Decentralized Capital Allocation via Budgeting Boxes

Daniel Kronovet, Aron Fischer, Jack du Rose

{*krono,aron,jack*}*@colony.io*

December 11, 2018

Abstract

Colony's developer incentive mechanism will support development of applications on top of the Colony Network, thereby expanding the scope of the network's utility and adoption, and therefore value. We present a structured curation mechanism for efficiently, reliably, and decentrally allocating newly minted CLNY across an unbounded set of applications, proportional to their utility to the Colony Network.

The mechanism is similar in intent to Proof of Work: the more hashing power you have, the greater your share of block rewards, hence the incentive to compete for hashing power. In Colony's case, the more valuable an application is to the network, the greater the share of newly minted CLNY its developer will receive, hence the incentive is to maximize the application's utility to the network. In this case, "value" is determined not objectively, but via reputation-weighted subjective inputs, mediated by a novel computational component we call a "budget box".

Keywords: Blockchain, Governance, Budgeting, Ranking

1. Introduction

Colony is the people layer of the decentralized protocol stack [1]. As with many emerging blockchain protocols, there is a desire to encourage the development of the ecosystem surrounding this protocol. For many projects, this support manifested via grants programs [2]. These programs, while not ineffective, do have some shortcomings: they centralize decision-making power in a small grants committee; they are not self-perpetuating and require ongoing administration on the part of the protocol; they provide only a finite amount of funding to execute on a specific project: essentially a large bounty, not sustainable income, so the incentive to maximize the utility of the application is limited.

As an alternative, we seek to develop a structured curation mechanism for autonomously allocating *recurring* funding (denominated in CLNY) over an unbounded number of projects in the Colony ecosystem on an ongoing basis. We have designed this mechanism to be feasible on-chain given the current limitations of the Ethereum network. The design is intentionally modular; we hope that components of this mechanism will find use in other contexts.

1.1. Challenges

Given that we essentially propose to give away money in a trustless and pseudonymous environment, it is essential that this mechanism be robust against many types of exploit and failure scenario, such as:

1. Spamming the mechanism with fake projects
2. Scammers claiming to represent legitimate projects
3. Projects bribing voters to vote for projects in excess of their true value
4. Winning projects using their reputation to vote themselves up
5. Winning projects colluding to claim a larger share of CLNY
6. Voter apathy
7. Cognitive biases of various kinds, like the Keynesian Beauty Contest
8. Voters voting dishonestly or randomly
9. Sybil attacks of any sort (either among voters or projects)

With respect to Goodhart's Law,[1] we take as axiomatic that any assessment based on objective, on-chain metrics will inevitably lead to projects

[1] `https://en.wikipedia.org/wiki/Goodhart%27s_law`

trying to "game the system", artificially inflating these metrics in order to secure disproportionate allocations of CLNY. To prevent this, we generate our assessment from subjective judgments only; our mechanism is designed to most effectively gather and analyze these judgments. It is not essential that the allocation be demonstrably "optimal" with respect to some formal objective at some specific moment in time, but rather that the allocation is maximally responsive to new information. In contrast to expensive one-shot allocation processes, this mechanism is designed to allow for simple and ongoing calibration of allocation in response to changing circumstances.

1.2. Our Approach

Before explicating the technical details, let us give a high-level introduction to our approach. In essence, our goal is to create a mechanism which *efficiently and reliably converts subjective inputs (preferences) to objective outputs (budgets or rankings)* in way which minimizes the cost of input and effectively (and interpretably) leverages the information in those inputs.

The first problem is the manner of representing subjective preference – a question which has dogged at the heels of social scientists for decades. Early approaches in which subjective preferences are directly assigned quantitative values have been soundly rejected by scholars, with economist Kenneth Arrow famously observing that "interpersonal comparison of utilities [have] no meaning" [3], as any set of quantitative values of subjective preferences can be scaled nearly arbitrarily without consequence (i.e. my "utilities" of 10 and 100 contain the same information as your 2 and 20). Attempts to address this by "standardizing the scale" to 5 or 10 points, or to an array of verbal statements (via a Likert scale) have themselves run aground, as the personal and social dynamics affecting their usage makes analysis of these data difficult [4][5]; both YouTube (in 2009)[2] and Netflix (in 2017)[3] have replaced their 5-star scales with binary up/down votes, which they argue provide cleaner signals. Some contemporary researchers have sought to bypass this problem entirely by positing "implicit utility functions" observed *indirectly* through voting data [6], a technique known as *implicit utilitarian voting*.

We take a different approach, in which we recognize that while absolute

[2] `https://youtube.googleblog.com/2009/09/five-stars-dominate-ratings.html`
[3] `https://variety.com/2017/digital/news/netflix-thumbs-vs-stars-1202010492`

utility for individual items is undefined, *relative utility for pairs of items* are well defined, and can in fact be encoded in a single bit (a "pairwise preference"). In an important sense, these pairwise preferences are "quanta of subjectivity" and have been used successfully to infer quantitative measurements for social values [7].

In addition, the use of pairwise preferences has a number of technical advantages, the most important being the ease and flexibility of analysis. A set of binary preferences over a fixed set of items can be stored in a matrix; these matrices form an algebraic structure known as a *semigroup*. This semigroup permits the addition operation, allowing for easy aggregation of the preferences of groups; it is less obvious how one might aggregate, for example, a set of lists. While Arrow has observed that "it seems to make no sense to add the utility of one individual, a psychic magnitude in his mind, with the utility of another individual" [3], we can easily speak of summing the number of times members of a group have preferred a to b.

Another major advantage of pairwise preferences is the minimal cognitive burden placed on the voter, being literally the smallest possible input (one bit). Compared to the cognitively demanding, often adversarial, and inevitably political process of budgeting directly (for example, by sitting in a boardroom debating line items), pairwise distinctions are easy to make for humans and the experience of pairwise voting can be fun and engaging [8].

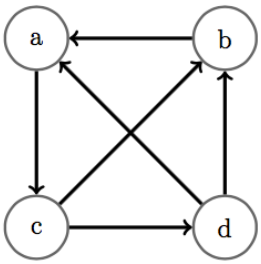
It is well known that large-scale collaboration over granular decisions often consumes more time and energy than the magnitude of the decision might otherwise warrant, a phenomena known as the "bikeshed" effect. Pairwise preferences reduce this friction by enabling voters to provide individual feedback at an *optimal level of abstraction*, while allowing for flexible aggregation and algorithmic analysis of feedback to determine actual quantitative allocations [9]. In addition, pairwise preferences can capture more information than alternatives like ordered lists (such as intransitive preference structure), leading to more accurate rankings, in exchange for somewhat higher computational costs. As an added benefit, the nonlinear transformation from pairwise preferences to final allocations impedes tactical voting.

One might ask why we don't simply "count the likes" and assign each item a score equal to the sum of the (also binary) upvotes – an approach arguably more efficient than one which necessitates gathering a large set of pairwise

feedback.[4] The answer is that when allocating a fixed budget, the question we answer is not "what is absolutely good", but rather "what is relatively more valuable." It is a fundamentally relational question. No elected official would dispute the importance of roads and schools, yet they fund pensions and defense. Further, asking voters to upvote a subset of items requires them to *implicitly* compare each item to every other, as each endorsement reduces the available surplus. We ask the question we seek to answer.

2. Budgeting Boxes

The cornerstone of this mechanism is a novel component called a *Budgeting Box*, implementing a simple, general, and powerful governance algorithm. As discussed in Section 1.2, budgeting boxes aggregate and process sets of pairwise preferences. Our approach involves converting our set of pairwise preferences into a Markov transition matrix $\mathbf{M}$, and using this matrix to find an eigenvector $\mathbf{v}$ corresponding to a probability distribution over the items, which we may alternatively interpret as a budget or ranking. Intuitively, we can think of the final probability as the likelihood of that item being "the most important thing", based on the observed preferences.



$$\mathbf{M} = \begin{bmatrix} .66 & .33 & 0 & .33 \\ 0 & .66 & .33 & .33 \\ .33 & 0 & .33 & 0 \\ 0 & 0 & .33 & .33 \end{bmatrix} \qquad \mathbf{v} = [.4, .3, .2, .1]$$

Figure 1: Example $\mathbf{M}$ and $\mathbf{v}$ for $K = 4$, $d = 1$.

This method is similar to those used in the past to rank sports teams [10], calculate relevancy of web pages [11], and determine node trustworthiness in peer-to-peer networks [12], and has the nice property in that it can generate a distribution even if the underlying preferences are intransitive, making the method particularly well-suited to solving problems of social choice [9]. The primary difference between those approaches and ours is that in our

[4]For an example of this, see `https://dashnexus.org/leaderboard`

approach, the interactions come not endogenously from the items themselves, but exogenously via a voting process. This distinction will have important consequences for the computational scalability of our approach.

2.1. Technical Exposition

We begin with a set $\mathcal{R}$ of voters, and an ordered set of K projects. Each voter $r \in \mathcal{R}$ will consider some subset of the $K(K-1)/2$ possible pairs of items and generate a matrix $\mathbf{X}^r$ of votes:

$$\mathbf{X}^r \in \{0,1\}^{K \times K}$$

with $\mathbf{X}^r_{i,j} = 1$ representing r's preference of item i over item j. As preference cannot be bi-directional or reflexive (I cannot prefer both i to j and j to i, nor i to i), this implies both that $\mathbf{X}^r_{i,j} + \mathbf{X}^r_{j,i} \in \{0,1\}$ and $\mathbf{X}^r_{i,i} = 0$.

Then, we aggregate across all the voters:

$$\mathbf{X} := \sum_{r \in \mathcal{R}} \mathbf{X}^r$$

Note that the while the matrix $\mathbf{X}$ is a lossless aggregation of all the preference information, further processing is required to create a Markov matrix $\mathbf{M}$. We perform this conversion in four steps.

First, we initialize:

$$\mathbf{M} := \mathbf{X}$$

Second, we set the diagonal to the sum of the rows (each item's "victories"). Adding this diagonal ensures that items with many victories retain their probability mass:

$$\mathbf{M}_{i,i} := \sum_j \mathbf{M}_{i,j}$$

Third, we normalize the transition probabilities by dividing each column by its sum.[5] This changes the interpretation of $\mathbf{M}$ from a simple count of votes to a set of probability distributions, in which each column j represents the probabilities of transitioning *from that j to any i*:

[5]The $\oslash$ symbol represents Hadamard division, here taken to be column-wise division by the corresponding column sums.

$$\mathbf{M}_{*,j} := \mathbf{M}_{*,j} \oslash \sum_i \mathbf{M}_{i,j}$$

Fourth and finally, we add a "damping factor" $d \in [0, 1]$. This factor was introduced by Brin and Page in their original PageRank paper [11] to act as a regularizer and ensure that *some* probability mass accrues to even unpopular items:

$$\mathbf{M} := d\mathbf{M} + \frac{1-d}{K}$$

We can think of d as a type of prior which bounds the *maximum ratio* between the probabilities assigned to the most likely and least likely items: for $d = 1$, the ratio approaches infinity, as it is possible for 100% of the probability to accrue to a single item; for $d = 0$, the ratio is 1 as every item will have a probability of $1/K$. We will revisit d in Section 6.

Our voting information has now been converted into an *irreducible, aperiodic, ergodic* Markov matrix guaranteed [13] to possess a unique principal eigenvector (or "steady-state distribution") $\mathbf{v} \in \triangle^{K-1}$ (the K-dimensional simplex, where all elements are less than 1 and which sum to 1), interpretable as a probability distribution over the items, or alternatively as either a ranking or allocation of a budget. A number of methods exist for finding eigenvectors; we will use *power iteration* (see Algorithm 1), a straightforward algorithm returning $\mathbf{v}$ to a precision of ϵ.

Algorithm 1 Power iteration

```
procedure POWER(M, ε)
    v := 1 1/K
    while ||v − Mv||_2 ≥ ε do          ▷ O(K)
        v := Mv                        ▷ O(K^2)
    end while
    return v
end procedure
```

Intuitively, every iteration first asks "which items are the most popular?" and then "for every item, which items are even more popular?" Over time, probability mass converges to the items which have been relatively preferred, as popular items accumulate more probability, which they then "send forward" to items more popular than they.

2.2. Scalability

In terms of computational complexity, the key observation is that finding the principal eigenvector of a matrix is *cubic* in the number of items, $O(K^3)$, while constructing the Markov matrix $\mathbf{M}$ is *linear* in the number of voters and *quadratic* in the number of items, $O(|\mathcal{R}|K^2/2)$. This is the crux of what makes this application computationally feasible: as long as K – the number of projects competing in a single budget box – is kept small (on the order of 10 or so) then these computations can be done on-chain, even if the number of voters $|\mathcal{R}|$ is large. See Appendix A for implementation details.

As we will see below, a limit of K items per box does not restrict a mechanism to K items overall, as multiple budgeting boxes can be composed to accommodate larger sets of items. One approach to the composition is to assemble boxes into a tournament bracket (as we do in Section 3); another approach is to compose them into a recursive hierarchy of "abstract to concrete" (think public services → education → primary schools).[6]

The composition of budgeting boxes reduces cognitive complexity in tandem with computational complexity. For 100 items, there are 4,950 possible pairs; compare this to $10 \cdot 45 = 450$ pairs when this same 100 is divided into 10 groups of 10. These 450 pairs are a subset of the original 4,950 – but not a random one. If we posit that these 100 items can be ordered by quality, and that only comparisons between items of similar quality provide useful information, then subdividing the set of items into subsets of items of similar quality allows us to avoid making 4,500 low-value comparisons: a 10x improvement in cognitive efficiency.

We conclude this section by emphasizing that, inasmuch as *matrices are functions* and *budgets are policy*, the new affordance of *programmable money* made possible by distributed ledgers allow us to think in fresh ways about the possibilities of broad-based *programmable government*.

3. Curation Mechanism

We now describe our mechanism in full. Let $B \in \mathbb{N}$ be our total available budget (in wei), let $\mathcal{R}$ be the set of reputation-holding Ethereum addresses,

[6] An approach which pairs well with liquid democracy or participatory budgeting.

and $\mathcal{P}$ be the set of Ethereum addresses representing projects being considered. Our goal is to gather input from the reputation holders and to use this input to determine an allocation b^p for each project $p \in \mathcal{P}$.

Let $B^{\mathcal{P}}$ be the total budget allocated to all projects, and let $B^{\mathcal{R}}$ be the total budget allocated to the voters, i.e.

$$\sum_{p \in \mathcal{P}} b^p = B^{\mathcal{P}}$$

representing the allocation to the projects, and

$$\sum_{r \in \mathcal{R}} b^r = B^{\mathcal{R}}.$$

representing the allocation to the voters, subject to $B^{\mathcal{P}} + B^{\mathcal{R}} = B$. The fraction of the total budget put aside to compensate voters must be set in advance; the remainder will be available for the projects.

3.1. Division into Leagues and Lanes

To make the best use of the limited attentional resources of the voters, we divide the set of items into two categories: those in the *leagues* (each league being essentially a thin wrapper around a budgeting box), and the remainder in one of several lanes of the *pool*. All items begin in the pool; items which receive sufficient of support will be promoted to the leagues (see Figure 2). Items in the leagues are then compared against each other in a pairwise fashion and ranked; items which rank highly may be promoted to higher leagues and become eligible for larger rewards.

The set of projects $\mathcal{P}$ is partitioned into $G + L$ subsets of tournament lea**g**ues and pool **l**anes, respectively. Each league contains K projects and each lane contains up to 256 projects. Each project appears exactly once, either in a league or in a lane. As such, $\mathcal{P}^g$ and $\mathcal{P}^l$ denote non-overlapping subsets of $\mathcal{P}$, such that

$$\sum_{g=1}^{G} |\mathcal{P}^g| + \sum_{l=1}^{L} |\mathcal{P}^l| = |\mathcal{P}|.$$

Leagues are aranged in a binary tree of Z complete tiers, thus we have $G \triangleq 2^Z - 1$ at all times. The number of pool lanes L will also be depend on Z and on the "shuffling factor" C that is introduced later on.

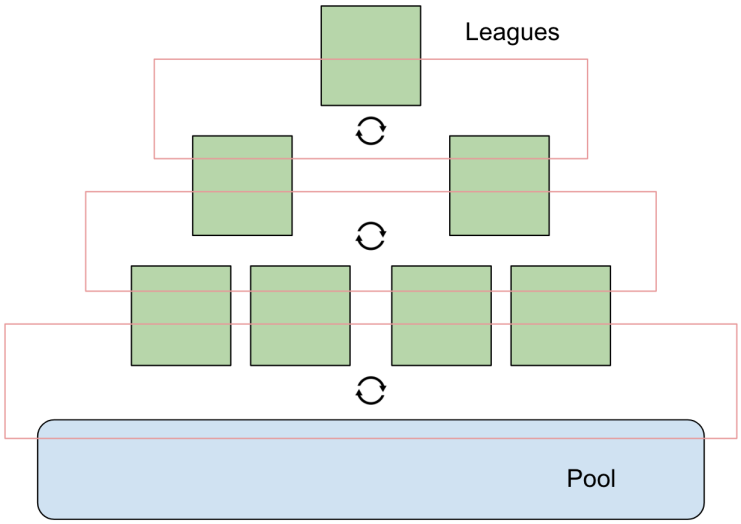


Figure 2: Pool and Leagues for $Z = 3$

3.2. Scoring the Leagues

We have G leagues arranged in a binary tree of $Z \in \mathbb{N}$ tiers, with each league containing K projects. For example, if there are three tiers (as pictured in Figure 2), there is one premier league, two leagues in tier two, and four leagues in tier three, containing a total of $7K$ projects. Leagues are added one complete tier at a time (by incrementing Z). Votes in the leagues consist of pairwise comparisons between the projects.

As described in Section 2, for a league $g \in \{1 \dots G\}$, reputation holders $r \in \mathcal{R}$ generate vote matrices $\mathbf{X}^{g,r}$. We now aggregate them into *reputation-weighted* sums (where $rep : r \to \mathbb{N}$ gives us the reputation of r):

$$\mathbf{X}^g := \sum_{r \in \mathcal{R}} rep(r) \cdot \mathbf{X}^{g,r}$$

We then transform $\mathbf{X}^g \to \mathbf{M}^g$ as described in Section 2.1, and obtain:

$$\mathbf{v}^g := power(\mathbf{M}^g, \epsilon)$$

As mentioned in Section 2.1, $\mathbf{v}^g$ is a vector of K elements, the values of which are all less than 1 and which sum to 1, and which we will use to both determine rewards and update league assignments.

3.3. Scoring the Lanes

Projects in the leagues receive rewards, paid out via the Colony task mechanism. The pool, on the other hand, serves as a reservoir of projects hoping to get into the leagues; projects in the pool do not receive rewards.

Consequently, in comparison to the high-stakes, high-attention leagues, the mechanism for processing projects in the pool is simpler, providing only a coarse filter. The pool is divided into L "lanes", each of which can hold up to 256 projects. Projects enter a lane by staking some fixed amount of CLNY tokens. Once in a lane, projects receive votes of approval only; projects receive scores equal to the reputation-weighted sum of their approvals.

While league votes are stored in a pairwise comparison matrix, per-reputation-holder lane votes can be stored in a vector:

$$\mathbf{y}^{l,r} \in \{0,1\}^{256}$$

Here, a value of 1 represents approval and 0 represents no approval. The per-reputation holder votes $\mathbf{y}^{l,r}$ are aggregated much in the same way as the pairwise matrices:

$$\mathbf{y}^{l} := \sum_{r \in \mathcal{R}} rep(r) \cdot \mathbf{y}^{l,r}$$

At the end of the voting period, we have an ordering of each pool lane l in which projects are ranked based on their total backing reputation.

The staking requirement is intended as an anti-spam mechanism. As a further protection against low-quality submissions (which drain voter attention), projects in the pool with a score less than 0.2 of the median will have their stakes burnt. To prevent projects from withdrawing their stake in anticipation of being burnt, we allow projects to leave the pool only during the first third of the voting period.

3.4. Moving Between Leagues

Our mechanism functions much like a sports league, in which those that rank at the bottom at the end of the season are relegated to a lower league, while those that rank highly will move to a higher one. A "season" in our case is one voting period. At the end of a voting period, all the projects

$p \in \mathcal{P}$ are scored, rewards are paid out where appropriate, and the projects are assigned a league for the following period.

The reassignment is governed by the parameters Z and a "shuffling factor" C, which determines how many projects move between periods. As the leagues are arranged in a binary tree of G elements, it is possible to index every node and traverse the tree via the following relation:

$$children(g) = (2g, 2g + 1).$$

Reassignment amounts to swapping (denoted $\leftrightsquigarrow$) the $2C$ lowest-ranked projects in league g with the C highest-ranked projects in each of $children(g)$. Thus the parameter C must be chosen such that $1 \leqslant C \leqslant \lfloor K/3 \rfloor$; smaller values lead to less volatility in the leagues. The number of bottom-tier leagues (leaves in the league tree) is always 2^{Z-1} and so $2^{Z-1} \cdot 2C$ projects will be lifted from the pool into the bottom tier in every period. For clarity we will use $\tilde{g} \triangleq 2^{Z-1}$ to indicate the first leaf league.

3.4.1. Moving from one League to another

The bottom (after sorting by score) $2C$ projects in league g are swapped with the top $C + C$ projects from league g's children, where the demotion from g to $children(g)$ happens in alternating fashion (see Figure 3). For example, if the leagues contain 10 projects ($K = 10$) and the 4 lowest ranked projects are to be demoted ($C = 2$), then the projects ranked 7 and 9 will be demoted to the 'left' child league ($2g$) and the projects ranked 8 and 10 will be demoted to the 'right' child league ($2g + 1$).

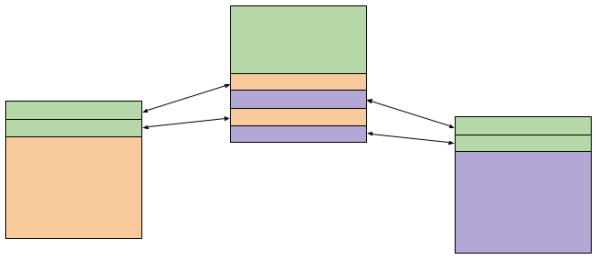


Figure 3: Swapping Between Leagues for $C = 2$

3.4.2. Moving from the Lanes to the Leagues

Projects that rank at the bottom of the bottom-tier leagues $g \in \{\tilde{g}, ..., G\}$ are expunged from the system (with their stake returned). They are free to reenter the pool by posting a new stake. The newly vacated spots are filled by the top projects in the pool lanes. We aim to have no more than half of the pool enter the leagues at any one time, thus each lane can support a maximum of $\lfloor 256/4C \rfloor$ leagues and so $L \triangleq \left\lceil 2^{Z-1} \cdot 4C/256 \right\rceil$.

When there are multiple lanes, they are assigned to leagues in a round-robin fashion i.e. for a leaf league g, the associated pool lane would be:

$$lane(g) = (g - \tilde{g}) \bmod L + 1.$$

Conversely, we denote the leagues associated with a lane by $leagues(l)$. The goal is to replace the bottom $2C$ projects in all the leaf leagues with the top $2C \cdot |leagues(l)|$ projects in the pool lanes. So for each lane we replace

$$\bigcup_{g \in leagues(l)} \mathcal{P}^g[K - 2C : K] \leftrightsquigarrow \mathcal{P}^l[0 : 2C \cdot |leagues(l)|].$$

To keep the distribution of candidate quality balanced amongst the leagues, the implementation should interleave the swap in a round-robin fashion such that the strongest candidates are distributed evenly amongst the leagues which they enter.

3.5. Determining Allocation

Once projects have been voted on and the votes processed, but before they are moved between leagues, their final budget allocations are determined. Only projects in the leagues receive an allocation, governed by $\mathbf{v}$. Recall that we included a damping factor d which allows for evening out the distribution over all projects in the league. Relatedly, we must decide how to distribute the available budget *across* the leagues.

This allocation is governed by a parameter $Q \in [0, 1]$, which guides us between two extremes: at $Q = 1$ we have a uniform allocation over all leagues ($B^{\mathcal{P}}/G$ each), and at $Q = 0$ an exponentially diminishing allocation ($B^{\mathcal{P}}/Z$ to each tier, shared uniformly among the $2^{\lfloor log_2(g) \rfloor}$ leagues in the tier).

Knowing Q we have the total budget available to league g:

$$\mathcal{B}^g = \left(\frac{QB^{\mathcal{P}}}{G} + \frac{(1-Q)B^{\mathcal{P}}}{Z \cdot 2^{\lfloor log_2(g) \rfloor}} \right).$$

For a given league, the allocation will be distributed amongst the projects according to their score, leveraging the "budgetary" interpretation of $\mathbf{v}$. The specific allocation for project p in league g can be determined as follows:

$$b^p = score(\mathbf{v}^g, p) \cdot \mathcal{B}^g$$

where $score(\mathbf{v}^g, p)$ refers to the final weight accrued by project p in $\mathbf{v}^g$.

3.6. Expanding the Mechanism

At the beginning of this paper, we claimed this mechanism could support an unbounded number of projects. This is true, in that the mechanism *can be expanded* to accommodate an arbitrarily large number of projects. Yet, for a given set of parameters, $|\mathcal{P}|$ is bounded via the following relation:

$$|\mathcal{P}| \leq GK + \left\lceil \frac{\tilde{g} \cdot 4C}{256} \right\rceil 256.$$

The first term on the right hand side is the number of projects occupying the leagues; the second is the maximum number of projects which can occupy the pool. Note that the mechanism's capacity is driven by three parameters: Z, which determines the number of leagues, K, which determines their size, and C, which determines the degree of movement; jointly they determine the size of the pool, which we require to be able to hold twice the number of projects which can enter the leagues in any period.

As the Colony Network matures and the number of projects looking to participate in this mechanism grows, the mechanism can be expanded by incrementing Z; every increment will double the size of the leagues and create new lanes as necessary. Note that as lanes are added, existing candidates are not redistributed, creating a type of "arbitrage opportunity" where projects can stake themselves into lanes where competition is less intense; this will naturally lead to an even distribution of project quality amongst the lanes.

4. Voting Process

In order to make this mechanism Sybil-, bribery-, and collusion-resistant, we first appeal to the Colony reputation system [1]. With reputation-weighted voting, those with the most influence also have the most to gain from the growth of the Colony Network, and are thus the least incentivized to accept

immediate payouts (bribes) in exchange for worse long-term prospects. Further, as reputation derives from work done for the Meta Colony, those with more influence ostensibly have better knowledge of the ecosystem and can make better determinations about the relative value of projects.

4.1. Compensation

In order to encourage participation, we will set aside a portion of B for voter compensation, with per-voter compensation a function of *relative* reputation. Making compensation proportional to reputation serves the dual purpose of Sibyl-resistance and encouraging those with the most influence to expend the most effort in making judicious decisions (to increase their long-term rewards) and further decrease the relative incentive to accept bribes in exchange for favorable treatment [14]. If $B^{\mathcal{R}}$ is the portion of the budget set aside for voter compensation, then the per-voter compensation b^r is determined as follows:

$$b^r \triangleq \frac{rep(r)}{\sum_{r \in \mathcal{R}} rep(r)} B^{\mathcal{R}}$$

4.2. Voting Subdivision

Another consideration is the subdivision of the voting process. Should a "unit of voting" involve voting on every pair and project? Or should reputation holders be able to contribute smaller units, as motivation permits?

Fortunately, the "pool and leagues" organization of projects leads to a natural compartmentalization of work. Voting on one league or one lane constitutes one "unit" of voting. Compensation for one "unit of work" will be $b^r/(G + L)$. It is possible to submit partial sets of votes, trading off expediency for coverage; as such, the per-unit voting requirement should be set in advance.

4.3. Voting Constraints

Having divided the voting process into "units of work", we can ask what additional structure could be introduced to encourage reputation-holders to vote in the long-term interest of the Colony Network.

The first thing to consider is the scenario in which $\mathcal{P} \cap \mathcal{R} \neq \emptyset$, i.e. the (expected) case where a project holds reputation. In the case when a project p would have the ability to vote for itself, we should assume that it will do

so. While in one sense, self-voting is not a fatal flaw (strong projects which accumulate influence in the network are likely to be deserving of funding), it nonetheless introduces an undesirable bias to the mechanism. To address this, we impose a "eurovision rule",[7] in which reputation-holding accounts cannot vote in leagues in which they appear:

$$canVote_1(r,g) \triangleq \begin{cases} true & \text{if } r \notin \mathcal{P}^g \\ false & \text{otherwise} \end{cases}$$

The next consideration is the pattern of attentional distribution to the leagues. If we make the reasonable assumption that reputation holders will not consistently vote on all of the leagues, we would like to encourage a relatively even distribution of attention across all of the leagues. We would like to avoid a situation where all reputation holders vote on league 1 only (containing the "top" projects and receiving the largest share of the payout) and ignore the "down-tier" leagues.

To address this, we impose a "random-walk" rule, in which reputation holders can vote on leagues only in a particular order, corresponding to a per-address, per-period random permutation of the sequence $(1, ..., G)$. Such a per-address permutation is possible using algorithms such as the Fisher-Yates shuffle[8], which runs time linear to the number of items being shuffled (in this case, $O(G)$, which is small). If $\mathcal{X}^r \triangleq \{\mathbf{X}^{1,r}, ..., \mathbf{X}^{n,r}\}$ is the set of votes previously submitted by r across all leagues and $FY(r,G)$ is the random permutation, then:

$$canVote_2(r,g) \triangleq \begin{cases} true & \text{if } FY(r,G)_{n+1} = g \\ false & \text{otherwise} \end{cases}$$

If we assume that reputation-holders care about voting in league 1, and will continue to vote until they are able to do so (and then stop), then this rule will deliver a uniform distribution of attention to leagues $2, ..., G$.

5. Payout Process

The curation mechanism described in this paper will be implemented as a stand-alone set of contracts and integrated into the Colony network using

[7] After the well-known international song contest.

[8] `https://stackoverflow.com/questions/6771219`

the Colony permissions system [1]. When deploying this mechanism, a new top-level domain will be created in the Meta Colony, and the mechanism contract will be given *task* and *funding* permissions in that domain.

In every time period, B CLNY tokens will be allocated to this domain, and at the conclusion of the voting process, the mechanism contract will create and finalize tasks corresponding to every payout b^r, b^p, assigning the recipient as the worker. It is unlikely that the entirety of the voter compensation budget $B^{\mathcal{R}}$ will be claimed in any given time period; any remains will be rolled over into the next.

6. Governing the Mechanism

As described above, the mechanism as a whole is governed by a number of *hyperparameters*: K, which determines the size of the leagues; Z, which determines their number; C, which determines the degree of movement between periods; B, which determines the total budget; and d and Q, which affect the distribution of budget within and between the leagues, respectively. Unlike the pairwise votes, which capture *personal* preference, these hyperparameters capture *political* preference, as they define the behavior of the system as a whole: the size of the payouts, the number of projects eligible to compete, the permissible level of inequality in the payouts going to the popular versus unpopular projects, and the speed with which the allocation a specific projects can rise or fall.

All of these parameters, except for the fixed K, can be updated via *reputation-weighted voting* in the Meta Colony. Our expectation is that Z and B will begin small and be increased over time as the network matures and the number of projects increases. We anticipate that d, Q, and C will be updated less frequently to fine-tune the behavior of the mechanism; we expect to launch the mechanism with low values of each, such that payouts become meaningfully larger in higher leagues, that "losing" projects in higher leagues will still earn more than "winning" projects in lower leagues, and that movement between leagues is slow enough that projects have a meaningful time horizon of income.

It is intentional that winning projects will accumulate reputation and influence in the Meta Colony over time. As these projects both support and depend on the Colony Network, it is appropriate that they are able to both participate in the governance and share in the rewards of the network.

7. Conclusion

We have presented a game-like mechanism for collective budgeting, which leverages the "wisdom of the crowd" while sidestepping many of the hangups and pitfalls associated with collective decision-making. While aspects of the mechanism may seem complex, closer inspection will reveal that the relationships and interactions between the components are computationally straightforward, and allow the entire mechanism to be governed by only a handful of parameters.

In our introduction, we introduced a number of possible attack vectors and failure scenarios, and claimed to have developed defenses against them. We will take them point by point:

1. Spam. Soln: minimum stake makes spamming expensive.
2. Scammers. Soln: low initial payouts makes scamming uneconomic.
3. Bribes. Soln: reputation-weighted voting aligning long-term incentives.
4. Self-voting. Soln: eurovision rule.
5. Collusion. Soln: nonlinear vote interactions, random walks.
6. Voter apathy. Soln: engaging experience, voter compensation.
7. Cognitive biases. Soln: nonlinear vote interactions.
8. Random voting. Soln: reputation-weighted voting and compensation.
9. Sybil attacks. Soln: reputation-weighted voting and compensation.

It is important to note that no mechanism can ever be fully resistant to attack or failure; every mechanism, even proof-of-work, functions conditional on some reality. The mechanism we describe assumes *a reasonably diverse set of engaged reputation holders* looking to benefit from the long-term success of the Colony Network. The goal of this mechanism is not to provide a fool-proof framework for decision-making, but rather to solve a specific set of problems without introducing a large set of new ones.

We believe that the use of pairwise preferences to solve collective ranking and budgeting problems represents an important tool in the decentralized governance toolkit.

References

[1] A. Rea, A. Fischer, J. du Rose, Colony: Technical white paper, `https://colony.io/whitepaper.pdf` (2018).

[2] N. Eghbal, Decentralized funding? an analysis of three programs, https://nadiaeghbal.com/grant-programs/ (2018).

[3] K. J. Arrow, Social Choice and Individual Values, Yale University Press, 2nd edition, 1970.

[4] S. Jamieson, Likert scales: How to (ab)use them, Medical Education (2004).

[5] H. H. Nax, The logic of collective rating, Frontiers in Physics (2016).

[6] G. Benade, S. Nath, A. D. Procaccia, N. Shah, Preference elicitation for participatory budgeting, Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence (AAAI-17) (2017).

[7] L. L. Thurstone, The method of paired comparisons for social values, Journal of Abnormal and Social Psychology (1927).

[8] M. J. Salganik, K. E. C. Levy, Wiki surveys: Open and quantifiable social data collection, PLoS ONE (2015).

[9] D. Kronovet, An analysis of pairwise preference, https://github.com/kronosapiens/masters-thesis (2017).

[10] J. P. Keener, The perron-frobenius theorem and the ranking of football teams, SIAM Review (1993).

[11] S. Brin, L. Page, The anatomy of a large-scale hypertextual web search engine, Computer Networks and ISDN Systems (1998).

[12] S. D. Kamvar, M. T. Schlosser, H. Garcia-Molina, The eigentrust algorithm for reputation management in p2p networks, WWW2003 (2003).

[13] H. W. Lin, M. Tegmark, Criticality in formal languages and statistical physics, arXiv (2017).

[14] C. Ferraz, F. Finan, Motivating politicians: The impacts of monetary incentives on quality and performance, NBER Working Paper 14906 (2009).

Appendix A. Gas Analysis

In Section 2.2, we asserted that limits on K make the power iteration algorithm feasible on-chain. Here we present `BudgetBox`,[9] a prototype implemention, and look at gas costs for various operations. There are two main costs for a budgeting box: submitting the votes and finding the eigenvector.

Appendix A.1. Power Iteration

First we will consider the largest chunk of work, the *power* algorithm. In general, finding an eigenvector is an $O(K^3)$ operation, but some matrices require more computation than others. Here we look at three different voting styles, and the amount of gas required to find $\mathbf{v}$ for each.

The first style, *random*, involves voters choosing between pairs at random. This leads to a $\mathbf{v}$ which is close to uniform, and requires few iterations to converge, due to our algorithm beginning with a uniform vector. The second style, *transitive*, involves voters choosing between pairs in a way that is consistent with a transitive ordering ($a \leftarrow b \leftarrow c$) among the items. Transitive orderings lead to a $\mathbf{v}$ which is far from uniform, but converges quickly due to the consistency which with probability mass flows from losers to winners. The third style, *mixed*, mixes random and transitive orderings in a way which negates the computational efficiencies of each. In Figure A.4, we plot gas costs from running power iteration (including the transformation from $\mathbf{X} \rightarrow \mathbf{M}$) on sets of 100, 200, 400, 600, and 1000 votes, for different K, with all three styles.

Looking at these results, we see that for all voting styles, larger K used more gas. Further, we see that for every K, mixed voting used more gas than random or transitive voting. All permutations were able to execute within 4m gas, less than half of the current 8m Ethereum block gas limit.

Appendix A.2. Manipulating Votes

Next we consider the gas costs associated with submitting and manipulating raw vote information. A raw vote can be encoded cheaply. If we enumerate each of the $K(K-1)/2$ pairs, and reserve two bits per pair (the first to indicate whether the pair was voted on, and the second to indicate

[9] `https://github.com/JoinColony/budgetBox`

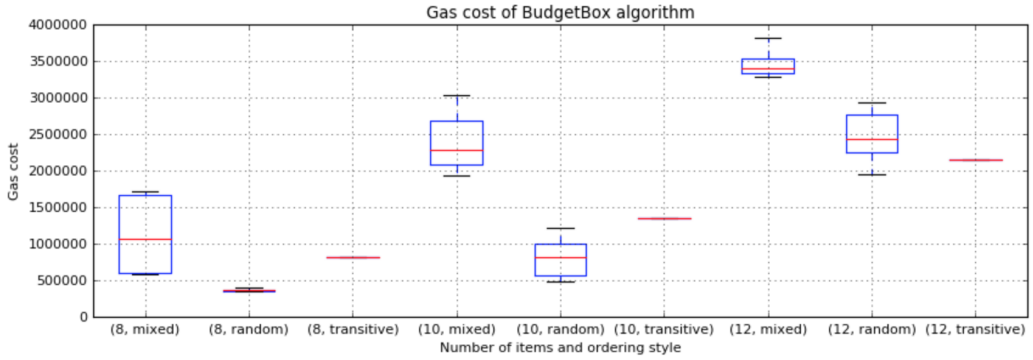


Figure A.4: Gas Costs for Power Iteration

the direction of the preference), we can encode preferences for up to 16 items in a `uint256`, appended to an array in storage.

The difficulty comes from loading the array of votes into the matrix. It is not strictly necessary to save $\mathbf{X}$ in storage, but as each vote requires $O(K^2)$ operations to parse, loading more than a few hundred votes can consume gas in excess of the block limit. A "divide-and-conquer" strategy is to periodically "load" some fixed number of votes (say, 50) into a storage matrix. Each loading transaction involves setting $O(K^2)$ storage variables; including more votes lets us amortize this cost (and leverage a larger refund from clearing more words from storage). Smaller values of K allow more votes to be included in a load operation, leading to greater gas efficiency.

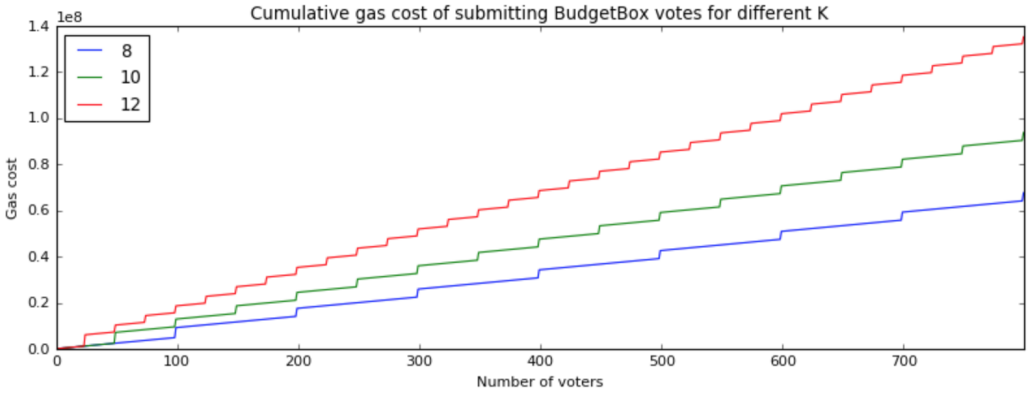


Figure A.5: Gas Costs for Voting