Computing Optimal Subsets*

Maxim Binshtok
Dept of Comp. Sci.
Ben-Gurion University
maximbi@cs.bgu.ac.il

Ronen I. Brafman
Dept of Comp. Sci.
Ben-Gurion University
brafman@cs.bgu.ac.il

Solomon E. Shimony
Dept of Comp. Sci.
Ben-Gurion University
shimony@cs.bgu.ac.il

Ajay Martin
Dept of Comp. Sci.
Stanford University
ajaym@cs.stanford.edu

Craig Boutilier
Dept of Comp. Sci.
University of Toronto
cebly@cs.toronto.edu

Abstract

Various tasks in decision making and decision support require selecting a preferred subset of items from a given set of feasible items. Recent work in this area considered methods for specifying such preferences based on the attribute values of individual elements within the set. Of these, the approach of (Brafman *et al.* 2006) appears to be the most general. In this paper, we consider the problem of computing an optimal subset given such a specification. The problem is shown to be NP-hard in the general case, necessitating heuristic search methods. We consider two algorithm classes for this problem: direct set construction, and implicit enumeration as solutions to appropriate CSPs. New algorithms are presented in each class and compared empirically against previous results.

Introduction

Work on reasoning with preferences focuses mostly on the task of recognizing preferred elements within a given set. However, recent work has begun to consider the problem of selecting an optimal *subset* of elements. Optimal subset selection is an important problem with many applications: the choice of feature subsets in machine learning, selection of a preferred bundle of goods (as in, e.g., a home entertainment system), finding the best set of items to display on the user's screen, selecting the best set of articles for a newspaper or the best members for a committee, etc. Of particular interest is the case where the candidate items have attributes, and their desirability depends on the value of their attributes.

Optimal subset selection poses a serious computational challenge: the number of subsets is exponential in the size of the domain, and so the number of possible orderings is doubly exponential. This combinatorial explosion is problematic both for the preference elicitation task and for the task of optimal subset selection given some specification. Existing approaches attempt to overcome this by considering properties of subsets. (desJardins & Wagstaff 2005) offer a formalism for preference specification in which users can specify their preferences about the set of values each attribute attains within the selected set of items, asserting whether it should be diverse or concentrated around some particular value. (desJardins & Wagstaff 2005) also suggests a heuristic search algorithm for finding good sets.

(Brafman *et al.* 2006) present a more general two tiered approach for set preferences. This approach consists of a language for specifying certain types of set properties combined with an arbitrary preference specification language (for single elements). The basic idea is that each set is now associated with particular property values, and sets are ordered based on these values. This approach is intuitive and quite powerful. It generalizes the approach of (desJardins & Wagstaff 2005) because diversity and specificity are just two set properties that it can capture. More general properties that consider multiple attributes can be expressed, as well as more general conditional preferences over the values of these properties.

Essentially, (Brafman *et al.* 2006) reduce the problem of specifying preferences over sets to that of specifying preferences over single items. Now, the items are the possible sets and their attributes are their (user-defined) set-property values. Thus, in principle, this approach allows us to re-use any algorithm for computing preferences over single items. Indeed, (Brafman *et al.* 2006) consider the case where preference specification is based on TCP-nets (Brafman, Domshlak, & Shimony 2006) and adapt existing algorithms for computing optimal elements in TCP-nets to the problem of computing an optimal subset. Of course, the main problem is that the number of "items" is very large. Thus, the set selection algorithm must utilize the special structure of these "items". (Brafman *et al.* 2006) does this, to a certain extent, but much can be done to improve their algorithm.

This paper takes a closer look at the problem of computing an optimal subset given a preference specification. We consider two classes of algorithms: search over property values and search over sets. The first extends and improves the algorithm of (Brafman *et al.* 2006) by better exploiting properties of the special search space, employing stronger forward checking techniques and incrementally building on information obtained from previously solved instances. The second, and quite different, approach explicitly performs branch and bound search over the space of sets. We formulate these algorithms in quite a general way, showing how they could be used with an arbitrary preference spec-

*Brafman and Martin were supported in part by NSF grant IIS-0534662, Binshtok, Brafmam and Shimony were supported by Deutsche Telekom Laboratories at Ben-Gurion University, by the Paul Ivanier Center for Robotics Research and Production Management, and by the Lynn and William Frankel Center for Computer Science.

ification. Then, we proceed to compare their performance empirically.

In the next section we describe the property-based approach to set preference specification. Then, we describe the two algorithmic approaches to optimal subset selection as well as some complexity results. Finally, the algorithms are evaluated empirically.

Specifying Set Preferences

Our discussion here follows (Brafman *et al.* 2006).

Item Properties. The formalism we use for set-preference specification makes one fundamental assumption: the items from which sets are built are associated with attributes, and the values of these attributes are what distinguish different elements. We shall use $\mathcal{O}$ to denote the set of individual items, and $\mathcal{X}$ to denote the set of attributes of these elements. For example, imagine that the "items" in question are US senate members, and the attributes and their values are: Party affiliation (Republican, Democrat), Views (liberal, conservative, ultra conservative), and Experience (experienced, inexperienced).

Given the set $\mathcal{X}$ of attributes, we can talk about primitive item properties. For example: "Senate members with liberal views." Or, we can talk about more general item properties: "inexperienced, conservative senate members." More formally, define:

$$\overline{\mathcal{X}} = \{X = x \mid X \in \mathcal{X}, x \in Dom(X)\}$$

Let $\mathcal{L}_{\overline{\mathcal{X}}}$ be the propositional language defined over $\overline{\mathcal{X}}$ with the usual logical operators. $\mathcal{L}_{\overline{\mathcal{X}}}$ provides us with a language for describing properties of individual elements. Since objects in $\mathcal{O}$ can be viewed as models of $\mathcal{L}_{\overline{\mathcal{X}}}$, we can write $o \models \varphi$ whenever $o \in \mathcal{O}$ and o is an item that satisfies the property $\varphi \in \mathcal{L}_{\overline{\mathcal{X}}}$.

Set Properties. We can define set properties based on properties of items in a set. For example: the property of having at least two Democrats, or having more Democrats than Republicans. More generally, given any item property $\varphi \in \mathcal{L}_{\overline{\mathcal{X}}}$, we can talk about the number of items in a set that have property φ, which we denote by $|\varphi|(S)$, i.e., $|\varphi|(S) = |\{o \in S | o \models \varphi\}|$. Often the set S is implicitly defined, and we simply write $|\varphi|$. Thus, $|\text{Experience=experienced}|(S)$ is the number of experienced members in S. Often, we simply abbreviate this as $|\text{experienced}|$.

$|\varphi|(\cdot)$ is an integer-valued property of sets. We can also specify boolean set properties as follows: $\langle|\varphi| \text{ REL } k\rangle$, where $\varphi \in \mathcal{L}_{\overline{\mathcal{X}}}$, REL is a relational operator over integers, and $k \in \mathbb{Z}^*$ is a non-negative integer. This property is satisfied by a set S if $|\{o \in S | o \models \varphi\}|$ REL k. In our running example we use the following three set properties:

P_1 : $\langle|\text{Republican} \vee \text{conservative}| \geq 2\rangle$

P_2 : $\langle|\text{experienced}| \geq 2\rangle$

P_3 : $\langle|\text{liberal}| \geq 1\rangle$

P_1 is satisfied by sets with at least two members that are either Republican or conservative. P_2 is satisfied by sets with at least 2 experienced members. P_3 is satisfied by sets with at least one liberal.

We can also write $\langle|\varphi| \text{ REL } |\psi|\rangle$, with a similar interpretation. For example, $\langle|\text{Republican}| > |\text{Democrat}|\rangle$ holds for sets containing more Republicans than Democrats. An even more general language could include arithmetic operators (e.g., require twice as many Republicans as Democrats) and aggregate functions (e.g., avg. number of years on the job). We do not pursue such extensions here, but they make little fundamental impact on our algorithms.

Set Preferences. Once we have set properties, we can define preferences over their values using any preference specification formalism. Here we will discuss two specific formalisms: TCP-nets (Brafman, Domshlak, & Shimony 2006), an extension of CP-nets (Boutilier *et al.* 2004), and Generalized Adaptively Independent (GAI)-utility functions (Bacchus & Grove 1995; Fishburn 1969). The former is a qualitative preference specification, yielding a partial order. The latter is a quantitative specification, which can represent any utility function.

Let $\mathcal{P} = P_1, \ldots, P_k$ be some collection of set properties. A TCP-net over $\mathcal{P}$ depicts statements of the following type:

Conditional Value Preference Statements. "If $P_{i_1} = p_{i_1} \wedge \cdots \wedge P_{i_j} = p_{i_j}$ then $P_l = p_l$ is preferred to $P_l = p_l'$." That is, when $P_{i_1}, \ldots, P_{i_j}$ have a certain value, we prefer one value for P_l over another value for P_l.

Relative Importance Statements. "If $P_{i_1} = p_{i_1} \wedge \cdots \wedge P_{i_j} = p_{i_j}$ then P_l is more important than P_m." That is, when $P_{i_1}, \ldots, P_{i_j}$ have a certain value, we prefer a better value for P_l even if we have to compromise on the value of P_m.

Each such statement allows us to compare between certain pairs of elements (sets in our case) as follows:

- The statement **"If** $P_{i_1} = p_{i_1} \wedge \cdots \wedge P_{i_j} = p_{i_j}$ **then** $P_l = p_l$ **is preferred to** $P_l = p_l'$**"** implies that given any two sets S, S' for which (1) $P_{i_1} = p_{i_1} \wedge \cdots \wedge P_{i_j} = p_{i_j}$ holds. (2) S satisfies $P_l = p_l$ and S' satisfies $P_l = p_l'$. (3) S and S' have identical values on all attributes except P_i. We have that S is preferred to S'.
- The statement **"If** $P_{i_1} = p_{i_1} \wedge \cdots \wedge P_{i_j} = p_{i_j}$ **then** P_l **is more important than** P_m**"** implies that given any two sets S, S' for which (1) $P_{i_1} = p_{i_1} \wedge \cdots \wedge P_{i_j} = p_{i_j}$ holds. (2) S has a more preferred value for P_l. (3) S and S' have identical values on all attributes except P_l and P_m. We have that S is preferred to S'. (Notice, that we do not care about the value of P_m if P_l is improved.)

We refer the reader to (Brafman, Domshlak, & Shimony 2006) for more details on TCP-nets, their graphical structure, their consistency, etc. The algorithms in this paper, when used with TCP-nets, assume an acyclic TCP-net. This ensures its consistency, and ensures the existence of certain "good" orderings of $\mathcal{P}$.

As an example, consider the following preferences of the president for forming a committee. He prefers at least two members that are either Republican or conservative, i.e., he prefers P_1 to $\bar{P}_1$ unconditionally. (Depending on context, P is used to denote both the property P and the value P =*true*. We use $\bar{P}$ to denote P =*false*.) If P_1 holds, he prefers P_2 over $\bar{P}_2$ (i.e., at least two experienced members), so that the committee recommendations carry more weight. If $\bar{P}_1$

holds, he prefers $\bar{P}_2$ to P_2 (i.e., all but one are inexperienced) so that it would be easier to influence their decision. The president unconditionally prefers to have at least one liberal, i.e., he prefers P_3 to $\bar{P}_3$, so as to give the appearance of balance. However, P_3 is less important than both P_1 and P_2. There is an additional external constraint (or possibly preference) that the total number of members be 3.

GAI value functions map the elements of interest into real-value functions of the following form: $U(S) = \sum_{i=1,\ldots,n} U_i(\mathcal{P}_i(S))$, where $\mathcal{P}_i \subset \mathcal{P}$ is some subset of properties. For example, the President's preferences imply the following GAI structure: $U(S) = U_1(P_1(S), P_2(S)) + U_2(P_3(S))$ because the President's conditional preferences over P_2's value tie P_1 and P_2 together, but are independent of P_3's value. U_1 would capture the weight of this conditional preference, combined with the absolute preference for P_1's value. U_2 would represent the value of property P_3. We might quantify these preferences as follows: $U_1(P_1, P_2) = 10$, $U_1(P_1, \bar{P}_2) = 8$, $U_1(\bar{P}_1, P_2) = 2$, $U_1(\bar{P}_1, \bar{P}_2) = 5$; while $U_2(P_3) = 1, U_2(\bar{P}_3) = 0$. Of course, many other quantifications are possible.

Finding an Optimal Subset

In this section we consider two classes of algorithms for finding an optimal subsets which differ in the space in which they search. In the next section, we compare them empirically. We assume we are given a preference specification and a set S of available items. Our goal is to find a subset of S that is optimal with respect to the preference specification. That is, a set $S_{opt} \subseteq S$ such that for any other set $S' \subseteq S$, we have that the properties S_{opt} satisfies are no less desirable than the properties S' satisfies. For our running example we use the following set of candidates:

o_1	Republican	conservative	inexperienced
o_2	Republican	u. conservative	experienced
o_3	Democrat	conservative	experienced
o_4	Democrat	liberal	experienced

Searching over CSPs

Here is a conceptually simple way of computing an optimal subset, pursued in (Brafman *et al.* 2006): (1) Generate all possible combinations of set-property values in some order. (2) For each such combination of properties, search for a set that satisfies these properties. (3) Output a set satisfying the best set of satisfiable properties found.

To make this approach efficient, we have to do two things: (1) Find a way to prune many property combinations as sub-optimal. (2) Given a set of properties, quickly determine whether a set satisfying these properties exists.

Searching over Property Combinations. For the first task consider the following approach: order set properties in some manner. Given an ordering $P_1, \ldots, P_k$, incrementally generate a tree of property combinations. The root node contains an empty set of properties. For each node n with property values $P_1 = p_1, \ldots, P_j = p_j$, and for every possible value p_{j+1} of property P_{j+1}, add the node corresponding to $P_1 = p_1, \ldots, P_j = p_j, P_{j+1} = p_{j+1}$ as n's child. Leaf nodes in this tree assign a value to each set property in $\mathcal{P}$. Each node in this tree is also implicitly associated with a (possibly empty) set of subsets – those subsets that have the property values assigned to this node. Such a tree for our example appears in Figure 1.

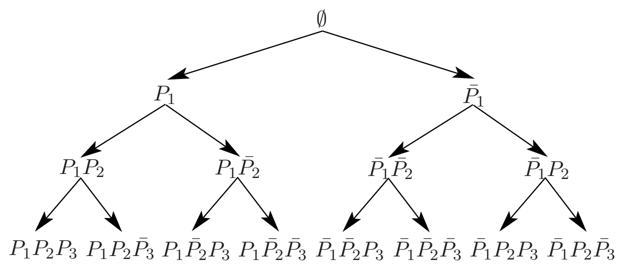

Figure 1: Search Tree

In our search for an optimal set, we will expand this tree of set property-combinations, but we would like to expand as few nodes of it as possible by pruning certain property combinations as sub-optimal or unsatisfiable. A standard way to do this is using branch&bound search. This requires that we be able to associate an upper and a lower bound on the value of the best subset of each combination of properties.

B&B can be implemented in various ways, e.g., using best-first search or depth-first search. In all cases, the order we associate with properties and their values affects our ability to prune nodes throughout the search process. To get the most leverage out of our bounds, we would like to order the children of a node that correspond to a more valuable property value, earlier. Moreover, when we order the properties themselves, we want properties that can potentially contribute more to appear earlier in this ordering. In our running example, P_1's value has greater influence than P_2's value, so P_1 is assigned first. For P_1, *true* is a better value. For P_2, *true* is a better value when P_1 is *true*, and *false* is a better value otherwise. This is reflected in the tree above.

Finding Sets that Satisfy a Collection of Properties. Consider some collection of set-property values. To see if there is a subset of available items that satisfies these properties, we set up the following CSP. The CSP has a boolean variable for every available item. Intuitively, a value of 1 means that the item appears in our set, and a value of 0 indicates that it does not appear in our set. In our example, we use $x_1, \ldots, x_4$ for $o_1, \ldots, o_4$ respectively. Next, we translate every set property value into a constraint on these variables. For P_1 we have: $C_1 : x_1 + x_2 + x_3 \geq 2$ expressing the fact that we want at least two of the elements that satisfy Republican $\vee$ conservative, because o_1, o_2, o_3 are all candidates that are either Republican or conservative. For $\bar{P}_1$ we have $\bar{C}_1 : x_1 + x_2 + x_3 < 2$. For P_2 we would have $C_2 : x_2 + x_3 + x_4 \geq 2$ because o_2, o_3, o_4 are the experienced members. For P_3 we would have $C_3 : x_4 \geq 1$. Thus, every collection of set-property values corresponds to a CSP that is solvable IFF our database of items contains a subset with these properties. Moreover, the solution of this CSP provides us with such a set.

It is worth briefly pointing out the difference between the CSPs we generate here and the more typical CSPs discussed

Algorithm 1 B&B over CSPs

```
1: Fix an ordering over properties $P$
2: Fix an ordering over the values of each property
3: Fix an ordering over all available items
4: $Q \leftarrow \{Node(\emptyset, \emptyset, 0, \infty)\}$;
5: Best $\leftarrow \emptyset$
6: while $Q$ is not empty do
7:     $N = Pop(Q)$;
8:     CSP($N$);
9:     if $N$.Solution is not false & $N$.Upper $>$ Value(Best) then
10:        if Value ($N$.Solution) $>$ Value(Best) then
11:            Best $\leftarrow N$.Solution;
12:        Let $P$ be the next set property unassigned in $N$;
13:        for each possible value $p$ of $P$ do
14:            Create a new node $N'$ that extends $N.assigned$ with
               $P = p$;
15:            $N'.solution \leftarrow N.solution$
16:            Insert $N'$ into $Q$ //Its position depends on search strat-
               egy used
17: Return Best;
Function CSP(node $N$)
1: Generate set constraints from $N.assigned$
2: Let $x_1, \ldots x_m$ be boolean variables, one for every available
   item;
3: $x_i$ is initialized to true IFF item $i$ is in $N.solution$
4: Continue depth-first traversal over possible assignments to the
   $x_i$s. (We used fixed variable ordering, forward pruning, and a
   form of bounds consistency for cardinality constraints).
5: if solution not found then
6:     Node.solution = false
7: else Node.solution = solution
8: Node.lower = Value(Node.solution)
9: Node.upper = upper-bound estimate
```

in the literature. Most work on CSPs deals with constraints that are local to a small set of variables – most commonly two variables. On the other hand, the constraints in the CSPs generated in our optimization process are global and are relevant to all variables. Moreover, there is a sense in which it is meaningful to talk about partial assignments in our context. Variables that are not assigned a value can be regarded as if assigned the value "0" by default, i.e., an item, by default, does not belong to the set.

Because collections of set properties map to CSPs, each node in our tree of set-property collections maps to a CSP, and the entire tree can be viewed as a tree of CSPs. The important property of the tree-of-CSPs is that the children of each CSP node are CSPs obtained by adding one additional constraint to the parent CSP – a constraint corresponding to the additional property value that we want the set to satisfy. This implies that if some node in this tree is unsatisfiable, then all of its descendants are unsatisfiable. (Brafman *et al.* 2006) use this to prune the search tree. We make stronger use of the nature of this search tree, recognizing that we can reuse the work done on a parent node to speed up the solution of its children.

To see this, consider some CSP C in our Tree-of-CSPs. Let set S be some solution to this CSP, and consider C', a child CSP of C. C' extends C with another constraint c. Thus, any set S' that we ruled-out in our solution for C, can be ruled out in our solution for C'. If C and C' consider sets in the same order (e.g., by using the same property and property-value ordering), then when we solve C', we can start from the leaf node corresponding to S, the solution generated for C. Moreover, when c represents a boolean property then if S is not a solution to $C' = C \cup c$, it must be a solution to $C \cup \neg c$, which is the sibling of C'. Using these ideas, we share the work done on different CSPs nodes of the Tree-of-CSPs. In fact when all set properties are boolean this approach needs to backtrack over each property at most once. This considerably improves the performance of the algorithm with respect to (Brafman *et al.* 2006).

The algorithm's pseudo-code appears above. It is formulated for the quantitative case (the qualitative case is essentially the same, but there are minor differences and space constraints limit our ability to discuss them). The *node* data structure has four attributes: *assigned* is a list of assigned property values; *solution* is a set with these properties or the value *false*; *lower* and *upper* are its associated lower and upper bounds. The function Value(S) returns the value of a set S. In the pseudo-code we assume a fixed ordering over values (Line 2), but one can vary it depending on earlier values (as we do in our implementation). Finally, we have not specified the particular search strategy used by B&B, which depends on the insertion order in Line 16.

Consider our running example. Recall that we also have a constraint that the set size is 3, which translates into $C: x_1 + x_2 + x_3 + x_4 = 3$. The first CSP we consider has $\{C, C_1\}$ as its only constraints. Assume the CSP variables are ordered x_1, x_2, x_3, x_4, with value 1 preceding 0. In that case, the first solution we will find is $s_1: x_1 = 1, x_2 = 1, x_3 = 1, x_4 = 0$. Our next CSP adds the constraint C_2. When we solve it, we continue to search (using the same order on the x_i's and their values) from the *current* solution s_1, which turns out to satisfy C_2 as well. Thus, virtually no effort is required for this CSP. Next, we want to satisfy C_3. This corresponds to a leaf node in the tree-of-CSPs which corresponds to the complete assignment $P_1 P_2 P_3$ to the set properties. Our current item set does not have a liberal, so we have to continue to the assignment $s_2: x_1 = 1, x_2 = 0, x_3 = 1, x_4 = 1$ (requiring us to backtrack in the CSP-solution space over the assignments to x_4, x_3, and x_2). We now have a set that satisfies the properties in the leftmost leaf node in our tree-of-CSPs. If we can prove that this property combination is optimal using our upper/lower bounds, we are done. Otherwise, we need to explore additional nodes in our tree-of-CSPs. In that case, the next CSP we would visit corresponds to $P_1, P_2, \bar{P}_3$, with constraints $\{C, C_1, C_2, \bar{C}_3\}$. But, we already have a solution to this node: s_1. s_1 was a solution to the parent of our current CSP, but it was not a solution to its sibling $\{C, C_1, C_2, C_3\}$, so it must satisfy $\{C, C_1, C_2, \bar{C}_3\}$.

NoGood Recording

The standard definition of a NoGood in the CSP literature is that of a partial assignment that cannot be extended to a full solution of the problem. Once we learn a NoGood, we can use it to prune certain paths in the search tree. The smaller the NoGood, the more occasions we have to use it, the greater its pruning power. Thus, we typically attempt to recognize minimal NoGoods, and different techniques were developed to perform NoGood resolution in order to produce the best and most general NoGoods possible.

As noted earlier, the CSPs we generate differ significantly from the more typical binary CSPs. Consequently, the NoGood recording algorithm has to be adapted accordingly. In particular, because our constraints are global, it makes sense to try to generate NoGoods that are global, too. Thus, instead of recording assignments to variables, we record the influence of the current assignment on the constraints. Every variable influences a set of constraints[1]. Thus, as a NoGood, we store the influence the set selected so far has on all the constraints. That is, suppose we have generated the set S_1, and recognized that it is not extensible to a set satisfying the constraints. (This immediately follows from the fact tat we backtracked over this set.) We now generate a NoGood N that records for each property associated with each constraint, how many items satisfying this property occur in S_1. Now, suppose we encounter a different set S_2 that has the same effect N on the constraints. If there are fewer options to extend S_2 than there are to extend S_1, we know that S_2, as well, cannot be extended into a solution. However, if there are more options to extend S_2 than S_1, we cannot conclude that S_2 is a NoGood at this point. To recognize the amount of options that were available to extend S_1 we record, beyond the actual NoGood N, the level (depth) in the assignment tree in which it was generated. Given that the CSP solver uses a fixed variable ordering, we know that if we encounter a set S that generates the same properties as the NoGood N, at a level no higher than that of S_1, we can safely prune its extensions. The reason for that is that there are no more options to extend S than there were to extend S_1.

Correctness The correctness of the NoGood recording mechanism proposed here, depends on having a fixed variable ordering, as well as a specific value ordering for all the variables in the CSP, namely, $\langle 1, 0\rangle$. To show correctness, we should pay attention that a NoGood can be used only after it was recorded. Consequently any node using a NoGood would be to the right in the search tree of a node the NoGood was recorded at. We would like to stress again that since the constraints are global, it does not matter what items were taken, but rather what influence these items had on the constraints. Any two sets having exactly the same influence on the constraints are identical for the optimization purposes. For a more detailed and formal description of the NoGood recording mechanism and its correctness, we refer the reader to (**?**).

Theorem 1 *Every subtree not explored due to use of a NoGood does not have a valid solution.*

Proof: Let v_1 be the node we recorded NoGood n at while backtracking from it. Now assume by negation that there exists a node v_2 deeper in the search tree than v_1, that is pruned by NoGood n, but it is on the path to a valid solution.

[1]We assume without loss of generality that every item in the selection set influences at least one constraint in the constraint set $\mathscr{C}$, since items that influence no constraint can be safely eliminated from the selection set.

Since v_2 is deeper than v_1, there is exactly one path from v_1 to v_2 assigning 0 value to all the intermediate nodes (variables) between v_1 and v_2: p_{zero}. Assigning 0 value to all intermediate variables does not change the subset selected along the path leading from the root to v_1: p_{r1}, thus we reach node v_2 with the same set selected as in node v_1, obviously having exactly the same influence on the constraints cardinalities. By assumption, there exists a path leading to a solution which starts at v_2 and has the cardinalities recorded in the NoGood n: p_{2s}. But then, we can obtain a path $p_{rs} = p_{r1}+p_{zero}+p_{2s}$ leading to solution and going through v_1. However this contradicts the backtracking from v_1 we performed earlier and recorded the NoGood. Please see Fig. 2 for the visual outline of the proof.

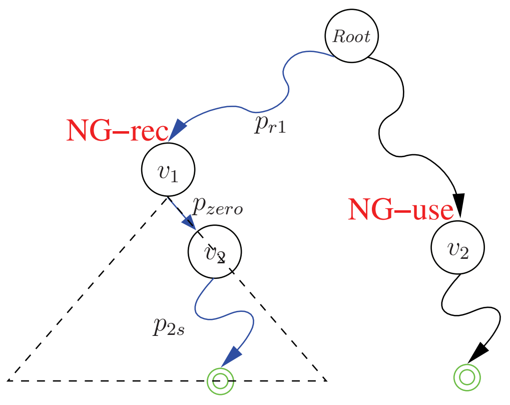

Figure 2: The left path denotes the contradicting solution path — $p_{rs} = p_{r1} + p_{zero} + p_{2s}$

Finally, it is important to establish that NoGoods remain valid during the incremental addition of constraints, i.e, we can re-use NoGoods found in one CSP within a CSP that appears below it in the tree-of-CSPs. The reasoning is very simple. Obviously adding a new constraint does not enlarge the available solutions space, thus previously recorded NoGoods will remain valid after the addition of a new constraint.

Searching in Sets Space

In the Tree-of-CSP approach, each node searches implicitly over a space of sets (i.e., the possible solutions to each CSP). When we examine CSPs generated along a path in the Tree-of-CSPs, we see that a set will never be generated twice in our algorithm. However, different path within the Tree-of-CSPs will consider overlapping item sets. Why not simply go over the possible sets, generating each set once, and find the optimal set directly. This is precisely the idea behind the Branch&Bound in Set-Space algorithm.

At each stage the algorithm maintains a queue of sets. For each set, it maintains an upper and a lower bound on the maximal value of all sets that extend that set. It selects the first set S in the queue, whose lower and upper bounds are denoted L and U, respectively. Next, it removes all sets whose upper bound $\leq L$ and it adds all children of S into the queue. The children of a set are all sets that extend it with a single element. The pseudo-code is described below. Different implementations of the algorithm differ in how

Algorithm 2 B&B in the Space of Sets

```
1: $Q \leftarrow \{\emptyset\}$
2: Best $\leftarrow \emptyset$
3: while $Q$ contains an item $S$ s.t.
        Upper-Bound $(S) >$ Value(Best) do
4:   $S \leftarrow$ element of $Q$ with best upper bound.
5:   Insert all sets that extend $S$ with one element into $Q$.
6:   if $S'$ is a new set in $Q$ such that $Value(S') >$ Value(Best)
     then
7:     Best $\leftarrow S'$
8:   Remove from $Q$ any set $S$ s.t.
     Lower-Bound(Best) $>$ Upper-Bound $S$
9: Return Best
```

they sort the queue. The best-first version sorts the queue according to a heuristic value of the set – we use the upper bound. The depth-first version always positions the children of the newly expanded node at the front of the queue. We implemented and tested both versions.

The method used to generate bounds for a set S must depend on the actual preference representation formalism and the idea is more natural given a quantitative utility function. For a lower bound we use the value of S. For an upper bound, we proceed as follows: First, we consider which set-property values are consistent with S. That is, for each set-property, we examine what values S and any of its supersets can potentially have. For example, consider P_2 and suppose S contains a single experienced member. So currently, $\bar{P}_2$ holds. However, we can satisfy P_2 if we add one more experienced member. Thus, both values of P_2 are consistent with S. However, if we had two experienced members in S then $\bar{P}_2$ is inconsistent with S because no matter who we add to S, we can never satisfy $\bar{P}_2$. Next, given the set of possible values for each set-property w.r.t. set S, we can bound the value of S and its supersets by maximizing values locally. In a GAI utility function, we can look at each local function U_i, and consider which assignment to it, from among the consistent values, would maximize its values. This is an over-estimate, since we don't know whether these joint assignments are consistent. Similar ideas can be used with other quantitative representations, as in various soft-constraint formalisms (Bistarelli *et al.* 1999).

Consider our running example with the GAI utility function given earlier with search based on DFS B&B and items ordered o_1, o_2, o_3, o_4. We start with the empty set. Its property values are $\bar{P}_1, \bar{P}_2, \bar{P}_3$. Thus, the lower bound, which is the value of the empty-set, is 5. For the upper-bound, we consider the best properties that are individually consistent with the current set, which are P_1, P_2, P_3, and their value is 11. **Best** is also initialized to the empty set. Next, we generate all of the children of the current node, which are all singleton sets: $\{o_1\}, \{o_2\}, \{o_3\}, \{o_4\}$. They all have lower and upper bounds identical to the empty set, and are inserted into the queue. Suppose $\{o_1\}$ is the first queue element, and we select it for expansion. We now get $\{o_1, o_2\}, \{o_1, o_3\}, \{o_1, o_4\}$. Their lower and upper bounds are $(8, 11), (8, 11), (6, 11)$, respectively. We've also updated **Best** to be $\{o_1, o_2\}$. We use DFS B&B, so we insert them in the front of the queue, and proceed to expand $\{o_1, o_2\}$, obtaining $\{o_1, o_2, o_3\}, \{o_1, o_2, o_4\}$ with lower and upper bounds $(10, 11)$ and $(11, 11)$. With a lower bound of 11 for $\{o_1, o_2, o_4\}$ we can prune all nodes currently in the queue, and we are done.

An important issue for DFS B&B is the order by which sets are generated. In our implementation, an item is ordered based on its weight, which is a sum of the utility of the properties it can help satisfy. For example, a conservative member such as o_1 could help us satisfy P_1.

Qualitative preferences typically induce a partial ordering over property collections. In this case, it is harder to generate strict upper and lower bounds – as they must be comparable to any possible solution. One way to handle this is to linearize the ordering and require the stronger property of optimality w.r.t. the resulting total order. Here, TCP-nets present themselves as a good choice for two reasons. First, there is an efficient and simple way of generating a utility function consistent with a TCP-net. This utility function retains the structure of the original network which is important to make the bound computations efficient (i.e., each U_i depends on a small number of property values). Second, we can actually work with TCP-nets directly. Given a node n, we can generate a lower bound L and an upper bound U such that any leaf node in n's sub-tree is comparable to L and U, and use the algorithm exactly as in the case of GAI utility functions. We discus the specifics for TCP-nets in more detail in the full version of this paper.

Complexity

We presented two algorithms for computing an optimal subset. This begs the question of whether the problem itself is hard. With external constraints, the subset optimization is obviously NP-hard. Nevertheless, even without the external constraints, the problem remains NP-hard, even with some further restrictions of the problem.

Theorem 2 *Given TCP preferences, the subset optimization problem is NP-hard, even if all item attributes are binary.*

Proof outline: by reduction from vertex cover, where each item is mapped to a graph vertex, and edges are non-zero attribute values.

Theorem 3 *Given TCP preferences, the subset optimization problem is NP-hard, even if the number of item attributes is bounded by a constant, and where all the properties are fully preferentially independent.*

Proof outline: by reduction from $3\mu3$-SAT (i.e. 3-SAT problems where each variable appears at most 3 times). Reduces to subset optimization where the items have 4 attributes, and no preferential dependency between properties.

Experimental Results

We evaluate the different algorithms using the movie dataset used by (Brafman *et al.* 2006) (from `imdb.com`).

Experiments were conducted using Pentium 3.4 GHz processor with 2GB memory running Java - 1.5 under Windows XP. Initial experiments quickly show that B&B in Set Space does not scale up. With just over 20 elements, it did not converge to an optimal solution within an hour even when the preference specification involved only 5 properties. This was true for both qualitative and quantitative preference specifications, for depth-first B&B, best-First B&B, and queue elements ordered based on upper-bound, lower-bound, and weighted combinations of them. Here is a sample of these results for preferences over nine set properties with qualitative preferences (i.e., TCP-based): DFS is much

Items	Method	Sets Generated	Sets Until Best	Time (sec)
8	BFS	4075	18	0.56
8	DFS	630	83	0.19
10	BFS	15048	40	2.34
10	DFS	2935	672	0.47
15	BFS	104504	7879	68.23
15	DFS	30547	11434	3.13
20	BFS	486079	28407	1584.67
20	DFS	231616	28407	28.578

Table 1: B&B in Sets Space

better than BFS, but the branching factor of larger databases overwhelms this approach. Various improvements may be possible, but given the much better performance of the other approach, they are unlikely to make a difference. It may be thought that with larger databases, it will be easy to quickly generate really good sets, but we found that for moderately larger (e.g,. 25+) and much larger (e.g., 3000) databases, this approach is too slow.

Next, we compared between improved B&B over CSPs without NoGoods (denoted B&B below), with (Brafman *et al.* 2006) (denoted BDSS06) on the (qualitative) dataset used in (Brafman *et al.* 2006) which includes three preference specifications with 5, 9 and 14 variables – and 4 database sizes. We added another specification based on 9 properties, and one more 14 variable-based specification (14-3) designed intentionally to cause many backtracks in the space of *set*-property assignments. "–" indicates that the run did not complete within four hours. Results below report time in seconds.

Properties	Method	400 vars	1000 vars	1600 vars	3200 vars
5	BDSS06	0.3	0.77	1.30	4.02
5	B&B	0.13	0.14	0.17	0.25
9	BDSS06	0.43	1.42	2.42	6.58
9	B&B	0.12	0.14	0.17	0.23
14-1	BDSS06	0.66	2.03	4.69	14.92
14-1	B&B	0.16	0.19	0.27	0.34
14-3	BDSS06	4113.48	–	–	–
14-3	B&B	81.03	5523	16643	–

Table 2: Search over CSPs: New vs. Old Algorithm with Qualitative Preferences

We conclude that Search-over-Sets, at least in its current form, cannot escape the effect of the large branching factor, while the improved Search-over-CSPs shows much better potential. When we can find the optimal solution with little backtrack in the space of CSPs, it is very effective for both qualitative and quantitative preferences. When we need to explore many different CSPs, performance degrades. On larger databases, such backtracks often indicate an inherent conflict between desirable properties that could be recognized and resolved off-line. Sharing information between the solutions of different CSPs might also help.

Next, we compared the improved B&B over CSPs without NoGoods with the same version that employs NoGood recording (denoted as B&B+NG). The $\frac{\text{B\&B+}}{\text{B\&B+NG}}$ rows show the speedup factor between B&B+ algorithm and B&B+NG algorithm. For the simple preference specifications, the overhead of maintaining NoGoods did not pay off, and the speed up was negative. However for the more complex problems which require more intense CSP solving, the use of NoGood recording proved to be very useful, letting us solve previously infeasible instances both for qualitative and quantitative specifications. We created additional 14-2 preference specification, based on 14-3 one, which has slightly less backtrack.

Properties	Method	400 vars	1000 vars	1600 vars	3200 vars
14-2	B&B	6.5	27.1	259	–
14-2	B&B+NG	4.7	18.4	76.3	210
14-2	$\frac{\text{B\&B}}{\text{B\&B+NG}}$	1.4	1.47	3.65	∞
14-3	B&B	81.03	5523	16643	–
14-3	B&B+NG	107.9	266.8	646.8	3013
14-3	$\frac{\text{B\&B}}{\text{B\&B+NG}}$	0.75	20.1	25.7	∞

Table 3: Search over CSPs: Employing NoGood recording

We can clearly see a significant improvement of B&B+ algorithm compared to (Brafman *et al.* 2006) and of B&B+NG over B&B+ without NoGood recording. What is not clear in these results however, is the donation of the incremental approach. In the Table 4 we present the results obtained from incremental vs. non-incremental versions of the algorithms.

Properties	Method	1000 vars	1600 vars	3200 vars
14-1	B&B-nonInc	0.4	1.09	0.78
14-1	B&B	0.19	0.27	0.34
14-1	B&B+NG-nonInc	0.57	1.06	0.95
14-1	B&B+NG	0.19	0.38	0.54
14-2	B&B-nonInc	27.1	27.86	–
14-2	B&B	27.1	25.9	–
14-2	B&B+NG-nonInc	19.42	77.08	230.25
14-2	B&B+NG	18.40	76.31	210.82
14-3	B&B-nonInc	5370	16306	–
14-3	B&B	5523	16643	–
14-3	B&B+NG-nonInc	269.9	646.8	333.5
14-3	B&B+NG	266.8	646.1	301.3

Table 4: Search over CSPs: Employing NoGood recording

It seems that incremental approach is useful in general, especially for the smaller problems which require not so many backtrack in the space of *set*-property assignments. But its contribution at least for the current state is not substantial. NoGood recording by itself for example seems to contribute much more to the efficiency of the optimization process. Moreover for the more complex problems, incremental version sometimes performs worse compared to non-incremental one. The overhead of maintaining and copying the partial solution in this cases just does not pay off.

Next, we tested the B&B over CSPs algorithm with the GAI utility function. Preferences were obtained by quantifying the qualitative preferences used for the earlier tests. With 5 and 9 set properties, the algorithm performs well. While with 14 set properties, performance degrades significantly and the improved algorithm without using NoGood recording can not solve instances with more than 1000 variables, while using NoGoods proves to significantly improve the optimization process.

Properties	Method	400 vars	1000 vars	1800 vars	3200 vars
5	B&B	0.24	0.14	0.17	0.26
9	B&B	0.16	0.25	0.28	0.41
14	B&B	38.91	2376.40	–	–
14	B&B+NG	19	160.53	494.98	1349.9
14	$\frac{\text{B\&B}}{\text{B\&B+NG}}$	2.03	14.8	∞	∞

Table 5: Search over CSPs: GAI Utility Functions

There are a number of reasons for this difference in performance. First, qualitative preferences can admit more than one optimal (non-dominated) solution, and so one such solution may be easier to find. Second, for TCP-based preferences there are variable orderings that ensure that the first solution found will be an optimal one. For GAI utilities, typically, after we generate the best solution we still have to explore the tree in order to *prove* that no better solution exists. This forces us to solve an exponential in the properties number of CSPs in the worst case. Finally, solutions to intermediate nodes in the space of set-property assignments which provide important guidance for search in the qualitative case provide much less information in the quantitative case.

Conclusion

We considered the problem of computing an optimal subset given a preference specification based on the methodology of (Brafman *et al.* 2006). Our main contributions in this paper are a considerably improved search-over-CSPs algorithm with and without use of NoGood recording, a new search-over-sets algorithm, more uniform and general treatment of different types of preference models, and hardness proofs for TCP preferences. Additionally, we provide an empirical analysis that illuminates the possibilities and pitfalls for both qualitative and quantitative preferences; showing what can be done efficiently today, and where we would benefit from additional exploitation of the special properties of this search problem.

It is an interesting question whether an efficient NoGood recording scheme that does not rely on static variable and value orderings exists. Intuitively such a scheme should exist since the CSPs can be efficiently encoded into SAT as a boolean CNF formulæ. And clause learning is a widely known learning technique in SAT solving. Moreover we have also seen that while the incremental approach usually improves the overall performance, its contribution is not substantial and what really improves the performance is better individual CSP solving. Adding to the above, recent advancements in the areas of Pseudo Boolean Constraints and SAT solving, it seems worthwhile to try solve the resulting CSPs using one of the available Pseudo Boolean solvers, or alternatively encode the resulting CSPs to CNF formulas and solve them using a SAT solver.

References

Bacchus, F., and Grove, A. 1995. Graphical models for preference and utility. In *UAI'95*, 3–10. San Francisco, CA: Morgan Kaufmann Publishers.

Bistarelli, S.; Fargier, H.; Montanari, U.; Rossi, F.; Schiex, T.; and Verfaillie, G. 1999. Semiring-based CSPs and valued CSPs: Frameworks, properties, and comparison. *Constraints* 4(3):275–316.

Boutilier, C.; Brafman, R.; Domshlak, C.; Hoos, H.; and Poole, D. 2004. CP-nets: A tool for representing and reasoning about conditional *ceteris paribus* preference statements. *JAIR* 21:135–191.

Brafman, R. I.; Domshlak, C.; Shimony, S. E.; and Silver, Y. 2006. Preferences over sets. In *AAAI*.

Brafman, R. I.; Domshlak, C.; and Shimony, S. E. 2006. On graphical modeling of preference and importance. *Journal of AI Research* 25:389–424.

desJardins, M., and Wagstaff, K. L. 2005. Dd-pref: A language for expressing preferences over subsets. In *AAAI'05*.

Fishburn, P. C. 1969. *Utility Theory for Decision Making*. John Wiley & Sons.