International Journal on Artificial Intelligence Tools
Vol. 20, No. 2 (2011) 297–312

DOI: 10.1142/S0218213011000164

World Scientific
www.worldscientific.com

BFX: DIAGNOSING CONFLICTING REQUIREMENTS IN CONSTRAINT-BASED RECOMMENDATION

MONIKA SCHUBERT and ALEXANDER FELFERNIG

Applied Software Engineering, IST, Graz University of Technology
Inffeldgasse 16b/II 8010 Graz, Austria
monika.schubert@ist.tugraz.at
alexander.felfernig@ist.tugraz.at

When interacting with constraint-based recommender applications, users describe their preferences with the goal of identifying the products that fit their wishes and needs. In such a scenario, users are repeatedly adapting and changing their requirements. As a consequence, situations occur where none of the products completely fulfils the given set of requirements and users need a support in terms of an indicator of minimal sets of requirements that need to be changed in order to be able to find a recommendation. The identification of such minimal sets relies heavily on the existence of (minimal) conflict sets. In this paper we introduce BFX (Boosted FastXplain), a conflict detection algorithm which exploits the basic structural properties of constraint-based recommendation problems. BFX shows a significantly better performance compared to existing conflict detection algorithms. In order to demonstrate the performance of BFX, we report the results of a comparative performance evaluation.

Keywords: Constraint-based recommender systems; conflict detection; automated query adaptation.

1. Introduction

Recommender systems are interactive applications that support users finding interesting items from a large range of products in a personalized way (see Burke[1]). These systems are becoming more and more popular due to the increasing size and complexity of product assortments offered by online selling environments. Widespread recommender applications use concepts of collaborative filtering (see Konstan *et al.*[11]) or content-based filtering (see Pazzani *et al.*[17]). These approaches are well applicable for simple products such as news, movies, music and so on. For applications where the users refine their requirements in an interactive process (e.g. in computers, mobile phones, financial services or holiday packages) constraint-based recommenders are used (see Felfernig *et al.*[2]). These recommenders support the identification of items (products and services) based on a given set of explicitly defined requirements, knowledge about the item assortment, and knowledge about which items should be selected/recommended in which context (see Burke[1]). For the purposes of this paper we will assume that the product assortment is stored

in a database table and customer requirements are represented as selection criteria (constraints). These constraints can be seen as part of a conjunctive query. This is a basic approach in the field of constraint-based recommendation.

Typically, we are interested in recommendations that fulfil all specified requirements. In order to retrieve the products that fulfil the given set of requirements from the product table, a corresponding conjunctive query is generated and executed on this table. The conjunctive query is a conjunction of the criteria derived from the specified set of customer requirements. If a query does not return any products, the system needs to calculate explanations (see Felfernig *et al.*[3,5] and O'Sullivan[16]) that indicate minimal sets of changes such that at least one item can be suggested to the user. Existing approaches to handle these changes focus on maximal succeeding sub-queries (see, e.g., McSherry[14]) or on minimal-cardinality diagnoses (see Felfernig *et al.*[3,5]).

The remainder of this paper is organized as follows. In Section 2 we introduce a introductory example from the domain of navigation systems. This example will be used throughout the paper to explain the basic properties of the BFX algorithm — the algorithm itself is introduced in Section 3. Evaluation results including a comparison of existing conflict detection algorithms are presented in Section 4. In Section 5 we discuss related work. We conclude the paper and give an impression on future work with Section 6.

2. Introductory Example

In this section we are introducing a working example which will be used for more demonstrative explanations throughout the paper. We are defining a product table (*navigationsystems*) which contains seven different navigation systems (see *navigation-systems* $= \{P_1, P_2, P_3, P_4, P_5, P_6, P_7\}$ in Table 1). Each of these navigation systems is described with the following attributes: *price, route saving, maps, touch screen* and *screen size*. The *price* specifies the costs of the navigation system. *Route saving* describes whether it is possible to save different routes in the system in order to be able to view them again, for example, when interacting the next time with the navigation system. The number of maps for different countries is specified by *maps*. The attributes *touch screen* and *screen size* describe the screen in more detail. The user can specify their requirements regarding each of these attributes.

Table 1. An example assortment of navigation systems (the product table *navigation-systems* $= \{P_1, P_2, P_3, P_4, P_5, P_6, P_7\}$).

	P_1	P_2	P_3	P_4	P_5	P_6	P_7
price	140	160	170	200	140	190	180
route saving	no	no	yes	yes	no	no	no
maps	5	6	2	10	4	1	5
touch screen	no	yes	yes	no	yes	no	yes
screen-size	4.3	3.5	4.2	4.3	3.5	4.1	4.3

Table 2. Example customer (user) requirements $C_R = \{c_1, c_2, c_3, c_4, c_5\}$. In this example the customer specifies their preferences for each attribute.

price	route-saving	maps	touchscreen	screen-size
< 150	yes	> 3	yes	> 4.0

Each attribute a_i has a corresponding domain $dom(a_i)$. For our example products, the attribute domains are: $dom(price) = \{140, 160, 170, 180, 190, 200\}$, $dom(route\text{-}saving) = \{\text{yes, no}\}$, $dom(maps) = \{1, 2, 4, 5, 6, 10\}$, $dom(touchscreen) = \{yes, no\}$ and $dom(screen\text{-}size) = \{3.5, 4.1, 4.2, 4.3\}$. To complete our working example, we also need a set of user requirements C_R: c_1: *price* < 150, c_2: *route savings* $=$ *yes*, c_3: *maps* > 3, c_4: *touch screen* $=$ *yes* and c_5: *screen size* > 4.0 (see Table 2). For this set of user requirements, a corresponding database query would be as follows: *SELECT * FROM* navigation-systems *WHERE* price < 150 *AND* route-savings $=$ *yes AND* maps > 3 *AND* touch-screen $=$ *yes AND* screen-size > 4.0.

When we are executing this query on Table 1, no product is returned by the query since there does not exist a navigation system which satisfies all user requirements. In such a situation we have to deal with the so-called *no solution could be found* dilemma. In order to tackle this challenge, Felfernig *et al.*[5] introduced an approach to support the user with repair proposals, which are minimal sets of changes that if accepted by the user, guarantee the identification of at least one item that completely satisfies the given set of requirements. An example for such a minimal set of changes is to inform the user about the possibility to change the route-saving option from *yes* to *no* and to change the touchscreen option from *yes* to *no*.

In order to be able to determine such minimal sets of changes, we need to calculate minimal conflict sets (e.g. Junker,[9] Mauss[12]). Such conflict sets have to be calculated efficiently, since we are dealing with interactive recommendation sessions where users are expecting an acceptable performance. In the following we are introducing the BFX algorithm for identifying minimal conflict sets. The algorithm manages to determine minimal conflict sets very efficiently which makes it extremely useful for interactive settings.

3. The BFX Algorithm

In this section we are going to explain the BFX (Boosted FastXplain) algorithm. This algorithm is an extension of the FastXplain algorithm that has been introduced in Schubert *et al.*[21] The performance improvements of BFX compared to the FastXplain[21] can be explained by the way minimal hitting sets are determined — within both algorithms. As a result of the BFX algorithm we can suggest adaptations to the user. In interactive settings these adaptations increases the usability as it helps the user finding a way out of the *no solution can be found* dilemma.

3.1. *Preliminaries*

Existing conflict detection algorithms (see Junker[9]) include an explicit consistency checking step. In general settings consistency checking is very costly and therefore should be avoided. On the other hand these consistency checks make the algorithm applicable for configurable products where the requirements of the user configure the product itself. Recommender systems often deal a predefined solution space such as a predefined set of products or items which is often represented in a product table. This results in a characteristic structural property of constraint-based recommendation problems namely that the set of products is finite and given a-priori.

Compared to existing conflict detection algorithms (see Junker[9] or Mauss[12]) which do not take this structural property into account, the BFX algorithm takes advantage from the knowledge about the underlying structure. The BFX algorithm exploits — due to a given set of predefined products — all hitting sets for a given set of customer requirements beforehand (for a detailed description see Jannach:[7] Table 3 shows the data structure used for representing the mentioned (not necessarily minimal) hitting sets. For example, the first column of Table 3 $(1, 0, 1, 0, 1)$ represents the hitting set $\{c_2, c_4\}$ which indicates that at least c_2 and c_4 have to be deleted or adapted in order to be able to identify at least one recommendation.

The standard approach to determine all conflict sets is to use an existing conflict detection algorithm (similar to QuickXplain[9] or Mauss[12]) and build a hitting set acyclic directed graph (see Reiter[18]). This graph is built by continuously relaxing the user requirements. A (minimal) conflict set is defined as follows (see Definition 1).

Definition 1 (Conflict Set). A conflict set is defined as a subset $CS = \{c_1, c_2, \ldots, c_m\} \subseteq C_R$ s.t. a query with the selection criteria specified in CS on the product table P does not result in a solution. CS is *minimal* if there exists no conflict set CS' which is a proper subset of CS i.e. $CS' \subset CS$.

In other words, a conflict set is a subset of the given user requirements such that none of the items in P satisfies all constraints in CS. The minimal conflict sets for our working example are $\{c_1, c_2\}$, $\{c_2, c_3, c_4\}$ and $\{c_1, c_4, c_5\}$.

In order to resolve all conflict sets a corresponding *hitting set directed acyclic graph* (*HSDAG*) (see Reiter[18]) can be constructed. In this graph the resolution of all minimal conflict sets automatically corresponds to the identification of all the existing minimal diagnoses. In our application context (constraint-based recommender applications), a *minimal diagnosis* is a minimal set of user requirements that has to be deleted (or adapted) from the C_R in order to retrieve at least one product from the product table. The HSDAG for our working example is shown in Figure 3 — the corresponding identified minimal diagnoses are $\{c_1, c_2\}$, $\{c_1, c_3\}$, $\{c_1, c_5\}$, $\{c_2, c_4\}$ and $\{c_2, c_5\}$.

Due to the possibility of representing the relationship between customer requirements and products in the data structure exemplified in Table 3, we can further improve the performance of the underlying diagnosis and conflict detection algorithms. In the following we will introduce the basic concepts of BFX which is an efficient algorithm for the determination of minimal conflict sets.

Table 3. Table of products and constraints with satisfaction values (1 if the user requirement (constraint) is satisfied, 0 otherwise).

	P_1	P_2	P_3	P_4	P_5	P_6	P_7
$c_1(price < 150)$	1	0	0	0	1	0	0
$c_2(routesaving)$	0	0	1	1	0	0	0
$c_3(maps > 3)$	1	1	0	1	1	0	1
$c_4(touchscreen)$	0	1	1	0	1	0	1
$c_5(screensize > 4.0)$	1	0	1	1	0	1	1

3.2. *Identification of minimal conflict sets*

The underlying data structure of our algorithm is a table representation of the relation between the constraints $C_R = \{c_1, c_2, ..., c_5\}$ and the products $P = \{P_1, P_2, ..., P_7\}$ (see Table 3). If a constraint is satisfied by an item, a 1 (true) is stored in the table. If the item can not fulfil the requirement (constraint) a 0 (false) is stored in the table. Table 3 shows this representation for our working example. This table must be recalculated when user changes their requirements and it has to be reduced if the user does not provide preferences for some of the product attributes.

	P_1	P_2	P_3	P_4	P_5	P_6	P_7	$\sum$
$c_1(price < 150)$	1	0	0	0	1	0	0	2
$c_2(routesaving)$	~~0~~	~~0~~	~~1~~	~~1~~	~~0~~	~~0~~	~~0~~	~~2~~
$c_3(maps > 3)$	1	1	0	1	1	0	1	5
$c_4(touchscreen)$	~~0~~	~~1~~	~~1~~	~~0~~	~~1~~	~~0~~	~~1~~	~~4~~
$c_5(screensize > 4.0)$	1	0	1	1	0	1	1	5
weight	6	9	7	6	7	13	4	

Fig. 1. Calculation of the total weight of the hitting set derived by product P_1 used for the BFX algorithm. This weight sums up all the row values which are part of the hitting set. The hitting set consists of c_2 and c_4, the row values are 2 for c_2 and 4 for c_4 and the sum is 6. Thus the weight of the hitting set $\{c_2, c_4\}$ is 6.

The BFX algorithm focuses on the calculation of minimal conflict sets on the basis of hitting sets that can be directly derived from the representation shown in Table 3. Such an algorithm is extremely useful for interactive recommendation

settings. The determined minimal conflicts can be directly presented to the user who is responsible for deciding which requirements should remain the same and which requirements should be changed (interactive repair scenario).In the remaining sections of the paper we will discuss in detail the BFX algorithm and its underlying ideas.

In order to calculate minimal conflict sets, FastXplain[21] (which is the foundation of BFX) exploits hitting sets from the product assortment and takes the one with the lowest cardinality (lowest number of constraints in the hitting set). In our working example this could be the hitting set $\{c_2, c_4\}$ extracted from product P_1. If there are more than one hitting set with the same cardinality, the first one is taken. In comparison to this the BFX calculates weights for the retrieval of the 'best' hitting set. For the calculation of this weight the first step is to calculate the sum of each row. In Figure 1 we can see that for the example given in Section 2 the sum of the first row is 2 ($r_1 = 2$), the second is 2 ($r_2 = 2$), the third is 5 ($r_3 = 5$), the forth is 4 ($r_4 = 4$) and the fifth is 5 ($r_5 = 5$). Based of this calculation we can calculate the total weight of each hitting set. This is done by summing up the row value (r_j) for each constraint that is part of the hitting set. For each constraint that is part of the hitting set the value (v_j) is 0 like in the table representation. For the other constraints the value (v_j) is 1 as in the table. On a more formal level we can say:

$$weight(HS_i) = \sum (1 - v_j) * r_j \, .$$

Figure 1 exemplifies this based on product P_1. The hitting set of P_1 is $\{c_2, c_4\}$. The value of the row c_2 is 2 and the value of the row c_4 is 4. Thus the weight of the hitting set $\{c_2, c_4\}$ is 6 which represents the weight of this hitting set alternative. We select the hitting set with the lowest weight. This is the hitting set $\{c_1, c_2\}$ for our working example. Based on the selected hitting set we can build a directed acyclic graph similar to the *hitting set acyclic direceted graph* — HSDAG (see Reiter[18]), where the nodes are sets of products and the edges are labeled with constraints (the graph of the example for the FastXplain is shown in Figure 2 and for the BFX the tree is shown in Figure 3). The root node consists of all products from the product table. After adding the edges labeled with the elements of the hitting set to the root node we calculate the product set for the children. Consequently, we exclude all products from the assortment which do not satisfy the constraint. For example for the constraint c_2 only the products $\{P_3, P_4\}$ fulfil this constraint.

In the next step we start with the reduced set of products of the first child (for the FastXplain this is c_2 and for the BFX it is the constraint c_1 in our working example). As a detailed illustration of the FastXplain algorithm is given in Schubert *et al.*[21] we are focusing only on the BFX algorithm in the following description. The hitting set with the lowest weight for the remaining product set $\{P_1, P_5\}$ is $\{c_2, c_4\}$. We add all constraints of this diagnosis to the tree. For every leaf in the tree we identify the remaining products of the set. When calculating the remaining

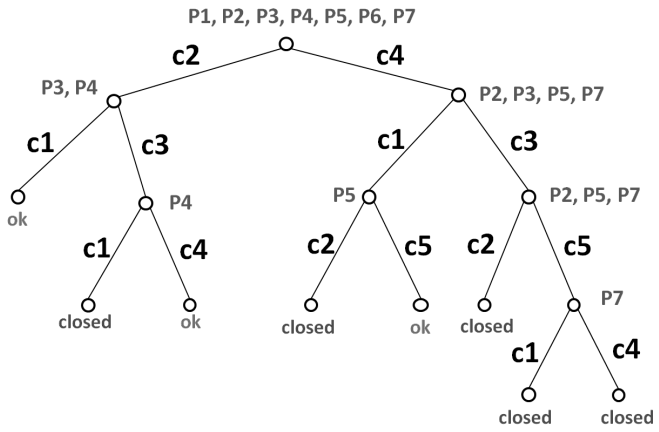

Fig. 2. Directed acyclic graph built to identify all minimal conflict sets using hitting sets. This graph is constructed by the algorithm FastXplain. The path to every leaf marked with *ok* is a minimal conflict set.

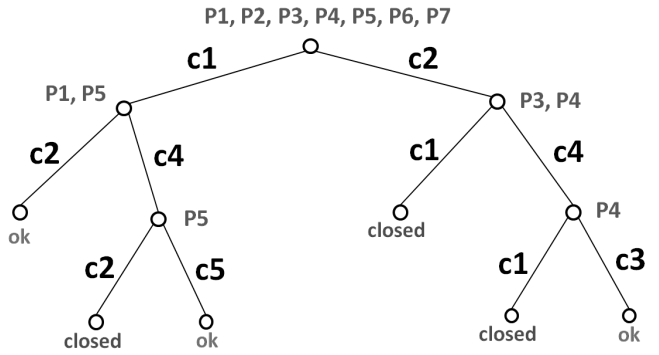

Fig. 3. Directed acyclic graph built to identify all minimal conflict sets using hitting sets. This graph is constructed by the algorithm BFX. The path to every leaf marked with *ok* is a minimal conflict set.

products for the branch $\{c_1, c_2\}$ no product is left. This means the path $\{c_1, c_2\}$ is a minimal set. We ensure the minimality of the set by performing a breadth-first search in the directed acyclic graph.

For calculating all minimal conflict sets, this method has to be continued until each leaf has either an empty set of remaining products or the path to the leaf is not minimal (the set of constraints of the path is a superset of another minimal conflict set). If we expand the tree further to calculate all minimal conflict sets, we would get the following minimal conflict sets as a result: $MCS = \{c_1, c_2\}$, $\{c_1, c_4, c_5\}$ and $\{c_2, c_4, c_3\}$.

When we take a look at Figure 2 and Figure 3 we can see that the directed acyclic graph calculated by the BFX algorithm (Figure 3) is smaller compared to

the one constructed by FastXplain[21] (Figure 2. This is based on the weight which tries to eliminate as many products as possible.

3.3. *The Algorithm*

In this section we dive into a more formal description of BFX. BFX was inspired by the FastXplain[21] and HSDAG[18] thus we keep the same level of description. The input values for the BFX are a root node for the tree and the product table (like Table 3). The root node is the main reference to the resulting tree which holds all information. The result of the algorithm (all minimal conflict sets) is stored in one global variable MCS, which is empty at the beginning.

Algorithm 1 BFX(root, p)

```
{Input: p - table of constraints and products}
{Input: root - the root node of the resulting tree}
{Global: MCS - set of all minimal conflict sets}
d ← getMinWeightHittingSet(p)
for all constraint c from d do
  p' ← reduce(c, p)
  child ← addChild(c, p')
  if p' = {} then
    child ← ok
    if path(child) ∉ MCS then
      MCS ← path(child)
      return
    end if
  end if
  if ∃ cs ∈ MCS : cs ⊆ path(child) then
    child ← closed
  end if
  if child ≠ closed then
    BFX(child, p')
  end if
end for
```

In the first step the algorithm calculates the hitting set with the lowest weight (*getMinWeightHittingSet*) as shown in Figure 1. In the example tree of Figure 3 this set is $\{c_1, c_2\}$. If there exists more than one diagnosis with the same weight the first one is chosen. For all constraints of this diagnosis a set of products (P') containing all products that satisfy this constraint is calculated (*reduced*). In our working example this is the set $\{P_1, P_5\}$ for the constraint c_1 and for the constraint c_2 it is $\{P_3, P_4\}$. Based on this reduced set of products combined with the constraint

c_i a node in the tree is created and added to the root node (*addChild*). If the set of products is empty a minimal conflict set (path from the root node of the tree to the child) is found. The minimality is ensured by the breadth-first search of the algorithm. If the path to the child is a minimal conflict set, the child is marked as "*ok*" and no further investigation is needed for this leaf. If this path is not an element of MCS (set of all minimal conflict sets) yet, then it is added.

If the path or a subset of it is already a minimal conflict set, then there is no need to expand the node anymore, because the conflict set found would not be minimal. In this situation the child is marked as "*closed*". In all other cases the tree is constructed further in breadth-first manner. In the example this expansion would be the node c_4 on the first level. When calculating all minimal conflicts sets a tree is created where all paths to the leafs marked with "*ok*" are minimal conflict sets. Otherwise (for non-minimal conflict sets) the leafs are marked with "*closed*".

4. Performance Evaluation

In this section we discuss the performance of the BFX algorithm. We first look on the best and the worst case of the algorithm. A table which represents those two cases are shown in Figure 4. The best case for the BFX algorithm is when there exists only one hitting set (see right part of the Figure 4). In this situation the algorithm only needs one step. On the other hand the worst case is if the number of different hitting sets is large. An example for such a situation can be seen on the left side of Figure 4. In this situation the algorithm needs *number of constraint* iterations. In recommender systems the tables are variable. Thus we evaluated the performance of the BFX algorithm in different settings with different characteristics. In order to analyze the runtime we compared the BFX algorithm with the algorithms FastXplain[21] and QuickXplain.[9] All algorithms are implemented in Java 1.6 and all experiments were performed on a normal desktop PC (*Intel®Core^{TM}2 Quad CPU Q9400* CPU with *2.66GHz* and *2GB* RAM). The QuickXplain[9] algorithm calculates one minimal conflict set at a time. In order to calculate all minimal conflict sets using the QuickXplain we combined it with HSDAG (Hitting Set Directed Acyclic Graph) of Reiter.[18] With this combination the QuickXplain is called at every point in the HSDAG. The resulting tree is similar to the one that is constructed by the BFX, with the major difference that the QuickXplain approach calculates the minimal conflict sets directly and stores in one level of the tree. In comparison to this the BFX stores the diagnoses in one level.

The crucial point of the performance of the QuickXplain[9] is the consistency checking. One possibility to check the consistency is to use a theorem prover. Another one is to use a database when operating on a product table. If no product is returned, then the consistency checking failed. Another possibility for this calculation is to create a table consisting of the constraints and products (see Table 3). From this table it can be determined if at least one product of the assortment fulfils all constraints of the set that need to be checked. A product fulfils the current

	P1	P2	P3	...	Pn
c_1	0	1	1	...	1
c_2	1	0	1	...	1
c_3	1	1	0	...	1
...	...	...	...	0	...
c_n	1	1	1	1	0

$MCS = \{ c_1, c_2, c_3, \ldots, c_n \}$

	P1	P2	P3	...	Pn
c_1	0	0	0	...	0
c_2	1	1	1	...	1
c_3	1	1	1	...	1
...	...	...	...	...	...
c_n	1	1	1	1	1

$MCS = \{ c_1 \}$

Fig. 4. On the left hand we can see the worst problem that can occur for the BFX algorithm. The algorithm needs n iterations for this problem. On the right a problem is demonstrated where the BFX algorithm performs best (one iteration is needed).

constraints if for this product (column) all selected rows (depending on the set of constraints) are set to 1.

In our evaluation we want to compare the BFX with the version of the QuickXplain[9] which has the best performance according to the structural properties (product table) of recommender systems. We evaluated all versions (usage of a theorem prover, consistency checking through the database and using the table data structure) for different settings and the results have shown clearly that the version with the consistency checking using the table data structure containing the constraints and items (similar to Table 3) is the one with the best performance. We used the version of QuickXplain[9] with the best performance for our evaluation.

4.1. *Different number of items*

One critical point when analysing algorithms for recommendation problems is the suitability for a large number of items. The typical number of items depends on the domain of the recommender system. To study if the algorithm BFX is feasible for a large spectrum of applications we compared the runtime to the one of FastXplain and QuickXplain for 10 and 20 user requirements and an increasing amount of items. In our test cases on an average 50% of the attributes of an item fulfil the user constraint.

In order to evaluate the performance we compared the run time of all three algorithms calculating all minimal conflict sets. This conflict set consisting of a subset of user requirements can be used to help the user finding a solution for an inconsistent setting. The usability of a system depends on the performance of the underlying algorithm to help the user find a solution. Thus a fast computation of at least one minimal conflict set is important.

Figure 5 shows the average runtime of the algorithms for calculating all minimal conflict sets for 10 user requirements and 500, 1000, 2000, 4000 and 8000 items. Each

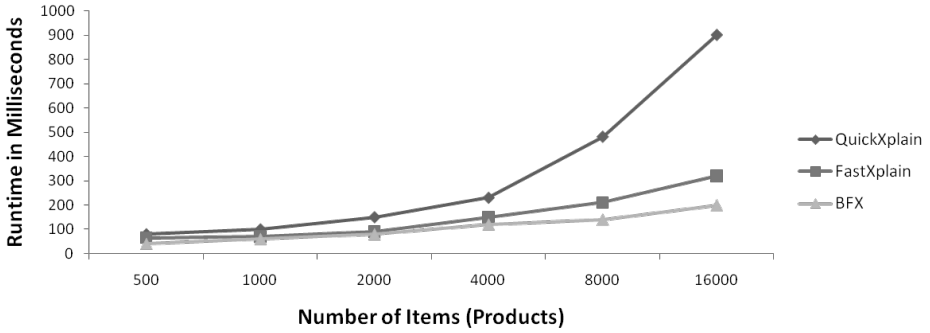

Fig. 5. Runtime comparison of the algorithms BFX, FastXplain and QuickXplain to calculate all minimal conflict sets (10 user constraints and an increasing number of items).

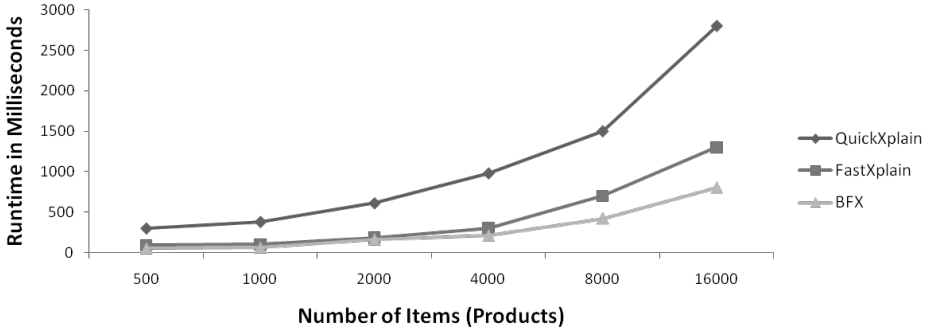

Fig. 6. Runtime comparison of the algorithms BFX, FastXplain and QuickXplain to calculate all minimal conflict sets (20 user constraints and an increasing number of items).

test case was performed 50 times. In this figure we can clearly see that the BFX algorithm outperforms the FastXplain and the QuickXplain algorithms. Figure 6 shows the performance of the same tests, using 20 different user requirements. When we take a closer look to this Figure 6 we can see that 16000 items and 20 constraints the QuickXplain takes about 3 seconds to calculate the results for the user. In interactive settings the user looses their attention when waiting for such a long time. The BFX algorithm needs for the same problem less then one seconds which makes this algorithm much more suitable for interactive settings.

Sometimes it is enough to calculate one minimal conflict set for helping the user in their selection process. For example for a constraint-based decision process in interactive settings. Thus we studied the performance of the algorithms. The conclusion is similar compared to the one for calculating all minimal conflict sets. The performance of the BFX is similar to the FastXplain and slightly better compared to the QuickXplain, but the difference in these runtimes was quite low compared to the one where all minimal conflict sets are calculated. Based on this evaluation

we can say that all three algorithms are suited for calculating one minimal conflict set independent from the number of items. But when it comes to the calculation of all minimal conflict set the BFX performs best.

4.2. *Different number of constraints*

In different recommender applications the number of constraints varies quite a bit. Thus we also evaluated the influence of the number of user constraints on the runtime. We used setting with 6, 8, 10, 12, 14, 16, 18 and 20 constraints. In Figure 7 we show the runtime of the algorithms BFX, FastXplain and QuickXplain using 1000 items calculating all minimal conflict sets. This Figure shows similar to the Figures 5 and 6 that the BFX is faster compared to the FastXplain and QuickXplain.

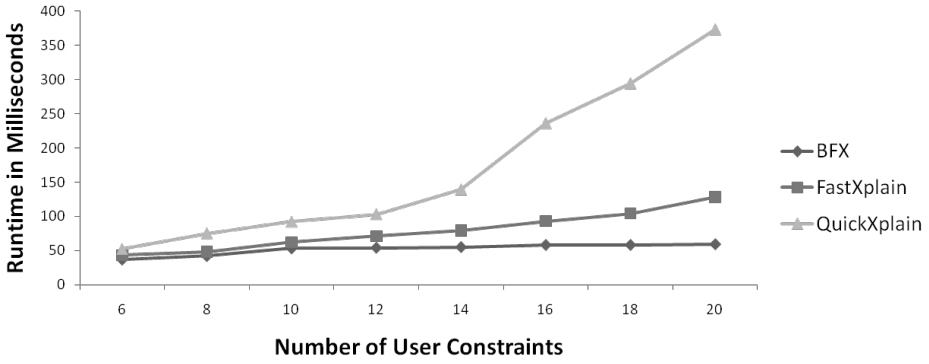

Fig. 7. Runtime comparison of the algorithms BFX, FastXplain and QuickXplain to calculate all minimal conflict sets using 1000 items and an increasing number of constraints (6, 8, 10, 12, 14, 16, 18 and 20).

For a broader study we increased the number of items to 4000 and performed the evaluation again. As shown in Figure 8 the relation between the runtimes does not change a lot, although the overall runtime increased. When looking at Figure 8 we see that the difference is not that big between the BFX and the FastXplain. This can be explained by the fact that the BFX is based on the FastXplain.[21]

4.3. *Different number of minimal conflict sets*

In some applications it is neither needed to calculate only one minimal conflict set nor is it suited to calculate all minimal conflict sets — just a few are enough. Thus we conducted a study using 2000 items and 10 user requirements (results can be seen in Figure 9). The runtime of the QuickXplain algorithm increases with the number of minimal conflict sets. This is based on the consistency checks for the calculations. In comparison to this the runtime of BFX and FastXplain are nearly

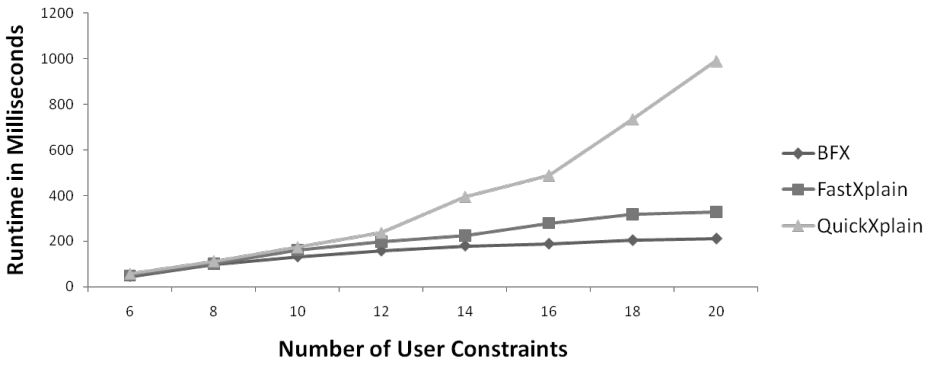

Fig. 8. Runtime comparison of the algorithms BFX, FastXplain and QuickXplain to calculate all minimal conflict sets using 4000 items and an increasing number of constraints (6, 8, 10, 12, 14, 16, 18 and 20).

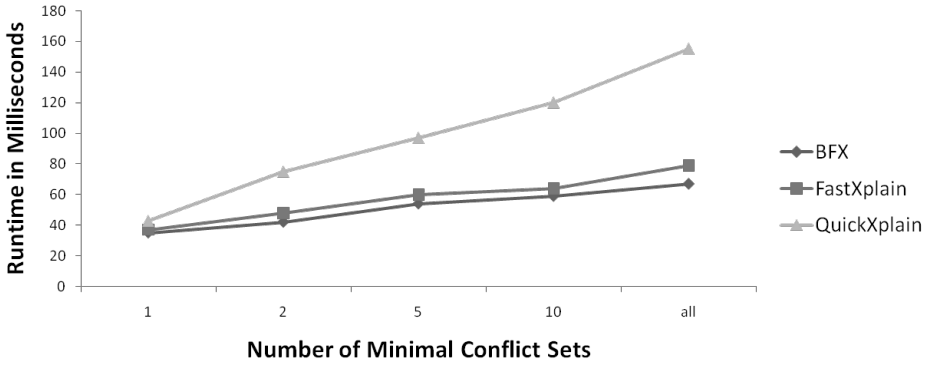

Fig. 9. Runtime comparison of the algorithms BFX, FastXplain and QuickXplain to calculate an increasing number of minimal conflict sets using 2000 items and 10 constraints.

independent (only very slightly increasing) of the number of minimal conflict sets. Thus no matter if only one or a few minimal conflict sets are needed, the BFX and FastXplain are suited for all settings. Summarizing we can say that the BFX and the FastXplain can be used to calculate one minimal conflict set as well as to calculate all minimal conflict sets. Both types of calculations are used in constraint-based recommender systems.

5. Related Work

The concepts presented in this paper are extremely useful for constraint-based recommendation settings. Felfernig *et al.*[4] introduced an integrated environment for the development of constraint-based (knowledge-based) recommender applications. This environment automatically generates recommender applications based on a

corresponding graphical model. One major functionality of the corresponding recommender user interface is the determination of minimal conflict sets. This supports the user in addressing the *no solution could be found* dilemma. Further descriptions of knowledge-based recommendation environments can be found in (see Thompson *et al.*,[22] Mirzadeh *et al.*,[15] Jiang *et al.*[8]). Many of those approaches rely on intelligent mechanisms to proactively support the user in situations where no solution can be found.

In the context of configuration problems Felfernig *et al.*[5] have developed concepts for identifying inconsistent user requirements. The idea was to determine minimal cardinality sets of requirements that need to be changed in order to be able to find a solution. The calculation of repair sets is based on the calculation of minimal conflict sets and diagnoses. The idea is to combine a conflict detection algorithm such as QuickXplain[9] with the hitting set algorithm used in model-based diagnosis (MBD) (see Reiter[18]) for the calculation of minimal diagnoses. Based on these diagnoses repair actions can be identified. In Felfernig *et al.*[6] MBD is applied for the identification of faulty utility constraint sets in the context of knowledge-based recommendation. In both scenarios, the BFX algorithm is not applicable since it is not possible to construct the intermediate representation: configuration knowledge bases are an implicit representation of the complete set of possible configurations; the same holds for utility constraint sets.

The conflict sets exploited in Felfernig *et al.*[3] are not necessarily minimal which increases the runtime due to a larger Hitting Set Directed Acyclic Graph (HSDAG). An efficient algorithm to identify minimal conflict sets is QuickXplain which has been introduced by Junker.[9] This approach is based on a recursive divide-and-conquer strategy which calculates one minimal conflict set at a time. It is not bounded to finite solution domains (like the product table of the constraint-based recommender setting), but can be used for a broader range of over-constrained problems (including finite and infinite variable domains). Mauss[12] introduced another algorithm to find minimal subsets (in finite and infinite solution domains) that are responsible for an inconsistency. Compared to Junker[9] that approach by Mauss does not take the ordering into account. Besides that it does not calculate all minimal conflict sets, but only one and with an extension more.

The approach described in Schlobach *et al.*[19] uses pinpointing for the identification of repairs for incoherent terminologies. These pinpoints prevent the algorithm from calculating minimal hitting sets by using the superset to approximate minimal diagnoses. To compute the pinpoints themselves, all minimal conflict sets are needed. Compared to the minimal conflict sets used in Model-Based Diagnoses (see Reiter[18]) are computed on demand.

Schubert *et al.*[20] have developed an approach to calculate minimal conflict sets for recommender settings based on product tables. This approach is inspired by the concepts of network analysis and was developed especially for knowledge-based recommender systems. The evaluation in Schubert *et al.*[20] compares the QuickXplain[9] in a version where the consistency checks are done with a database. As the study

of this paper has shown, the table data structure to check the consistency is much faster than using database queries, we decided to compare the runtime of the BFX with the - to the best of our knowledge - fastest implementation of QuickXplain.[9]

6. Conclusions and Future Work

In this paper we presented an approach to identify minimal conflict sets for inconsistent user requirements in constraint-based recommendation scenarios. The efficient determination of minimal conflicts is crucial for the usability and acceptance of constraint-based recommender systems. We came up with the algorithm BFX which is based on an intermediate representation derived from the user requirements and the products of the assortment. One the basis of this intermediate representation we applied the Hitting Set Directed Acyclic Graph (HSDAG) algorithm[18] for the calculation of minimal conflict sets. The results of a performance evaluation show that our approach performs significantly better than existing state-of-the-art approaches such as QuickXplain.[9]

In a future work the BFX algorithm can be extended using personalization strategies. This personalization could include strategies like similarity (see McSherry[13]), the MAUT — multi attribute utility theory or any other collaborative problem solving concept.

References

1. R. Burke, Knowledge-based recommender systems in *Encyclopedia of Library and Information Systems*, volume 69, pp. 180–200. New York, NY, USA, 2000.
2. A. Felfernig and R. Burke, Constraint-based recommender systems: Technologies and research issues, in *IEEE International Conference on Electronic Commerce*, pp. 1–10, 2008.
3. A. Felfernig, G. Friedrich, D. Jannach, and M. Stumptner. Consistency-based diagnosis of configuration knowledge bases, *Artificial Intelligence* **152**(2) (2004) 213–234.
4. A. Felfernig, G. Friedrich, D. Jannach, and M. Zanker, An Environment for the development of knowledge-based recommender applications, *International Journal of Electronic Commerce* (*IJEC*) **11**(2) (2006) 11–34.
5. A. Felfernig, G. Friedrich, M. Schubert, M. Mandl, M. Mairitsch, and E. Teppan, Plausible repairs for inconsistent requirements, in *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, pp. 791–796, 2009.
6. A. Felfernig, G. Friedrich, E. Teppan, and K. Isak, Intelligent debugging and repair of utility constraint sets in knowledge-based recommender applications, in *Proceedings of 13th ACM International Conference on Intelligent User Interfaces* (*IUI2008*), pp. 218–226, 2008.
7. D. Jannach, Finding preferred query relaxations in content-based recommenders, *Intelligent Techniques and Tools for Novel System Architectures* (2008) 81–97.
8. B. Jiang, W. Wang, and I. Benbasat, Multimedia-based interactive advising technology for online consumer decision support, *Communications of the ACM* **48**(9) (2004) 93–98.
9. U. Junker, Quickxplain: Preferred explanations and relaxations for over-constrained problems, in *Proceedings of the 19th National Conference on Artificial Intelligence*, pp. 167–172, AAAI Press/The MIT Press, 2004.

10. M. T. Gomez-Lopez, R. Ceballos, R. M. Gasca, and S. Pozo, Determination of possible minimal conflict sets using constraint databases technology and clustering, in *IBERAMIA*, pp. 942–952, Springer-Verlag Berlin Heidelberg, 2004.
11. J. A. Konstan, B. N. Miller, D. Maltz, J. L. Herlocker, L. R. Gordon, and J. Riedl, Grouplens: Applying collaborative filtering to usenet news, *Communication ACM* **40**(3) (1997) 1–10.
12. J. Mauss and M. Tatar, Computing minimal conflicts for rich constraint languages, in *Proceedings of the 15th European Conference on Artifical Intelligence*, pp. 151–160, 2002, Lyon, France.
13. D. McSherry, Similarity and compromise, in *Proceedings of the International Conference on Case-based Reasoning* (*ICCBR03*), pp. 291–305, 2003.
14. D. McSherry, Maximally successful relaxations of unsuccessful queries, in *Proceedings of the 15th Conference on Artificial, Intelligence and Cognitive Science*, Galway, Ireland, pp. 127–136, 2004
15. N. Mirzadeh, F. Ricci, and M. Bansal, Supporting user query relaxation in a recommender system, *LNCS 3182*, Zaragoza, Spain, pp. 31–40, 2004.
16. B. O'Sullivan, A. Papadopoulos, B. Faltings, and P. Pu, Representative explanations for over-constrained problems, in *Proceedings of the National Conference on Artificial Intelligence*, pp. 323–328, 2007, Cork Constraint Computation Centre, University College Cork, Ireland.
17. M. Pazzani and D. Billsus, Learning and revising user profiles: The identification of interesting web sites, *Machine Learning* **27**(3) (1997) 313–331.
18. R. Reiter, A theory of diagnosis from first principles, *Artificial Intelligence* **32**(1) (1987) 57–95.
19. S. Schlobach, Z. Huang, R. Cornet, and F. van Harmelen, Debugging incoherent terminologies, *Journal of Automated Reasoning* **39**(3) (2007) 317–349.
20. M. Schubert, A. Felfernig, and M. Mandl. Solving over-constrained problems using network analysis, *Proceedings of International Conference on Adaptive and Intelligent Systems*, Klagenfurt, Austria, pp. 9–14, 2009.
21. M. Schubert, A. Felfernig, and M. Mandl. FastXplain: Conflict detection for constraint-based recommendation problems, *Trends in Applied Intelligent Systems: 23rd International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems*, Cordoba, Spain, pp. 621–630, 2010.
22. C. Thompson, M. Goeker, and P. Langley, A personalized system for conversational recommendations, *Journal of Artificial Intelligence Research* **21** (2004) 393–428.