Learning Voting Trees

Ariel D. Procaccia and **Aviv Zohar** and **Yoni Peleg** and **Jeffrey S. Rosenschein**
School of Engineering and Computer Science
The Hebrew University of Jerusalem
Jerusalem, Israel
{arielpro, avivz, jonip, jeff}@cs.huji.ac.il

Abstract

Binary voting trees provide a succinct representation for a large and prominent class of voting rules. In this paper, we investigate the PAC-learnability of this class of rules. We show that, while in general a learning algorithm would require an exponential number of samples, if the number of leaves is polynomial in the size of the set of alternatives then a polynomial training set suffices. We apply these results in an emerging theory: automated design of voting rules by learning.

Introduction

Voting has recently attracted attention from computer scientists (particularly in artificial intelligence) as a method for preference aggregation. A substantial body of research now exists on computational aspects of social choice (Conitzer & Sandholm 2002; Procaccia & Rosenschein 2007), as well as on many of its applications (Ephrati & Rosenschein 1997; Haynes *et al.* 1997; Ghosh *et al.* 1999).

In general, in voting settings it is common to consider a set $N = \{1, \ldots, n\}$ of *voters*, and a set $A = \{x_1, \ldots, x_m\}$ of *alternatives*. Each voter $i \in N$ is associated with a linear ordering R^i of the alternatives; that is, each voter ranks the alternatives in a way that reflects its preferences. The winner of the election is determined according to a *voting rule*—essentially a function from rankings to alternatives.

Voting Rules and Voting Trees

The theoretical properties of many voting rules have been investigated (Brams & Fishburn 2002). For example, in the simple *Plurality* rule used in most real-life elections, every voter gives one point to the alternative it ranks first, and the alternative with the most points wins the election. Other voting rules rely on the concept of *pairwise elections*: alternative a beats alternative b in the pairwise election between a and b if the majority[1] of voters prefers a to b. Ideally, we would like to select an alternative that beats every other alternative in a pairwise election, but such an alternative (called a *Condorcet winner*) does not always exist.

However, there are other prominent voting rules that rely on the concept of pairwise elections, which select an alternative in a sense "close" to the Condorcet winner. In the Copeland rule, for example, the score of an alternative is the number of alternatives it beats in a pairwise election; the alternative with the highest score wins. In the Maximin rule, the score of an alternative is its worst pairwise election (the least number of voters that prefer it to some alternative), and, predictably, the winner is the alternative that scores highest.

When discussing such voting rules, it is possible to consider a more abstract setting. A *tournament* "$\succ$" over A is a complete binary irreflexive relation over A (that is, for any two alternatives a and b, $a \succ b$ or $b \succ a$, but not both). Clearly, the aforementioned majority relation induces a tournament (a beats b in the pairwise election iff $a \succ b$). More generally, this relation can reflect a reality that goes beyond a strict voting scenario. For example, the tournament can represent a basketball league, where $a \succ b$ if team a is expected to beat team b in a game. Denote the set of all tournaments over A by $\mathcal{T} = \mathcal{T}(A)$.

So let us look at voting rules as simply functions $F : \mathcal{T} \to A$. The most prominent class of such functions is the class of *binary voting trees*. Each function in the class is represented by a binary tree, with the leaves labeled by alternatives. At each node, the alternatives at the two children compete; the winner ascends to the node (so if a and b compete and $a \succ b$, a ascends). The winner-determination procedure starts at the leaves and proceeds upwards towards the root; the alternative that survives to the root is the winner of the election.

For example, assume that the alternatives are a, b and c, and $b \succ a$, $c \succ b$ and $a \succ c$. In the tree given in Figure 1, b beats a and is subsequently beaten by c in the right subtree, while a beats c in the left subtree. a and c ultimately compete at the root, making a the winner of the election.

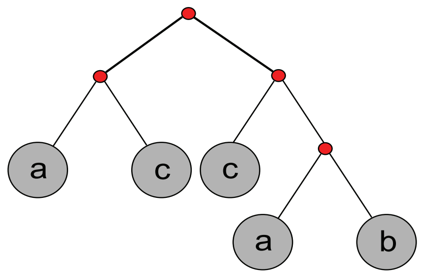

Figure 1: A binary voting tree

[1]We will assume, for simplicity, an odd number of voters.

Notice that we allow an alternative to appear in multiple leaves; further, some alternatives may not appear at all (so, for example, a singleton tree is a constant function). Not every voting rule $F : \mathcal{T} \rightarrow A$ can be implemented by a voting tree; however, some of the prominent rules, such as Copeland, can be implemented (Trick 2006).

Automated Design of Voting Rules by Learning

In this paper, we characterize the complexity of *learning* voting trees. The basic setting is as follows: there is a *designer*, who has a desired voting rule in mind. In other words, for any given tournament, the designer is able to specify who the winner of the election is. Further, we assume that the voting rule the designer has in mind can be represented by a voting tree (as we mentioned above, this is the case for many prominent voting rules). However, the designer does not know how to actually represent the rule as a tree. The goal is to learn the designer's voting rule through exposure to examples of specific tournaments and their winners. The overall approach is known as *automated design of voting rules by learning*.

There are several rationales for studying this problem. Note that the number of voting rules $F : \mathcal{T} \rightarrow A$ is $m^{2^{\binom{m}{2}}}$, so in general a representation of a voting rule requires a number of bits which is exponential in m. On the other hand, it is clear that voting trees can be concisely represented. Therefore, the designer may wish to transform an extremely inefficient representation he might have for a voting rule into a concise one—and the learning procedure produces such a representation.

Additionally, it might be the case that the designer wishes to build a voting rule that satisfies different desirable properties, such as election of a Condorcet winner if one exists. The designer could use a representation of these properties to designate the just winner in every tournament given to him as a query. By configuring the structure of voting trees, it is possible to obtain a wide range of properties; thus, the learning process could ultimately produce a voting tree that satisfies the desiderata, if one exists. For further motivating details regarding automated design of voting rules in general, see the work of Procaccia, Zohar and Rosenschein (2006).

Our learning model can be described as follows. We assume the designer has in mind a voting rule—the *target* voting rule—which can be represented by a tree. The learner draws tournaments according to a fixed distribution D over $\mathcal{T}$; these tournaments, in the context of learning, are called *examples*. When faced with each such tournament, the designer responds by giving the correct winner in the tournament, with respect to the target voting rule. The goal of the learner, given enough examples, is to come up with a voting rule that is "close" to the target. More precisely, we allow only a small probability that, when tournaments are drawn according to D, our rule disagrees with the target rule. The reader may recognize that we are talking about learning in the formal model of learning theory—the PAC model. Relevant technical details about the model are given in the Preliminaries section, below.

Our main question in this paper is: how many examples are needed in order to construct a tree that is "close" to the target voting tree? We will show that the answer is two-fold: in general, due to the expressiveness and possible complexity of binary trees, the number of examples required is exponential in m. However, if we assume that the number of leaves is polynomial in m, then the required number of examples is also polynomial in m. In addition, we investigate the computational complexity of problems associated with the learning process.

Related Work

This paper can be seen as extending the very recent work of Procaccia, Zohar and Rosenschein (2006). The authors presented and motivated the use of learning techniques to automatically design voting rules. However, that work investigated learning *scoring rules*, an important but small family of voting rules (which includes, for instance, the Plurality rule mentioned above). In contrast, the current paper studies the learnability of the much larger (and disjoint) family of voting trees, and shows that they are harder to learn than scoring rules. To the best of our knowledge, the current paper and the abovementioned paper are the only ones to apply learning in the context of voting.

Learning has been applied, however, in other economic settings. A prominent example is a paper by Lahaie and Parkes (2004), which discusses preference elicitation in combinatorial auctions, and shows that learning algorithms can be used as a basis for preference elicitation algorithms. The learning model in that work (exact learning) is different from ours. Another paper in this direction, which does use the PAC model, is that of Beigman and Vohra (2006); they applied PAC learning to compute utility functions that are rationalizations of given sequences of prices and demands.

Preliminaries

In this section we give a very short introduction to the PAC model and the generalized dimension of a function class. A more comprehensive (and slightly more formal) overview of the model, and results concerning the dimension, can be found in (Natarajan 1991).

In the PAC model, the learner is attempting to learn a function $f : Z \rightarrow Y$, which belongs to a class $\mathcal{F}$ of functions from Z to Y. The learner is given a *training set*—a set $\{z_1, z_2, \ldots, z_t\}$ of points in Z, which are sampled i.i.d. (independently and identically distributed) according to a distribution D over the sample space Z. D is unknown, but is fixed throughout the learning process. In this paper, we assume the "realizable" case, where a target function $f^*(z)$ exists, and the given training examples are in fact labeled by the target function: $\{(z_k, f^*(z_k))\}_{k=1}^t$. The *error* of a function $f \in \mathcal{F}$ is defined as

$$err(f) = \Pr_{z \sim D}[f(z) \neq f^*(z)]. \qquad (1)$$

$\epsilon > 0$ is a parameter given to the learner that defines the *accuracy* of the learning process: we would like to achieve $err(h) \leq \epsilon$. Notice that $err(f^*) = 0$. The learner is also given an *accuracy* parameter $\delta > 0$, that provides an upper bound on the probability that $err(h) > \epsilon$:

$$\Pr[err(h) > \epsilon] < \delta. \tag{2}$$

We now formalize the discussion above:

Definition 1.

1. *A* learning algorithm L *is a function from the set of all training examples to* $\mathcal{F}$ *with the following property: given* $\epsilon, \delta \in (0,1)$ *there exists an integer* $s(\epsilon, \delta)$*—the* sample complexity*—such that for any distribution* D *on* Z*, if* S *is a sample of size at least* s *where the samples are drawn i.i.d. according to* D*, then with probability at least* $1-\delta$ *it holds that* $err(L(S)) \leq \epsilon$.
2. *A function class* $\mathcal{F}$ *is* PAC-learnable *if there is a learning algorithm for* $\mathcal{F}$.

The sample complexity of a learning algorithm for $\mathcal{F}$ is closely related to a measure of the class's combinatorial richness known as the generalized dimension.

Definition 2. Let $\mathcal{F}$ be a class of functions from Z to Y. We say $\mathcal{F}$ *shatters* $S \subseteq Z$ if there exist two functions $f, g \in \mathcal{F}$ such that

1. For all $z \in S$, $f(z) \neq g(z)$.
2. For all $S_1 \subseteq S$, there exists $h \in \mathcal{F}$ such that for all $z \in S_1$, $h(z) = f(z)$, and for all $z \in S \setminus S_1$, $h(z) = g(z)$.

Definition 3. Let $\mathcal{F}$ be a class of functions from a set Z to a set Y. The *generalized dimension* of $\mathcal{F}$, denoted by $D_G(\mathcal{F})$, is the greatest integer d such that there exists a set of cardinality d that is shattered by $\mathcal{F}$.

Lemma 1. *(Natarajan 1991, Lemma 5.1) Let* Z *and* Y *be two finite sets and let* $\mathcal{F}$ *be a set of total functions from* Z *to* Y*. If* $d = D_G(F)$*, then* $2^d \leq |\mathcal{F}|$.

A function's generalized dimension provides both upper and lower bounds on the sample complexity of algorithms.

Theorem 1. *(Natarajan 1991, Theorem 5.1) Let* $\mathcal{F}$ *be a class of functions from* Z *to* Y *of generalized dimension* d*. Let* L *be an algorithm such that, when given a set of* t *labeled examples* $\{(z_k, f^*(z_k))\}_k$ *of some* $f^* \in \mathcal{F}$*, sampled i.i.d. according to some fixed but unknown distribution over the instance space* X*, produces an output* $f \in \mathcal{F}$ *that is consistent with the training set. Then* L *is an* (ϵ, δ)*-learning algorithm for* $\mathcal{F}$ *provided that the sample size obeys:*

$$s \geq \frac{1}{\epsilon}\left((\sigma_1 + \sigma_2 + 3)d \ln 2 + \ln\left(\frac{1}{\delta}\right)\right) \tag{3}$$

where σ_1 *and* σ_2 *are the sizes of the representation of elements in* Z *and* Y*, respectively.*

Theorem 2. *(Natarajan 1991, Theorem 5.2) Let* $\mathcal{F}$ *be a function class of generalized dimension* $d \geq 8$*. Then any* (ϵ, δ)*-learning algorithm for* $\mathcal{F}$*, where* $\epsilon \leq 1/8$ *and* $\delta < 1/4$*, must use sample size* $s \geq \frac{d}{16\epsilon}$.

Learnability of Large Voting Trees

Recall that we are dealing with a set of *alternatives* $A = \{x_1, \ldots, x_m\}$; sometimes, we will also denote alternatives by $a, b, c \in A$. A *tournament* is a complete binary irreflexive relation $\succ$ over A; we denote the set of all possible tournaments by $\mathcal{T} = \mathcal{T}(A)$. A *voting rule* is a function $F : \mathcal{T} \to A$.

A *binary voting tree* is a binary tree with leaves labeled by alternatives. To determine the winner of the election with respect to a tournament $\succ$, one must iteratively select two siblings, label their parent by the winner according to $\succ$, and remove the siblings from the tree. This process is repeated until the root is labeled, and its label is the winner of the election.

Let us denote the class of voting trees over m alternatives by $\mathcal{F}_m$; we would like to know what the sample complexity of learning functions in $\mathcal{F}_m$ is. To elaborate a bit, we think of voting trees as functions from $\mathcal{T}$ to A, so the sample space is $\mathcal{T}$. In this section, we will show that in general, the answer is that the complexity is exponential in m. We will prove this by relying on Theorem 2; the theorem implies that in order to prove such a claim, it is sufficient to demonstrate that the generalized dimension of $\mathcal{F}_m$ is at least exponential in m. This is the task we presently turn to.

Theorem 3. $D_G(\mathcal{F}_m)$ *is exponential in* m.

Proof. Without loss of generality, we let $m = 2k + 2$. We will associate every distinct binary vector $v = \langle v_1, \ldots, v_k \rangle \in \{0,1\}^k$ with a distinct example in our set of tournaments $S \subseteq \mathcal{T}$. To prove the theorem, we will show that $\mathcal{F}_m$ shatters this set S of size 2^k.

Let the set of alternatives be:

$$A = \{a, b, x_1^0, x_1^1, x_2^0, x_2^1, \ldots, x_k^0, x_k^1\}.$$

For every vector $v \in \{0,1\}^k$, define a tournament $\succ_v$ as follows: for $i = 1, \ldots, k$, if $v_i = 0$, we let $x_i^0 \succ_v b \succ_v x_i^1$; otherwise, if $v_i = 1$, then $x_i^1 \succ_v b \succ_v x_i^0$. In addition, for all tournaments $\succ_v$, and all $i = 1, \ldots, k$, $j = 0, 1$, a beats x_i^j, but a loses to b. We denote by S the set of these 2^k tournaments.[2] Let f be the constant function b, i.e., a voting tree which consists of only the node b; let g be the constant function a. We must prove that for every $S_1 \subseteq S$, there is a voting tree such that b wins for every tournament in S_1 (in other words, the tree agrees with f), and a wins for every tournament in $S \setminus S_1$ (the tree agrees with g). Consider the tree in Figure 2, which we refer to as the i'th *2-gadget*.

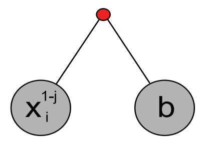

Figure 2: 2-gadget

With respect to this tree, b wins a tournament $\succ_v \in S$ iff $v_i = j$. Indeed, if $v_i = j$, the $x_i^j \succ_v b \succ_v x_i^{1-j}$, and in particular b beats x_i^{1-j}; if $v_i \neq j$, then $x_i^{1-j} \succ_v b \succ_v x_i^j$, so b loses to x_i^{1-j}.

[2]The relations described above are not complete, but the way they are completed is of no consequence.

Let $v \in \{0,1\}^k$. We will now use the 2-gadget to build a tree where b wins only the tournament $\succ_v \in S$, and loses every other tournament in S. Consider a balanced tree such that the deepest nodes in the tree are in fact 2-gadgets (as in Figure 3). As before, b wins in the i'th 2-gadget iff $v_i = j$. We will refer to this tree as a v-gadget.

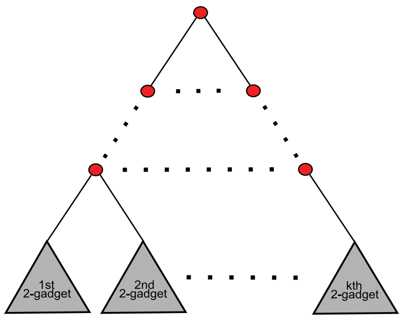

Figure 3: v-gadget

Now, notice that if b wins in each of the 2-gadgets (and this is the case in the tournament $\succ_v$), then b is the winner of the entire election. On the other hand, let $v' \neq v$, i.e., there exists $i \in \{1, \dots, k\}$ such that w.l.o.g. $0 = v'_i \neq v_i = 1$. Then it holds that $x_i^0 \succ_{v'} b \succ_{v'} x_i^1$; this implies that x_i^0 wins in the i'th 2-gadget. x_i^0 proceeds to win the entire election, unless it is beaten in some stage by some other alternative x_l^j—but this must be also an alternative that beats b, as it survived the l'th 2-gadget. In any case, b cannot win the election.

Consider the small extension, in Figure 4, of the v-gadget, which (for lack of a better name) we call the v-gadget*.

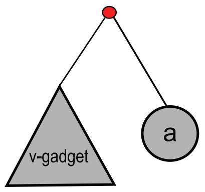

Figure 4: v-gadget*

Recall that, in every tournament in S, a beats any alternative x_j^i but loses to b. Therefore, by our discussion regarding the v-gadget, b wins the election described by the v-gadget* only in the tournament $\succ^v$; for any other tournament in S, alternative a wins the election.

We now present a tree and prove that it is as required, i.e., in any tournament in S_1, b is the winner, and in any tournament in $S \setminus S_1$, a prevails. Let us enumerate the tournaments in S_1:

$$S_1 = \{\succ_{v_1}, \dots, \succ_{v_r}\}.$$

We construct a balanced tree, as in Figure 5, where the bottom levels consist of the v_l-gadgets*, for $l = 1, \dots, r$.

Let $\succ_{v_l} \in S_1$. What is the result of this tournament in the election described by this tree? First, note that b prevails

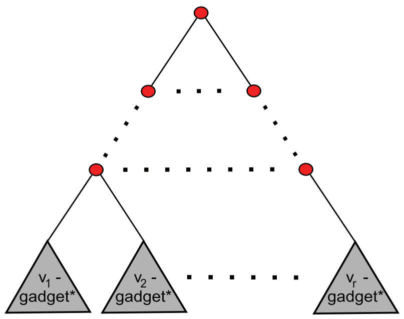

Figure 5: The constructed tree

in the v_l-gadget*. The only alternatives that can reach any level above the gadgets are a and b, and b always beats a. Therefore, b proceeds to win the election. Conversely, let $\succ_v \in S \setminus S_1$. Then a survives in *every* v_l-gadget*, for $l = 1, \dots, r$. a surely proceeds to win the entire election.

We have shown that $\mathcal{F}_m$ shatters S, thus completing the proof. □

Remark 1. Even if we restrict our attention to the class of balanced voting trees (corresponding to a playoff schedule), the dimension of the class is still exponential in m. Indeed, any unbalanced tree can be transformed to an identical (as a voting rule) balanced tree. If the tree's height is h, this can be done by replacing every leaf at depth $d < h$, labeled by an alternative a, by a balanced subtree of height $d - h$ in which all the leaves are labeled by a. This implies that the class of balanced trees can shatter any sample which is shattered by $\mathcal{F}_m$.

Remark 2. The proof we have just completed, along with Lemma 1, imply that the number of different voting functions that can be represented by trees is double exponential in m, which shows the high expressiveness of voting trees.

Learnability of Small Voting Trees

In the previous section, we have seen that in general, a large number of examples is needed in order to learn voting trees in the PAC model. This result relied on the number of leaves in the trees being exponential in the number of alternatives. However, in many realistic settings one can expect the voting tree to be compactly represented, and in particular one can usually expect the number of leaves to be at most polynomial in m. Let us denote by $\mathcal{F}_m^n$ the class of voting trees over m alternatives, with at most n leaves. Our goal in this section is to prove the following theorem.

Theorem 4. $D_G(\mathcal{F}_m^n) \leq n(\log m + \log n)$.

This theorem implies, in particular, that if the number of leaves n is polynomial in m, then the dimension of $\mathcal{F}_m^n$ is polynomial in m. In turn, this implies by Lemma 1 that the sample complexity of $\mathcal{F}_m^n$ is only polynomial in m. In other words, given a training set of size polynomial in m, $1/\epsilon$ and $1/\delta$, any algorithm that returns some tree consistent with the training set is an (ϵ, δ)-learning algorithm for $\mathcal{F}_m^n$.

To prove the theorem, we require the following lemma.

Lemma 2. $|\mathcal{F}_m^n| \leq m^n \cdot n!$

Proof. The number of voting trees with exactly n leaves is at most the number of binary tree structures multiplied by the number of possible assignments of alternatives to leaves. The number of assignments is clearly bounded by m^n. In order to count the number of distinct structures a tree with n leaves can have, we define a *split* operation on a leaf v of a binary tree T: the tree T' which is obtained by the operation is exactly the same as T, except that v is now an internal node, and is the parent of two leaves (see Figure 6).

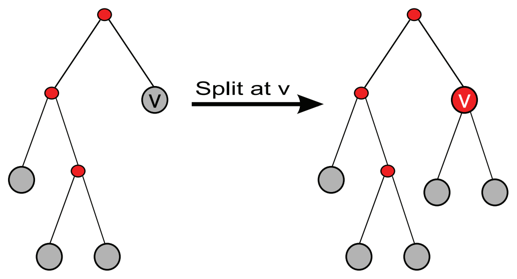

Figure 6: A binary tree split at a node v.

Lemma 3. *Any binary tree with n leaves can be obtained by starting with a singleton tree, and performing $n-1$ splits (proof omitted due to lack of space).*

Lemma 3 allows us to bound the number of distinct structures a binary tree with exactly n leaves might have. Starting with the singleton tree, we have one option for a split; for a tree with two leaves, we have two options; and so on. Thus, the total number of ways to construct a binary tree with n leaves is at most:

$$1 \cdot 2 \cdots n-1 = (n-1)!$$

So, the total number of voting trees with exactly n leaves is bounded by $m^n \cdot (n-1)!$, and the number of voting trees with *at most* n leaves is at most:

$$n \cdot (m^n \cdot (n-1)!) = m^n \cdot n!$$

□

We are now ready to prove Theorem 4.

Proof of Theorem 4. By Lemma 3, we have that $|\mathcal{F}_m^n| \leq m^n \cdot n!$. Therefore, by Lemma 1:

$$D_G(\mathcal{F}_m^n) \leq \log(m^n \cdot n!) \leq n \log m + n \log n.$$

□

Computational Complexity

In the previous section, we restricted our attention to voting trees where the number of leaves is polynomial in m. We have demonstrated that the dimension of this class is polynomial in m, which implies that the sample complexity of the class is polynomial in m. Therefore, any algorithm that is consistent with a training set of polynomial size is a suitable learning algorithm (Theorem 1).

It seems that the significant bottleneck, especially in the setting of automated voting rule design (finding a compact representation for a voting rule that the designer has in mind), is the number of queries posed to the designer, so in this regard we are satisfied that realistic voting trees are learnable. Nonetheless, in some contexts we may also be interested in computational complexity: given a training set of polynomial size, how computationally hard is it to find a voting tree which is consistent with the training set?

In this section we explore that question. We will assume hereinafter that the structure of the voting tree is known *a priori*. This is an assumption that we did not make above, but observe that, at least for balanced trees, Theorems 3 and 4 hold regardless. We shall try to determine how hard it is to find an assignment to the leaves which is consistent with the training set. We will refer to the computational problem as TREE-SAT (pun intended).

Definition 4. In the TREE-SAT problem, we are given a binary tree, where some of the leaves are already labeled by alternatives, and a training set that consists of pairs $(\succ_j, x_{i_j})$, where $\succ_j \in \mathcal{T}$ and $x_{i_j} \in A$. We are asked whether there exists an assignment of alternatives to the rest of the leaves which is consistent with the training set, i.e., for all j, the winner in $\succ_j$ with respect to the tree is x_{i_j}.

Notice that in our formulation of the problem, some of the leaves are already labeled. However, it is reasonable to expect any efficient algorithm that finds a consistent tree, given that one exists, to be able to solve the TREE-SAT problem. Hence, an $\mathcal{NP}$-hardness result implies that such an algorithm is not likely to exist. This is actually the case (proof omitted due to lack of space):

Theorem 5. TREE-SAT *is $\mathcal{NP}$-complete.*

Empirical Results

Despite Theorem 5, it seems that in practice, solving the TREE-SAT problem is sometimes possible; here we shall empirically demonstrate this.

Our simulations were carried out as follows. Given a fixed tree structure, we randomly assigned alternatives (out of a pool of 32 alternatives) to the leaves of the tree. We then used this tree to determine the winners in 20 random tournaments over our 32 alternatives. Next, we measured the time it took to find some assignment to the leaves of the tree (not necessarily the original one) which is consistent with the training set of 20 tournaments. We repeated this procedure 10 times for each number of leaves in $\{4, 8, 16, 32, 64\}$, and took the average of all ten runs.

The problem of finding a consistent tree can easily be represented as a constraint satisfaction problem, or in particular as a SAT problem. Indeed, for every node, one simply has to add one constraint per tournament which involves the node and its two children. To find a satisfying assignment, we used the SAT solver zChaff. The simulations were carried out on a PC with a Pentium D (dual core) CPU, running Linux, with 2GB of RAM and a 2.8GHz clock speed.

We experimented with two different tree structures. The first is seemingly the simplest—a binary tree which is as close to a chain as possible, i.e., every node is either a leaf, or the parent of a leaf; we refer to these trees as "almost chains". The second is intuitively the most complicated: a balanced tree. Notice that, given that the number of leaves is n, the number of nodes in both cases is $2n - 1$. The simulation results are shown in Figure 7.

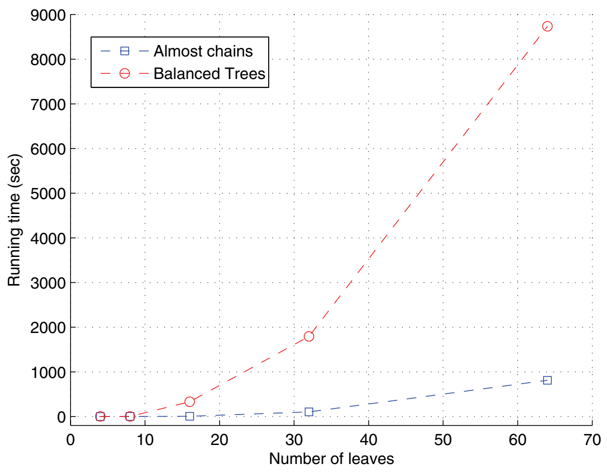

Figure 7: Time to find a satisfying assignment

In the case of balanced trees, it is indeed hard to find a consistent tree. Adding more sample tournaments would add even more constraints and make the task harder. However, in most elections the number of candidates is usually not above several dozen, and the problem may still be solvable. Furthermore, the problem is far easier with respect to trees that are almost chains (although the reduction in Theorem 5 builds trees that are "almost $\{$almost chains$\}$"). Therefore, we surmise that for many tree structures, it may be practically possible (in terms of the computational effort) to find a consistent assignment, even when the input is relatively large, while for others the problem is quite computationally hard even in practice.

Conclusions

We have discussed the learnability of voting trees in the PAC model. Our main results imply that when the number of leaves in the tree is unbounded, the number of samples required in order to achieve arbitrary accuracy and confidence is exponential in the number of alternatives. However, if the number of leaves is polynomial, a training set of polynomial size suffices. In this case, any tree which is consistent with the training set will do; our empirical results show that finding these trees may prove to be a feasible computational task, depending on the structure of the tree that is being learned.

Our results have several important implications. Most significantly, with respect to automated design of voting rules, we have essentially shown that when the designer has in mind a voting rule that can be concisely represented using a "small" voting tree, an approximation of this concise representation can be achieved. So, using our approach, the designer can translate a cumbersome representation of a voting rule into a concise one, or find a voting tree that satisfies a given set of desiderata. A limitation of our approach in this context is that there are voting rules $F : \mathcal{T} \to A$ that cannot be implemented by voting trees (Trick 2006); the designer may have such a voting rule in mind. Nevertheless, the class of voting rules that can be represented by trees is large and prominent, including, among others, the Copeland rule.

Acknowledgments

The authors would like to thank Noam Nisan for suggesting the topic of this paper. This work was partially supported by Israel Science Foundation grant #898/05.

References

Beigman, E., and Vohra, R. 2006. Learning from revealed preference. In *Proceedings of the 7th ACM Conference on Electronic Commerce*, 36–42.

Brams, S. J., and Fishburn, P. C. 2002. Voting procedures. In Arrow, K. J.; Sen, A. K.; and Suzumura, K., eds., *Handbook of Social Choice and Welfare*. North-Holland. chapter 4.

Conitzer, V., and Sandholm, T. 2002. Complexity of manipulating elections with few candidates. In *The National Conference on Artificial Intelligence*, 314–319.

Ephrati, E., and Rosenschein, J. S. 1997. A heuristic technique for multiagent planning. *Annals of Mathematics and Artificial Intelligence* 20:13–67.

Ghosh, S.; Mundhe, M.; Hernandez, K.; and Sen, S. 1999. Voting for movies: the anatomy of a recommender system. In *Proceedings of the Third Annual Conference on Autonomous Agents*, 434–435.

Haynes, T.; Sen, S.; Arora, N.; and Nadella, R. 1997. An automated meeting scheduling system that utilizes user preferences. In *Proceedings of the First International Conference on Autonomous Agents*, 308–315.

Lahaie, S., and Parkes, D. C. 2004. Applying learning algorithms to preference elicitation. In *Proceedings of the Fifth ACM Conference on Electronic Commerce*, 180–188.

Natarajan, B. K. 1991. *Machine Learning: A Theoretical Approach*. Morgan Kaufmann.

Procaccia, A. D., and Rosenschein, J. S. 2007. Junta distributions and the average-case complexity of manipulating elections. *Journal of Artificial Intelligence Research* 28:157–181.

Procaccia, A. D.; Zohar, A.; and Rosenschein, J. S. 2006. Automated design of voting rules by learning from examples. In *Proceedings of the First International Workshop on Computational Social Choice*, 436–449.

Trick, M. 2006. Small binary voting trees. In *Proceedings of the First International Workshop on Computational Social Choice*, 500–511.