Gas Efficient Ranked Choice Voting

Here we describe a method for selecting the Condorcet winner from a set of ballots suitable for implementation in a smart contract. To do so we guarantee that if a proposal can be created it can be voted on and executed within gas limits, by performing tallying in constant time over the number of votes cast. We provide a complete implementation of this method as a DAO DAO proposal module and formally verify the conditions for proposals being passed and rejected early.

Why Ranked Choice is Hard

The most common form of ranked choice voting, instant run off, works like this:

1. Voters submit a list of candidates sorted by their preference.
2. If there is an option with the majority of first-preference votes, that option is the winner.
3. Otherwise, remove the option with the fewest first-preference votes from all preference lists and repeat.

This algorithm presents a problem: the compute cost for tallying an election result increases with the number of votes cast (step 3 is at least $O(voters)$). On a blockchain this presents a problem as voting power is typically fungible tokens and can be split among many addresses by a single actor (like a sybil attack). Thus, to stop a proposal from being executable, an attacker can split their vote among more and more addresses until the number of votes cast causes tallying to hit compute limits, making it impossible to pass the proposal.

This is all to say: in order for a ranked choice voting system to be suitable for use in a smart contract the cost of tallying votes can't scale with the number of votes cast.

The Condorcet Method

> Without looking terribly deeply into the history of it, this appears to have been first described by Paul Cuff Et al. [1] in 2012. We did not come up with this idea.

A Condorcet winner is a candidate in an election who would win a 1v1 with every other candidate. If we assume that voters won't change their relative preferences when candidates are removed we can find the Condorcet winner in a set of ranked choice ballots. For example:

```
[a, b, c]
[b, a, c]
```

```
[c, a, b]
```

Under this assumption, to see who would win in a 1v1 we can remove all other candidates from the ballots and compare them using majority-wins.

```
a vs b  |  a vs c  |  a vs b vs c
        |          |
[a, b]  |  [a, c]  |  [a, b, c]
[b, a]  |  [a, c]  |  [b, a, c]
[a, b]  |  [c, a]  |  [c, a, b]
```

In this example, under the reordering assumption, `a` is a Condorcet winner.

Now, In Constant Time

In order to implement the Condorcet Method in a smart contract, we have two requirements.

1. Tallying results is constant time over the number of votes cast.
2. All created proposals can be executed and voted on within gas limits, i.e. `gas(proposal_creation) >= gas(vote) && gas(proposal_creation) >= gas(execute)`.

Constant Time Over Votes Cast

Let C be the set of candidates and V be the set of voters and their vote weights such that $\forall v \in V\ v[c]$ is the weight given to candidate c by voter v. we can then say that c_i is a Condorcet winner if [1]:

$$\forall c \in C, c \neq c_i \implies 2 * ||\{v \mid v \in V, v[c_i] > v[c]\}|| > ||V||$$

From here, we set our sights on pre-computing this value whenever a vote is cast. to do so, we define a new matrix M:

$$M[i][j] := ||\{(a,b) \mid a \in V, b \in V, a \neq b, a[i] > a[j]\}|| - ||\{(a,b) \mid a \in V, b \in V, a \neq b, a[j] > a[i]\}||$$

The math notation here is unpleasant, so in pseudocode:

```
M[i][j] = number_of_times_i_has_beaten_j - number_of_times_j_has_beaten_i
```

Given such a matrix, a candidate c_i is a winner if:

$$\forall p \in M[c_i], p > 0$$

In english: a candidate wins if the number of times they were preferred more-often-than-not in a 1v1 with every other candidate.

The complexity of finding this candidate is $O(candidates^2)$ which does not scale with the number of votes so this satisfies our requirement that tallying doesn't scale with number of votes cast.

This runtime is made clear by in this example implementation of this method:

```
pub struct Condorcet {
    m: Vec<Vec<i32>>,
}

impl Condorcet {
    pub fn new(candidates: usize) -> Self {
        Self {
            m: vec![vec![0; candidates]; candidates],
        }
    }

    pub fn vote(&mut self, preferences: Vec<usize>) {
        for (index, preference) in preferences.iter().enumerate() {
            // increment every value in self.m[preference] which
            // appears in preferences[index + 1..] as preference is
            // ranked higher.
            for victory in (index + 1)..preferences.len() {
                self.m[*preference][preferences[victory]] += 1
            }
            // decrement every value in self.m[preference] which
            // appears in preferences[0..index] as perference is
            // ranked lower.
            for defeat in 0..index {
                self.m[*preference][preferences[defeat]] -= 1
            }
        }
    }

    pub fn winner(&self) -> Option<usize> {
        // a winner is someone who wins a majority runoff with all of
        // the other candidates.
        self.m
            .iter()
            .enumerate()
            .find(|(index, row)| row.iter().skip_nth(*index).all(|&p| p >
0))
            .map(|(index, _)| index)
    }
}
```

Created Proposals Can Be Executed

Our second requirement for our system is that all proposals that can be created can also be voted on and executed. The solution to this lacks math theory, and is primarily an implementation detail, though an important one.

At a high level, to accomplish this:

1. We divide up state such that more state is read and written when creating a proposal than when executing and voting on it.
2. When a proposal is created we perform a tally on the empty M matrix to ensure that the gas cost in terms of compute of creating a proposal is >= voting on a proposal as the only compute needed when voting is a tally.

A proof of this requirement being met for our implementation can be found here.

Diagonalization

An interesting property of our M matrix is that it is reflected across the line $y = x$.

$$M[i][j] = -M[j][i]$$

In english:

```
M[i][j] = number_of_times_i_has_beaten_j - number_of_times_j_has_beaten_i
M[j][i] = number_of_times_j_has_beaten_i - number_of_times_i_has_beaten_j
M[i][j] = -M[j][i]
```

This means that we can decrease the size of M in storage to $\frac{N(N-1)}{2}$ by storing only the upper diagonal of the matrix in a vector V. To translate an index in M to an index in V:

$$index(x, y) = y * N - (y + 1) * y/2 + x - (y + 1)$$

And to retrieve a value from M:

$$get(x, y) = \begin{cases} \neg get(y, x), & \text{if } x < y \\ V[index(x, y)], & \text{otherwise} \end{cases}$$

This indexing method can be hard to understand without working out on paper. Here's our implementation of this with some comments which may help build intuition:

```rust
fn index(&self, (x, y): (u32, u32)) -> u32 {
        let n = self.n;
        // the start of the row in `self.cells`.
        //
        // the easiest way to conceptualize this is
        // geometrically. `y*n` is the area of the whole matrix up to
        // row `y`, and thus the start index of the row if
        // `self.cells` was not diagonalized [1]. `(y + 1) * y / 2` is the
        // area of the space that is not in the upper diagonal.
        //
        // whole_area - area_of_non_diagonal = area_of_diagonal
        //
        // because we're in the land of discrete math and things are
        // zero-indexed, area_of_diagonal == start_of_row.
        let row = y * n - (y + 1) * y / 2;
        // we know that x > y, so to get the index in `self.cells` we
        // offset x by the distance of x from the line x = y (the
        // diagonal), as `self.cells`' first index corresponds to the
        // first item in that row of the upper diagonal.
        let offset = x - (y + 1);
        row + offset
}
```

Complexity Conclusion

This completes a description of an implementation of the Condorcet Method that is constant time over the number of votes cast, storage efficient, and that guarantees that created proposals can be voted on.

Passing Proposals Early

A proposal can be cast early if there is currently a winner, and no sequence of votes can change that winner.

For the Condorcet Method, there is an undisputed winner if there is a winner who's smallest margin of victory in a 1v1 is larger than all outstanding voting power. For example, if candidate `a` is winning their closest 1v1 by a margin of 6 votes, and there are 5 votes outstanding, candidate `a` is the undisputed winner.

In terms of our M matrix, there is a column with only positive values, and the minimum value in that column is > the outstanding voting power.

The Filibuster

A side-effect of this is that a group of voters who's voting power is larger than the smallest margin of the winning candidate can attempt to collude to prevent a proposal from having a winner by ranking the winning candidate last on all of their ballots. For example, if `c` was the current winner, an example ballot modification might be `[a, c, b] -> [a, b, c].

Fortunately, as discussed here, if the other voters catch on to this they may also modify their ballots in the opposite way by always ranking `c > a` and thus prevent the filibuster and still electing the Condorcet winner.

Practically speaking, the takeaway here is that this method biases itself towards no outcome. A relatively small group of voters preferring no action to the winning option may cause no winner to be selected unless the larger voting body is active in their resistance to this happening.

Strategic Voting

This filibustering is a form of strategic voting wherein voters vote against their true preferences to cause an outcome. What about other forms of strategic voting?

Fortunately, we appear to be quite safe. There is a rather large collection of literature which explores and confirms the resistance of Condorcet methods to strategic voting (see: 2, 3, 4).

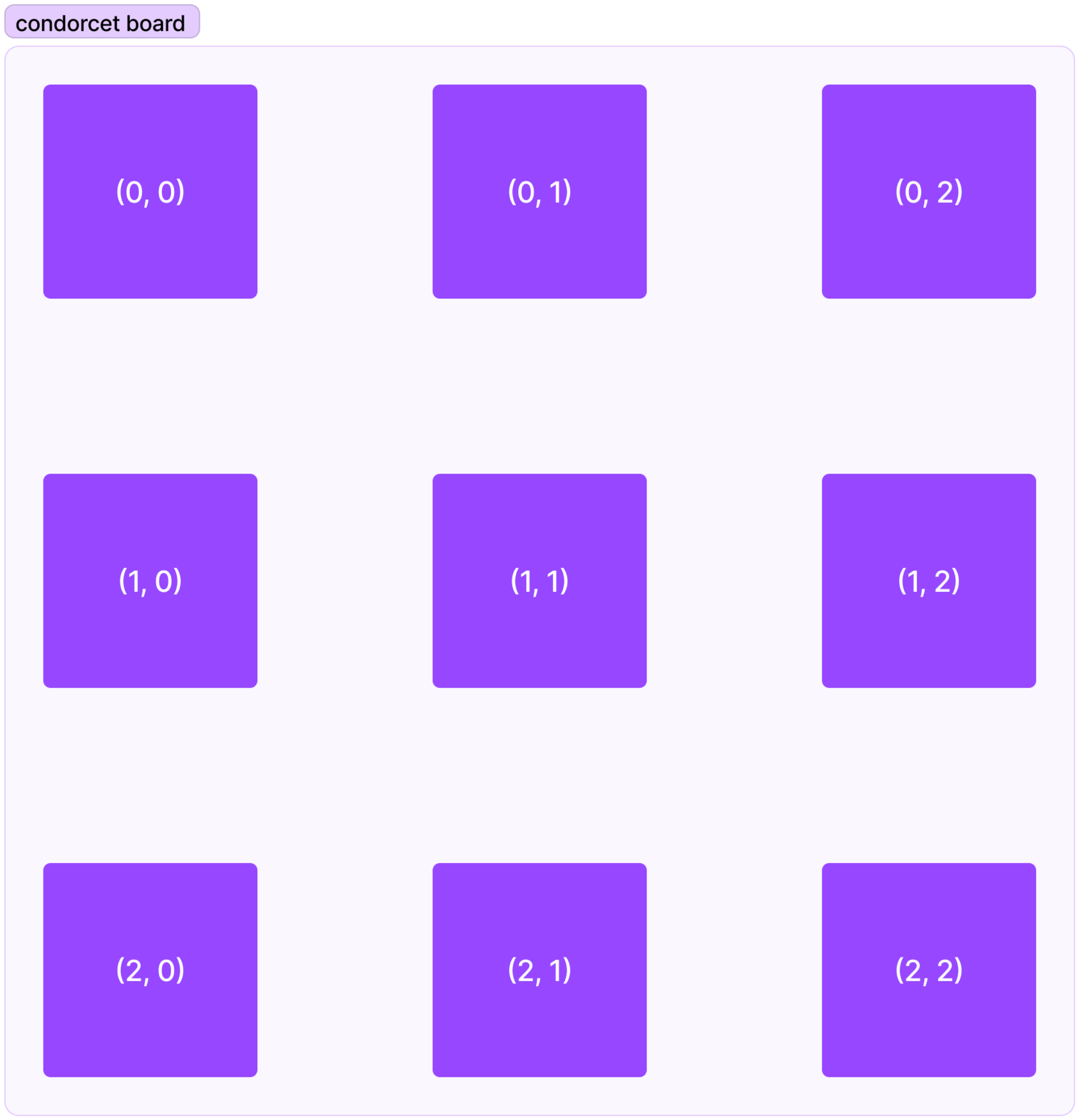
We do not claim to be experts on voting systems and their math, so it is quite possible that there exist unknown unknowns here.

Strategic Voting Intuition

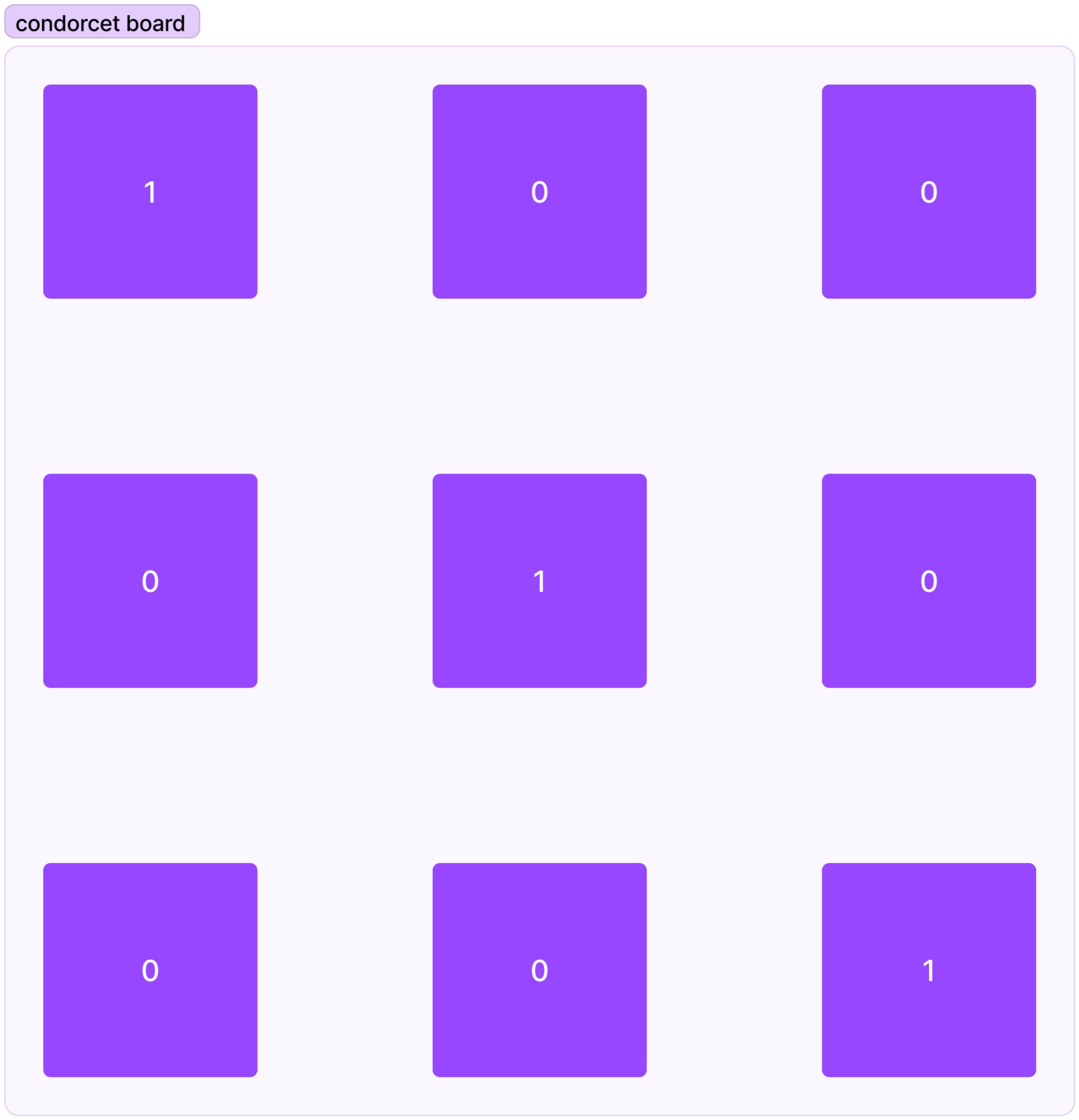
To help build intuition about strategic voting, we designed a game for which winning strategies will also be strategies for strategic voting. We were unable to come up with any strategies other than the filibuster described earlier.

The Game

This game takes place on a $N \times N$ game board. Each tile on the board can be identified by its (x, y) coordinate.



The game begins with all tiles having a value of zero, except for those along the diagonal which begin with a value of one. The game ends when there is a row or column that contains only positive, non-zero values.

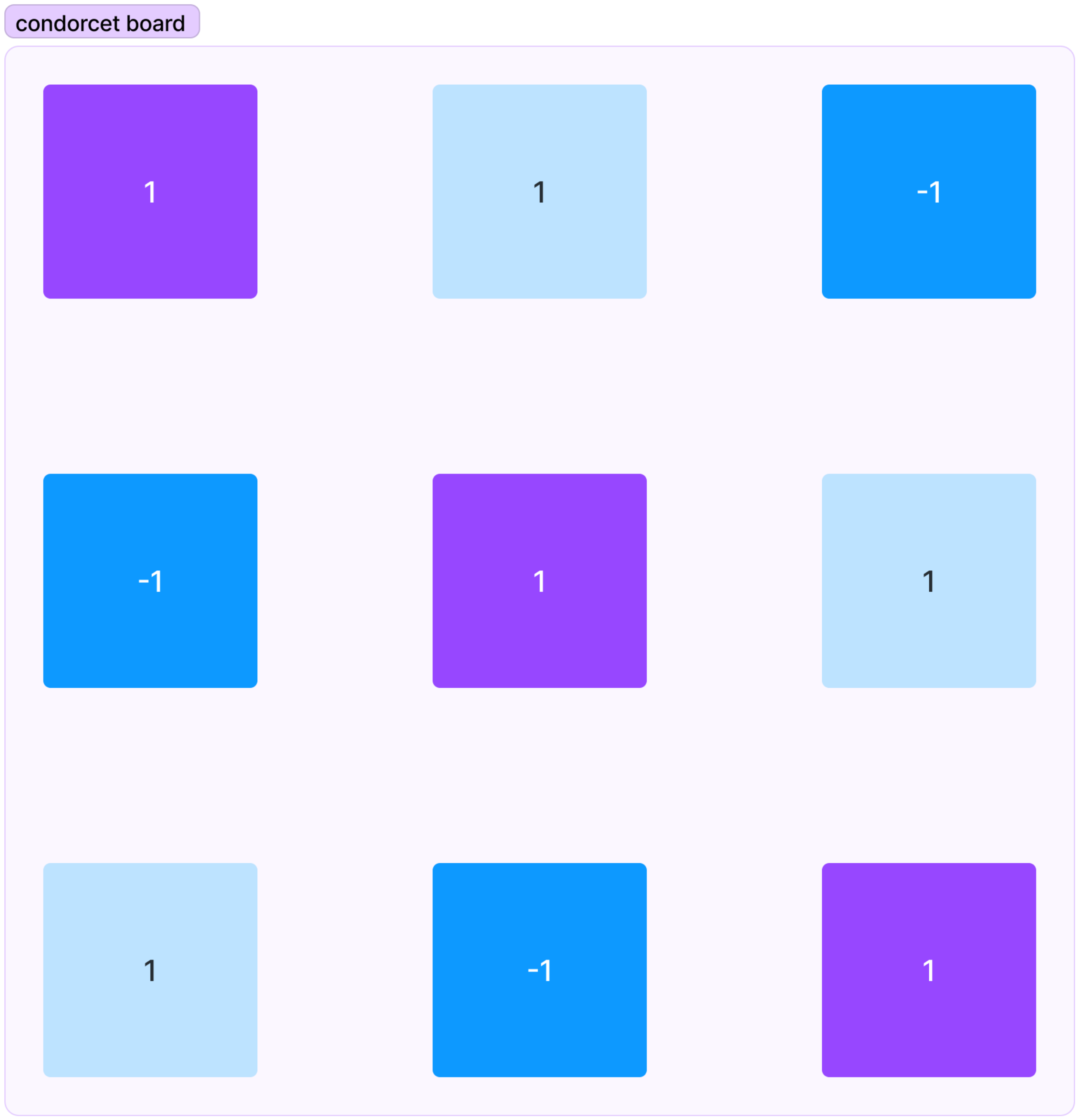


condorcet board

1	0	0
0	1	0
0	0	1

On your turn, you select $\frac{N(N-1)}{2}$ tiles and add one to their value. Whenever you add one to tile (x, y), the value of tile's "mirror", (y, x), decreases by one. The only constraints on your move are that the the tiles you select must be unique, and you may not select both a tile and its mirror.

Here's an example move on our board. The tiles colored light blue have had one added to them, the tiles colored dark blue are their mirrors and have had one subtracted from them.



You are playing with a non-zero number of players who make moves randomly. What strategy would you use if your goal was to keep the game from ending?

In terms our of method, the game board is M and on each round you cast a vote to attempt to prevent the proposal from completing.

Rejecting Proposals Early

Here we prove the conditions for early proposal rejection. Reproduced from this wiki page.

Definitions

- A Condorcet matrix M that represents a set of ranked choices of N given choices is an NxN matrix, where the value of a cell [m][n] = (the number of times m has been ranked over n) - (the number of times n has been ranked over m).

- A Condorcet Winner is the candidate, given a set of ranked choices of candidates, that would beat every other candidate in a head-to-head race. That means that given a pairwise comparison of every candidate to every other candidate, the candidate that is preferred in the majority of cases to each of the others is the Condorcet winner.

- A Condorcet winner in a Condorcet Matrix will therefore be the column C in M that has a positive value in every cell except for the cell which corresponds to itself.

- A Condorcet paradox (or a Condorcet cycle) is a situation wherein no candidate is majority-preferred to every other candidate in a pairwise comparison. In the Condorcet Matrix, this would appear as a matrix with no rows or columns that have all positive values.

- For a column col, let **distance_from_positivity**(col) = sum(1 + abs(v) for v in col if v <= 0). Let min_distance_from_positivity = min(distance_from_positivity(col) for col in M), and let MIN_COL be the column for which distance_from_positivity(MIN_COL) = min_distance_from_positivity.

- For a column col, let **max_negative_magnitude**(col) = max(abs(v) for v in col if v <= 0)).

- Let **power_outstanding** be the remaining voting power to be cast. Every unit of voting power corresponds to one ballot.

- When a vote is cast with voting power V, and its highest ranked choice is m, then V is added to every cell for the column of m excluding the cell for m itself. This is because for each of the N-1 choices besides m, there are now V more times that m has been ranked over that choice. Therefore, V*(N-1) is added in total to m's column.

It helps to visualize the following Condorcet Paradox when thinking through these proofs:

```
Choices: A, B, C

Ranked votes:
```

```
ABC
BCA
CAB
```

Condorcet matrix:

	A	B	C
A	0	1	-1
B	-1	0	1
C	1	-1	0

Claims and proofs

```
Claim A: In a matrix with a Condorcet Paradox, there will be no Condorcet
Winner found if min_distance_from_positivity > power_outstanding * (N-1).

Claim B: In a matrix with a Condorcet Paradox, there will be no Condorcet
Winner found if for every column col such that distance_from_positivity(col)
<= power_outstanding * (N-1), max_negative_magnitude(col) >=
power_outstanding.
```

Proof Of Claim A

```
Our premise is that there is a matrix in a Condorcet Paradox M such that
min_distance_from_positivity > power_outstanding * (N-1).

Assume that there is a way that you can create a Condorcet Winner in matrix
M using the remaining voting power.

Let this Condorcet Winner be W, and its corresponding column in M be C, and
let C' be C's state after it becomes the Condorcet Winner.

let X = distance_from_positivity(C).

If C' is the Condorcet Winner, then power_outstanding * (N-1) >= X.

min_distance_from_positivity must be <= X by definition, so
min_distance_from_positivity <= X <= power_outstanding * (N-1).
```

```
But this contradicts our premise that min_distance_from_positivity >
power_outstanding * (N-1).

Therefore, if min_distance_from_positivity > power_outstanding * (N-1), then
there can be no possible Condorcet Winner in M.
```

Proof Of Claim B

```
Our premise is that there is a matrix in a Condorcet Paradox M such that for
every column col such that distance_from_positivity(col) <=
power_outstanding * (N-1), max_negative_magnitude(col) >= power_outstanding.

Let C be a column in M such that distance_from_positivity(C) <=
power_outstanding * (N-1) and max_negative_magnitude(C) >=
power_outstanding.

Assume that C can be made a Condorcet Winner. Let C' be C's state after it
becomes the Condorcet Winner.

Let Y = max_negative_magnitude(C), and z be the corresponding cell for which
C[z] = Y.

Let p = C'[z] - C[z]. p must be > Y since C'[z] must be positive by the
definition of Condorcet Winner.

p must be <= power_outstanding, since the most that C[z] could have been
increased by is power_outstanding.

So this means power_outstanding >= p > Y.


But this contradicts our premise that for every column col such that
distance_from_positivity(col) <= power_outstanding * (N-1),
max_negative_magnitude(col) >= power_outstanding.

Therefore, if for every column col such that distance_from_positivity(col)
<= power_outstanding * (N-1), max_negative_magnitude(col) >=
power_outstanding, then there can be no possible Condorcet Winner in M.
```

Conclusion

This GitHub repository has a complete implementation of the system described here.

---

1. Note that the condition for winning a majority can be written as $2 * votes > total$.↩