Recompiling Utility Functions in a Changing World

Michael McGeachie
Computer Science and AI Lab, MIT
32 Vassar St. 32-250
Cambridge, MA 02139
mmcgeach@csail.mit.edu

Abstract

Our previous work considers a method for building a numeric utility function out of qualitative *ceteris paribus* preferences, or preferences other things held equal. Dynamic domains embody changes in preferences. This can come in many forms. The preferences themselves may change, the variables over which preferences are expressed may change, or the forms of utility independence that hold in the domain may change. We consider the consequences for each type of change to our system, and conclude that inherent ambiguities in our representation allow for simpler handling of change than might otherwise be the case.

Decision Support and Preference

While classical decision theory can appear too limited or too rigorous to support decision-making in dynamic domains and under changing conditions, many decision-theoretic formalisms have great conceptual benefit. Researchers in artificial intelligence have made several formulations of qualitative decision theory (Wellman & Doyle 1991; Tan & Pearl 1994; Boutilier *et al.* 1999), applicable in domains where accurate probabilities and time consuming preference elicitation techniques are either unavailable or undesirable. Such qualitative preference formulations allow decision makers to make natural statements, representing user's intuitions about the decision space and easing the preference elicitation task.

Including qualitative preferences in decision support systems allows personalization. Decision support systems that aim to help more than one decision maker need a way to represent the differences between different users. The most direct method is to represent their preferences. By *preferences* we mean the desires, tastes, and priorities that a person has, and that furthermore differ from person to person, causing two different people faced with the same decision to make different choices.

Ceteris Paribus Preferences in Dynamic Domains

Our previous work includes a methodology for going from preferences over a domain to making decisions in that domain (McGeachie & Doyle 2004). Our framework takes a set of *ceteris paribus* preferences (other things being equal) and compiles a utility function. This approach has several properties amenable to functioning in a dynamic domain. Firstly, we allow some latitude in the way a user is able to specify preferences. These preferences my well be incomplete, may not refer to all variables in the domain, may be conditional or unconditional, and may exhibit both utility dependencies and utility independencies (McGeachie & Doyle 2002). Secondly, we can add variables to the domain without altering or invalidating the existing preferences. Removing variables is also easy, however the existing preferences referencing those variables must be truncated, potentially shifting their meaning.

In the following section, we provide basic background regarding the representation and semantics of *ceteris paribus* preferences. This exegesis is a shorter version of that appearing in (McGeachie & Doyle 2004). Readers familiar with that source or the general theory of *ceteris paribus* might skip the following section.

Representation of Preference

Doyle and Wellman (Wellman & Doyle 1991) have observed that qualitative representations of preferences are a succinct and reasonable approximation of at least one type of common human preferences. Doyle, Shoham, and Wellman (Doyle, Shoham, & Wellman 1991) present a theoretical formulation of human preferences of generalization in terms of *ceteris paribus* preferences, *i.e.,* all-else-equal preferences. *Ceteris paribus* relations express a preference over sets of possible worlds. We consider all possible worlds (or outcomes) to be describable by some (large) set of binary features F. Then each *ceteris paribus* rule specifies some features of outcomes, and a preference over them, while ignoring the remaining features. The specified features are instantiated to either true or false, while the ignored features are "fixed," or held constant. A *ceteris paribus* rule might be "we prefer programming tutors receiving an A in Software Engineering to tutors not receiving an A, other things being equal." In this example, we can imagine a universe of computer science tutors, each describable by some set of binary features F. Perhaps F = $\{$*Graduated, SoftwareEngineering_A, ComputerSystems_A, Cambridge_resident, Willing_to_work_on_Tuesdays*, $\ldots\}$. The preferences expressed above state that, for a particular computer science tutor, they are more desirable if they received an A in the Software Engineering course, all other features being equal.

We employ a restricted logical language $\mathcal{L}$, patterned after (Doyle, Shoham, & Wellman 1991) but using only the standard logical operators $\neg$ (negation) and $\wedge$ (conjunction) to construct finite sentences over a set of atoms $\mathcal{A}$.[1] Each atom $a \in \mathcal{A}$ corresponds to a feature $f \in F$, a space of binary features describing possible worlds. We write $f(a)$ for the feature corresponding to atom a. By $literals(\mathcal{A})$ we denote the atoms of $\mathcal{A}$ and their negations; $literals(\mathcal{A}) = \mathcal{A} \cup \{\neg a \mid a \in \mathcal{A}\}$. A complete consistent set of literals m is a *model*. That is, m is a model iff exactly one of a and $\neg a$ are in m, for all $a \in \mathcal{A}$. We use $\mathcal{M}$ for the set of all models of $\mathcal{L}$.

A model of $\mathcal{L}$ assigns truth values to all atoms of $\mathcal{L}$, and therefore to all formula in $\mathcal{L}$ and all features in F. We write $f_i(m)$ for the truth value assigned to feature f_i by model m. A model *satisfies* a sentence p of $\mathcal{L}$ if the truth values m assigns to the atoms of p make p true. We write $m \models p$ when m satisfies p. We define a *proposition* expressed by a sentence p, by $[p] = \{m \in \mathcal{M} \mid m \models p\}$.

A *preference order* is a complete preorder (reflexive and transitive relation) $\succsim$ over $\mathcal{M}$. When $m \succsim m'$, we say that m is *weakly preferred* to m'. If $m \succsim m'$ and $m' \not\succsim m$, we write $m \succ m'$ and say that m is *strictly preferred* to m'. If $m \succsim m'$ and $m' \succsim m$, then we say m is *indifferent* to m', written $m \sim m'$.

A statement of desire is an expression of *ceteris paribus* preferences. We write $p \trianglerighteq q$ when p is desired at least as much as q. This is the statement that p is desired over q exactly when any model making p true and q false is weakly preferred to any model making p false and q true, whenever the two models assign the same truth values to all atoms logically independent of p and of q.

Feature Vectors

We define the "feature vector representation" relative to an enumeration $\mathcal{V} = (f_1, \ldots, f_N)$ of $F(C)$.

We define the language $\mathcal{L}_{\mathrm{r}}(\mathcal{V})$ of feature vector rules in terms of a language $\mathcal{L}(\mathcal{V})$ of propositions over the ternary alphabet $\Gamma = \{0, 1, *\}$.

A statement in $\mathcal{L}(\mathcal{V})$ consists of a sequence of N letters drawn from the alphabet Γ, so that $\mathcal{L}(\mathcal{V})$ consists of words of length N over Γ. For example, if $\mathcal{V} = (f_1, f_2, f_3)$, we have $*10 \in \mathcal{L}(\mathcal{V})$. Given a statement $p \in \mathcal{L}(\mathcal{V})$ and a feature $f \in F(C)$, we write $f(p)$ for the value in Γ assigned to f in p. In particular, if $f = \mathcal{V}_i$, then $f(p) = p_i$.

A feature vector rule in $\mathcal{L}_{\mathrm{r}}(\mathcal{V})$ consists of a triple $p \succ q$ in which $p, q \in \mathcal{L}(\mathcal{V})$ have matching $*$ values. That is, $p \succ q$ is in $\mathcal{L}_{\mathrm{r}}(\mathcal{V})$ just in case $p_i = *$ if and only if $q_i = *$ for all $1 \leq i \leq N$. For example, if $\mathcal{V} = (f_1, f_2, f_3)$, $\mathcal{L}_{\mathrm{r}}(\mathcal{V})$ contains the expression $*10 \succ *00$ but not the expression $*10 \succ 0{*}0$. We refer to the statement in $\mathcal{L}(\mathcal{V})$ left of the $\succ$ symbol in a rule r as the left-hand side of r, and denote it $LHS(r)$. We define right-hand side $RHS(r)$ analogously. Thus $p = LHS(p \succ q)$ and $q = RHS(p \succ q)$.

We regard statements of $\mathcal{L}(\mathcal{V})$ containing no $*$ letters as *models* of $\mathcal{L}(\mathcal{V})$, and write $\mathcal{M}(\mathcal{V})$ to denote the set of all such models. We say a model m *satisfies* s, written $m \models s$, just in case m assigns the same truth value to each feature as s does for each non $*$ feature in s. That is, $m \models s$ iff $f(m) = f(s)$ for each $f \in F(C)$ such that $f(s) \neq *$. For example, 0011 satisfies both $*0{*}1$ and $00{*}{*}$.

We project models in $\mathcal{M}$ to models in $\mathcal{M}(\mathcal{V})$ by a mapping α :

Definition 0.1 (Model Projection) *The translation* $\alpha : \mathcal{M} \rightarrow \mathcal{M}(\mathcal{V})$ *is defined for each* $m \in \mathcal{M}$ *and* $f \in F(C)$ *by* $\alpha(m) = m'$, $m' \in \mathcal{M}(\mathcal{V})$. *For all* $f_i \in \mathcal{V}$,

- $f(\alpha(m)) = 1$ *if* $f \in m$
- $f(\alpha(m)) = 0$ *if* $\neg f \in m$

This projection induces an equivalence relation on $\mathcal{M}$, and we write $[m]$ to mean the set of models in $\mathcal{M}$ mapped to the same model in $\mathcal{M}(\mathcal{V})$ as m:

$$[m] = \{m' \in \mathcal{M} \mid \alpha(m') = \alpha(m)\} \tag{1}$$

The translation α specifies that m and m' must assign the same truth values to features that appear in $\mathcal{L}(\mathcal{V})$, but that on features not appearing therein, there is no restriction. When the feature vector of $\mathcal{L}(\mathcal{V})$ is the set of features F, there is a one-to-one correspondence of models in $\mathcal{L}(\mathcal{V})$ and $\mathcal{L}$.

We say that a pair of models (m, m') of $\mathcal{L}(\mathcal{V})$ *satisfies* a rule r in $\mathcal{L}_{\mathrm{r}}(\mathcal{V})$, and write $(m, m') \models r$, if m satisfies $LHS(r)$, m' satisfies $RHS(r)$, and m, m' have the same value for those features represented by $*$ in r, that is, $m_i = m'_i$ for each $1 \leq i \leq N$ such that $LHS(r)_i = *$. For example, $(100, 010) \models 10* \succ 01*$, but $(101, 010) \not\models 10* \succ 01*$.

The meaning $[r]$ of a rule r in $\mathcal{L}_{\mathrm{r}}(\mathcal{V})$ is the set of all preference orders $\succ$ over $\mathcal{M}$ such that for each $m, m' \in \mathcal{M}$, if $(\alpha(m), \alpha(m')) \models r$, then $m \succ m'$. The meaning of a set R of rules consists of the set of preference orders consistent with each rule in the set, that is, $[R] = \bigcap_{r \in R}[r]$. Thus a rule $**01 \succ **10$ represents four specific preferences

$$\begin{array}{c} 0001 \succ 0010 \\ 0101 \succ 0110 \\ 1001 \succ 1010 \\ 1101 \succ 1110 \end{array}$$

Note that this says nothing at all about the preference relationship between, *e.g.*, 0101 and 1010.

The *support features* of a statement p in $\mathcal{L}(\mathcal{V})$, written $s(p)$, are exactly those features in p that are assigned value either 0 or 1, which represent the least set of features needed to determine if a model of $\mathcal{L}(\mathcal{V})$ satisfies p. The support features of a rule r in $\mathcal{L}_{\mathrm{r}}(\mathcal{V})$, denoted $s(r)$, are the features in $s(LHS(r))$. The definition of $\mathcal{L}_{\mathrm{r}}(\mathcal{V})$ implies that $s(LHS(r))$ = $s(RHS(r))$.

We say that a pair of models (m, m') of $\mathcal{L}(\mathcal{V})$ *satisfies* a rule r in $\mathcal{L}_{\mathrm{r}}(\mathcal{V})$, and write $(m, m') \models r$, if m satisfies $LHS(r)$, m' satisfies $RHS(r)$, and m, m' have the same value for those features represented by $*$ in r, that is, $m_i = m'_i$ for each $1 \leq i \leq N$ such that $LHS(r)_i = *$. For example, $(100, 010) \models 10* \succ 01*$, but $(101, 010) \not\models 10* \succ 01*$.

[1]We disallow the operators $\vee, \rightarrow, \leftrightarrow$ in $\mathcal{L}$. Logical sentences using disjunction, implication, and equivalence can be translated into (possibly larger) equivalent logical sentences in $\mathcal{L}$.

The meaning $[r]$ of a rule r in $\mathcal{L}_{\mathrm{r}}(\mathcal{V})$ is the set of all preference orders $\succ$ over $\mathcal{M}$ such that for each $m, m' \in \mathcal{M}$, if $(\alpha(m), \alpha(m')) \models r$, then $m \succ m'$. The meaning of a set R of rules consists of the set of preference orders consistent with each rule in the set, that is, $[R] = \bigcap_{r \in R}[r]$. Thus a rule $**01 \succ **10$ represents four specific preferences

$$\begin{array}{c} 0001 \succ 0010 \\ 0101 \succ 0110 \\ 1001 \succ 1010 \\ 1101 \succ 1110 \end{array}$$

Note that this says nothing at all about the preference relationship between, *e.g.*, 0101 and 1010.

Building a Utility Function

Writing *ceteris paribus* preferences in a feature-vector formation makes the relationship between the preference and the preorder over outcomes more explicit. Each preference statement is shorthand for a specific group of preference relations $m \succ m'$ over outcomes. Thus, it is easy in principle to enumerate the particular relations implied by each preference statement. In previous work, we did this with what we called a *model graph*.

The model graph is a directed graph with $2^{|F(C)|}$ nodes, representing models, and edges, representing preference. Each edge from m to m' indicates a directly-expressed preference for m over m'. Thus, there exists an edge $e(m, m')$ if and only if $(m, m') \models r$ for some preference rule r expressed in the feature-vector representation. The graph is then constructed by considering each preference r and including edges indicated by r.

When the model graph is complete, we can determine if m is preferred to m' by looking for a path from m to m' and a path from m' to m. If the path from m to m' exists, we can conclude that $m \succ m'$ according to the expressed preferences. If the path from m' to m exists, we conclude the opposite, that $m' \succ m$. If both pathes exist, then we conclude that m is preferred to itself; which we consider to be an indication that the preferences themselves are inconsistent.

Further, such a graph can be used to assign utility values to models. For example, a function $u(m)$ that assigns to a node m the integer value of the number of distinct nodes reachable from pathes starting with m, is a utility function consistent with the set of preferences used to construct the model graph. This is an intuitively salient result; it relies on the fact that when $m \succ m'$ then m' is reachable from a path starting with m, and thus, $u(m) > u(m')$ since every node reachable from m' is also reachable from m, plus the node m' itself. We refer to such a function as a *Graphical Utility Function*. This function is *ordinal* in the sense that neither $u(m) - u(m')$ or $u(m)/u(m')$ have meaningful values; only the ordering of the values of u are important.

Utility Independence

For reasons of computational efficiency, it is desirable to have a utility function that is not computed directly from a graph defined on the entire feature space, but rather a utility function that is a linear combination of subutility functions, each computed from model graphs of subsets of the feature space. In particular we consider *generalized additively independent utility functions*, which are functions of the form

$$u(m) = \sum_i t_i u_i(m)$$

where each subutility function u_i is a function of only a subset of the features $F(C)$, and the features used in subutility functions cover the set $F(C)$. If the features used in subutility functions form a partition of $F(C)$ then this is a *additive utility decomposition* rather than a *generalized* additive decomposition. In either case, the computation of utility for a feature in u_i must be *utility independent* of features not included in the domain of u_i. This is the idea that a person's preferences for one feature are independent of the (fixed) values of another feature. See (Keeney & Raiffa 1976) for a discussion of various kinds of utility independence and properties thereof. In most cases assuming that the domains of subutility functions partition the space greatly simplifies presentation without changing results in a substantive manner. Thus finding a suitable partition of the features into subutility functions is an important task in the construction and design of a utility function.

Keeney and Raifa (1976) give criteria for when two features are utility independent. This is, broadly speaking, a powerful concept requiring significant constraints on the utility function and the decision maker. We consider it more expedient to assume that each feature is utility independent of each other feature and then look for evidence of utility dependence between features. This evidence is of a simple form: a preference on one feature reverses itself when the value of another feature changes. For example, if the weather is raining I may prefer carrying an umbrella with me to leaving it at home. However, when the value of the weather changes I immediately reverse my preference on the umbrella, and instead prefer to leave the umbrella at home in good weather than to carry it with me (Boutilier 1994). In this case the utility of carrying the umbrella depends on the value of the *weather* feature. It is not necessary for the converse to be true; the utility of the weather may be independent of the presence of my umbrella.

In the simplest case, such a preference reversal is easy to detect in groups of feature-vector preference rules. A preference reversal on feature 1, *umbrella*, in response to a change in the value of feature 2, *rain*, would look like this:

$$\begin{array}{c} 00 \succ 10 \\ 11 \succ 01. \end{array}$$

In this case the preference for *umbrella* is $0 \succ 1$ in the case of *rain* = 0 (shown in the first preference), and the preference for *umbrella* then changes to $1 \succ 0$ when *rain* changes to 1 (shown in the second preference). This preference reversal indicates that *umbrella* is utility dependent on *rain*. Using this feature-vector representation, it is a simple manner to perform string-parsing on pairs of preference rules to detect this type of utility dependence between features. There may still be utility dependencies that are not discovered via this method, for example, those resulting from the transitive closure of many such preferences. However, this problem can be fixed at later stages of the utility construction algorithm.

Subutility Functions and Local Inconsistency

Given a partition of the feature space into utility independent subsets, the next task is to construct subutility functions for each such subset of features. This proceeds in several steps. First a set of "restricted" preferences is generated for each subutility function. Then a graphical utility function is generated from these preferences for each subutility function. Linear programming is then used to solve for the scaling parameters of the full additive decomposition utility function.

Each subutility function u_i corresponds to a set of features S_i such that S_i is utility independent of $\overline{S_i}$. Each preference rule r is a statement over the features $F(C)$. A rule r can be projected onto the features of each subutility function by simply retaining the values for features in S_i and ignoring or deleting the values for other features. In this way, we can obtain a set R_i of feature-vector preferences over S_i. If the resulting preferences are acyclic on S_i, they can be used to define a graphical utility function over S_i, and this is then used as the subutility function u_i for features S_i. If the graph for each subutility function is acyclic, then we are free to set the scaling parameters t_i of the utility function to any value greater than zero we desire, in particular, 1. Thus, in the totally acyclic case, we have a compiled a completed utility function $u(m) = \sum_i u_i(m)$ where each u_i is a graphical function as described above.

However, when the preferences R_i are cyclic we develop a constraint satisfaction problem to break the cycles. We choose some preference rules in R_i and remove them, such that the remaining rules are acyclic. The removed rules are used as input to the linear programming problem in the next step of the algorithm, described below. We use constraint satisfaction solvers to determine which rules to remove in this way. The first constraint is that no rule can be removed from every subset R_i. And the second and final constraint is that each particular subset R_i must be cycle-free. By generating a satisfaction problem in conjunctive normal form, we can use a boolean satisfaction solver (SAT-Solver) to arrive at a solution to the problem. Results in (McGeachie & Doyle 2004) show that any solution to this problem, when it such a solution exists, is correct. Using the solution to this problem, we choose some preference rules from R_i, remove them, and build cycle-free graphical utility functions out of the remaining rules.

Preference rules that are removed in the previous step in order to obtain cycle-free subutility functions are given consideration in the linear programming step that determines scaling parameters for the additive decomposition utility function. Recall that our utility function is of the form $u(m) = \sum_i t_i u_i(m)$, or a weighted sum of subutility functions. The linear programming step assigns values to each scaling parameter t_i based on the preferences removed from R_i. For each such preference r, we add a linear inequality to a set of inequalities I for each value of $u_i(m) - u_i(m')$ for all $(m, m') \models r$. The inequality contributed is of the form:

$$\sum_i t_i(u_i(m) - u_i(m')) > 0.$$

In (McGeachie & Doyle 2004) we give more details about this process, including arguments of correctness.

From the set of linear inequalities I, we can solve for the values of t_i using linear programming. The solution gives us the final form of the utility function, $u(m) = \sum_i t_i u_i(m)$. We summarize the steps of the entire *Utility Function Construction* algorithm in Figure 1.

The algorithm outputs a utility function u consistent with a set of input *ceteris paribus* preferences C.

1. Compute the set of relevant features $F(C)$, which is the support of C.
2. Compute C^* from C by converting a set of preferences C in language $\mathcal{L}$ to a set of preferences C^* in the language $\mathcal{L}(\mathcal{V})$.
3. Compute a partition $S' = \{S'_1, S'_2, ..., S'_Q\}$ of $F(C)$ into utility-independent feature sets.
4. Construct projected rule sets R_i for each S_i by projecting each rule r onto each set S_i.
5. Construct the a model graph $G_i(R_i)$ for each set of rules R_i over the features S_i.
6. Compute a complete set Y_i of cycles Y_{ik} for each graph $G_i(R_i)$ such that each cycle Y_{ik} is a set of rules from R_i.
7. Construct a satisfiability problem $P(C^*, S)$ from all cycles Y_{ik}, indicating which rules participate in which cycles.
8. Find a solution Θ to $P(C^*, S)$.
9. Choose conflict-free rule sets $\overline{R}_i \subseteq R_i$ sets using solution Θ of $P(C^*, S)$.
10. Construct cycle-free model graphs $G'_i(\overline{R}_i)$
11. Define each subutility function u_i to be the graphical subutility function based on $G'_i(\overline{R}_i)$.
12. Construct a system of linear inequalities relating the parameters t_i, $I(C^*, S, \overline{R})$.
13. Solve $I(C^*, S, \overline{R})$ for each t_i using linear programming.
    (a) If $I(C^*, S, \overline{R})$ has a solution, pick a solution, and use the solution's values for u_i and t_i to construct and output a utility function $u(m) = \sum_i t_i u_i(m)$.
    (b) If $I(C^*, S, \overline{R})$ has no solution, construct u to be the graphical utility function based on $G(C^*)$, and output u.

Figure 1: Utility Construction Algorithm

Recompiling for Change

Performing preference elicitation and preference reasoning in a dynamic environment presents a significant challenge to our work. Ideally, we feel our system should adapt better to changing or additional preferences. Adding preferences to our formulation currently requires a recompile of our utility function, which can be computationally costly. Future work will investigate partial compilation methods and alternative representations with better incremental compilation properties.

However, our representation of preference is suited to two kinds of preference change: utility independence changes, and changes in the underlying feature space.

Utility Independence, Utility Dependence

Our representation of preference, while utilizing utility independence of features as a major computational expedient, makes no semantic or syntactic commitment to utility independence. This allows our representation to reason efficiently with preferences that exhibit substantial utility independence and reason less efficiently with inherently more complicated, utility dependent preferences, as the particular domain and problem require. Furthermore, a user's preferences may change in such a way that some features that were utility independent become dependent, and vice versa. Our system allows this shift without requiring specialized semantics or syntax.

Suppose we have a set C of *ceteris paribus* preferences. Then suppose some preferences are added to C, resulting in a set C'. Let S be the partition of $F(C)$ associated with the set of preferences C. S has been computed by starting with a set of singleton sets (for each feature f_i, there is a set $S_i = \{f_i\} \in S$) and then merging sets S_i and S_j when f_i is found to be utility dependent on feature f_j. Since C' is a superset of C, we know that the previous utility dependencies are still valid. We need only check if preferences in $C' \backslash C$ result in new utility dependencies. This can be done by our feature-vector parsing method described above. Once a new partition S' is computed, we must consider if any of the feature sets $s \in S'$ are the same as feature sets in the original partition. If so, we break feature sets into two groups, new and old. If the new feature sets are cycle-free, we can use the previous graphical utility functions for each of the old feature sets. The new feature sets will have to use new graphical subutility functions. The linear inequality set must be regenerated and solved again, providing new values of scaling parameters for the utility function. If the new feature sets have cycles, then the rest of the utility function construction algorithm can be re-run: we create new subutility functions for all feature sets, then resolve all cycles through constraint satisfaction, and finally set scaling parameters through linear programming.

Adding and Deleting Domain Variables

In a changing domain, the domain model itself is subject to change. We consider two instances of such change, first that a feature of the domain would be deleted, and second that a feature of the domain would be added.

Suppose we have a set of preferences C over a set of features $F(C)$. Then consider what happens when f_i is removed from the domain. We have a new set of features $F(C)'$. Let u_i be the subutility function who's domain contained f_i. If u_i contained no other features, then u_i will be removed from the utility function, otherwise it will need to have its graphical utility function reconstructed from a new set of projected preference rules R'_i. In either case, we must consider if the old subutility function u_i contributed to the set of linear inequalities. If it did, then the set must be regenerated and solved again, arriving at different values for the scaling parameters t_i. If not, then we need not perform the SAT-solving step for cycle-breaking, nor the linear programming step for scaling parameter setting. A subutility function contributes to the set of linear inequalities when either a) it is itself cyclic or b) preference rules that *overlap* with u_i are members of cycles on some other subutility function u_j. Here we define a preference rule *overlaps with a subutility function* u_j when it has either letters 0 or 1 for any feature f in the domain of u_j.

Similarly, we can add a new feature without performing an entire recompile of the utility function if the new feature does not contribute to the linear inequality step of the equation. Suppose we have a new feature f_{n+1} added to our set of features $F(C)$. We also assume that there are some new preference statements C' that refer to the feature f_{n+1}. If these new preferences are such that they do not overlap with f_{n+1} and participate in cycles on some other feature, and the graphical subutility function for f_{n+1} is cycle-free, then all previous graphical subutility functions can remain and we can skip the linear inequality step, using previous values for scaling parameters t_i and setting $t_{n+1} = 1$. If this is not the case, we must check that the new feature is not utility dependent on any other feature, or some other feature utility dependent upon it. We must then perform the satisfiability step again, build new graphical subutility functions for each feature set S_i, and generate and solve a new set of linear inequalities. We will then have a new utility function $u(m) = \sum_i t_i u_i(m)$.

Altered Preferences

In general, preference changes are a source of instability for our algorithm. If preferences change arbitrarily, then we have to repeat the work already done to build a utility function, and build a new one largely unrelated to the previous utility function. However, our criterion of changing preferences participating in cycles on some subutility function characterizes the cases where a complete recompile is not necessary. For the following, we say that a preference rule r *changes between* C *and* C' if it is either not in C or it is not in C'.

Theorem 1(Cycle Agnosticism) *For all preference rules r changing between C and C', if r is such that it is a member of no cycle on any subutility function u_i built from preferences C or any subutility function u'_i built from preferences C', then building utility function u' from C' does not require redoing the cycle-breaking constraint step or the linear inequalities scaling parameter step.*

This theorem says that we are free to keep the subutility functions and scaling parameter assignments of a utility function u for C when building u' for C', when C' obeys the proper conditions. This means that all that must be done is the generation of graphical subutility functions for each feature set S_i in the partition S. In general, this avoids the computationally inefficient parts of the algorithm; boolean satisfiability is famously NP-Complete, and although solving a system of linear inequalities is in P, in the worst case our system of linear inequalities can be of exponential size.

Conclusions

Our utility function construction algorithm requires more work before it is ready to tackle rapidly changing preferences in unstable domains. In general, we must rebuild our utility function when preferences change. However, we have identified an important case of preference changes that do not require complete rebuilds of our utility function. This achieves one of our long-standing goals: to have methods for identifying when partial recompilation is possible and how to proceed in those cases.

References

Boutilier, C.; Brafman, R. I.; Hoos, H. H.; and Poole, D. 1999. Reasoning with conditional ceteris paribus preference statements. In *Proceedings of Uncertainty in Artificial Intelligence 1999 (UAI-99)*.

Boutilier, C. 1994. Toward a logic for qualitative decision theory. In Doyle, J.; Sandewall, E.; and Torasso, P., eds., *KR94*. San Francisco: Morgan Kaufmann.

Doyle, J.; Shoham, Y.; and Wellman, M. P. 1991. A logic of relative desire (preliminary report). In Ras, Z., ed., *Proceedings of the Sixth International Symposium on Methodologies for Intelligent Systems*, Lecture Notes in Computer Science, 16–31. Berlin: Springer-Verlag.

Hansson, S. O. 1989. A new semantical approach to the logic of preference. *Erkenntnis* 31:1–42.

Keeney, R., and Raiffa, H. 1976. *Decisions with Multiple Objectives: Preferences and Value Tradeoffs*. New York: Wiley and Sons.

McGeachie, M., and Doyle, J. 2002. Efficient utility functions for ceteris paribus preferences. In *AAAI Eighteenth National Conference on Artificial Intelligence*.

McGeachie, M., and Doyle, J. 2004. Utility functions for ceteris paribus preferences. *Computational Intelligence: Special Issue on Preferences* 20(2):158–217.

Tan, S.-W., and Pearl, J. 1994. Qualitative decision theory. In *AAAI94*. Menlo Park, CA: AAAI Press.

Wellman, M., and Doyle, J. 1991. Preferential semantics for goals. In Dean, T., and McKeown, K., eds., *Proceedings of the Ninth National Conference on Artificial Intelligence*, 698–703. Menlo Park, California: AAAI Press.