# Preference-based Treatment of Empty Result Sets in Product Finders and Knowledge-based Recommenders

Dietmar Jannach

Institute for Business Informatics & Application Systems
University Klagenfurt, A-9020 Klagenfurt, Austria
dietmar.jannach@ifit.uni-klu.ac.at

**Abstract.** "Your search returned 0 results" is an undesirable message for customers using an online product-finder or a web-based sales advisory system. Such a situation typically occurs when a filter-based recommender system is used and the user's requirements are unrealistic or inconsistent. In this paper, we present two orthogonal techniques for dealing with such situations, whereby the work is based on a general model of this class of recommendation systems.
First, an algorithm for priority-based filter relaxation is presented where the end user can actively participate in the problem resolution process by specifying the importance of the predefined advisory rules that led to the empty result.
Second, we adapt an algorithm from the field of model-based diagnosis and repair in order to compute a set of action alternatives for the customer. Each of these alternatives corresponds to a possible (small) revision of the originally inconsistent requirements, such that a product proposal can be made.
The paper finally describes practical results from experiences in different real-world application domains and discusses domain-specific heuristics and techniques for further search time improvements for complex problems and multi-user environments.

## Introduction

Due to the huge variety of available products and services, many companies run product finders or more intelligent systems like recommender systems or sales advisory systems on their corporate web-sites. Based on the users' inputs, these systems filter out those products that fulfil the given specifications or match the needs and preferences of the customer. In particular for technical domains, where recommendations are not based on the concepts of "quality" and "taste", knowledge-based recommender systems have their specific advantages compared to other prominent approaches based on, e.g., collaborative filtering. Once the knowledge of the expert is made explicit and encoded in a knowledge base, the recommender system will behave like an experienced sales assistant and the quality of the recommendation will be constantly high, even if there are new products or new users involved. Even more, when adopting a knowledge-based approach, such systems will also be able to "explain" their recommendations to the customer. One of the main criticisms of such filter-based approaches [1] is that the undesirable situation can arise where all products are filtered out and no proposal can be made. “Your search returned 0 results” is the only response of many online systems in such situations. An experienced sales assistant, however, would explain to his customer why there are no results, i.e., which advisory guidelines and rules he did obey. Even more, he could inform the customer what he can do about the situation, for instance reconsidering the potentially conflicting requirements, or he could even propose alternative solutions that satisfy most of the given customer requirements.

In this paper, we present two orthogonal techniques for dealing with such situations. First, we show how filter-relaxation based on priorities can be exploited in order to take the different importance levels of the given advisory guidelines into account and help the user in understanding the proposals. Second, we describe an algorithm for computing a suitable set of alternatives that tries to minimize the differences between the original requirements and other slightly changed requirements such that a solution is possible and respects the strict time restrictions of online recommendation systems.

## Example and definitions

For demonstration purposes, we shall use a small example from the domain of digital cameras, a typical problem area where hundreds of different products are available and where online sales advisory systems are already used to assist the customer in finding the right product. We use the following very general model of a filter-based recommender system. Each *product* in the knowledge base is characterized by a set of attributes, whereby in many domains also set-valued attributes must be supported. Next, there is a set of *variables* that correspond to the user preferences, e.g., direct inputs or indirectly computed characteristics. Finally, a set of *filtering rules* describes the relation between the customer characteristics and the fitting product properties.

Our digital cameras are described by the property set *P*, whereby P = {weight, price, interfaces} and the following products exist in the product knowledge-base

```
PKB = {
   {id(p1). weight(100). price(80). interface(usb)}
   {id(p2). weight(100). price(150). interfaces(usb). interfaces(firewire).}
   {id(p3). weight(200). price(200). interfaces(usb). interfaces(firewire).
    interfaces(dockingstation).}
}
```

During the advisory session the customer can specify his preferences by assigning values to the variables from the set *V* that contains *pref_weight (*with the domain “*low, medium, irrelevant”), pref_class (cheap, middle, high, premium, irrelevant)* and *pref_interfaces (standard, advanced)*[1]. We represent the actual customer requirements in a set *REQ* of positive ground literals that use the predicate symbols from *V,* e.g.,

*REQ = {pref_class(premium). pref_weight(low). pref_interfaces(advanced).}*

[1] Note that in many technical domains it is advantageous to question the customer about his preferences and not directly about product properties, in particular when the knowledge level of the customers can be low [2].

The expert's filter rules *FR = {f1, f2, f3}* determining the contents of the result set *RS* are as follows, whereby the trivial axiom $RS \subseteq PKB$ has to hold.

***f1***: $pref\_weight(low) \in REQ \Rightarrow \forall X,P: P \in RS: weight(X) \in P \wedge X < 150.$
***f2***: $pref\_interfaces(advanced) \in REQ \Rightarrow \forall P: P \in RS: interface(firewire) \in P.$
***f3***: $pref\_class(premium) \in REQ \vee pref\_class(high) \in REQ$
$\Rightarrow \forall X,P: P \in RS: interfaces(dockingstation) \in P \wedge (price(X) \in P \wedge X > 180).$

Given that the customer wants a premium class camera with an advanced set of interfaces and a model that also has a low weight, the application of the rules will result in no suitable product. One approach to deal with the problem is to assign priorities to each filter rule in advance and to iteratively retract the rules until a sufficient number of products remains. Retracting the weight-rule *f1* results in product *p3*, retracting *f2* alone will not help, and retracting *f3* leads to product *p2*. If we assume that the domain expert annotates the filter rules with initial priorities *f3* > *f1* > *f2*, because from experience he knows that the Firewire – requirement (*f2*) is not that important for most customers, the system will initially come up with solution *p3*, i.e., the rules *f2* and *f1* will be relaxed. If we assume that for each rule an explanatory text is maintained both for the case that the rule is applied and for the case it is relaxed, the system could explain:

- (f3 - positive): Based on your preferences, I propose the premium or high class model "p3" that also ships with a convenient docking station.
- (f1 - negative): Due to your other requirements, I included a camera in the proposal that does not fulfill your requirements with respect to a low weight.

Note that although filter *f2* was retracted, the conditions of the filter are fulfilled for product *p3*. As such, we can include the positive explanation for *f2*.

- (f2 - positive): This camera fulfils your requirements on advanced interfaces.

After this initial proposal and the explanation, the customer can be given the possibility to dynamically change the priorities of the rules, if they differ from the domain expert's predefined priorities and trigger the computation of another result.

Nonetheless, there are domains where there are strict rules that should never be relaxed or situations where the customer interactively states that he explicitly stipulates the application of specific rules. Consequently, a situation with an empty result set still can arise. In this case, it would be helpful for the customer if the system provides a set of alternatives of slight changes in the requirements such that a product can be recommended. If we assume that all filter rules in our example are strict ones, some *good* options for the customer could be as follows.

- (1) If you change your weight requirement from "low" to "middle", I can propose the following products: p3.
- (2) If you change your class requirement from "premium" to "middle", I can propose the following products: p2.

Theoretically, there are lots of other alternatives that differ from the previous ones in their "quality". For instance, all repair-alternatives that contain (1), as well as additional changes in the requirements can be seen as suboptimal. Furthermore, alternatives where requirements are completely taken back, i.e., changes where the removal of a predicate from *REQ* leads to a solution, are also not optimal.

After the description of the algorithm for preference-based relaxation in the next section, we will describe a technique for efficient computation of suitable repair alternatives in cases where no result exists.

We will use the following general definition of knowledge-based recommendation problems for the subsequent algorithms[2].

***Definition**: A Knowledge-based Recommendation Problem (KBRP) is a tuple <P, V, PKB, FR, REQ>, where P and V are sets of predicate symbols that are used for describing product properties and customer requirements. PKB represents the available products and is a set of sets of positive ground literals using the predicate symbols from P. FR is a set of logical sentences with the structure of "filter rules". REQ is a set of positive ground literals using the symbols from V and correspond to actual customer requirements.*
*A"filter rule" is a logical sentence in form of an implication, where the antecedent is a logical expression where predicate symbols from V are allowed, and where the consequent describes restrictions on elements of PBK by the usage of predicate symbols from P.*□

***Definition**: Given a Knowledge-based Recommendation Problem KBRP<P, V, PKB, FR, REQ>, KBRP is a called a "Valid KBRP" if there exists a set REQ (including the empty set) such that there exists a subset RS of PKB where FR $\cup$ REQ $\cup$ RS is satisfiable.* □

Generally, a *knowledge-based recommender* in this implementation independent problem formulation is a software system capable of computing a valid recommendation *RS*, given the parameters *PKB, FR,* and *REQ*, or reporting failure of finding a solution.

## Priority-based relaxation

The algorithm for priority-based relaxation is straight-forward and from the basic idea similar to the Hierarchical Constraint Satisfaction approach [3]. We extend the KBRP<*P, V, PKB, FR, REQ*> with a function $\Phi$ that relates each filter rule from *FR* with a priority value, i.e., a positive integer, whereby higher values stand for lower priorities and zero means that a rule should not be relaxed. The algorithm takes KBRP, $\Phi$ and a search limit as input and returns a set of products that remain after a successful relaxation process or otherwise an empty set. In addition, for each product in the result set, the sets of applied and removed filter rules for the explanation

[2] We use first order logic as a representation language in order to facilitate a clear and precise presentation.
In this context, we use the more general term "knowledge-based recommendation problem"; in the literature, the terms "filter-based recommendation" and "filter-based product retrieval" are also used to describe this class of systems.

process is constructed, whereby we include those filter rules in the set of applied filters that were originally removed due to a higher priority, but would hold for a given product, see filter rule *f2* in the example section.

```
Algorithm RELAX(PKB, FR, REQ, Φ, limit)
result = ∅                                  // initialize the result
allrelaxed = ∅                              // remember everything we relaxed
relaxable = subset of elements f of FR where Φ(f) > 0
while (| result | < limit ∧ | relaxable | > 0)   // as long as there are relaxable filters
                                            // and the limit is not reached.
   priority = highest value of Φ(f), where f ∈ relaxable
                                            // determine the set of filters to remove.
   relaxset = subset of relaxable, were Φ(f) = priority
   relaxable = relaxable \ relaxset         // remove the set from the original set.
   allrelaxed = allrelaxed ∪ relaxset       // remember the removal
   result = filter(PKB, relaxable , REQ);   // call the recommender/product finder.
end while
// compute the correct explanation sets
// variable "explanations" contains triples of products, applied, and relaxed filters
explanations = computeExplanations(relaxable, allrelaxed, result)
return result;
```

**Algorithm 1. Priority-based relaxation**

```
Algorithm COMPUTEEXPLANATIONS(applied, relaxed, result)
// The algorithm computes a set of positive and negative explanations
// for each product in the result set by testing each filter in the applied
// and relaxed sets.
explanations = ∅;
for each p ∈ result
   // prepare the real sets for the explanations
   pos_arguments = applied
   neg_arguments = relaxed
   for each f ∈ relaxed                     // test all relaxed individually
      rs = filter(p, f , REQ)               // call the recommender, check if the
                                            // current product fulfils the filter rule
      if (| result | > 0)                   // if filter consistent for that product
         pos_arguments = pos_arguments  ∪ f  // update the explanation sets, e.g.,
         neg_arguments = neg_arguments \ f   //  filter f2 in the example
   end for
   // add the new explanation tuple.
   explanations = explanations ∪ <p, pos_arguments, neg_arguments >
end for
return explanations
```

**Algorithm 2. Computing adequate explanations**

With regard to complexity of the algorithm, the computation of the result set without the explanations takes at most as many iterations as there are different values in the range of Φ. For each element in the result set, the number of iterations for the explanations is $|result|$ * $|relaxed|$. Note that in a practical setting we will not pre-compute all explanations in advance but rather on demand, when a user asks for the explanation for a specific product.

## Computing repair alternatives

In domains where some of the rules are strict and cannot be relaxed definitely, a situation with an empty proposal can still arise. Then, it would be helpful if the system can propose the customer a set of "repair" actions which corresponds to slight changes and revisions of the initial requirements. In principle, two basic approaches are possible. First, the underlying recommendation knowledge base can be extended with additional domain knowledge, i.e., a set of explicitly modelled repair rules like shown in [4] for the domain of product configuration and reconfiguration. Such approaches, however, are costly in terms of knowledge acquisition and in particular maintenance as for each change in the knowledge base also the repair rules have to be checked and possibly adapted. Even more, such knowledge bases tend to get complex because of the strong interdependencies between the rules. On the other hand, model-based (diagnosis) approaches like, e.g., [5] or [6], that try to minimize the amount of additional needed domain knowledge for the repair task, face the problem of large search spaces for the computation of possible repair alternatives. Applied to real-world problem settings, such systems rely on domain-specific heuristics for discriminating between the alternatives or different forms of (structural) abstractions in order to produce adequate results within limited resource bounds.

If we follow such a model-based approach, the search complexity in our domain is determined by the size of the domain of the variables (i.e., the user inputs) and the property, whether these variables can be multi-valued or not. For each single-valued variable with a domain-size of $n$, we have $n+1$ possible assignments when we include a special *null*-value with the meaning that the user did not specify a requirement for a variable. For each multi-valued variable, there are $2^n$ possible answer combinations. If we assume that there are three single-valued and three multi-valued variables involved in the remaining strict rules and we have an average number of four possible answers, there are theoretically more than 60.000 possible answer combinations. Obviously, an exhaustive search for those combinations is not feasible, given the tight time limits of an online recommendation system[3].

In the following, we present algorithms and techniques for efficient computation of a suitable set of alternatives where we use domain-independent heuristics as well as application-dependent variants in order to reduce the required search times for practical settings.

**Diagnosing the requirements**. Depending on the application domain it can be sufficient that the system presents the user a list of requirements that should be

[3] An offline pre-computation for *all* possible input combinations is not possible, given the vast search space for, e.g., twenty to twenty-five questions in a realistic scenario.

completely retracted for a possible solution, e.g., "*If you skip your requirements on the weight, I can propose the following solution...*" In such cases, the computation of alternatives can be performed by using the standard *Hitting-Set* algorithm that is commonly used in the field of model-based diagnosis ([7], [8]). The algorithm is used in a way that the nodes of the Hitting-Set DAG[4] are labelled with *conflicts* which are in our case subsets of user requirements that cause the result set to be empty. If there is no conflict generation support, it will be the union of all variables that are used in the non-relaxable filters that have to be applied in the current situation.

***Definition****. Given a Knowledge-based Recommendation Problem KBRP<P, V, PKB, FR, REQ>, a "conflict" for KBRP is a set C $\subseteq$ REQ such that there is no set RS $\subseteq$ PKB for which FR $\cup$ RS $\cup$ C is satisfiable.*

*A conflict is "minimal", if there is no proper subset C' of C such that C' is also a conflict for KBRP<P, V, PKB, FR, REQ>.* □

The outgoing edges of the DAG are labelled with variable names from the nodes which are retracted in the current search phase. The main advantage of this approach is that the diagnoses are computed in ascending order with respect to their cardinality, which is reasonable because we assume that alternatives with fewer changes in the requirements are preferred by the users. Even more, the tree pruning techniques from [7] can be exploited to reduce the search space, as we are typically only interested in *minimal diagnoses*. As an example, if we found that omitting the *weight* requirement results in a solution, we can remove branches in the search space that involve the *weight* requirement and any additional requirement.

As a simple heuristic which helps us to find a first solution more quickly is to try those variables first which are involved in more non-relaxable filter rules.

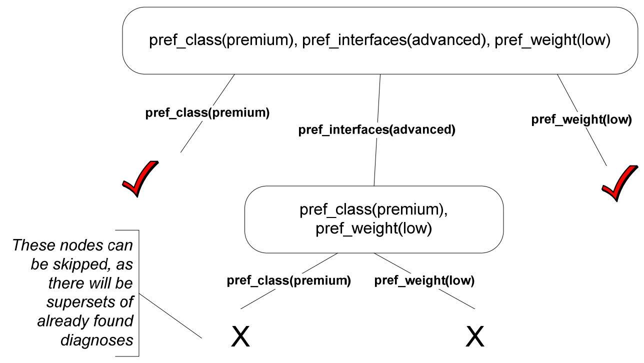
pref_class(premium), pref_interfaces(advanced), pref_weight(low)
pref_class(premium)
pref_interfaces(advanced)
pref_weight(low)
pref_class(premium), pref_weight(low)
pref_class(premium)
pref_weight(low)
These nodes can be skipped, as there will be supersets of already found diagnoses
X
X

**Fig. 1.** A simple example for HS-DAG construction

During the breadth-first HS-DAG construction (Fig. 1), each call to the *Theorem Prover (TP)* [7] corresponds to a search for products performed by the underlying knowledge-based recommender, where we leave out all requirements that are on the path from the root node of the DAG to the current node. If this search results in a

[4] Directed acyclic graph.

solution, we can close this node *n* and add *H(n)*[5] [7] to the set of diagnoses. Note that for each TP-call all the other current user requirements that are not involved in the current set of non-relaxable filters have also to be taken into account. Therefore, in order to guarantee that the algorithm will always find at least one solution, we make the assumption that there are no filter rules in the knowledge base whose antecedent defines a condition on the *non-existence* of a requirement, like

$\forall X: pref\_x(...) \notin REQ \Rightarrow ....$

If this – for many domains realistic assumption – holds, we can guarantee that by retracting user requirements no additional filter rules will get active during the search for solutions and we can thus safely prune the search tree.

*Optimizations.*

1. In general, we cannot assume that the underlying recommender system is capable of generating (minimal) *conflict sets*, such that we start with one single conflict that includes all problematic, i.e., conflicting user requirements. Nonetheless, during the HS-DAG construction, we remove elements from this conflict and check if a solution exists. If we encounter a situation where the removal of one input does not lead to a solution, we know that the remaining user inputs alone cause a conflict. As an online recommendation system will be used by many customers and subsequent customers may have a similar set of requirements, we can cache these already minimized conflicts and reuse them in the next session where a similar repair is needed. A safe reuse of such a conflict is possible if the cached conflict is a subset of the requirements of the next customer (compare, e.g., to [9]).
2. Because we allow disjunctions of predicates in the filters' pre-conditions, it can be the case that there are unassigned variables in the union set of the variables of the non-relaxable filters. Therefore, these variables can be removed from the initial *conflict* – which consequently reduces the search space – such that these variables do not have to be considered during the HS-DAG construction process.

**Searching for alternative solutions**. In some application domains, just presenting a set of requirements to be removed might not be satisfying for the customer. Even more, when individual requirements are fully removed, the customer might get a proposal that is not "near" to his original preferences. Fully removing a "low price" preference, for instance, could result in the proposal of premium priced products if no other filter rules constrain the price limit. Consequently, it would be preferable if the user can choose among several variants and interactively decide which of his requirements he is willing to give up or relax. In order to cope with such situations, the following value-based variant of the first approach can be used (see Fig. 2). Similar to the first approach, we proceed in a breadth-first approach. However, upon creation of a new node, all alternative settings for the variable under examination are tested individually except for the conflicting assignment. In cases, where more than one variable is tested (e.g., at the second level), we theoretically check all possible combinations of these variables, i.e., the Cartesian product of the possible values.[6]

[5] *H(n)* equals the set of all edge-labels from the root of the DAG to *n*.

[6] At the moment, the value-level approach handles user requirements with finite domains of enumeration values. For free-input requirements like a price limit we reduce the search space to respecting the requirement or not but do not search for alternative values.

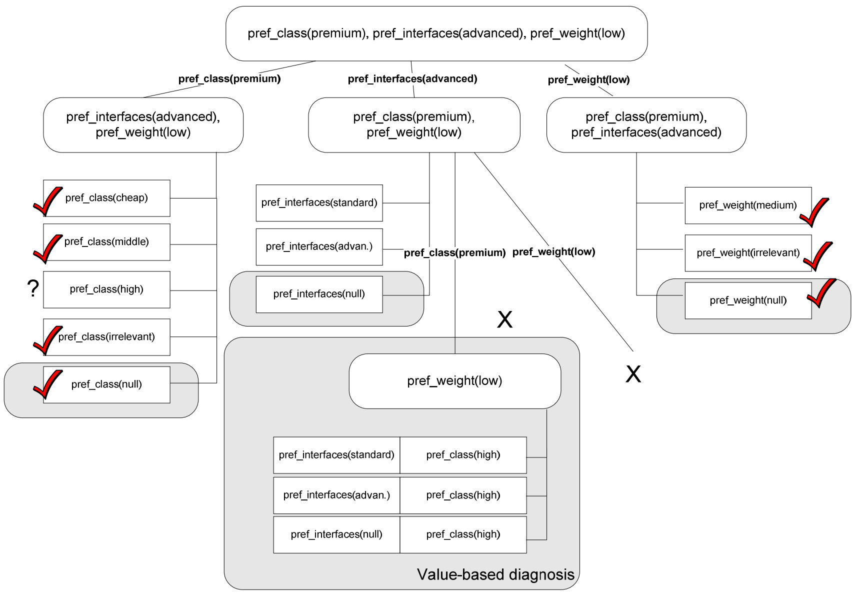

**Fig. 2. Value-based algorithm**

Note that in the value-based approach, we add a special "null" value (marked grey in Fig. 2) to the domain of each variable as the full removal of a requirement could be a possible option in some domains.

The main advantage of the breadth-first approach is that alternative solutions with fewer changes are found first. In practice, proposing alternatives with more than three or four variables changed does not help the users too much anyway, as the differences to the original preferences are not satisfying for the users. Therefore, in practical situations, the search is stopped when the first few alternatives are found.

When using the value-based approach, pruning has also to be performed on the value level. As we can see in Fig. 2, the node with the path (*pref_interfaces*, *pref_class*) cannot be immediately closed like in the first algorithm. Although we already found several solutions by changing the value of *pref_class* alone, there was no solution when we tried the value "*high*" (indicated by the question mark). Nonetheless, as we make no restrictions on the filter rules except for those on the structure described in the definitions, it could be the case that there exists a combination of *pref_class(high)* and some value of *pref_interfaces* such that a solution with changes in both variables is possible. However, note that we only have to check the combinations with *pref_class(high)* because combinations with other values of *pref_class* would result in a superset of an already found diagnosis. On the other hand, nodes that would involve *pref_weight* can be immediately closed since all alternative assignments to *pref_weight* result in a suboptimal solution.

Nonetheless, additional pruning approaches can be applied to reduce the search space, but come at the cost of loss of solutions. First, we could omit such cases as described in the last paragraph and prematurely prune the node with the path (*pref_interfaces*, *pref_class*), when we have domain-specific knowledge about the filter constraints. In addition, for each node we can stop examining the possible value combinations, once we found a first solution, e.g., after trying the “cheap” value for *pref_class* we do not check the other alternatives. Such an approach can be useful in cases where we are only interested repair alternatives that involve single changes.

In general, the diagnosis algorithm will always find at least one solution given a valid recommendation problem, as the search space is exhaustively searched and no minimal solutions get lost by the value-based pruning techniques.

**Discriminating between diagnoses**. In cases when there are lots of solutions, i.e., repair alternatives returned by the search alternatives, these proposals should be ranked such that the customer can choose the alternative that matches his preferences best. A quite intuitive ranking will be based on the number of changes that are required in the requirements. Depending on the domain, other application-specific orderings can be used in order to rank the alternatives with the same number of changes. First, we can prefer those alternatives that maximize the number of products that can be proposed if the changes are applied. Another ordering can take the differences to the original requirements on the value level into account. In many domains, the possible values for a specific variable have an implicit ordering, like a price preference (low, medium, high etc.). As a consequence, from a practical perspective, it can even be better to promote an alternative where two of the requirements are changed to a neighbouring value, than to propose a single-change alternative where a customer’s preference has to be inverted, e.g., changed from “yes” to “no”. In principle, an initial “cost model” for changes can be computed without further domain knowledge, if we assume that the possible values for a variable are defined in such an implicit order. Before the results are presented to the user, the system can rank the alternatives based on these costs which are determined by the *distance* to the original requirement and the overall number of the possible values[7]. For multi-valued variables, changes where the original requirement statement and some more or fewer values lead to a solution can be for instance preferred over others that have complete different values.

If desired, such a model of “change costs” can also be explicitly defined in the knowledge base. The definition of such a model however can require significant knowledge acquisition efforts as these cost functions have to be defined manually for each variable. Nonetheless, compared with approaches where “repair rules” are explicitly defined, the definition of cost functions has the advantage that changes in the model only have local effects and the overall consistency of the knowledge base is not affected.

Finally, when such a cost model exists, it can be exploited during the search phase in settings where we perform an incomplete search and stop at a certain threshold, e.g., when a defined number of alternatives is found. As an example, we could stop examining other possible values for a price preference, when we found that a change from “low” to “medium” results in a solution, as we know that increasing the price limit further will only result in a suboptimal alternative with respect to the costs, i.e., the quality of the repair alternative.

[7] The number of possible values is important, because the *distance* from “yes” to “no” is small, but there are no other alternatives, so the *cost* estimate should be correspondingly high.

## Experimental results

The described algorithms were implemented as add-on to the ADVISOR SUITE [10] framework, a research toolkit for rapid development of personalized online sales advisory systems. For the evaluation process, knowledge bases from several real-world problems from different application domains were tested. The application domains range from complex areas like investment alternatives to technical items like digital cameras or skis, up to "quality and taste" domains like wine or cigars.

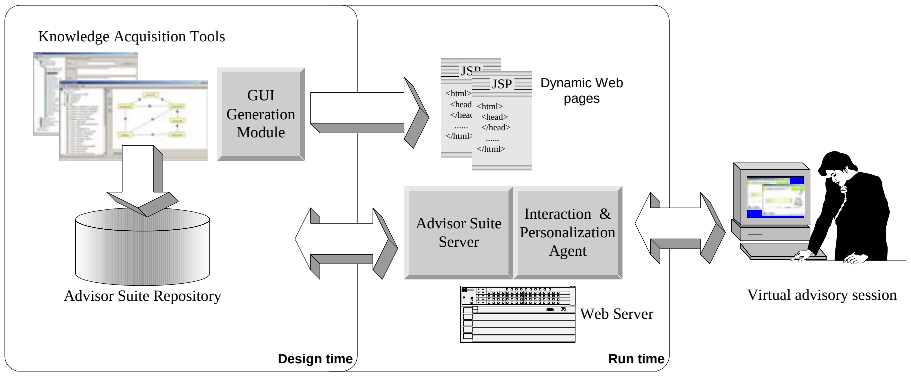

**Fig. 3.** Overall architecture of ADVISOR SUITE

Figure 3 depicts the overall architecture of the ADVISOR SUITE framework. The required knowledge for recommendation and personalization is captured using graphical knowledge acquisition tools and stored in a knowledge base which is built on top of a relational database system. In the Java-based ADVISOR SUITE SERVER module, the core recommendation logic is implemented; the individual advisory sessions are managed by the INTERACTION AND PERSONALIZATION AGENT. Short response times and high overall performance are key issues for such online Web-based systems. In general, the knowledge-based ADVISOR SUITE framework thus extensively caches and pre-loads the contents of the knowledge base and pre-compiles the recommendation rules into a compact internal format, such that a fast rule-evaluation process is possible.

**Problem size.** In our real-world advisory problems, the knowledge-base typically comprises twenty to thirty questions (variables) whereby – based on a personalization process – not every user is asked every question [2]. The number of filtering rules that determine the product selection mostly remained manageable and ranges from twenty to sixty expert rules[8]. The number of available products varies with the domain, from a few dozens when "services" are recommended, to several hundred when the domain is investment advisory or digital cameras.

**Priority-based relaxation.** The computation of explanations in terms of applied and relaxed advisory rules (Algorithm 1) is not problematic, as the search complexity

[8] The overall knowledge base is more complex, as it contains additional knowledge for the personalization of the dialogue flow, the adaptive user interface, as well as for the personalized ordering of the results records.

is linear with the number of priority levels, typically not more then twenty or thirty. Each check whether a sufficient number of results will be available upon relaxation of the filter rules on a certain level, can be performed within 10 to 20 milliseconds on a standard PC and database system depending on the number of available products, the overall response times for all tested cases were below one second. From the end-user perspective, however, this feature was highly appreciated by users in all domains where such an advisory application was implemented. First of all, the existence of a personalized explanation significantly increases the customers' confidence in the proposal, i.e., when reading the natural language explanations of the applied advisory rules and the justification for the products the user implicitly "learns" things about the domain. Furthermore, the possibility to stipulate the application of individual rules or decrease the priority of a rule lets the user express his real preferences in an intuitive and comprehensible way, in cases where the priorities which are predefined by the domain expert do not match his personal interests. Nonetheless, we also found that some of the users were overwhelmed when they are asked to assign relative priorities to the rules, such that in most cases we followed an approach where the only possibilities are to force the application of a rule or completely ignore it and to undo these choices.

**Search for alternatives.** The search for possible alternatives is computationally complex and has to be fine-tuned for each application domain. In typical applications, the list of explanations of the expert rules leading to the empty result set is presented to the user, followed by the list of repair alternatives. Depending on the domain, the complexity of the problem, and also the skill-level of the users, we have to tune the parameters of the diagnosis process: In some cases, only single-step or two-step repairs are comprehensible for the users; in other domains users want to have a choice of many different and possibly complex alternatives. In most cases, the maximum cardinality of the desired repairs is three or four as repairs that involve more changes are hard to comprehend and asses for the end user. If for instance five or more of the originally fifteen requirements have to be changed, the customer's confidence in the proposal might be low, because the product that is proposed after the change does not match a large part of his preferences.In our test cases, the computation of diagnoses up to cardinality three with single-valued attributes showed acceptable response times below one second for the computation of all possible repair alternatives without any further optimization, even in cases where nearly all of the customer requirements were involved in a conflict. For the computation of diagnoses with higher cardinality, we introduced several optimizations described as follows, such that the response times stay within the acceptable limit of two or three seconds for the hardest real-world problems with cardinality five.

1. *Result caching*. For each node in the search tree, we have to check whether changing a value results in a solution or not which corresponds to a call to the recommender system. In the ADVISOR SUITE framework, the results of previous sessions (together with explanations and repairs) are compactly stored in memory and on disk such that the system constantly "learns" new relations between

requirements and solution[9]. Our experiences show that only a fragment of the theoretically possible input combinations actually occur and many users have same or similar requirements such that the space requirements for this cache are limited. A typical example for such a "pattern" in user requirements is that customers who specify a high price-limit also have a high preference for brand products. We encode such previous results (and also those that origin during the diagnosis process) compactly, such that the check for a solution for an already known combination is then less then one millisecond. Finally, we also cache the possible repair alternatives which can subsequently be accessed without reasoning, when the same set of requirements occurs a second time. The experiences of an application with about fifteen thousand online advisory sessions in the first week of deployment shows that a high "cache-hit" rate can be achieved that justifies the limited overhead of managing such a cache.

2. *Conflicts re-using*. The same principle of such a system-wide cache can also be applied for conflicts, as in particular the computation of minimal conflicts can be a time-consuming task.
3. *Domain-dependent heuristics*. Finally, domain-dependent heuristics can help us in reducing the search space, in particular for multi-valued requirements that significantly enlarge the size of the search space. In many domains, we know for each multi-valued attribute, whether more values in such a requirement reduce the set of possible products or broaden this set, e.g., when the requirements have an implicit "one-of" semantics. Therefore, depending on the semantics, we can limit the search to alternatives with more or fewer values in the requirements and prune out all other constellations, which will not result in good solutions.

In general, the diagnostic approach for the search of repair alternatives has to be more parameterized and fine-tuned for a specific application than the relaxation technique. In particular, special attention has to be paid that the underlying complexity is hidden from the user and the user is not overwhelmed by a large set of complex alternatives. From the user interface perspective it is therefore important that we for instance aggregate the alternatives on the same variable set (e.g., "change the weight requirement to *middle* or *irrelevant*") and give the user the chance to accept and apply the alternative in a single interaction.

## Conclusions

We have presented two techniques for dealing with empty result sets in knowledge-based recommender systems, whereby we based our work on a general model of this class of recommender systems that in practice base the search on product filtering.

The technique described for finding alternative requirements follows the tradition of approaches described in ([6],[8],[11], or [15]), where similar algorithms were used for computing action repair actions both for the domains classical model-based

[9] The pre-computation of all possible user input combinations and corresponding results is not feasible both with respect to time and space requirements. The cache has to be invalidated upon the periodic changes in the knowledge base, e.g., when new filter rules are introduced.

diagnosis (e.g., electronic circuits) as well as for the domains of re-configuration and debugging of software and knowledge-bases. The algorithms in this paper are designed in a way that they can be implemented non-intrusively as add-on to existing advisory systems and product finders. The only requirements are that filter rules can be selectively removed and added to the knowledge base (priority-based relaxation) and user inputs can be removed or changed dynamically (search for repair alternatives). The underlying implementation of the product finder has not to be changed or known for the application of the described techniques.

From the end user perspective, the usage of one or the other of the techniques adds a preference aspect ([12], [13]) to the advisory system, where the personalization of proposals is not only limited to preferences expressed on initial user requirements; the end user is also given a certain degree of freedom in his choice of repairs (compromises in the conflicting requirements) and the prioritization of advisory rules. The experiences show that the acceptance of the system and the confidence in the proposal increase, when the end user is able to understand and manipulate the results of the advisory process in these ways.

The presented work also strongly corresponds with the field of "Cooperative Answering" [18] in database systems, where the problem exists, that direct answers to database queries like "yes" or "no" are not always the best answers, i.e., an intelligent system would e.g. allow for the usage of "soft constraints" or preferences and cooperatively provide extra or alternative information. In the work of [19], for instance, automated analysis of the individual parts of a failing query is proposed, such that the system can provide an explanation which parts of the query caused the failure. While the approach in [19] involves high costs for identifying such sub-queries, this division in sub-queries is our application domain given by the filter constraints themselves. In addition, our approach allows us to define natural-language explanations for failed sub-queries as well as interactive, preference-based selection of sub-queries to be applied.

Our future work includes the analysis whether similar techniques can be applied to the class of recommendation systems based on Case-based Reasoning ([16], [17]) in particular when the *case base* has to be updated after changes of the recommendation rules or when new products are available. Vice-versa, it will be analyzed if techniques from the CBR field can be adopted for filter-based recommender systems. In addition, we will further evaluate successful approaches from the areas of configuration and constraint satisfaction; in particular techniques based on *local search* [14] whose principle of iterative solution improvement matches the requirements of our application domain. Finally, further work will be spent on personalizing and improving the quality the resulting explanations such that they contain no spurious elements (see, e.g., [20]) or are adapted according to the current user's skills or preferences.

## References

[1] D. Bridge. Product recommendation systems: A new direction. In R. Weber and C. Wangenheim, editors, Procs. of the Workshop Programme at the Fourth International Conference on Case-Based Reasoning, p. 79-86, 2001.
[2] L. Ardissono, A. Felfernig, G. Friedrich, D. Jannach, M. Zanker, and R. Schäfer. A framework for the development of personalized, distributed web-based configuration systems. AI Magazine, 24(3):97-110, 2003.
[3] T. Schiex, H. Fargier, and G. Verfaille. Valued constraint satisfaction problems: Hard and easy problems. In International Joint Conference on Artificial Intelligence, p. 631-639, Montreal, Canada, 1995.
[4] T. Männistö, T. Soininen, J. Tiihonen, and R. Sulonen. Framework and Conceptual Model for Reconfiguration. In Configuration Papers from the AAAI Workshop, AAAI Technical Report WS-99-05. AAAI Press, 1999, p. 59-64.
[5] S. Srinivas and E. Horvitz, Exploiting System Hierarchy to Compute Repair Plans in Probabilistic Model-Based Diagnosis, In: Proceedings of Eleventh Conference on Uncertainty in Artificial Intelligence, Montreal, 1995, Morgan Kaufmann, p. 523-531.
[6] G. Friedrich, G. Gottlob, and W. Nejdl: Formalizing the Repair Process - Extended Report. Annals of Mathematics and Artificial Intelligence, Vol. 11(1-4): 187-201 (1994)
[7] R. Reiter. A theory of diagnosis from first principles. Artificial Intelligence, 32(1), Elsevier, 1987, p. 57-95.
[8] A. Felfernig, G. Friedrich, D. Jannach, M. Stumptner. Consistency-based diagnosis of configuration knowledge bases, Artificial Intelligence, 152(2), p. 213-234, 2004.
[9] R. Greiner, B.A. Smith, R.W. Wilkerson. A correction to the algorithm in Reiter's theory of diagnosis. Artificial Intelligence, 41(1), Elsevier, 1989, p. 79-88.
[10] D. Jannach and G. Kreutler, Building on-line sales assistance systems with ADVISOR SUITE, In Proceedings: 16th Intl. Conference on Software Engineering and Knowledge Engineering (SEKE'04), Banff, CAN, 2004.
[11] M. Stumptner and F. Wotawa, Reconfiguration using Model-based Diagnosis, in: Proceedings of the International Workshop on Diagnosis (DX99), June 1999.
[12] U. Junker, Preference-Based Search for Scheduling. In Proceedings: AAAI/IAAI, Austin, TX, USA, 2000. p. 904-909.
[13] U. Junker, Preference programming for configuration, In Proceedings IJCAI'01 – Workshop on Configuration, Seattle, 2001.
[14] R. Sosic and J. Gu. Efficient Local Search with Conflict Minimization: A Case Study of the N-Queens Problem. IEEE Transactions on Knowledge and Data Engineering, Vol. 6, 5, p. 661-668, Oct 1994.
[15] L. Console, G. Friedrich, D. T. Dupré: Model-Based Diagnosis Meets Error Diagnosis in Logic Programs. IJCAI 1993, Chambéry, France, p. 1494-1501.
[16] R. Burke, The Wasabi Personal Shopper: A Case-Based Recommender System, In Proceedings: AAAI/IAAI, Orlando, Florida, 1999, p. 844-849.
[17] R. Burke, Knowledge-based Recommender Systems. In A. Kent (ed.), Encyclopedia of Library and Information Systems. Vol. 69, Supplement 32. Marcel Dekker, 2000.
[18] T. Gaasterland, P. Godfrey, and J. Mincker, An overview of Cooperative Answering, Journal of Intelligent Information Systems Vol. 1(2), pp. 123-157, Kluwer, 1992.
[19] J.M. Janas. On the Feasibility of Informative Answers. In: Gallaire et al., Advances in in Database Theory, Vol. 1. Plenum Press, 1981
[20] G. Friedrich, Elimination of spurious explanations, Proc. of 16th European Conference on Artificial Intelligence, Valencia, Spain, 2004.