Fast computation of query relaxations for knowledge-based recommenders

Dietmar Jannach
Department of Computer Science, TU Dortmund, 44221 Dortmund, Germany, dietmar.jannach@udo.edu

"No matching product found" is an undesirable message for the user of an online product finder application or interactive recommender system. *Query Relaxation* is an approach to recovery from such retrieval failures and works by eliminating individual parts of the original query in order to find products that satisfy as many of the user's constraints as possible.

In this paper, new techniques for the fast computation of "user-optimal" query relaxations are proposed. We show how the number of costly catalog queries can be minimized with the help of a query pre-processing approach, how we can compute relaxations that contain *at-least-n* items in the recommendation, and finally, how a recent conflict-detection algorithm can be applied for fast determination of preferred conflicts in interactive recovery scenarios[1].

Keywords: Knowledge-based recommender systems, query relaxation, user interaction.

1. Introduction

In knowledge-based recommendation approaches the main task of the system is to retrieve and rank a set of items from a given catalog that match the current user's requirements or preferences. The preference elicitation and matching process can be designed in different ways; examples include critiquing, similarity-based retrieval and case-based reasoning as well as constraint-based or rule-based filtering ([2,3,4,5,9,17,18,20,21,22,24]).

Although these techniques both use different forms of means-end knowledge to match items with user requirements and may rely on different user interaction styles, in many implementations the customer requirements are directly viewed as constraints on item features, at least in some phase in the retrieval process [15]. However, when user requirements are treated as constraints on items, the undesirable situation can easily arise that none of the products in the catalog fulfills the users constraints. One approach to recovery from retrieval failures in such situations is to search for a *relaxation* of the user's query with the goal of finding those products in the catalog that satisfy as many of the user's constraints as possible. This strategy of relaxing constraints is used for instance by algorithms proposed by from McSherry [17] and Ricci et al. [20]. The general goal of these algorithms is to find an "optimal" subset of the originally posted constraints that (a) are all satisfied by at least one of the catalog items and (b) optimize some evaluation function, which can be for instance simply the number of fulfilled constraints or a more complex function that takes "costs of compromise" into account, i.e., that account for the fact that not all constraints may be equally important to the user.

The main challenge when determining such optimal subsets *within the narrow time limits* of an interactive recommendation session is the size of the search space [1]. If there are n different subqueries (constraints) in the original user query, the number of possible combinations of constraints is 2^n. McSherry's algorithm for finding all "maximal succeeding subqueries" [17] for instance in the worst case scans this complete set of possible subquery combinations. Such a situation can occur when every single user constraint – even if applied individually – causes the result set to be empty. If we assume that the check whether a subquery is successful or not is implemented as a count query to the database, we would need more than one thousand such queries in the worst case for this single user request, if there were ten user constraints.

This issue of search complexity is also discussed and addressed in [20]. Their approach relies on

[1]This paper builds upon and significantly extends the work from [10] and [11].

what they call “feature abstraction hierarchies” for clustering related constraints that are subsequently relaxed as a group. While this technique reduces the theoretical search space as the much smaller number of constraint groups determines the complexity, it comes at the price of incompleteness, i.e., not all theoretically possible relaxations can be found.

In this paper we propose new techniques for the fast determination of “optimal” relaxations for failing queries in knowledge-based recommender systems. First, we show how the number of required data base queries can be limited to the number of subqueries by evaluating the user’s constraints individually (in advance) and storing the partial results in compact in-memory data structures. In addition, we propose an algorithm with which we can determine relaxations that lead to *at-least-n* items by analyzing these data structures in memory without requiring further queries to the catalog. Both proposed algorithms are capable of taking “costs of compromise” into account, which means that each constraint can be annotated by a cost value that is subsequently used by the algorithm to minimize the overall costs.

For application scenarios in which the user is interactively involved in the recovery process as described in [14] or [15], we show how a recent, general-purpose conflict detection algorithm [13] can be applied to search for “preferred”, minimal conflicts in the user requirements. Again, the overall goal is to minimize the number of database queries.

The rest of the paper is organized as follows. After the introductory example in the next section, we describe our algorithms for non-interactive relaxation and show formally that none of the possible relaxations will be missed, even though the user constraints are only evaluated individually. Next, we propose an algorithm capable of finding optimal relaxations that lead to *at-least-n* results, which is often required in practical application scenarios. Finally, algorithms for interactive recovery scenarios are described before we discuss the implementation of the algorithms in a domain-independent framework for building knowledge-based recommender systems.

2. Example problem for non-interactive failure recovery

We will illustrate the different algorithms based on the following small example from the domain of digital cameras. Our item catalog (Figure 1) contains four cameras $p1$ to $p4$; each camera is characterized by a set of feature values.

ID	USB	Firewire	Price	Resolution	Make
p1	true	false	400	5 MP	Canon
p2	false	true	500	5 MP	Canon
p3	true	false	200	4 MP	Fuji
p4	false	true	400	5 MP	HP

Fig. 1. Product database of digital cameras.

Let us assume that the online user has specified the following four requirements $Q1$ to $Q4$.

$$\begin{aligned} Q \equiv \{ \quad & usb = true\ (Q1), \\ & firewire = true\ (Q2), \\ & price < 300\ (Q3), \\ & resolution \geq 5MP\ (Q4)\} \end{aligned}$$

If we construct a conjunctive query to the product catalog from these requirements, we can easily see that no items will be retrieved as none of the products satisfies all constraints.

When using query relaxation as a recovery strategy, we thus need to search for a subset of the user constraints $Q1$ ... $Q4$ which, if used in a conjunctive catalog query, does not lead to an empty result set. The set of possible subsets of the four constraints ($2^4 = 16$ subsets) is depicted in Figure 2. The algorithm proposed by McSherry [17] initially generates this full set of possible combinations and then scans this set in decreasing order with respect to the cardinality of the combinations. If a succeeding subquery $S \subset Q$ is found, all other subsets $S' \subset S$ are also removed from the search space. This pruning technique may have however only limited effects. In the example, there is no three-element subquery that leads to a result, i.e., at least two constraints have to be dropped to find a camera that fulfils the remaining ones. The algorithm from [17] will however scan all three-element combinations before noticing that for instance the subquery consisting of $Q2$ and $Q4$ will succeed. The proposed approach will thus only prune the single-element subqueries $Q2$ and $Q4$

from the search space, i.e., the effects of pruning are rather limited.

With respect to terminology, note that in our work we use the concept of a "minimal relaxation", which corresponds to a subset $R \subset Q$, such that $Q \setminus R$ (the "maximal succeeding subquery") leads to a non-empty result set. In other words, R contains those constraints from Q which have to be given up in order to find matching items in the catalog.

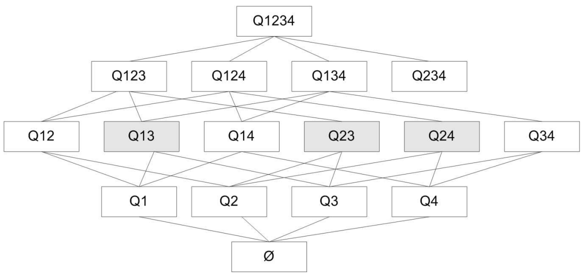
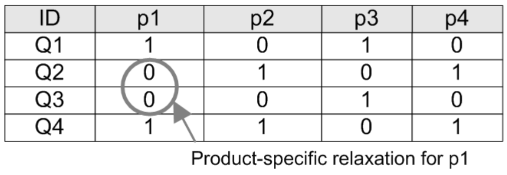


Fig. 2. Lattice of possible subqueries, minimal sets of "dropped" constraints (relaxations) are printed in shaded boxes.

The minimal relaxations for our example are therefore $\{Q1, Q3\}$, $\{Q2, Q3\}$, and $\{Q2, Q4\}$ and are printed in shaded boxes in Figure 2. Applying relaxation $\{Q1, Q3\}$ would for instance mean that we give up both the constraint on USB-connectivity (Q1) and the restriction on the price (Q4). As the products $p2$ and $p4$ satisfy the remaining constraints $Q2$ and $Q4$, they would be part of the proposal generated by the system.

In our algorithms for determining such minimal relaxations, we will use a strategy which is not based on the lattice of combinations depicted in Figure 2. As mentioned above, our primary goal is to reduce the number of required catalog queries which are typically the most costly operations. We therefore propose a technique in which we can limit the number of needed catalog queries to the number of constraints in the original query Q, which means that in our case we will only need four queries. We can achieve this by first evaluating the subqueries individually and then analyzing and combining the partial results in memory. Our procedure for determining (minimal or optimal) relaxations for non-interactive settings is thus based on a pre-processing step in which the matching items for each constraint are retrieved individually and stored in a compact in-memory data structure.



ID	p1	p2	p3	p4
Q1	1	0	1	0
Q2	0	1	0	1
Q3	0	0	1	0
Q4	1	1	0	1

Product-specific relaxation for p1

Fig. 3. Results of evaluating the individual constraints in the user query.

Figure 3 shows a sketch of this data structure. Each row of the matrix corresponds to one of the constraints of the user. The values in the columns indicate whether product p_i satisfies the constraint ("1") or not ("0"): $Q3$, the price constraint, will for instance filter out all cameras except $p3$. Note that for constructing the matrix, exactly four catalog queries (count queries to the database) are required and this in-memory data structure can be shared among all users of the recommender application.

The search for minimal relaxations (and the corresponding maximal succeeding subqueries) proceeds as follows. For each of the products p_i in the catalog, we can easily determine which of the user constraints will cause p_i to be filtered out. In the example in Figure 3, we for instance see that $p1$ does not satisfy constraints $Q2$ and $Q3$. Thus, if we removed $Q2$ and $Q3$ from the original query Q, $p1$ would be in the result set, which in turn means that $\{Q2, Q3\}$ also represents a possible relaxation to the overall query Q, though is not necessarily minimal.

We shall refer to such a relaxation as *product-specific relaxation PSX*. For example, PSX(Q,p1) = {Q2,Q3}, PSX(Q,p2) = {Q1,Q3} and so forth. Although product-specific relaxations are not necessarily minimal with respect to the overall query, we can easily determine one optimal or the set of all minimal relaxations by analyzing the set of given PSXs. In order to find the set of all minimal relaxations we need to iterate once over the set of PSXs. If the currently regarded PSX(Q,p_i) is a superset of an already found PSX, ignore it. Else, add it to the result list and remove all those PSXs that were already found and for which the current PSX(Q,p_i) is a subset.

In the example, the algorithm would incrementally add PSX(Q,p1) = {Q2,Q3}, PSX(Q,p2) = {Q1,Q3} and PSX(Q,p3) = {Q2,Q4} to the set of minimal relaxations. Since PSX(Q,p2) = PSX(Q,p4), this last PSX would not be added to the set.

Once all the minimal relaxations are determined and stored in the results matrix, we can select the one that promises to be the most useful for the customer. When there is no "cost model" for the individual parts of the query, a suitable strategy will be to choose the relaxation with the smallest cardinality. If a cost model exists (e.g., background information that users rather tend to compromise on the make than on the price), we can also determine the relaxation that minimizes such a cost function. Note that only minimal relaxations have to be considered when we assume a cost function that monotonically increases with the number of involved constraints.

Overall, the number of required database queries is thus limited to the number of sub-constraints in user query and does not grow exponentially as in previous approaches. When comparing the computational complexity of our approach with McSherry's [17] algorithm in extreme cases, note that given for instance a setting with only 3 constraints but 1000 products, McSherry's algorithm would consider at most 7 combinations of constraints (candidate relaxations) and require 7 database queries. Our algorithm would require only 3 queries but there would be 1,000 candidate relaxations to be considered in the results matrix. Database queries are however by orders of magnitude more costly than in-memory bit-set operations. Moreover, finding the best relaxation among the 1000 candidates within the internal data structure can be done in linear time, which will be discussed in a later section.

In the next section we will discuss and analyze our approach and algorithms in more detail based on the formalisms developed in [8], [15], and [17].

3. Finding minimal and optimal relaxations

The goal of the this section is to prove that the algorithm described informally in the previous section is sound and complete, i.e., that it finds all minimal relaxations and that all relaxations computed by the algorithm provided are in fact minimal.

The following basic definitions shall apply.

Definition 1 *(Query): A query Q is a conjunctive query formula, i.e., $Q \equiv A_1 \wedge ... \wedge A_k$. Each A_i is an atom (constraint).*

In the following we denote the number of atoms in the query by $|Q|$ (query length).

Definition 2 *(Subquery): Given a query Q consisting of the constraints $A_1 \wedge ... \wedge A_k$, a query Q' is called a subquery of Q iff $Q' \equiv A_{s_1} \wedge ... \wedge A_{s_j}$, and $\{s_1, ...s_j\} \subset \{1, ..., k\}$*

Lemma 1 *If Q' is a subquery of Q and Q' fails, then Q must also fail.*

Note that in [15] and [16] queries are split into subqueries according to the attributes that the query involves. Within our framework, however, such a specific form of partitioning is not required. Furthermore, note that query atoms can be arbitrarily complex and one single query atom could for instance consist of a disjunction of several predicates, which is a common situation in knowledge-based and interactive recommender applications.

Valid and minimal relaxations and maximal succeeding subqueries are defined as follows.

Definition 3 *(Valid relaxation): If Q is a failing query and Q' is a succeeding subquery of Q, the set of atoms of Q which are not part of Q' is called a valid relaxation of Q.*

Definition 4 *(Minimal relaxation): A valid relaxation R of a failing query Q is called minimal, if there exists no other valid relaxation R' of Q which is a subset of R.*

Lemma 2 *Given a failing query Q and a non-empty product catalog P, a relaxation R for Q always exists. In the worst case, all constraints in Q can be dropped ($R = Q$).*

Maximal succeeding subqueries in the sense of [8] are directly related to minimal relaxations.

Definition 5 *(Maximal succeeding subquery – XSS): Given a failing query Q, a Maximal Succeeding Sub-query (XSS) for Q is a non-failing sub-query Q^* of Q such that there exists no other query Q' which is also a non-failing sub-query of Q and is such that Q^* is a sub-query of Q'.*

Lemma 3 *Given a maximal succeeding subquery Q^* for Q, the set of atoms of Q which are not in Q^* represent a minimal relaxation R for Q.*

Next, we define the rows of the results matrix described in Section 2 as functions of the product catalog and the individual query atoms. A "partial query result" for a single query atom shall contain only those products that satisfy the corresponding constraint. Technically, these partial results can be represented as zeros and ones as described in the previous section.

Definition 6 *(Partial query results – PQRS): Let $P = \{p_1, ..., p_n\}$ be the set of products in the catalog. Given a query Q consisting of atoms A_1, ..., A_k, then $PQRS(A_i, P)$ is a function that returns the subset $P' \subseteq P$ for which constraint A_i is satisfied, $i \in \{1, ..., k\}$.*

Our algorithm for finding all minimal relaxations operates solely on the basis of partial query results and analyzes the given product-specific relaxations to determine the optimum. Hence, only $|Q|$ catalog queries are required overall.

Definition 7 *(Product-specific relaxation – PSX): Let Q be a query consisting of the atoms A_1, ..., A_k, P the product catalog, and p_i an element of P. $PSX(Q, p_i)$ is defined to be a function that returns the set of atoms A_i from $A_1, ..., A_k$ that are not satisfied by product p_i.*

Lemma 4 *The set of atoms returned by $PSX(Q, p_i)$ is a valid relaxation for Q.*

3.1. Finding all minimal relaxations

Figure 4 shows *MinRelax*, an algorithm for determining all minimal relaxations for a query Q and a product catalog P. The algorithm scans all product-specific relaxations and returns a specific subset of all PSXs, i.e. it works under the assumption that all minimal relaxations are among the given PSXs.

Subsequently, we shall show that Algorithm *MinRelax* is sound and complete, i.e., it only returns minimal relaxations and it does not miss any of the minimal relaxations. Let us consider as a first step how many minimal relaxations there can be in theory.

Proposition 1 *Given a failing query Q and a product database P containing n products, at most n minimal relaxations can exist.*

Proof: For each product $p_i \in P$ there exists exactly one subset PSX of atoms of Q which p_i does not satisfy and which have to be definitely relaxed altogether in order to have p_i in the result set. Given n products in P, there exist exactly n such PSXs. Thus, any valid relaxation of Q has to contain all the elements of at least one of these PSXs for obtaining one of the products of P in the result set. Consequently, any relaxation which is not in the set of all PSXs of Q must be a proper superset of one of the PSXs and is consequently no longer a minimal relaxation. This finally means that any minimal relaxation must be contained in the PSXs of all products and not more than $|P| = n$ such minimal relaxations can exist. □

Proposition 2 *Algorithm MinRelax is sound and complete, i.e., it returns exactly all minimal relaxations for a failing query Q.*

Proof: The algorithm iteratively processes the product-specific relaxations PSXs for all products $p_i \in P$. From Lemma 4 we know that all these PSXs are already valid relaxations. Minimality of the relaxations returned by *MinRelax* is guaranteed by the algorithm, because (a) supersets of already discovered PSXs are ignored during result construction and (b) already discovered PSXs that are supersets of the current PSX are removed from the result set. As such, there cannot exist two relaxations $R1$ and $R2$ in the result set for which $R1$ is a subset of $R2$ or vice versa. In addition, we know from Proposition 1 that all minimal relaxations are contained in the PSXs of the products of P. Since *MinRelax* always processes all of these elements, it is guaranteed that none of the minimal relaxations is missed by the algorithm. □

The complexity of *MinRelax* is characterized as follows. In the algorithm, the number of required executions of the typically most costly operation – querying the catalog – is equal to the number of atoms of the original query. The other operation that potentially contributes to computation times is the determination of the subset and superset property when iterating over the products. In the theoretically worst case, at each iteration this check has to be done for all of the previously found relaxations. This means that if we have n products, $(n * (n + 1))/2$ of such checks have to be done in the worst case. However, such checks can be efficiently done in memory and we have found that in realistic cases the number of actual checks

```
ALGORITHM: MinRelax
In: A query $Q$, a product catalog $P$
Out: Set of minimal relaxations $minRS$ for Q
MinRS = $\emptyset$
forall $p_i \in P$ do
    PSX = Compute the product-specific relaxation $PSX(Q, p_i)$
    % Check relaxations that were already found
    SUB = $\{r \in MinRS \mid r \subset PSX\}$
    if $SUB \neq \emptyset$
        % Current relaxation is superset of existing relaxation
        continue with next $p_i$
    endif
    SUPER = $\{r \in MinRS \mid$ PSX $\subset r\}$
    if $SUPER \neq \emptyset$
        % Remove supersets
        $MinRS = MinRS \setminus SUPER$
    endif
    % Store the new relaxation
    $MinRS = MinRS \cup \{PSX\}$
endfor
return MinRS
```

Fig. 4. Algorithm for determining all minimal relaxations.

that have to be made is at least an order of magnitude lower than the theoretical upper bound.

The *space complexity* for storing the partial results is also strictly limited. For each of the individual atoms of the query, a list of matching products has to be stored. If there are p products and the query consists of a atoms, we need *exactly* $p*a$ bits for storing the raw information when using a representation based on bit-sets.

3.2. Finding preferred relaxations

Up to now, we have mostly assumed that relaxations of smaller size, i.e., those that contain fewer conditions to be removed from the query, are preferable for the user. In an interactive recommender application we therefore might pick one of the smallest relaxations and present it to the user. However, as noted in [15], the users might have different priorities with respect to product characteristics they are more willing to compromise. Such specific preferences can be incorporated into our approach by associating *costs* (of relaxation) with each atom of the query and defining an overall cost function that for instance takes both the number of atoms in the relaxation and these individual costs into account.

Definition 8 *(Cost function) Let Q be a failing query and R a valid relaxation for Q consisting of the atoms* $a_1, ..., a_k$*. If ICOSTS is a function that associates a positive integer number with each* a_i *expressing the individual costs of relaxing atom* a_i*, the overall costs for R for Q can be described by any function* $COSTS(Q, R, ICOSTS)$ *that returns a positive integer number expressing the overall costs of R. In addition it has to hold that*

$$COSTS(Q, R', ICOSTS) < COSTS(Q, R, ICOSTS)$$

if $R' \subset R$ *thus ensuring that adding more atoms to a relaxation can only increase its cost.*

Given such a cost function, we can define an ordering between the possible relaxations and describe the properties of an *optimal* relaxation.

Definition 9 *(Optimal relaxation): Given a failing query Q, a valid relaxation R for Q is said to be optimal, if it is minimal and there exists no other valid relaxation* R' *for which*

$$COSTS(Q, R', ICOSTS) < COSTS(Q, R, ICOSTS)$$

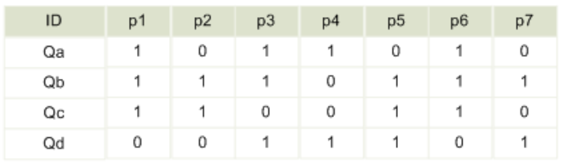
Given a cost function that obeys the abovementioned reasonable characteristics, determining the optimal relaxation in our PSX representation can be easily done by scanning the set of PSXs, evaluating the cost function individually, and remembering the PSX that minimizes this cost function. Similarly, the "top-n" relaxations can be memorized during this single run.

3.3. Preferred relaxations with at least n products

In many practical applications, it is desirable to present more than one single item to the user such that the user can compare these items. However, when using $MinRelax$ it is only guaranteed that at least one product will be in the result set. In the following section we show how we can compute relaxations that result in *at least n* products to be proposed while at the same time the relaxation costs are minimized. The idea of our second algorithm $NRelax$ is to determine such relaxations solely on the basis of the already given in-memory data structures.

We use the example shown in Figure 5 (with seven products and four constraints Qa to Qd) to illustrate the $NRelax$ algorithm. The example starts with the partial query results (the results matrix); the list of product-specific relaxations for Figure 5 is as follows:

PSXs = ⟨ {Qd}, {Qa, Qd}, {Qc},
{Qb, Qc}, {Qa}, {Qd}, {Qa, Qc} ⟩



ID	p1	p2	p3	p4	p5	p6	p7
Qa	1	0	1	1	0	1	0
Qb	1	1	1	0	1	1	1
Qc	1	1	0	0	1	1	0
Qd	0	0	1	1	1	0	1

Fig. 5. Partial results for a query with 4 constraints and a database containing 7 items.

Consider the following example. Let us assume that we are interested in a single relaxation that leads to at least 3 items in the result set. As can be seen in Figure 5, no singleton relaxation will lead to this desired effects. Relaxing for instance Qa will lead to the result set {p5}, relaxing Qd to result set $\{p1, p6\}$. The relaxation $\{Qa, Qc\}$ leading to result set $\{p3, p5, p7\}$ will however satisfy the requirement on the result set size.

Note that in order to ensure that a minimum number of items is included in the result set, one could follow a simple strategy in which the number of satisfied constraints by each product is counted and the items are presented in decreasing order of constraint fulfillment. In the example, this would be the set $\{p1, p3, p5, p6\}$ since all of them satisfy three constraints. However, the problem is that in theory each element of the result set requires a different relaxation. Such situations are not only hard to explain to the end user (who will get a different explanation for every single item) but may also lead to recommendation lists that include items with strongly varying characteristics. Thus, based on our practical experiences, a different approach has to be chosen.

The algorithm $NRelax$ was designed to avoid such problems by calculating cost-optimal relaxations with a given result set size. $NRelax$ is described and exemplified in Figures 6 and 8 and works by analyzing the possible combinations of all product-specific relaxations (and does not require any additional database queries). Note that *NRelax* starts with the full list of the original *PSXs*, because using only the minimal relaxations would be insufficient for our purposes as the optimal relaxation for "at least n" products could be lost.

During the construction of the search space following data structure *RNode* will be used for storing costs and number of products associated with a search node.

```
struct RNode:
        atoms: List of atoms of PSX
        cost: costs of node
        nbProducts: number of products
        closed: flag, if node was closed
endstruct
```

During the algorithm's search process, a list ("agenda") of open nodes is maintained which contains the combinations that still have to be explored (see also the algorithm listing). The recursive calls of $NRelaxInt$ for the example (with threshold $minp = 10$) are illustrated in Figure 7.

sAs shown in Figure 8, the algorithm starts with the initial list of *PSXs* and systematically constructs the possible combinations in order of their cardinality. In the example, we assume the existence of a cost model for user constraints. If there exists no such model, we can assign equal costs to all constraints. When the algorithm proceeds and two *PSXs* are to be combined, the union of the involved atoms is generated, the number of products for the relaxation is determined, and the corresponding cost function is calculated. When a combination is found that leads to enough products, we remember the cost value and subsequently prune the search space by removing combinations that cannot lead to a better result.

In the following detailed example we shall use a simple cost function shown in the following table.

Constraint	Cost
Qa	1
Qb	2
Qc	3
Qd	4

As a first step, we compress the original *PSX* list to a collapsed list *CPSX*. We remove any duplicates from the list of *PSXs* and annotate each new element *CPSX* with (a) the corresponding costs and (b) with the number of products that will result from that relaxation. Then we sort the elements in increasing order of the costs in order to reach maximal pruning. Determining the number of products for a certain relaxation can be done by checking for subset relations in the original *PSXs*: PSX $\{Qa, Qc\}$ will for instance have costs of $1 + 3 = 4$ and will result in 3 products. Technically, the set of products for the relaxation $\{Qa, Qc\}$ can be efficiently determined by applying a bit-wise AND-combination of the remaining PSX vectors and by the counting non-zero entries in the result.

The collapsed list *CPSX* of our example is therefore as follows. We use a notation where $\{Qc\}(3/1)$ denotes that the relaxation Qc has costs of 3 and results in one product.

CPSX = ⟨ {Qa} (1/1), {Qc} (3/1), {Qd} (4/2), {Qa,Qc} (4/3), {Qa, Qd} (5/4), {Qb, Qc} (5/2) ⟩

Starting with this initial list, we now analyze the combinations of the elements of *CPSX*. Figure 6 illustrates how the combinations of the example are generated and how the search space can be pruned. The different aspects of the algorithm are marked with numbers in the diagram: At (1), a new node {Qa,Qc} is constructed from {Qa} and {Qc}. At (2), the node {Qa,Qc} from the open agenda (at the first level in Figure 6, see Algorithm 8) can be closed as it will be further explored in the next round of expansion. In Figure 6 this fact is indicated with a rhombus symbol. At (3), the successor of {Qa} and {Qa,Qc} would be {Qa,Qc}. However, we already added that node to the new agenda in (1) and can ignore it for further exploration, i.e., we do not add the node again. Nodes that are pruned from the future search space in this way are marked with an "x". At (4) we have found a relaxation that comprises all possible atoms, which means that no further expansion is possible.

Regarding the size of the search space, note that the number of nodes on the top level (and more importantly, overall) is limited to the number of possible relaxations for the given query Q, i.e. if $|Q| = m$, there can only be $2^m - 1$ nodes in the worst case, independent of the number of products in the catalog. If there are already $2^m - 1$ different nodes on the first level, no further expansion will be required since all possible combinations are already contained in this first level[1].

All the computations can be done on the basis of the pre-evaluated partial results and do not require any further database queries. Also, compared with an approach that works by constructing all 2^m combinations of all atoms of the original query, we can also restrict the search space based on the already existing partial results and can leave out those that will definitely not lead to more products. In our example, {Qb} is not a product-specific relaxation and we therefore will never consider superset combinations like {Qa,Qb}, {Qb,Qd} and so forth, since we already now that no additional product will be in the result set when adding {Qb} alone. The completeness of the algorithm is guaranteed by the systematic construction of all combinations of the given *PSXs*.

In practical and more complex examples, the described cost-based tree pruning techniques significantly reduces the number of nodes to be explored. Therefore, in these more complex settings, only small fractions of the theoretical search space will be explored. If we for instance search for a relaxation with at least 3 products for the given example, we will find {Qa,Qc} to be the best relaxation in the original agenda and will not have to add a second-level element to the agenda due to the already known cost-optimality of the node.

[1]Note that Figure 6 has more than $2^4 - 1 = 15$ nodes, because we left the pruned nodes in the diagram for better understandability of the algorithm.

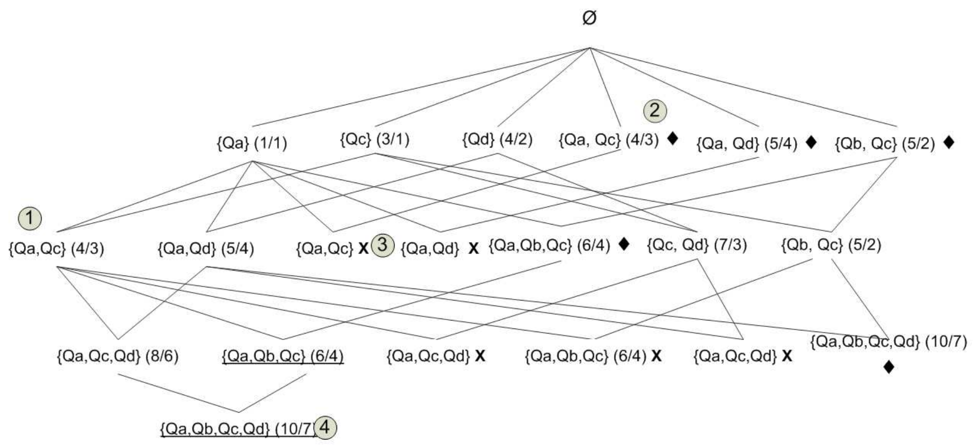
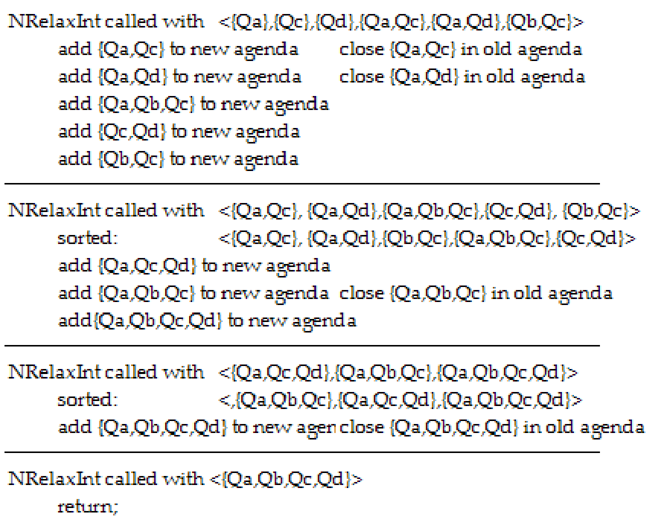


Fig. 6. Searching for at least n products.



```
NRelaxInt called with   <{Qa},{Qc},{Qd},{Qa,Qc},{Qa,Qd},{Qb,Qc}>
        add {Qa,Qc} to new agenda        close {Qa,Qc} in old agenda
        add {Qa,Qd} to new agenda        close {Qa,Qd} in old agenda
        add {Qa,Qb,Qc} to new agenda
        add {Qc,Qd} to new agenda
        add {Qb,Qc} to new agenda
```

```
NRelaxInt called with   <{Qa,Qc}, {Qa,Qd},{Qa,Qb,Qc},{Qc,Qd}, {Qb,Qc}>
        sorted:         <{Qa,Qc}, {Qa,Qd},{Qb,Qc},{Qa,Qb,Qc},{Qc,Qd}>
        add {Qa,Qc,Qd} to new agenda
        add {Qa,Qb,Qc} to new agenda  close {Qa,Qb,Qc} in old agenda
        add{Qa,Qb,Qc,Qd} to new agenda
```

```
NRelaxInt called with   <{Qa,Qc,Qd},{Qa,Qb,Qc},{Qa,Qb,Qc,Qd}>
        sorted:         <,{Qa,Qb,Qc},{Qa,Qc,Qd},{Qa,Qb,Qc,Qd}>
        add {Qa,Qb,Qc,Qd} to new agen close {Qa,Qb,Qc,Qd} in old agenda
```

```
NRelaxInt called with <{Qa,Qb,Qc,Qd}>
        return;
```

Fig. 7. Recursive calls in *NRelax*.

4. Interactive failure recovery

The algorithms presented so far are suitable for non-interactive recommendation scenarios in which the system automatically relaxes some of the user's constraints when no matching item can be found. As an alternative one could however also let the user interactively decide on which requirements he or she is willing to compromise. For example, an algorithm for incremental relaxation is proposed in [15]. The key idea is to find minimal subsets of the user constraints (conflicts) which cause the result set to become empty.

Definition 10 *(Minimal Failing Subquery – MFS): A failing subquery Q^* of a given query Q is a minimal failing subquery of Q if no proper subquery of Q^* is a failing query.*

When given a failing query Q, the algorithm from [15] calculates *all* MFSs (conflicts) contained in Q before the interactive relaxation procedure starts. In this interactive setting, the system incrementally presents conflicts to the user, who can then decide on which constraint to relax. This cycle is repeated until all MFSs in the requirements are resolved and the result set is no longer empty.

However, determining *all* MFS is a computationally complex task; see [8] for an in-depth anal-

```
ALGORITHM: NRelax
In: A set of product-specific relaxations $PSXs$, threshold $minp$
Out: An optimal relaxation containing at least $minp$ products
$CPSX$ = Collapse $PSXs$ as sequence of $RNodes$
$bestNode$ = new $RNode$ with infinite costs.
return NRelaxInt(CPSX,minp, bestNode)

function NRelaxInt(agenda,minp, bestNode)
In:  agenda: Sequence of RNodes to explore, minp: threshold,
     bestNode: currently best node
Out:  An optimal relaxation for minp products
if $|agenda| = 0$ then return bestNode.atoms endif
Sort agenda by increasing costs
% Check for new optimum
$newBest$ = node from agenda with lowest costs for which $node.costs$
            are lower than $bestNode.costs$ and nbProducts $> n$
if $newBest \neq null$ then $bestNode = newBest$ endif
% Nothing more to combine
if $|agenda| = 1$ then return $bestNode.atoms$ endif
% Set up new agenda
$newAgenda = <>$
% Combine elements of given agenda
for i=0 to $|agenda| - 1$
    for j=i+1 to $|agenda|$
     $n1$ = agenda[i]
     $n2$ = agenda[j]
     % ignore closed nodes
     if n1.closed or n2.closed then continue with next $j$ endif
     $newNode$ = combine atoms, costs, products of $n1$ and $n2$
     if not exists $n \in newAgenda$ where $n.atoms = newNode.atoms$ then
      Close nodes $n$ in agenda for which $n.atoms=newNode.atoms$
      if newNode.costs < bestNode.costs then
         add $newNode$ to $newAgenda$
      endif
     endif
   endfor
endfor
return NRelaxInt(newAgenda, minp, bestNode)
```

Fig. 8. Algorithm for finding relaxation with a minimum number of products in the result set.

ysis of the complexity of enumerating one, some, or all MFSs of a query. This in turn means that an approach based on an in-advance computation of all MFSs might not be applicable in interactive settings except for very small problem sizes.

We therefore propose to apply Junker's recent QuickXPlain [13] algorithm for computing MFSs *on demand*: The overall scenario is that when we have a failing query, we aim at finding *one preferred* and minimal conflict (instead of all) in these requirements and let the user decide how to proceed.

Preferred means that we shall try to identify those conflicts (among the possibly many conflicts which we cannot present to the user anyway) that contain requirements for which we assume that a typical user might be willing to compromise. In the digital camera domain we could for instance assume or learn that experts in digital photography searching for cameras supporting "Firewire" connectivity are more willing to compromise on the price than on the technical requirement. In practice, the estimated cost (priority) values for each requirement can either be annotated in advance or they can be learned from the interaction history of different users over time.

Originally, QuickXPlain was developed for finding conflicts (corresponding to MFSs) in constraint satisfaction problems, but its general, non-intrusive nature allows us to adapt it for our purposes (Figure 9). QuickXPlain is based on a *divide-and-conquer* strategy: In the decomposition phase it partitions the problem into subproblems of smaller size (thus pruning irrelevant parts of the problem) and subsequently tries to re-add individual elements and checks for consistency while at the same time taking preferences into account.

Depending on n, the number of atoms in the query, the size of the preferred conflict k, and the splitting point (e.g., $n/2$), QuickXPlain in the best case only needs $log(n/k) + 2k$ consistency checks (database queries in our case) and $2k * log(n/k) + 2k$ in the worst case [13].

When we consider our initial example from Section 2, we see that there are three minimal conflicts according to Definition 10 in the requirements, i.e.,

$$\{\text{usb} = \text{true, firewire} = \text{true}\},$$
$$\{\text{firewire} = \text{true, price} \le 300\},$$
$$\{\text{price} \le 300\text{, resolution} \ge 5\text{MP}\}.$$

Let us assume that the partial order of costs $\prec$ (in the sense of [13]) among the individual user constraints is "*price* $\prec$ *firewire* $\prec$ *usb* $\prec$ *resolution*", meaning that compromises are preferred to be made on the price than on firewire and so forth. Thus, if we were to choose which one of the three conflicts we should present the user to resolve, we would select the middle one, because it only comprises constraints of which we assume that they induce little costs of compromise for the user.

Our adapted QuickXPlain algorithm listed in Figure 9 will find the preferred, minimal conflict as follows. First, it immediately partitions the constraints on price, firewire, USB, and resolution (denoted as P,F,U and R for short) into the subsets {P,F} and {U,R}. In the first recursive call, the algorithm will detect that {P,F} contains conflicting requirements and thus proceeds by further analyzing this subset alone, which means that half of the atoms from the original query will not be further taken into consideration in subsequent steps.

```
ALGORITHM: mfsQX
In: A failing query Q
Out: A preferred conflict of $Q$
$A$ = sorted list of atoms of $Q$
return $mfsQI(\emptyset, A)$

function mfsQI (BG, A)
In: BG: List of atoms in background
    A: List of atoms of failing query
Out: A preferred conflict of $Q$
    % Construct and check the current
    % set of atoms
    $query = \bigwedge_{b \in BG}(b)$
    if query is not successful
        return $\emptyset$
    endif
    if $|A| = 1$
        return $A$
    endif
    % Split remaining atoms into two parts
    $C1 = \{a_i \in A | i < (|A|/2)\}$
    $C2 = A \setminus C1$
    % Evaluate branches
    $\Delta_1 = mfsQI(BG \cup C1, C2)$
    $\Delta_2 = mfsQI(BG \cup \Delta_1, C1)$
    return $\Delta_1 \cup \Delta_2$
```

Fig. 9. Using QuickXPlain for computing preferred MFS.

Next, QuickXPlain proceeds with the next sets of atoms {P} and {F} in our example and can immediately determine that both {P} and {F} have to be in the minimal conflict, since the number of the remaining atoms is $|A| = 1$ in both cases. Thus, the algorithm returns the preferred conflict {P,F}.

A general algorithm that shows how QuickXPlain can be integrated into an interactive relaxation procedure is sketched in Figure 10. When a retrieval failure occurs, the algorithm computes the first conflict (MFS) and uses the conflict elements as choice point for future backtracking, which is comparable to the basic backtracking procedure for Constraint Satisfaction Problems in [23]. The conflict is presented to the user who can decide to either make a compromise on one of the constraints in the conflict or go back to the previous set of choices. If the user selects one constraint to relax, the algorithm removes this constraint from the query and recursively checks if the remaining query (now without the removed con-

```
ALGORITHM: InteractiveRelax
In: Sorted list of atoms $A$ of failing query $Q$
query = $\bigwedge_{a \in A}(a)$
if query is not successful
    % Compute a minimal preferred conflict
    conflict = mfsQX($\emptyset$, A)
    remaining = conflict
    % Set up the choice points
    do $|conflict|$ times
        choice = Ask user to select an option
                 from remaining or 'backtrack'
        if choice = 'backtrack' return
        remaining = remaining $\setminus$ {choice}}
        % Remove the choice and try again
        interactiveRelax(A $\setminus$ {choice})
    end do
else
    Minimize relaxation and compute results
    Report success and show proposal to user.
    response = Ask user if happy with result
    if response = 'yes'
        exit function
    % backtrack to last choice point
    else return
```

Fig. 10. Basic algorithm for interactive relaxation.

straint) is successful or not. If the query is still not successful, the next conflict is generated.

If during the procedure the query becomes successful and all conflicts are resolved, one could directly present the set of matching items to the user. However, as the relaxation set that results from the interactive procedure might be non-minimal, we propose to first "minimize" the relaxation and remove all unnecessary compromises. This step can be done in linear time with respect to the number of elements in the relaxation [13]. The optimized result can then be presented to the user. Still, if the user does not like the proposal, he can go back to the last choice point and revise his decision.

Note that with the help of conflicts computed with QuickXPlain we can also compute the set of minimal or optimal relaxations based on Reiter's Hitting-Set Algorithm [19]. This can be seen as an alternative to our approach based on product-specific relaxations from the previous section. The run-time performance of such an adapted Hitting-Set algorithm has been evaluated in [12] for different problem instances. The results showed that even if we do not rely on the pre-computation of the *PSXs*, no more than one second is required for finding the optimal relaxation at least for medium-sized instances. However, based on the full calculation of the Hitting Sets as proposed in [12], also other schemes for interactive query relaxation as described in [15] become possible. When knowing the full space of possible relaxation in advance, the user can be for instance interactively guided to relaxations with lower cardinality. Our practical experiences however indicate that there is some risk that such rich user interfaces with many options to proceed can quickly overwhelm the average web user.

A different approach to guiding the end user in the interactive preference elicitation process was proposed by Bridge and Ricci in [1]. In their approach, a system that monitors the user's action in a "query-editing" item retrieval scheme is designed. In this interaction scheme, the end user is allowed to repeatedly modify the query in order to narrow down the results to the set of desired products. Based on the observed behavior, the proposed system learns the user's utility model and then generates advices for the user about how to proceed, i.e., what to try next. To some extent this work is similar to ours as (a) their main goal is to help the user reduce the number of required interactions to find suitable products and (b) query relaxation is seen as one possible query-editing move, which could be implemented based on the techniques described in this paper. In our approach, however, we do not make any assumptions on how the cost model was acquired as described in Section 3. Learning the user's utility model (i.e., cost model) automatically can be seen as one possible option.

Finally, also other approaches to retrieval failure recovery, based e.g. on query revision or similarity-based retrieval are also possible; some of them are described in [6] and [7]. This discussion of these aspects however is beyond the scope of our paper that focuses on improving the efficiency of relaxation-based approaches.

5. Practical results

After having evaluated the algorithms analytically in the previous sections, we will finally report some practical experiences with the proposed techniques, which have been implemented in the Advisor Suite system, a knowledge-based framework for the development of interactive recommender applications; see [5,9].

Within the Advisor Suite framework the initial set of products to be presented to the user is determined with the help of "if-then-style" *filter rules* that relate customer requirements with product characteristics. This indirection allows us to implement a more user-oriented interaction view, such that (non-experienced) users do not have to be questioned directly about desired product characteristics. If we consider the example from Section 2, we would for instance not ask the user about his need for "Firewire" support, but rather try to find out what his mobility and connectivity requirements are. Typical *filter rules* for our example problem could be the following:

F1: **if** high-quality-printouts are required **then** only recommend cameras with a resolution ≥ 5

F2: **if** user entered price limit L **then** only recommend cameras that cost less than L

F3: **if** user needs high connectivity **then** only recommend cameras that support "Firewire"

At run-time, when the requirements have been elicited and a product proposal has to be generated, the *filter rules* are evaluated by the system. For each rule it is determined whether the condition in the antecedent of the rule is satisfied, i.e., whether the filter rule is *active* or not. The combination of the conclusions of the active rules form the conjunctive query to the catalog. Note that the conclusions of the rules can contain arbitrary complex expressions on product characteristics, e.g., consisting of several conjunctions and disjunctions. This modular way of modeling recommendation rules also forms the basis for splitting up the conjunctive query into individual atoms for relaxation in a natural way. The filter rules themselves are modeled in Advisor Suite with the help of graphical editing tools (see Figure 11). Each rule can also be annotated with explanatory texts which are presented to the user in the explanation phase: a text can be maintained both for the case when the rule was successfully applied as well as for the case that the rule had to be relaxed. Finally, we can define an a-priori *priority value* for each rule, which corresponds to the costs of relaxing the rule when no product satisfies all requirements. In practical applications, these priority values are defined by the domain expert who for instance knows that typical customers rather compromise on the manufacturer of a camera than on other characteristics.

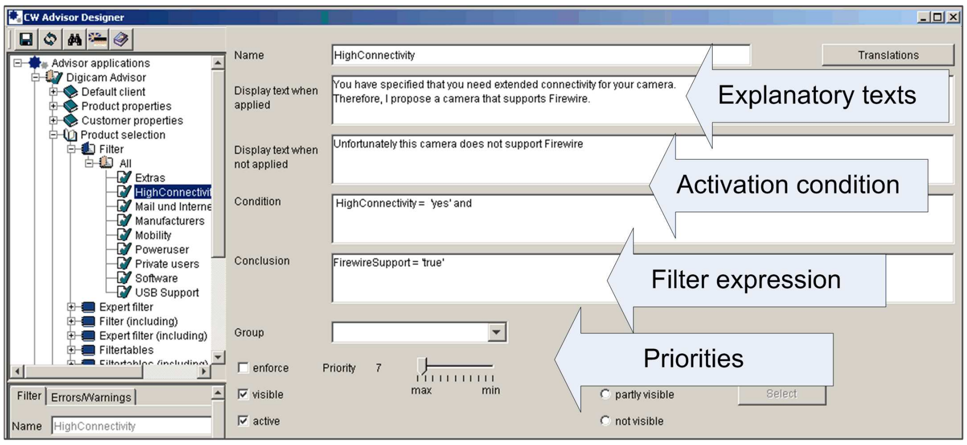
Up to now, about twenty different recommender applications for various domains have been built with Advisor Suite and have been successfully deployed in commercial settings, which gives us in particular a good impression of the size and complexity of realistic knowledge bases:

– The number of products available in the catalog typically ranges from a few dozen to several hundred.
– The number of filter rules remains manageable, i.e., only a few dozen rules were required in nearly all cases.
– Many of the rules are mutually exclusive with regard to their activation condition, i.e., only some of the rules are actually *active* when relaxation has to be done. The resulting query lengths encountered in practical settings depend on various factors and the given application scenario. For advisory sessions in which novice users are asked about functional requirements of the products, query lengths of around 10 to 15 constraints are common. For in-depth requirements specification screens for expert users up to 20 or 30 parameters (constraints) can be specified by the user.

The first aspect we have evaluated are the running times for determining (optimal) relaxations. Advisor Suite is a Java-based system that operates on top of standard relational database systems; all tests and measurements have been performed on standard desktop PCs with a Pentium M 2 GHz processor and 512 megabytes of RAM.

The most costly operation when determining relaxations in our approach are the queries that are required to compute the result sets for the individual filters, i.e., for each *active* filter, exactly one query has to be executed. The query time mostly depends on the number of products in the catalog and to a smaller extent on the complexity of the query. On average, a single query takes around 5 milliseconds in test cases with around 500 products.

If we assume that we have a knowledge base containing 70 filter rules, and 35 of them are active



Name: HighConnectivity

Display text when applied: You have specified that you need extended connectivity for your camera. Therefore, I propose a camera that supports Firewire.

Display text when not applied: Unfortunately this camera does not support Firewire

Condition: HighConnectivity = 'yes' and

Conclusion: FirewireSupport = 'true'

Group:

☐ enforce Priority 7 max min

☑ visible

☑ active

○ partly visible

○ not visible

Name: HighConnectivity

Fig. 11. Graphical editing tool for filter rules.

– which is a problem of above average complexity in realistic settings – the time for computing the individual filter results (partial query results) is $35 * 5\ ms = 175\ ms$, which is well within our targeted time frame of less than one second.

In Advisor Suite, the results of the individual filter rules are represented in memory in the form of *Java BitSets*, which means that (a) the memory requirement for the raw data is limited to *NumberOfProducts * NumberOfActiveFilters* bits, and (b) that the analysis of the partial results can be efficiently done based on fast bit-set operations. In general, the usage of such specific data structures is not mandatory but rather a further optimization in our implementation.

In all recommender applications that are in productive use, our strategy is to initially compute only one *optimal* relaxation with respect to size and relaxation costs, which is subsequently used to explain the proposal to the user. The time needed for determining this optimum depends on the complexity of the cost function and the number of products. However, these operations can be performed in-memory and in our test cases they required about a tenth of the query time (e.g., 17.5 ms in the above-mentioned example).

Regarding query time, we also exploited a feature of the Advisor Suite approach of modeling *filter rules*. In many cases, the consequent of the rules contains no "variables' (e.g., *"attribute usb must be true'*). Therefore, we can pre-compute and cache the partial results for such rules in advance, e.g., when the recommendation server is started up. In addition, these partial results can also be shared among different recommendation sessions of different users, since the partial results remain stable as long as the filter rule is not changed and the set of products is the same. Such pre-computation, however, is not possible if the consequent contains variables, as in filter rule $F2$ in the example above which takes the user input with respect to allowed costs directly into account. Still, our experiences from different application domains show that the most of the filter rules do not contain such variables, which means that the number of needed queries per relaxation task can be significantly reduced. Overall, even if no such pre-computation was done, in none of our test cases more than 500 ms were required to find the optimal relaxation.

Regarding usability and end-user acceptance of a relaxation and explanation facility, a first survey was also conducted in the context of a commercial digital camera advisor built for Austria's largest e-commerce site [25]. The evaluation of around 1,600 feedback forms showed that these extended relaxation capabilities of the recommender system were particularly appreciated by end users as additional features which cannot be found in other systems.

6. Conclusions

We have presented new techniques and algorithms for fast query relaxation for content-based recommender systems that particularly aim at minimizing the number of required database queries and take a-priori or learned preferences into account.

Based on the initial work and formalisms from [15,20] and [8] we have shown how we can efficiently determine preferred relaxations for a failing query by evaluating the user constraints individually in a pre-processing step and base subsequent calculations on such compact in-memory data structures.

In addition, we proposed a general procedure for interactive and incremental query relaxation which is based on the on-demand computation of preferred conflicts and the usage of a recent, general-purpose conflict detection algorithm.

References

[1] Derek Bridge and Francesco Ricci. Supporting product selection with query editing recommendations. In Joseph A. Konstan, John Riedl, and Barry Smyth, editors, *Procs. of the First ACM Conference on Recommender Systems*, pages 65–72. ACM Press, 2007.

[2] R.D. Burke, K.J. Hammond, and B.C. Yound. The findme approach to assisted browsing. *IEEE Expert*, 12(4):32–40, Jul/Aug 1997.

[3] Robin Burke. The wasabi personal shopper: a case-based recommender system. In *Proceedings of the 16th AAAI/11th IAAI*, pages 844–849, Orlando, Florida, United States, 1999.

[4] Li Chen and Pearl Pu. Hybrid critiquing-based recommender systems. In *Proceedings of the 12th international conference on Intelligent User Interfaces, IUI'07*, pages 22–31, Honolulu, Hawaii, USA, 2007.

[5] Alexander Felfernig, Gerhard Friedrich, Dietmar Jannach, and Markus Zanker. An integrated environment for the development of knowledge-based recommender applications. *International Journal of Electronic Commerce*, 11(2):11–34, Winter 2006-7 2007.

[6] A. Ferguson and D. Bridge. Options for query revision when interacting with case retrieval systems. In *Procs. of Fourth UK Case-Based Reasoning Workshop*, pages 1–17, 1999.

[7] Alex Ferguson and Derek Bridge. Options for query revision when interacting with case retrieval systems. *Expert Update*, 3(1):16–27, 2000.

[8] P. Godfrey. Minimization in cooperative response to failing database queries. *International Journal of Cooperative Information Systems*, 6(2):95–149, 1997.

[9] D. Jannach. Advisor Suite - a knowledge-based sales advisory system. In R. Lopez de Mantaras and L. Saitta, editors, *Proceedings of 16th European Conference on Artificial Intelligence*, pages 720–724, Valencia, Spain, 2004. IOS Press.

[10] D. Jannach. Finding preferred query relaxations in content-based recommenders. In *Proceedings of 3rd IEEE Intelligent Systems Conference IS'2006*, pages 355–360, Westminster, UK, 2006. IEEE Press.

[11] D. Jannach. Techniques for fast query relaxation in content-based recommender systems. In C. Freksa, M. Kohlhase, and K. Schill, editors, *KI 2006 - 29th German Conference on AI*, pages 49–63, Bremen, Germany, 2006. Springer LNAI 4314.

[12] D. Jannach and J. Liegl. Conflict-directed relaxation of constraints in content-based recommender systems. In *Proc. 19th International Conference on Industrial, Engineering & Other Applications of Applied Intelligent Systems (IEA/AIE'06)*, pages 166–176, Annecy, France, 2006. Springer LNAI 4031.

[13] U. Junker. Quickxplain: Preferred explanations and relaxations for over-constrained problems. In *Proceedings of National Conference on Artificial Intelligence - AAAI'04*, pages 167–172, San Jose, 2004. AAAI Press.

[14] D. McSherry. Explanation of retrieval mismatches in recommender system dialogues. In *Proceedings of ICCBR Workshop on Mixed-Initiative Case-Based Reasoning*, pages 191–199, Trondheim, 2003.

[15] D. McSherry. Incremental relaxation of unsuccessful queries. In P. Funk and P.A. Gonzalez Calero, editors, *Proceedings of the 6th European Conference on Case-based Reasoning*, pages 331–345. Springer LNAI 3155, 2004.

[16] D. McSherry. Maximally successful relaxations of unsuccessful queries. In *Proceedings of the 15th Conference on Artificial Intelligence and Cognitive Science*, pages 127–136, Castlebar, Ireland, 2004.

[17] D. McSherry. Retrieval failure and recovery in recommender systems. *Artificial Intelligence Review*, 24(3/4):319–338, 2005.

[18] James Reilly, Kevin McCarthy, Lorraine McGinty, and Barry Smyth. Dynamic critiquing. In *Proccedings of 7th European Conference on Case-Based Reasoning*, pages 763–777, 2004.

[19] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987.

[20] F. Ricci, N. Mirzadeh, and M. Bansal. Supporting user query relaxation in a recommender system. In *Proceedings of the* 5^{th} *International Conference in E-Commerce and Web-Technologies - EC-Web*, pages 31–40, Zaragoza, Spain, 2004.

[21] Francesco Ricci, Bora Arslan, Nader Mirzadeh, and Adriano Venturini. ITR: A case-based travel advisory system. In *Proceedings of 6th European Conference on Case Based reasoning*, pages 613–627, 2002.

[22] Cynthia A. Thompson, Mehmet H. Gker, and Pat Langley. A personalized system for conversational recommendations. *Journal of Artificial Intelligence Research*, 21:393–428, 2002.

[23] Edward Tsang. *Foundations of Constraint Satisfaction.* Academic Press Limited, UK, 1993.

[24] Paolo Viappiani, Pearl Pu, and Boi Faltings. Conversational recommenders with adaptive suggestions. In *Proceedings 2007 ACM conference on Recommender systems*, pages 89–96, Minneapolis, MN, USA, 2007.

[25] Markus Zanker and Christian Russ. Geizhals.at: vom preisvergleich zur e-commerce serviceplattform. *in: S. M. Salment and M. Gröschel (ed): Handbuch Electronic Customer Care*, 2004.