Prerequisites: https://vitalik.ca/general/2019/12/07/quadratic.html and Pairwise coordination subsidies: a new quadratic funding design

- Let C_p

be the vector of sqrt-contributions to p

, so $C_p[i]$

equals the square root of the amount that i

contributed to project p

.

- Let $M_p = C_p^{\top}C_p$

(this is the square matrix where $M_p[i, j]$

equals $C_p[i] * C_p[j]$

) and let $M^* = \sum_p M_p$

- In normal

QF, we would calculate the subsidy as $\sum_{i \ne j} M_p[i, j]$

. In pairwise-bounded

QF, we would calculate the subsidy as $\sum_{i \ne j} \frac{M_p[i, j] * D}{D + M^*[i, j]}$

where D

is a bounding coefficient.

- Here, we instead let $N_p = M_p^2$

and $N^* = (\sum_p M_p)^2$

and calculate the subsidy as $\sqrt{\sum_{i \ne j} \frac{N_p[i, j] * D}{D + N^*[i,j]}}$

.

Here it is in code:

import numpy

BOUND_COEFF = 100

def get_subsidies(donation_lists): coeff_squares = [] for donations in donation_lists: sqrt_vec = numpy.matrix([[d**0.5 for d in donations]]) coeffs = numpy.transpose(sqrt_vec) * sqrt_vec coeffs_sq = coeffs ** 2 coeff_squares.append(coeffs_sq) o = [] for donations, squares in zip(donation_lists, coeff_squares): total_sub = 0 for i in range(len(donations)): for j in range(len(donations)): if j != i: total_sub += BOUND_COEFF * squares[i, j] / (BOUND_COEFF + sum([c[i,j]**0.5 for c in coeff_squares])**2) o.append(total_sub ** 0.5) return o

The idea is roughly as follows. The matrix $M_p = C_p^{\top}C_p$

counts pairs in C_o

that both support the same project. The matrix $N_p = M_p^2$

counts triplets

of users (x_1, x_2, x_3)

that all support the same project. The matrix $N^* = (\sum_p M_p)^2$

counts length-2 paths between different users (ie. A and B both support P1 and B and C both support P2), and is used to compute a “richer” measure of “closeness” between two participants that takes into account second-degree similarities.

The mechanism has some interesting properties (here we define get_subsidy(x)

as the single-project equivalent, equal to get_subsidies([x])[0]

):

- Low-end multiplicativeness

: get_subsidy(A * k) = k * get_subsidy(A)

for A

with small-sized elements (same as traditional QF and pairwise-bounded QF)

- Low-end square-root incentive curve

: get_subsidy(concat(A, [r])) - get_subsidy(A)

\approx k * \sqrt{r}

for small r

(same as traditional QF and pairwise-bounded QF)

- Split invariance

: sum(get_subsidies([A, A])) = get_subsidy(A * 2)

(same as traditional QF and pairwise-bounded QF)

- Monotonic partial derivatives

: get_subsidy(A + [r + d]) - get_subsidy(A + [r])

is increasing for small A

, r

, d

- Low-end N^1.5 returns

: get_subsidy([k] * N)

\approx N^{1.5}

for small k

and N

- N^1.5 max extraction from collusion

: get_subsidy([k] * N)

< c * N^{1.5}

(note that the strategy for doing this is itself involved and requires many projects; with a single project you can only extract O(N)

funds)

Thus, this version of QF seems to have all the nice properties of both traditional and pairwise, while having stronger properties against colluding attackers.

Note that there may be some flaw here, and indeed adding complexity generally increases risk of flaws, but it seems

to work fine.