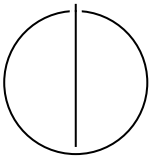
SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Exploring the Resolution of Delegations in Liquid Democracy with Fractional Delegation

David Nicolaus Matthäus Holzwarth



SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

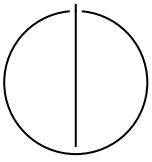
TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Exploring the Resolution of Delegations in Liquid Democracy with Fractional Delegation

Eine Untersuchung der Stimmgewichtberechnung in Liquid Democracy mit fraktionalem Delegieren

Author:	David Nicolaus Matthäus Holzwarth
Examiner:	Prof. Dr. Pramod Bhatotia
Supervisor:	Prof. Dr. Bryan Alexander Ford
Submission Date:	05 August 2025



I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, 05 August 2025 David Nicolaus Matthäus Holzwarth

Abstract

This thesis explores fractional delegation in Liquid Democracy, where voters can split their vote among multiple delegates, aiming to reduce vote concentration and improve representational fairness. We formalize the mode and a method of resolving the final voting power of each participant. We then present and evaluate three implementations of this method: using a solver for systems of linear equations, a linear programming solver, and an iterative implementation. We prove that, under well-formed conditions, delegation graphs have a unique, power-conserving solution. A preprocessing pipeline ensures graphs are resolvable by eliminating problematic cycles. Through evaluation on synthetic, social, and real-world graphs, we find that the solver for systems of linear equations is fastest in most cases, while the iterative implementation struggles with certain graphs. Our findings demonstrate that fractional delegation is both feasible and scalable, paving the way for more expressive digital democratic systems.

Contents

1. Introduction

In democratic systems, the balance between direct participation and practical representation remains an ongoing challenge. On the one hand, direct democracy ensures that voters retain full control over political outcomes. On the other hand, representative systems introduce intermediaries who can make informed decisions on behalf of voters, addressing scalability and engagement issues. In recent years, Liquid Democracy has emerged as a compelling hybrid model, allowing voters to either vote directly or delegate their vote to another agent, who may in turn delegate again. This approach has the potential to increase participation while maintaining flexibility and accountability.

However, while the flexibility of Liquid Democracy is attractive, its implementation introduces several theoretical and technical difficulties. Chief among these are the concentration of voting power in few highly trusted individuals, the presence of cycles in delegation graphs which can trap votes, and the computational burden of resolving such graphs, meaning to determine who ends up with how many votes. These challenges become more pronounced as the number of voters and complexity of delegation relationships grow.

Motivated by this problem, this thesis introduces a variant of Liquid Democracy with fractional delegation, allowing voters to split their vote among multiple delegates. This approach captures the diversity and redundancy of trust in real communities, increases resilience to vote loss, and reduces the risks of vote concentration. However, this generalization complicates the task of computing the final distribution of voting power, especially in the presence of cycles.

We formalize this model of fractional Liquid Democracy and propose three different methods for resolving delegation graphs: using a solver for systems of linear equations, using a linear programming formulation, and simulating the delegation process with an iterative algorithm. We also introduce preprocessing techniques to handle ill-formed graphs and turn arbitrary graphs into delegation graphs.

The core contribution of this thesis is that delegation graphs in a fractional Liquid Democracy model can be resolved efficiently and fairly using methods based on systems of linear equations, while maintaining conservation of voting power and tolerating cyclic delegations through preprocessing.

We evaluate these approaches through benchmarks on synthetic, social, and real-world graphs. We show that in many cases the solver for systems of linear equations

beats the other two implementations, but that there are exceptions, such as very large graphs.

This thesis contributes: (1) a formal definition of fractional delegation in Liquid Democracy, (2) three distinct and implementable resolution algorithms, (3) a preprocessing method for handling delegation cycles, and (4) a benchmark analysis across a variety of graph classes. These insights contribute to building scalable, fair, and expressive voting systems for digital democracy.

The rest of the paper is structured as follows: Chapter 2 provides necessary background on Liquid Democracy and fractional delegation. Chapter 3 formalizes the problem and our design choices. Chapter 4 discusses the implementation of the proposed algorithms. Chapter 5 evaluates their performance across diverse delegation graphs. Chapter 6 reviews related literature, and Chapter 7 concludes with key insights and future work.

2. Background

2.1. Liquid Democracy

Liquid Democracy is a voting system that blends aspects of direct and representative democracy. Although there is no universally accepted definition, Liquid Democracy generally allows agents to either cast their votes directly or delegate them to a proxy who votes on their behalf. Most formulations of Liquid Democracy also support transitive delegation: an agent who receives delegated votes may, in turn, delegate them further, creating chains of delegations. [10, 5, 26, 4]

2.1.1. Motivation

Liquid democracy is generally motivated by two core shortcomings of traditional democratic systems: the low participation often observed in direct democracy, and the limited accountability that is characteristic of representative democracy [11, 6]. While democracy depends on active participation, direct voting is not always feasible or convenient for individual voters due to constraints such as lack of information, interest, or time. Liquid democracy addresses this by allowing voters to delegate their vote to a trusted proxy, thereby enabling indirect yet meaningful participation. Conversely, representative democracies frequently suffer from diminished accountability, particularly when representatives are elected for long, fixed terms and drawn from a small pool. Liquid democracy mitigates this by opening the pool of potential representatives to all voters. Furthermore, it allows delegations to be updated or revoked at any time, restoring agency to the voter [6]. In doing so, liquid democracy emerges as a promising hybrid model, offering a flexible middle ground between participatory engagement and representational practicality.

2.1.2. Challenges

While liquid democracy offers a promising balance between direct and representative models, it also introduces several challenges. First, it is inherently more complex than traditional voting mechanisms. Voters must understand not only how to vote or delegate, but also the implications of transitive delegation. This added complexity

may deter participation, especially among less politically engaged people. Second, empirical studies, most notably within the German Pirate Party, have highlighted a recurring problem of vote concentration, where a small number of highly visible or trusted individuals accumulate disproportionate amounts of voting power. This phenomenon will be introduced in detail in section 2.2.1. Finally, from a computational standpoint, resolving the outcome of a delegative vote is no longer a matter of just counting ballots. Instead, the resolution process involves traversing potentially large and cyclic delegation graphs. These technical hurdles necessitate robust infrastructure and may raise questions about transparency, efficiency, and verifiability in large-scale deployments.

2.1.3. Applications

One of the most prominent real-world applications of Liquid Democracy was in the German Pirate Party, where members participated in decision-making through a Liquid Democracy platform between 2010 and 2015. Throughout the period 2010 - 2013, 499,009 votes on 6,517 topics were cast, with pirate party members having made 14,964 delegations. [16] Other case studies of Liquid Democracy include the Student Union of the Faculty of Information Studies in Novo Mesto, ProposteAmbrosoli2013, a pilot used in regional election in the Lombardi Region of Italy, Google Votes - a proposal dissemination feature used within Google's internal corporate network - and the Partido de la Red - an Argentinian political party. [24]

2.2. Fractional Delegation

The subject of this paper is an implementation of liquid democracy, in which agents do not need to choose only one proxy to delegate their vote to. They can delegate fractions of their vote to multiple other agents. We call this feature **fractional delegation**.

2.2.1. Motivation

Allowing fractional delegation is motivated by the observation that implementations of classic Liquid Democracy, where each agent may delegate their vote to only one other person, suffer from a well-documented tendency for voting power to concentrate in the hands of a few individuals, or, in some cases, even a single person [16, 8, 1]. When the German Pirate Party used Liquid democracy to allow their members to vote on the party's goals, Martin Haase, a linguistics professor, gained such a large backing, that his vote was "like a decree"; he was able decide the outcome of votes practically alone. [2] This concentration undermines the democratic ideal, effectively creating an

oligarchic structure in which a small group of powerful individuals can determine voting outcomes with little accountability to their delegators. Such a system is not only less representative but also more vulnerable to corruption or manipulation, as influencing a few powerful delegates may be easier and cheaper than persuading a broad and diverse electorate. Moreover, if a powerful agent fails to participate in a decision, a large number of voters find themselves voiceless in the outcome.

A further shortcoming of classic liquid democracy is that agents are forced to either vote themselves or delegate their one vote to exactly one person. Even if agents don't end up using the option of delegating to multiple people, we still believe it to be a valuable feature, as it better reflects the nuanced trust relationships present in real-world communities. In many cases, agents may trust several individuals to represent different aspects of their interests or to provide redundancy. By allowing fractional delegation, this diversity of trust is better captured, leading to a more resilient and representative aggregation of preferences.

Finally, Liquid Democracy faces the challenge of cyclic delegation. When one participant, say A, delegates their vote to another, B, and B in turn delegates it back to A, the vote becomes trapped in a cycle and is effectively lost. [3] Allowing fractional delegations mitigates this problem: if either A or B had delegated a portion of their vote to a third party, that fraction could eventually reach someone who casts a vote, thus reducing the number of votes lost within the system.

2.2.2. Existing Methods to Deal with Vote Concentration and Cyclic Delegation

The problem of vote concentration has been addressed in literature. Partly in response to this problem, Boldi et al. propose Viscous Democracy, which introduces a damping factor into the delegation process: the further a vote is delegated, the weaker it becomes. Viscous Democracy offers a promising way to prevent excessive concentration of power and reflects the intuition that trust diminishes as a vote moves further from its original source. However, it also conflicts with the democratic principle that every vote should carry equal weight. [5]

Gölz et al. and Kotsialou & Riley take a different approach. They both propose allowing agents to nominate multiple potential delegates. An algorithm then selects the most suitable delegate for each agent, aiming to minimize power concentration and avoid delegation cycles. Even in this approach, however, each delegator ultimately entrusts their vote to only one delegate. Kotsialou & Riley's algorithm also tries to minimize lost power through cyclic delegations. [17, 14]

3. Design

This section describes our implementation of liquid democracy with fractional delegation. We start by introducing the problem, then introducing prerequisite definitions and finally the method to resolve delegations.

3.1. Problem Statement

We consider a fractional delegation model, where voters may distribute their vote across multiple delegates. Each voter can either retain their full vote or delegate it to others in fractional amounts summing to one. The voter's final power must be zero if they delegate, and equal to the proportion of votes delegated to them in addition to their own initial vote if they don't. All votes must be conserved, unless they are lost due to cyclic delegation.

More precisely, we pose the following problem.

1. **Given** a set of voters and their delegations, where each voter has one initial vote, and either:
   a) votes directly
   b) delegates their vote in its entirety
2. **We want to find** the final voting power of each voter ("resolve the delegation graph") such that:
   a) A voter's final voting power is zero if they choose to delegate and not vote directly.
   b) A voter's final voting power is equal to the amount of votes delegated to them, including their own initial vote and transitive delegations, otherwise.
   c) Power is conserved. The sum of the final power of all nodes must be equal to the amount of votes initially in the graph. This does not include power that is stuck as a result of cyclic delegation. What exactly this entails will be detailed in section 3.2.3

3.2. Implementing Liquid Democracy with Fractional Delegation

3.2.1. Definitions

Delegation graphs represent **delegations** between voters using weighted, directed edges between nodes. **Power** refers to a fractional amount of votes. Each node v initially has one vote, or an **initial power** $p_v^{(0)} = 1$. Each delegation between two nodes has a **weight** $w \in [0, 1]$[1]. **Resolving delegations** means to determine how much power each node holds according to the delegations. After resolving delegations, a node v's **final power** is p_v. A more rigid definition of a nodes final power will be introduced in section 3.3.

As per the problem statement, voters are strictly given the choice to either delegate their vote (fractionally) in its entirety, or vote directly. The electorate is thus divided into two disjoint sets, **sinks** S, who actually vote, and **delegators** D.

We thus define a **delegation graph** as a finite, directed, weighted graph $G = (V, E)$, with sinks S and delegators D as follows:

1. $V = S \dot{\bigcup} D$, meaning that V is the union of the two *disjoint* sets of sinks and delegators.
2. Each edge $e \in E$ is a triple (u, v, w) denoting a delegation from node u to node v of weight w.
3. Each sink $s \in S$ has no outgoing edges.
4. Each delegator $d \in D$ has $n \in \mathbb{N}$ outgoing edges, each with a positive weight, such that the sum of all of its outgoing edge weights equals 1.

3.2.2. Conservation of Power

A vital property we set for the delegation graph in the problem statement is the conservation of power. While some authors have experimented with implementations of liquid democracy where this is not the case, we believe that for a system to be truly democratic, we must assert delegating is not penalized, so a vote cast by a sink should not be different in value to a vote cast by a sink through delegation from a delegator. [4, 5] Thus, any implementation should have mechanism to ensure that the sum of the

[1] Edge weights should generally not be zero, since that signifies the absence of a delegation, hence the absence of an edge. However, this does not need to be enforced, since edges with a weight of zero don't break the method.

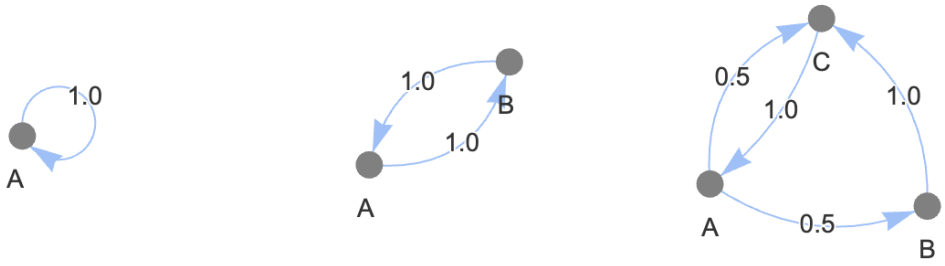


Figure 3.1.: Closed delegation cycles

final power of all sinks is equal to the sum of the initial power of all nodes. We touch upon how this requirement can be implemented in section 3.3.

3.2.3. Closed Delegation Cycles

We define a **closed delegation cycle** $C \subseteq V$ in a delegation graph $G = (S \dot{\bigcup} D, E)$ as a cycle in G such that for every node $v \in C$, there exists no path from v to any sink node in S.

Figure 3.1 shows exemplary closed delegation cycles. These cycles lead to contradictory situations, as power delegated within never reaches a sink. Some works discuss ways to handle power stuck in such cycles or mitigate the risk of such cycles appearing, but effectively it is lost [3, 6]. This means that none of the nodes in a closed cycle will vote, which is in line with the will of voters, who all wish to not vote themselves, instead delegate their power, letting their delegate(s) decide what to do with this power.

In practice, such cycles need to be addressed before resolving delegations in a preprocessing step, since our method of resolving delegation graphs introduced in section 3.3 is not equipped to handle such cycles by itself. Our approach to this is to find all such cycles, and collapse them into an additional sink node in the graph. Any delegation toward nodes within the cycle is redirected into this added sink, thus ensuring the graph no longer has any closed delegation cycles.

This is true since, before the preprocessing step, any power delegated by a node not affected by a closed delegation cycle, meaning it is neither part of the delegation cycle nor delegates its votes such that it ends up in a delegation cycle, had a way to a sink. After the preprocessing step, even power delegated by nodes that are affected by a closed delegation cycle is either removed from the graph since its node got collapsed, or has a way to a sink, namely the specially added sink.

The algorithm for this preprocessing step so can be found in the appendix in sec-

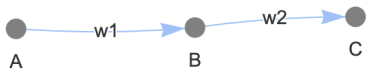


Figure 3.2.: Sample delegations

tion A.1.3

We prove below, that given the absence of such closed delegation cycles, delegations are resolvable in a delegation graph.

Theorem 1. *Let $G = (S \dot{\bigcup} D, E)$ be a delegation graph. If G contains no closed delegation cycles, then for every delegator $d \in D$, there exists a path from d to a sink node $s \in S$.*

Proof. Suppose, for contradiction, that G contains no closed delegation cycle, but there exists a $d \in D$ such that no path from d leads to any sink $s \in S$. Since G is a finite graph, any walk from d must eventually repeat nodes, implying a cycle. If at least one node in this cycle can reach a sink, there would be a path for all others in the cycle to reach a sink via this node as well, thus all nodes in the cycle can not reach a sink either. Thus, G does contain a closed delegation cycle. $\lightning$ □

We define a **well-formed delegation graph** as a delegation graph, which contains no closed delegation cycles. Note, that while a self loop, so a delegates to oneself, of weight one is trivially **not** allowed in a well-formed delegation graph, a self loop of weight $w < 1$ is permitted as long as the rest of the node's power eventually flows to a sink. Since a delegator cannot vote themselves, any power it delegates to itself will "flow" back into the node, and then be redistributed to its delegates.

3.3. Resolving Delegations

We will use the sample delegation chain in fig. 3.2 to create an intuition on how we will resolve delegation graphs. Sink node C receives its own initial vote, and is also delegated a fraction w_2 of B's vote. Node B, in turn, receives its own vote and a fraction w_1 of A's vote. Let p'_A and p'_B denote the **standing power** of nodes A and B, i.e. the total amount of power they have, including their own initial vote and any power delegated to them. Then, the final power of C, assuming no other incoming delegations, is:

$$\begin{aligned} p_C &= 1 + w_2 p'_B \\ &= 1 + w_2(1 + w_1 p'_A) \\ &= 1 + w_2(1 + w_1 \cdot 1) \end{aligned}$$

This motivates the recursive definition of standing power in a delegation graph $G = (V, E)$ as:

$$p'_v = 1 + \sum_{(u,v,w) \in E} w p'_u$$

Using this definition, and in line with the problem statement that delegating nodes must not retain any power, we define the final voting power p_v of a node as:

$$p_v = \begin{cases} p'_v & \text{if } v \in S \\ 0 & \text{if } v \in D \end{cases}$$

The problem of finding each node's standing power is thus a problem of solving a system of linear equations, namely calculating the standing power for all nodes. We prove below, that given a well-formed delegation graph, this method returns a unique solution, in which power is also conserved.

3.3.1. Existence of a Unique Solution

We first introduce notation, which will be used within the proof. The system of linear equations $p'_v = 1 + \sum_{(u,v,w) \in E} w p'_u$ can also be rearranged to be in matrix form:

$$\begin{aligned} p' = \mathbb{1} + W p' &\implies p' - W p' = \mathbb{1} \\ &\implies (I - W) p' = \mathbb{1} \end{aligned}$$

where $p' \in \mathbb{R}_+^{|V|}$ is the vector of standing power values for each node, $W \in [0,1]^{|V| \times |V|}$ is the adjacency matrix of the graph (with W_{ij} denoting the weight of the edge from node j to node i), and $\mathbb{1}$ is the all-ones vector.

Theorem 2. *Given a well-formed delegation graph $G = (V, E)$, the equation $p'_v = 1 + \sum_{(u,v,w) \in E} w p'_u$ has a unique solution for all $v \in V$.*

Proof. We prove the theorem by modifying the delegation graph into an absorbing Markov chain, and then making use of some of its properties.

The theorem holds trivially if $|V| = 0$, since the statement "for all $v \in V$" is vacuously true.

Assume now that $G = (V, E)$ contains at least one node. Let $G' = (V, E')$, where

$$E' = E \cup \{(s, s, 1) \mid s \text{ is a sink in } V\}.$$

By construction, each node in G' has outgoing edges whose weights sum to 1, so G' is a Markov chain. Furthermore, G satisfies the following:

1. There exists at least one sink (follows from well-formedness of G and $|V| > 0$),
2. Every node has a path to at least one sink (follows from well-formedness of G).

Thus, G' is an absorbing Markov chain: every state can reach an absorbing state (a sink with a self loop) in finite steps. A standard result for such Markov chains is that their transition matrix P can be rearranged as:

$$P = \begin{bmatrix} Q & R \\ 0 & I_r \end{bmatrix},$$

where Q describes transitions between transient (non-sink) states, R transitions from transient to absorbing states, and I_r, the identity matrix, the transitions from absorbing states, which necessarily always transition back into themselves. After infinite transitions, the probability of still being in a transient state is zero, thus $\lim_{k\to\infty} Q^k = 0$, which implies that Q's spectral radius $\rho(Q) < 1$.

Now consider the system of linear equations

$$(I - W)p' = \mathbb{1}$$

Let W^T be W's transpose. Then W^T structurally resembles a Markov chain's transition matrix, with row sums = 1.

Define the subgraph $D \subset G$, containing only all delegating (non-sink) nodes. Let W_D^T be the transpose of the weight matrix for D, which only includes delegations among delegating nodes.

Then $W_D^T = Q$, so $W_D = Q^T$ and thus $\rho(W_D) = \rho(Q) < 1$. The equality of the two matrices follows from the observation that the construction of G' from G only adds self-loops to sink nodes, leaving all delegating nodes and their outgoing edges unchanged.

Note that standing power values for delegating nodes depend only on the values of other delegators — never on sink nodes. This justifies restricting the analysis to the submatrix W_D, as the equation system governing these nodes is self-contained.

We now restrict our attention to the system of equations over only the transient nodes:

$$(I - W_D)p'_D = \mathbb{1}.$$

Since $\rho(W_D) < 1$, the Neumann series

$$(I - W_D)^{-1} = \sum_{k=0}^{\infty} W_D^k$$

converges, and thus $(I - W_D)$ is invertible. Therefore, p'_D has a unique solution.

Finally, since the standing power of each sink depends only on the standing power values of delegators, and those are uniquely determined, the standing power value of all nodes is uniquely determined as well. □

3.3.2. Conservation of Power

In order to assure that the power is conserved during delegation, it may seem intuitive to add a constraint $\sum_{s \in S} p_s = |V|$ to the system of linear equations. However, we prove that such an equation is not necessary, as the other equations in the system of equations already imply the conservation of power.

Theorem 3. *For a well-formed delegation graph $G = (V, E)$, with $V = S \dot{\bigcup} D$, $\sum_{s \in S} p_s = |V|$ holds.*

Proof. We start with the solutions $\{p'_v | v \in V\}$.

Summing over all $v \in V$:

$$\begin{aligned}
\sum_{v \in V} p'v &= \sum_{v \in V} \left(1 + \sum_{(u,v,w) \in E} w p'_u\right) \\
&= \sum_{v \in V} 1 + \sum_{v \in V} \left(\sum_{(u,v,w) \in E} w p'_u\right) \\
&= |V| + \sum_{(u,v,w) \in E} w p'_u
\end{aligned} \tag{3.1}$$

Now regroup the second term by the source node u:

$$\begin{aligned}
\sum_{(u,v,w) \in E} w p'_u &= \sum_{u \in V} \left(\sum_{(u,v,w) \in E} w p'_u\right) \\
&= \sum_{u \in V} p'_u \sum_{(u,v,w) \in E} w
\end{aligned}$$

According to our definition of a delegation graph, all sinks have no outgoing notes, and all delegators's outgoing node weights add up to 1. So, for any node u:

$$\sum_{(u,v,w)\in E} w = \begin{cases} 1, u \in D \\ 0, u \in S \end{cases}$$

Thus we can split the outer sum:

$$\begin{aligned} &= \left(\sum_{u\in D} p'_u \sum_{(u,v,w)\in E} w\right) + \left(\sum_{u\in S} p'_u \sum_{(u,v,w)\in E} w\right) \\ &= \left(\sum_{u\in D} p'_u \cdot 1\right) + \left(\sum_{u\in S} p'_u \cdot 0\right) \\ &= \sum_{u\in D} p'_u \end{aligned} \tag{3.2}$$

At the same time, $V = S \bigcup D$, and S and D are disjunct, we can split the term $\sum_{v\in V} p'_v$ into:

$$\sum_{v\in V} p'_v = \sum_{v\in S} p'_v + \sum_{v\in D} p'_v \tag{3.3}$$

Therefore, combining (3.2) and (3.3), the original equality (3.1) turns into:

$$\begin{aligned} \sum_{v\in S} p'_v + \sum_{v\in D} p'_v &= |V| + \sum_{u\in D} p'_u \\ \implies \sum_{v\in S} p'_v &= |V| \end{aligned}$$

□

3.3.3. Resolving Delegations by Solving a System of Linear Equations

With the insights gained in the previous sections in mind, it is now possible to formulate the following method to resolving delegation graphs.

1. Set up a system of linear equations, such that for each node $v \in V$ there is an equation $p'_v = 1 + \sum_{(u,v,w)\in E} w p'_u$
2. Solve the system of linear equations to find the value of p'_v for all $v \in V$
3. For each $s \in S$ set $p_s = p'_s$
4. For each $d \in D$ set $p_d = 0$

4. Implementation

This design above allows for multiple implementations, which will be introduced in this section. This section will also discuss briefly the robustness of the implementations, meaning how they respond to invalid input graphs. This paper will cover three implementations, which were chosen as they promise efficiency and scalability.

The implementations were coded in Python. Python is versatile, simple, performant, and offers a large collection of helpful libraries like NetworkX, a library for working with graphs [15]. Links to the implementations can be found in the appendix at section A.1.1

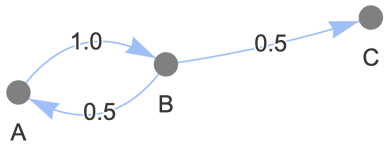
The algorithms take as input python dictionaries ("dicts"), which map a key to a value [25]. The delegation graph is represented in a "dict of dicts" format, where every key in the outer dict is a node, and the value is another dict, which has the node's delegates as keys, and the weight of the delegation as value. The algorithms use as input inverse dict-of-dicts, where the inner dictionaries represent a node's incoming rather than outdoing delegations. Figure 4.1 shows an example of this. Considering that the standing power equations used in the system of linear equations list contain the incoming delegations for each node, this design choice improves efficiency as an algorithm can look up a node in the dictionary and immediately learn about all of its incoming delegations.

4.1. Linear Systems Solver

The first approach uses a dedicated linear system solver. We use SciPy's `scipy.sparse.linalg.spsolve` solver, which is optimized for sparse matrices [29]. A sparse solver is better equipped to resolve delegations, if we assume that realistically each delegator only delegates to a few delegates. Since each entry in matrix W can be mapped to one unique edge $(u, v, p) \in E$, the matrix likely has relatively few non zero entries compared to its size.

The implementation makes use of SciPy's Compressed Sparse Column (CSC) arrays, which builds matrixes using (x, y) coordinates and their corresponding data, which unless explicitly set, is 0 [29].

The solver solves the system of linear equations directly, using the SuperLU solver, meaning it will always solve the system of linear equations perfectly accurately, unlike



(a) Delegation graph

```
{'A': {'B': 0.5},
 'B': {'A': 1},
 'C': {'B': 0.5},
 }
```

(b) Inverse dict representation

Figure 4.1.: Delegation graph and its inverse dict representation

the other implementations discussed below [21]. The solver's results are then cleaned, setting each node with outgoing edges', so each delegator's, power to zero, and returned.

The implementation will be referred to as Linear Systems Solver (LS) throughout the paper.

4.2. Linear Programming Solver

Secondly, we use the Python library PuLP, which provides an interface to linear programming solvers [23]. To resolve the delegation graphs, we use the "Coin-or branch and cut" (CBC) solver, since it is free and open-source [13]. Given the academic context and the moderate size of our delegation graphs, CBC provides a balance between performance and accessibility. Moreover, since our model solves a system of linear equations with a unique solution, the choice of solver has little influence on the outcome itself, even if CBC is not the most optimized solver for this class of problems. While commercial solvers may offer faster runtimes, CBC is sufficient for our use case and ensures reproducibility without licensing constraints.

The algorithm first sets up the linear program, setting up an equation $p'_v = \sum_{(u,v,w)\in E} 1 + wp'_u$ for each node $v \in V$. This is then solved by the CBC solver, with the primal tolerance set to $5 * 10^{-3}$ to level the playing field compared to the iterative algorithm, which will be introduced in the next section. A tolerance of $5 * 10^{-3}$ assures that $|p'_v - \sum_{(u,v,w)\in E} 1 + wp'_u| < 5 * 10^{-3}$, so the solutions will be correct when rounded to the second decimal place [12]. Finally, the algorithm cleans the p'_v values, setting any delegators power to 0.

The implementation will be referred to as Linear Programing Solver (LP) throughout the paper.

4.3. Iterative Solver

The iterative solver aims to leverage the format of the input, and eliminate any unnecessary overhead. It is based on the Jacobi method of solving systems of linear equations, which solves the system by iteratively refining the solution [7], however we will prioritize an intuitive explanation of the procedure.

4.3.1. Approach

A delegation can be thought about as liquid throwing through a graph. Each delegator is a "source", and power flows from its source between nodes until it eventually ends in a sink. If a delegator A delegates half their vote to B and the other half to other nodes, half of A's power should flow to B. An algorithm should thus add 0.5 to Bs power, and remove it from A. If B is a sink, the algorithm is done resolving this delegation. However, B may not be a sink, in which case, the power continues to flow further, to B's delegates. An algorithm would need to iterate over the graph multiple times, until an equilibrium has been reached, where all power in the graph has flown into a sink. Algorithm 1 shows such an algorithm drafted in pseudocode. Each iteration, a snapshot of the power's of each node is taken, and the reassignments of power are based on this snapshot[1].

Another valid approach would be a queue-approach, where the algorithm pops node off a queue and delegates their power, and each delegate of this node gets re-added to the queue. A sweeping method treating the entire graph at once was chosen due to its increased simplicity and runtime analysis.

The following notation will be used throughout the next sections.

Let $p_v^{(i)}, i \in \mathbb{N}_0$ be `powers`[v] after the i-th iteration of the repeat-until loop, with $p_v^{(0)}$ being the initial power of a node before the first iteration has started. Using this notation, our termination condition for the repeat-until loop of algorithm 1 is: $\forall v \in V : p_v^{(i-1)} = p_v^{(i)}$

Let $P_D^{(i)} = \sum_{d \in D} p_d^{(i)}$ and $P_S^{(i)} = \sum_{s \in S} p_s^{(i)}$ be the sums of the power values all delegators and all sinks after each iteration.

Let $\delta_{(u,v,w)}^{(i)} = w * p_u^{(i)}$ be the delta assigned in line 10 of algorithm 1 during the ith iteration.

[1] If the algorithm forwent the use of such a snapshot, it would lead to inconsistencies in the edge case of a self-delegation of weight less than 1, since the self-delegator's power would change in the middle of reassigning the power. This is also how the Jacobi method approaches this challenge. Each iteration, a solution vector containing intermediate results is created, and passed as input into the next iteration.

Algorithm 1 Iterative algorithm

```
1: // Initialize each node's power to 1.0
2: for all v ∈ nodes do
3:     powers[v] ← 1.0
4: end for
5: repeat
6:     prev_powers ← powers.copy()                ▷ snapshot of previous iteration
7:     for all v ∈ nodes do
8:         // For each incoming delegation (u → v), move w_uv× previous power of u
9:         for all (u, w) ∈ delegations[v] do
10:            δ ← w × prev_powers[u]
11:            powers[u] −= δ
12:            powers[v] += δ
13:        end for
14:    end for
15: until prev_powers = powers                    ▷ a steady state has been reached
```

4.3.2. Conservation of Power

We show, that this algorithm conserves power throughout iterations.

Theorem 4. *Given a well-formed delegation graph, in algorithm 1,* $P_t^{(i)} = P_D^{(i)} + P_S^{(i)}$ *is equal to* $|V|$ *for any* $i \in \mathbb{N}_0$.

Proof. We prove the theorem inductively. When $i = 0$ (before the first iteration), each node is assigned a power of 1. So

$$\forall v \in V : p_v^{(0)} = 1 \implies P_t^{(0)} = |V|$$

Assume that for a $k \in \mathbb{N}_0 : P_t^{(k)} = |V|$. During iteration $k+1$, the algorithm will iterate over all delegations, and for each $(u, v, w) \in E$, it will remove some $\delta_{(u,v,w)}^{(k+1)}$ from node u, but add this same amount to node v. Since the delegation graph is well formed, the outgoing weights of any delegator add up to 1, so for all delegators $u \in D$, the total amount of power they delegate away during iteration $k+1$ adds up to the power they held in iteration k. Formally, for any node $u \in V$:

$$\begin{aligned}\sum_{(u,v,w)\in E} \delta^{(k+1)}_{(u,v,w)} &= \sum_{(u,v,w)\in E} w p^{(k)}_u \\ &= p^{(k)}_u \cdot \sum_{(u,v,w)\in E} w \\ &= p^{(k)}_u \cdot 1 \\ &= p^{(k)}_u\end{aligned}$$

Thus, throughout the iteration of the outer loop, any delegator u only ever moves power it already has, and for each "moving around" of power, conservation is guaranteed since any power subtracted from a delegator is re-added to the delegate. Thus $P^{(k+1)}_t = |V|$.

By the principles of induction, the assumption holds for any $i \in \mathbb{N}_0$

□

Similarity to the Previous Approach

Observing the algorithm reveals that the same equations used in the previous approach to resolve delegations can be re-found here. The algorithm starts with a vector of ones, indicating an initial power of each node of one. Furthermore, during each iteration, each node $v \in V$ gains power amounting to $\sum_{(u,v,w)\in E} \delta^{(i)}_{(u,v,w)}$. This can be rearranged as follows:

$$\begin{aligned}p^{(i)}_v += &\sum_{(u,v,w)\in E} \delta^{(i)}_{(u,v,w)} \\ += &\sum_{(u,v,w)\in E} w p^{(i-1)}_u\end{aligned}$$

Since power is conserved, if this node is a delegator, power amounting to $p^{(i-1)}_v$ will be delegated out of it. Nevertheless, each iteration the algorithm essentially solves:

$$p^{(i)}_v = \sum_{(u,v,w)\in E} w p^{(i-1)}_v \forall v \in V$$

This is the same as the standing power assigned to all nodes in the previous approach. ($p'_v = 1 + \sum_{(u,v,w)\in E} w p'_u$). Thus, this algorithm solves the same problem as the system of linear equations introduced in the previous section.

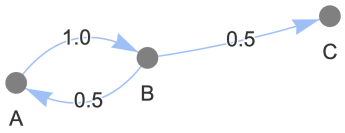


Figure 4.2.: Delegation graph with a cycle

This section will show, that the algorithm does not necessarily terminate despite being input a well formed delegation graph. We then propose an amended algorithm.

We first prove that the algorithm terminates at iteration $i+1$ exactly when $P_D^{(i)} = 0$.

Lemma 5. $p_v^{(i)} = p_v^{(i+1)} \forall v \in V \Leftrightarrow P_D^{(i)} = 0$.

Proof.

$$\begin{aligned} P_D^{(i)} = 0 &\Leftrightarrow p_d^{(i)} = 0, \forall d \in D \\ &\Leftrightarrow \nexists (d,v,w) \in E : \delta_{(d,v,w)}^{(i+1)} > 0 \quad (\delta \text{ of a node with power 0 is 0}) \\ &\Leftrightarrow p_v^{(i)} = p_v^{(i+1)} \forall v \in V \quad \square \qquad (p_v \text{ doesn't change if zero is added to it}) \end{aligned}$$

□

This insights lets us prove, that the algorithm may never terminate.

Theorem 6. *Given a well-formed delegation graph, algorithm 1 may not terminate.*

Proof. Assume the algorithm terminates on a well-formed delegation graph..

Take the following well formed delegation graph $G = (S \dot{\bigcup} D, E)$ with $S = \{C\}$ and $D = \{A, B\}$, visualized in fig. 4.2. Since the algorithm terminates, there must be an $i \in \mathbb{N}$ such that $P_D^{(i)} = 0$ (theorem 5).

In each iteration i, half of B's power is passed to A, which immediately returns it in the next iteration. This creates an infinite back-and-forth delegation loop between A and B, where power keeps circulating and leaking only partially to the sink C.

The power values of the three nodes as the algorithm iterates are shown in table 4.1. The power values for A and B decrease, but never reach zero, which implies that:

$$\forall i : P_D^{(i)} > 0$$

□

Table 4.1.: $p_v^{(i)}$ values of nodes in the graph in fig. 4.2

i	p_A	p_B	p_C
0	1	1	1
1	0.5	1	1.5
2	0.5	0.5	2
3	0.25	0.5	2.25
4	0.25	0.25	2.50
5	0.125	0.25	2.625
...			

Practically, the algorithm needs a cutoff condition, which terminates the repeat-until loop once the power values calculated are close enough to the real, final values. Since these are unknown before the algorithm terminates, we can count how much power is being shifted throughout the graph each iteration, and terminate once this value is sufficiently small. An extension to algorithm 1 could look like algorithm 2. We now show that is also conserves power, and that it terminates.

Conservation of Power

Theorem 4 states that algorithm 1 conserves power across iterations. The same proof applies to algorithm 2, since only the if-condition of the outer loop has changed, but the algorithm works the same way. So while the algorithm will iterate less, power remains conserved across iterations.

Termination

Lemma 7. *Given a well-formed delegation graph, algorithm 2 terminates if* $\texttt{cutoff} > 0$.

Proof. For this proof, we use the fact that this method of resolving delegation essentially implements the Jacobi method for solving linear equations. In the Jacobi method, the system of linear equations is iteratively refined as follows. Note, that we use the matrix representation of the system of linear equations from section 3.3.1.

$$p_{(i+1)} = Wp_{(i)} + 1$$

$p_{(i)}$ is the vector of power values after iteration i. Unfolding this recursive equation, with $p_0 = \mathbb{1}$, the vector of ones, yields:

Algorithm 2 Iterative Algorithm with a cuttoff value. Changes from algorithm 1 are highlighted.

```
// Initialize each node's power to 1.0
for all v ∈ nodes do
    powers[v] ← 1.0
end for
repeat
    prev_powers ← powers.copy()
    total_change ← 0
    for all v ∈ nodes do
        // For each incoming delegation (u → v), move w_uv× previous power of u
        for all (u, w) ∈ delegations[v] do
            δ ← w × prev_powers[u]
            powers[v] += δ
            powers[u] −= δ
            total_change += δ
        end for
    end for
until total_change < cutoff
```

$$
\begin{aligned}
p^{(1)} &= Wp^{(0)} + \mathbb{1} \\
p^{(2)} &= Wp^{(1)} + \mathbb{1} = W(Wp^{(0)} + \mathbb{1}) + \mathbb{1} = W^2p^{(0)} + W\mathbb{1} + \mathbb{1} \\
p^{(3)} &= Wp^{(2)} + \mathbb{1} = W^3p^{(0)} + W^2\mathbb{1} + W\mathbb{1} + \mathbb{1} \\
&\vdots \\
p^{(n)} &= W^np^{(0)} + \sum_{k=0}^{n-1} W^k\mathbb{1}
\end{aligned}
$$

Since the system of linear equations has a unique solution, as proven in section 3.3.1, we know that matrix $\rho(W) < 1$. This implies, that:

$$\lim_{k\to\infty} W^k = 0 \quad \text{and} \quad \sum_{k=0}^{\infty} W^k = (I - W)^{-1}$$

Thus, $p^{(n)}$ converges. This means, that the changes in power per iteration must

shrink strictly monotonically, meaning the `total_change` shrinks strictly. Thus, it will eventually fall under the `cutoff` and the algorithm terminates.

□

4.4. Robustness

This section describes the different implementations behavior when a delegation graph is not well formed. Specifically, their behaviors when outgoing delegation weights are invalid, so not adding up to 1, and if the delegation graph contains a closed delegation cycle.

4.4.1. Invalid delegations

On their own, neither of the three implementations will definitively cause an error when delegations are invalid. The iterative implementation is "dumb", in the sense that it moves around power as is instructed by the delegations. If a delegator delegates more than they are meant to, meaning the outgoing edge weights add up to a value greater than 1, the algorithm behavior becomes undefined, since the delegators power may become negative, at which point the delta in power calculated from its power also becomes negative, which messes with the `total_change` value in unpredictable ways.

If a delegate delegates less than their vote, this causes less of an issue. As long as the delta calculated at $\delta \leftarrow w \times \texttt{prev\_powers}[u]$ does not become negative, the algorithm's will still find the power correctly. A well-formed delegation graph is allowed to contain self-delegations as long as their weight is lower than 1, such as the delegation in fig. 4.3a. In this situation, power still leaves the node, but less slowly. When the iterative algorithm goes over the graph, not delegating enough weight has the same effect as such a self-delegation, since in the former situation power gets subtracted and then re-added to the node, while in the latter it just remains untouched. Thus, the two graphs in fig. 4.3 yield the same result, B having a power of 2.

For the two implementations directly based on solving systems of linear equations, this is different. Node `B`'s power would end up as 1.1, since the system of linear equations looks as follows:

$$p_A = 1$$
$$p_B = 1 + 0.1 p_A$$

As long as the delegations form a matrix that is singular, there will be a unique solution, so even with invalid delegations, the algorithm will find power values, however

(a) Permissible delegation graph

(b) Invalid delegation graph

Figure 4.3.: Two similar delegation graphs

they most likely differ from the power values that would be expected, and probably will not conserve total power properly.

4.4.2. Closed Delegation Cycles

If the delegations form a closed delegation cycle, the iterative algorithm will not terminate, since the algorithm will iterate any power that is in or enters the cycle around the cycle indefinitely.

For the other two algorithms however, such a cycle can be caught. The equations for the standing power of the nodes within the cycle are linearly dependent on each other, thus the matrix resulting from them is not singular, and hence doesn't have a single solution. Solvers of systems of linear equations catch this and throw an error. Similarly, the LP solver will find that the linear program is infeasible.

What is worth mentioning however, is that since we allow fractional delegation, it suffices if just one node in a closed delegation cycle decides to delegate to a node outside of a delegation cycle (or turns into a sink), for the delegations to become resolvable again. The cycles that will be explored in section 5.2.4 are example of such a situation.

5. Evaluation

The three solvers will be evaluated based on their runtime and scalability. The evaluation will first cover synthetically generated delegation graphs including randomly generated small and big graphs as well as corner cases, then delegations graphs generated based on social behaviors, so-called social graphs, and finally graphs based on real-world datasets.

All results and code used in this section can be found in the annex in section A.2.

5.1. Method

5.1.1. Generating Random Delegation Graphs

For section 5.2 on synthetically generated graphs we built an algorithm, that builds custom delegation graphs. The algorithm generates an empty graph with n nodes, and then adds between zero and three delegations per node to random other nodes, ensuring that there are no closed delegation cycles. This algorithm can be found using the link at section A.1.2. We acknowledge that these assumptions may not accurately reflect real-world delegation graphs. As has been introduced in section 2.2.1, studies have shown that delegates do not choose randomly; instead, votes often concentrate among a few individuals. This approach also overlooks potential behavioral tendencies, such as voters preferring to delegate to those they perceive as more competent or confident, and the emergence of highly popular "super-voters" who accumulate disproportionate influence. These concerns are addressed in section 5.3, when delegation graphs based on social graphs are benchmarked.

5.1.2. Preprocessing

In order to be able to benchmark algorithms that resolve delegations, the input graphs need to be well-formed delegation graphs. Many of the graphs we use for benchmarking are not well-formed delegation graphs out-of-the-box. This subsection details the process of how any arbitrary graphs, including undirected and unweighted graphs, can be turned into well-formed delegation graphs. An overview of this process is shown in fig. 5.1.

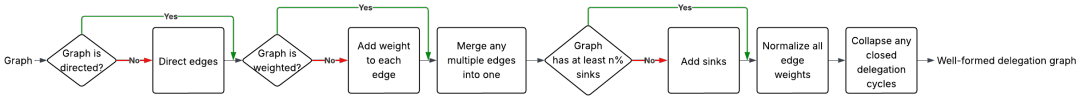


Figure 5.1.: Preprocessign pipeline to turn any graph into a delegation graph

If the graph is undirected, is it given a direction. This is done arbitrarily, with the algorithm interpreting undirected edges stored as (u, v) as directed edges from u to v. If the algorithm fails to find a weight for an edge, it will also assign it a weight of one. Next, any multiple edges, so parallel edges going from the same node to the same node are merged, with any weights being added together. If less than n% of the graph's nodes are sinks, the algorithm randomly removes all outgoing delegations of delegators, turning them into sinks. This n value can be adjusted depending on the user case. After this, the algorithm searches for any closed delegation cycles, and collapses all it finds into a single sink node. Specifically, the algorithm searches for strongly connected components (STCCs) in the graph, so components of the graph where each node can reach each other node, and checks if it it is a closed delegation cycle, by checking if any of the nodes within this STCC delegate to a node outside of the STCC. An exception to this are sinks who have no delegators, these are technically STCCs with no outgoing edges, however they are not closed delegation cycles. All closed delegation cycles are collapsed into a "lost" node, which means that any delegations to the cycle get re-directed to this specially created node. As shown in section 3.2.3, the resulting graph from this operation is a well-formed delegation graph, since all power that flows into closed delegation cycles now flows into a sink, so the graph is free of closed delegation cycles.

The code for the preprocessing algorithm can be found in section A.1.3.

5.1.3. Measurement

Despite all algorithm's taking in put in inverse dict-of-dicts format, there may still be preprocessing necessary. While the iterative solver can use the inverse dict-of-dicts directly, using it as a lookup table as it spreads power around the graph, the other solvers require the system of linear equations in specific formats, which need to be set up from the inverse dict-of-dicts input. Including such set-up time in benchmarks may be misleading, as this time is not spent on actually resolving delegations, thus we separated the set-up and resolving, and in the benchmarks only the time spent actually resolving the delegations is used; any set-up time is ignored. Nevertheless, in practice, the set-up time can be a relevant factor, depending on the use case and

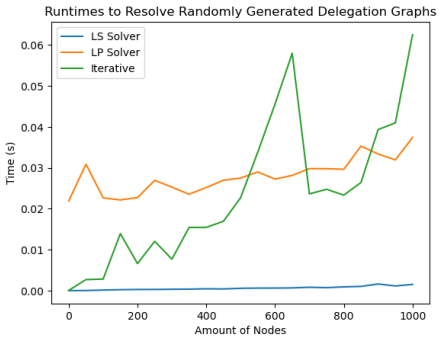


Figure 5.2.: Runtimes to resolve randomly generated delegation graphs

data format, when choosing between different approaches or implementations. The set-up procedures required for each implementation are described in more detail in the sections below.

To minimize the impact of background noise and measurement fluctuations on the benchmarks, algorithms with very short runtimes were executed multiple times, and the average runtime was recorded. The recorded runtimes always indicate just the runtime for the algorithms to resolve the delegations, times for set-up are not included.

Furthermore, all runtimes are reported based on the actual number of nodes present in the resolved graph. For example, if a delegation graph initially contains 5000 nodes, but 500 of are in closed delegation cycles, these are collapsed into a single sink node. When the algorithm is thus resolving the graph, all of these 500 nodes are in effect just one node. Thus, when presenting benchmarks results, for this graph, we will show the amount of nodes resulting in the presented runtime as 4501.

5.2. Synthetic Graphs

5.2.1. Small Graphs

In order to explore the three algorithm's behavior on small graphs, we used the graph generator to generate graphs with zero to 1000 nodes. Figure 5.2 shows the results of this benchmark.

We see, that the LS Implementation, optimized for sparse matrices, outperforms the other two algorithms. Its growth in runtime is so small, that the line looks to be staying flat on the x-axis. However, with a graph of 1000 nodes, its runtime is about 0.002 seconds. Both the LS and LP implementations display a rather steady, yet growing runtime. The LP solver seems to have some overhead, since even when the graph has zero nodes, it has a runtime of about 0.02 seconds.

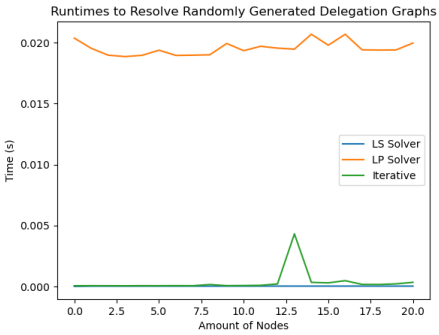


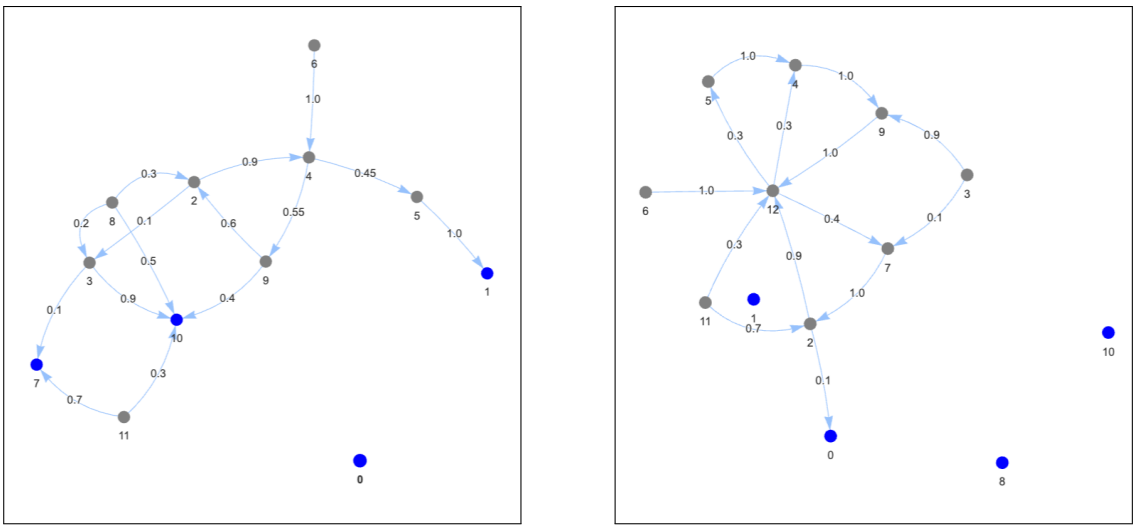
Figure 5.3.: Runtimes to resolve randomly generated delegation graphs

Furthermore, we can observe large spikes in the runtime of the iterative approach. For example, resolving the delegation graph with 650 nodes takes the algorithm more than double the amount of time than the graph with 700 nodes. Exploring this phenomenon more closely, we find that a graph with 13 nodes takes the iterative algorithm a lot more time than the graph with 12 or 14 nodes, as shown in fig. 5.3. At 12 nodes, the runtime of the iterative algorithm is just about 0.2 milliseconds, at 14 nodes it is about 0.3 milliseconds, but at 13 nodes it is 4.3 milliseconds.

A possible explanation for this spike may be, that when the graph has 12 and 14 nodes, it iterates only 23 and 39 times respectively, before cutting off, while when it has 13 nodes it iterates 740 times before cutting off. Figure 5.4 shows the two graphs with 12 and 13 nodes.

When the graph has 13 nodes, power entering the sink node `0` only has a delegation of weight 0.1. Each iteration, 10% of the power within node `2` enters the sink, but the other 90% is dispersed into the graph. This forces the iterative algorithm to iterate this power around the graph until eventually enough of it has collected in node `0`. In the graph with 12 nodes on the other hand, this effect is visibly less present. A delegation from node `3` to sink node `7` only has a weight of 0.1 too, but the other 0.9 of this vote are directed toward another sink node `10`.

This is an important shortcoming of the iterative algorithm. Power can easily get trapped within permissible delegation cycles that only have a small drain allowing the power to escape from the cycle. Each iteration, if a great proportion of the nodes with draining edges' power is sent back into a cycle, the algorithm needs to continuously iterate until the power is back at the drain nodes, however depending on the cycle this may happen very inefficiently. This phenomenon will be tested more in section 5.2.4



(a) 12 nodes

(b) 13 nodes

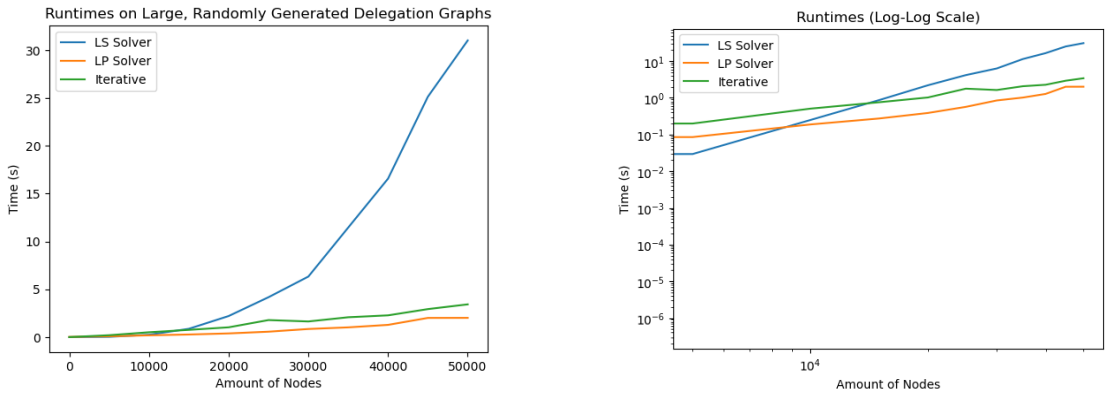
Figure 5.4.: Delegation graphs with 12 and 13 nodes (blue nodes are sinks)

5.2.2. Large Graphs

Delegation graphs may grow arbitrarily large. National elections for example can contains up to hundreds of millions of participants. This section explores how the algorithms perform when having to resolve graphs with a lot of nodes. Again, the graphs will be randomly generated, such that each nodes has between 0 and 3 delegates.

Figure 5.5 shows that even as the delegation graphs get larger, the LS solver's runtime grows faster than that of the other two implementations. For resolving smaller graphs, the LS solver outperforms the LP solver, with a runtime of almost zero for empty or very small graphs, while the LP solver has a clearly non-zero runtime even for very small graphs. However, at around 12 000 nodes, this changes, as the LP solver's runtime's slower growth catches up with that of the LS solver, closely followed by the iterative implementation.

Looking at the loglog graph, the runtime growths seem to be following a power law. Fitting the data into different curves confirms, that the implementations likely all grow according to a power law, with the LS solver growing the quickest, about $O(n^{3.07})$, n being the amount of nodes, the LP solver fitting into an $O(n^{1.41})$ curve and the iterative solver $O(n^{1.20})$. These growth classes are probably not generalizable to all delegation graphs, since the runtime may grow with different coefficients depending on the underlying delegation graphs. This will be explored in the following sections.



(a) Linear scale

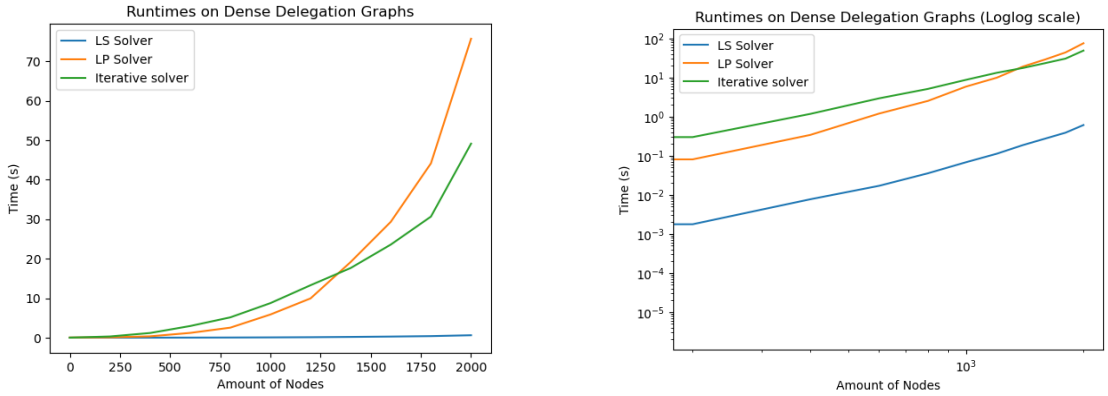
(b) Loglog scale

Figure 5.5.: Runtimes to resolve large, randomly generated delegation graphs

5.2.3. Dense Graphs

While we expect most delegators in any delegation graph to only delegate to a handful of people, a well formed delegation graph can have any number of delegates per delegator. Thus, it is also interesting to compare how the three algorithms compare when resolving more dense graphs. In this section, we test the three implementations on NetworkX's $G_{n,p}$ graph generator `gnp_random_graph`, which returns a directed graph with n nodes, where each node connected to each other node with probability p, which is set to 0.5 for the remainder of this section [15]. All of the nodes in these graphs are not sinks, since they all have outgoing edges. Normally, these graphs would be one large closed delegation cycle, where nobody votes, and the preprocessing pipeline would collapse them all into one sink node. Thus, in order to be able to resolve meaningful power values, we turn 10% of nodes into sinks by removing the outgoing edges. Then, each delegators vote is equally distributed among all of its outgoing edges, such that the edge weights add up to 1. Finally, any remaining closed delegation cycles are collapsed, however, all graphs that the benchmark was run on ended up having no closed delegations after the measures were applied.

Figure 5.6 shows the runtime of these three algorithms. The runtimes are greater than the runtimes found in the previous section. At 2000 nodes, the runtimes on the randomly generated, relatively sparse, graphs was well under a second for each algorithm, while it takes the iterative and LP solver 49 and 76 seconds respectively. The LS solver's runtime grows from about 0.03 to 0.61 to resolve the dense graph. The LS solver surprisingly outperforms both solvers, however looking at its runtime growth in a loglog graph reveals, that it grows at a similar rate to the others. Which algorithm



(a) Linear scale

(b) Loglog scale

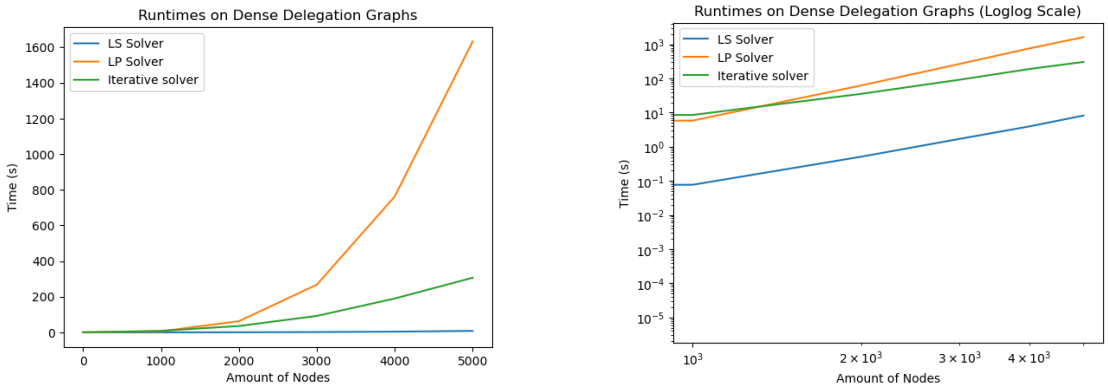
Figure 5.6.: Runtimes to resolve dense delegation graphs

is the slowest depends on the amount of nodes in the graph, with the iterative solver showing slower runtime growth, eventually outperforming the LP solver at around 1300 nodes. The delegations in this graph are very fine grained, since each delegating node delegates to half of all nodes, and the delegators vote is distributed equally to all its delegates, the weight of each delegation is just:

$$\frac{1}{0.5n} = \frac{2}{n}$$

Intuitively, this should put the iterative algorithm at a disadvantage, since it needs to iterate around a lot of delegations, each only moving around small amounts of power at a time. However, it seems that the LP implementation struggles with these graphs as well, likely because the large amount of delegations results in lot of long linear equations to solve. Even though the LS solver is optimized for sparse matrices, it demonstrates impressive efficiency at solving these kinds of problems.

Testing the three algorithms on larger dense graphs, reveals that the LP solver's runtime is considerably worse than that of both the iterative and LS solver. The results of this benchmark are visualized in fig. 5.7. A dense graph with 5,000 nodes, takes the LS solver only about eight seconds, the iterative solver 306 seconds, and the LP solver 1632 seconds. Fitting curves on these runtimes again reveals that they likely follow a power law, the runtime classes for the LS, LP and iterative solver being $O(n^{2.89})$, $O(n^{3.51})$ and $O(n^{2.24})$ respectively. As is also visible in fig. 5.7b, this means the iterative algorithm has the best runtime class, however for dense graphs of the size that was tested, it is not the most performant algorithm.



(a) Linear scale

(b) Loglog scale

Figure 5.7.: Runtimes to resolve larger dense delegation graphs

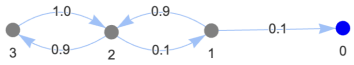


Figure 5.8.: A delegation graph with a cycle that retains a lot of its power

5.2.4. Cycles Which Retain a Lot of their Power

To explore one of the iterative algorithm's shortcomings, this section will explore and compare runtime behavior for delegation cycles which are not closed, but contain only few, weak edges for power to drain, thus forcing power to iterate around in the cycle before it reaches a sink. Such situations cause the iterative algorithm's runtime to spike, as has already been observed in fig. 5.4b, as the algorithm needs to move around the power within these cycles until enough as drained for the `total_change` to fall below the threshold. A further example of such a cycle, motived by graphs which we encountered while experimenting on delegation graphs is the tail-shaped graph in fig. 5.8.

For the benchmarking, we construct graphs with a circular shape, where delegates all delegate power to the next node in the cycle. One node in the cycle contains an edge with weight 0.1 to a sink, while the other 0.9 of its power goes to the first node in the cycle. Figure 5.9 contains an exemplary image of such a graph with 10 nodes.

The runtimes in fig. 5.10a show, that as expected, the iterative algorithm struggles considerably with the resolution of these graphs, while the other two algorithms exhibit

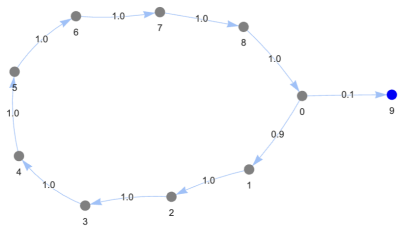
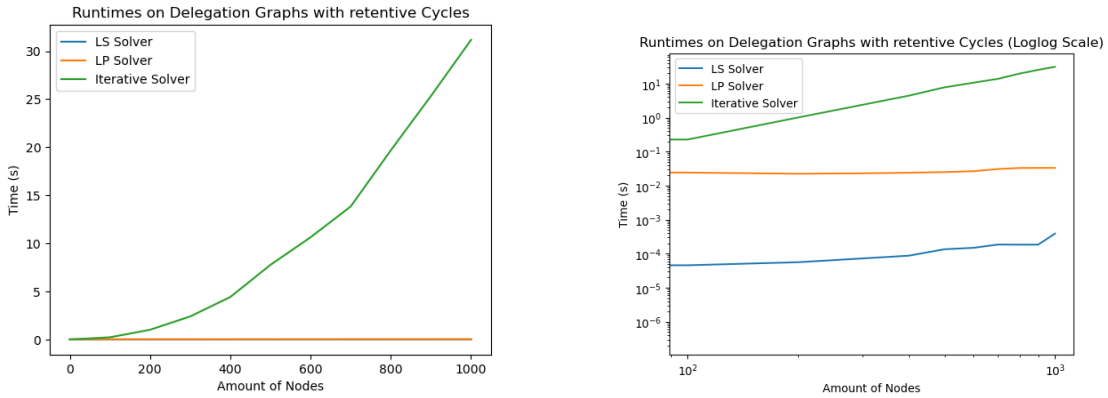


Figure 5.9.: An example of the cycles used for the benchmarks. The blue node is the sink



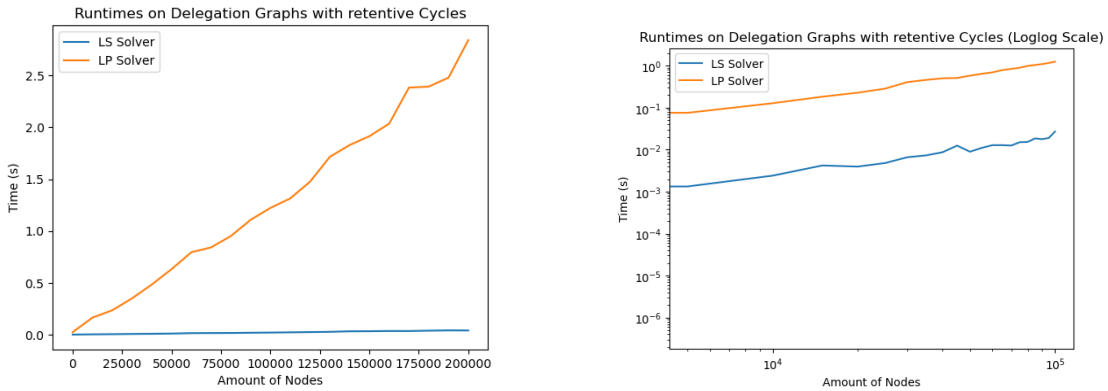
(a) Linear scale

(b) Loglog scale

Figure 5.10.: Runtimes to resolve cycles which retain a lot of their power

behavior similar to that on randomly generated sparse delegation graphs. The growth of the runtimes seems to be polynomial, with the iterative algorithm belonging to the runtime class $O(n^{2.12})$. Being able to resolve these kinds of loops is one of the greatest strength of the two approaches, which don't simulate power as flow through the graph. By directly solving the system of linear equations, they better equipped to deal with this corner case.

As seen in fig. 5.11, which shows the LS and LP solver's runtimes as this type of graph scales to 200000 nodes, their runtime growth is linear. In fact, they fit almost perfectly into a linear regression, both with a slope of almost zero, suggesting that even as these graphs scale to even greater orders of magnitude, the solver's runtime is barely affected. An explanation for this efficiency may include the very sparse nature of this graph, as each node except for one has only one delegation.



(a) Linear scale

(b) Loglog scale

Figure 5.11.: Runtimes to resolve cycles which retain a lot of their power. The iterative solver's runtime, to allow inspection of the LS and LP solver's runtimes.

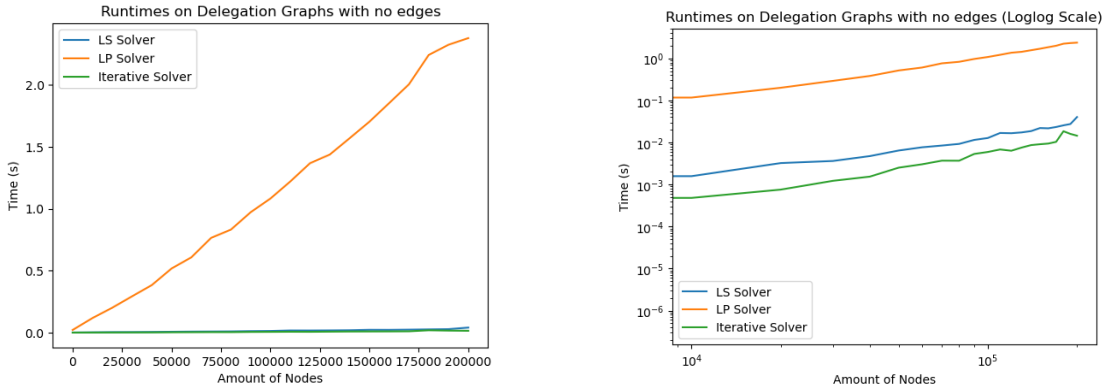
5.2.5. No Delegations

The runtime of the implementations on delegation graphs where all nodes have no outgoing edges can also provide insight, such as the runtime behavior on graphs where only few nodes delegate, or delegations are very short. Figure 5.12 shows how long it takes for the implementations to resolve graphs which have no edges at all.

The iterative algorithm outperforms the other two on these graphs, which is not surprising, since it only needs to iterate over the empty dict of delegations once, registers a `total_change` of zero, and terminates. All three solvers' runtimes seem to grow about linearly, where even at 200 000 nodes, the slowest solver, the LP solver, takes only about 2.5 seconds, suggesting they grow linearly with a slope close to one. A linear regressions confirms this observation.

5.3. Social Graphs

Social graphs provide an excellent way to create scalable models of trust within communities without real datasets. We will use them to create sample delegation graphs based on social behaviors, which can predict ways humans may delegate if given the chance to delegate fractionally. These graphs are only based on models, but they have the advantage that they can be scaled, allowing us to explore how the algorithms scale.



(a) Linear scale

(b) Loglog scale

Figure 5.12.: Runtimes to resolve delegation graphs with no edges

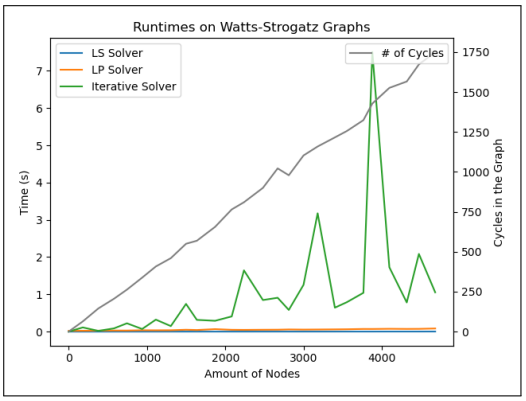
5.3.1. Small World Graphs

Small world graphs are graph which exhibit a relatively high clustering coefficient, meaning nodes are very interconnected, with short path lengths between arbitrary nodes. Watts and Strogatz propose a graph generator to generate these kinds of graphs artificially. The graph generator takes three parameters n, k, and p. It connects each of the n nodes with k of their neighbors, and once this is done, "rewires" each edge to a different node with probability p. Watts and Strogatz recommend a value for p of around 0.1, to get the two desired qualities of a small world graph: short path lengths between all nodes and high "cliqueness", meaning if two nodes are friends, their respective friends are also likely to be friends with eachother. Furthermore, they suggest that $k >> ln(n)$ in order to guarantee that the graph is connected. [30]

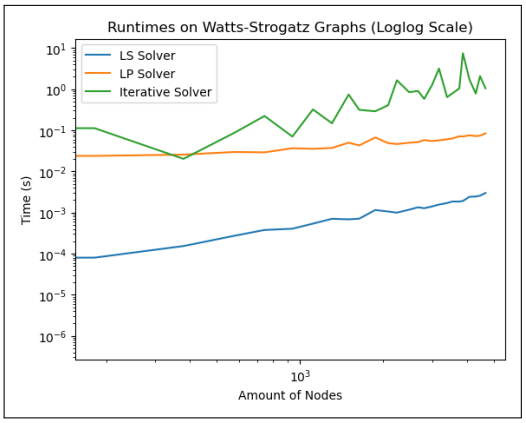
The graphs as they are recommended by Watts and Strogatz are not usable as delegation graphs. Firstly, no node in this graph is a sink, since each node has k outgoing edges. Secondly, each node has too many, a lot more than $ln(n)$, delegates. This is does not scale. In a graph with e.g. 10000 nodes, each node would delegate to almost ten people, and as the graph grows this number increases. Normally we would not expect the total size of the graph to have a very big effect on the amount of delegations per person, since delegation of votes is a personal, individual question. Thus, we adapt these Watts-Strogatz graph generation graphs in the following ways.

1. Each node is connected to its exactly $k = 4$ neighbors.
2. 60% of the edges in this graph are removed.
3. Finally, as the graphs get preprocessed in the preprocessing pipeline, we enforce

Figure 5.13.: A Watts-Strogatz graph with $(n, k, p) = (40, 4, 0.1)$, where 60% of the edges have been removed



(a) Linear scale



(b) Loglog scale

Figure 5.14.: Runtimes to resolve the Watt-Strogatz based delegation graphs. The grey line indicates the amount of cycles found in the graph. It does not include closed delegation cycles, which the preprocessing pipeline collapsed.

that 20% of the nodes in this graph are sinks my removing outdoing edges of nodes.

This way, we assume that each node has four trusted friends, but of these trust relationships, only 40% are strong enough for the nodes to want to let their friend vote for them. The value of 20% is a middle ground between observations which will be made in the following sections, where some social graphs have a lot more than 20% sinks, while other have a lot less. The p of 0.1 thats Watts and Strogatz recommend is kept. A sample of how a graph generated with way is shown in fig. 5.13. The algorithm for generating these graphs can be found in section A.1.2.

Figure 5.14 shows the benchmarks for these graphs. They are similar to the results

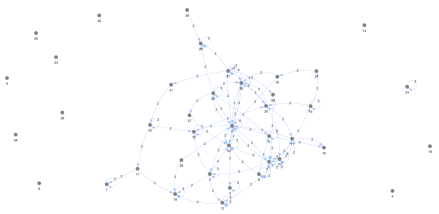


Figure 5.15.: A directed, weighted R-Mat graph with $(a, b, c, d) = (0.45, 0.25, 0.25, 0.15)$, $N = 40$, $M = 400$, where edges with weight < 2 removed

achieved when benchmarking the big cycle graphs, where the LS and LP solvers perform quite well compared to the iterative solver. The iterative solver also has a lot rather unpredictable peaks. Investigating this pattern reveals, that even despite our measures, the graphs still contained a lot of (permissible) cycles, i.e. cycles which the LS and LP solvers can solve directly, but where the iterative algorithm needs to iterate power around the loops. The amount of such cycles is shown in fig. 5.14 as a grey line. Since there is bit of randomness involved in the creation of these graphs, it can happen that those cycles vary starkly in retentiveness, meaning the amount of power they force the algorithm to re-loop through a cycle, which leads to peaks and troughs in the iterative solver's runtime.

5.3.2. R-Mat Graphs

Another method for generating artificial social graphs is the R-MAT (Recursive Matrix) model. This model requires four parameters, a, b, c, and d, which are probabilities that sum to one, as well as the desired number of nodes N and edges M. The algorithm begins with an empty $\sqrt{N} \times \sqrt{N}$ adjacency matrix, where a nonzero entry at position (i, j) indicates a directed edge from node i to node j. To determine where to place each edge, the matrix is recursively subdivided into four quadrants, with the probability of selecting a quadrant governed by the parameters a, b, c, and d. This recursive partitioning continues until a single cell (1×1) is reached, and an edge is added at that location. The process is repeated until all M edges have been assigned. [9]

As parameters we will use $(a, b, c, d) = (0.45, 0.15, 0.15, 0.25)$. These are often considered the "default" parameters to generate social graphs, and fit the recommended scheme by Chakrabarti et al., the creators of this algorithm, to generate social graphs that resemble "real-world scenarios". Furthermore, a common estimate for the average outdegree in a social graph is between around three and 15. [9, 31] Thus, we set $M = 10N$. R-Mat graphs tend to have well-connected cores, well connected enough to turn the entire core into one, big closed delegation cycle. This is not an interesting

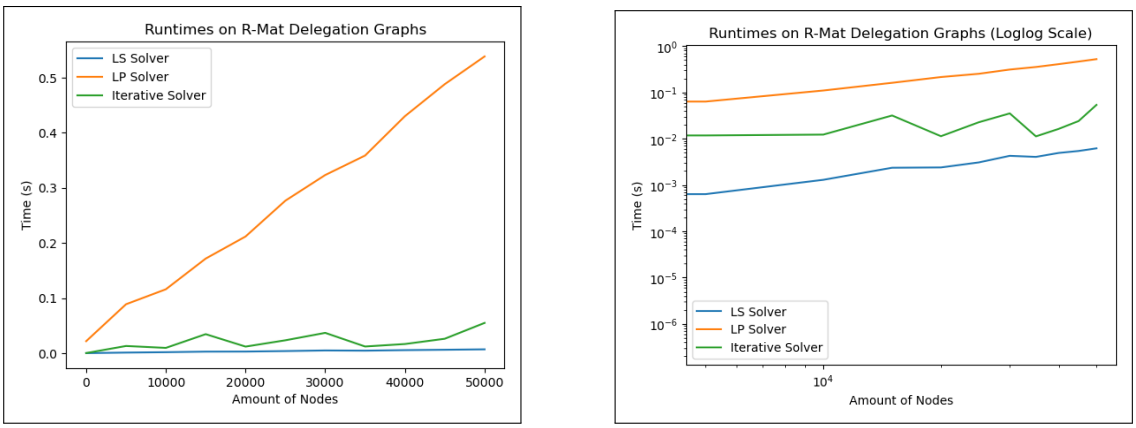
graph to explore, since the preprocessing pipeline would collapse this core into a single node, ignoring the complex connections within the core. To avoid this, we implemented a multi-edge version of the R-Mat algorithm, which, unlike the original algorithm, places edges even if they happen to land in a cell that already has an edge. We then add the total amount of edges per node-pair and direction, and remove all edges where this sum adds up to less than two. This way, we define the threshold for trust that is strong enough to warrant a delegation as edges which the algorithm placed at least twice. Another important difference to the original R-Mat algorithm is that our resulting graphs are directed, whereby we simply interpret the y-axis of the adjacency matrix as the source node and the x-axis as the destination node when placing edges. Figure 5.15 shows an example of such a generated graph with 40 nodes.

As these graphs get pre-processed, there is no need to add any artificial sinks to the graphs, since unlike the Watts-Strogatz graphs, they do generally contain sinks. In the event of closed delegation cycles, these get collapsed into a common sink node, where all the power delegated into closed delegation cycles gets collected. In the case of these delegation graphs, we find that generally well under 1% of the graph's nodes are affected by this collapse. Furthermore, the weight of the nodes is taken into account adding delegation nodes, so a node with two outdoing edges of weight two and three respectively, will end up delegating to the first node with weight 0.4 ($= \frac{2}{2+3}$), and to the second node with a weight of 0.6.

The runtimes of the algorithms on these graphs can be seen in fig. 5.16. Here, the runtimes resemble more closely those of the no-edges synthetic graphs and the runtimes of the LS and LP solver resemble the runtime they exhibited when resolving the big cycle graphs. As mentioned before, the generated R-Mat graphs contain a relatively dense community at the core, and many single nodes which neither delegate nor are delegated to. By enforcing a minimum level of trust, the amount of nodes in the periphery of the graph increases, since nodes which are connected to the core by just a single edge are disconnected from the core by the algorithm. Thus, only between around $7\% - 9\%$ of nodes in these graphs actually have delegations either incoming or outdoing, thus explaining why the runtimes behave very similar to how they did when the graph contained no edges at all.

5.4. Real-World Datasets

This section evaluates the three algorithms on some real-world datasets. While liquid democracy without fractional delegation has been implemented and tested in studies, to the authors knowledge there are no datasets for authentic fractional delegations. As an alternative, we have fallen back to transforming datasets which may resemble fractional



(a) Linear scale

(b) Loglog scale

Figure 5.16.: Runtimes to resolve the R-Mat based delegation graphs

delegations into delegation graphs. We introduce the three datasets individually, followed by a joint evaluation in section 5.4.4.

5.4.1. Bitcoin OTC Trust Network

Users trading Bitcoin on the platform "Bitcoin OTC" maintain a record of trust to other users, in order to prevent transactions with untrustworthy users. The Stanford Network Analysis Project (SNAP) provides the Bitcoin OTC Trust Graph, a graph of this trust between users. [19, 18] The graph is directed and weighted, with weights ranging from -10 to 10, total distrust to total trust.

The graph was first cleaned, to remove all edges with a non-positive trust values, before being turned into a delegation graph via the preprocessing pipeline, again not adding any sinks artificially. The preprocessing pipeline normalizes edge weights, meaning outgoing trust levels are scaled down proportionally to add up to one, preserving relative differences in trust. The pipeline turns no edges into sinks; the n-value of the algorithm is set to zero. The finished graph contains 5573 nodes, of which about 0.15% are sinks. The outdegree distribution in the Bitcoin OTC Trust Graph is shown in fig. 5.18a.

During the preprocessing of this graph, 43 of the graph's 5573 nodes were to be removed since they were in a closed delegation cycle. About 111.2 units of power were lost to closed delegation cycles, 0.02% of the total power in the graph. The distribution of powers after resolving is shown in fig. 5.20a

5.4.2. Epinions

Epinions.com is a "general consumer review site", in which members can decide whether to "trust" each other. SNAP provides a web-of-trust graph generated from this relations. [27] The graph is directed and unweighted, thus an existing edge implies trust, and a missing edge implies the lack thereof.

After preprocessing the graph into a delegation graph, with the n% sink threshold set to zero, so the algorithm does not add any new sinks to the graph by removing outgoing edges of nodes, we observe the following statistics for the delegation graph, which will be called the Epinions Graph. The Epinions Graph contains 75139 nodes, of which about 0.21% are sinks. Figure 5.18b shows the distribution of outdegrees in the Epinions Graph, outdegree meaning the amount of outdoing edges of a node. The mean outdegree is 6.76.

330 closed delegation cycles were collapsed, which affected 740 nodes, about 0.01% of nodes in the graph. Interestingly, the most powerful node after resolving is the "lost" node, the node where power goes, that was delegated into closed delegation cycles. The total amount of power lost adds up to 2777.056087, which accounts for about 0.037% of power in the graph. The distribution of powers after resolving is shown in fig. 5.20b.

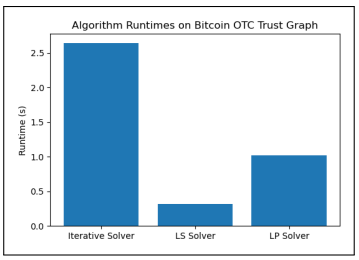
5.4.3. Slashdot Zoo

The Slashdot technology news size has a "zoo" feature, in which users can tag other users as friends and foes. The Distributed AI Laboratory in Berlin (DAI Labor) provides a graph based on this data. [20] It is a directed and weighted graph, where an edge weight of +1 indicates a friend relationship, and an edge weight of -1 indicates a foe relationship.

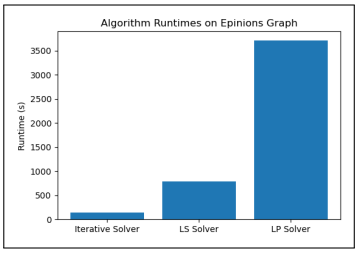
This graph was also cleaned to only contain positive edges and turned into a delegation graph, again not adding any sinks artificially. The delegation graph contains 69995 edges, of which about 0.4% are sinks. The outdegree distribution in this graph is shown in fig. 5.18c. 1061 nodes were removed due to being in a closed delegation cycle.

5.4.4. Evaluation of the datasets

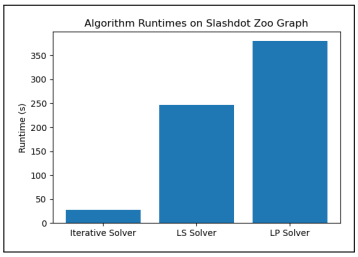
The runtime results show that for the two bigger graphs, the iterative solver is the fastest, followed by the LS solver, and then the LP solver. For the Bitcoin OTC graph, the LS solver is the fastest, followed by the LP and then the iterative solver. Figure 5.18 shows histograms for the outdegrees of the three graphs. The x-axis uses a logarithmic scale, so these graphs show, that most nodes in the graphs have a rather low outdegree. Figure 5.19 shows, that most nodes have an outdegree of zero and one. Furthermore,



(a) Bitcoin OTC dataset

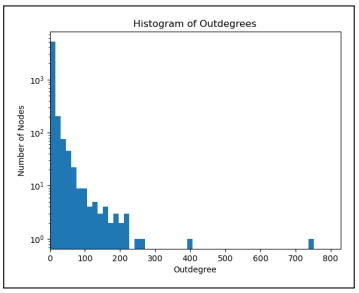


(b) Epinions dataset

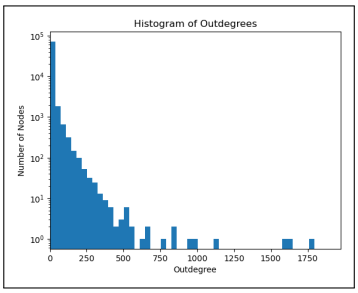


(c) Slashdot Zoo dataset

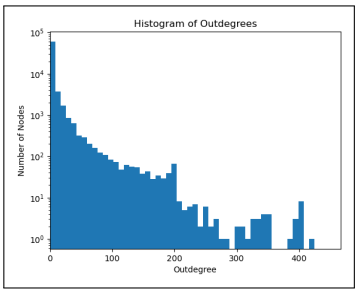
Figure 5.17.: Runtimes



(a) Bitcoin OTC dataset



(b) Epinions dataset



(c) Slashdot Zoo dataset

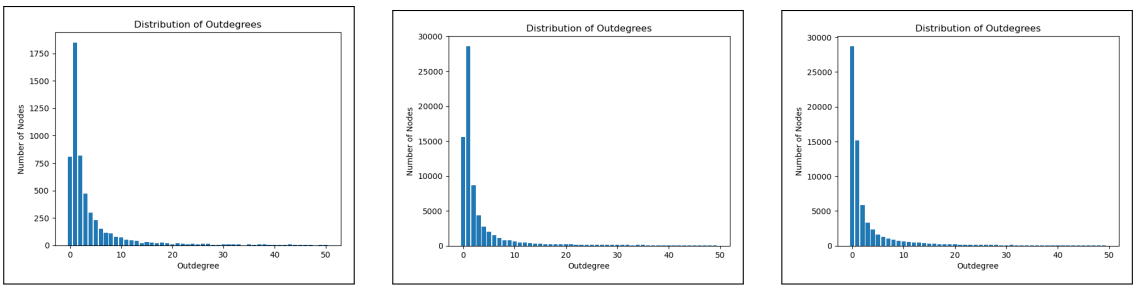
Figure 5.18.: Histogram of outdegrees

histograms of the final nodes power in fig. 5.20 show that most nodes also have very low power values.

Interestingly, there are no nodes in either of the three datasets that have a power of exactly 1.0, suggesting that there are no isolated nodes. Despite this, the iterative solver performs the fastest on the two bigger graphs, while it is clearly outperformed in the Bitcoin OTC graph, the smallest of the three. This suggests, that this solver scales better than the other two. It can also suggest, that the Epinions and Slashdot Zoo graphs are rather free of retentive cycles. Other explanations for this low runtime can include that paths to sinks are rather short. This would mean that the algorithm does not need to iterate over the graph very often.

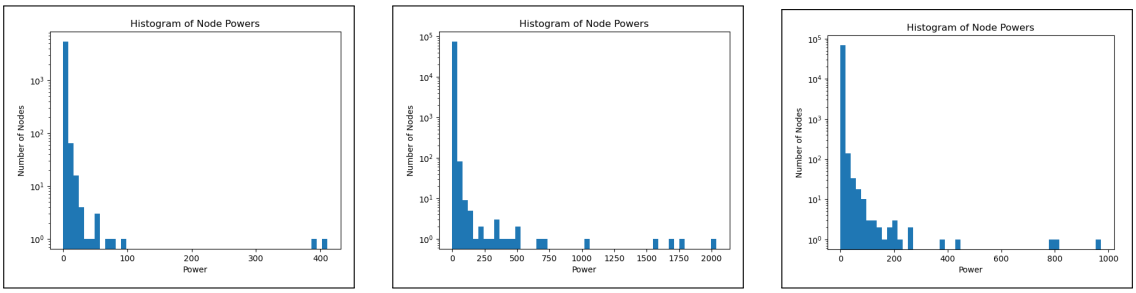
5.5. Key Insights

This evaluation effectively compares three different solvers for systems of linear equations in the context of resolving delegation graphs. The investigations showed, that for smaller delegation graphs, the LS solver generally provides the best results. When



(a) Bitcoin OTC dataset (b) Epinions dataset (c) Slashdot Zoo dataset

Figure 5.19.: Distribution of outdegrees between 0 and 50



(a) Bitcoin OTC dataset (b) Epinions dataset (c) Slashdot Zoo dataset

Figure 5.20.: Distribution of powers

graphs are very sparse or very large, the iterative solver outperforms the LS solver, however, the iterative solver is less precise than the LS solver, meaning that unless a bit of uncertainty in the power values is acceptable, the LS solver provides both great, scalable performance and perfect accuracy. When resolving large, randomly generated delegation graphs, the LS solver's runtime grows a lot faster than the LP solver's for big graphs. Also when resolving Watts-Strogatz based delegation graphs, the LS solver's runtime grew at a faster rate. However, in general the LS solver is still the more efficient choice since it generally has a faster or similar runtime growth than the LP solver, but faster runtime overall.

6. Related Work

The idea of allowing the fractional splitting of votes in Liquid Democracy is not entirely new. In 2014, Degrave first proposed "multi-proxy delegation", which closely resembles our understanding fractional delegation, in that each delegators can delegate to more than one proxy (delegate) at a time. They enforce in their implementation that the delegated vote is divided equally among the chosen proxies; for example, a voter delegating to three proxies would assign one third of their vote to each. Degrave also introduces a method for resolving multi-proxy delegations, which also entails constructing a system of linear equations, however being rather short, the article does not describe in detail how or why the method works, and does not evaluate any practical implementations of it. [10]

Bersetche revisits and extends the idea of fractional delegation in 2022 under the term multi-agent delegation. Their approach allows an arbitrary fraction of votes to be delegated, not necessarily equal fractions to each delegate. Furthermore, voters are permitted to delegate part of their vote while still retaining a fraction for themselves—enabling them to vote directly and delegate simultaneously. They explore the "presence of equillibrium states" in "delegation games" using multi-agent delegation, meaning a collection of delegations so that no agent can unilaterally change their delegations to increase their voting power. The paper finds that such states exist for delegation graphs allowing multi-agent delegation. [4]

Utke and Schmidt-Kraepelin use the term fractional delegation, although with a slightly different meaning than in this thesis. Their 2023 paper studies delegation rules that take as input ranked delegations, where each voter specifies a preference ordering over potential delegates, but not explicit vote fractions. The delegation rule then distributes voting power *fractionally* across sinks based on these rankings. While delegators cannot directly assign fractions to each delegate, the outcome resembles fractional delegation, in that multiple sinks may receive fractional amounts of a single voter's power. The authors analyze such rules and show that, unlike non-fractional delegation rules, they can simultaneously satisfy desirable properties like anonymity, confluence, and copy-robustness. [28]

Nils Wandel has implemented fractional delegation and makes the product and its codebase available via a web interface and a GitHub repository. The website can be visited at `electric.vote`. No accompanying literature exists, but an inspection of the

source code reveals that Wandel also resolves delegations using a system of linear equations. Unlike in this paper, Wandel does not calculate the power values of nodes explicitly, rather the code determines the outcome of votes with multiple option to vote for. The algorithm takes as input votes from sinks and delegations from delegators, and then first determines the standing power of each delegator, and then combines this with the votes of the sinks to determine the final aggregated result for each proposal. That this split of first resolving the standing power of delegators and then applying this to sinks is possible follows from the proof in section section 3.3.1, where the matrix P can be split into sub-matrices. Transposed, effectively turning it into the W matrix of our resolution problem, P looks as follows

$$P^T = \begin{bmatrix} Q & 0 \\ R & I_r \end{bmatrix},$$

The first rows of this matrix are the equations for the standing power of delegators (transient states). Evidently, as seen by the zero matrix next to Q, they don't depend on the standing power of any sinks, thus their standing power can be found out without knowing the standing power of any sinks. [22]

7. Conclusion and Further Research

This thesis explored the resolution of delegation graphs in Liquid Democracy under the extension of fractional delegation. By allowing voters to distribute their vote across multiple delegates, we tried to address shortcomings of traditional Liquid Democracy, including vote concentration and vulnerability to closed delegation cycles.

We proposed a formal model of fractional delegation and introduced three implementations to compute final voting power: using a solver for systems of linear equations, using a linear programming solver, and an iterative simulation of power flow. We demonstrated that the linear systems formulation yields a unique solution and ensures conservation of power under well-formed graphs. Furthermore, we introduced a preprocessing pipeline that transforms arbitrary graphs into well-formed delegation graphs, enabling resolution of delegations even in the presence of complex structures like cycles.

Our evaluation across synthetic, social, and real-world graphs shows that the linear systems solver outperforms other methods on sparse graphs, while the iterative solver scales more favorably in dense settings, void of cycles which retain a lot of power. The Linear Programming-based method proved less efficient in most circumstances.

Looking ahead, this thesis leaves several questions open. While we hypothesize that fractional delegation may reduce vote concentration by allowing voters to distribute their trust among multiple delegates, this claim requires empirical validation. A practical implemented fractional delegation platform would enable real-world testing of not only the resolution of delegations, but the entire Liquid Democracy process, including the collection of votes, and the calculation of which option wins the vote. Moreover, a user study could provide insight into how people delegate, and how easy it is for them to grasp and engage with fractional delegation, which is an open and critical question when it comes to evaluating how feasible, effective, and democratic Liquid Democracy with fractional delegation is.

List of Figures

List of Tables

Bibliography

[1] R. Becker, G. D'Angelo, E. Delfaraz, and H. Gilbert. *When Can Liquid Democracy Unveil the Truth?* Apr. 2021. DOI: 10.48550/arXiv.2104.01828. arXiv: 2104.01828 [cs].

[2] S. Becker. "Web Platform Makes Professor Most Powerful Pirate." In: *Spiegel International* (Feb. 2012).

[3] J. Behrens. *Circular Delegations – Myth or Disaster?* Jan. 2015.

[4] F. M. Bersetche. *Generalizing Liquid Democracy to Multi-Agent Delegation: A Voting Power Measure and Equilibrium Analysis*. 2022. DOI: 10.48550/ARXIV.2209.14128.

[5] P. Boldi, F. Bonchi, C. Castillo, and S. Vigna. "Viscous Democracy for Social Networks." In: *Communications of the ACM* 54.6 (June 2011), pp. 129–137. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/1953122.1953154.

[6] M. Brill. "Interactive Democracy." In: *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*. Stockholm, Sweden: International Foundation for Autonomous Agents and Multiagent Systems, 2018, pp. 1183–1187.

[7] R. Burden and J. Faires. *Numerical Analysis*. Brooks/Cole, Cengage Learning, 2011. ISBN: 978-0-538-73564-3.

[8] I. Caragiannis and E. Micha. "A Contribution to the Critique of Liquid Democracy." In: *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence*. Macao, China: International Joint Conferences on Artificial Intelligence Organization, Aug. 2019, pp. 116–122. ISBN: 978-0-9992411-4-1. DOI: 10.24963/ijcai.2019/17.

[9] D. Chakrabarti, Y. Zhan, and C. Faloutsos. "R-MAT: A Recursive Model for Graph Mining." In: *Proceedings of the 2004 SIAM International Conference on Data Mining*. Society for Industrial and Applied Mathematics, Apr. 2004, pp. 442–446. ISBN: 978-0-89871-568-2 978-1-61197-274-0. DOI: 10.1137/1.9781611972740.43.

[10] J. Degrave. *Resolving Multi-Proxy Transitive Vote Delegation*. Dec. 2014. DOI: 10.48550/arXiv.1412.4039. arXiv: 1412.4039 [cs].

[11] B. Ford. *Delegative Democracy*. May 2002.

[12] J. Forrest and R. Lougee-Heimer. *CBC User Guide*. 2005.

[13] J. Forrest, T. Ralphs, S. Vigerske, H. G. Santos, L. Hafer, B. Kristjansson, jpfasano, EdwinStraver, Jan-Willem, M. Lubin, rlougee, a-andre, jpgoncal1, S. Brito, h-i-gassmann, Cristina, M. Saltzman, tosttost, B. Pitrus, F. MATSUSHIMA, P. Vossler, R. @. SWGY, and to-st. *Coin-or/Cbc: Release Releases/2.10.12*. Zenodo. Aug. 2024. DOI: 10.5281/ZENODO.2720283.

[14] P. GÖlz, A. Kahng, S. Mackenzie, and A. D. Procaccia. "The Fluid Mechanics of Liquid Democracy." In: *ACM Transactions on Economics and Computation* 9.4 (Dec. 2021), pp. 1–39. ISSN: 2167-8375, 2167-8383. DOI: 10.1145/3485012.

[15] A. A. Hagberg, D. A. Schult, and P. J. Swart. "Exploring Network Structure, Dynamics, and Function Using NetworkX." In: *Python in Science Conference*. Pasadena, California, June 2008, pp. 11–15. DOI: 10.25080/TCWV9851.

[16] C. C. Kling, J. Kunegis, H. Hartmann, M. Strohmaier, and S. Staab. *Voting Behaviour and Power in Online Democracy: A Study of LiquidFeedback in Germany's Pirate Party*. Mar. 2015. DOI: 10.48550/arXiv.1503.07723. arXiv: 1503.07723 [cs].

[17] G. Kotsialou and L. Riley. *Incentivising Participation in Liquid Democracy with Breadth-First Delegation*. Feb. 2019. DOI: 10.48550/arXiv.1811.03710. arXiv: 1811.03710 [econ].

[18] S. Kumar, B. Hooi, D. Makhija, M. Kumar, C. Faloutsos, and Subrahmanian. "Rev2: Fraudulent User Prediction in Rating Platforms." In: *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*. ACM, 2018, pp. 333–341.

[19] S. Kumar, F. Spezzano, Subrahmanian, and C. Faloutsos. "Edge Weight Prediction in Weighted Signed Networks." In: *Data Mining (ICDM), 2016 IEEE 16th International Conference On*. IEEE, 2016, pp. 221–230.

[20] J. Kunegis, A. Lommatzsch, and C. Bauckhage. "The Slashdot Zoo: Mining a Social Network with Negative Edges." In: *Proc. Int. World Wide Web Conf.* 2009, pp. 741–750.

[21] X. Li, J. Demmel, J. Gilbert, L. Grigori, M. Shao, and I. Yamazaki. *SuperLU Users' Guide*. Tech Report LBNL-44289. Lawrence Berkeley National Laboratory, Sept. 1999.

[22] W. Nils. *Electric.Vote*. Last updated 17 Jan 2020.

[23] M. O'Sullivan, S. Mitchell, and I. Dunning. *PuLP : A Linear Programming Toolkit for Python*. 2011.

[24] A. Paulin. "An Overview of Ten Years of Liquid Democracy Research." In: *The 21st Annual International Conference on Digital Government Research*. Seoul Republic of Korea: ACM, June 2020, pp. 116–121. ISBN: 978-1-4503-8791-0. DOI: 10.1145/3396956.3396963.

[25] Python Software Foundation. *The Python Tutorial, Section 5: Data Structures*.

[26] M. Revel, D. Halpern, A. Berinsky, and A. Jadbabaie. "Liquid Democracy in Practice: An Empirical Analysis of Its Epistemic Performance." In: *ACM Conference on Equity and Access in Algorithms, Mechanisms, and Optimization*. 2022.

[27] M. Richardson, R. Agrawal, and P. Domingos. "Trust Management for the Semantic Web." In: *The Semantic Web - ISWC 2003*. Ed. by G. Goos, J. Hartmanis, J. Van Leeuwen, D. Fensel, K. Sycara, and J. Mylopoulos. Vol. 2870. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 351–368. ISBN: 978-3-540-20362-9 978-3-540-39718-2.

[28] M. Utke and U. Schmidt-Kraepelin. "Anonymous and Copy-Robust Delegations for Liquid Democracy." In: *Advances in Neural Information Processing Systems*. Ed. by A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine. Vol. 36. Curran Associates, Inc., 2023, pp. 69441–69463.

[29] P. Virtanen, R. Gommers, T. E. Oliphant, et al. "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python." In: *Nature Methods* 17.3 (Mar. 2020), pp. 261–272. ISSN: 1548-7091, 1548-7105. DOI: 10.1038/s41592-019-0686-2.

[30] D. J. Watts and S. H. Strogatz. "Collective Dynamics of 'Small-World' Networks." In: *Nature* 393.6684 (June 1998), pp. 440–442. ISSN: 1476-4687. DOI: 10.1038/30918.

[31] B. Zhou and J. Pei. "The K-Anonymity and l-Diversity Approaches for Privacy Preservation in Social Networks against Neighborhood Attacks." In: *Knowledge and Information Systems* 28.1 (July 2011), pp. 47–77. ISSN: 0219-1377, 0219-3116. DOI: 10.1007/s10115-010-0311-2.

A. Appendix

A.1. Algorithms

A.1.1. Resolution Algorithms

Linear Systems Solver: `https://github.com/davidholzwarth/bachelors-thesis/blob/main/LE.py`

Linear Programming Solver: `https://github.com/davidholzwarth/bachelors-thesis/blob/main/LP.py`

Iterative Solver: `https://github.com/davidholzwarth/bachelors-thesis/blob/main/iterative.py`

A.1.2. Graph Generator

Random Delegation Graph Generator: `https://github.com/davidholzwarth/bachelors-thesis/blob/main/graph_gen.py`

A.1.3. Preprocessing

The algorithm for removing closed delegation cycles is in step five of the pipeline at the following link.

`https://github.com/davidholzwarth/bachelors-thesis/blob/main/graph_tools.py`

A.2. Results

The runtimes for the benchmarks for these graphs can be found at the following link. The files follow the following naming scheme: "n-m_type", where n is the lower bound of nodes in the benchmark, m is the upper bound of nodes, type is the type code of graph this benchmark was run on. Each subsection of this annex will detail what type code is used for graphs in this section. For example, dense graphs have type code `dense`, so the runtimes for such a dense graph on graphs containing between zero and 2000 nodes will be: `0-2000_dense`

`https://github.com/davidholzwarth/bachelors-thesis/tree/main/data`

A.2.1. Small Graphs

Type code: `random`
Other statistics and the code to generate the graph can be found at this link:
`https://github.com/davidholzwarth/bachelors-thesis/blob/main/Benchmarking/runtime_exploration.ipynb`

A.2.2. Large Graphs

Type code: `random`
Other statistics and the code to generate the graph can be found at this link:
`https://github.com/davidholzwarth/bachelors-thesis/blob/main/Benchmarking/runtime_exploration.ipynb`

A.2.3. Dense Graphs

Type code: `dense`
Other statistics and the code to generate the graph can be found at this link:
`https://github.com/davidholzwarth/bachelors-thesis/blob/main/Benchmarking/dense_graphs.ipynb`

A.2.4. Cycles Which Retain a Lot of their Power

Type code: `big_loop`
Other statistics and the code to generate the graph can be found at this link:
`https://github.com/davidholzwarth/bachelors-thesis/blob/main/Benchmarking/big_loop.ipynb`

A.2.5. No Delegations

Type code: `no_del`
Other statistics and the code to generate the graph can be found at this link:
`https://github.com/davidholzwarth/bachelors-thesis/blob/main/Benchmarking/no_edges.ipynb`

A.2.6. Watts-Strogatz Small World Graphs

Type code: `small_world`
Other statistics and the code to generate the graph can be found at this link:
`https://github.com/davidholzwarth/bachelors-thesis/blob/main/Benchmarking/small_world.ipynb`

A.2.7. R-Mat Graphs

Type code: `rmat`

Other statistics and the code to generate the graph can be found at this link:
`https://github.com/davidholzwarth/bachelors-thesis/blob/main/Benchmarking/r-mat.ipynb`

A.2.8. Bitcoin OTC Trust Network

The runtimes can be found here:
`https://github.com/davidholzwarth/bachelors-thesis/blob/main/data/bitcoinotc_dataset.txt`

The code to prepare this graph and its analysis can be found here:
`https://github.com/davidholzwarth/bachelors-thesis/blob/main/Benchmarking/bitcoin_otc_trust_dataset.ipynb`

A.2.9. Epinions

The runtimes can be found here:
`https://github.com/davidholzwarth/bachelors-thesis/blob/main/data/epinions_dataset.txt`

The code to prepare this graph and its analysis can be found here:
`https://github.com/davidholzwarth/bachelors-thesis/blob/main/Benchmarking/epinions_dataset.ipynb`

A.2.10. Slashdot Zoo

The runtimes can be found here:
`https://github.com/davidholzwarth/bachelors-thesis/blob/main/data/slashdot_dataset.txt`

The code to prepare this graph and its analysis can be found here:
`https://github.com/davidholzwarth/bachelors-thesis/blob/main/Benchmarking/slashdot_zoo.ipynb`