SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Thesis title

David Nicolaus Matthäus Holzwarth

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Thesis title

Titel der Abschlussarbeit

Author:	David Nicolaus Matthäus Holzwarth
Examiner:	Prof. Dr. Pramod Bhatotia
Supervisor:	Prof. Dr. Bryan Ford
Submission Date:	Submission date

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

Munich, Submission date David Nicolaus Matthäus Holzwarth

Acknowledgments

Abstract

Contents

Acknowledgments **iii**

Abstract **iv**

1 Introduction **1**
- 1.1 Section 1
  - 1.1.1 Subsection 1

2 Introduction **3**

3 Background **4**
- 3.1 Liquid Democracy 4
  - 3.1.1 Motivation 4
  - 3.1.2 Challenges 4
  - 3.1.3 Applications 5
- 3.2 Fractional Delegation 5
  - 3.2.1 Motivation 5
  - 3.2.2 Existing Methods to Deal with Vote Concentration 6
  - 3.2.3 Other Works on Fractional Delegation 6

4 Design **8**
- 4.1 Problem Statement 8
- 4.2 Implementing Liquid Democracy with Fractional Delegation 9
  - 4.2.1 Definitions 9
  - 4.2.2 Conservation of Power 9
  - 4.2.3 Closed Delegation Cycles 10
- 4.3 Resolving Delegations 11
  - 4.3.1 Existence of a Unique Solution 12
  - 4.3.2 Conservation of Power 12
  - 4.3.3 Resolving Delegations by Solving a System of Linear Equations 13

5 Implementation **14**
- 5.1 Linear System Solver (LS) 14

5.2 Linear Programing Solver (LP) 15
5.3 Iterative Solver (Iterative) 16
5.3.1 Approach 16
5.3.2 Conservation of Power 17
5.4 Robustness 23
5.4.1 Invalid delegations 23
5.4.2 Closed Delegation Cycles 24

6 Evaluation **25**
6.1 Method 25
6.1.1 Generating Random Delegation Graphs 25
6.1.2 Preprocessing 26
6.1.3 Measurement 27
6.2 Synthetic Graphs 27
6.2.1 Small Graphs 27
6.2.2 Large Graphs 29
6.2.3 Dense Graphs 30
6.2.4 Cycles Which Retain a Lot of their Power 31
6.3 Social Graphs 33
6.3.1 Small World Graphs 33
6.3.2 R-Mat Graphs 34
6.4 Real-World Datasets 34
6.4.1 Epinions 35
6.4.2 Bitcoin OTC Trust Network 36
6.4.3 Slashdot Zoo 37
6.4.4 Evaluation of the datasets 37

7 Related Work **39**

8 Conclusion **40**

Abbreviations **41**

List of Figures **42**

List of Tables **44**

Bibliography **45**

1 Introduction

1.1 Section

Citation test [**latex**].

Acronyms must be added in `main.tex` and are referenced using macros. The first occurrence is automatically replaced with the long version of the acronym, while all subsequent usages use the abbreviation.

E.g. `\ac{TUM}, \ac{TUM}` ⇒ cro:TUMTechnical University of Munich (TUM), TUM

For more details, see the documentation of the `acronym` package[1].

1.1.1 Subsection

See Table 1.1, Figure 1.1, Figure 1.2, Figure 1.3.

Table 1.1: An example for a simple table.

A	B	C	D
1	2	1	2
2	3	2	3

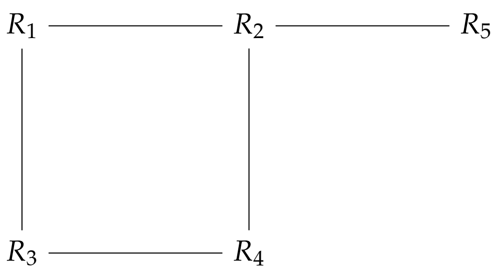


Figure 1.1: An example for a simple drawing.

[1]`https://ctan.org/pkg/acronym`

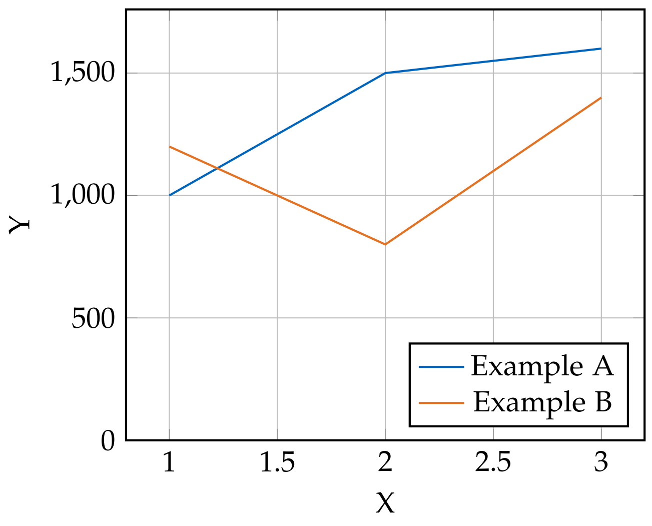


Figure 1.2: An example for a simple plot.

```
SELECT * FROM tbl WHERE tbl.str = "str"
```

Figure 1.3: An example for a source code listing.

2 Introduction

The introduction is a longer writeup that gently eases the reader into your thesis [**dinesh20oakland**]. Use the first paragraph to discuss the setting.

In the second paragraph you can introduce the main challenge that you see.

The third paragraph lists why related work is insufficient.

The fourth and fifth paragraphs discuss your approach and why it is needed.

The sixth paragraph will introduce your thesis statement. Think how you can distill the essence of your thesis into a single sentence.

The seventh paragraph will highlight some of your results

The eights paragraph discusses your core contribution.

This section is usually 3-5 pages.

Also include a description of how the paper is set up.

3 Background

3.1 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 delegation. [Deg14; Bol+11; Rev+22; Ber22]

3.1.1 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 characteristic of representative democracy. [**fordDelegativeDemocracy2002**; Bri18] 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, effectively flattening hierarchies. Furthermore, it allows delegations to be updated or revoked at any time, restoring agency to the voter. [Bri18] In doing so, liquid democracy emerges as a promising hybrid model—offering a flexible middle ground between participatory engagement and representational practicality.

3.1.2 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 paragraph XX. 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.

3.1.3 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. [Pau20] Throughout the period 2010 - 2013, 499,009 votes on 6,517 topics were cast, with pirate party members having made 14,964 delegations. [Kli+15] 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 socialcorporate network - and the Partido de la Red, an Argentinian political party. [Pau20]

3.2 Fractional Delegation

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

3.2.1 Motivation

Classic liquid democracies, 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. [Kli+15; CM19; Bec+21] 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". [**beckerWebPlatformMakes2012**] 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 citizens may 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. [Beh15] 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. This reduces the number of votes lost within the system.

3.2.2 Existing Methods to Deal with Vote Concentration

The problem of vote concentration has been addressed in literature.

Partly in response to this problem, Boldi et al. propose introducing a damping factor into the delegation process: the further a vote is delegated, the weaker it becomes. [Bol+11] This approach 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.

Gölz et al. and Kotsialou & Riley take a different approach. They 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. [KR19; GÖl+21] Even in this approach, however, each delegator ultimately entrusts their vote to only one delegate.

3.2.3 Other Works on Fractional Delegation

The idea of allowing the fractional splitting of votes in Liquid Democracy has been introduced works. Degrave first proposes so-called "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. [Deg14] 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.

Bertsche revisits and extends this idea under the term multi-agent delegation. [Ber22] 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 find, that delegation graphs allowing fractional delegation allow for an "equilibrium state", there is a collection of delegations so that no agent can unilaterally change their delegations to increase their voting power.

4 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.

4.1 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.

We pose the following problem.

1. **Given** a set of voters and their delegations, where each voter has one vote, and either:
   a) votes directly, or
   b) delegates their vote such that their vote is delegated to other delegates 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 delegates to them, including transitive delegations, otherwise
   c) the sum of the final power of all nodes must be equal to the amount of votes initially in the graph

4.2 Implementing Liquid Democracy with Fractional Delegation

4.2.1 Definitions

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

As per the problem statement, voters are strictly given the choice to either delegate their vote (fractional) in its entirety, or vote directly. The electorate is thus divided into two disjoint **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 $e \in E$ 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 [1], such that the sum of all of its outgoing edge weights equals 1.
5. Each voter $v \in V$ has corresponding power value, which is initially 1.

4.2.2 Conservation of Power

A vital property we set for the delegation graph is the conservation of power. While some authors have experimented with implementations of liquid democracy where this is not the case [Ber22; Bol+11], we believe that for a system to be truly democratic, we must assert delegating is not penalised, 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. Thus, any implementation needs a mechanism to ensure that the sum of the final power of all sinks is equal to the sum of the initial power of all nodes.

[1]Edge weights should not be 0, since the edge should be entirely omitted in this case.

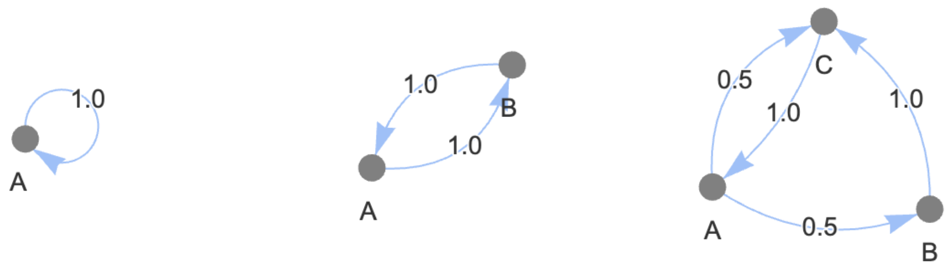


Figure 4.1: Closed delegation cycles

4.2.3 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. That is,

$$\forall v \in C, \nexists \text{ path from } v \text{ to any } s \in S.$$

Figure 4.1 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. [Beh15; Bri18] 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. Our approach to this is to find all such cycles, and collapse them into an additional sink node in the graph, the **cycle sink node**. Any delegation into the cycle is redirected into the cycle sink node, thus ensuring the graph no longer has any closed delegation cycles. The algorithm to do so can be found in annex XX. *TODO: Add this into the annex, if that necessary...*

We prove below, that given the absence of such closed delegation cycles, delegations are resolvable given 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 $\exists d \in D$ 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

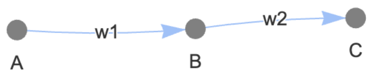


Figure 4.2: Sample delegations

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. $\lightning$ $\square$

A **well-formed delegation graph** G is defined as a delegation graph, which contains no closed delegation cycles. Note, that while a self loop of weight one is not allowed in a well-formed delegation graph, a self loop of weight $w \leq 1$ is allowed as long as the rest of the node's power eventually flows to a sink. Since a delegator can't vote, any power a delegator delegates to themselves will "flow" back into the node, and then be redistributed to the nodes delegates.

4.3 Resolving Delegations

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 andp'_B denote the **standing power** of nodes A and B, i.e. the total amount of power delegated to them. Then, the final power of C, assuming no other incoming delegations, is:

$$p_C = 1 + w_2 p'_B = 1 + w_2(1 + w_1 p'_A)$$

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, we define the final voting power p_v of a node as:

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

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 can prove, that given a well-formed delegation graph, this method returns a unique solution, and that given a well formed delegation graph and the above definition of the final power of a node, power is conserved.
1. power is conserved

2. there is a unique solution when the graph is well formed?

4.3.1 Existence of a Unique Solution

TODO: There are two ways I can prove this. Firstly is to just quote (p.74-76). Alternatively, a more complicated but less weird source would be to prove this shit using (theore 8.1.26 and some other theorem, see chatgpt Chat "Trace behavior of matrices"). the second one I am not 100% convinced yet that it actually works

first link `file:///Users/DavidHolzwarth/Downloads/978-1-84996-299-5.pdf` (Max-Linear Systems: Theory and Algorithms) by Butkovic

second link `https://www.anandinstitute.org/pdf/Roger_A.Horn.%20_Matrix_Analysis_2nd_edition(BookSee.org).pdf`

4.3.2 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 strictly necessary, as the other equations in the system of equations already imply the conservation of power.

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

$$\begin{aligned}
\forall v \in V : p'_v &= 1 + \sum_{(u,v,w) \in E} (wp'_u) \\
\implies \sum_{v \in V} p'_v &= \sum_{v \in V} (1 + \sum_{(u,v,w) \in E} (wp'_u)) \\
&= \sum_{v \in V} 1 + \sum_{v \in V} \sum_{(u,v,w) \in E} wp'_u \\
&= |V| + \sum_{v \in V} \sum_{(u,v,w) \in E} wp'_u \\
&= |V| + \sum_{(u,v,w) \in E} wp'_u
\end{aligned}$$

Focusing on the $\sum_{(u,v,w) \in E} wp'_u$ term, this can be re-grouped by u as follows

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

Since, 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, we know that

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

Thus, we can rewrite the above equation.

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

Focusing now on term $\sum_{v\in V} p'_v$, since $V = S \bigcup D$, and S and D are disjunct we can say

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

Thus, these two equations, we get

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

4.3.3 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 $p'_v = 1 + \sum_{(u,v,w)\in E} wp'_u$
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 $p_d = 0$

5 Implementation

The 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. They will be explored and evaluated in section XX. Specifically, 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, and offers a large collection of helpful libraries like NetworkX, a library for working with graphs [HSS08].

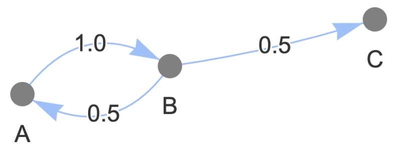
The algorithms take as input python dictionaries ("dicts"), as inputs, which map a key to a value [Pyt]. The delegation graph is represented in a "dict of dicts" format, where every key in the outer dict is a node, and the value is another dict, which has the node's neighbors as keys, and a weight as value. The algorithm's use as input "inverse" delegation graphs, where the inner dictionaries represent a node's incoming delegations rather than its outgoing edges. Figure 5.1 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 learn about all its incoming delegations.

5.1 Linear System Solver (LS)

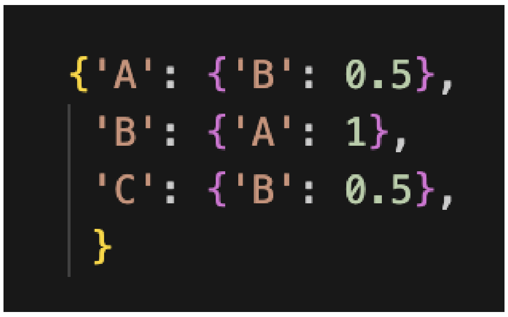
The first approach uses a dedicated linear system solver. We use SciPy's `scipy.sparse.linalg.spsolve` solver, which is optimized for sparse matrices [Vir+20]. A sparse solver is better equipped to resolve delegations if we assume that realistically each delegator only delegates to a few delegates. Since each wp_v term in the system of linear equations can be mapped to one unique edge $(u, v, p) \in E$, the matrix corresponding to the system of linear equations likely has relatively few non zero entries compared to its size. In other words:

$$\text{For a small } n \in \mathbb{N}\text{, and a large } |V|, D \subset V : n \cdot |D| < |V| \cdot |V|$$

TODO: This explanation is ugly, but idk if its worth spending a lot more time on...



(a) Delegation graph



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

(b) Inverse dict representation

Figure 5.1: Delegation graph and its inverse dict representation

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 specified otherwise is 0 [Vir+20].

TODO: This sentence below abt the perfect accuracy makes no sense atm, change it to fit into this section The solver solves the system of linear equations directly, using the SuperLU solver, so it is not possible to trade off runtime for accuracy, as done in the previous methods [Li+99].

5.2 Linear Programing Solver (LP)

Secondly, we use the Python library PuLP, which provides an interface to Linear Programming (LP) solvers [OMD11]. To resolve the delegation graphs, we use the "Coin-or branch and cut" (CBC) solver, since it is free and open-source [For+24]. Given the academic context and the moderate size of our delegation graphs, CBC provides a practical balance between performance and accessibility. Moreover, since our model essentially solves a system of linear equations with a unique correct 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.

TODO: The mention below abt the perfect accuracy makes no sense atm, since we have not introduced the iterative solver yet, change it to fit into this section 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 does not have perfect accuracy either. A tolerance of $5 * 10^{-3}$ assures that $|p'_v - \sum_{(u,v,w) \in E} 1 + wp'_u| \leq 5 * 10^{-3}$, so the solutions will be correct when rounded to the second decimal place [FL05]. Finally, the algorithm cleans the p'_v values, setting any delegators power to 0.

TODO: Add links to the implementations on GitHub

5.3 Iterative Solver (Iterative)

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

5.3.1 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.

$G = (V, E)$ is a well-formed delegation graph, with $V = D \dot{\bigcup} S$, where D is the set of delegators and S is the set of sinks.

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

[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 $v \in$ nodes do
3:      powers[$v$] $\leftarrow$ 1.0
4:  end for
5:  repeat
6:      prev_powers $\leftarrow$ powers.copy()                    ▷ snapshot of previous iteration
7:      for all $v \in$ nodes do
8:          // For each incoming delegation $(u \to v)$, move $w_{uv} \times$ previous power of u
9:          for all $(u, w) \in$ delegations[$v$] do
10:             $\delta \leftarrow w \times$ prev_powers[$u$]
11:             powers[$u$] $-= \delta$
12:             powers[$v$] $+= \delta$
13:         end for
14:     end for
15: until prev_powers = powers                                  ▷ a steady state has been reached
```

this notation, our termination condition for the repeat-until loop at algorithm 1 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 all delegators and all sinks after each iteration.

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

5.3.2 Conservation of Power

We show, that this algorithm conserves power throughout iterations. This insight .... XXX TODO: here

Theorem 2. *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

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

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 in algorithm 1, but add this same amount to node v in algorithm 1. 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.

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

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. This means that power is only ever moved around, but not lost, and $P^{(k+1)}_t = |V|$.

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

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. Algorithm starts with a vector of ones, indicating an initial power of each node of one. Furthermore, each iteration, each node $u \in V$ gains power amounting to $\sum_{(u,v,w)\in E} \delta^{(i)}_{(u,v,w)}$. This term can be rearranged as follows:

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

Since power is conserved, the same amount is also removed from the node, thus we can say, that

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

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

TODO: This section may still be missing an explanation of the jacobi method. I proved that each iteration I resolve something resembling the initial system of linear equations method, but since the reader does not know the jacobi method, they might not be able to notice the similarity.

Implementation

This section will show, that the algorithm does not necessarily terminate despite input with a well formed delegation graph, after which we propose an amended algorithm.

First, we prove that a sink's power can't shrink, since there is no outgoing edge going out of a sink.

Lemma 3. $\forall s \in S : p_s^{(i)} \geq p_s^{(i-1)}$

Proof. Assume $p_s^{(i)} < p_s^{(i-1)}$. The power that left s needs to have gone somewhere, since the algorithm conserves power. This implies, that there is a delegation $(s, v, w) \in E$ such that $\exists \delta > 0 : \delta \leftarrow w * p_s^{(i-1)}$. This contradicts our definition of a well-formed liquid delegation graph, since any sink can not have any outgoing edges. □

Next, we prove that if a node's power is 0, all its delegator's powers must have been 0 after the previous iteration

Lemma 4. $p_v^{(i)} = 0 \implies \forall (u, v, w) \in E : p_u^{(i-1)} = 0.$

Proof. Assume $p_v^{(i)} = 0$, but $\exists (u, v, w) \in E : p_u^{(i-1)} > 0$

Let $d \in \mathbb{R}_0$ be any additional power a node receives, that is not explicitly mentioned.

$$p_u^{(i-1)} > 0 \implies \delta_{(u,v,w)}^{(i)} > 0 \implies p_v^{(i)} = \delta_{(u,v,w)}^{(i)} + d \implies p_v^{(i)} > 0 \lightning$$

□

Next, we 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.

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

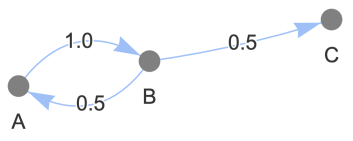


Figure 5.2: Delegation graph with a cycle.

□

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 $D = \{A, B\}$. B delegates half their vote to A, and half their vote to C, while A delegates its entire vote back to B. Since the algorithm terminates, there must be an $i \in \mathbb{N}$ such that $P_D^{(i)} = 0$.

Both (B, A, 0.5) and (A, B, 1) $\in E$, so A is a predecessor of B, and B is a predecessor of A, thus B is its own predecessor.

$$\begin{aligned} P_D^{(i)} = 0 &\implies p_B^{(i)} = 0 \\ &\implies p_A^{(i-1)} = 0 && \text{(Lemma theorem 4)} \\ &\implies p_B^{(i-2)} = 0 && \text{(Lemma theorem 4)} \end{aligned}$$

This implication chain can be drawn arbitrarily long. In order for B to have a power of 0, it can never have held any power in the first place, contradicting our well-formed delegation graph, which dictates that all nodes start with a power of 1. □

If the graph is acyclic, this algorithm would need at most $|V|$ iterations, such as in a directed path, where each node gives their entire vote to the next node except for the final sink. However, as soon as cycles are introduced into the graph, the spreading of power only terminates after an infinite amount of steps. Looking further into the graph in fig. 5.2, the expected resolution of these delegations would be that C holds node A and B's powers, as well as its own initial vote, so a power of three. However, looking at the powers as the algorithm iterates over this graph reveals that after the first iteration, C will have 1.5 votes, then 2, then 2.25, 2,50, 2.75, ..., however only after infinitely many steps it will have three.

Table 5.1: $p_v(i)$ values of nodes in the graph in fig. 5.2

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 while 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.

Conservation of Power

Theorem 2 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 `cutoff` > 0.*

TODO: The proof is incomplete, I'll work over it later...

Proof. We differentiate two cases. Fix an $i \in \mathbb{N}_0$.

Case 1: $P_D^{(i)} = 0$

In this case, no delegator has any power, so the `total_change` can be at most 0, which is always smaller than `cutoff`. So the algorithm terminates.

Case 2: $P_D^{(i)} > 0$

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

```
// Initialize each node's power to 1.0
for all $v \in$ nodes do
    powers[$v$] $\leftarrow$ 1.0
end for
repeat
    prev_powers $\leftarrow$ powers.copy()
    total_change $\leftarrow$ 0
    for all $v \in$ nodes do
        // For each incoming delegation $(u \to v)$, move $w_{uv} \times$ previous power of u
        for all $(u, w) \in$ delegations[$v$] do
            $\delta \leftarrow w \times$ prev_powers[$u$]
            powers[$v$] $+= \delta$
            powers[$u$] $-= \delta$
            total_change $+= \delta$
        end for
    end for
until total_change < cutoff
```

$$P_D^{(i)} > 0 \implies \exists d_0 \in D : p_{d_0}^{(i)} > 0$$

$$\implies \exists s \in S, \exists k \geq 1, \exists (v_0, ..., v_k) : v_0 = d_0, v_k = s, (v_j, v_{(j+1)}, w) \in E \forall 0 \leq j \leq k$$

(There is a path between d_0 and a sink)

$$\implies \exists d_{k-1} \in D : (d_{k-1}, s, w) \in E$$

(d_{k-1} is the node on the path from d_0 to s that has an edge to s)

Assume ${p_{d_{k-1}}}^{(i)} = 0$.
Left $\text{Pred}^*(d_0)$ be defined as follows:

$$\text{Pred}^*(d_0) = \{u \in V \mid \exists \text{ a path } u \rightsquigarrow d_0\}$$

Also, let $\text{dist}(d_0, p), p \in V$ be defined as follows :

$$\text{dist}(d_0, p) = \min\{|P| : P \text{ is a path } d_0 \rightsquigarrow p\}.$$

$${p_{d_{k-1}}}^{(i)} = 0 \implies$$

$\max\{\text{dist}(d_0, p) \mid p \in \text{Pred}^*(d_0)\}$
...

TODO: todo continue here, I am too lazy atm to try and figure out how to prove that P_D shrinks properly

$\square$

5.4 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.

5.4.1 Invalid delegations

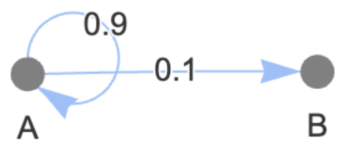
On their own, neither of the three implementations will definitively cause an error when delegations are invalid. The iterative implementation is "dumb", in the sense that it moves around power as it finds them in the delegations. If a delegator delegates more than they are meant to, the algorithm behavior 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 behavior becomes quite predictable. 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. 5.3a. 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.

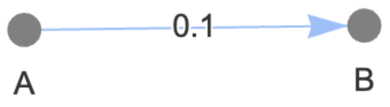
For the two implementations directly based on solving systems of linear equations, this is a little different. Node `B`'s power would end up as 1.1, since the system of linear equations looks as follows:

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

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 5.3: Two similar delegation graphs

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

5.4.2 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 don't have a single solution. Solvers of systems of linear equations catch 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 is a closed delegation cycle does also delegates to a node outside of a delegation cycle (or turns into a sink), for the delegations to become resolvable again. The cycles that were explored in section 6.2.4 are example of such a situation.

6 Evaluation

TODO: Fix and improve all graphs, especially title and labels...

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

6.1 Method

6.1.1 Generating Random Delegation Graphs

For the first section, we built an algorithm, that builds a graph with n nodes, and then adds between zero and three delegations per node to random other nodes, ensuring that there are no delegation cycles without a sink. A better explanation of how these graphs are artificially constructed can be found in the Annex. *TODO: Create this annex*
We acknowledge, that these assumptions may not be representative of real delegation graphs, where, as studies have shown, delegates tend to not delegate randomly, but to a subset of experts, such as TV personalities, thus centering power to one or few people. This approach also overlooks potential tendencies of voters, such as delegating to individuals they perceive as more competent or confident than them, or the tendency of power to concentrate among a few, very popular so-called "super-voters" [Kli+15]. These concerns are addressed in section XX, when we benchmark delegation graphs that are based on social graphs.

6.1.2 Preprocessing

In order to be able to benchmark algorithms that resolve delegations, the input graphs need to be well-formed delegation graphs. In section XX and YY, we use graphs which 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. 6.1.

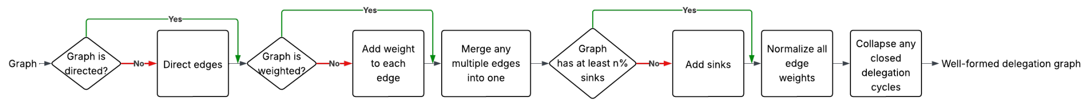

Graph
Graph is directed?
Yes
No
Direct edges
Graph is weighted?
Yes
No
Add weight to each edge
Merge any multiple edges into one
Graph has at least n% sinks
Yes
No
Add sinks
Normalize all edge weights
Collapse any closed delegation cycles
Well-formed delegation graph

Figure 6.1: Process to clean 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 in the shape (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. By default this n value is 20, however depending on the use case it can be increased or decreased. After this, the algorithm searched for any closed delegation cycles, and collapses all it finds into a single sink node. Specifically, the algorithm searched 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 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 a specially created node. 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.

6.1.3 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 solver 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 data format—when choosing between different approaches or implementations. The

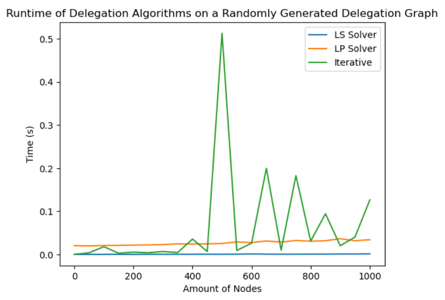


Figure 6.2: Runtime of delegation algorithms on a randomly generated delegation graph.

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, such as the time for building the linear program, were not included.

6.2 Synthetic Graphs

6.2.1 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 6.2 shows the results of this benchmark.

We can 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 0.01 seconds. Both the LS and LP implementation 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.

Furthermore, we can interestingly observe large spikes in the runtime of the iterative approach. Exploring this more closely, we find that the graph with 11 nodes takes the iterative algorithm a lot more time than the graph with 10 or 12 nodes, as shown in fig. 6.3. At 10 nodes, the runtime of the iterative algorithm is just about 0.004 seconds, at 12 nodes it is 0.001 seconds, so even slightly faster than the slightly smaller graph, but when the graph has 11 nodes, the runtime skyrockets to about 0.056 seconds.

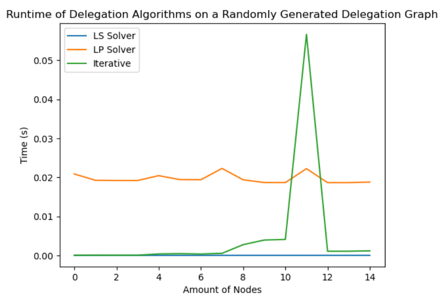
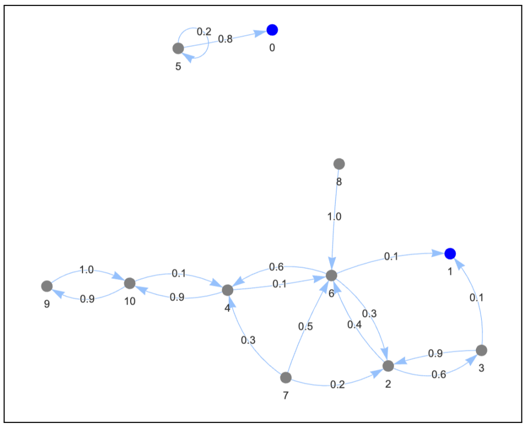
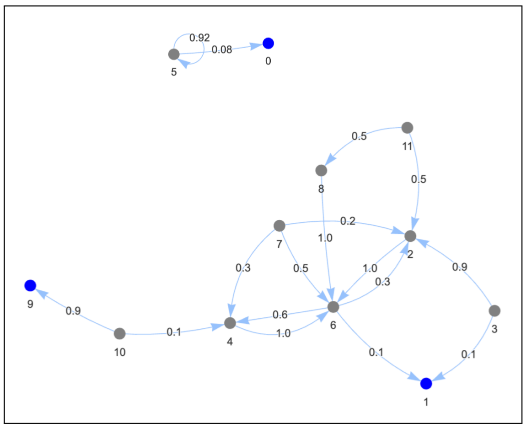


Figure 6.3: Runtime of delegation algorithms on a randomly generated delegation graph.



(a) 11 nodes



(b) 12 nodes

Figure 6.4: Delegation graphs with 11 and 12 nodes (Blue nodes are sinks)

A possible explanation for this spike may be, that when the graph has 10 and 12 nodes, it iterates only 758 and 170 times respectively, before cutting off, while when it has 11 nodes it iterates 8735 times before cutting off. Figure 6.4 shows the two graphs with 11 and 12 nodes.

Inspecting the graphs reveals a possible explanation for this behavior. When the graph has 12 nodes, node `9` is a sink, while in the graph with 11 node it delegates its power back to node `10`. In the latter case, power going out of node `9` needs to pass to node `10`, `4` and `6` before reaching a sink. While passing through node `4`, we can see that 90% of the power is delegated back into the cycle between nodes `4`, `10` and `9`. The algorithm will iterate power through this loop, until enough has been drained out for the `total_change` to fall below the cutoff.

This is an important shortcoming of the iterative algorithm. Power can easily get

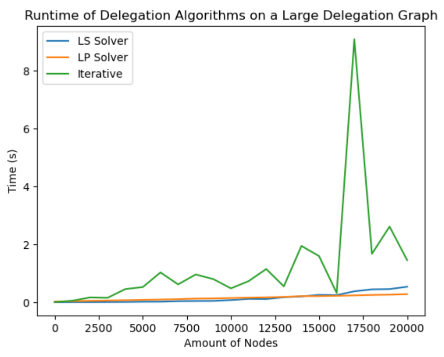


Figure 6.5: Runtime of delegation algorithms on a randomly generated delegation graph.

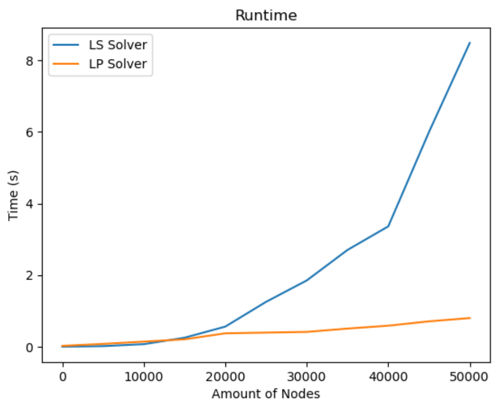
trapped within permissible delegation cycles that only have a small drain allowing the power to escape from the cycle. Each iteration, if a great proportion of the nodes with draining edges' power is sent back into a cycle, the algorithm needs to continuously iterate until the power is back at the drain nodes, however depending on the cycle this may happen very inefficiently. This phenomenon will be tested more in section 6.2.4

6.2.2 Large Graphs

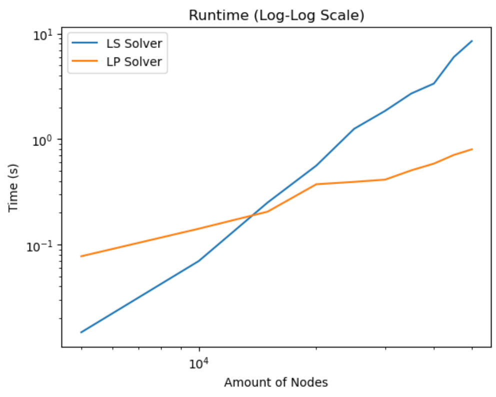
Delegation graphs may grow arbitrarily large. National elections for example can contains up to hundreds of millions of participants. This section explores how the algorithms perform when having to resolve graphs with a lot of nodes. Again, the graphs will be randomly generated, such that each nodes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

In fig. 6.5 we can see, that it is difficult to determine a pattern in the runtime for the iterative algorithm. Depending on the underlying delegation graph, the runtime can grow unpredictably large. What is evident from the runtime graph however, is that in as the graphs get larger its runtime never subceeds the runtimes of the other two algorithms, while it is worth mentioning that for some graphs, the iterative algorithm's runtime is not a lot longer than that of the other two algorithms. It is difficult to make any statement about the runtime class of the iterative algorithm based just on the number of nodes in the graph, since, depending on the structure of the graph and the cutoff value the runtime can get arbitrarily high. The comparatively high runtime of the iterative algorithm overshadows the runtimes of the other two, so in order to better discuss and analyze their performance, fig. 6.7 shows the same graph as in fig. 6.5, without the runtimes for the iterative algorithm.

Figure 6.6a shows, that as the delegation graph grows, the LP solver's runtime grows more slowly than the LS Solver's. For resolving smaller graphs, the LS solver



(a) Linear scale



(b) Loglog scale

Figure 6.6: Runtime of delegation algorithms on a randomly generated delegation graph.

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.

The type of growth, so the runtime class, is not immediately clear from the graphs, although the LP solver's growth seems to be more linear than that of the LS solver. Looking at the same results on a loglog graph reveals, that the LS solvers runtime may follow a power law.

Fitting the data into different kinds of curves reveals, that the LP implementations runtime likely has linear growth, while the LS solver grows following a power distribution, such that it is in the runtime class of $O(n^{2.778})$.

TODO: Put the runtime results and/or the code and the regression results into the annex, or into the text...

6.2.3 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 [HSS08]. These graphs are not well-formed

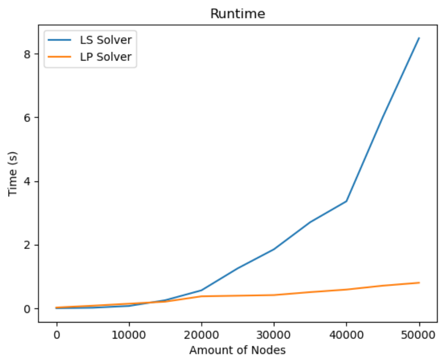


Figure 6.7: Runtime of delegation algorithms on a randomly generated delegation graph.

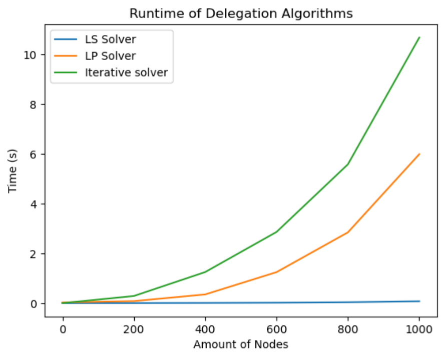


Figure 6.8: Runtime of delegation algorithms on a randomly generated delegation graph.

delegation graphs out-of-the box, thus we adapt them by removing outgoing edges of nodes, turning them into sinks, until 10% of the nodes are sinks. Then, each delegators vote is equally distributed to all of its outgoing edges, such that the edge weights add up to 1. Finally, any closed delegation cycles are removed by removing a random edge in the cycle (and re-normalizing the edge weights).

Figure 6.8 shows the runtime of these three algorithms. The runtime of the iterative algorithm lacks the spikes found when resolving sparse graphs. This is likely due to the nature of the graphs we create. Every delegator is connected to half of all other nodes ($p = 0.5$), and of these 10% are sinks, thus power drains quickly into sinks, and situations where power iterates a long time without seeing a sink are less frequent. Regardless, the iterative algorithm exhibits the worst runtime of the three.

Testing the three algorithms on larger dense graphs, reveals surprisingly, that the LP

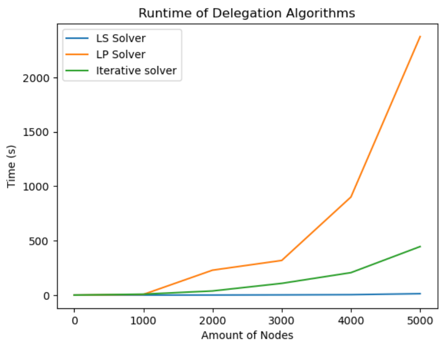


Figure 6.9: Runtime of delegation algorithms on a randomly generated delegation graph.

solver's runtime is considerably worse than that of both the iterative and LS solver. A dense graph with 5,000 nodes, and thus about 125,000 delegations, takes the LS solver only about 12 seconds, the iterative solver 445 seconds, and the LP solver almost 2,400 seconds. Even though the LS solver is optimized for sparse matrices, it outperforms the other two implementations on dense graphs.

6.2.4 Cycles Which Retain a Lot of their Power

To further 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. Specifically, we construct graphs with delegates all delegating 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 go to the first node in the cycle. Figure 6.10 contains an exemplary image of such a graph with 10 nodes.

The runtimes in fig. 6.11a show, that as expected, the iterative algorithm struggles considerably with the resolution of these graphs, while the other two algorithms exhibit 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 are a lot more well equipped to deal with this specific corner case.

As seen in fig. 6.12, which shows the same graph as in fig. 6.11a without the iterative algorithm's runtime, the runtime of the LS and LP Solvers grows similarly to when resolving randomly generated (sparse) delegation graphs, such as the ones found in

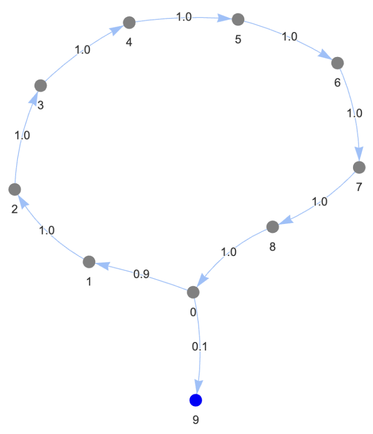


Figure 6.10: An example of the cycles used for the benchmarks. The blue node is the sink

section 6.2.1.

Such a cycle as the one we deliberately constructed is not the only situation in which the iterative algorithm will struggle. As long as power is not efficiently funneled toward a sink, the iterative algorithm will have to spend more time moving the power around until enough has drained into a sink for `total_change` to fall below the cutoff value. A further example of such behavior is the cycle that caused runtime to spike in fig. 6.4a. An artificial example of this situation is shown below in fig. 6.10.

6.3 Social Graphs

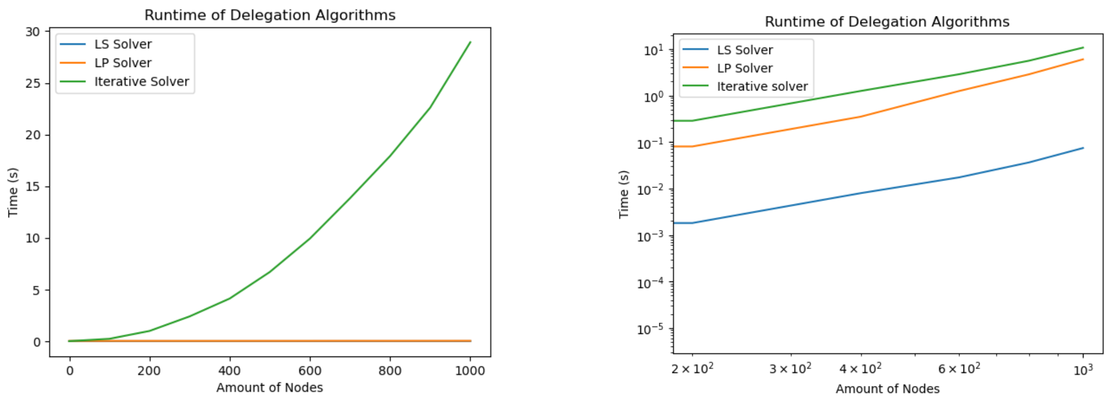
Social graphs provide an excellent way to *TODO: Finish this sentence*
We will use them to create sample delegation graphs based on social behaviors, which can predict ways humans may delegate if given the chance to delegate fractionally. These graphs are still only based on models, however since they are artificial, we can scale them and explore how the algorithms scale.

TODO: For each social graph, include a sentence or two on why its good

6.3.1 Small World Graphs

TODO: Introduce those woganotiz stroff Small World graphs and the way I generated them (with prepare_graph 20% sinks, etc.)

Fig XX shows the results of the benchmarks on these graphs.



(a) Linear scale (b) Loglog scale

Figure 6.11: Runtime of delegation algorithms on a randomly generated delegation graph.

6.3.2 R-Mat Graphs

Another method for generating artificial social graphs is the R-MAT (Recursive Matrix) model. *TODO: cite https://epubs.siam.org/doi/pdf/10.1137/1.9781611972740.43* 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.

6.4 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 *TODO: Cite*, 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 delegations into delegation graphs. We introduce the three datasets individually, followed by a joint evaluation in section 6.4.4.

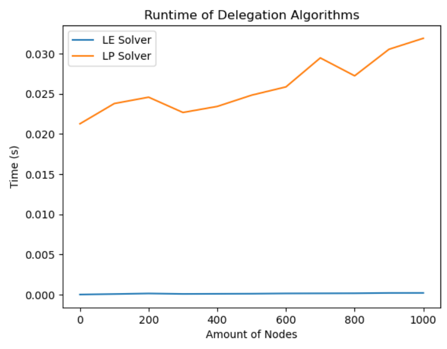


Figure 6.12: Runtime of delegation algorithms on a randomly generated delegation graph.

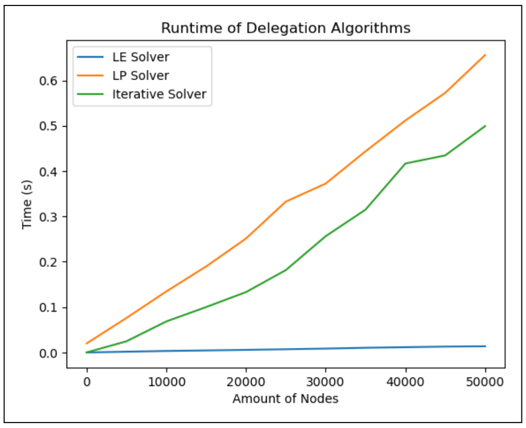
6.4.1 Epinions

Epinions.com is a "general consumer review site", in which members can decide whether to "trust" each other. *TODO: cite: `https://snap.stanford.edu/data/soc-Epinions1.html`*
The Stanford Network Analysis Project (SNAP) provides a web-of-trust graph generated from this relations. *TODO: cite: `https://snap.stanford.edu/data/soc-Epinions1.html`*
The graph is directed and unweighted, thus an existing edge implies trust, and a missing edge implies the lack thereof.

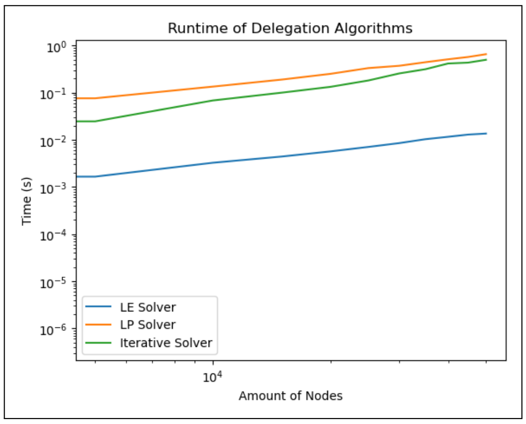
After turning the graph into a delegation graph (see pipeline in fig XX), with the n% sink threshold set to zero, so the algorithm does not add any new sinks to the graph by removing outgoing edges of nodes, we observe the following statistics for the delegation graph, which will be called the Epinions Graph. The Epinions Graph contains 75139 nodes, of which about 0.21% are sinks. Figure 6.14a 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, most powerful node after resolving is the "lost" node, so 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. 6.15a.

TODO: Fix and make more clean the captions for runtime graphs. Currently they mention "delegation algorithms", which is outdated terminology



(a) Linear scale



(b) Loglog scale

Figure 6.13: Runtime of delegation algorithms on a randomly generated delegation graph.

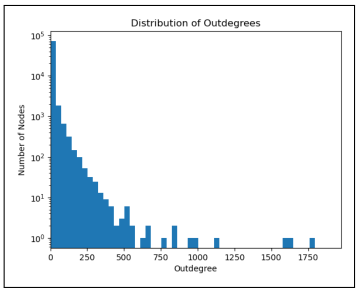
6.4.2 Bitcoin OTC Trust Network

Users trading Bitcoin on the platform "Bitcoin OTC" maintain a record of trust to other users, in order to prevent transactions with untrustworthy users. SNAP provides the Bitcoin OTC Trust Graph, a graph of this trust between users. *TODO: cite:* `https://snap.` `soc-sign-bitcoin-otc.html`

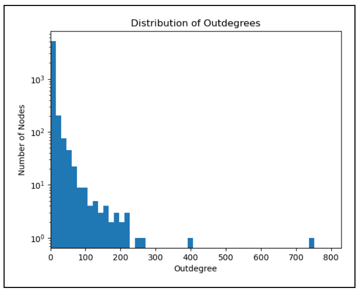
The graph is directed and weighted, with weights ranging from -10 to 10, total distrust to total trust. *TODO: cite:* `https://snap.stanford.edu/data/soc-sign-bitcoin-otc.html`

The graph was first cleaned, to remove all edges with a non-positive trust values, before being turned into a delegation graph as per fig XX, again not adding any sinks artificially. The preprocessing pipeline normalises edge weights, meaning outgoing trust levels are scaled down proportionally to add up to one, preserving relative differences in trust. 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. 6.14b.

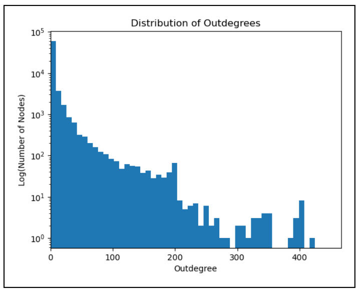
During the preprocessing of this graph, 43 of the graph's 5573 nodes were to be removed since they were in a closed delegation cycle. About 111.2 units of power were lost to closed delegation cycles, 0.02% of the total power in the graph. The distribution of powers after resolving is shown in fig. 6.15b



(a) Epinions Dataset



(b) Bitcoin OTC Dataset



(c) Slashdot Zoo Graph.

Figure 6.14: Distribution of outdegrees

6.4.3 Slashdot Zoo

The Slashdot technology news size has a so-called "zoo" feature, in which users can tag other users as friends and foes. *TODO: Cite: https://dai-labor.de/en/publications/the-slashdot-zoo-mining-a-social-network-with-negative-edges/* The Distributed AI Laboratory in Berlin (DAI Labor) provides a graph based on this data. *TODO: Cite: https://dai-labor.de/en/publications/the-slashdot-zoo-mining-a-social-network-with-negative-edges/* 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. 6.14c.

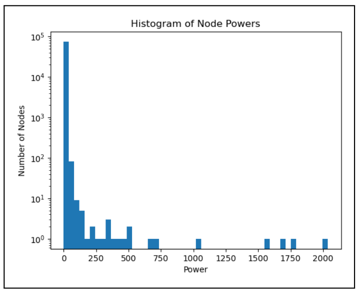
1061 nodes were removed due to being in a closed delegation cycle...

TODO: continue this once I am more sure what I should actually include.

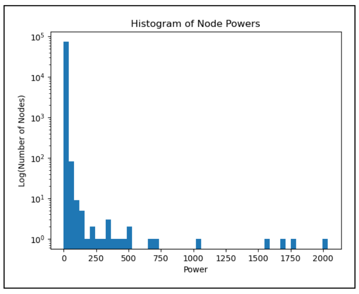
6.4.4 Evaluation of the datasets

TODO: Maybe mention also at some point, that this evaluation is a comparison between different solvers for systems of linear equations

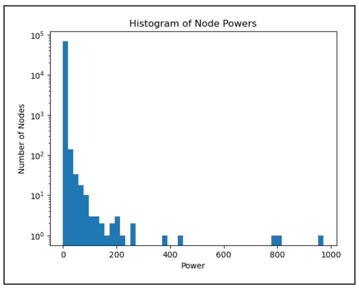
TODO: Mention that the iterative solver is a lot less precise



(a) Epinions Dataset

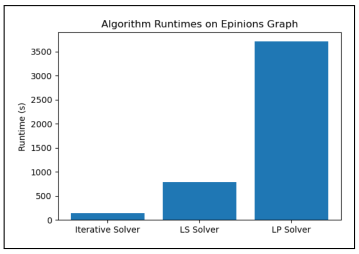


(b) Bitcoin OTC Dataset

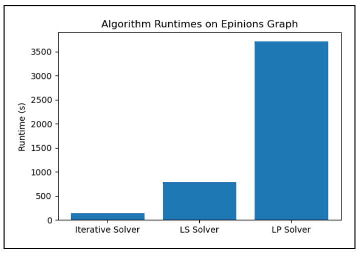


(c) Slashdot Zoo Graph.

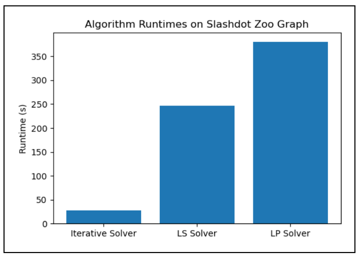
Figure 6.15: Distribution of powers



(a) Epinions Dataset



(b) Bitcoin OTC Dataset



(c) Slashdot Zoo Graph.

Figure 6.16: Runtimes

7 Related Work

TODO: Maybe move this up, toward the background section or introduction

- Degrave paper https://arxiv.org/pdf/1412.4039 Similar to what we're doing, but they don't consider arbitrary splits of delegations, Propose calculating the final power through systems of linear equations

- Bersetche paper "A Voting Power Measure for Liquid Democracy with Multiple Delegation" [Ber22] Here, they propose fractional delegation (called Multiple Delegation) Delegates can also retain power for themselves Also they propose a penalty factor, in order to XXX

- The Bertsche paper cites some practical experiments with LD, I could include those as well

- The viscous democracy paper might also be relevant

- I remember reading some papers that mention why fractional delegation is benefitial, if it was not the Bersetche paper I'll try to find it again..

8 Conclusion

In the conclusion you repeat the main result and finalize the discussion of your project. Mention the core results and why as well as how your system advances the status quo.

Abbreviations

TUM	Technical University of Munich

List of Figures

1.1 Example drawing 1
1.2 Example plot 2
1.3 Example listing 2

4.1 Closed delegation cycles 10
4.2 Sample delegations 11

5.1 Delegation graph and its inverse dict representation 15
5.2 Delegation graph with a cycle. 20
5.3 Two similar delegation graphs 24

6.1 Process to clean any graph into a delegation graph. 26
6.2 Runtime of delegation algorithms on a randomly generated delegation graph. 27
6.3 Runtime of delegation algorithms on a randomly generated delegation graph. 28
6.4 Delegation graphs with 11 and 12 nodes (Blue nodes are sinks) 29
6.5 Runtime of delegation algorithms on a randomly generated delegation graph. 30
6.6 Runtime of delegation algorithms on a randomly generated delegation graph. 31
6.7 Runtime of delegation algorithms on a randomly generated delegation graph. 32
6.8 Runtime of delegation algorithms on a randomly generated delegation graph. 32
6.9 Runtime of delegation algorithms on a randomly generated delegation graph. 33
6.10 An example of the cycles used for the benchmarks. The blue node is the sink 34
6.11 Runtime of delegation algorithms on a randomly generated delegation graph. 35
6.12 Runtime of delegation algorithms on a randomly generated delegation graph. 36

6.13 Runtime of delegation algorithms on a randomly generated delegation graph. 37
6.14 Distribution of outdegrees 38
6.15 Distribution of powers 38
6.16 Runtimes 38

List of Tables

1.1 Example table 1

5.1 $p_v(i)$ values of nodes in the graph in fig. 5.2 21

Bibliography

[Bec+21] 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].

[Beh15] J. Behrens. *Circular Delegations – Myth or Disaster?* Jan. 2015.

[Ber22] F. M. Bersetche. *Generalizing Liquid Democracy to Multi-Agent Delegation: A Voting Power Measure and Equilibrium Analysis*. 2022. DOI: 10.48550/ARXIV.2209.14128.

[Bol+11] 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.

[Bri18] 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.

[BS15] J. Behrens and B. Swierczek. *Preferential Delegation and the Problem of Negative Voting Weight*. Jan. 2015.

[CM19] 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.

[Deg14] J. Degrave. *Resolving Multi-Proxy Transitive Vote Delegation*. Dec. 2014. DOI: 10.48550/arXiv.1412.4039. arXiv: 1412.4039 [cs].

[FL05] J. Forrest and R. Lougee-Heimer. *CBC User Guide*. 2005.

[For+24] 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.

[GÖl+21] 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`.

[HSS08] 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`.

[Kli+15] 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]`.

[KR19] 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]`.

[Li+99] 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.

[OMD11] M. O'Sullivan, S. Mitchell, and I. Dunning. *PuLP : A Linear Programming Toolkit for Python*. 2011.

[Pau20] 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`.

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

[Rev+22] 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.

[Vir+20] 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`.