Online Query Relaxation via Bayesian Causal Structures Discovery

Ion Muslea & Thomas J. Lee
SRI International
333 Ravenswood
Menlo Park, California 94025
muslea@ai.sri.com, thomas.lee@sri.com

Abstract

We introduce a novel algorithm, TOQR, for relaxing *failed queries* over databases; i.e., over-constrained DNF queries that return an empty result. TOQR uses a small dataset to discover the implicit relationships among the domain attributes, and then it exploits this domain knowledge to relax the failed query. TOQR starts with a relaxed query that does not include any constraint, and it tries to add to it as many as possible of the original constraints or their relaxations. The order in which the constraints are added is derived from the domain's causal structure, which is learned by applying the TAN algorithm to the small training dataset. Our experiments show that TOQR clearly outperforms other approaches: even when trained on a handful of examples, it successfully relaxes more that 97% of the failed queries; furthermore, TOQR's relaxed queries are highly similar to the original failed query.

Introduction

Manually relaxing *failed queries*, which do not match any tuple in a database, is a frustrating, tedious, time-consuming process. *Automated* query relaxation algorithms (Gaasterland 1997; Chu *et al.* 1996b) are typically trained *offline* to acquire domain knowledge that is then used to relax *all* failed queries. In contrast, LOQR (Muslea 2004) takes an *online, query-guided* approach: the domain knowledge is extracted *online* in a process driven by the actual constraints in each failed query. Even though this approach was shown to be extremely successful, when trained on small datasets LOQR tends to generate short queries that contain only a small fraction of the constraints from the failed query.

We introduce a novel algorithm, TOQR, that is similar to LOQR, without sharing its weakness: even when trained on just a handful of examples, TOQR generates non-failing relaxed queries that are highly similar to the failed ones. In order to better explain our contribution, let us first summarize the similarities between TOQR and LOQR. They both use a small dataset $\mathcal{D}$ to generate - via machine learning - queries Q_i that are then used to relax the failed query. These queries Q_i are created so that (1) they are as similar as possible to the failed query and (2) they do not fail on $\mathcal{D}$ (if $\mathcal{D}$ is representative of the target database TD, it follows that Q_i is unlikely to fail on TD).

Our main contribution is a novel approach to generate non-failing queries Q_i that are highly similar to the failed query Q_f. In contrast to LOQR, which uses $\mathcal{D}$ to learn decision rules that are then converted into queries, TOQR starts with an empty query Q_i, to which it tries to add as many as possible of Q_f's constraints (or their relaxations). The order in which TOQR considers the constraints is derived from the domain's causal structure, which is learned from $\mathcal{D}$.

More precisely, for each constraint in Q_f, TOQR uses TAN (Friedman, Goldszmidt, & Lee 1998) to learn both the topology and the parameters of a Bayesian network that predicts whether that constraint is satisfied. TOQR tries to add the Q_f constraints to Q_i in the order of the breadth-first traversal of the learned topology. Intuitively, the breadth-first traversal minimizes the conflicts between the constraints on the various domain attributes (in the TAN-generated Bayesian network, the nodes are independent of each other, given their parents). Our empirical evaluation shows that the order in which the attributes are considered is critical: adding the constraints to Q_i in an arbitrary order leads to a significantly poorer performance.

Related Work

CO-OP (Kaplan 1982) was the first system to address the problem of failing queries. CO-OP transforms the failed query into an intermediate, graph-oriented language in which the connected sub-graphs represent the query's *presuppositions*. CO-OP tests each of these presupposition against the database by converting the subgraphs into subqueries. FLEX (Motro 1990), which is a generalization of CO-OP, is highly tolerant to incorrect queries because of its ability to iteratively interpret the query at lower levels of correctness. When possible, FLEX proposes non-failing queries that are similar to the failing ones; otherwise it just provides an explanation for the query's failure.

As finding all *minimal failing* and *maximal succeeding* sub-queries is NP-hard (Godfrey 1997), CO-OP and FLEX have a high computational cost, which comes from evaluating a large number of queries against *the entire database*. To speed up the process, (Motro 1986) introduces heuristics for constraining the search, while (Gaasterland 1997) controls the query relaxation process via heuristics based on

semantic query-optimization.

CoBase (Chu *et al.* 1996a; 1996b; Chu, Chen, & Huang 1994), which uses machine learning techniques to relax the failed queries, is the closest approach to TOQR and LOQR. By clustering all the tuples in the target database (Merzbacher & Chu 1993), CoBase automatically generates Type Abstraction Hierarchies (TAHs) that synthesize the database schema and tuples into a compact form. To relax a failing query, CoBase uses three types of TAH-based operators: generalization, specialization, and association (i.e., moving up, down, or between the hierarchies, respectively). Note that CoBase performs the clustering only once, on *the entire database*, and independently of the actual constraints in the failing queries. In contrast, TOQR's learning process is performed online and is driven by the constraints in each individual query; furthermore, TOQR uses only a small training set, thus not requiring access to all the tuples in the target database.

The Intuition

Consider an illustrative laptop domain, in which the query

$$Q_f: \quad Price \leq \$2,000 \bigwedge CPU \geq 2.5\,GHz \bigwedge Display \geq 17'' \bigwedge Weight \leq 3\,lbs \bigwedge HDD \geq 60GB$$

fails because laptops under $3\,lbs$ have displays smaller than $17''$ (and, vice-versa, laptops with displays over $17''$ weigh more than $3\,lbs$).

In order to relax Q_f, TOQR proceeds in three steps: first, it uses a small dataset to learn the domain's causal structure, which is then exploited to generate queries that are guaranteed *not* to fail on the training data. Second, it identifies the generated query Q_{sim} that is most similar to Q_f; finally, it uses the constraints from Q_{sim} to relax Q_f.

Step 1: Extracting domain knowledge

TOQR uses a small dataset $\mathcal{D}$ to discover knowledge that can be used for query relaxation. TOQR considers Q_f's constraints independently of each other and learns from $\mathcal{D}$ "what does it take" to fulfill each particular constraint. This knowledge is then used to create a relaxed query that is similar to Q_f, but does not fail on $\mathcal{D}$ (as already mentioned, if $\mathcal{D}$ is representative of the target database TD, the relaxed query is also unlikely to fail on TD).

For example, consider the dataset $\mathcal{D}$ in Table 1, which consists of various laptop configurations. In order to learn to predict whether $Price \leq \$2,000$ is satisfied, TOQR creates a duplicate D_1 of $\mathcal{D}$; for each example in D_1, TOQR replaces the original value of $Price$ by a binary one that indicates whether or not $Price \leq \$2,000$ is satisfied; finally, the binary attribute $Price$ is designated as D_1's class attribute.

In order to discover "what does it take" to satisfy the constraint $Price \leq \$2,000$, TOQR uses D_1 to train the TAN learner (Friedman, Goldszmidt, & Lee 1998). From the given data, TAN learns both the topology and the parameters of a Bayesian network classifier. In order to keep the computation tractable, TAN considers only topologies that are similar to the one shown in Figure 1:

RAM	Price	CPU	HDD	Weight	Screen
1024	$2299	3.0 GHz	50 GB	3.1 lbs	18"
128	$1999	1.6 GHz	80 GB	3.6 lbs	14"
64	$1999	2.0 GHz	20 GB	2.9 lbs	12"
512	$1898	2.5 GHz	60 GB	4.3 lbs	16"
256	$1998	2.8 GHz	60 GB	4.1 lbs	17"

Table 1: The dataset $\mathcal{D}$.

- the network's root (i.e., $Price$) influences the values of all domain attributes (see dotted arrows in Figure 1);
- each non-class node can also have *at most* an additional parent (e.g., $Display$ also depends on $Weight$).

One can read the network in Figure 1 as follows: besides the dependencies on the class attribute, the only significant dependencies discovered by TAN are the influence of $Weight$ on $Display$, and of CPU on RAM and HDD. Intuitively, this means that once we set the class value (e.g., computers under $\$2,000$), the only interactions among the values of the attributes are the one between $Weight$ and $Display$, and the one among CPU, RAM, and HDD. In turn, this implies that Q_f's failure is due to the incompatible values of the attributes in one or both of these groups of attributes.

TOQR uses the extracted domain knowledge (i.e., the Bayesian network) to generate a query Q'_{Price} that is similar to Q_f but does not fail on $\mathcal{D}$. TOQR starts with an empty query Q'_{Price}, to which it tries to add - one at the time - the constraints from Q_f. The order in which TOQR considers the constraints is derived from the topology of the Bayesian network. More precisely, TOQR proceeds as follows:

- it detects all the constraints imposed by Q_f on the *parentless* nodes in the network;
- it ranks these constraints by the number of tuples in $\mathcal{D}$ that satisfy *both* the current Q' and the constraint itself;
- it *greedily* tries to add to Q'_{Price} as many as possible of the constraints (one at the time, higher-ranked first). If adding a constraint C leads to the failure of Q'_{Price}, then C is removed and the process continues with next highest-ranking constraint.
- it deletes from the Bayesian network the parentless nodes, together with the directed edges leaving them;
- it repeats the steps above until all the nodes are visited.

In our running example, the algorithm above is executed as follows. In the first iteration, TOQR starts with an empty Q'_{Price} and considers the $Price$ attribute (as the network's root, this is the only parentless node). By adding to Q'_{Price} the $Price$ constraint, we get a $Q'_{Price} = Price \leq \$2,000$, which matches several tuples in $\mathcal{D}$. As there are no other parentless nodes to consider, the $Price$ node and all the dotted arcs are deleted from the network. The remaining forest consists on two trees, rooted in $Weight$ and CPU, respectively. These two new parentless nodes are the ones considered by TOQR in the next iteration.

It is easy to see that TOQR ranks the CPU constraint higher than the $Weight$ one: among the tuples matching

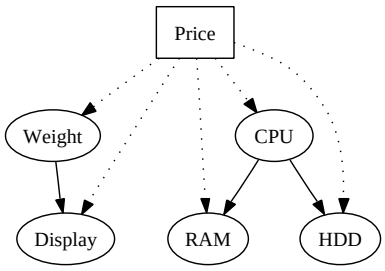

Figure 1: Bayesian network learned by TAN.

Q'_{Price} (i.e., the bottom four in Table 1), $CPU \geq 2.5\,GHz$ matches two tuples, while $Weight \leq 3\,lbs$ matches only one. Consequently, TOQR tries first to add the CPU constraint, and Q'_{Price} becomes $Price \leq \$2,000 \bigwedge CPU \geq 2.5\,GHz$.

Then TOQR tries to add $Weight \leq 3\ lbs$ to this new Q'_{Price}, but the resulting query $Price \leq \$2,000 \bigwedge CPU \geq 2.5GHz \bigwedge Weight \leq 3lbs$ does not match any tuple from Table 1; consequently, the $Weight$ constraint is removed from Q'_{Price}. As both parentless nodes were considered, TOQR deletes the nodes $Weight$ and CPU, together with edges originating in them (i.e., all remaining arcs).

In the last iteration, TOQR considers the nodes $Display$ and HDD (RAM is ignored because it does not appear in Q_f). Of the two tuples matched by Q'_{Price} (i.e., the bottom ones in Table 1), $Display \geq 17''$ matches only one, while $HDD \geq 60GB$ matches both. Consequently, $HDD \geq 60GB$ is added to Q'_{Price}, which becomes $Price \leq \$2,000 \bigwedge CPU \geq 2.5\ GHz \bigwedge HDD \geq 60GB$ (and still matches the same two tuples). Then TOQR adds $Display \geq 17''$ to Q'_{Price}; as the resulting query matches one tuple in $\mathcal{D}$, we get the final result

$$Q'_{Price}: \quad Price \leq \$2,000 \bigwedge CPU \geq 2.5\,GHz \bigwedge Display \geq 17'' \bigwedge HDD \geq 60GB$$

TOQR performs the algorithm above once for each constraint in Q_f; i.e., TOQR also creates the datasets $D_2 - D_5$, in which the binary class attributes reflect whether or not the constraints on *CPU*, *HDD*, *Weight*, and *Display* are satisfied, respectively. Then TAN is applied to each of these datasets, and the corresponding queries Q'_{CPU}, Q'_{HDD}, Q'_{Weight}, and $Q'_{Display}$ are generated.

At this point, let us re-emphasize that the learning process above takes place *online*, for each failing query Q_f; furthermore, the process is also *query-guided* in the sense that each of the datasets $D_1 - D_5$ is created *at runtime* by using the actual constraints from the failed query. This *online, query-guided* nature of both TOQR and LOQR distinguishes them from all other existing approaches.

The key characteristic of TOQR, which also represents our main contribution, is the use of the learned network topology for deriving the order in which the constraints are added to Q'. As our experiments will show, an algorithm identical to TOQR - except that it adds the constraints in an arbitrary order - performs significantly worse than TOQR. Intuitively, this is due to the fact that - early in the search - one may inadvertently add a constraint that dramatically constraints the range of values for the other domain attribute.

For example, assume that TOQR begins by considering first the constraint on $Weight$ (rather than the one on $Price$). Then TOQR starts with $Q'_{Price} = Weight \leq 3\ lbs$, which matches a single example in $\mathcal{D}$ (i.e., the third one). As the only Q_f constraint that can be added to this query is the one on $Price$, it follows that the final result is $Price \leq \$2,000 \bigwedge Weight \leq 3\ lbs$. It is easy to see that this two-constraint query is less similar to Q_f than the four-constraint one created earlier.

Steps 2 & 3: Relaxing the failing query

In order to complete the query relaxation process, TOQR proceeds in two steps, which are identical to the ones performed by LOQR (Muslea 2004). First, it finds - among the queries generated in the previous step - the one that is most similar to Q_f. Second, it uses the constraints from this "most similar" query to relax the constraints in Q_f.

For pedagogical purposes, let us assume that when learning to predict whether $CPU \geq 2.5\ Ghz$ is satisfied, TOQR generates the query

$$Q'_{CPU}: \quad Price \leq \$3,000 \bigwedge CPU \geq 2.5\,GHz \bigwedge Weight \leq 4\,lbs$$

Note that two of the values in the constraints above are not identical to those in Q_f; this is an illustration of TOQR's ability to relax the numerical values from the constraints that could not be added unchanged to Q' (see next section for details).

It is easy to see that Q'_{Price} is more similar to Q_f than Q'_{CPU}: the only difference between Q'_{Price} and Q_f is that the former does not impose a constraint on the $Weight$; in contrast, Q'_{CPU} includes a weaker constraint on $Price$, without imposing any constraints on display or hard disk sizes. More formally, TOQR uses the similarity metric from (Muslea 2004), in which the importance/relevance of each attribute is described by user-provided weights.

To illustrate TOQR's third step, let us assume that, among the queries generated after applying TAN to $D_1 - D_5$, the one that is the most similar to Q_f is

$$Q'_{Display}: \quad Display \geq 17'' \bigwedge Price \leq \$2,300 \bigwedge Weight \leq 3.1\ lbs \bigwedge CPU \geq 2.5\,GHz$$

Then TOQR creates a relaxed query Q_{rlx} that contains only constraints on attributes that appear both in Q_f and $Q_{Display}$; for each of these constraints, Q_{rlx} uses the least constraining of the numeric values in Q_f and $Q_{Display}$. In our example, we get

$$Q_{rlx}: \quad Price \leq \$2,300 \bigwedge CPU \geq 2.5\,GHz \bigwedge Display \geq 17'' \bigwedge Weight \leq 3.1\,lbs$$

which is obtained by dropping the original constraint on the hard disk (since it appears only in Q_f), keeping the constraint on CPU and $Display$ unchanged (Q_f and $Q_{Display}$

have identical constraints on these attributes), and setting the values for $Price$ and $Weight$ to the least constraining ones.

The approach above has two advantages. First, as $Q_{Display}$ is the statement the most similar to Q_f, TOQR makes minimal changes to the original failing query. Second, as the constraints in Q_{rlx} are a subset of those in $Q'_{Display}$, and they are *at most* as tight as those in $Q'_{Display}$ (some of them may use looser values from Q_f), it follows that all examples that satisfy $Q'_{Display}$ also satisfy Q_{rlx}. In turn, this implies that Q_{rlx} is *guaranteed* not to fail on $\mathcal{D}$, which makes it unlikely to fail on the target database.

The TOQR algorithm

As shown in Figure 2, TOQR takes as input a failed DNF query $Q_f = C_1 \bigvee C_2 \bigvee \ldots \bigvee C_n$ and relaxes its disjuncts C_k independently of each other (for a DNF query to fail, all of its disjuncts must fail). Each disjunct C_k is a *conjunction* of constraints imposed on (a subset of) the domain attributes:

$$C_k = Constr(A_{i_1}) \bigwedge Constr(A_{i_2}) \bigwedge \ldots \bigwedge Constr(A_{i_k}).$$

We use the notation $Constr_{C_k}(A_j)$ to denote the constraint imposed by C_k on the attribute A_j. In this paper, we consider constraints of the type `Attr Operator NumVal`, where `Attr` is a domain attribute, `NumVal` is a numeric value, while `Operator` is one of $\leq$, $<$, $\geq$, or $>$.

As we have already mentioned, TOQR's second and third steps (see Figure 2) are identical to the ones in LOQR. As the intuition behind them was presented in the previous section, for a formal description of these steps we refer the reader to (Muslea 2004). In the remainder of this paper we focus on TOQR's first step, which represents our main contribution.

Step 1: Extracting the domain knowledge

TOQR uses a dataset $\mathcal{D}$ to discover the implicit relationships that hold among the domain attributes. This is done by learning to predict, for each attribute A_j in C_k, "what does it take" for $Constr_{C_k}(A_j)$ to be satisfied; then the learned information is used to generate a query that does not fail on $\mathcal{D}$ and contains $Constr_{C_k}(A_j)$, together with as many as possible of the other constraints in C_k.

As shown in Figure 3 (see **ExtractDomainKnowledge()**), for each attribute A_j in C_k, TOQR proceeds as follows:

1. it creates a copy D_j of $\mathcal{D}$; in each example in D_j, A_j is set to `yes` or `no`, depending on whether or not $Constr_{C_k}(A_j)$ is satisfied. This binary attribute A_j is then designated as D_j's class attribute.

2. it applies TAN to D_j, thus learning the domain's causal structure, which is expressed as a restricted Bayesian network (each non-class node has as parents the class attribute and *at most* another node).

3. it uses the learned Bayesian network to generate a query (see "**BN2Query()**") that

- does not fail on $\mathcal{D}$, which also makes it highly unlikely to fail on the target database;
- is as similar as possible to the original disjunct C_k.

"**BN2Query()**" starts with an empty candidate query Q', to which it tries to add as many as possible of the constraints in

Given:
- a failed DNF query $Q_f = C_1 \bigvee C_2 \bigvee \ldots \bigvee C_n$
- a small dataset $\mathcal{D}$ representative of the target database

$RelaxedQuery = \emptyset$
`FOR EACH` of Q's failing conjunctions C_k `DO`
- **Step 1:** $Queries$ = **ExtractDomainKnowledge**(C_k, $\mathcal{D}$)
- **Step 2:** $Refiner$ = **FindMostSimilar**(C_k, $Queries$)
- **Step 3:** $RelaxedConjunction$ = **Refine**(C_k, $Refiner$)
- $RelaxedQuery = RelaxedQuery \bigvee RelaxedConjunction$

Figure 2: TOQR independently relaxes each conjunction.

C_k or their relaxations. As shown in Figure 3, "**BN2Query()**" is a 3-step iterative process. First, it detects all the parentless nodes in the network (in the first iteration, it will be only the class node). Second, it sorts these nodes according to the effect that they have on the coverage of Q' (i.e., how many examples in $\mathcal{D}$ would satisfy Q' if C_k's constraint on that attribute is added to Q'). Third, it greedily adds to Q' the constraints on the parentless nodes, starting with the ones that lead to higher coverage. If adding $Constr_{C_k}(A)$ to Q' leads to the failure of the new query, A is added to the $RetryAttribs$ list; in a second pass, "**BN2Query()**" tries to relax the constraints on A by changing its numeric value by one, two, or three standard deviations (these statistics are computed from $\mathcal{D}$). Finally, the parentless nodes and their out-going arcs are eliminated from the network, and the entire process is repeated until all the nodes are visited.

Experimental results

We empirically compare TOQR with LOQR and two baselines, *AddOne* and *DropOne*. *AddOne* is identical to TOQR, except for adding the constraints in an arbitrary order (rather then by exploiting the learned Bayesian structure). *DropOne* starts with the original query and arbitrarily removes one constraint at a time until the resulting query does not fail.

The Datasets and the Setup

We follow the experimental setup that was proposed for LOQR's evaluation (Muslea 2004), with two exceptions. First, as in many real-world applications one can rarely get a dataset $\mathcal{D}$ that consists of more than a few dozen examples, we consider only datasets $\mathcal{D}$ of at most 100 examples (also remember that LOQR performs inadequately on such small datasets). Second, we use only five of the six datasets used for LOQR's evaluation: Laptops, Breast Cancer (Wisconsin), Pima, Water, and Waveform. This is because the sixth dataset, LRS, has a large number of attributes (99 versus 5, 10, 8, 21, 38, respectively), which leads to slow running times (remember that for each query relaxation, LOQR and TOQR invoke their respective learners - C4.5 and TAN - once for each domain attribute).

For each of the five domains, we use the seven failing queries proposed in (Muslea 2004). We consider datasets $\mathcal{D}$ of sizes 10, 20, ..., 100 examples; for each of these sizes, we create 20 arbitrary instances of $\mathcal{D}$. Each algorithm uses $\mathcal{D}$ to create a relaxed query Q_R, which is then evaluated

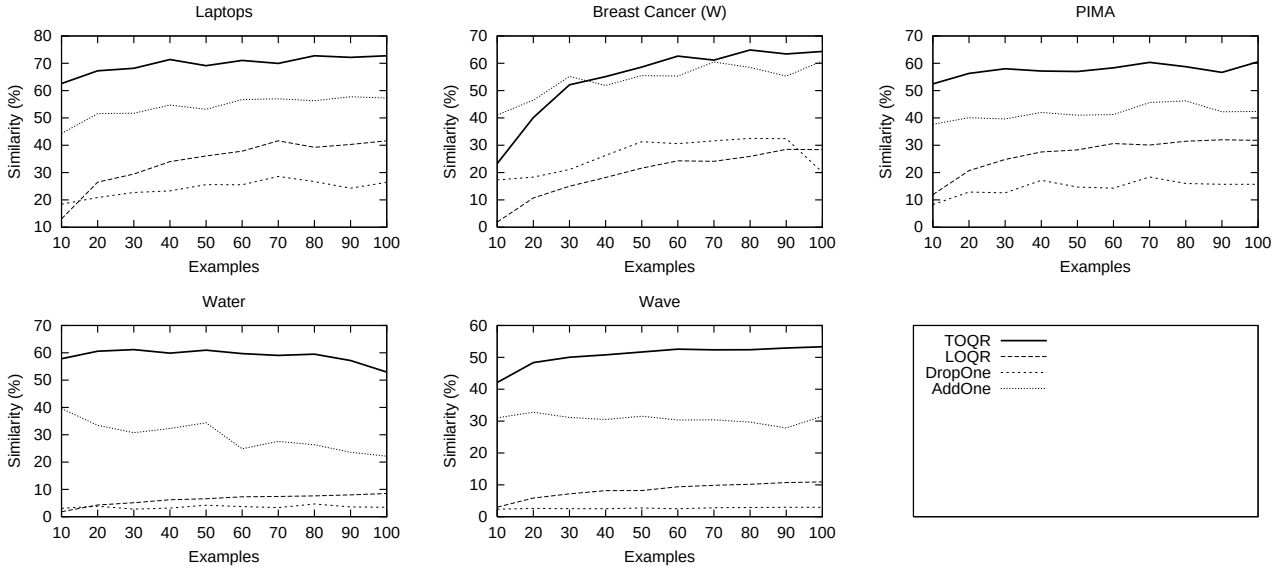

Figure 4: Similarity: how similar is the relaxed query to the failed one?

ExtractDomainKnowledge (conjunction C_k, dataset $\mathcal{D}$**)**
- $Queries = \emptyset$
FOR EACH attribute A_j that appears in C_k DO
  - create a *binary classification* dataset D_j as follows:
    - FOR EACH example $ex \in \mathcal{D}$ DO
      - make a copy ex' of ex
      - IF $ex'.A_j$ satisfies $Constr_{C_k}(A_j)$
        THEN set $ex'.A_j$ to "yes"
        ELSE set $ex'.A_j$ to "no"
      - add ex' to D_j
    - designate A_j as the (binary) class attribute of D_j
  - apply TAN to D_j, with BN_j being the learned Bayesian network
  - $Queries = Queries \bigcup$ **BN2Query**(D, BN_j, C_k)
- return $Queries$

BN2Query(dataset $\mathcal{D}$, TAN network BN, conjunction C_k)
- $Q' = \emptyset$
WHILE there are unvisited nodes in BN DO
  - let $Nodes$ be the set of *parentless* vertices in BN
  - let $Cands = \{Q_i | \forall A_i \in Nodes, Q_i = Q' \wedge Constr_{C_k}(A_i)\}$
  - let $Match(Q_i) = \{x \in D | Satisfies(x, Q_i)\}$
  - sort $Cand$ in the decreasing order of $Match(Q_i), Q_i \in Cand$
  - let $RetryAttribs = \emptyset$
  FOR EACH Q_i in the **sorted** $Cands$ DO
    IF $Q' \bigwedge Constr_{C_k}(A_i)$ does not fail on $\mathcal{D}$ THEN
      - $Q' = Q' \bigwedge Constr_{C_k}(A_i)$
    ELSE $RetryAttribs = RetryAttribs \bigcup \{A_i\}$
  FOR EACH $A \in RetryAttribs$ DO
    - let $RlxConstr =$ **RelaxConstraint**(A)
    IF $Q' \bigwedge RlxConstr$ does not fail on $\mathcal{D}$ THEN
      - $Q' = Q' \bigwedge RlxConstr$
  - remove from BN all $Nodes$ and the arcs leaving them
- return Q'

Figure 3: Extracting domain knowledge by using the *learned structure* of the Bayesian network to generate non-failing queries.

on a *test set* that consists of all examples from the target database that are not in $\mathcal{D}$. For each size of $\mathcal{D}$ and each of the seven failing queries, each algorithm is run 20 times (once for each instance of $\mathcal{D}$); consequently, the reported results are the average of these 140 runs.

The Results

In our experiments, we focus on two performance measures:

- *robustness*: what percentage of the failing queries are successfully relaxed (i.e., they don't fail anymore)?
- *similarity*: how similar to Q_f is the relaxed query? We define the similarity between two conjunctions C and C' as the average - over all domain attributes - of the attribute-wise similarity

  $$Sim_{A_j}(C, C') = \frac{\|Value_C(A_j) - Value_{C'}(A_j)\|}{maxValue_D(A_j) - minValue_D(A_j)}$$

  (by definition , if an attribute appears only in of the conjunctions, $Sim_{A_j}(C, C') = 0$).

Figures 4 and 5 show the *similarity* and *robustness* results on the five domains. TOQR obtains by far the best similarity results: on four of the five domains its similarity levels are dramatically higher than those of the other algorithms; the only exception is Breast Cancer, where *AddOne* performs slightly better. TOQR is also extremely robust: on four of the five domains, it succeeds on more than 99% of the 140 relaxation tasks (i.e., 20 distinct training sets for each of the seven failed queries); on the fifth domain, Water, TOQR still reaches a robustness of 97%.

Overall, TOQR emerges as a clear winner. *DropOne*, which is just a strawman, performs poorly on all domains. The other two algorithms score well either in robustness or in similarity, but at the price of a poor score on the other measure. For example, in terms of robustness, LOQR is competitive with TOQR on most domains; however, on four of

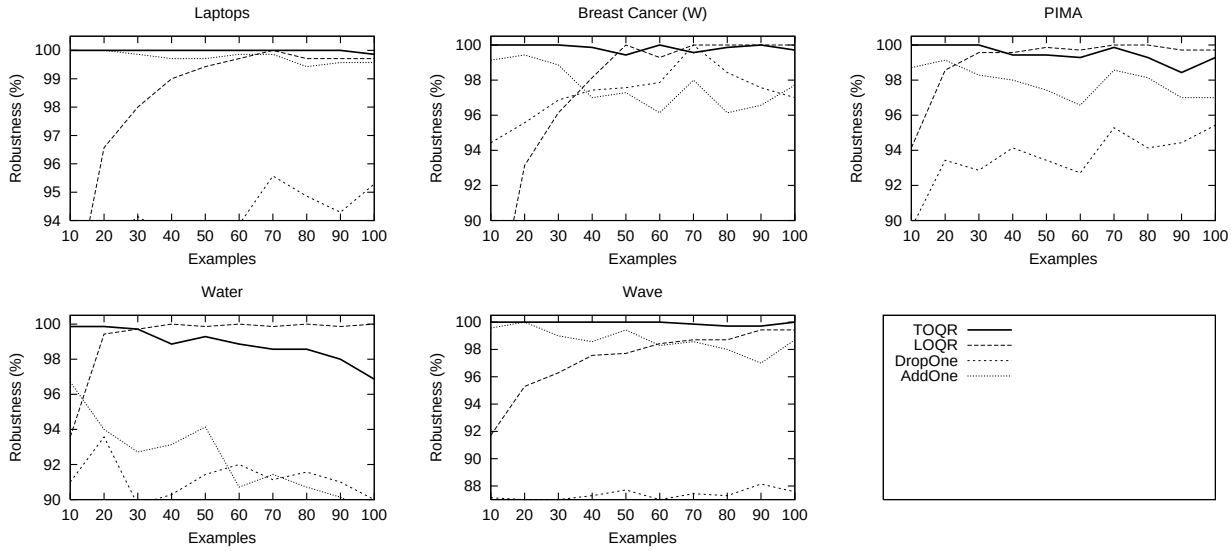

Figure 5: Robustness: what percentage of the relaxed queries are not failing?

the five domains, LOQR's queries are only half as similar to Q_f as the TOQR generated queries.

Finally, let us emphasize an unexpected result: when trained on a datasets $\mathcal{D}$ of at most 30 examples, TOQR typically reaches a robustness of 99-100%; however, as the size of $\mathcal{D}$ increases, the robustness tends to decrease by 1-3%. This is due to the fact that - in larger $\mathcal{D}$s - there may be a few "outliers" that mis-lead TOQR. We analyzed TOQR's traces on these few unsuccessful relaxations, and we noticed that such atypical examples (i.e., no similar examples exist in the test set) may lead to TOQR greedily adding to the relaxed query Q' a constraint that causes its failure on the test set. As a few straightforward strategies to cope with the problem failed, this remains a topic for future work.

Conclusions

We have introduced TOQR, which is an online, query-driven approach to query relaxation. TOQR uses a small dataset to learn the domain's causal structure, which is then used to relax the failing query. We have shown that, even when trained on a handful of examples, TOQR successfully relaxes more than 97% of the failing queries; furthermore, it also generates relaxed queries that are highly similar to the original, failing query. In the future, we plan to create a mixed initiative system that allows the user to explore the space of possible query relaxations. This is motivated by the fact that a user's preferences are rarely cast in iron: even though initially the user may be unwilling to relax (some of) the original constraints, often times, she may change her mind while browsing several (imperfect) solutions.

Acknowledgments

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA), through the Department of the Interior, NBC, Acquisition Services Division, under Contract No. NBCHD030010.

References

Chu, W.; Chiang, K.; Hsu, C.-C.; and Yau, H. 1996a. An error-based conceptual clustering method for providing approximate query answers. *Communications of* ACM 39(12):216–230.

Chu, W.; Yang, H.; Chiang, K.; Minock, M.; Chow, G.; and Larson, C. 1996b. Cobase: A scalable and extensible cooperative information system. *Journal of Intelligent Information Systems* 6(2/3):223–59.

Chu, W.; Chen, Q.; and Huang, A. 1994. Query answering via cooperative data inference. *J of Intelligent Information Systems* 3(1):57–87.

Friedman, N.; Goldszmidt, M.; and Lee, T. J. 1998. Bayesian network classification with continuous attributes: getting the best of both discretization and parametric fitting. In *Proceedings of the International Conference on Machine Learning*, 179–187.

Gaasterland, T. 1997. Cooperative answering through controlled query relaxation. *IEEE Expert* 12(5):48–59.

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

Kaplan, S. 1982. Cooperative aspects of database interactions. *Artificial Intelligence* 19(2):165–87.

Merzbacher, M., and Chu, W. 1993. Pattern-based clustering for database attribute values. In *Proceedings of AAAI Workshop on Knowledge Discovery in Databases*.

Motro, A. 1986. SEAVE: a mechanism for verifying user presuppositions in query system. ACM *Transactions on Information Systems* 4(4):312–330.

Motro, A. 1990. Flex: A tolerant and cooperative user interface databases. *IEEE Transactions on Knowledge and Data Engineering* 2(2):231–246.

Muslea, I. 2004. Machine learning for online query relaxation. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining*, 246–255.