COMPUTATIONAL COMPLEXITY CHARACTERIZATION OF PROTECTING ELECTIONS FROM BRIBERY

KAYLA HUANG, AUSTIN LI, NATHAN SUN, AND CHRISTINA XIAO

Abstract. Electoral bribery is an important normative and empirical issue that has abounded through history and the present. Following the work of Chen et al. [1], we consider the destructive protective problem: a defender with a given defense budget pays voters such that they can no longer be attacked, and an attacker attempts to bribe undefended voters to change their preferences such that the true winning candidate no longer wins. We extend the proof of this problem as NP-complete under r-approval to the voting rule of Borda count. We further verify the problem's complexity experimentally for both r-approval and Borda count. Finally, we consider an approximation of the destructive protective problem under both voting rules and find promising polynomial results with implications for governments wishing to protect their elections from bribery.

1. Introduction

We often think of voting in elections as the premier act of political participation [2] — without which democracy would crumble. From a normative political theory perspective, electoral bribery has deeply uncomfortable implications for democracy. Yet empirical evidence for electoral bribery — the act of influencing voters' ballots with money, goods, or favors — has abounded through history and the present. In 18th-century England, candidates promised to pay voters' hefty property taxes in exchange for their vote [4]. American colonies inherited this tradition: especially in the South, candidates treated voters to large amounts of food and drink to secure votes [4]. Today, electoral bribery remains a prominent issue in the Philippines [5], Latin America [6], and West Africa [7].

Contemporary electoral bribery has evolved strategy as well. In the Philippines, electoral attackers target well-connected individuals when social networks are dense, and switch over to people with reciprocal personalities when social networks become sparse [5]. From the Gibbard–Satterthwaite theorem, we know that strategy-proofness is a difficult ideal to balance with notions of fairness. Thus, rather than attempt to design an electoral system that cannot be manipulated by either voters or brokers, we aim to protect existing non-strategy-proof voting systems from bribery as best we can.

We introduce the following problems based on the work of Chen et al. [1]. In the *bribery problem*, an attacker attempts to manipulate the election by bribing some voters into reporting preferences of the attacker's choice (rather than their true preferences). Each voter has a price for being bribed, and the attacker has a budget for bribing voters. We focus specifically on the bribery problem with a *destructive attacker*, who attempts to make the true winning candidate — the candidate who would have otherwise won the election in the attacker's absence — lose the election.

This provides the background for the *protection problem*. A defender aims to protect the election from bribery. They are given a defense budget to use to award a subset of voters so that they cannot be bribed by the attacker. We ask: *Given this defense budget, is it possible*

Date: February 8, 2024.

for the defender to protect the election (i.e. assure that the attacker cannot influence the result of the election)?

Chen et al. [1] proves that this destructive protection problem, when voters are assigned weights and arbitrary prices, is NP-complete for r-approval. In this paper, we extend this result to the voting rule of Borda count and verify its complexity experimentally. We also simulate an approximation of the protection problem and examine its experimental complexity to evaluate this problem practically, with promising suggestions for governments seeking to protect their elections.

2. Problem definition

We will follow the notation given by Chen et al. [1]. We consider m candidates $\mathcal{C} = \{c_1, c_2, \ldots, c_m\}$ and n voters $\mathcal{V} = \{v_1, v_2, \ldots, v_n\}$. Each voter v_j has a preference list τ_j, which is a permutation over candidates.

Moreover, we consider an attacker that is not in $\mathcal{C} \cup \mathcal{V}$ but manipulates the election by bribing voters. Each voter v_j has *bribing price* p_j^b. If v_j receives p_j^b, then the voter will change their preference list to the list given by the attacker. The attacker has budget B. In the *protection problem*, we additionally consider a defender with budget F. Each voter v_j also has *awarding price* p_j^a. Upon receiving their awarding price by the defender, the voter will report their preferences truthfully and *cannot be bribed*.

We now define the Borda count voting rule.

Definition 2.1 (Borda count)**.** The *Borda count* is a family of positional voting rules where candidates receive points corresponding to their rank. In the original Borda count variant, a candidate in the kth position gets $n - k$ points, where n is the total number of candidates.

We will let n-Borda denote the Borda count with n candidates. In this paper, we will consider *normalized* n-Borda scoring.

Definition 2.2 (Normalized n-Borda)**.** In n-Borda with *normalized scores*, the candidate in the kth position gets score $\frac{n-k}{n-1}$. We simply divide scores in the original variant by $n - 1$.

2.1. Problem statement.

We formalize the destructive protection problem as follows.

Definition 2.3 (Destructive protection problem)**.** Consider a set $\mathcal{C}$ of m candidates and a set of $\mathcal{V}$ of n voters. Each voter v_j has a preference list of candidates denoted τ_j, an awarding price $p_j^a \in \mathbb{Z}_{\geq 0}$ and a bribing price of $p_j^b \in \mathbb{Z}_{\geq 0}$. Consider a scoring rule for selecting a winner. There is a defender with a defense budget $F \in \mathbb{Z}_{\geq 0}$ and an attacker with attack budget $B \in \mathbb{Z}_{\geq 0}$ attempting to make candidate c_m lose the election.

We seek to decide whether there is a subset $\mathcal{V}_F \subseteq \mathcal{V}$:

- $\sum_{j:v_j \in \mathcal{V}_F} p_j^a \leq F$ and
- for any subset $\mathcal{V}_B \subseteq \mathcal{V} \setminus \mathcal{V}_F$ with $\sum_{j:v_j \in \mathcal{V}_B} p_j^b \leq B$, no candidate $c \in \mathcal{C} \setminus \{c_m\}$ can get a strictly higher score than c_m despite the attacker bribing $\mathcal{V}_B$.

3. Theoretical result

Chen et al. [1] proved that the r-approval weighted destruction problem is NP-complete (specifically when $r \geq 3$), but not much work has been done to show a similar result for Borda count. We will show a similar result holds for $n \geq 4$.

Theorem 3.1. *The Borda count destructive weighted-problem is NP-hard for any $n \geq 4$.*

Proof. We prove the above for $n = 4$. Note that the case of $n > 4$ can be addressed by introducing dummy candidates and letting each voter vote for $n-4$ distinct dummy candidates.

We follow a similar logic as introduced in [1]. We first draw a reduction from 3DM, which has been shown by [3] to be NP-hard.

We reduce from a variant of the 3DM problem where every element appears at most $d = O(1)$ times. Given a 3DM instance with $3\zeta = |W \cup X \cup Y|$ elements and $\eta = |M|$ triples so that every element appears at most $d = O(1)$ times in M. Without loss of generality, we assume that $\eta \geq \zeta + 2d$.

Now we re-index the elements in $W \cup X \cup Y$ as $z_1, z_2, \cdots, z_{3\zeta}$. Then let our weight $Q = 2\eta+1$ with $3\zeta+1$ key candidates. Here we can set Q to be large enough such that only the key voters (which will be defined later) will be considered by the attacker or defender, so we can consider the case where there are two different weights: one of Q and the other with unit weight. We have two kinds of candidates

- 3ζ be element candidates. Call then $c_1, \ldots, c_{3\zeta}$, which correspond to $z_1, \ldots, z_{3\zeta}$, each with a score of $Q \cdot f(z_i)$;
- one leading candidate, who is the original winner with a score of $Q \cdot f(z_{3\zeta+1})$. Call this candidate $c_{3\zeta+1}$.

Note $c_{3\zeta+1} \notin W \cup X \cup Y$. In our construction, we also let there be sufficiently many dummy candidates indexed c_i, where $i > 3\zeta + 1$. Each dummy candidate has has score $\frac{1}{n}$.

Now let there be η key voters, indexed from $v_1, \ldots v_\eta$ of weight Q. These key voters give nonzero scores for 3 candidates under 4-Borda, so let each voter vote for a triple $(z_i, z_j, z_k) \in M$, where z_i represents the candidate with score 1 given by this voter, z_j represents the candidate with score $\frac{2}{3}$ given by this voter, and z_k represents the candidate with score $\frac{1}{3}$ given by this voter. Besides the key voters, there are also sufficiently many dummy voters v_i for $i > \eta$ with unit weight. Each dummy voter votes for one key candidate and two distinct dummy candidates.

Let the attack budget be $F = \zeta$ and the defense budget be $B = \eta - \zeta$. We define the *vote disparity* to be

$$\Delta_{ij} = \begin{cases} m, & \text{if } v_j \text{ gives } c_{3\zeta+1} \text{ score } m \\ 1 - m', & \text{additionally if } v_j \text{ gives } c_i \text{ gets score } m' \end{cases} \tag{1}$$

Note that $\Delta_{ij} \in [0, 2]$ under normalized Borda scores.

Further, as defined in [1], let

$$\Delta_{max} = \max_{1 \leq i \leq 3\zeta} \sum_{j=1}^{\eta} \Delta_{ij}, \qquad d_{max} = \max_{1 \leq i \leq 3\zeta} d(z_i). \tag{2}$$

Now we define

$$f(z_i) = 2\eta + d_{max} + \Delta_{max} - \sum_{j=1}^{n} \Delta_{ij}, \qquad f(z_{3\zeta+1}) = 2\eta + d_{max} + \Delta_{max} - \zeta + \frac{1}{n}. \tag{3}$$

Observe that

$$\sum_{j=1}^{\eta} \Delta_{ij} \geq \eta - d > \zeta \implies f(z_{3\zeta+1}) > f(z_i) \tag{4}$$

as every element appears at most d times in triples, and thus $c_{3\zeta+1}$ is indeed original winner.

Negative instance of 3DM implies negative instance of destructive protection. We

first show that a negative instance of 3DM implies a negative instance of the destructive-weighted protection problem.

Suppose the 3DM instance does not admit a perfect matching. Let U to be the subset of key voters who are fixed by the defender. Without loss of generality, because the defender has the budget to, we consider a larger set U', where $|U'| = \zeta$. Let the attacker bribe $\zeta - \eta$ voters.

With bribery, the score of key candidate c_i becomes

$$Q(f(z_i) + \sum_{j=1}^{\eta} \Delta_{ij}) = Q(f(z_{3\zeta+1}) + \zeta - \frac{1}{n}). \tag{5}$$

But there are key voters who aren't able to be bribed, so they must be subtracted from the score of key candidate c_i. Since there does not exist a perfect matching in $W \cup X \cup Y$, there exists some z_k in two triples — let this be candidate c_k.

Since at least two voters gave nonzero scores for candidate c_k,

$$\sum_{j:v_j \in V} \Delta_{kj} \leq \zeta - \frac{2}{n} \tag{6}$$

So subtracting the key voters who aren't bribed implies that the score of the key candidate c_i becomes at least $Q(f(z_{3\zeta+1}) + \frac{1}{n})$, implying that after bribery c_i will get a higher score than $c_{3\zeta+1}$. And hence it becomes impossible to protect against bribery.

Positive instance of 3DM implies positive instance of destructive protection. Now we show that a positive instance of 3DM implies a positive instance of the destructive-weighted protection problem.

Suppose the 3DM instance admits a solution. Let $T \subseteq M$ be the perfect matching. So $|T| = \zeta$ and let the defender protect the triples in T. Recall each dummy candidate has score $\frac{1}{n}$. If the attacker bribes all key voters, the score of each dummy candidate will increase by $Q\sum_{j=1}^{\eta} \Delta_{ij}$. Note that

$$\frac{1}{n} + Q\sum_{j=1}^{\eta} \Delta_{ij} \leq 2\eta Q + \frac{1}{n} \leq Qf(z_{3\zeta+1}). \tag{7}$$

This demonstrates that no dummy candidate can win. For each key candidate, their score after bribery will be $Q(f(z_{3\zeta+1}) + \zeta - \frac{1}{n})$. But since the defender has protected the voters in T, $Q\zeta$ must be subtracted. So each key candidate's final score is at most $Q(f(z_{3\zeta+1}) - \frac{1}{n})$. Therefore no key candidate can win either, and hence the defender can protect against a destructive attacker. □

4. Experimental verification result

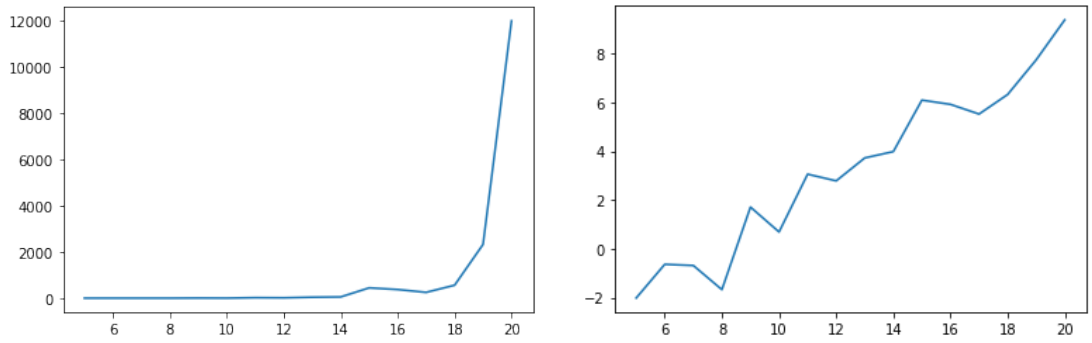
4.1. **Brute force algorithm overview.** To verify NP-completeness for the destructive protection problem under r-approval and Borda count, we consider a brute force algorithm. We iterate through defending every possible subset of voters. If protection is possible given the defending budget, we then iterate through bribing every possible subset of voters. We consider the election to be successfully defended when a defended subset, for all possible bribed subsets, always results in the true winner winning the election.

All code for experimental verification of NP-completeness can be found in Appendix A.

4.2. **Results for r-approval.** We iterate through all possible defense budgets for each simulation of $n \in [5, 20]$ voters to determine the first point at which the election can be successfully defended, upon which the simulation terminates.

Algorithm 1 Brute force algorithm for bribing problem

```
for every subset size s do
    for every subset in the set of all possible subsets of size s do
        if subset is defendable given the budget then
            if voters cannot be bribed given these defended voters then
                return true
            end if
        end if
    end for
end for
return false
```



(A) Run time results for r-approval with brute force algorithm with increasing number of voters.

(B) Log run time results for r-approval with brute force algorithm.

FIGURE 1. Run time for r-approval simulations.

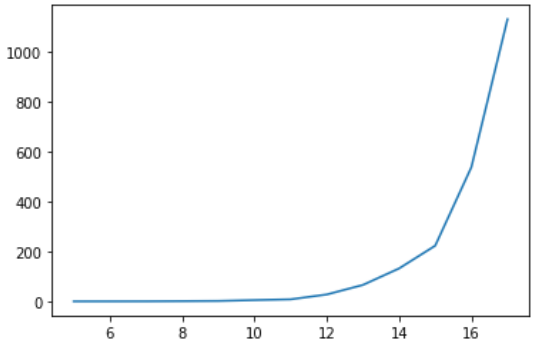
As seen in Figure 1a, the run time for 20 voters is around 3 hours and 20 minutes. Our machines were unable to provide results for $n \geq 21$. Clearly, the run time increases at a high rate.

To verify exponential growth, we plot the log of the run times in Figure 1b. This is roughly linear, though there is noise due to the randomization of voters. This confirms our belief that the brute force algorithm is approximately exponential in the number of voters.

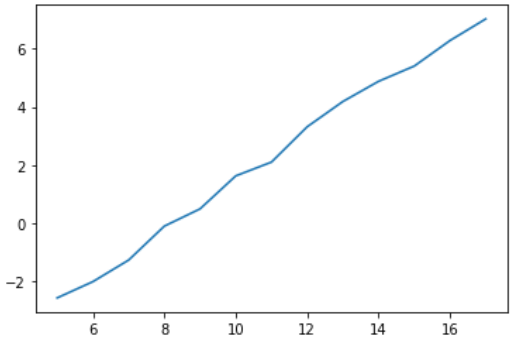
4.3. **Results on Borda count.** Results for Borda, as expected, were more computationally expensive than r-approval. Our machines lacked sufficient memory for $n \geq 18$. The results for $n \in [5, 17]$ are averaged over four iterations to produce the plots.

It is clear that the run time of this algorithm grows near-exponentially in the number of voters. To verify this claim, we plot the log of the run times in Figure 2b This is roughly linear. We can conclude that the run time of the brute force method increases exponentially with the number of voters and is practically infeasible.

We confirm that the destructive protection problem, for both r-approval and Borda count, is exponentially intractable. This has unfortunate implications for the practical implementation by governments of methods to protect against bribery.



(A) Run time results for Borda count with brute force algorithm with varying number of voters.



(B) Log run time results for Borda count with brute force algorithm.

FIGURE 2. Run time for Borda simulations.

5. Experimental approximation result

Political science research reveals strategies employed by attackers [5]. This gives us reason to believe that governments may approximate a solution to the destructive protection problem by focusing on simple greedy strategies.

5.1. **Greedy approximation algorithm for bribing problem.** We approximate the bribing problem with a greedy algorithm designed to bribe as many voters as possible into supporting the true second-place candidate and not supporting the true winner.

First, sort voters based on a combination of their bribe price, weight, and how highly they rank the true winner (this is considered irrelevant for r-approval, as all approved candidates gain the same number of points). This approximately orders voters based on how "good" they are to bribe; low-cost, high-weight voters who highly rank the true winner are ideal bribery victims.

Iterate through these sorted voters and if the attacker has enough money and the voter is not defended, bribe the voter to move the true second-place candidate to the beginning of their preference and the true winner to the end. The only exception is in r-approval with voters who already approve of the true second-place candidate and do not for the true winner.

Algorithm 2 Greedy approximation for bribing problem

```
initialize briber with budget
sort voters
for each voter v do
    if v is not defended and budget is enough to bribe v then
        bribe v to change preference
        decrease budget
        find the winner w with new voter preferences
        if w is not true winner then
            return true
        end if
    end if
end for
```

5.2. **Greedy approximation algorithm for defending problem.** Now, we assume that the defending government has some idea of the true winner and the attacker's strategy. We approximate the defending problem with the following greedy algorithm.

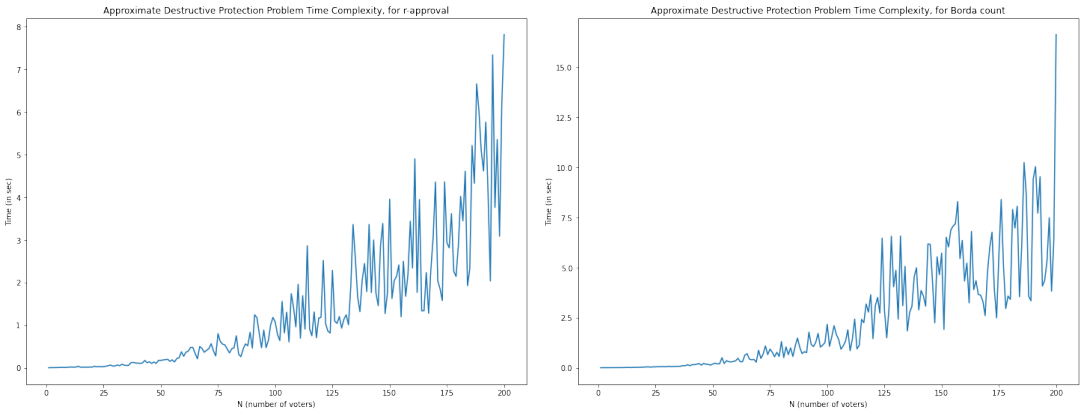
Sort the voters similarly to the attacking problem. Iterate through these voters: if the defender has enough money and the voter supports the true winner (in r-approval, approves of; in Borda count, places in the first half of their preference), defend the voter. After all these "true winner-supporting" voters are defended, iterate through the remaining voters in order and defend them if there is money left to do so.

Algorithm 3 Greedy approximation for defending problem

```
initialize defender with budget
sort voters
for each voter v do
    if v highly supports true winner and budget is enough to defend v then
        defend v
        decrease budget
    end if
end for
if budget ≥ 0 then
    for each voter v do
        if v is not defended and budget is enough to defend v then
            defend v
            decrease budget
        end if
    end for
end if
```

5.3. **Simulation and results.** As in the prior section, we simulate n random voters in a setting with 5 candidates. We set the budget for the attacker at the maximum needed to bribe all voters and slowly decrease it. Within each of these loops, the defender's budget begins at 0 and slowly increases, to represent how a government may think about responding to an attacker by increasing their spending on electoral protection. For $n \in [1, 200]$, we time the entire process and only stop when the defender succeeds in preventing the attacker from succeeding.

All code for this experimental approximation can be found in Appendix B. Here are the resulting plots for r-approval (where $r = 3$) and Borda count:



5.4. **Discussion.** We see that the approximations for both voting rules seem to exhibit polynomial patterns of time complexity. Borda count appears to reach twice as large a magnitude of time per run; this is expected, as Borda count is more expensive to compute than r-approval. The polynomial behavior is also somewhat expected due to the greedy algorithms keeping most computations polynomial in the size of the number of voters.

This experimental approximation that runs in polynomial time is a great improvement upon the NP-completeness of the true destructive protection problem, and much more feasible for real-world governments to attempt. The requirements for this method to be attempted are our assumptions when designing the defending greedy algorithm: knowledge of the true winner and aspects of the attacker's strategy. For governments attempting to protect their elections from bribery, public opinion polling to determine the truly preferred winner before any electoral bribery occurs and research into the strategies of electoral attackers like Ravanilla et al.'s [5] may be the difference between a NP-complete and a polynomial problem.

6. Conclusion and further research

In this paper, we have shown the destructive weighted-protection problem to be NP-complete for Borda count. We further verified these results for both r-approval and Borda count experimentally. Finally, we designed and simulated a greedy approximation of the destructive protection problem, in which the problem can be reduced to polynomial time based on a few key assumptions and strategic maneuvers.

There remain a great deal of results to prove in the electoral bribery space [1]. Notably, we have only been considering the destructive attacker case, ignoring the case of the constructive attacker, who attempts to get a candidate of their choosing to win the election. We know that the constructive protection problem is harder than the destructive one [1]. Determining an approximation for the constructive protection problem may prove even more useful for governments.

We also only prove this result for Borda count. Recall that the Borda count voting rule is in a larger class of positional voting rules. We note that an extension of this work to *all positional voting rules* is much stronger and general.

References

[1] L. Chen, A. I. Sunny, L. Xu, S. Xu, Z. Gao, Y. Lu, W. Shi, and N. Shah. Computational complexity characterization of protecting elections from bribery. *Theoretical Computer Science*, 891:189–209, 2021.

[2] R. J. Dalton. *Citizen Politics: Public Opinion and Political Parties in Advanced Industrial Democracies*, chapter 3, pages 39–64. CQ Press, Thousand Oaks, CA, 7 edition, 2020.

[3] V. Kann. Maximum bounded 3-dimensional matching is max snp-complete. *Information Processing Letters*, 37(1):27–35, 1991.

[4] E. S. Morgan. *Inventing the People: The Rise of Popular Sovereignty in England and America*. W. W. Norton Company, New York, 1988.

[5] N. Ravanilla, D. Haim, and A. Hicken. Brokers, social networks, reciprocity, and clientelism. *American Journal of Political Science*, 00(0):1–18, 5 2021.

[6] S. C. Stokes. Perverse accountability: A formal model of machine politics with evidence from argentina. *American Political Science Review*, 99(3):315–325, 8 2005.

[7] P. C. Vicente. Is vote buying effective? evidence from a field experiment in west africa. *Economic Journal*, 124(574):356–387, 9 2014.

Appendix A. Code for Experimental Verification of NP-Completeness of Destructive Protection Problem

```
### FIRST SET OF HELPER FUNCTIONS

def make_subsets(size, n):
    """ returns a list of indices which refer to subsets
    i.e. make_subsets(1, n) = [0, 1, 2, 3, ..., n-1] """
    return [list(tup) for tup in list(itertools.combinations([i for i in
                    range(n)], size))]


def defend_subset(subset, voters, defense_budget):
    """ returns possible (bool)
    note that subset is a list """

    subset_sum = 0
    for voter in np.array(voters)[subset]:
        subset_sum += voter.award_price

    if subset_sum > defense_budget: return False
    else: return True


def bribe_subset(subset, voters, bribe_budget, defended_subset):
    """ returns possible (bool)
    note that subset is a list """

    subset_sum = 0
    for voter in np.array(voters)[subset]:
        subset_sum += voter.bribe_price

    if subset_sum > bribe_budget: return False
    else: return True


### MAIN HELPERS AND MAIN


def can_be_bribed(defended_subset, bribe_budget, voters, candidates, is_r_approval):
    """ returns true if, given defended voters,
    any subset of bribery will work against it """
    for subset_size in range(1, n+1):
        for subset in make_subsets(subset_size, n):
            if bribe_subset(subset, voters, bribe_budget, defended_subset):
            # subset can be afforded

                for voter in np.array(voters)[defended_subset]:
                    voter.defended = True
```

```
                if bribe(candidates, voters, bribe_budget, is_r_approval):
                    return True

    return False # cannot be successfully bribed


# Main function that determines if the voters can be defended.
# This is the function which will be timed
def can_be_defended(candidates, voters, defense_budget, bribe_budget, is_r_approval):

    n = len(voters)

    for subset_size in range(1, n+1):
        for subset in make_subsets(subset_size, n):
            if defend_subset(subset, voters, defense_budget):
                if not can_be_bribed(subset, bribe_budget, voters, candidates, is_r_approval):
                    return 1 # successfully defended
            reset_voters(voters)

    return 0 # cannot be successfully defended
```

Appendix B. Code for Experimental Approximation of Destructive Protection Problem

```
# first: classes for the people involved

class Voter:
    def __init__(self):
        self.pref = random.sample(candidates, m) # in order from most to least preferred
        self.weight = random.random()
        self.defended = False
        self.bribed = False
        self.bribe_pref = None
        self.bribe_price = random.random() * 10
        self.award_price = random.random() * 10

class Briber:
    def __init__(self, budget):
        self.kind = "destructive" # based on lemma 13
        self.budget = budget

class Defender:
    def __init__(self, budget):
        self.budget = budget


# second: find winner function

def r_approval(cands, voter):
    # based off alpha in Chen et al.
    alpha = [0] * len(cands)
    if voter.bribed:
        pref_list = voter.bribe_pref
    else:
        pref_list = voter.pref

    # only candidates who are approved get 1 point; others get 0
    for approved in pref_list[:r]:
        alpha[approved] = 1
    return np.array(alpha)

def borda_count(cands, voter):
    alpha = [0] * len(cands)
    if voter.bribed:
        pref_list = voter.bribe_pref
    else:
        pref_list = voter.pref

    # candidate in rank 1 gets m points, 2 gets m-1, and so on
    for i, can in enumerate(pref_list):
        alpha[can] = m-i
```

```
    return np.array(alpha)

def find_winner(cands, voters, is_r_approval):
    cand_scores = np.zeros(len(cands))
    for v in voters:
        if is_r_approval:
            cand_scores += v.weight * r_approval(cands, v)
        else:
            cand_scores += v.weight * borda_count(cands, v)
    ordering = sorted(enumerate(cand_scores), key=lambda x: x[1], reverse=True)
    # returns candidates in order of decreasing score
    return [cands[o[0]] for o in ordering]


# third: defending problem

def defend(candidates, voters, defender_budget, is_r_approval):
    defender = Defender(defender_budget)

    if is_r_approval:
        voters_sorted = sorted(voters, key=lambda x: x.award_price / x.weight)
    else:
        voters_sorted = sorted(voters, key=lambda x: (x.award_price / x.weight) /
          (m - x.pref.index(true_winner) - 1 + 1e-20))

    for v in voters_sorted:
        # see if defending this voter would help election against destruction
        if (is_r_approval and true_winner in v.pref[:r])
          or (not is_r_approval and true_winner in v.pref[:len(v.pref)//2]):
            # see if can pay for voter
            if defender.budget - v.award_price >= 0
              or math.isclose(defender.budget, v.award_price):
                # defend voter, pay
                v.defended = True
                defender.budget -= v.award_price

    # if there's still budget, start defending other undefended voters in order
    if defender.budget >= 0:
        for v in voters_sorted:
            if not v.defended:
                # see if can pay for voter
                if defender.budget - v.award_price >= 0
                  or math.isclose(defender.budget, v.award_price):
                    # defend voter, pay
                    v.defended = True
                    defender.budget -= v.award_price

    return
```

```
# fourth: bribery problem

def valid_bribee(voter, true_profile, is_r_approval):
    first = true_profile[0]
    second = true_profile[1]
    pref = voter.pref.copy()

    if voter.defended: # cannot bribe defended voters
        return None, None

    # under $r$-approval, don't bribe voters who already
    # approve of second and disapprove of first
    if is_r_approval and (second in pref[:r] and first not in pref[:r]):
        return None, None

    pref.remove(first)
    pref.remove(second)
    pref.insert(0, second)
    pref.append(first)
    return 1, pref


def bribe(candidates, voters, briber_budget, is_r_approval):
    briber = Briber(briber_budget)

    if is_r_approval:
        voters_sorted = sorted(voters, key=lambda x: x.bribe_price / x.weight)
    else:
        voters_sorted = sorted(voters, key=lambda x: (x.bribe_price / x.weight) /
          (m - x.pref.index(true_winner) - 1 + 1e-20))

    for i, voter in enumerate(voters_sorted):
        valid, new_pref = valid_bribee(voter, true_profile, is_r_approval)
        # change their bribed profile if we have enough money to bribe
        if valid is not None and (briber.budget >= voter.bribe_price
          or math.isclose(briber.budget, voter.bribe_price)):
            voter.bribed = True
            voter.bribe_pref = new_pref
            briber.budget -= voter.bribe_price

            new_profile = find_winner(candidates, voters, is_r_approval)
            if new_profile[0] != true_winner:
                # bribe was successful, changed winner
                return True

    return False
```

```
def reset_voters(voters):
    for v in voters:
        v.defended = False
        v.bribed = False
        v.bribe_pref = None
    return


# fifth: time & plot!

is_r_approval = False

m = 5
r = 3
candidates = [i for i in range(m)]

times = []

for n in range(1, 201):
    voters = [Voter() for i in range(n)]
    true_winner_list = find_winner(candidates, voters, is_r_approval)
    true_winner = true_winner_list[0]

    total_award_price = 0
    total_bribe_price = 0
    for v in voters:
        total_award_price += v.award_price
        total_bribe_price += v.bribe_price

    start = time.time()

    done = False
    for attack_money in range(int(total_bribe_price)+1, -1, -1):
        for defense_money in range(0, int(total_award_price)+2):
            defend(candidates, voters, defense_money, is_r_approval)
            bribe_success = bribe(candidates, voters, attack_money, is_r_approval)

            if not bribe_success:
                done = True
                break

            reset_voters(voters)

        if done:
            break

    end = time.time()

    times.append(end - start)
```

```
ns = np.arange(1, len(times)+1)

plt.rcParams["figure.figsize"] = (12, 9)
plt.plot(ns, times)
plt.xlabel('N (number of voters)')
plt.ylabel('Time (in sec)')
plt.title("Approximate Destructive Protection Problem Time Complexity, for ___")
plt.show()
```

Kayla Huang
Email address: kaylahuang@college.harvard.edu

Austin Li
Email address: awli@college.harvard.edu

Nathan Sun
Email address: nsun@college.harvard.edu

Christina Xiao
Email address: christinaxiao@college.harvard.edu