob er Teil des Smith-Sets ist oder nicht.

Beweisansatz

Da auch dieser Algorithmus selbst entwickelt wurde und die Bedingung nicht selbsterklärend ist, muss noch gezeigt werden, dass

$$pointSum = candCount \cdot (2 \cdot \mathsf{n}_{\mathsf{cand}} - 1 - candCount)$$

genau für dominierende Mengen gilt. Eine dominierende Menge ist eine Teilmenge von Kandidaten, welche in jedem direkten Duell gegen einen nicht in dieser Menge enthaltenen Kandidaten gewinnen. Es wird an dieser Stelle aber nur eine Intuition für diesen Sachverhalt angegeben und

kein abgeschlossener Beweis. Dieser Beweisansatz zeigt nacheinander beide Richtungen folgender logischer Aussage:

$$S \subseteq C \text{ ist dominierend} \Leftrightarrow \sum_{s \in S} cop(s) = |S| \cdot (2 \cdot \mathsf{n}_{\mathsf{cand}} - 1 - |S|)$$

C ist dabei die Gesamtmenge von Kandidaten. $|S|$ ist die Größe der dominierenden Menge und entspricht im Code dem *candCount*. $cop : C \to \mathbb{N}_0$ gibt die Copeland-Punkte für jeden Kandidaten an. Im Code wird die gesamte Summe durch *pointSum* referenziert.

Angenommen es existiert eine dominierende Menge $S \subseteq C$. Dann existiert für diese auch eine Summe aller Copeland-Punkte. Copeland-Punkte werden folgendermaßen vergeben: Für jedes Duell zwischen zwei Kandidaten werden aus der *aggregation* zwei Copeland-Punkte verteilt. Gewinnt einer, so erhält er zwei. Bei einem Unentschieden bekommt jeder einen Punkt. Wir teilen alle Duelle in drei unterschiedlichen Typen auf:

(1) Duelle, in welchen beide Kandidaten Elemente von S sind

(2) Duelle, in welchen einer der Kandidaten Element von S ist und der andere nicht

(3) Duelle, in denen beide Kandidaten nicht Element von S sind

Für die Gesamtsumme $\sum_{s \in S} cop(s)$ werden nun die Punkte betrachtet, welche aus den unterschiedlichen Kategorien an Kandidaten aus S gehen. Für alle Duelle aus (1) gilt: Egal wie es ausgeht, es gibt immer zwei Punkte pro Duell für Kandidaten aus S. Davon gibt es $\frac{|S| \cdot (|S|-1)}{2}$ Stück. Auch für jedes Duell aus (2) kommen zwei Punkte zur Summe hinzu, da laut Definition der Kandidat aus der dominierenden Menge alle Duelle nach außen gewinnt. Es gibt $|S| \cdot (\mathsf{n}_{\mathsf{cand}} - |S|)$ Duelle aus Kategorie (2). Duelle aus Kategorie (3) können keine Punkte für die Summe liefern. Somit muss gelten:

$$\begin{aligned}
\sum_{s \in S} cop(s) &= \underbrace{2 \cdot \frac{|S| \cdot (|S| - 1)}{2}}_{(1)} + \underbrace{2 \cdot (|S| \cdot (\mathsf{n}_{\mathsf{cand}} - |S|))}_{(2)} + \underbrace{0}_{(3)} \\
&= 2 \cdot |S| \cdot (\frac{|S| - 1}{2} + \mathsf{n}_{\mathsf{cand}} - |S|) \\
&= |S| \cdot (|S| - 1 + 2 \cdot \mathsf{n}_{\mathsf{cand}} - 2 \cdot |S|) \\
&= |S| \cdot (2 \cdot \mathsf{n}_{\mathsf{cand}} - 1 - |S|)
\end{aligned}$$

Damit ist gezeigt, dass für eine dominierende Menge $S \subseteq C$ die Gleichung $\sum_{s \in S} cop(s) = |S| \cdot (2 \cdot \mathsf{n}_{\mathsf{cand}} - 1 - |S|)$ gilt.

Nun wird angenommen, dass $\sum_{s \in S} cop(s) = |S| \cdot (2 \cdot \mathsf{n}_{\mathsf{cand}} - 1 - |S|)$ für eine Menge $S \in C$ gilt. Auch hier können die drei Kategorien betrachtet werden. Somit besteht die Summe wieder aus folgenden Bestandteilen, wobei die Variablen *cat_1*, *cat_2* und *cat_3* sinngemäß den Summen aus den Kategorien entsprechen:

$$\sum_{s \in S} cop(s) = \text{cat\_1} + \text{cat\_2} + \text{cat\_3}$$

Für jede Menge gilt, dass jedes Duell aus Kategorie (1) zwei Punkte für solch eine Summe liefert. Das gilt nicht nur für dominierende. Damit gilt wieder:

$$\text{cat\_1} = 2 \cdot \frac{|S| \cdot (|S| - 1)}{2}$$

Außerdem muss auch hier cat_3 = 0 gelten, egal ob S dominierend ist oder nicht. Für Kategorie (2) bleibt somit folgende Punkzahl übrig:

$$\begin{aligned}
\text{cat_2} &= \sum_{s \in S} cop(s) - \text{cat_1} - \text{cat_3} \\
&= |S| \cdot (2 \cdot \mathsf{n}_{\mathsf{cand}} - 1 - |S|) - 2 \cdot \frac{|S| \cdot (|S| - 1)}{2} - 0 \\
&= |S| \cdot (2 \cdot \mathsf{n}_{\mathsf{cand}} - 1 - |S|) - |S| \cdot (|S| - 1) \\
&= |S| \cdot (2 \cdot \mathsf{n}_{\mathsf{cand}} - 1 - |S| - |S| + 1) \\
&= |S| \cdot (2 \cdot \mathsf{n}_{\mathsf{cand}} - 2 \cdot |S|) \\
&= 2 \cdot |S| \cdot (\mathsf{n}_{\mathsf{cand}} - |S|)
\end{aligned}$$

Es gibt $|S| \cdot (\mathsf{n}_{\mathsf{cand}} - |S|)$ Duelle zwischen einem Kandidaten innerhalb S und einem außerhalb S. Somit müssen im Schnitt für jedes dieser Duelle $\frac{2 \cdot |S| \cdot (\mathsf{n}_{\mathsf{cand}} - |S|)}{|S| \cdot (\mathsf{n}_{\mathsf{cand}} - |S|)} = 2$ Punkte an den jeweiligen Kandidaten aus S gehen. Da die Punktzahl aber nie größer als 2 sein kann, müssen für jedes Duell aus Kategorie (2) 2 Punkte zur Summe addiert werden. Das heißt, dass in jedem dieser Duelle der Kandidat aus S gewinnt. Damit ist S eine dominierende Menge und damit ist auch die Rückrichtung gezeigt.

6.2.7 Evaluation - Schulze

Minimax ist bis hierhin die einzig komplexere Condorcet-Methode, welche im Gegensatz zu Smith dafür konzipiert ist, einen einzelnen Sieger zu finden. Sie erfüllt aber nicht das Smith-Kriterium und andere Kriterien. So kann Minimax zum Beispiel bei ungewöhnlichen Stimmabgaben dasselbe Ergebnis erzeugen, wie wenn alle Stimmzettel invertiert werden. Mehr hierzu ist unter in der Veröffentlichung von Markus Schulze (siehe [Sch11]) zu finden. Die Schulze-Methode behebt dieses Problem und ist deshalb die am häufigsten verwendete Condorcet-Methode. Wie bereits erwähnt, wird sie auf Piratenparteitagen, bei Ubuntu und bei Debian intern verwendet (siehe [Adl17]). Der größte Nachteil dieser Evaluation ist die hohe Rechenkomplexität.

Ausgangspunkt für die Schulze-Methode ist wieder die normale Condorcet-Aggregierung. Es ist also für jedes Duell bekannt, wie oft der eine über dem anderen gelistet wurde und umgekehrt. Die Auswertung lässt sich am besten mit Graphen beschreiben. Dabei sind die Kandidaten die Knoten.

Die gerichteten Kanten tragen Gewichte mit dem Wert einer Metrik, welche Duelle bewerten kann. Wie bei Minimax gibt es hierfür mehrere Varianten. Es wird im Folgenden die *margin*-Variante betrachtet, da sie die einfachste Metrik besitzt und intuitiv ist (siehe [Sch11]). Dabei trägt die gerichtete Kante (cand_a, cand_b) den Wert $aggregation[\mathsf{cand}_a][\mathsf{cand}_b] - aggregation[\mathsf{cand}_b][\mathsf{cand}_a]$. Zwischen jedem Knotenpaar $\{\mathsf{cand}_a, \mathsf{cand}_b\}$ existieren die Kanten (cand_a, cand_b) und (cand_b, cand_a).

Es wird nun eine neue leere Duell-Matrix betrachtet. Jedes Kandidatenpaar (cand_a, cand_b) wird nun betrachtet und nach allen möglichen Wegen von cand_a nach cand_b untersucht. Der Wert eines Weges entspricht dem Wert des schwächsten Gliedes, also dem niedrigsten Wert einer verwendeten Kante. Gefüllt wird jede Zelle mit dem höchst möglichen Wert eines Weges von Zeilenkandidat zu Spaltenkandidat. Ein Beispiel könnte im Graph und dann als Tabelle der besten Pfade folgendermaßen aussehen:

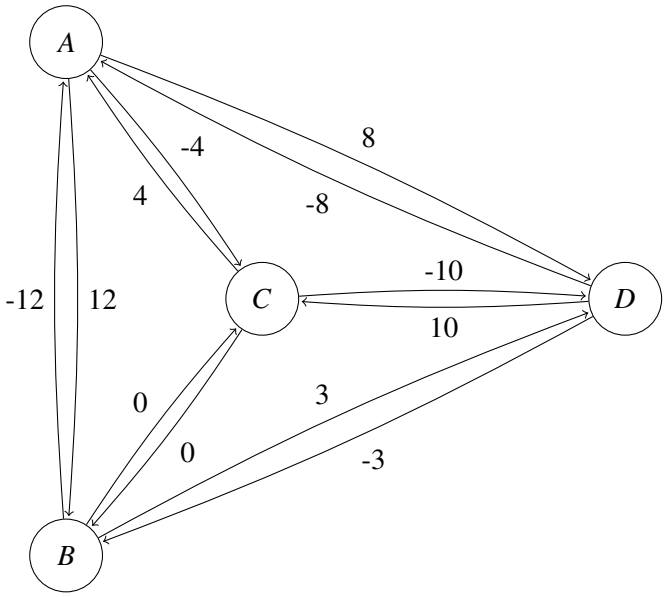

Abbildung 6.2: Beispiel Schulze: Graph der *margin*-Metrik

	A	B	C	D
A	-	**12** (A-B)	**8** (A-D-C)	**8** (A-D)
B	3 (B-D-C-A)	-	3 (B-D-C)	3 (B-D)
C	4 (C-A)	**4** (C-A-B)	-	3 (C-A-B-D)
D	4 (D-C-A)	**4** (D-C-A-B)	**10** (D-C)	-

Tabelle 6.7: Beispiel Schulze: Beste Pfade

Nun werden in dieser Tabelle alle Duelle neu betrachtet. Die Zelle (cand_a, cand_b) wird mit (cand_b, cand_a) verglichen und die mit größerem Wert markiert. In [Sch11] ist bewiesen, dass es mindestens einen Kandidaten gibt, der in diesem Fall mit einem $\geq$ alle Duelle gewinnt. Dieser ist Sieger des Wahlverfahrens. Im Beispiel ist dies Kandidat A. Mit dieser Condorcet-Methode lässt sich sogar einen Rangfolge zwischen allen Kandidaten aufstellen. Wir konzentrieren uns im Rahmen der aktuellen Implementierung jedoch nur auf den Sieger der Wahl.

Algorithmus 6.9 Schulze

```
input     aggregation ← matrix with added ballots (see Unterabschnitt 6.2.1)
          n_votes ← number of valid votes
          possibleWinners ← list of regarded candidates, candidates represented as indices
output    the winner by Schulze method
procedure SCHULZE(aggregation, n_votes, possibleWinners)
    path ← EmptyMatrix(possibleWinners.length)
    for all cand_a in possibleWinners do
        for all cand_b in possibleWinners do
            if cand_a != cand_b then
                path[cand_a][cand_b] ← aggregation[cand_a][cand_b] − aggregation[cand_b][cand_a]
            end if
        end for
    end for
    for all cand_a in possibleWinners do
        for all cand_b in possibleWinners do
            for all cand_c in possibleWinners do
                if cand_a != cand_b and cand_a != cand_c and cand_c != cand_b then
                    minimum ← GETMINIMUM([path[cand_b][cand_a], path[cand_a][cand_c]])
                    path[cand_b][cand_c] ← GETMAXIMUM([path[cand_b][cand_c], minimum])
                end if
            end for
        end for
    end for
    duelMatrix^weak_Schulze ← EmptyMatrix(possibleWinners.length)
    for all cand_a in possibleWinners do
        for all cand_b in possibleWinners do
            if cand_a < cand_b then                    // < possible because candidates are seen as indices
                gtEq ← greaterThan(path[cand_a][cand_b], path[cand_b][cand_a])
                eq ← equals(path[cand_a][cand_b], path[cand_b][cand_a])
                duelMatrix^weak_Schulze[cand_a][cand_b] ← gtEq
                duelMatrix^weak_Schulze[cand_b][cand_a] ← enc(1) − gtEq + eq
            end if
        end for
    end for
    winners ← EmptyList()
    for all cand_a in possibleWinners do
        winningSum ← enc(0)
        for all cand_b in possibleWinners do
            if cand_a != cand_b then
                winningSum ← winningSum + duelMatrix^weak_Schulze[cand_a][cand_b]
            end if
        end for
        winningIndicator ← equals(winningSum, enc(possibleWinners.length − 1))
        if decrypt(winningIndicator) is 1 then
            winners.append(cand_a)
        end if
    end for
    return winners
end procedure
```

	Greater-Than	Equals	Multiplicate	Decrypt
Anzahl (Plain Condorcet)	$\frac{n^2-n}{2}$[Wähler]	$\frac{n^2-n}{2}$[Wähler]	0	0
Bei evaluateCondorcet=true	0	n[Wähler]	0	n
Anzahl (Schulze)	$2 \cdot n^3 - 5,5 \cdot n^2 + 3,5 \cdot n$ [Wähler]	$\frac{n^2-n}{2}$[Wähler] n[Kandidaten]	$4 \cdot n^3 - 12 \cdot n^2 + 8 \cdot n$	n
Lfz. (5 Kandidaten)	02:15:17 h	00:05:40 h	00:05:10 h	00:00:01 h
Lfz. (20 Kandidaten)	9 Tage 10:26:39 h	01:32:00 h	09:48:14 h	00:00:03 h

Entschlüsselung	Liste der Sieger nach Schulze-Methode evaluateCondorcet ⇒ Unterscheidung zu Condorcet-Sieger

Tabelle 6.8: Zeitkritische Operationen und Entschlüsselungen für das Schulze-Verfahren

Beschreibung

Der Algorithmus wurde aus der Veröffentlichung von Markus Schulze (siehe [Sch11]) sinngemäß übernommen. Er besteht aus vier Phasen. In jeder sind es mindestens zwei ineinander verschachtelte Schleifen. Im ersten Teil wird initialisiert. Der Graph wie aus der vorangegangen Beschreibung wird durch die Matrix *pathSchulze* repräsentiert. Jede Kante wird hier mit dem Wert der *margins*-Metrik belegt. Kanten von einem Knoten zu sich selber existieren nicht und werden auch in jeder anderen Phase ignoriert. Nach der Initialisierung werden in einer dreifach verschachtelten Schleife die besten Wege gesucht. Dabei werden die Wege in der Variable *pathSchulze* immer wieder überschrieben. Diese enthält nach diesem Abschnitt dann für jedes Kandidatenpaar den Wert des besten Weges in beide Richtungen. Wie hier weiterhin in der Variable *pathSchulze* gearbeitet werden kann und wie es zum gewollten Ergebnis des Schrittes kommt, kann im Paper von Markus Schulze (siehe [Sch11]) nachgelesen werden. In der dritten Phase werden nun alle direkten Duelle in *pathSchulze* neu geprüft. Für ein Duell zwischen cand_a und cand_b wird untersucht, ob der beste Weg von cand_a nach cand_b besser ist als von cand_b nach cand_a, ob es umgekehrt ist oder ob sie den gleichen Wert haben. Zuletzt wird nun für jeden Kandidaten geprüft, wie viele Duelle er verliert. Alle Kandidaten, welche hier keine Niederlage haben, sind nach der Schulze-Methode Sieger.

Performanz

Dieser Algorithmus ist der aufwändigste aller Evaluationen. Er liegt in der Aufwandsklasse $O(n^3)$ und verwendet dazu recht viele *Greater-Than*-Vergleiche. In der Initialisierung sind keine laufzeitkritische Befehle verwendet. Die dreifach verschachtelte Schleife verwendet ein GetMaximum und ein GetMinimum. Diese werden jedoch jeweils nur mit zwei Elementen aufgerufen, weshalb auch nur ein *Greater-Than* pro Funktion verwendet wird. Die Bedingung des Ifs tritt in

$$n \cdot (n-1) \cdot (n-2) = n^3 - 3 \cdot n^2 + 2 \cdot n$$

Fällen ein. In allen anderen Fällen sind zwei der drei Kandidaten dieselben und können übersprungen werden. In der dritten Phase wird wieder mit zwei Schleifen iteriert. Da jedes Duell aber nur ein Mal betrachtet wird, sind nur $\frac{n \cdot (n-1)}{2} = \frac{n^2-n}{2}$ Situationen zu behandeln. Es wird jeweils ein *Greater-Than* und ein *Equals* augerufen. In der letzten Phase gibt es zwar wieder zwei verschachtelte Schleifen, jedoch verwendet die innere nur Additionen. Das *Equals* und *Decrypt* wird pro Kandidaten ein Mal ausgeführt.

Greater-Than: $\underbrace{(n^3 - 3 \cdot n^2 + 2 \cdot n) \cdot 2}_{2.Phase}$[Wähler]+ $\underbrace{\frac{n^2 - n}{2}}_{3.Phase}$[Wähler] $= 2 \cdot n^3 - 6 \cdot n^2 + 4 \cdot n + \frac{n^2}{2} - \frac{n}{2}$[Wähler] $= 2 \cdot n^3 - 5,5 \cdot n^2 + 3,5 \cdot n$[Wähler]
Equals: $\underbrace{\frac{n^2 - n}{2}}_{3.Phase}$[Wähler]+$n$[Kandidaten]
Multiplicate: $\underbrace{((n^3 - 3 \cdot n^2 + 2 \cdot n) \cdot 2) \cdot 2}_{2.Phase} = 4 \cdot n^3 - 12 \cdot n^2 + 8 \cdot n$
Decrypt: n

Diese Methode lässt sich gut optimieren durch Reduzieren der *possibleWinners*. Da hier eine hohe Komplexität bezüglich n vorliegt, können so einige Operationen gespart werden. Weil die Schulze-Methode das Smith-Kriterium erfüllt, kann durch vorausgehendes Berechnen des Smith-Sets die Laufzeit in manchen Fällen deutlich verringert werden. Hierzu sollte die Wahrscheinlichkeitsverteilung über die Smith-Set-Größen untersucht werden. Mit der Berechnung des Smith-Sets müsste man jedoch auf jeden Fall Nachteile im Tally-Hiding hinnehmen, da das Smith-Set dann öffentlich ist. Auch mit mit dem Smith-Set ist nicht garantiert, dass es bedeutend schneller geht, da das Smith-Set zumindest in zufälligen Wahlen nicht selten alle teilnehmenden Kandidaten beinhaltet. Alle Schulze-Evaluationen mit mehr als neun Kandidaten benötigen unter den untersuchten Umständen mehr als einen Tag Berechnungszeit.

Tally-Hiding

Der Algorithmus verrät nichts außer dem gewünschten Ergebnis. Alle Gewichte der Kanten bleiben verschlüsselt. Auch die Auswertung darüber, wer in den abschließenden Duellen gewinnt ist geheim. Erst ganz am Ende wird das erste *Decrypt* verwendet. Es entschlüsselt, ob ein Kandidat mindestens ein Duell verliert. Dies entspricht genau der Information, ob ein Kandidat Sieger nach dem Schulze-Verfahren ist oder nicht.

7 Performanz-Vergleich

Mit diesem Kapitel sollen alle Algorithmen in ihrer Laufzeit miteinander verglichen werden können. Die folgenden Tabellen beinhalten erneut die Informationen zur Laufzeit der unterschiedlichen Algorithmen. Es sind dieselben Daten wie in der Tabelle nach den entsprechenden Algorithmen. Teilweise wurden hier noch Informationen weggelassen, um den Vergleich zu vereinfachen.

Zunächst werden die unterschiedlichen Borda-Evaluationen betrachtet. Die *Aggregierung* wird wie bei Condorcet hier ignoriert, da sie keinen relevanten Einfluss auf die Laufzeit des Systems hat. Bei Borda ist die Punktelimit-Evaluation die performanteste. Sie gibt aber nur aus, welche Kandidaten eine bestimmte Punktzahl erreicht haben. Damit ist sie nicht geeignet einen einzelnen Sieger zu finden, sondern nur für Wahlsysteme, in denen eine Hürde überwältigt werden muss. Die Siegerevaluation ist aber nicht viel langsamer. Beide Algorithmen haben die Komplexität $O(n)$ und die beiden rechenintensiven Funktionen *Greater-Than* und *Equals* werden jeweils nicht häufiger als ein Mal pro Kandidat aufgerufen. Dagegen ist die Positionslimit-Evaluation doch recht aufwändig. Sie liegt in $O(n^2)$. Damit ist sie ähnlich aufwändig wie die einfachen Condorcet-Evaluationen. Bei über zehn Kandidaten benötigt die Evaluation unter den in Abschnitt 3.1 genannten Bedingungen über eine Stunde. Die anderen beiden können in dieser Zeit knapp 50 Kandidaten auswerten.

		Greater-Than	Equals	Multiplicate	Decrypt
Borda Punktelimit	Anzahl	n[Wähler]	0	0	n
	5 Kandidaten	00:04:50 h	0	0	00:00:01 h
	20 Kandidaten	00:19:20 h	0	0	00:00:03 h
Borda Positionslimit	Anzahl	$\frac{n\cdot(n-1)}{2}$[Wähler] n[Kandidaten]	$\frac{n\cdot(n-1)}{2}$[Wähler]	0	n
	5 Kandidaten	00:12:06 h	00:02:19 h	0	00:00:01 h
	20 Kandidaten	03:13:20 h	00:43:53 h	0	00:00:03 h
Borda Siegerevaluation	Anzahl	$n-1$[Wähler]	n[Wähler]	$2\cdot(n-1)$	n
	5 Kandidaten	00:03:52 h	00:01:09 h	00:00:10 h	00:00:01 h
	20 Kandidaten	00:18:22 h	00:04:37 h	00:00:49 h	00:00:03 h

Tabelle 7.1: Performanz-Vergleich aller Borda-Evaluationen

Die Condorcet-Evaluationen sind in der Tabelle 7.2 zusammengefasst. Diese benötigen meist mehr Zeit als die von Borda. Hier liegen alle mindestens in der Aufwandsklasse $O(n^2)$. Die Schulze-Methode liegt sogar in $O(n^3)$. Alle Condorcet-Evaluationen verwenden die Vorberechnung mit $\frac{n^2-n}{2}$ Aufrufen von *Greater-Than* und *Equals*. Es wird hier immer davon ausgegangen, dass *evaluateCondorcet* nicht verwendet wurde, also dabei nicht nach einem Condorcet-Sieger gesucht wurde. Wenn dies doch getan wird, kommen zwar ein *Equals* und *Decrypt* pro Kandidat hinzu, dafür kann aber im Falle eines Condorcet-Siegers sofort abgebrochen und somit einiges an Komplexität erspart werden. Plain Condorcet ohne *evaluateCondorcet* ist keine eigenständige Evaluation und

wird deshalb nicht extra betrachtet. Es ist in allen anderen Condorcet-Methoden berücksichtigt. Für Condorcet Weak und Copeland muss nach der Vorberechnung nicht mehr viel getan werden. Deshalb liegen sie bei der Laufzeit in der Größenordnung von Plain Condorcet. Auch wenn die Laufzeitoptimierung mit *leakMaxCopeland* zwar einen großen Unterschied ausmacht für die reine Copeland-Evaluierung, ist sie aber im Verhältnis zur Gesamtlaufzeit mit der Vorberechnung nicht mehr sehr ausschlaggebend. Bei 20 Kandidaten könnte die Einsparung zehn Minuten betragen, was jedoch nach einer Vorberechnung von circa vier Stunden von geringem Wert ist.
Die Smith-Evaluation ohne Optimierung benötigt ungefähr doppelt so viel Zeit wie Plain Condorcet. Mit der Optimierung braucht die Evaluation bei einem kleinen Smith-Set jedoch nur knapp länger als Plain Condorcet. Wie die Verteilung über die Smith-Set-Größen in Wahlen ist, müsste noch weiter untersucht werden. Grundsätzlich hat das Smith-Set recht häufig die Größe eins, was einem Condorcet-Sieger entspricht. Wenn dieser nicht existiert, umfasst das Smith-Set recht häufig alle oder alle bis auf einen Kandidaten. In diesen Fällen bringt die Optimierung nichts oder sehr wenig.
Minimax liegt auch noch in derselben Komplexitätsklasse. Es hat jedoch deutlich mehr Aufrufe von *Greater-Than*, *Equals* und *Multiplicate*. Damit kommt es zu einer Laufzeit von knapp dem dreifachen der des Plain Condorcets.
Die Schulze-Methode ist deutlich aufwändiger als alle anderen Algorithmen. Schon bei fünf Kandidaten sind es knapp zweieinhalb Stunden Berechnungszeit. Ab zehn Kandidaten ist das Ergebnis schon nicht mehr an einem Tag zu berechnen. Bei 20 sind es über neun Tage.

Die Smith-Evaluation kann auch genutzt werden, um die Laufzeiten anderer Evaluationen zu reduzieren. Da zum Beispiel die Schulze das Smith-Kriterium erfüllt, können durch den Smith-Algorithmus Kandidaten ausgeschlossen werden. Bei Schulze würde eine Reduzierung auf eine Teilmenge der Kandidaten einen sehr großen Unterschied für die Laufzeit ausmachen, da dieser die Komplexitätklasse $O(n^3)$ hat. Jedoch gilt auch für diesen Ansatz, dass das Smith-Set leider recht häufig sehr groß ist. Mit einer Wahrscheinlichkeitsverteilung könnte eine erwartete Laufzeit angegeben werden.

		Greater-Than	Equals	Multiplicate	Decrypt
Plain Condorcet evaluateCondorcet		$\frac{2 \cdot k \cdot n - k^2 - k}{2}$[Wähler]	$\frac{2 \cdot k \cdot n - k^2 - k}{2}$[Wähler] k[Kandidaten]	0	k
	5 Kandidaten, $k = 2$	00:06:46 h	00:02:02 h	0	00:00:00 h
	5 Kandidaten, $k = 5$	00:09:40 h	00:03:22 h	0	00:00:01 h
	20 Kandidaten, $k = 2$	00:35:45 h	00:08:58 h	0	00:00:00 h
	20 Kandidaten, $k = 20$	03:03:36 h	00:48:07 h	0	00:00:03 h
Condorcet Weak	Anzahl (Vorberechnung)	$\frac{n^2-n}{2}$[Wähler]	$\frac{n^2-n}{2}$[Wähler]	0	0
	Anzahl (Weak)	0	n[Kandidaten]	0	n
	5 Kandidaten	00:09:40 h	00:03:22 h	0	00:00:01 h
	20 Kandidaten	03:03:36 h	00:48:07 h	0	00:00:03 h
Copeland-Methode	Anzahl (Plain)	$\frac{n^2-n}{2}$[Wähler]	$\frac{n^2-n}{2}$[Wähler]	0	0
leakMaxCopeland	Anzahl	0	$c \cdot n$[Kandidaten]	0	$c \cdot n$
	5 Kandidaten, $c = 3$	00:09:40 h	00:05:28 h	0	00:00:02 h
	20 Kandidaten, $c = 3$	03:03:36 h	00:56:33 h	0	00:00:08 h
!leakMaxCopeland	Anzahl	$n - 1$[Wähler]	n[Wähler]	$2 \cdot (n - 1)$	n
	5 Kandidaten	00:13:32 h	00:03:28 h	00:00:10 h	00:00:01 h
	20 Kandidaten	03:21:58 h	00:48:31 h	00:00:49 h	00:00:03 h
Minimax-Methode	Anzahl (Plain)	$\frac{n^2-n}{2}$[Wähler]	$\frac{n^2-n}{2}$[Wähler]	0	0
	Anzahl (Minimax)	$n^2 - n - 1$[Wähler]	n[Wähler]	$3 \cdot n^2 - 3 \cdot n - 2$	n
	5 Kandidaten	00:28:01 h	00:03:28 h	00:01:15 h	00:00:01 h
	20 Kandidaten	09:09:51 h	00:48:31 h	00:24:28 h	00:00:03 h
Smith-Methode	Anzahl (Plain)	$\frac{n^2-n}{2}$[Wähler]	$\frac{n^2-n}{2}$[Wähler]	0	0
leakMinCopeland	Anzahl	0	$s \cdot (n + 1)$[Kandidaten] s[Wähler]	$s \cdot 2$	$s + n$
	5 Kandidaten, $s = 4$	00:09:40 h	00:08:18 h	00:00:10 h	00:00:01 h
	5 Kandidaten, $s = 7$	00:09:40 h	00:12:47 h	00:00:18 h	00:00:02 h
	20 Kandidaten, $s = 4$	03:03:36 h	01:02:32 h	00:00:10 h	00:00:03 h
	20 Kandidaten, $s = 37$	03:03:36 h	03:36:23 h	00:01:35 h	00:00:08 h
!leakMinCopeland	Anzahl	0	$2 \cdot (n^2 - 1)$[Kandidaten] $2 \cdot (n - 1)$[Wähler]	$8 \cdot (n - 1)$	n
	5 Kandidaten	00:09:40 h	00:14:17 h	00:00:41 h	00:00:01 h
	20 Kandidaten	03:03:36 h	03:41:03 h	00:03:16 h	00:00:03 h
Schulze-Methode	Anzahl (Plain)	$\frac{n^2-n}{2}$[Wähler]	$\frac{n^2-n}{2}$[Wähler]	0	0
	Anzahl (Schulze)	$2 \cdot n^3 - 5,5 \cdot n^2 + 3,5 \cdot n$ [Wähler]	$\frac{n^2-n}{2}$[Wähler] n[Kandidaten]	$4 \cdot n^3 - 12 \cdot n^2 + 8 \cdot n$	n
	5 Kandidaten	02:15:17 h	00:05:40 h	00:05:10 h	00:00:01 h
	20 Kandidaten	9 Tage 10:26:39 h	01:32:00 h	09:48:14 h	00:00:03 h

Tabelle 7.2: Performanz-Vergleich aller Condorcet-Evaluationen

8 Tally-Hiding Vergleich

Wie in den Grundlagen angesprochen, gehen wir von einigen Sicherheitsvorraussetzungen aus. Es wird vorausgesetzt, dass im System ausreichend vertrauenswürdige Komponenten agieren und allein das Aufrufen der *Decrypt*-Funktion Daten veröffentlicht. Die genaue Untersuchung über Zuverlässigkeit und Sicherheit der Komponenten findet sich in anderen Veröffentlichungen zu Ordinos. Da es die Aufgabe dieser Arbeit war, Algorithmen für Wahlverfahren umzusetzen, werden auch nur diese nach deren Tally-Hiding untersucht. Die Algorithmen müssen Daten veröffentlichen, sollten jedoch nur so viel verraten, wie von ihnen gefordert. Im Rahmen von Condorcet wird dies jedoch nicht immer eingehalten. Wie es zu den veröffentlichten Informationen, kommt ist immer mit dem Algorithmus beschrieben worden. Nun werden alle Algorithmen verglichen und betrachtet, ob sie das gewünschte Tally-Hiding umsetzen können.

	Entschlüsselung
Borda Aggregierung	—
Borda Punktelimit	∀(Kandidat): Gesamtpunkte[Kandidat]≥Punktelimit
Borda Positionslimit	∀(Kandidat): Position[Kandidat]≤Positionslimit
Borda Nur Sieger	∀(Kandidat): Gesamtpunkte[Kandidat]=Max(Gesamtpunkte)
Condorcet Aggregierung	—
Condorcet Plain Condorcet	evaluateCondorcet ⇒ Condorcet-Sieger; leere Liste wenn er nicht existiert !evaluateCondorcet ⇒ —
Condorcet Weak Condorcet	Liste der Weak-Condorcet-Sieger evaluateCondorcet ⇒ Unterscheidung zu Condorcet-Sieger
Condorcet Copeland-Methode	Liste der Sieger nach Copeland-Methode leakMaxCopeland ⇒ Max(Copeland-Punkte) evaluateCondorcet ⇒ Unterscheidung zu Condorcet-Sieger
Condorcet Minimax	Liste der Minimax-Sieger evaluateCondorcet ⇒ Unterscheidung zu Condorcet-Sieger
Condorcet Smith	Smith-Set leakMinCopeland ⇒ minimale Copeland-Punkten innerhalb dieses evaluateCondorcet ⇒ Unterscheidung zu Condorcet-Sieger
Condorcet Schulze	Liste der Sieger nach Schulze-Methode evaluateCondorcet ⇒ Unterscheidung zu Condorcet-Sieger

Tabelle 8.1: Tally-Hiding-Vergleich aller Algorithmen

Die Algorithmen für Borda verraten genau die Information, die als Wahlergebnis angefragt war. Es werden die Kandidaten in einer Liste zurückgegeben, welche die Bedingungen erfüllen. Bei Condorcet trifft das nicht durchgehend zu. Wenn eine performantere Variante gewählt wurde, den Condorcet-Sieger zu evaluieren, ist dieser von Siegern aus der jeweiligen Evaluation zu unterscheiden. Man erfährt somit, ob der Sieger beispielsweise nur die meisten Copeland-Punkte hatte oder sogar Condorcet-Sieger war. Diese Information sagt etwas über die Deutlichkeit des Sieges aus. Angenommen in einer Wahl gäbe es zwei Favoriten. Wenn der Sieger nachher ein Condorcet-Sieger ist, so ist auch klar, dass er im direkten Duell gegen seinen Konkurrenten gewonnen hat. Das kann durchaus interessant sein, im Rahmen von Ordinos sollen aber zusätzliche Information vermieden werden. Wenn *evaluateCondorcet* deaktiviert ist, ist diese Unterscheidung nicht mehr möglich.

Der Copeland-Algorithmus verrät mit der Variante *leakMaxCopeland* noch mehr Informationen. Diese ist zwar die performantere der beiden, verrät jedoch, wie viele Copeland-Punkte der Sieger hat. Dadurch werden wieder Informationen über die Deutlichkeit des Sieges bekannt. Hat der Sieger $2 \cdot n - 3$ Punkte, so hat dieser nur ein einziges Unentschieden und gewinnt die restlichen Duelle. Bei weniger Punkten hat er mindestens zwei Unentschieden oder eine Niederlage. Ähnlich verhält es sich bei der Smith-Evaluation bei *leakMinCopeland*. Hier wird die kleinste Copeland-Punktzahl eines Kandidaten im Smith-Set verraten. Es ist nicht bekannt welcher Kandidat es ist, daher ist die Information meist von geringem Wert. Es folgen nun ein paar Fallbeispiele. Ist nur ein einziger Kandidat im Smith-Set, ist dieser Condorcet-Sieger und hat in jedem Fall $2 \cdot n - 2$ Copeland-Punkte, womit die minimale Copeland-Punktzahl offensichtlich ist. Wenn es einen Kandidaten gibt, der alle Duelle verliert, wird dies nicht durch diese Schwachstelle öffentlich. Er kann nicht Teil des Smith-Sets sein und nur über dessen schlechtestes Mitglied verrät *leakMinCopeland* etwas. Die Aussagen, die *leakMinCopeland* über das Wahlergebnis hinaus machen kann, liegen zwischen:

- Im Smith-Set gibt es einen Kandidaten, welcher alle Duelle innerhalb dieses verliert bis auf ein Unentschieden
- Im Smith-Set ist es sehr ausgeglichenen. Sie haben alle dieselbe Copeland-Punktzahl

Für alle Algorithmen gibt es aber eine Variante, welche keine zusätzlichen Informationen preisgibt.

Man muss aber bedenken, dass alle Verfahren bis auf Plain Condorcet beliebig viele Sieger hervorbringen können. Das kann unter Umständen zu mehr Informationen führen als gewollt. Bei sehr ausgeglichenen Ergebnissen ist es möglich, dass nur ein einziger Kandidat nicht zu den Siegern gehört. Somit wäre der Verlierer der Wahl bekannt. In der Realität ist der Fall oft unwahrscheinlich. Falls er jedoch eintritt, kann das weitere Vorgehen sehr unterschiedlich sein. Es wäre in Ordinos umsetzbar, dass das Ergebnis nur veröffentlicht wird, wenn eine vorher definierte Siegerzahl nicht überschritten wird.

9 Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit wurden neue Algorithmen zur Wahlevaluation zur existierenden Implementierung des Systems Ordinos hinzugefügt. Außerdem wurde die Struktur der Software angepasst, um das Hinzufügen weiterer Wahlsysteme zu erleichtern und um einer realitätsnäheren Software zu entsprechen.

Die neuen Algorithmen zur Wahlevaluation setzen unterschiedliche Varianten der Wahlverfahren Borda und Condorcet um. Borda ist mit den Varianten der Stimmzettel und Punkteverteilungen einigermaßen vollständig umgesetzt, es können jedoch noch Varianten zum Tally-Hiding ergänzt werden. Im Kapitel Abschnitt 5.1 wurden Möglichkeiten angesprochen, welche Ergebnisse aus einem Borda- Verfahren möglich wären. Umgesetzt wurden hier die Suche nach den besten Kandidaten in beliebiger Anzahl und die Evaluation mit einem Punktelimit. Borda wird im Sport, im Alltag und beim ESC verwendet. Sportwertungen können gut für Testfälle genutzt werden, die Verwendung von Ordinos ist hier jedoch unwahrscheinlich in der Realität. Ein Grundkonzept für den gesamten Sport sind Zuschauer. Deshalb ist jeder einzelne Wettkampf öffentlich und das Gesamtergebnis kann von jedem Zuschauer berechnet werden. Sport lebt von Transparenz und nicht von Geheimnissen. Die Verwendung im Alltag verhält sich oft ähnlich zum Sport und hat meist die Schwierigkeit, dass der Aufwand das Ordinos-System zu verwenden zu groß ist. Da der Alltag jedoch sehr vielfältig ist, gibt es jedoch auch Fälle, in denen die Verwendung Sinn ergeben würde. Hierzu gehört zum Beispiel die Klassensprecherwahl. Die Klassensprecherwahl ist in vielen Fällen eine Variante von Borda. Schulen wären im Stande solch ein System aufzusetzen. Außerdem kann hier das Tally-Hiding eine wichtige Rolle spielen, da Peinlichkeiten und Mobbing unter Schülern relevante Themen sind. Konkret für den ESC will man vermutlich nicht direkt das Wahlverfahren ändern und Daten verheimlichen durch eine Verwendung von Ordinos. Es wäre aber durchaus möglich und könnte diskutiert werden, falls der Wunsch bestünde Tally-Hiding beim ESC zu verwenden. Dazu müssten aber mindestens Teile der Stimmabgaben auch geheim bleiben. Borda ist sicher noch in vielen anderen Wahlverfahren in Verwendung. Hier muss für jedes System unabhängig entschieden werden, ob eine Umsetzung mit Ordinos sinnvoll wäre.

Bei Condorcet gibt es im Gegensatz zu Borda ganz unterschiedliche Ansätze der Evaluation. Hier ist die *Aggregierung* nicht intuitiv auszuwerten. Deshalb ist jede Condorcet-Methode ein eigenes Wahlverfahren. Sie basieren nur auf denselben Daten und folgen alle dem Condorcet-Kriterium. Mit Condorcet wurden vor allem folgende drei vollständige Wahlsysteme umgesetzt: Copeland-Verfahren, Minimax und Schulze-Methode. Bis auf Ranked-Pairs wurden damit die bekanntesten Condorcet-Methoden umgesetzt. Diese Verfahren liefern immer ein Ergebnis und sind darauf ausgelegt möglichst nur einen Sieger zu finden. Die Schulze-Methode ist ein Verfahren, welches in mehreren Organisationen in der Realität verwendet wird. In diesen könnte auch die Verwendung von einem System mit Tally-Hiding interessant sein. Die Laufzeit ist hier aber im Vergleich zu allen anderen Verfahren deutlich schlechter und bei vielen Kandidaten kaum akzeptabel. An dieser Stelle könnte ein besserer Algorithmus gesucht werden, welcher für Ordinos optimiert ist. Es wurde bis jetzt nur der vorgeschlagene Algorithmus von Markus Schulze verwendet und das Thema nicht noch

weiter untersucht. Eine andere Hardwareumgebung oder Verschlüsselung könnten auch zu ganz anderen Laufzeiten führen. Es lohnt sich die Schulze-Methode weiter zu untersuchen und mit einer realistischen Implementierung könnte diese Evaluation zum Aushängeschild von Ordinos werden. Minimax und Copeland werden auch verwendet, haben aber keine sehr prominenten Einsätze. Hier hängt es von der konkreten Verwendung ab, ob Ordinos eine Rolle spielen könnte. Für alle Verfahren wird aktuell nur der Sieger bestimmt. Hier könnte man sich weiter überlegen, welche Informationen über ein Wahlergebnis interessant sein könnten. Vermutlich ist jedoch in den meisten Verwendungen der Sieger ausreichend.

Die Evaluationen Plain Condorcet, Weak-Condorcet und Smith sind Sonderfälle. Plain Condorcet ermöglicht ein schnelles Evaluieren des Condorcet-Siegers. Außerdem wird es für jede Condorcet-Methode als Vorberechnung benötigt. Es kann aber auch eigenständig verwendet werden, liefert dann aber nicht immer ein Ergebnis. Weak-Condorcet ist eine kleine Erweiterung und kommt manchmal zu einem Ergebnis, auch wenn Plain Condorcet keines hätte. Aber auch dieses kann ohne einen Sieger terminieren. Smith hat zwar immer eine Liste von Kandidaten als Ergebnis, welche aber immer mehr als einen Kandidaten enthält, außer dieser ist Condorcet-Sieger. Es eignet sich nicht dazu, einen einzelnen Sieger einer Wahl zu finden. Vielmehr wird es dazu verwendet Kandidaten auszuschließen und das Wahlverfahren an das Smith-Kriterium anzupassen. Wenn ein Verfahren schon das Smith-Kriterium erfüllt, kann mit dieser Methode eventuell die Laufzeit verbessert werden. Besonders im Zusammenhang mit Schulze könnte das zum Beispiel interessant sein.

Die Optimierungen sollten alle noch auf ihre Wirksamkeit untersucht werden. Das heißt über Wahrscheinlichkeiten sollte der Erwartungswert für die Laufzeit angegeben werden können. Dazu gehören die Untersuchungen der Smith-Set-Größe, Wahrscheinlichkeit eines Condorcet-Siegers und Verteilungen von Copeland-Punktzahlen bei zufälligen Wahlen. Bei der Optimierungen für Condorcet mit Borda-Punkten muss man sich noch Gedanken über Ausnahmefallbehandlungen machen oder nur bestimmte Wahlzettel erlauben. Generell kann man sich aber überlegen, ob die *leak-Borda*-Optimierungen wirklich den Aufwand der Implementierung wert ist. Die Implementierung ist zwar schon vorhanden, aber nicht optimal und macht den Code unschöner.

Neben Borda und Condorcet wurden auch andere Änderungen an der Software vorgenommen. Diese sollen das Implementieren weiterer Wahlverfahren vereinfachen, den Code leserlicher machen und an eine zukünftige realistische Umgebung anpassen. Auch die Untersuchung der Laufzeiten der aktuellen Implementierung wurde vereinfacht und genutzt. Das Hinzufügen weiterer Verfahren soll durch den generischen Stimmzettel und durch Komponenten wie in Abschnitt 4.2 angesprochen erleichtert werden. Die neuen Komponenten entsprechen in ihrer Architektur nun mehr einem realistischen System. Hier muss aber noch einiges getan werden, um alle Teilnehmer einer Wahl auf getrennten Geräten simulieren zu können. Dazu gehört zum Beispiel die Kommunikation der Wahlkonfiguration zu Beginn des Verfahrens, sodass alle Komponenten wissen, wie sie in diesem Wahlverfahren zu arbeiten haben.

Im Endeffekt hat sich das System mit dieser Arbeit ein Stück der Realität angenähert und unterstützt zusätzliche Wahlverfahren. Die Wahlverfahren sind mit ihren Varianten auf ihre Performanz und das Tally-Hiding untersucht. Mit den Wahrscheinlichkeiten können diese noch weiter wissenschaftlich untersucht werden und die Architektur muss noch weiter angepasst werden, um einen Platz in der Realität zu finden.

Literaturverzeichnis

[05] *Condorcet - Paradoxon - Verfahren*. 12.08.2005. URL: https://www.wahlrecht.de/lexikon/condorcet.html (zitiert auf S. 49, 57).

[06] *Borda Verfahren*. 9/5/2006. URL: https://www.wahlrecht.de/lexikon/borda.html (zitiert auf S. 38, 50).

[11] *Wahlen und deren Wahlsysteme » Condorcet-Methode*. 10.12.2011. URL: http://www.electionmethods.org/wahlverfahren/condorcet-methode.htm (zitiert auf S. 30).

[15] *Bewertungswahl*. 30.04.2015. URL: https://deacademic.com/dic.nsf/dewiki/2275027 (zitiert auf S. 30).

[20a] *Internet Voting*. 1.10.2020. URL: https://www.ndi.org/e-voting-guide/internet-voting (zitiert auf S. 13).

[20b] *Ski Alpin Weltcuppunkte - Ski Weltcup Punktevergabe - Punktevergabe im alpinen Weltcup*. 13.07.2020. URL: https://www.sportlexikon.com/ski-alpin-weltcuppunkte (zitiert auf S. 37).

[Adl17] A. Adler. *Liquid Democracy in Deutschland: Dissertation*. Bd. Band 59. Edition Politik. transcript Verlag, 2017. ISBN: 9783839442609. DOI: 10.14361/9783839442609. URL: https://books.google.de/books?id=wK9jDwAAQBAJ (zitiert auf S. 49, 70).

[ES20] M. Eisermann, F. Stoll. *Blatt14: Die Qual der Wahl des Wahlverfahrens*. 2020. URL: https://pnp.mathematik.uni-stuttgart.de/igt/eiserm/lehre/2019/Spieltheorie/Uebungen/blatt14.pdf (zitiert auf S. 37).

[FG76] P. C. Fishburn, W. V. Gehrlein. „Borda's rule, positional voting, and Condorcet's simple majority principle: Public Choice, 28(1), 79-88“. In: *Public Choice* 28.1 (1976), S. 79–88. ISSN: 1573-7101. DOI: 10.1007/BF01718459 (zitiert auf S. 58).

[KAAR14] H. K. Kim, S. I. Ao, M. A. Amouzegar, B. B. Rieger. *IAENG Transactions on Engineering Technologies: Special Issue of the World Congress on Engineering and Computer Science 2012*. Bd. 247. Lecture Notes in Electrical Engineering. Dordrecht und s.l.: Springer Netherlands, 2014. ISBN: 9789400768185. DOI: 10.1007/978-94-007-6818-5. URL: https://books.google.de/books?id=3cXBAAAAQBAJ (zitiert auf S. 61, 63).

[KLM+19] R. Küsters, J. Liedtke, J. Müller, D. Rausch, A. Vogt. „Ordinos: A Verifiable Tally-Hiding Remote E-Voting System (Full Version)“. In: 2019 (zitiert auf S. 14, 15).

[Sch11] M. Schulze. „A new monotonic, clone-independent, reversal symmetric, and condorcet-consistent single-winner election method“. In: *Social Choice and Welfare* 36.2 (2011), S. 267–303. ISSN: 0176-1714. DOI: 10.1007/s00355-010-0475-4 (zitiert auf S. 64, 70, 71, 73).

[Str80] P. D. Straffin. *Topics in the Theory of Voting*. UMAP expository monograph series. ERIC Clearinghouse, 1980. ISBN: 9783764330170. URL: https://books.google.de/books?id=tNXuAAAAMAAJ (zitiert auf S. 64).

Alle URLs wurden zuletzt am 05. 10. 2020 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift