Quadratic funding amplifies the inequality problem. Projects with a majority number of votes will get an even larger percentage of the matching pool. Sometimes it can discourage participation (and potentially encouraging sybil attacks even more).

In a recent grant on HackerLink, the top project obtained a huge amount of votes (>70% of total votes committed to the whole round), and matched 95% of the total pool of matching fund. After some analysis, it was surprising that most of the contribution was not from sybil attacks, rather, the project mobilized community to vote for them.

After the project dominated the leaderboard with an absolute advantage, the grant round received very few submissions for almost two weeks. The project team finally made an announcement to give away up to 50% of their matching pool funding so that every other project can have a share from the round, and after that we saw an increase of submissions. The redistribution was done by giving away the support_area of that project to all projects that have received votes >=10. It was a one-off solution to the problem at that moment, but it inspired us to further solve the inequality problem.

The most straightforward mechanism is to use a progressive taxation system. There are multipole ways to implement a progressive tax system in a quadratic funding round. The easiest one is to design a static progressive tax ladder. After each round, every project is charged with certain amount of tax and later put into a public funding pool. Then the public funding pool can be distributed according to rules that a community think fair.

Here we describe an algorithm that redistributes wealth in each vote() function call.

When a project gains a vote, we define a support area tax ratio k as:

$$k=Max\left(\frac{a_p-A_{min}}{a_{0}},0\right)\cdot\frac{a_p}{S}$$

Where,

a_p

is the support area this project already had before this vote.

A_{min}

is the minimum support area that a tax will apply. It’s a constant. It avoids extreme tax charge in corner cases.

a_0

is the current support area of the project with the largest support area in a round.

S

is the total support area at this moment.

Based on k

, the new support area increase is:

$$\Delta a_{new}=\Delta a_{old}\cdot(1-k)^2$$

Where,

Δa_{old}

is the support area increase without tax.

A project will be heavily taxed when the following criteria is met:

1. The project is ranked on the top
2. The project has a dominating percentage of total support area

We can simulate a quadratic funding grant with progressive taxation. The result from round with progressive taxation is flattened.

RULES ==================================

[

截屏2021-06-15 下午9.59.27.png

1076×548 34.4 KB

](https://ethresear.ch/uploads/default/original/2X/e/ebc614fcb9e6ee2baa24db697e210afa29012e02.png)

Javascript code for simulation：

```
class Rule { constructor() { this.projects = [] for (let i = 0; i < 10; i++) { this.projects[i] = { vote: 0, area: 0, } }

this.area = 0

}

result() { for (let i = 0; i < this.projects.length; i++) { const p = this.projects[i] const s = (p.area / this.area * 10000).toFixed(2)

  const r = [this.area, p.area, p.vote]
  this.vote(i)
  const d = Number((p.area - r[1]).toFixed(2))
  const da = ((p.area / this.area * 10000) - Number(s)).toFixed(2)
  this.area = r[0]
  p.area = r[1]
  p.vote = r[2]

  console.log(`P${i + 1}: \t${r[2]} votes  \t${s} areas  \t${da} dA  \t${d} d`)
}

} }

class OldRules extends Rule { vote(i) { const index = Math.min(i, this.projects.length - 1) const p = this.projects[index]
this.area += p.vote p.area += p.vote p.vote += 1 } }

class Rules2 extends Rule { constructor() { super() this.top = 1 }

vote(i) { const index = Math.min(i, this.projects.length - 1) const p = this.projects[index]

const k = Math.max(Math.min(1, (p.area - 5000) / this.top), 0)
  * p.area / Math.max(1, this.area)
const added = p.vote * (1 - k) ** 2

this.area += added
p.area += added
p.vote += 1

if (p.area > this.top) {
  this.top = p.area
}

} }

function test(rules, votes, ps) { for (let k = 0; k < votes.length; k++) { const v = votes[k] const p = ps[k] for (let n = 0; n < v;
n++) { let i = 0 while (p[i] && Math.random() > p[i]) { i++ } rules.forEach(r => { r.vote(i) }) } } rules.forEach((r, i) => {
console.log(RULES ${i + 1} ====================================) r.result() console.log('') }) }

const AVERAGE = [1/10, 1/9, 1/8, 1/7, 1/6, 1/5, 1/4, 1/3, 1/2] const BSC = [2/7, 1.8/6, 1.6/5, 1.4/4, 1.2/3, 1/2, 1/2, 1/2, 1/2]
const AMASS = new Array(9).fill(1/2) const AAMASS = [0.76, ...new Array(8).fill(0.6)] const BAMASS = [0.9, ...new
Array(8).fill(0.6)] const ATTACK = [0.01, 0.9, ...new Array(7).fill(1/2)]

test([ new OldRules(), new Rules2(), ], [40000, 2000], [AAMASS, ATTACK])
```