Preference Aggregation over Combinatorial Domains Using Heuristic Search

Michael Huelsman[1] and **Mirosław Truszczyński**[2]

Abstract. The problem of finding a preference model which represents the preferences of a group of agents is known as preference aggregation. Work on preference aggregation has primarily focused on domains where agents express preferences over sets of alternatives which are small in size. This work concerns the case when the space of alternatives is large, specifically when it forms a combinatorial domain. We propose and study preference aggregation over combinatorial domains by learning logical preference models from a set of examples using heuristic search techniques. Our method is agnostic as to how agent's preferences are originally represented, although there is some effect on performance. Overall, we show that learning joint preferences models for preference aggregation via heuristic search is a viable strategy when combinatorial domains are involved.

1 INTRODUCTION

A group of friends want to buy a pizza that will reasonably satisfy each member of the group. Assuming that ten toppings are available and any selection from that set can be purchased, the group wants to find the best pizza for the group. One method of deciding which pizza to get is to have each friend vote on pizzas, by providing a preorder over the possible pizzas, and then use some predetermined method of voting to find the best pizza according to the votes. This approach to preference aggregation has been studied in many contexts and there is a great deal of literature relating to voting theory [11]. Although voting might be the first approach to come to mind, it is not necessarily the right one.

Consider the number of possible pizzas. With ten possible toppings, of which each may be selected or not, there are 2^{10} or 1024 possible types of pizzas to order. Most voting schemes consist of having each voter list their preferences by enumerating alternatives (candidates) from most preferred to least preferred. This works well in scenarios where there are few candidates, such as elections; in cases where we have a large pool of alternatives, listing all the alternatives in order is inefficient or impractical. In such cases it is common to specify preference orders (votes) concisely and implicitly as expressions from some preference representation language [5].

Our pizza selection problem is of that type. In that problem, pizzas are described by means of attributes (here, toppings) that may assume one of a fixed set of values (here, each attribute may have value 1, when the corresponding topping is selected, or 0, otherwise). Such domains (spaces of alternatives) are labelled *combinatorial*. Their size is exponential in the number of attributes. Thus, enumerating preference orders over such domains becomes impractical even in cases when the number of attributes is as small as ten (as in our example). However, using concise implicit representations of preference orders over combinatorial domains has its own problems. Using these implicit representations with standard voting rules makes computing the winners a non-trivial task. In some cases efficient algorithms are available but more often than not the task is NP-hard [8, 10].

Suppose that the pizza chosen in the problem above is for a weekly meeting. While it may be possible to vote each week on the particular pizza to order, the process would be simplified if all the friends could vote once and then never again. But what if, one week, the pizza shop is out of mushrooms. In such cases, individual preference orders would have to be modified to exclude pizzas with mushrooms and the winner computation task (which, as we noted is often computationally hard) would have to be repeated. A solution to the problem is to aggregate individual preferences representations of all members of the group into a representation of a group preference order. Now, each time the group orders a pizza, the group preference order is used for decision making and a top alternative that satisfies all constraints in place that day is ordered.

This type of problem is known as rank aggregation [11]. Formally, rank aggregation consists of a set of agents (voters) $\boldsymbol{A} = \{a_1, a_2, \ldots, a_n\}$, and a set of candidates (alternatives) $\boldsymbol{C} = \{c_1, c_2, \ldots, c_m\}$, where each agent a_i has a preference relation $\succ_i$ over $\boldsymbol{C}$, and the objective of the problem is to find the preference ordering $\succ_{\boldsymbol{A}}$ which best satisfies the *group* $\boldsymbol{A}$ according to some social welfare function. Our task is then to come up with social welfare rules capable of aggregating concise preference representations into a concise group preference representation. The problem is challenging and, to the best of our knowledge, few satisfactory solutions based on voting rules are known (issue by issue aggregation may be one [8]). One of the core reasons for this is the sheer number of possible preference orders, whether compactly or non-compactly represented.

In this paper, we propose to overcome this problem by basing the aggregation not on preference orders (be it explicitly or implicitly defined) but on partial information about the orders given by sets of examples (comparisons consistent with the underlying order). In this way, our aggregation approach works even if all we know about individual agents preference orders are those examples. Specifically, given sets of examples collected from all agents in the group, we propose to learn a preference model from a particular preference representation language. This learning problem if formally defined as follows: given a family of example sets $\boldsymbol{\varepsilon} = \{\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_l\}$ over a combinatorial domain Ω, find a preference model from a given preference representation language $\mathcal{L}$ that best satisfies $\boldsymbol{\varepsilon}$ by optimizing a selected fairness condition (social welfare function).

For this work we adopt the maximin social welfare function, also known as the *egalitarian welfare function*, $s_w(O) =$

[1] University of Kentucky, USA, email: michael.huelsman@uky.edu
[2] University of Kentucky, USA, email: mirek@cs.engr.uky.edu

$\min_{\varepsilon \in \boldsymbol{\varepsilon}} s(O, \varepsilon)$, where $s(O, \varepsilon)$ is the proportion of examples in ε satisfied by a given order O. We use heuristic search to learn an aggregated preference model that aims to aggregate using the egalitarian social welfare function. Specifically, we propose and study heuristic algorithms that build a *ranking preference formula*, a certain compact preference representation model, based on examples extracted from agents preference orders. Because of the input they use, our algorithms do not depend on any specific preference representation language that the agents may be using to represent their preferences. In particular, this allows us to apply and study our algorithms in the settings when agents are *diverse*, that is, use different preference representation languages to represent their preference orders.

This paper is organized as follows the next section contains relevant background information including a theoretical evaluation of the problem at hand. The third section discusses the computational complexity of the problem we study. The fourth section describes our experimental methodology. The fifth section contains our results and relevant discussion of our findings. The sixth section contains a discussion of related works and how our contributions differ from other similar works. The final section provides some concluding remarks and proposed future work.

2 BACKGROUND

As stated in the introduction, our focus in this work is the problem of *rank aggregation*. In the most general terms the problem can be phrased as follows.

Problem 1. *Given a set of agents (voters) $\boldsymbol{A} = \{a_1, a_2, \cdots, a_k\}$ with preferences orders $\{\succ_1, \succ_2, \cdots, \succ_k\}$ over a set of candidates $\boldsymbol{C} = \{c_1, c_2, \cdots, c_m\}$, find a preference order $\succ_{\boldsymbol{A}}$ to serve as the consensus preference order of the group.*

In this work we study this problem under the assumptions that agent's preference orders $\succ_i$ are total preorders over the alternatives and that those alternatives are members of a *combinatorial domain*. A combinatorial domain $\mathcal{C}(\mathcal{V}, \mathcal{D})$ is formally defined by a set of *attributes*, $\mathcal{V} = \{v_1, v_2, \cdots, v_n\}$, and the set of the domains of those attributes $\mathcal{D} = \{D_1, D_2, \cdots, D_n\}$ (the finite sets of values the attributes can take). We use the shorthand $D(v_i)$ to denote the domain of an attribute v_i. A specific *alternative* α of a combinatorial domain is represented by a n-tuple of values – for each attribute one value from the domain of that attribute.

$$\alpha = (\alpha_1, \alpha_2, \cdots, \alpha_n) \in \mathcal{C}(\mathcal{V}, \mathcal{D}) \equiv \mathcal{D}(v_1) \times \mathcal{D}(v_2) \times \cdots \times \mathcal{D}(v_n).$$

Combinatorial domains are exponential in size with respect to the number of attributes. This means it is infeasible to represent preferences over a combinatorial domain by listing the alternatives from most to least preferred. Thus, as is common, we assume that preference orders of agents are represented compactly in terms of expressions from some preference representation language [5]. Moreover, when constructing algorithms to aggregate preference orders of agents into a consensus ordering, we assume that we only have partial information about these orders, that is, not the models themselves, but sets of examples of correctly ordered pairs of alternatives. Formally, an example is a triple (α, β, R) where α and β are alternatives in a combinatorial domain Ω and R is the relation between them. We limit R to either $\succ$ or $\approx$ for our example representation. We can now restate the problem of aggregation for the setting we consider.

Problem 2. *Given a set of agents $\boldsymbol{A} = \{a_1, \ldots, a_k\}$ with preferences orders $\{\succ_1, \succ_2, \cdots, \succ_k\}$ over a combinatorial domain $\mathcal{C}(\mathcal{V}, \mathcal{D})$, and a family of sets of examples $\boldsymbol{\varepsilon} = \{\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_l\}$, with each $\varepsilon_i \in \boldsymbol{\varepsilon}$ representing preferences agent $a_i \in A$ has on alternatives from $\mathcal{C}(\mathcal{V}, \mathcal{D})$, find a preference model $\succ_{\boldsymbol{A}}$ in a chosen preference representation language such that $O = \succ_{\boldsymbol{A}}$ maximizes $s_w(O) = \min_{\varepsilon \in \boldsymbol{\varepsilon}} s(O, \varepsilon)$.*

In the paper, we consider three preference representation languages. The first of them uses sets of examples to model preference orders. Specifically, given a set ε of examples, the corresponding order $\succ_\varepsilon$ is the closure of ε under reflexivity, transitivity (applied to both types of examples), and symmetry (applied to examples where R is $\approx$). If the closure has a "strict cycle (a sequence of alternatives, with each pair of consecutive ones occurring in an example in ε, and with at least one of these examples being a strict one), the resulting relation is not an order and is considered *inconsistent*. Otherwise, ε is consistent and defines a preference ordering relation, in general a preorder (not necessarily total) on the set of alternatives.

We use this representation only as a technical device in proving complexity results. In all other parts of the paper, we use two specific preference representation languages appropriate for representing total orders and preorders over combinatorial domains. Specifically, we concentrate on two compact preference representation languages: lexicographic preference models and ranking preference formulas.

A *lexicographic preference model* (LPM) [3] $\pi = (r, \succ)$ over the domain $\mathcal{C}(\mathcal{V}, \mathcal{D})$ consists of a ranking r which maps $\mathcal{V} \to [1 \ldots n]$ such that r is a one-to-one function, and $\succ = \{\succ_{v_1}, \succ_{v_2}, \ldots, \succ_{v_n}\}$ is a collection of total orders, with one order for each attribute's domain. Given two alternatives α and β, α is preferred to, or *dominates*, β, $\alpha \succ_\pi \beta$, if there exists some attribute a such that $\alpha[a] \succ_a \beta[a]$ and for all attributes b such that $r(b) < r(a)$, $\alpha[b] = \beta[b]$. In other words, one alternative is preferred to another if it has a more preferable value for the most important attribute where the two alternatives differ. It is easy to show that preference orders defined by LPMs are total orders.

A *ranking preference formula* (RPF), based on the answer set optimization preference representation [2], consists of an ordered finite tuple of boolean formulas $\boldsymbol{\varphi} = (\varphi_1, \varphi_2, \cdots, \varphi_k)$. These formulas are built from an alphabet of propositional variables (or atoms) $x_{v,d}$, where $v \in \mathcal{V}$ and $d \in \mathcal{D}(v)$. Alternatives are represented by truth assignments to these atoms. Specifically, if an alternative α has value d on an attribute v, the corresponding truth assignment assigns true to the atom $x_{v,d}$ and false to all atoms $x_{v,d'}$, where $d' \in \mathcal{D}(v) \setminus \{d\}$. We will denote this truth assignment as I_α. Propositional formulas over this alphabet represent properties of alternatives. An alternative α has a property ϕ, where ϕ is a propositional formula, if I_α satisfies ϕ according to the standard definition of satisfiability.

A tuple of formulas $\Phi = (\phi_1, \ldots, \phi_k)$ is called a *ranking preference formula* (RPF). The *satisfaction degree* of an alternative α wrt to an RPF Φ, denoted $s_\Phi(\alpha)$, is the smallest i such that I_α satisfies ϕ_i, or $k + 1$, if no such i exists. This notion induces a preference order on alternatives: an alternative α is at least as preferred as an alternative β on Φ, denoted $\alpha \succeq_\Phi \beta$, if $s_\Phi(\alpha) \leq s_\Phi(\beta)$. As with LPMs it is easy to show that the relation $\succeq_\Phi$ is a total preorder.

As mentioned above, in the context of combinatorial domains, one of the primary problems with the definition of rank aggregation is that the number of candidates is exponentially large. This means that agent preferences need to be compactly represented to make the problem practical. In that setting though, the complexity of the problem to decide the existence of "good aggregated orders (or of the search problem to compute a "good aggregated order) can vary. In some cases there exist polynomial time algorithms to apply voting rules to compact representations [10] while in other cases these problems are NP-complete (or, NP-Hard) [8], even when voting is done attribute by attribute.

Our work can be viewed as an attempt to address the problem by constructing aggregated preference representation from examples extracted from individual agents models. This preference representation can come from any compact preference representation language. In this work, we concentrate on the language of ranked preference formulas.

3 COMPLEXITY

In this section, we study the complexity of the problem of learning preference models from examples. Given the assumptions we adopted above, the decision version of Problem 2 that we consider can be stated as follows.

Problem 3 (Learn-RPF or LRPF, for short)**.** *LPRF consists of a pair $(k, \boldsymbol{\varepsilon})$ where k is a non-negative integer and $\boldsymbol{\varepsilon} = \{\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_l\}$ is family of example sets, one for each agent. An instance $(k, \boldsymbol{\varepsilon})$ of LRPF is valid if there exists an RPF $\boldsymbol{\Phi}$ such that for each $\varepsilon_i \in \boldsymbol{\varepsilon}$ at least k examples from ε_i are satisfied by $\boldsymbol{\Phi}$. The problem LRPF consists of deciding whether a given LRPF instance $(k, \boldsymbol{\varepsilon})$ is valid.*

We will show that this problem is NP-complete. To this end we first consider a closely related problem when the goal is to learn an example-based representation of a preference order. We formally define this problem in Problem 4 and show that it is NP-complete in Proposition 1.

Problem 4 (Consistent-Example-Set or CES, for short)**.** *Given an LRPF instance $(k, \boldsymbol{\varepsilon})$, decide if there exists a consistent set of examples ε' such that for each example set $\varepsilon \in \boldsymbol{\varepsilon}$, $|\varepsilon \cap \varepsilon'| \geq k$.*

Proposition 1. *The problem CES is NP-complete.*

Proof. (Membership) Given an LRPF instance, $(k, \boldsymbol{\varepsilon})$, we guess a set of examples $\varepsilon' \subseteq \bigcup_{\varepsilon \in \boldsymbol{\varepsilon}} \varepsilon$. We then check that ε' is consistent and for every $\varepsilon \in \boldsymbol{\varepsilon}$ that $|\varepsilon \cap \varepsilon'| \geq k$. To check consistency of ϵ, we first construct the graph $G_{\approx} = (V, E_{\approx})$, where V is the set of all alternatives in ε and $(\alpha, \beta) \in E_{\approx}$ if the example $(\alpha, \beta, \approx) \in \varepsilon'$. We then find all connected components of that graph, say $V_1, \ldots, V_m$. Next, we construct the graph $G_{\succ} = (V', E_{\succ})$, where $V' = \{V_1, \ldots, V_m\}$ and V_i and V_j are connected with a directed edge precisely when for some alternatives $\alpha \in V_i$ and $\beta \in V_j$, $(\alpha, \beta, \succ)$ is an example. This graph may contain loops. If this graph contains a cycle, ε is inconsistent; otherwise, it is consistent. Thus, checking consistency can be done in polynomial time, by testing acyclicity of the graph $G_{\succ}$. Checking the cardinality condition is clearly a polynomial-time task.

(Hardness) We prove hardness by a reduction from SAT. Let $\Phi \in$ SAT be a CNF formula and let $C_1, \ldots, C_n$ be its clauses. We define an LRPF instance $f(\Phi) = (k, \boldsymbol{\varepsilon})$ by setting $k = 1$ and $\boldsymbol{\varepsilon} = (\varepsilon_1, \ldots, \varepsilon_n)$, where each ε_i if constructed based on C_i. Namely for each literal $l \in C_i$ we include in ε_i the example $l \succ l'$ where l' denotes the dual literal to l. It is clear that $f(\Phi)$ can be constructed in time that is linear in the size of Φ.

⇒ If $\Phi \in$ SAT then $f(\Phi) \in$ CES.
If $\Phi \in$ SAT there exists an assignment to the literals in Φ such that at least one literal in each clause is true. For each true literal, x_i, in the satisfying assignment of Φ we select each equivalent example in $f(\Phi)$ such that $x_i \succ x'_i$. We then have a subset of examples ε' where at least one example is satisfied for each ε_i because one literal is satisfied in all clauses. We know that ε' is consistent because x_i and x'_i cannot both be true in the satisfying assignment. Furthermore, no subset of examples can produce any additional comparisons using the transitive property of $\succ$ given that each example is limited to a literal and its dual.

⇐ If $f(\Phi) \in$ CES then $\Phi \in$ SAT
If $f(\Phi) \in$ CES then there exists a set of examples $x_i \succ x'_i$ which are consistent and each agent in $f(\Phi)$ has at least one of their examples as a member in that set. If we build a truth assignment by making each x_i true in Φ if there is an example $x_i \succ x'_i$ in the set of consistent examples which makes $f(\Phi)$ a member of CES, and false to all other literals. Since each agent in $f(\Phi)$ represents a clause in Φ and each example represents a given literal in a clause this truth assignment will make at least one literal true for each clause. Since Φ is a CNF formula we know that this condition is sufficient to satisfy Φ. We also know, due to the construction of the examples, that x_i and its dual literal cannot both be set to true, otherwise the set of examples would have a contradiction, meaning it does not make $f(\Phi)$ a member of CES.

If $\Phi \notin$ SAT there is no satisfying truth assignment for Φ. This means for any truth assignment there must be a contradiction where x_i and x'_i both need to be true for Φ to be true. This translates into the examples $x_i \succ x'_i$ and $x'_i \succ x_i$ both being members of ε' for $f(\Phi)$, which makes ε' inconsistent, thus if there is no satisfying truth assignment for Φ then there is no consistent ε' which satisfies 1 example for each agent in $f(\Phi)$. □

The problems of CES and LRPF are related since it is possible to convert any consistent example set into an RPF which replicates those examples. We formally state this in Proposition 2.

Proposition 2. *Any consistent example set ε over a combinatorial domain $\mathcal{C}(\mathcal{V}, \mathcal{D})$ can be transformed into a* ranking preference formula *in which each example in ε holds in polynomial time in the size of ε.*

Proof. Construct the graph $G_{\succ}$ following the algorithm described in the proof of Proposition 1. Find a topological sort of $G_{\succ}$ which is a sorted list of equivalence classes of alternatives $\{\boldsymbol{\alpha}_1, \boldsymbol{\alpha}_2, \ldots, \boldsymbol{\alpha}_k\}$ such that $\boldsymbol{\alpha}_i \succ \boldsymbol{\alpha}_j$ if $i < j$. We assume that any alternative not appearing in the example set is less preferred than those that do. Given an equivalence class we convert it into a boolean formula $\Phi_{\boldsymbol{\alpha}_i} = \vee_{\alpha \in \boldsymbol{\alpha}_i} \Phi_\alpha$, where Φ_α is a boolean formula which is satisfied only by alternative α. This process takes time $\mathcal{O}(|V|)$ per alternative which is $\mathcal{O}(1)$ since the size of the domain is constant, thus the conversion process, which must convert at most two alternatives per example, takes $\mathcal{O}(|\varepsilon|)$ time. This means that the only satisfying assignments for $\Phi_{\boldsymbol{\alpha}_i}$ are those alternatives that make up the equivalence class $\boldsymbol{\alpha}_i$. We assign ranks to the formulas such that the rank of $\boldsymbol{\alpha}_i$ is i, with the formula T being given rank $k + 1$. Since each alternative satisfies either a given $\boldsymbol{\alpha}_i$ or T all alternatives can be compared, thus it is a total preorder. Moreover since we recreated the equivalence classes exactly and lower ranking alternatives are more preferred to those with higher ranks we reproduce all relations given in ε. □

These results allow us to show that LPRF is an NP-complete problem since the witness for any instance of CES be directly converted into a witness for the LPRF problem.

Proposition 3. *LRPF is NP-complete.*

Proof. (Membership) Given an instance of LRPF $(k, \boldsymbol{\varepsilon})$ we can guess a set of examples ε', and verify in polynomial time that it contains at least k examples from each example set $\varepsilon \in \boldsymbol{\varepsilon}$, (as discussed in Proposition 2). Proposition 2 shows there exists an RPF $\Phi_{\varepsilon'}$ corresponding to ε' and thus there exists an RPF which satisfies at least k examples from each example set $\varepsilon \in \boldsymbol{\varepsilon}$ and thus is a valid instance of the LRPF problem.
(Hardness) Hardness is shown by a reduction from CES. Let $I = (k, \boldsymbol{\varepsilon})$ be an instance of the CES problem. If I is a valid instance of CES then by the RPF construction given above I is also a valid instance of the LRPF problem. Conversely, if I is a valid instance of the LRPF problem there exists an RPF Φ which satisfies at least k examples for each example set $\varepsilon \in \boldsymbol{\varepsilon}$. If we extract the examples in $\boldsymbol{\varepsilon}$ which are satisfied by Φ then we have a set of examples showing that I is also a valid instance of CES. Thus an instance is valid wrt the CES problem if and only if it is a valid instance of the LRPF problem. □

The definitions of CES and LRPF are relevant for the egalitarian methods of aggregation. This is because by finding the maximum k for a set of examples represents the maximum number of examples which can be satisfied for all agents, thus maximizing the number of examples satisfied for the least satisfied agent.

4 ALGORITHMS AND EXPERIMENTAL SETUP

We developed two algorithms for determining the efficacy of using heuristic search to learn a maximin aggregation for a group of voters. Both these algorithms take as input a set of examples and an RPF specification. We represent boolean formulas in disjunctive normal form, a sum of *products* (conjunctions of literals) and construct RPFs of such formulas. We specify RPFs by means of a *type*, a triple (i, j, k) of positive integers. In such a triple, k denotes the number of formulas in an RPF, i represents the number of disjuncts in each DNF formula occurring in the RPF, and j represents that number of literals in each disjunct of each formula that occurs in the RPF. For instance, an RPF $(x_1 \wedge \neg x_3, x_2 \wedge x_3)$ has type $(1, 2, 2)$, and an RPF $(\neg x_1 \vee x_4, x_2 \vee x_3, x_1 \vee \neg x_3)$ has type $(2, 1, 3)$.

The use of disjunctive normal form is not only to simply how we specify RPF types, but a matter of practical use. Given an RPF whose formulas are in DNF it is easy to determine what features are considered desirable in an alternative. While we do not use this information in this analysis, using DNF boolean formulas is a practicality measure, and perhaps the use of other forms, such as conjunctive normal form, would alter the performance of our algorithm.

In our algorithms, we represent a specific RPF of type (i, j, k) using a sequence of literals. Where the first $i * j$ literals represent the first DNF (each segment of j literals representing a separate product), the second $i * j$ literals represent the second DNF, and so on. Our algorithm makes no effort to avoid duplication of literals, products, or formulas. The main reason is that the allowance of duplication allows larger RPFs to "simulate" smaller ones. For example the product $a \wedge a \wedge b$ is the same as the product $a \wedge b$. The same goes for products and formulas. Although duplications may be unlikely their allowance strengthens the learning power of an RPF rather than hindering it.

When generating the examples which are used by our learning algorithms we ensured that each example set was consistent and implicitly contained some structure. To do this our example generation process begins by randomly generating preference models. For our experiments we randomly generated either LPMs or RPFs. Our random generation of LPMs consisted of randomly selecting importance orders for attributes and preference orders for each of the attribute domains. When generating RPFs, we used the same specification as our learning algorithm, namely DNF formulas and a type. To generate a product, we randomly selected the required number of propositional variables and then negated each of them with the probability $\frac{1}{2}$. To generate a DNF formula we generated the required number of products in the way just described. To generate an RPF, we used the process above to generate the required number of DNF formulas.

After building our set of preference models, representing the preferences of our voters, we then generated example sets. To generate the example sets we start with the empty set $\varepsilon = \emptyset$ and then generate each example one at a time, ensuring no duplicate examples are in ε. We build our random examples by first selecting two different alternatives α, β from the domain of alternatives $\mathcal{C}(\mathcal{V}, \mathcal{D})$. If an example with the selected alternatives already exists in ε we simply generate a new pair, until the chosen pair does not show up in ε. After we have randomly selected our pair of alternatives we compare them using the agent's preference model and add the example to ε. This process is repeated for each agent and does allow pairs of alternatives to show up in the example sets of multiple agents.

The algorithms we built are based on two common heuristic search techniques. The first is based on simulated annealing [6] and the second is a genetic algorithm [4]. After implementing both algorithms we found that the simulated annealing algorithm consistently outperformed the genetic algorithm, see Figure 1. Due to limited resources we chose to focus solely on the performance of our simulated annealing algorithm.

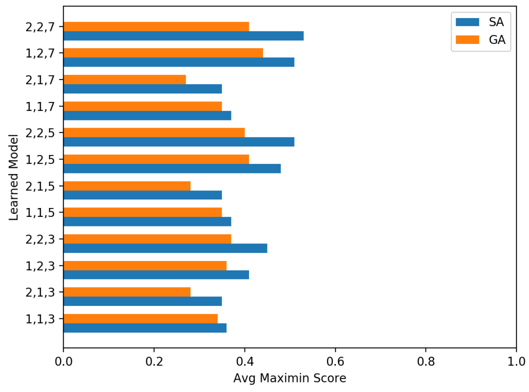

Figure 1. Comparison of using the genetic algorithm versus simulated annealing.

In order to apply a local search technique, simulated annealing, to learn an RPF we must first determine the neighbor between two RPFs. Using our type specification an RPF is specified by a sequence of literals, which we can correctly interpret because we know, from its type, how to divide the sequence. We define two RPFs, of equal size, to be neighbors if their literal sequence representations differ in only one place. For instance, ab and ac, ab and $a\neg c$, and ab and $a\neg b$ are three pairs of neighbors.

Given an instance $(\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_l)$ our simulated annealing algorithm consisted of randomly selecting our starting candidate solution and setting the initial temperature, 100. After this is done we perform a number of iterations where we randomly select a neighbor. If that neighbor performs better, according to maximin, we replace our current candidate solution with that neighbor. If the neighbor performs worse we randomly replace our current solution with that neighbor at probability $e^{\frac{-\Delta}{T}}$, where Δ is the difference in performance of the two neighboring RPFs and T is the current temperature. At the end of each iteration we cool the temperature by dividing it by 1.001. We continue to iterate in this manner until we reach a temperature of 10^{-7} at which point we stop and switch over to a basic hill climbing algorithm. This hill climbing algorithm consists of iterations where we take our candidate solution and test the performance of all its neighbors. We find the best neighbor and replace the candidate solution with that neighbor. We continue to iterate in this fashion until no improving neighbor exists. After the hill climbing is done we output the final candidate solution.

One prevailing concern which arose when learning our joint preference models was that the example sets we were using constitutes only a small portion of all possible examples for the domains we considered. This means that we are not guaranteed that any solution we arrive on will represent the wider preferences of the agents. In order to test how well our solutions generalized the agents preferences we used cross validation, where we remove some examples from the training set to test on after we have learned a solution. More specifically, we used a fold method of cross validation where we created five folds, running our tests five times, each time removing a single fold from the training set.

Further, we considered five agents, each represented by an LPM, for each agent we generated 100 examples, and to aggregate we learned RPFs. Our experiments tweak these aspects one at a time in order to show how each affects the overall results. Since our results depend largely on randomly generated data each result is the average over 100 individual runs of a particular set of experimental settings. In our figures below we specify RPF types using only their type.

In order to evaluate how well our method of heuristic search produces a joint preference model we developed a method for building an LPM based on the maximin score. This greedy algorithm is a simple modification of the greedy algorithm for learning LPMs in a single agent context given by Schmitt and Martignon [12] where we decide which attribute to choose next based on the maximin score rather than total number of examples satisfied. We make no claims as to the performance of the LPMs produced by this greedy algorithm, but we note that in the single agent case it guarantees at least half of all examples are satisfied and so this modification is a reasonable baseline to compare our search methods against.

5 RESULTS AND DISCUSSION

In all our experiments with simulated annealing and the genetic algorithm we found that on average the proportion of examples satisfied for the least satisfied agents, saw around a 0.2 decrease when the solution was applied to our validation set. For example if the maximin score was 0.5 for the training set then we would see a maximin score of 0.3 for the validation set. This implies that our produced models were overfitting the data, although it is very possible that some examples in the validation set could be very different from those in the training set, given that we are using a very small sample of a very large domain to train these models.

We experimented with changing the space of possible alternatives, the type of the agents generating examples, and the number of agents. In the case of changing the number of alternatives we looked at 8 binary attributes, 10 binary attributes, and 8 quaternary, 4-valued, attributes. This allowed us to study the effect of increasing the domain

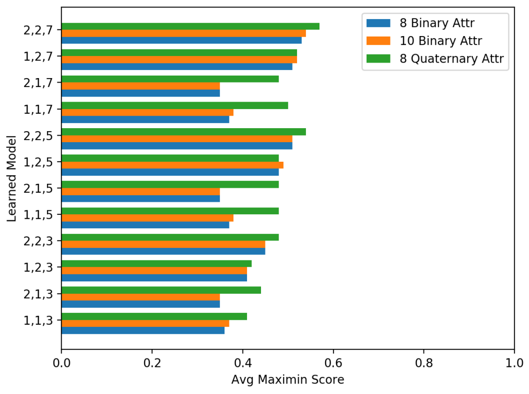

Figure 2. Comparison of performance with different domain sizes.

size, both in the case of the number of attributes and in the size of the attribute domains. In Figure 2 we see an interesting trend, the larger the domain the better the learned solution performs. While this may seem counterintuitive we conjecture that this is merely a result of an increased neighborhood and thus a higher chance of finding an improving neighbor.

By adding attributes to a combinatorial domain we extend the number of neighbors by $2ln$ where l is the number of attributes added and n is the size of the attribute domains, assuming homogeneously sized attribute domains. This increase comes from a larger number of atoms, and their negations, being allowed in the RPF we learn. Similarly when we increase the number of attribute values from 2 to 4 we increase the number of available atoms. An additional factor which may impact performance, specifically for simulated annealing, is the removal of redundant literals. For example, in the binary case $x_{1,0}$ and $\neg x_{1,1}$ are both only satisfied if $x_{1,0}$ is true, since either $x_{1,0}$ or $x_{1,1}$ must be true. This means that for the binary case a solution can have many neighbors which perform identically to itself. By increasing the size of the attribute domains this problem is alleviated and so the number of non-identical neighbors is reduced increasing the number of differently performing neighbors. While this increase for large domains is interesting the performance gains are relatively small and so there is no expectation that this scales as we continue increasing the domain size.

Another aspect of the problem that we experimented with is altering the types of models used to represent agents. In these experiments we generated examples from randomly generated RPFs rather than LPMs. We tested two types of example generating preference formulas $(1, 1, 3)$ and $(2, 2, 7)$, the smallest and largest type of preference formulas we dealt with, respectively. The results from these experiments are shown in Figure 3. We see that learning RPFs from RPFs has a greater average level of performance than learning RPFs from LPMs.

The one variable we found that makes the largest difference in the average maximin score is the number of agents. As one might expect the more agents we have the poorer the performance of our learned aggregated preference model, see Figure 4. We conjecture that more agents means more disagreement, and thus finding a good compromise is harder. It is important to note that even though we roughly double the number of agents both from 2 to 5 and from 5 to 10 there is less of a performance decrease from 5 to 10 than from 2 to 5. This is perhaps a byproduct of fewer new disagreements being

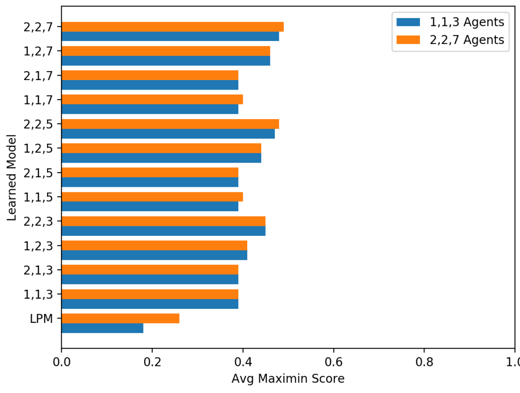

Figure 3. Performance when agents are preference formulas.

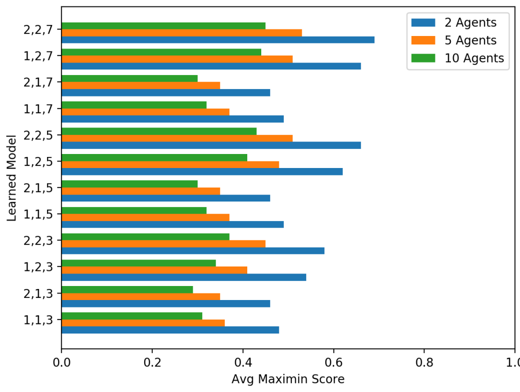

Figure 4. Comparison of performance between differing numbers of generating agents.

introduced as we increase the number of agents.

Another interesting area to look at is the performance of learning in the case where multiple agents agree. For this set of experiments we build only two unique preference models, but allow for multiple agents to be copies of these unique agents, creating two groups of agents, one with three agents, one with only two. Internally each group has the same preferences, but each agent still generates their own example set. As we can see from Figure 5 we have a gain in performance when compared to cases where agents are all unique, see Figures 1 and 3. One additional interesting observation is that the results, while not identical, are very similar to those shown in Figure 4 for cases where we have only two agents. This shows that the learned models are picking up on the implicit structure behind the examples, thus allowing examples which are from the same models, but given as different agents, to reinforce each other and improve overall maximin performance.

While these results show the efficacy of using heuristic search to produce joint preference models it is important to study how they compare to those obtained by another method. As we can see in Figure 3 using heuristic search outperforms our baseline, the greedy LPM method, when the agents are represented by RPFs. We also looked at the performance of the greedy LPM method when learning from LPMs. We found that when using the greedy LPM method

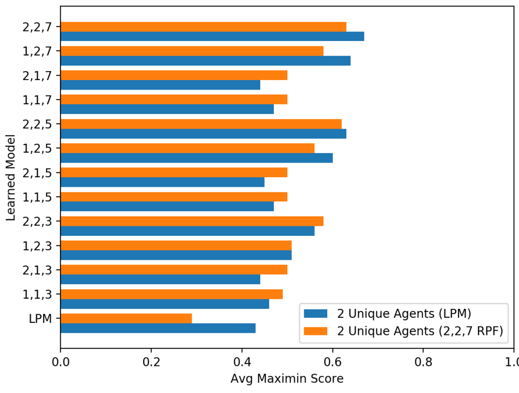

Figure 5. Comparison of performance when there are groups of identical agents.

to learn an LPM from examples produced by five LPMs the average maximin score was 0.42. If we compare this with the five agent data from Figure 4 then we see that there are cases where the greedy method outperforms the use of simulated annealing to learn RPFs, but not all cases are outperformed by the greedy method. Using a sufficiently complex RPF type produces better average performance than the greedy method, for example $(2, 2, 7)$ RPFs.

6 RELATED WORKS

The problem of aggregating over combinatorial domains is not new and one popular method is to apply voting procedures to compact preference representations. Of interest to this work specifically is the use of answer set programming to aggregate LP-trees, a type of lexicographic preference language, using voting rules [9]. Others have also studied the use of voting rules and LP-trees [7], but the use of answer set programming (ASP) is unique. By using ASP, Liu and Truszczyński [9] are showing that search based techniques, which are used by ASP solvers, can be applied to the problem of aggregation over combinatorial domains.

We are not the first to directly use heuristic search, or similar methods, in order to learn preferences. A recent paper by Allen, Siler, and Goldsmith showed that it is possible to learn tree-structured CP-nets using local search [1]. That work showed the viability of using local search to learn preference models in a complex environment, including when the presented data has noise. If one views rank aggregation as learning a preference model from noisy data, where that noise occurs due to the disagreement of agents, rather than poor data, then the problem studied by Allen et. al. is similar to the problem presented in this work. Where our work differs from their work, other than the choice of preference representation language, is that we are looking at aggregation with possibly different preference representation languages being used for the agents and the joint preference model. Allen et. al. only looked at learning from examples generated by CP-nets in order to learn CP-nets, and only in a one to one context, rather than as a tool for aggregation.

7 CONCLUSION

While preliminary, this work shows the efficacy of using heuristic search to produce joint preference models. These results were obtained in a setting where the direct method of building such models is computationally difficult and the domain of alternatives is large. An important component of the experiments performed was the use of a learned model type which could be different from that of the set of agents who generated the examples, specifically learning RPFs to aggregate examples from LPMs. These results were compared against a different baseline greedy algorithm. Overall, we determined that using simulated annealing to learn joint preference models is a promising avenue of research.

While this work has answered some questions as to the efficacy of using simulated annealing in rank aggregation, it has also introduced some open questions and further lines of research. We intend to continue this work by testing more preference representations, both as voters and as learning targets, as well as changing the social welfare function we use to evaluate our models. On the theoretical side while the general problem of maximin rank aggregation may be NP-complete we are using a limited subset of RPFs. These limits, or similar, may allow for a polynomial time algorithm to create sufficient joint preference models, or failing that there might be a good greedy algorithm, similar to that used to learn LPMs. It might prove fruitful to test learning joint preferences when the examples used are generated by a diverse set of agents, where agents can be of different preference representation languages. This work is also limited by its use of purely synthetic data. The question remains whether or not our technique would perform well when using real world data.

ACKNOWLEDGEMENTS

This work was funded by the NSF under the grant number IIS-1618783.

REFERENCES

[1] Thomas E Allen, Cory Siler, and Judy Goldsmith, 'Learning tree-structured CP-nets with local search', in *The Thirtieth International Flairs Conference*, (2017).

[2] Gerhard Brewka, Ilkka Niemelä, and Miroslaw Truszczynski, 'Answer set optimization', in *IJCAI*, volume 3, pp. 867–872, (2003).

[3] Peter C Fishburn, 'Lexicographic orders, utilities and decision rules: A survey', *Management science*, **20**(11), 1442–1471, (1974).

[4] John Henry Holland et al., *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*, MIT press, 1992.

[5] Souhila Kaci, *Working with Preferences: Less is More*, Springer, 2011.

[6] Scott Kirkpatrick, C Daniel Gelatt, and Mario P Vecchi, 'Optimization by simulated annealing', *science*, **220**(4598), 671–680, (1983).

[7] Jérôme Lang, Jérôme Mengin, and Lirong Xia, 'Aggregating conditionally lexicographic preferences on multi-issue domains', in *Principles and Practice of Constraint Programming*, pp. 973–987. Springer, (2012).

[8] Jérôme Lang and Lirong Xia, 'Sequential composition of voting rules in multi-issue domains', *Mathematical Social Sciences*, **57**(3), 304–324, (2009).

[9] Xudong Liu and Miroslaw Truszczynski, 'Aggregating conditionally lexicographic preferences using answer set programming solvers', in *International Conference on Algorithmic DecisionTheory*, pp. 244–258. Springer, (2013).

[10] Xudong Liu and Mirosław Truszczyński, 'New complexity results on aggregating lexicographic preference trees using positional scoring rules', in *Proceedings of the 6th International Conference on Algorithmic Decision Theory*, (2019).

[11] *Economics and Computation: An Introduction to Algorithm Game Theory, Computational Scoial Choice, and Fair Division*, ed., Jörg Rothe, Springer, 2016.

[12] Michael Schmitt and Laura Martignon, 'On the complexity of learning lexicographic strategies', *Journal of Machine Learning Research*, **7**(Jan), 55–83, (2006).