# Product Recommendation Systems: A New Direction*

**Derek Bridge**
Department of Computer Science,
University College, Cork
Ireland
d.bridge@cs.ucc.ie

## Introduction

This paper is about *content-based product recommender systems*. In *product recommendation*, a customer is presented with a selection of products from a product catalogue. *Content-based* approaches (in contradistinction to, e.g., collaborative approaches) select products by matching product descriptions from the catalogue with descriptions of customer preferences and requirements.

We will refer to each product description as a *case*, $c$, and we will refer to the product catalogue as a *case base*, CB. We assume a set of *attributes*, $A$, and, for each $a \in A$, a *projection function*, $\pi_a$, which obtains a *value* for the attribute from the case. For example, $\pi_{price}(c)$ returns the value of case $c$'s price attribute.

This formulation, using projection functions, has the advantage of being agnostic about the actual underlying representation of the cases. They might, for example, be stored as tuples in a relational database, objects in an object-oriented database, or XML documents; all of these can support projection functions.

It also allows the possibility of what one might call *virtual attributes*, where the value returned is not directly stored but is, instead, computed or inferred from what is stored. This is useful, for example, when the case base stores only 'technical' data (e.g. a car's fuel-tank capacity, fuel consumption and top speed) but product selection requires 'lifestyle' attributes (e.g. the sportiness of the car). The projection functions for the lifestyle attributes would infer their values from the technical data.

The values returned by a projection function will be of some particular *type*. For example, for a holiday case base, $\pi_{transport}$ might have type $\{train, plane, car, coach\}$; $\pi_{season}$ might have type $\{Jan, Feb, \ldots, Dec\}$; $\pi_{price}$ might have some suitable set of numbers as its type.

To simplify this paper, we will draw a distinction at this point between *ordered types* and *unordered types*. We will say that an ordered type is one that has a non-trivial *partial order* of its values that may be useful in product recommendation. $\pi_{price}$ is an example: since its type is numeric, the values are ordered by the usual ordering of the numbers ($<$). We will call this its *base order*.

An unordered type, by contrast, will be one whose values have no non-trivial ordering that is relevant to product recommendation. Instead, we will assume that for unordered types there exists a relevant *similarity measure* on the values. $\pi_{transport}$ is a probable example.

This is obviously not a proper mathematical distinction; it is completely informal. It may not even be a distinction that is easily drawn in practice. For example, should $\pi_{season}$ be regarded as ordered ($Jan < Feb$, etc.) or not? Some types may fall into both categories (having both a relevant base ordering and a relevant similarity measure); some types may have more than one relevant base ordering or similarity measure. We need not concern ourselves with these problems with the definitions, as the distinction is made only to make the exposition in this paper simpler.

We will now look at two existing ways of building product recommendation systems (Filter-Based Retrieval and Similarity-Based Retrieval) before we present a new approach (Order-Based Retrieval).

## Filter-Based Retrieval

Product recommendation systems have been built in which customer preferences and requirements are encoded using what we will call *filters*. Filters are absolute: products either satisfy them or they do not. Only products that satisfy the filters are recommended to customers.

For example, predicates such as equality and inequality can be used to compare an attribute value in a case $c$ to a customer-provided value, e.g. $\pi_{transport}(c) \neq coach$. In the case of attributes whose types are ordered, we can additionally use $<$, $\leq$, $>$ and $\geq$, e.g. $\pi_{price}(c) < 900$. Local filters (on individual attributes) can be composed into a global filter (on whole cases), e.g. using conjunction.

The problems with Filter-Based Retrieval for product recommendation are well-documented [9]. In particular, the customer's filter may be satisfied by no products at all. Dialogues with such systems can exhibit a form of 'stonewalling'.

## Similarity-Based Retrieval

The CBR community has pioneered an alternative way of building product recommendation systems, using *similarity*

*This work was partly funded by Enterprise Ireland (grant number ST/98/024).

*measures*, $\sigma$. A customer supplies 'ideal' values for some or all of the attributes. The degree of similarity between attribute values in cases and these 'ideal' values can be computed by local similarity measures. The degree of similarity of cases in the case base to the 'ideal' case is computed by a global similarity measure which aggregates the local degrees of similarity. The cases with the highest degrees of similarity to the 'ideal' case are the ones to be recommended to the customer. (There are variants of this approach, the most notable being the one that is embodied in the Entrée system; we will discuss this in a later section of this paper.)

The well-recognised advantage over Filter-Based Retrieval is that the result-set will never be empty. Systems built this way can offer commercial benefits [2]; and their advantages are being confirmed by empirical experiments [10].

A disadvantage of using *'pure'* Similarity-Based Retrieval is that it allows the customer to supply only 'ideal' values. For example, the customer can specify a preferred price (to retrieve cases that have similar prices), but the customer cannot specify a maximum price. The customer can specify a preferred holiday season, but not a dispreferred one.

In the remaining sections of this paper, a novel approach to content-based product recommendation is presented. Similarity still has a role to play (though its role is diminished). The new approach allows customers to specify values other than 'ideal' ones, without resorting to the use of filters.

## Order-Based Retrieval

A key observation is that the success of Similarity-Based Retrieval rests on its ability to order the cases in the case base, rather than to filter them. Similarity is being used to obtain an ordering: if we call the customer's 'ideal' case $c$, then case $c_i \in \text{CB}$ is lower in the ordering than case $c_j \in \text{CB}$ iff $\sigma(c_i, c) < \sigma(c_j, c)$.

But, using the relative similarities of cases to an 'ideal' case is only one way of ordering a case base. In what we will call Order-Based Retrieval, product recommendation will be based on the application of *partial orders*: the customer will supply a variety of information (preferred values, dispreferred values, maximum values and minimum values, for example) and we will construct an ordering relation from this information; we can use this to sort the case base, or to obtain the maxima of the case base.

There is a connection here with utility theory. In utility theory, the basic notion is a preference relation, which captures an agent's preferences among states. This relation has to satisfy certain axioms. If it does, then there exists at least one corresponding utility function, i.e. a function that gives numeric scores to the states in a way which reflects the preferences. The advantage of having numeric scores is that these can be combined with state probabilities to compute expected utilities. With this, we can build decision-making agents that choose among (non-deterministic) actions based on the expected utilities of the states that may result from those actions.

It may be worth exploring the idea that the partial orders that we will use in Order-Based Retrieval are the preference relations used in utility theory. However, we will not explore this further here. It remains to be seen whether it is worth adopting the axiomatisation used for preference relations; and it remains to be seen whether casting product selection as decision-making between actions with multiple uncertain outcomes brings any advantages to simple web-based e-commerce systems.

## Operators for Order-Based Retrieval

In this section, we define a number of operators for constructing partial orders. In the way that we list them here, their usefulness may not be immediately apparent. But they are exemplified in subsequent sections of this paper. The reader should skim this section on first reading, and then refer back to it as necessary when working through the examples.

In these definitions, $x$, $y$ and, where appropriate, $v$ may be attribute-values or whole cases.

We begin with an operator that constructs a partial order from a filter, then one that constructs a partial order from a similarity measure, and then two more that construct a partial order from an existing partial order:

**Filter-Ordering** (FO)**:** Given a unary predicate, $p$, which would ordinarily act as a filter, we can construct an ordering from $p$ as follows:

$$x <_{\text{FO}(p)} y \mathrel{\hat{=}} \neg p(x) \wedge p(y)$$

The definitions says that $x$ is lower than $y$ iff $y$ satisfies $p$ but $x$ does not.

**Similarity-Ordering** (SO)**:** Given a similarity measure, $\sigma$, and an 'ideal' value, $v$, then $x$ is lower than $y$ iff $x$'s similarity to $v$ is lower than $y$'s similarity to $v$:

$$x <_{\text{SO}(\sigma,v)} y \mathrel{\hat{=}} \sigma(x, v) < \sigma(y, v)$$

**Inverse-Ordering** (IO)**:** Given a partial order, $<$, the inverse is also a partial order:

$$x <_{\text{IO}(<)} y \mathrel{\hat{=}} x > y$$

**About-Ordering** (AO)**:** Given a partial order, $<$, and an 'ideal' value $v$, then a new ordering can be defined by 'breaking the back' of the existing ordering at $v$. Values will be ordered by their distance from $v$ in the original ordering:

$$x <_{\text{AO}(<,v)} y \mathrel{\hat{=}} (x < y \leq v) \vee (x > y \geq v)$$

We now give two operators that compose two existing partial orders, $<_1$ and $<_2$, into a single partial order.

**Non-Contradiction-Ordering** (NCO)**:** For $x$ to be lower than $y$ in this compound ordering requires that it be lower in at least one of the two partial orders and not higher in the other:

$$x <_{\text{NCO}(<_1,<_2)} y \mathrel{\hat{=}} (x <_1 y \wedge x \not>_2 y) \vee (x <_2 y \wedge x \not>_1 y)$$

The consequences of this definition are: if $x$ is less than $y$ in both, then $x$ is less than $y$ in the result; if $x$ is less

than $y$ in one, but incomparable to $y$ in the other, then $x$ is less than $y$ in the result; if $x$ is less than $y$ in one but $x$ is greater than $y$ in the other, then $x$ and $y$ are incomparable in the result.

**Less-Strict-Prioritisation-Ordering** (LSPO)**:** Here, $<_1$ takes precedence over $<_2$. In effect, the ordering is based on $<_1$ but, when $<_1$ judges two values to be equal or incomparable, the 'tie' is broken using $<_2$:[1]

$$x <_{\mathrm{LSPO}(<_1,<_2)} y \mathrel{\hat{=}} x <_1 y \vee ((x \not<_1 y \wedge y \not<_1 x) \wedge x <_2 y)$$

Many more operators could be given, but these form a good starting point for building Order-Based Retrieval product recommendation systems, as we will now illustrate.[2]

## Finding a Holiday

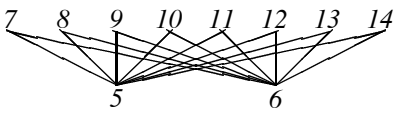
The examples we use to illustrate our operators are mostly based on the holiday case base that is available from the AI-CBR web site [1]. Our examples use the following attributes from this case base: *duration*, *accomm*, *price* and *season*. We do, however, pretend that the cases have an extra attribute: hotel *facilities* (e.g. whether the hotel takes credit cards, has a swimming pool, etc.).

### The right holiday duration ...

To keep our diagrams manageable, we will assume that the type of $\pi_{duration}$ is a small set: $\{5, 6, \ldots, 13, 14\}$. This is what we called earlier an ordered type: there is a relevant and non-trivial base ordering, viz: $5 < 6 < \ldots < 13 < 14$.

Suppose that our customer specifies that s/he wants a holiday of 7 or more days. We can define a unary predicate that expresses this: $\lambda x[x \geq 7]$. But we do not want to use this as a filter, as this could give us the problems associated with Filter-Based Retrieval. Instead, we can construct an ordering from this predicate using the Filter-Ordering (FO) operator:

**Example 1** *We want a holiday lasting no fewer than 7 days* $<_{\mathrm{FO}(\lambda x[x \geq 7])}$



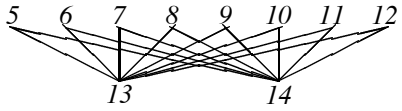

[1] Some readers may be wondering why this operator is called Less-Strict-Prioritisation-Ordering. In earlier work [8], we introduced an operator very similar to this one, which we later came to call Strict Prioritisation [4]. In Strict Prioritisation, a 'tie' is defined as equality. In Less-Strict-Prioritisation, a 'tie' is defined as equality or incomparability: it is less strict in its definition!

[2] Some readers may note similarities between this paper and one we published in 1996 [7]. One or two of the diagrams in the next section, in particular, may be reminiscent of the earlier paper. There are connections, but also differences. The goal of the 1996 paper was the definition of similarity measures, whereas here partial orders are primary. And, while Inverse-Ordering and About-Ordering (called *best* in [7]) are common to both papers, there are new operators herein.

If we were to sort the case base using this relation, holidays of 7 or more days' duration (if any) would be ranked above other holidays. Crucially, however, holidays lasting fewer than 7 days are not filtered away; they are simply dispreferred.

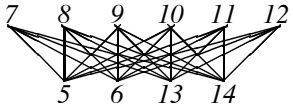
Suppose the customer must not take a holiday of more than 12 days. Again, we would apply the FO operator, this time to the predicate $\lambda x[x \leq 12]$:

**Example 2** *We can't take more than 12 days' holiday* $<_{\mathrm{FO}(\lambda x[x \leq 12])}$



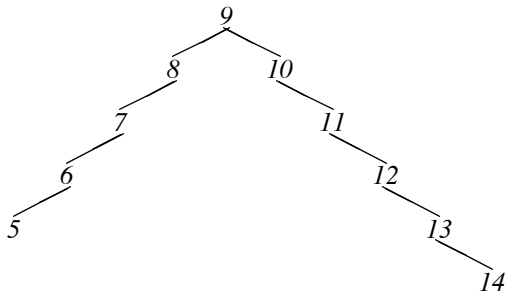

We can combine these two orderings into a single ordering relation using the Non-Contradiction-Ordering operator:

**Example 3** *No fewer than 7 days and no more than 12 days* $<_{\mathrm{NCO}(<_{\mathrm{FO}(\lambda x[x \geq 7])}, <_{\mathrm{FO}(\lambda x[x \leq 12])})}$



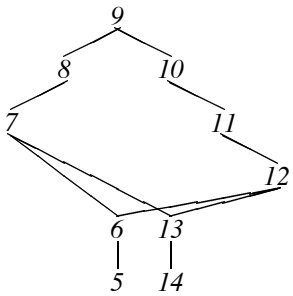

This might be the ordering the customer chooses to apply to the case base. But let us take the example one stage further. Suppose the customer goes on to specify that s/he would prefer a holiday of around 9 days. We can apply the Around-Ordering operator to $\pi_{duration}$'s base ordering:

**Example 4** *We'd prefer it to last about 9 days,* $<_{\mathrm{AO}(<,9)}$




We can now combine (3) and (4) into a single ordering using the Less-Strict-Prioritisation-Ordering operator:

**Example 5** *No fewer than 7 days, no more than 12, but then around 9*
$\leq_{\mathrm{LSPO}(<_{\mathrm{NCO}(<_{\mathrm{FO}(\lambda x[x \geq 7])}, <_{\mathrm{FO}(\lambda x[x \leq 12])})}, <_{\mathrm{AO}(<,9)})}$



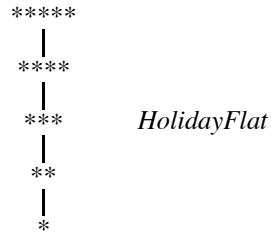
This exactly captures the customer's stated preferences: holidays lasting fewer than 7 or more than 12 days are lower in the ordering, but otherwise the values are ordered by their distance from 9 days.

This example also illustrates a common 'template'. A typical query is often constructed in this way. The 'constraints' that other approaches would consider to be filters, we convert to orders using FO. We combine these using NCO. And then we use LSPO to prioritise this over some other ordering. Here, the other ordering was the 'around 9' ordering, but it could just as well have been the base ordering or the inverse of the base ordering, as we will see in subsequent examples.

## The right accommodation ...

The type of $\pi_{accomm}$ is an ordered type. In its base ordering, hotel categories (star ratings) are ordered in the obvious way; Holiday Flat accommodation is incomparable (neither less than nor greater than) these hotel categories:

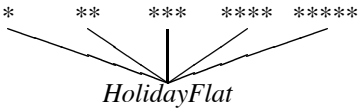
**Example 6** *The base ordering of* $\pi_{\text{accomm}}, <_{\text{accomm}}$




Suppose our customer does not want to stay in a Holiday Flat (self-catering is too much effort):

**Example 7** *We don't want a holiday flat*

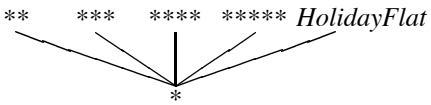
$<_{\text{FO}(\lambda x[x \neq \text{HolidayFlat}])}$




S/he will not stay in a one-star hotel (too basic):

**Example 8** *We don't want a one-star hotel*

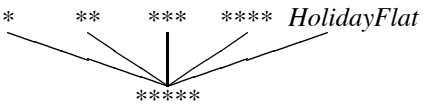
$<_{\text{FO}(\lambda x[x \neq *])}$




Nor will s/he stay in a 5-star hotel (too luxurious!):

**Example 9** *We don't want a 5-star hotel*

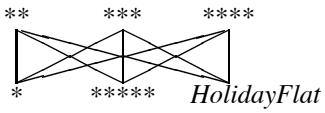
$<_{\text{FO}(\lambda x[x \neq *****])}$




We can combine (7), (8) and (9) using NCO:

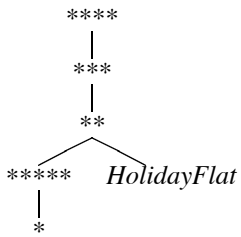
**Example 10** *Not a flat and not a one-star nor a five-star hotel*

$<_{\text{NCO}(<_{\text{FO}(\lambda x[x \neq \text{HolidayFlat}])}, <_{\text{FO}(\lambda x[x \neq *])}, <_{\text{FO}(\lambda x[x \neq *****])})}$




Finally, within these 'constraints', our customer would otherwise like the best accommodation available. The relevant ordering here is, in fact, the base ordering (6). So, we combine (10) with (6) using LSPO:

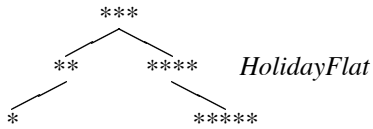
**Example 11** *Not a flat, not a one-star nor a five-star hotel, but otherwise as good as possible*




Suppose instead our customer wanted something around 3-stars (rather than the best available):

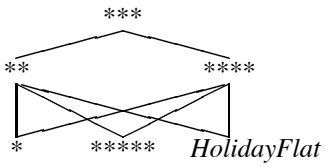
**Example 12** *We want something around 3-stars*

$<_{\text{AO}(<_{\text{accomm}}, ***)}$




We would then use LSPO to combine (10) with (12):

**Example 13** *Not a flat, not a one-star nor a five-star hotel, but otherwise something like a three-star hotel*



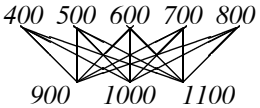

## The right price ...

To keep our diagrams manageable, we will assume that the type of $\pi_{price}$ is a small set: $\{400, 500, 600, 700, 800, 900, 1000, 1100\}$. Again, this is an ordered type: $400 < 500 < \ldots < 1000 < 1100$.

Suppose that our customer specifies that s/he wants to spend less than $900:

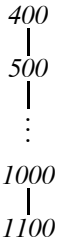
**Example 14** *We want to spend less than $900*

$<_{\text{FO}(\lambda x[x<900])}$



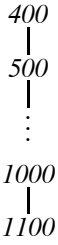

Suppose s/he then goes on to say that she additionally wants the cheapest holiday possible. The base ordering for $\pi_{price}$ orders holidays by increasing cost. To order holidays cheapest-first requires that we apply Inverse-Ordering to the base ordering:

**Example 15** *We want it to be as cheap as possible,* $<_{\mathrm{IO}(<)}$




Now we will see what happens when we use LSPO to combine (14) and (15):

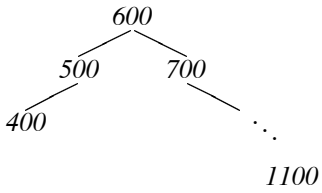
**Example 16** *Less than $900 and otherwise as cheap as possible,* $<_{\mathrm{LSPO}(<_{\mathrm{FO}(\lambda x[x<900])},<_{\mathrm{IO}(<)})}$




It turns out, in this example, that the customer 'constraint' that the holiday cost less than $900 is redundant in the light of the preference for the cheapest holiday.

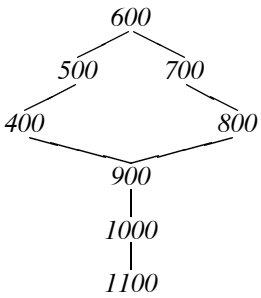
On the other hand, if the customer had said that s/he wanted a holiday costing around $600, we would have used AO:

**Example 17** *We'd expect to pay around $600,* $<_{\mathrm{AO}(<,600)}$




And we would then have used LSPO to combine (14) and (17):

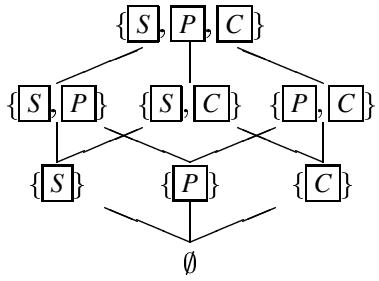
**Example 18** *Less than $900 and otherwise around $600*
$<_{\mathrm{LSPO}(<_{\mathrm{FO}(\lambda x[x<900])},<_{\mathrm{AO}(<,600)})}$




## The right hotel facilities . . .

We will now give an example using an attribute that does not appear in the original holiday case base [1] but which shows how we would handle set-valued attributes. Different hotels have different facilities, usually shown in brochures using collections of little icons. We will use only three icons: S means the hotel has a swimming pool, P means the hotel has parking and C means the hotel accepts credit cards. The type of $\pi_{facilities}$ is therefore $\wp\{\mathrm{S},\mathrm{P},\mathrm{C}\}$, i.e. subsets of the set $\{\mathrm{S},\mathrm{P},\mathrm{C}\}$. This is an ordered type:

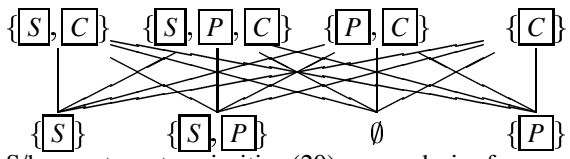
**Example 19** *The base ordering of* $\pi_{\mathrm{facilities}}$, $<_{\mathrm{facilities}}$




(A slightly different example could be given in which sets are ordered by their cardinalities rather than by set inclusion. In this case, the order would be a total one.)

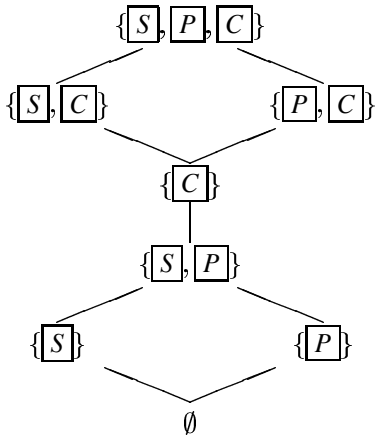
Our customer wants a hotel that accepts credit cards:

**Example 20** *We want to pay by credit card*
$<_{\mathrm{FO}(\lambda S[\mathrm{C}\in S])}$




S/he wants us to prioritise (20) over a desire for as many facilities as possible (19) (the base ordering):

**Example 21** *We want to pay by credit card but otherwise as many facilities as possible*
$<_{\mathrm{LSPO}(<_{\mathrm{FO}(\lambda S[\mathrm{C}\in S])},<_{\mathrm{facilities}})}$



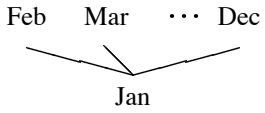

## The right season . . .

All the examples so far have involved attributes that have ordered types. Therefore, the base ordering usually plays a role somewhere. But we will now look at an example using an unordered type, where there is no base ordering (or, at

least, none that we shall make use of). In the case of the *season* attribute, we will assume that we have a similarity measure on months of the year, instead of a base ordering.

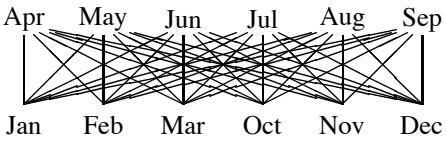
First, assume the customer does not want to holiday in January:

**Example 22** *We don't want to go in January*
$<_{\mathrm{FO}(\lambda x[x \neq \mathrm{Jan}])}$




S/he also does not want to holiday in *Feb*, *Mar*, *Oct*, *Nov* or *Dec* (individual diagrams not shown). We put all these together using NCO:

**Example 23** *Not in* Jan-Mar *nor in* Oct-Dec




But our customer would like to holiday in a month such as August. We will assume that our similarity measure, $\sigma_{season}$, when it compares the different months to August, returns the following degrees of similarity:

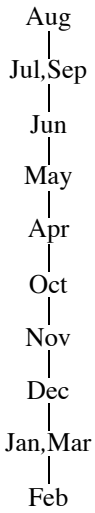
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| *Jan* | 0.17 | *Apr* | 0.33 | *Jul* | 0.83 | *Oct* | 0.67 |
| *Feb* | 0.0 | *May* | 0.5 | *Aug* | 1.0 | *Nov* | 0.5 |
| *Mar* | 0.17 | *Jun* | 0.67 | *Sep* | 0.83 | *Dec* | 0.33 |

This gives us an ordering:

**Example 24** *Months like August,* $<_{\mathrm{SO}(\sigma_{\mathrm{season}},\mathrm{Aug})}$


We can then use LSPO to combine the dispreferred months (23) with months like August (24):

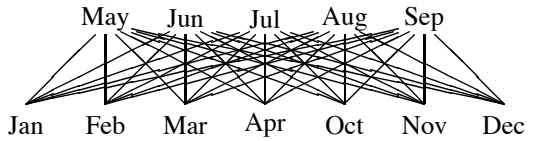
**Example 25** *Not* Jan-Mar *nor* Oct-Dec *but otherwise something like August*




By this point in the paper, we have shown lots of interesting examples. There is no shortage of other, quite innovative possibilities. For example, suppose the customer prefers months like August to months like December. How would we express this as an ordering?

It is actually done quite simply. First, we construct a filter, but one which makes use of $\sigma_{season}$. This filter takes in a month, $x$, and iff $x$'s similarity to August is greater than its similarity to December, then $x$ satisfies the filter. Then, we convert the filter to an ordering using FO:

**Example 26** *We prefer months like August to months like December*
$<_{\mathrm{FO}(\lambda x[\sigma_{\mathrm{season}}(x,\mathrm{Aug}) > \sigma_{\mathrm{season}}(x,\mathrm{Dec})])}$




Rather than give any more examples of this kind, we will now move on to other matters.

## Putting them together ...

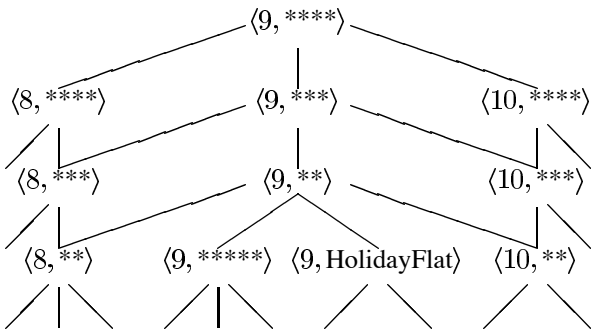
Our examples have, so far, all been attribute-specific (although, of course, any of them can be used to order whole cases, rather than just attribute values). We must now consider what we would do when orders are specified on two or more different attributes. In fact, the NCO and LSPO operators are useful here too. We will proceed again by example.

Suppose our customer wants a holiday of no fewer than 7 days and no more than 12 days. This would be ordering (3) but, rather than being an ordering simply on integers, it would be used to order whole cases. S/he wants a holiday lasting around 9 days. This would be ordering (4), again applied to whole cases. S/he also doesn't want to stay in a flat, a one-star hotel or a five-star hotel. This would be ordering (10) applied to whole cases. And s/he wants the best accommodation available. This would be ordering (6) applied to whole cases.

Our customer might then combine (3) and (4) using LSPO. S/he might also use LSPO to combine (10) with (6).

Finally, these two orderings can be combined using NCO. Part of the resulting order is shown below:

**Example 27** *We want a holiday lasting no fewer than 7 days, no more than 12 days, but otherwise lasting about 9 days, and we want to stay in neither a flat, a one-star hotel nor a five-star hotel but otherwise we want the best accommodation possible*




Example (27) has demonstrated that one way of combining orders on different attributes is to use our NCO and LSPO operators. Different applications of NCO and LSPO may give different results. (In some cases, however, the same ordering might result. It is a matter of future work to develop and prove algebraic equivalences using these operators.) Rather than demonstrate this any further, let us now look at another couple of ways of combining orders on different attributes.

In Similarity-Based Retrieval, overall degrees of similarity are computed by a global similarity measure, typically by taking a weighted average of local, attribute-specific degrees of similarity. The weights allow some indication of the relative importances of the different attributes.

There is, of course, nothing to prevent the use of weighted averages in Order-Based Retrieval. If we have several attributes with unordered types (such as *season*), we can combine their similarity measures into a global similarity measure prior to using the SO operator.

This gives us good compatibility with existing technology. Whether we should exploit this compatibility is a more complicated question. (In [5], we have argued that the exact effects of weighting schemes are not always obvious; customers may not find it easy to achieve definite effects by changing weights. Other approaches, including the one embodied in the Entrée system, may be more suitable.)

To conclude this section, we will mention one more, interesting possibility. In Order-Based Retrieval, we might be able to express *dependencies* between the values of different attributes. A simple example, which requires no new operators, is to construct a filter using material implication ($\Rightarrow$), and to then use FO to construct an order from this filter. For example, a customer might say that if s/he must stay in a holiday flat, then s/he expects the holiday price to be less than \$700. The filter

$$\lambda x[(\pi_{accomm}(x) = HolidayFlat) \Rightarrow (\pi_{price}(x) < 700)]$$

is satisfied by cases in which either the accommodation is a holiday flat and the price is less than \$700, or by cases in which the accommodation is not a holiday flat. We can construct an order from this using FO (in the way that is amply illustrated elsewhere in this paper).

It is our intention to investigate other ways of constructing orders that capture dependencies, perhaps making further use of connectives such as material implication.

## Entrée-Style Retrieval

The way most content-based product recommendation systems work is as follows. The customer supplies some preferences and requirements by filling in an on-screen form. On the basis of the values supplied, the system retrieves and displays one or more product descriptions. It will often be the case that the customer will not be immediately satisfied with the displayed products (or there may be some other reason why s/he wishes to see further products). In most systems, the customer must return to the on-screen form, and alter the data s/he entered. If the customer is trying to achieve a definite effect (e.g. to see cheaper holidays), this can be a cumbersome form of query refinement.

The Entrée system (and systems based on it) offer a more natural form of query refinement [6]. These systems allow the customer to select a product that is displayed on the screen and then launch queries that seek products that are similar to the selected product but which differ from it based on customer critiques: they seek products that are *"like this but different"*. The critique is sometimes also called a tweak.

We will examine, in more detail, how tweaks are implemented in Entrée. The kind of tweaks that a customer can request, and the way they are implemented, depend on whether the attribute that is being critiqued has an unordered type or an ordered type.

Suppose the critiqued attribute $a_t$ has an *unordered type*, e.g. *season*. In this case, a substitution is made. The current value of $a_t$, call it $v_t$, in the selected case, $c$, is replaced by a value, $v_t'$, that is provided by the customer. For example, suppose the customer has taken a liking to a particular holiday $c$ that is being displayed on the screen, but in this holiday *season* = *Apr* and s/he would prefer *season* = *Jul*. The value *Apr* is replaced by *Jul* in $c$ to give $c'$. Similarity-Based Retrieval is then used to find cases from the case base that are similar to $c'$.

Suppose, on the other hand, that in case $c$ the critiqued attribute $a_t$ has an *ordered type*, e.g. *price*. Again, let the current value of $a_t$ be $v_t$. The customer can tweak case $c$ by asking to see cases in which $a_t > v_t$ or cases in which $a_t < v_t$. For example, if the holiday s/he has taken a liking to, $c$, costs \$1000, s/he might ask to see holidays that are cheaper than $c$. Entrée takes such tweaks to be filters. Only cheaper holidays are sought. Within the cheaper holidays, Similarity-Based Retrieval is used to find those that are most similar to $c$.

This way of implementing tweaks is open to two criticisms:

- Tweaks on ordered attributes are treated as filters. This can give the usual problems associated with Filter-Based Retrieval.

- The two kinds of tweaks have different semantics. For unordered types, the tweaks are interpreted as a way of ordering cases (using similarity); for ordered types, the tweaks are absolute filters on cases.

Robin Burke writes " ... in the interest of uniformity, our newer implementation ... treats all tweaks as filters ... " [3]. While this newer implementation overcomes the second criticism, it makes the first criticism all the more pertinent.

We can implement tweaks in Order-Based Retrieval, and we can do so in a way that overcomes both criticisms above. Very simply, both tweaks are encoded as filters but then converted into orders using Filter-Ordering, FO. For example, in the case of *season*, the tweak becomes $<_{\mathrm{FO}(\lambda x[x=\mathit{Jul}])}$, which will rank holidays in July above holidays in other months. And, in the case of *price*, the tweak becomes $<_{\mathrm{FO}(\lambda x[x<1000])}$, which will rank holidays costing less than $1000 above holidays costing $1000 or more. This gives a uniform semantics and in neither case is the semantics absolute. These tweaks can then be composed into the rest of the query. (In Order-Based Retrieval, the rest of the query would apply SO and AO to the values of the non-tweaked attributes.)

Order-Based Retrieval might even allow new kinds of tweaks. For example, suppose the system has retrieved some cases and presented them to the customer. The customer might be allowed to state a simple preference between two cases (or two values). E.g. s/he might say that she prefers case 3 to case 7. From just this simple preference, we can construct a new ordering that can be composed into the query. The simplest new ordering is the one in which $c_7 < c_3$ and all other pairs of cases are incomparable. A more sophisticated version would capture the idea that cases that are more similar to $c_3$ than they are to $c_7$ are preferred. This would be done in a way that is similar to example (26).

## Conclusions

This paper proposes a change of perspective. It points out that Similarity-Based Retrieval in content-based product recommendation systems is really about ordering the case base. And so it suggests that we should be prepared to adopt any number of other ways of allowing the customer to construct orders on attribute-values and on cases.

The paper provides only a starting point in such a venture by proposing six operators for constructing orders. There are numerous ways of proceeding from here.

Firstly, there needs to be a more systematic exploration of ways of combining orders. In this paper, we have looked at a typical way of constructing a single ordering from a number of more basic ones. We need to find out which ways give the best results in practice. And we need to state and prove some equivalences and non-equivalences (for example, if $P$ and $Q$ are filters, then $<_{\mathrm{FO}(P \wedge Q)}$ is not necessarily the same as $<_{\mathrm{NCO}(<_{\mathrm{FO}(P)}, <_{\mathrm{FO}(Q)})}$, because NCO is not the same as the conjunction of two orders). Example (16) illustrates that we should also look for subsumption relationships and, perhaps, other relationships.

Secondly, we note that Order-Based Retrieval seems to offer a rich set of ways of constructing queries, but this richness can only be properly exploited if we can make it meaningfully available through our human-computer interface. This has to be the subject of further investigation.

Finally, we need to investigate implementations, especially in terms of their performance on large data sets and 'typical' customer requests. (In our current implementation, for example, we compute the maxima of an ordering applied to a case base of size $n$ using an algorithm whose worst-case time-complexity is $O(n^2)$. By contrast, its best-case time-complexity is linear; its performance on what we think might be 'typical' customer requests is often closer to the best-case than to the worst-case.)

## References


[1] AI-CBR: *Travel Agents' Case Base*, The AI-CBR Case Base Archive, `www.ai-cbr.org`

[2] Bergmann,R., Breen,S., Göker,M., Manago,M. & Wess,S.: *Developing Industrial Case-Based Reasoning Applications: The INRECA Methodology*, Springer, 1999

[3] Burke,R.: *Personal communication*, 2000

[4] Ferguson,A. & Bridge,D.G.: Generalised Prioritisation: A New Way of Combining Similarity Metrics, in Bridge,D.G. et al. (eds.), *Procs. of the 10th Irish Conference on Artificial Intelligence & Cognitive Science (AICS'99)*, pp.137–142, 1999

[5] Ferguson,A. & Bridge,D.G.: Partial Orders and Indifference Relations: Being Purposefully Vague in Case-Based Retrieval, in Blanzieri,E. & Portinale,L. (eds.), *Advances in Case-Based Reasoning (Procs. of the 5th European Workshop on Case-Based Reasoning)*, LNAI 1898, pp.74–85, Springer, 2000

[6] Hammond,K.J., Burke,R. & Schmitt,K.: A Case Based Approach to Knowledge Navigation, in Leake,D.B. (ed.), *Case-Based Reasoning — Experiences, Lessons and Future Directions*, pp.125–136, MIT Press, 1996

[7] Osborne,H.R. & Bridge,D.G.: A Case Base Similarity Framework, in Smith,I. & Faltings,B. (eds.), *Advances in Case-Based Reasoning (Procs. of the 3rd European Workshop on Case-Based Reasoning)*, LNAI 1168, pp.309–323, Springer, 1996

[8] Osborne,H.R. & Bridge,D.G.: Similarity Metrics: A Formal Unification of Cardinal and Non-Cardinal Similarity Measures, in Leake,D.B. & Plaza,E. (eds.), *Case-Based Reasoning Research and Development (Procs. of the 2nd International Conference on Case-Based Reasoning)*, LNAI 1266, pp.235–244, Springer, 1997

[9] Vollrath,I., Wilke,W. & Bergmann,R.: Case-Based Reasoning Support for Online Catalog Sales, *IEEE Internet Computing*, vol.2(4), pp.45–54, 1998

[10] Weibelzahl,S., Bergmann,R. & Weber,G.: Towards an Empirical Evaluation of CBR Approaches for Product Recommendation - In Electronic Shops, in *Procs. of the 8th German Workshop on Case-Based Reasoning*, `wwwagr.informatik.uni-kl.de/~gwcbr2`, 2000