Constrained Optimalities in Query Personalization*

Georgia Koutrika
University of Athens
Hellas
koutrika@di.uoa.gr

Yannis Ioannidis
University of Athens
Hellas
yannis@di.uoa.gr

ABSTRACT

Personalization is a powerful mechanism that helps users to cope with the abundance of information on the Web. Database query personalization achieves this by dynamically constructing queries that return results of high interest to the user. This, however, may conflict with other constraints on the query execution time and/or result size that may be imposed by the search context, such as the device used, the network connection, etc. For example, if the user is accessing information using a mobile phone, then it is desirable to construct a personalized query that executes quickly and returns a handful of answers. *Constrained Query Personalization* (*CQP*) is an integrated approach to database query answering that dynamically takes into account the queries issued, the user's interest in the results, response time, and result size in order to build personalized queries. In this paper, we introduce *CQP* as a family of constrained optimization problems, where each time one of the parameters of concern is optimized while the others remain within the bounds of range constraints. Taking into account some key (exact or approximate) properties of these parameters, we map CQP to a state search problem and provide several algorithms for the discovery of optimal solutions. Experimental results demonstrate the effectiveness of the proposed techniques and the appropriateness of the overall approach.

1. INTRODUCTION

Personalization has come about as a result of a long evolutionary process accelerated by the rapid development of the Web. Such a communication tool has enabled people with varied goals and characteristics to access an ever-growing amount of information. Emergence of hand-held electronic devices, such as palmtops and cellular phones, has increased the possibilities for information access from anywhere and anytime. Under these conditions, it is very difficult for users to find the information they need. As a solution to this problem, researchers from different communities have developed *personalization systems*, which adapt their behavior to the goals, interests, and other characteristics of their users as individuals or members of particular groups.

Query personalization [12, 13, 14] affects system behavior by dynamically enhancing a query with related preferences stored in a user profile with the purpose of providing personalized answers. In principle, query personalization is an *optimization problem*: given a query *q* posed by a user *u*, its goal is to identify the parts of the profile of *u* that, when combined with *q*, would *maximize* the interest of *u* in the results of *q*. In practice, this problem statement may lead to unrealistic solutions, since maximum interest is achieved by incorporating all preferences of *u* into *q*, but the resulting "over-personalized" query is likely to be very expensive or have an empty answer. Taking into account execution time and result size leads to a redefinition of query personalization as a *constrained optimization problem*, where constraints are expressed as an upper bound on execution time of the final query and/or a lower or upper (top-k) bound on its result size. Furthermore, under this more general point of view, one realizes that query personalization does not necessarily imply optimization of user interest, but could also be, for example, optimization of execution time under constraints on user interest.

The family of constrained optimization problems that arise in the spirit discussed above is referred to as *Constrained Query Personalization* (*CQP*). Depending on the parameter being optimized and the constraints placed on the others, different answers will be delivered even to the same user issuing the same query. Each time the correct CQP problem is determined by several real-time factors comprising the search context, such as the device used, the network connection, or even some transient user requirements of the moment, as in the example below.

Example: Al is registered with a web-based service providing tourist information for various places. The system responds to his requests by taking into account a profile of his personal preferences that it maintains as well as the search context at the time of the request. While planning his trip to Pisa, Al looks for general information on the city using his laptop in his office with a high-speed Internet connection, which allows the system to execute expensive queries and provide extensive results that match his profile perfectly. When Al is in Pisa, he may ask for a few local restaurants using his palmtop while walking in the old town. In this case, the search context is different: he is using a device with limited display capabilities and a low-bandwidth network connection. The system should quickly return a short and easily browsable answer with, say, three restaurants that are of Al's general liking.

In this paper, we study Constrained Query Personalization in the context of database queries. Our contributions are the following:

– We generalize earlier concepts of query personalization and introduce the CQP family of related optimization problems with constraints. (Mapping the search context onto the appropriate CQP problem is a policy issue and is not addressed here.)

– We formulate CQP as a state-space search problem. Each personalized query that is a potential solution is a state in a space

* Partially supported by the Information Society Technologies (IST) Program of the European Commission as part of the DELOS Network of Excellence on Digital Libraries (Contract G038-507618) and by a grant from Microsoft

(a node in a graph), characterized by degree of interest, execution cost, and result size. Transitions (edges) arrange states in partial orders based on each of the aforementioned query parameters.

– We devise state-space search algorithms that take advantage of these partial orders to solve CQP problems efficiently.

– We demonstrate the effectiveness of our approach through experimental results that evaluate the algorithms proposed with respect to several important features.

To the best of our knowledge, this is the first effort on a realistic and comprehensive approach to query personalization through its integration with aspects of query optimization.

2. RELATED WORK

Personalization is a very broad research area that includes methods such as information filtering and recommender systems [8]. Query personalization techniques have been proposed both by the IR [13, 14] and database communities [12]. The key difference of the present work lies in considering not only user interest in personalized results, but also execution time and result size of personalized queries, in order to identify the most appropriate one overall to execute. From earlier work [12], we have adopted the model for preferences stored in user profiles.

Query optimization is a well-established area within databases, studying the problem of execution-cost minimization of a given query [11, 15] by choosing the right execution plan. Traditional query optimization techniques do not lend themselves to solving CQP problems, as the latter take into account multiple query parameters at the same time. Multidimensional query optimization [3] finds a plan (rather than a query as in CQP) that is optimal wrt one criterion and satisfies range constraints wrt other criteria (the term is also used for some OLAP optimization, which is not relevant here). More importantly, for CQP problems, syntactic modifications to a state (query) have known implications (increase/decrease) on its degree of interest, cost, and size. The resulting syntax-based partial orders are taken advantage of by the special algorithms presented here for effective CQP. Execution-cost estimation is a common task of both areas, but again the accuracy requirements on the cost models are quite different, hence, one can afford to use a much less detailed cost model in CQP than the one found in a typical query optimizer.

Answering top-K queries is also related to CQP but only at a superficial level. Both deal with bounds on the query result size, but the former is concerned with returning a pre-specified number of tuples in the query answer [1, 2, 6], while CQP only places bounds on that number.

Interestingly, each of the three areas above addresses exactly one query parameter: user interest in the answer, execution cost, or result size, respectively. CQP is an integrated approach to query answering that dynamically takes into account all three together.

Another class of problems is related to CQP: knapsack problems [9]. However, even the most general, multi-constrained knapsack problem (Integer Programming with positive coefficients) deals with summation both in the objective and the constraints and assumes non-negative integer coefficients. CQP problems may include different (even nonlinear) functions anywhere and have more general coefficients. This, in conjunction with the particular properties of CQP mentioned above, i.e. syntax-based partial orders, makes knapsack algorithms not appropriate in this context.

Finally, given the formulation of CQP as state-space optimization several well-known algorithms are potentially applicable: genetic algorithms [5], simulated annealing [10], tabu search [4], etc. These are generic approaches, however, that do not take into account the problem's particularities or special properties, as mentioned above for CQP problems.

3. USER PREFERENCE MODEL

Any personalization effort requires a model for the representation of preferences in user profiles. In this work, we have adopted a slight simplification of an existing user preference model for relational databases [12], whose basic constituents are described below. For our examples, we consider the following relations, which comprise a subset of a database schema about movies:

```
MOVIE(mid, title, year, duration, did)
DIRECTOR(did, name), GENRE(mid, genre)
```

A user's preferences over a database's contents are expressed on top of the *personalization graph*. This is a directed graph G(V, E) that is an extension of the database schema graph. Nodes in V are (*a*) *relation nodes*, one for each relation in the schema, (*b*) *attribute nodes*, one for each attribute of each relation in the schema, and (*c*) *value nodes*, one for each value that is of any interest to this user. Edges in E are (*a*) *selection edges*, from an attribute node to a value node representing a potential selection condition, and (*b*) *join edges*, from an attribute node to another attribute node representing a potential join condition between these attributes.

Atomic Preferences: Preferences are stored in profiles at the level of atomic query elements (atomic selection and joins, edges in G). An atomic preference for a condition q is expressed by the *degree of interest (doi)* in q, doi(q), which is a real number in the range [0, 1]. doi=0 indicates lack of any interest in the condition, while doi=1 indicates extreme ('must-have') interest. Atomic selection preferences indicate user interest in the values of attributes. Atomic join preferences indicate to what degree related entities are mutually influenced by preferences. These are directed, in the sense that they indicate how preferences on the right-hand-side join relation influences the left-hand-side join relation. Figure 1 shows an example user profile.

p_1:	doi(GENRE.genre='musical')	= 0.5
p_2:	doi(MOVIE.mid = GENRE.mid)	= 0.9
p_3:	doi(MOVIE.did = DIRECTOR.did)	= 1.0
p_4:	doi(DIRECTOR.name = 'W. Allen')	= 0.8

Figure 1. An example user profile

Implicit Preferences: By composing atomic user preferences on conditions (edges) that are adjacent in the personalization graph, one obtains implicit preferences, i.e. preferences on complex query elements that are conjunctions of atomic ones (directed acyclic paths in G). In particular, if p is an implicit preference containing m atomic preferences p_i, then $p = p_1 \wedge \ldots \wedge p_m$. For example, preferences p_3 and p_4 are composed into the following implicit preference for movies directed by W. Allen:

```
MOVIE.did = DIRECTOR.did and DIRECTOR.name = 'W. Allen'
```

The degree of interest in an implicit preference p is a function $f_\otimes$ of the degrees of interest in the constituent atomic ones:

$$doi(p) = f_\otimes(doi(p_1), \ldots, doi(p_m)) \qquad (1)$$

Function $f_\otimes$ must be non-increasing as the length of the corresponding directed path increases:

$$f_{\otimes}(d_1, \ldots d_m) \le min(\{d_1, \ldots d_m\}), \text{ where } d_i = doi(p_i) \qquad (2)$$

Conjunctions of Preferences: The degree of interest in multiple non-adjacent (atomic or implicit) preferences being satisfied together is another function r of the constituent dois. That is, for the conjunction of a set of preferences $P_x = \{p_i \mid i = 1\ldots L\}$, it is

$$doi(P_x) = r(\, doi(p_1), \ldots, doi(p_L)\,) \qquad (3)$$

Clearly, the doi in a set of multiple preferences being satisfied together must increase as more preferences are added to the set:

$$P_x \subseteq P_y \Rightarrow doi(P_x) \le doi(P_y) \qquad (4)$$

The results of a personalized query should be ranked by function r based on the preferences that they satisfy in a profile.

4. CONSTRAINED QUERY PERSONALIZATION

4.1 Problem Description

A query Q is characterized by an execution cost, cost(Q), and a result size, size(Q). A personalized query is also characterized by a degree of interest, doi(Q). In the sequel, we collectively refer to these as query *parameters*. Given a query Q and a user profile U, let P be the set of selection preferences extracted from U and related to Q. A personalized query Q_x is a combination of Q and a subset P_X of P and is denoted $Q_x := Q \wedge P_X$. The objective of query personalization is to build a personalized query Q_U that is optimal with respect to one query parameter and satisfies constraints on the others. Query personalization is, therefore, optimization under constraints. We use the term *Constrained Query Personalization* (*CQP)* to collectively refer to several optimization problems that may be defined as different instantiations of this broad description. Not all conceivable optimization problems, however, are meaningful within the CQP family. This is due to the following properties of the query parameters involved in CQP:

doi: By definition, query personalization aims at producing results interesting to a user. Therefore, the doi parameter must be maximized or satisfy a lower bound.

cost: By nature, execution cost must be minimized or satisfy an upper bound.

size: By definition, query personalization aims at smaller responses. On the other hand, empty answers are always undesirable. Hence, the size parameter must always satisfy a lower bound (default is 1) and possibly an upper one.

Table 1 shows all possible CQP problems that may be conceived.

Table 1. CQP problems

Problem	doi	cost	size
1	MAX	-	$s_{min} \le$ size $\le s_{max}$
2	MAX	cost $\le c_{max}$	-
3	MAX	cost $\le c_{max}$	$s_{min} \le$ size $\le s_{max}$
4	-	MIN	$s_{min} \le$ size $\le s_{max}$
5	doi $\ge d_{min}$	MIN	-
6	doi $\ge d_{min}$	MIN	$s_{min} \le$ size $\le s_{max}$

Based on the above, for Problems 1-3, the optimal personalized query Q_U must satisfy the following:

$$doi(Q_U) = MAX\{doi(Q_X) \mid Q_x = Q \wedge P_x,\ P_x \subseteq P,\ Q_x \text{ satisfies CQP constraints}\}$$

Accordingly, for Problems 4-6, it must satisfy the following:

$$cost(Q_U) = MIN\{cost(Q_X) \mid Q_x = Q \wedge P_x, P_x \subseteq P, Q_x \text{ satisfies CQP constraints}\}$$

At query time, the decision on which CQP problem to solve depends on factors comprising the search context. For example, in the scenario where Al uses his palmtop while walking in Pisa, low bandwidth and user mobility pose an upper limit to system response time and result size. Under these conditions, Problem 3 appears most appropriate, seeking a personalized query with optimal doi that remains within specific bounds on the other parameters. The particular bounds can be derived based on statistics kept by the system or provided by the user on the spot, e.g., if Al requests up to three restaurants, this implies s_{max}=3.

4.2 System Architecture

Figure 2 presents the general architecture of a CQP system. The modules in grey color are the traditional parts of a relational database system. We discuss the remaining CQP modules below.

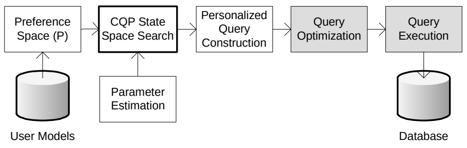

Figure 2. CQP system architecture

Preference Space: Given a query Q and a user profile U, it determines the set P of atomic and implicit selection preferences that are extracted from U and are related to Q.

Parameter Estimation: It provides estimations on the parameters of personalized queries produced by integrating preferences of a subset of P into the original query Q.

CQP State Space Search: This is the main system module. It examines possible personalized queries and identifies the optimal one with respect to the parameter being optimized and the constraints provided for the others. For this purpose, it employs a search strategy for examining the space of personalized queries.

Personalized Query Construction: After 'CQP State Space Search' has selected the optimal subset of preferences to be integrated into Q, this module does the actual modification of the query. Among several possible query rewritings one could use to personalize a query with a set of preferences, we have adopted the one illustrated below with an example: Consider a user, whose profile is shown in Figure 1, issuing a query about movies:

```
select  title from   MOVIE
```

Assume that the following preferences have been selected by the system for inclusion in the query:

```
MOVIE.did = DIRECTOR.did and DIRECTOR.name = 'W. Allen'
MOVIE.mid = GENRE.did and GENRE.genre = 'musical'
```

First, a set of sub-queries is constructed, each one separately integrating one of these preferences into the original query[1].

```
Q1: select  title from   MOVIE M, DIRECTOR D
    where   M.did = D.did and D.name = 'W. Allen'
Q2: select  title from   MOVIE M, GENRE G
    where   M.mid=G.mid and G.genre='musical'
```

[1] There are various cases where multiple preferences can be effectively combined into one sub-query, but investigating the details of this is beyond the scope of this paper.

The final query is built as the union of these sub-queries:

```
select    title  from   Q1 Union All Q2
group by  title  having count(*)= 2
```

Finally, the resulting query is passed on the query optimizer of the underlying database system, where it is executed. The results of this query may be ranked based on their degree of interest.

4.3 Parameter Estimation

In this subsection, we discuss issues of query parameter estimation only to the level required by the forthcoming description of algorithms. More details on estimation formulas used in our experiments are given in Section 7.1. Recall that a personalized query Q_x is a combination of a query Q and a set of preferences $P_x = \{p_i \mid i = 1\ldots L\} \subseteq P$, i.e. $Q_x = Q \wedge P_x$. We are concerned with queries of conjunctions of preferences implemented as unions of sub-queries followed by a `groupby/having` clause (as in the example above).

Degree of interest: Naturally, the doi of Q_x is the doi in the conjunction of the participating preferences (Formula (3)):

$$\mathrm{doi}(Q_x) = r(\ \mathrm{doi}(p_1), \ldots, \mathrm{doi}(p_L)\ ) \qquad (5)$$

Cost: Traditionally, the execution cost of a query is estimated by the database query optimizer. During query personalization, however, we cannot afford to invoke the optimizer for each of the (potentially exponential in number) personalized queries tested. Moreover, duplication of classical query optimization techniques at this stage makes query personalization expensive and is actually meaningless as CQP has more relaxed accuracy requirements on cost estimates. On these grounds, we have adopted an approximate cost model for personalized queries of the form discussed in Section 4.2, based on two assumptions: (*a*) the cost of `groupby/having` is negligible; (*b*) the cost of a union of disjoint sub-queries q_i equals to the sum of individual costs of q_i's; hence, for Q_x the following holds:

$$\mathrm{cost}(Q_x) = \mathrm{cost}(\cup(q_i)) = \Sigma\mathrm{cost}(q_i) \qquad (6)$$

Although not always valid, (6) has proved to be sufficient for the level of accuracy needed in CQP. (In Section 7.1, we discuss how we have estimated the cost of individual sub-queries in our experiments.) Note that, from (6), we derive the following:

$$P_x \subseteq P_y \Rightarrow \mathrm{cost}(Q \wedge P_x) \leq \mathrm{cost}(Q \wedge P_y) \qquad (7)$$

This is similar to (4), but differs from it in that, in contrast to the degree of interest, cost does depend on the initial query Q.

Size: Clearly, the following holds:

$$P_x \subseteq P_y \Rightarrow \mathrm{size}(Q \wedge P_x) \geq \mathrm{size}(Q \wedge P_y) \qquad (8)$$

Formulas (4), (7), and (8) define useful partial orders, which are exploited by CQP algorithms presented in this paper. We observe that: (*a*) for the preference model we adopted [12], the partial orders do not depend on any assumptions far from reality (Formula 4); (*b*) preference inclusion always implies result-set inclusion (Formula 8); and (*c*) it implies cost dominance (Formula 7) under the near-accurate assumptions above. The latter is sufficient for monotonicity of personalized query cost wrt individual preference costs, for typical single-query optimizers. Consequently, nothing else affects the algorithms described here, which take advantage of these partial orders. In addition, from (4), (7), and (8), it is clear that incremental computation of query parameters is possible.

In the rest of the paper, for notational convenience, parameters of query $Q \wedge P_x$ are often referred to as parameters of P_x, e.g., for query Q_x, $\mathrm{cost}(P_x)$ refers to $\mathrm{cost}(Q \wedge P_x)$, $\mathrm{doi}(p_i)$ refers to $\mathrm{doi}(q_i)$, etc.

4.4 Preference Space

Given a query Q and a user profile U, this module determines the set P of selection preferences extracted from U and related to Q. The latter refers to syntactic relationships, i.e. preferences whose paths on the personalization graph are attached to a relation included in Q. For example, the preferences of the example in Section 4.2 are related to the query on movies.

To facilitate some of the CQP state-space search algorithms, we associate with P three vectors of pointers, each one capturing preference order according to one of the CQP parameters (K is the number of preferences in P):

$$D = \{d_i \mid i \in [1\ldots K],\ \mathrm{doi}(p_{d_i}) \geq \mathrm{doi}(p_{d_{i+1}})\ \}$$

$$C = \{c_i \mid i \in [1\ldots K],\ \mathrm{cost}(Q \wedge p_{c_i}) \geq \mathrm{cost}(Q \wedge p_{c_{i+1}})\ \}$$

$$S = \{s_i \mid i \in [1\ldots K],\ \mathrm{size}(Q \wedge p_{s_i}) \leq \mathrm{size}(Q \wedge p_{s_{i+1}})\ \}$$

Consider $P = \{p_1, p_2, p_3\}$ with parameter values given in Table 2. Then, its vectors are: $D = \{2, 3, 1\}$, $C = \{3, 1, 2\}$, $S = \{2, 1, 3\}$.

Table 2. Parameter values for P = {p$_1$, p$_2$, p$_3$}

preference	doi	cost	size
p_1	0.5	10	3
p_2	0.8	5	2
p_3	0.7	12	10

The preference space algorithm is given in Figure 3. Its inputs are a query Q, a profile U, and CQP constraints. It outputs the set P, its vectors D, C, and S, and the cardinality K of P. It takes advantage of the fact that the doi in a preference is a non-increasing function of the length of the corresponding path (Section 3), and performs a best-first traversal of the personalization graph corresponding to U [12] to extract preferences in decreasing order of doi.

```
Preference Space Algorithm
Input:   query Q, profile U, CQP constraints
Output: P, D, C, S, K
1.      P := {}; D := {}; C := {}; S:= {}; K := 0; QP := {};
2.      For each atomic preference p_i in U syntactically related to Q
2.1.        If Q∧p_i satisfies CQP constraints then QP:= add(QP, p_i) fi
        end for
3.      While QP not empty
3.1.        Get head p from QP;
3.2.        If Q ∧ p satisfies CQP constraints then
3.2.1.          If p is selection then
                  P := P ∪ {p}; K := K+1; D := D ∪ {K};
                  C := addrank(C, p, K); S := addrank(S, p, K);
                else  /* p is a join preference */
3.2.2.            For each atomic preference p_i in U adjacent to p
                      If Q∧(p∧p_i) satisfies CQP constraints and p∧p_i is acyclic
                      then QP:= add(QP, p∧p_i) fi
                  end for
                fi
3.3.        else exit fi
        wend
```

Figure 3. Preference Space Algorithm

The algorithm keeps a queue QP of candidate preferences in decreasing order of doi. In each round, it picks from QP the preference p with the highest doi. If p is selection, then it is added at the tail of P. D is updated by inserting K, the cardinality of P, at its tail (since preferences are added in P in decreasing order of doi). Accordingly, addrank(C, p, K) (resp., addrank(S, p, K)) procedure adds K into C (resp., S) in the appropriate place, taking into account the cost (resp., size) of p. If p is join, then for each atomic preference p_i adjacent to p in the personalization graph, a new preference $p \wedge p_i$ is generated and inserted into QP. At various points, the algorithm takes into account the CQP constraints to prune down preferences that can never lead to successful personalized queries. The details of such optimizations are omitted for lack of space.

5. STATE SPACE SEARCH

CQP problems have similar formulation, query parameter properties and partial orders derivable from syntactic transformations of personalized queries. These correspondences enable us to treat them in a very similar way. Therefore, for presentation purposes, the discussion below is focused on one CQP problem, i.e. problem 2 in Table 1. Required adaptations so as to handle all types of CQP problems are discussed in Section 6.

5.1 State Space

Each solution to a combinatorial optimization problem can be thought of as a *state* in a space, i.e. a node in the graph that includes all such solutions. Each state has a feature associated with it, which is given by some problem-specific feature function. The states that can be reached in one move from a state S are called the *neighbors* of S. We model a CQP problem as a state space search problem, as follows.

States. Each state in a CQP problem corresponds to a query built by integrating a set of preferences from the user profile into the initial query, i.e. $Q_x := Q \wedge P_x$, where $P_x \subseteq P$. In the sequel, we will interchangeably use Q_x and P_x to refer to a state. Query parameters comprise the features of the corresponding state.

Transitions. The neighbors of a state are determined by a set of transitions. Transitions are based on transformation rules that are applied on one state and produce a neighbor one. We define two categories of transitions, cost-based and doi-based. Each category creates a different state space (same nodes, different edges). Taking advantage of Formulas (4), (7), and (8), all transitions are based on syntactic modifications to a state with known implications (increase/decrease) on state parameters. This property of transitions actually enables algorithms to work with the pointer vectors, D, C, and S instead of P.

5.2 State Space Search Algorithms

The number of all possible subsets of P is exponential. So is the number of potential personalized queries defined based on Q and U. Thus, the complexity of an exhaustive CQP algorithm is $O(2^K)$. Following subsections provide several precise and heuristic algorithms that improve to varying degrees upon this.

5.2.1 Algorithms on the Cost State Space

Consider the cost vector C of P. Each $C_x \subseteq C$ is also ordered and corresponds to a state in the cost state space. We define the following cost-based transitions:

Horizontal $(C_x) := C_y$ such that

$$C_y := C_x \cup \{c_{i+1}\},\ i = \max(\{k \mid k \text{ s.t. } c_k \in C_x\}) \text{ and } c_{i+1} \in C$$

In words, the Horizontal neighbor of a state C_x is derived by inserting the preference from C that immediately follows the lowest cost preference of C_x. Based on formulas (4), and (7), neighbor C_y has higher cost and higher degree of interest than C_x.

Vertical(C_x) :=

$$\{C_i \mid C_i := (C_x - \{c_i\}) \cup \{c_{i+1}\},\ c_{i+1} \in C,\ c_{i+1} \notin C_x, \text{ and } \text{cost}(C_i) \geq \text{cost}(C_{i+1}),\ \forall c_i \in C_x\}$$

Vertical neighbors of a state C_x are derived by replacing a preference in C_x by its successor from C provided that the latter is not already in C_x. Vertical neighbors are ordered in decreasing cost.

PROPOSITION 1. The destination of a transition from a source state C_x is also a state in the space.

DEFINITION 1. Nodes with the same number of preferences belong to a *group* with *group size* equal to the number of preferences.

Table 3. States of a graph

Group Size	States					
1	c_1	c_2	c_3	c_4		
2	c_1c_2	c_1c_3	c_2c_3	c_1c_4	c_2c_4	c_3c_4
3	$c_1c_2c_3$	$c_1c_2c_4$	$c_2c_3c_4$	$c_1c_3c_4$		
4	$c_1c_2c_3c_4$					

Assume $C = \{c_1, c_2, c_3, c_4\}$. The set of possible states is given in Table 3 (the initial query is omitted). Figure 4 shows the cost state space based on the transitions above. Vertical transitions are depicted in dashed lines, and Horizontal ones in solid lines. For instance, Horizontal$(c_1c_3) = c_1c_3c_4$ and Vertical$(c_1c_3) = \{c_1c_4, c_2c_3\}$.

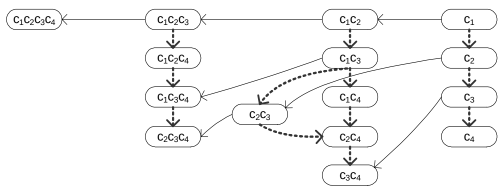

Figure 4. A Cost State Space

OBSERVATION 1. Both types of transitions are based on syntactic changes to a state that have known implications (increase/decrease) on state parameters. This generates syntax-based partial orders of states, which are exploited by the algorithms described below.

Table 4. Cost-based transitions

Transition	cost	doi
Vertical	↓	-
Horizontal	↑	↑

As Table 4 illustrates Horizontal moves towards nodes of higher doi and higher cost. Vertical moves towards nodes of lower cost and unknown doi. Consequently, one can devise algorithms that work with the cost vector C, and are roughly built around this idea: Horizontal *transitions can be applied up to the point where the cost of a node produced does not satisfy the cost constraint.* Vertical *moves are applied until the cost of a node produced satisfies the cost constraint.*

Algorithm C-BOUNDARIES. The first algorithm is based on the idea of finding a set of nodes that are not reachable from each other and satisfy the cost constraint, while their parents do not. These nodes are called *boundaries*. Boundaries on every group form a virtual borderline that partitions the cost state space into two sets of nodes: those satisfying the cost constraint, and those not. Then, the solution to the CQP problem under consideration is a node of maximum doi belonging to the first set.

Algorithm C-BOUNDARIES implements this idea in two phases. *In the first phase*, it searches for boundaries in every group, starting from the group of size 1. Search within a group is performed using Vertical transitions. From all boundaries found in one group, the algorithm moves to their Horizontal neighbors in the next group. From these new nodes, the algorithm starts searching for boundaries in their group. If no boundary is found in a group, this phase of the algorithm ends. *In the second phase*, the algorithm searches among the nodes below the boundaries to find the one with the best doi. In what follows, we give a set of propositions and a theorem that prove the correctness of the algorithm. Proofs are omitted due to space considerations.

PROPOSITION 2. All Vertical predecessors of a boundary do not satisfy the cost constraint.

PROPOSITION 3. All Vertical predecessors of a Horizontal neighbor of a boundary do not satisfy the cost constraint.

Based on the above, the following proposition can be proved.

PROPOSITION 4. By mapping boundaries of a group to their Horizontal neighbors in the next group, all nodes satisfying the cost constraint in the latter group, if any, are reached through Vertical transitions from the new nodes.

PROPOSITION 5. If there is no boundary in one group, then there are no boundaries in all groups of greater size either.

Based on the above, the following theorems can be proved.

THEOREM 1. The first phase of C-BOUNDARIES finds all boundaries.

THEOREM 2. Algorithm C-BOUNDARIES finds the optimal solution wrt a cost constraint, provided that one exists, i.e. it is correct.

The algorithm is provided in Figure 5. Its inputs are the query Q, the set of preferences P, its cardinality K, its cost vector C, and an upper cost bound c_{max} (the CQP constraint), and it generates a set of preferences P_U to be integrated into Q. The first phase of the algorithm is implemented by FINDBOUNDARY, and the second one by C_FINDMAXDOI. Note that the algorithm does not actually store the part of graph visited, hence, conserving memory.

FINDBOUNDARY constructs the set of boundaries, Boundaries, based on the input upper cost bound c_{max}. This set is ordered in decreasing group size. Based on OBSERVATION 1, each c_k in C may be represented by its index k, and each $C_x \subseteq C$ by the set of indices R of its member preferences. FINDBOUNDARY searches for boundaries in a breadth-first fashion, i.e. it finds boundaries in one group of states and then proceeds with boundaries in the next group. For this purpose, a queue RQ is used that maintains candidate nodes not yet examined. At each iteration, the first element R of RQ is obtained. At this point, the cost of the state corresponding to R is calculated (cost(R, C, P)). If this cost is lower than c_{max}, then R becomes a boundary (push(Boundaries, R)), and R's Horizontal neighbor is placed at the tail of RQ. If this cost is higher than c_{max}, then each Vertical neighbor of R is placed at the head of RQ. In this way, we first examine all states belonging to the same group and then proceed to the next group's states. If no boundaries are found in a group, then RQ becomes empty and FINDBOUNDARY stops.

Figure 6 shows an example of executing FINDBOUNDARY (note that all possible Vertical moves are shown in dashed lines for presentation reasons). For c_{max}=185, its output is {{1}, {1, 3}, {2, 3, 4}, {2, 4, 5}}, which corresponds to {c_1, c_1c_3, $c_2c_3c_4$, $c_2c_4c_5$}.

```
Algorithm C-BOUNDARIES
Input:   Q, P, K, C, $c_{max}$
Output: $P_U$, MaxDoi
1.    Boundaries := FINDBOUNDARY(Q, C, P, $c_{max}$);
2.    $P_U$ := C_FINDMAXDOI(Boundaries, C, K, P, MaxDoi);
```

```
function FINDBOUNDARY
Input:   Q, C, P,  $c_{max}$
Output: Boundaries
1.       R := {1}; Boundaries:= {};
2.       Enqueue(RQ, R);
3.       While RQ $\neq$ {}
3.1.       R :=Dequeue(RQ);
3.2.       If cost(Q, R, C, P) $\leq c_{max}$ then
3.2.1.       Boundaries:= push(Boundaries, R);
3.2.2.       R':= Horizontal(R);
3.2.3.       If R'$\neq${} then Enqueue(RQ, R') fi
           else
3.2.4        VR:= Vertical(R);
3.2.5        For each R' in VR
                 If prune(R') = FALSE then Enqueue(RQ, R') fi
             end for
           fi
         wend
```

```
function C_FINDMAXDOI
Input:   Boundaries, C, K, P
Output: $P_U$, MaxDoi
1.       MaxDoi:= 0; $P_U$ := {};
2.       $K_R$ := K; BestExpectedDoi := doi(P);
3.       For each R in Boundaries
3.1.       If count(R) < $K_R$ then
3.1.1        BestExpectedDoi := doi({$p_i$ | $p_i \in$ P, i $\in$ [1 … $K_R$]});
3.1.2.       If MaxDoi > BestExpectedDoi then exit fi;
           fi
3.2.       $K_R$ := count(R); $P_X$:= {}; Used={};
3.3.       For i:= $K_R$ to 1
3.3.1.       k:= R[i];
3.3.2.       $m_0$ := $min$({C[j] | j $\geq$ k, j $\leq$ K } – Used); Used = Used $\cup$ {$m_0$};
3.3.3.       $P_X$:= $P_X \cup$ {$p_{m_0}$};
           end for
3.4.       If doi($P_X$) > MaxDoi then MaxDoi := doi($P_X$); $P_U$ := $P_X$ fi
         end for
```

Figure 5. Algorithm C-BOUNDARIES

A couple of issues need to be taken into account during the traversal of the state space. The first one arises from not storing any part of the graph visited except for boundaries found. While leading to memory savings, this policy creates the risk of visiting parts of the graph previously seen. Another issue concerns visiting nodes below some boundary. For example, consider the graph of Figure 4. Assume the algorithm visits node c_1c_2 followed by c_1c_3. The latter has Vertical neighbors, c_1c_4 and c_2c_3. The algorithm classifies c_1c_4 as a boundary. As a result, its Vertical neighbors need not be visited. The algorithm proceeds with c_2c_3. However, all Vertical neighbors of this node are below the boundary previously found and, hence, need not be visited either. Moreover, by visiting these nodes, the algorithm will find a new boundary, c_2c_5. However, this one lies below boundary c_1c_4, i.e. it is reachable from the latter. This is in conflict with the definition of boundaries. Based on the above, we have implemented a method, prune(.), for pruning parts of the graph either because they have already been visited or because they are below boundaries found. The details are skipped for space reasons.

C_FINDMAXDOI searches below each boundary R found during the first phase for the node P_X, which may correspond to the boundary or some node below it, with the maximum doi. The boundaries are considered in order of decreasing group size. The output of this phase is P_U which is the node with the best doi (MaxDoi) among all P_X's encountered. In order to avoid examining all boundaries, BestExpectedDoi is used, which is an estimate of the best possible doi expected from groups not yet examined. As groups are considered in decreasing group size, BestExpectedDoi is the best doi expected by the largest group not yet considered. If BestExpectedDoi becomes worse than the best doi so far, MaxDoi, the algorithm stops.

Interestingly, in order to find node P_X given a boundary R, C_FINDMAXDOI does not make use of degrees of interest. Recall that each k stored in R, corresponds to c_k in C; c_k (C[k] in the algorithm) stores a pointer m_0 in P. Therefore, for each k stored in R, the algorithm replaces c_k with c_j in C ($k \leq j \leq K$) that has the minimum m_0. m_0 points at p_{m_0}, which is the preference with the best doi. These preferences constitute the best local solution P_X.

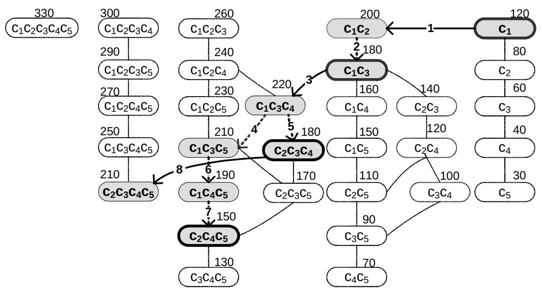

Figure 6. Example for C-BOUNDARIES (c_{max}=185)

A last point concerns function cost(Q, R, C, P). Each time it computes the cost of a node that is slightly different from a previous one. Since Formula (6) permits incremental cost computation, cost(.) has been implemented in this way. Costs that may be re-used are cached. This technique is used in all algorithms proposed. Note that the initial query Q is an input to this function, since the cost of a state depends upon Q.

Algorithm C-MAXBOUNDS. Observing the behavior and output of C-BOUNDARIES, it is clear that the set of boundaries found is a superset of those required for finding an optimal solution. One reason is that the algorithm generates boundaries on each group. As a result, boundaries in one group may be supersets of boundaries in groups searched earlier. For example, the output of FINDBOUNDARY in Figure 6 should not contain c_1, since this is a subset of c_1c_3, and, hence, has a lower doi than the latter. Thus, consideration of c_1 will never lead to better solutions than c_1c_3. Observing Figure 6, we discover another problem: FINDBOUNDARY classifies $c_2c_4c_5$ as a boundary and then discovers boundary $c_2c_3c_4$, which is, however, above the former. Obviously, $c_2c_4c_5$ has been wrongly identified as a boundary. If $c_2c_3c_4$ was found first, then $c_2c_4c_5$ would not have been visited in the first place.

To remedy these problems, C-MAXBOUNDS tries to build *maximal boundaries* such that none is subset of or reachable by another. It is a two-phase algorithm with the second phase being implemented in the same way as previously. *In the first phase*, the algorithm considers preferences in C one at a time. In each round, it starts from a state in the space that corresponds to the most expensive preference c_k in C not yet examined, and tries to build maximal boundaries that contain this preference. For this purpose, FINDMAXBOUND is called which treats the input node as a "seed". In order to generate a maximal boundary from a "seed" node, FINDMAXBOUND applies horizontal transitions up to the point where the cost of a node produced does not satisfy the cost constraint. The last node in this sequence that satisfies the constraint is a maximal boundary. Then, FINDMAXBOUND searches among Vertical neighbors of this node. The top-most ones that satisfy the constraint are used as new seeds and the whole process is repeated as long as there are meaningful transitions to perform. If FINDMAXBOUND returns a maximal boundary that includes all preferences following c_k in C, then the first phase of the algorithm ends, since any subsequent boundaries will be subsets of that boundary.

In order to build maximal boundaries, the philosophy of the algorithm is to insert as many preferences as possible into a given set of preferences before storing it as a maximal boundary. For this purpose, it uses a slightly different Horizontal.

Horizontal2(C_x) :=
$\{C_i \mid C_i := C_x \cup \{c_i\},\ c_i \in C,\ c_i \notin C_x$
and $\text{cost}(C_i) \geq \text{cost}(C_{i+1}),\ \forall c_i \in C\}$

Horizontal2 neighbors are ordered in decreasing cost. In Figure 8, Horizontal2(c_2) = {c_1c_2, c_2c_3, c_2c_4, c_2c_5}.

```
Algorithm C-MAXBOUNDS
Input:   Q, P, K, C, $c_{max}$
Output: $P_U$, MaxDoi
1.    MaxBounds:= {}; LastSolutionSize :=0;
2.    k :=1;
3.    While k + LastSolutionSize ≤ K
3.1.    R := {k};
3.2.    MaxBounds:= FINDMAXBOUND(Q, k, R, C, MaxBounds, $c_{max}$);
3.3.    LastSolutionSize:= count(head(MaxBounds)); k:= k +1;
      wend
4.    $P_U$ := C_FINDMAXDOI(MaxBounds, C, K, P, MaxDoi);
```

```
function FINDMAXBOUND
Input:   Q, index k, Node R, C, MaxBounds, $c_{max}$
Output: MaxBounds
1.       Enqueue(RQ, R);
2.       While RQ ≠ {}
2.1.       R :=Dequeue(RQ); $R_0$ := R;
2.2.       HR:= Horizontal2(R);
2.3.       While HR ≠ {}
2.3.1.       For each R' in HR
2.3.2.         If cost(Q, R', C, P) ≤ $c_{max}$ then R := R'; exit for fi
             end for
2.3.3.       HR:= Horizontal2(R);
           wend
2.4.       If R ≠ $R_0$ then MaxBounds:= push(MaxBounds, R) fi
2.5.       VR:= Vertical(R);
           For each R' in VR
             If R' ∩ { k } = {} then exit for fi
             If prune(R') = FALSE then Enqueue(RQ, R') fi
           end for
         wend
```

Figure 7. Algorithm C-MAXBOUNDS

The algorithm is presented in Figure 7. Its inputs are the query Q, the set P, its cardinality K, its vector C, and the upper cost bound

c_{max}. It generates the set of preferences P_U to be integrated into Q. Note that the second phase of the algorithm is implemented by C_FINDMAXDOI as described earlier. Subsequently, we describe only the first phase in more detail. As in the case of C-BOUNDARIES, based on OBSERVATION 1, each c_k in C may be represented by its index k, and each $C_x \subseteq C$ by the set of indices R of its member preferences. In each round, the algorithm starts with R containing an index k to the most expensive preference in C not yet examined. R is passed to FINDMAXBOUND, which constructs a set of maximal boundaries including (the index to) c_k. MaxBounds is the set of all maximal boundaries found by successive calls of FINDMAXBOUND, and is ordered in decreasing group size. If FINDMAXBOUND returns a maximal boundary that includes all preferences in C not yet examined, then the first phase of the algorithm ends. For this purpose, LastSolutionSize is the largest group size found in MaxBounds. If k+LastSolutionSize is greater than K, then the first phase terminates.

FINDMAXBOUND maintains a queue RQ of nodes not yet examined. This initially contains the node provided as input to the function. In each round, the first element R of RQ is obtained, and its Horizontal2 neighbors are examined. If there is one that satisfies the cost constraint, then FINDMAXBOUND examines the Horizontal2 neighbors of that node. This process is repeated as long as a neighbor that satisfies the cost constraint is found. At the end, if the resulting node is different from the node initially obtained from RQ, the former becomes a maximal boundary. The Vertical nodes of this node, provided that they subsume the node used as input to the function, are inserted in RQ.

Figure 8 shows an example of the first phase of C-MAXBOUNDS. The output for c_{max}=185 is {c_1c_3, $c_2c_3c_4$}, a strict subset of FINDBOUNDARY's solution, depicted in Figure 6.

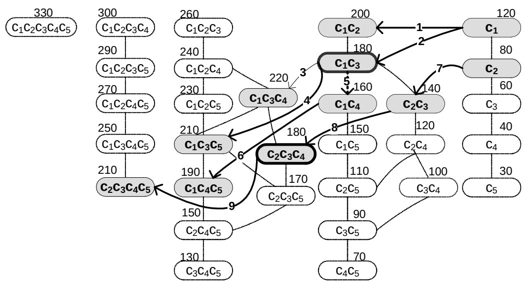

Figure 8. Example for C-MaxBounds (c_{max}=185)

5.2.2 *Algorithms on the Doi State Space*

Consider the doi vector D of P. Each $D_x \subseteq D$ is also ordered and corresponds to a state in the doi state space. Then, by replacing C with D in the definitions of transitions in the previous section, we obtain a set of doi-based transitions, and a similar doi-based state space. As Table 5 shows, doi-based Horizontal moves towards nodes of greater doi and higher cost, and Vertical moves towards nodes of lower doi and unknown cost.

Table 5. Doi-based transitions

Transition	cost	doi
Vertical	-	↓
Horizontal	↑	↑

The algorithms that we propose for the doi-based space follow a similar philosophy to the previous ones, i.e. Horizontal *transitions are applied up to the point that the cost of a node produced does not satisfy the cost constraint.* Vertical *transitions are applied until the cost of the resulting node satisfies the constraint.*

Algorithm D-MAXDOI. It is a two-phase algorithm following an approach similar to C-BOUNDARIES, thus we will only provide a sketch of its functionality. The algorithm is described in Figure 9.

```
Algorithm D-MAXDOI
Input:   Q, P, K, $c_{max}$
Output: $P_U$, MaxDoi
1.    Solutions := FINDOPTIMAL(Q, P, $c_{max}$);
2.    $P_U$ := D_FINDMAXDOI(Solutions, K, P, MaxDoi);
```

```
function FINDOPTIMAL
Input:   Q, P, $c_{max}$
Output: Solutions
1.       R := {1}; Solutions:= {};
2.       Enqueue(RQ, R);
3.       While RQ ≠{}
3.1.       R :=Dequeue(RQ);
3.2.       If cost(Q, R, P) ≤ $c_{max}$ then
3.2.1.       R':= Horizontal(R);
3.2.2.       While cost(Q, R', P) ≤ $c_{max}$
                R := R'; R':= Horizontal(R);
             wend
3.2.3.       Solutions:= push(Solutions, R);
           fi
3.3.       VR:= Vertical(R');
3.4.       For each R'' in VR
             If prune(R'') = FALSE then Enqueue(RQ, R'') fi
           end for
         wend
```

```
function D_FINDMAXDOI
Input:   Solutions, K, P
Output: $P_U$, MaxDoi
1.       MaxDoi:= 0; $P_U$ := {};
2.       $K_R$ := K; BestExpectedDoi := doi(P);
3.       For each R in Solutions
3.1.       If count(R) < $K_R$ then
3.1.1          BestExpectedDoi := doi({$p_i$ | $p_i \in$ P, i $\in$ [1 … $K_R$]});
3.1.2.         If MaxDoi > BestExpectedDoi then exit fi;
           fi
3.2.       If doi(R) > MaxDoi then
               MaxDoi:= doi(R); $P_U$:={$p_i$| $p_i \in$ P, $\forall$i $\in$ R} fi
         end for
```

Figure 9. Algorithm D-MaxDoi

Its inputs are the query Q, the set P, its cardinality K, and the upper cost bound c_{max}. It generates the set of preferences P_U to be integrated into Q. *In the first phase*, FINDOPTIMAL builds a set of possible CQP solutions, namely Solutions. The basic idea is the following. There is a queue RQ keeping nodes not yet examined. It initially contains the node corresponding to the most interesting preference. In each round, a node is picked from the queue, and Horizontal transitions are applied up to the point where the cost of a node produced does not satisfy the cost constraint. The last node in this sequence that satisfies the constraint is a possible solution to the CQP problem and is inserted into Solutions. Its Horizontal successor, which does not satisfy the cost constraint, is then considered. That node's Vertical neighbors are added into RQ. *During the second phase*, D_FINDMAXDOI searches among Solutions for the optimal one following a philosophy similar to C_FINDMAXDOI. Its implementation is though simpler as Figure 9 shows. Note that this algorithm, as well as the rest of the algorithms of this subsection, works with R, which is a set of indexes, such that each k in R corresponds to a preference p_k in P. The following theorem may be proved in a similar way as THEOREM 2.

THEOREM 3. The algorithm D-MAXDOI finds the optimal solution wrt a cost constraint, provided that one exists, i.e. it is correct.

A possible advantage of working with doi based transitions is that Preference Space needs not produce C thus saving time. This is investigated in the experiments.

Algorithm D-SINGLEMAXDOI. This algorithm is based on the same idea as C-MAXBOUNDS, i.e. it follows a greedy approach trying to build maximal sets of preferences. A basic difference is that it has only one phase, during which, it keeps track of the best so far solution, P_U, which has degree of interest MaxDoi. Furthermore, it keeps BestExpectedDoi that is the best degree expected from the part of the graph not yet examined. If the maximum degree of interest so far, MaxDoi, is higher than the best expected degree, no more staes should be examined. The algorithm is given in Figure 10.

```
Algorithm D-SINGLEMAXDOI
Input:  Q, P, K, $c_{max}$
Output: $P_U$, MaxDoi
1.      MaxDoi:= 0; BestExpectedDoi := doi(P);
2.      k := 1;
3.      While MaxDoi ≤ BestExpectedDoi
3.1.      R := { k };
3.2.      Enqueue(RQ, R);
3.3.      While RQ ≠{}
3.3.1.      R :=Dequeue(RQ); $R_0$ := R; HR:= Horizontal2(R);
3.3.2.      While HR ≠ {}
              For each R' in HR
                If cost(Q, R', P) ≤ $c_{max}$ then R := R'; exit for; fi
              end for
              HR:= Horizontal2(R);
            wend
3.3.3.      If doi(R)> MaxDoi then
              MaxDoi:= doi(R); $P_U$:={$p_i$ | $p_i \in$ P, $\forall$i $\in$ R} fi
3.3.4.      VR:= Vertical(R);
3.3.5.      For each R' in VR
              If R' ∩ { k } = {} then exit for fi
              If prune(R') = FALSE then Enqueue(RQ, R') fi
            end for
          wend
3.4.      BestExpectedDoi := doi({$p_j$| $p_j \in$ P, j ∈ [k … K]}); k:= k +1;
        wend
```

Figure 10. Algorithm D-SINGLEMAXDOI

Algorithm D-HEURDOI. This algorithm is built on the same idea as the previous one. Its differences lie in the use of heuristics for reducing the number of states examined. The basic idea is the following: in each round, the algorithm picks the most expensive preference p_k in P not yet examined. Then, these steps are followed. (*a*) Its Horizontal2 neighbors are successively examined. If one of them satisfies the cost constraint, then it becomes the current node and the same procedure is repeated for this node. At the end, if the degree of interest of the current node is greater than the maximum degree of interest so far, MaxDoi, the current node becomes the best solution known. (*b*) Heuristics are used for obtaining possibly better solutions than the current one: we remove the cheapest preference from the current node. The resulting node becomes the current one. We perform the same process for this node as in step (*a*). This step is repeated until the current node is reduced to the initial preference p_k in P.

```
Algorithm D-HEURDOI
Input:  Q, P, $c_{max}$
Output: $P_U$, MaxDoi
1.      MaxDoi:= 0; BestExpectedDoi := doi(P); k :=1;
2.      While MaxDoi ≤ BestExpectedDoi
2.1.      $R_x$ := {k}; HR:= Horizontal2(R);
2.2.      While HR ≠ {}
2.2.1.      For each R' in HR
              If cost(Q, R', P) ≤ $c_{max}$ then R := R'; exit for fi
            end for
2.2.2.      HR:= Horizontal2(R);
          wend
2.3.      if doi(R)> MaxDoi then
            MaxDoi:= doi(R); $P_U$:={$p_i$ | $p_i \in$ P, $\forall$i $\in$ R} fi
2.4.      $K_R$ := count(R);
2.5.      For k:= $K_R$ to 2
2.5.1.      R':= { R[j] | $\forall$j < k }; HR:= Horizontal2(R');
2.5.2.      While HR ≠ {}
              For each R'' in HR, R'' ≠ R'
                If cost(Q, R'', P) ≤ $c_{max}$ then R' := R''; exit for fi
              end for
              HR:= Horizontal2(R');
            wend
2.5.3.      if doi(R')> MaxDoi then
              MaxDoi:= doi(R'); $P_U$:={$p_i$ | $p_i \in$ P, $\forall$i $\in$ R'} fi
          end for
2.6.      BestExpectedDoi := doi({$p_j$| $p_j \in$ P, j ∈ [k … K]}); k:= k +1;
        wend
```

Figure 11. Algorithm D-HEURDOI

6. OTHER CQP PROBLEMS

As indicated earlier, all CQP problems presented in Table 1 are quite similar to each other, both in terms of their formulation but also in terms of characteristic properties of the query parameters that are involved in them. In particular, formulas (4), (7), and (8), lead to very similar partial orders for each parameter that can be derived based on purely syntactic transformations of personalized queries. Hence, for all problems in Table 1, it is essentially the same kind of state spaces that are available for search with analogous states and transitions. Moreover, given the particular approaches to estimation of these parameters that we have chosen, incremental computation of their values is available in all cases. Therefore, all algorithms presented in Section 5 are applicable in all CQP problems. The only adaptation that is required in each case is making the appropriate choice of the direction of Horizontal and Vertical transitions. This is influenced by three factors: whether the CQP problem requires minimization or maximization; whether the parameter of concern increases or decreases with the group size; and whether the CQP constraints are in the form of an upper or a lower bound. For every case, the appropriate choice for Horizontal and Vertical direction generates the same search problem as before.

As an example, consider Problem 1, where doi is optimized again, but this time there are both upper and lower bound constraints on the result size. In this case, we use vector S as the main representation form of the states in the search space, containing preferences in order of result size. Horizontal transitions move towards states of smaller result size and of higher degree of interest. Vertical transitions move towards states of larger size, without knowing how degrees of interest change. Essentially, taking into account the properties of result size, we have reversed the direction of the two categories of transitions compared to Problem 2 so that the search must follow exactly the same moves as before; hence, we can employ the algorithms of subsection 5.2.1. If both an upper and a lower bound for size exist, then a slight enhancement is required. In particular, two lists of boundaries are generated by all algorithms. In the first phase, the algorithm first finds a boundary corresponding to the upper limit. This is added to the first list, UpBoundaries. Then, instead of searching for the next upper boundary, it continues searching in the same group, as if the first boundary were not found, until a second boundary corresponding to the lower bound is found, if such exists. This is kept in a second list, LowBoundaries. In this way, pairs of boundaries are produced. In the second phase, the algorithm checks the nodes between the upper and lower boundaries in order to find the one with the best doi.

As another, more general example, consider Problem 3, where doi is maximized under constraints on both cost and size. In this case, we may use either one of the algorithms, on the size or the cost space. Assume that in the first phase, we use the algorithm searching the cost state space. In the second phase, the algorithm keeps track of the solution with the currently maximum degree of interest that also satisfies the cost constraint. If a solution is found that also satisfies the size constraints, then this becomes the current optimal solution. The algorithm stops when no solution with better degree of interest can be found.

7. EXPERIMENTAL RESULTS

We have implemented the algorithms described above on top of Oracle 9i and have conducted experiments to compare their efficiency and effectiveness. Our data was from the Internet Movies Database [7]. We adopted the evaluation setting of [12], which is not repeated for lack of space. It includes a broad range of doi values, doi-value deviations, queries, etc. We only discuss the experimental results for the CQP algorithms of Section 5.2; similar results were obtained for the other CQP problems as well.

7.1 Implementation Issues and Setup

In our experiments, we have used the following formulas for the estimation of the key parameters of a personalized query.

Degree of Interest. For the doi of an implicit preference p, we have chosen multiplication as the function $f_\otimes$ in Formula (1) [12]:

$$\text{doi}(p) = \text{doi}(p_1) \times \ldots \times \text{doi}(p_m). \qquad (9)$$

For the doi of a conjunction of preferences, we have used the following function [12]:

$$\text{doi}(P_x) = 1 - \prod_{\forall p_i \in P_x}(1 - \text{doi}(p_i)) \qquad (10)$$

Note that both formulas can be incrementally computed.

Execution Cost. For the execution cost of a personalized query, we have used formula (6), repeated here for convenience:

$$\text{cost}(Q_x) = \text{cost}(\cup(q_i)) = \sum_{q_i \in Q_x} \text{cost}(q_i)$$

For simplicity of exposition, we have made the following assumptions for each sub-query q_i: (*a*) cost(q_i) is measured in terms of I/O only; (*b*) there is enough memory to store all data required by a query until its completion; (*c*) there are no indexes used. Based on the above, execution cost is simply the cost of reading from disk all required data once. Hence, the execution cost of a sub-query q_i on relations $R_{i1}, \ldots R_{iN}$, is estimated as

$$\text{cost}(q_i) = b \times \sum_{R_{ij} \in q_i} \text{blocks}(R_{ij}) \qquad (11)$$

where blocks(R_{ij}) is the number of blocks of relation R_{ij}, and b is the time to read a single block from disk into memory. For our experiments, we have considered b = 1ms.

7.2 Comparison of Algorithms

We have compared the algorithms described above on their memory requirements, execution times, and quality of solution found. Two parameters affect their behavior: (*a*) the number of preferences K extracted from the profile and used by a CQP algorithm and (*b*) the upper bound c_{max} on the execution time. We have experimented with values of K in [10, 40] and values of c_{max} from 10% to 100% of the execution time of the query that incorporates all K preferences into the original query (Supreme Cost), which is the most expensive query based on our cost assumptions. As defaults, we have used a constant c_{max}=400ms and K =20. Each result shown below represents the average of 200 different experiment runs (20 profiles × 10 queries) with the same characteristics.

7.2.1 Execution Times

Figure 12(a) presents execution times of the CQP algorithms discussed earlier, as a function of K. Naturally, all algorithms have growing execution times with K, but they can be clearly partitioned into two classes. In the first class are D-MAXDOI and D-SINGLEMAXDOI, whose poor behavior is due to their use of doi-based transitions: while Horizontal transitions move them towards states of greater interest and cost, Vertical transitions are "blind" with respect to execution cost, resulting in unevenly larger parts of the search space being explored as K increases. Although being cost-based, C-BOUNDARIES also exhibits poor performance for higher K primarily due to its producing a large superset of the Boundaries actually required to find the optimal solution. In the second class are C-MAXBOUNDS and D-HEURDOI. Both use heuristics to avoid the pitfalls of the others and explore a small part of the search space, showing excellent performance.

In principle, in addition to the time required by the CQP algorithms to run, one should take into account the time taken by the 'Preference Space' module as well (Figure 2) to create all K preferences to be investigated. Figure 12(b) presents this for a range of values of K for two different algorithms: D_PrefSelTime producing preferences ordered only on doi (useful for doi-based CQP algorithms), and C_PrefSelTime, producing preferences ordered both on doi and cost (useful for cost-based CQP algorithms). Clearly, for all practical purposes and all cases, the time required by the 'Preference Space' module is negligible compared to that of the CQP algorithms and can be ignored.

Figure 12(c) presents execution times of CQP algorithms for different values of c_{max}, while Figure 12(d) zooms in on the fastest algorithms to distinguish among them. For all algorithms, execution time grows up to about c_{max}=50% of the Supreme Cost and then shrinks as c_{max} moves to 100%. The reason is that, for low c_{max}, all algorithms stay primarily within groups of small size, where cost remains low. Thus, few Horizontal transitions are performed. As c_{max} increases, more Horizontal and Vertical transitions are performed and more groups and states within them are explored. Maximum time is reached at about c_{max}=50% of Supreme Cost, beyond which the number of Vertical transitions taken decreases, as algorithms search within groups of large size only, bringing execution time down again. The only practical methods remain C-MAXBOUNDS and D-HEURDOI, with the latter being extremely efficient and almost unaffected by c_{max}.

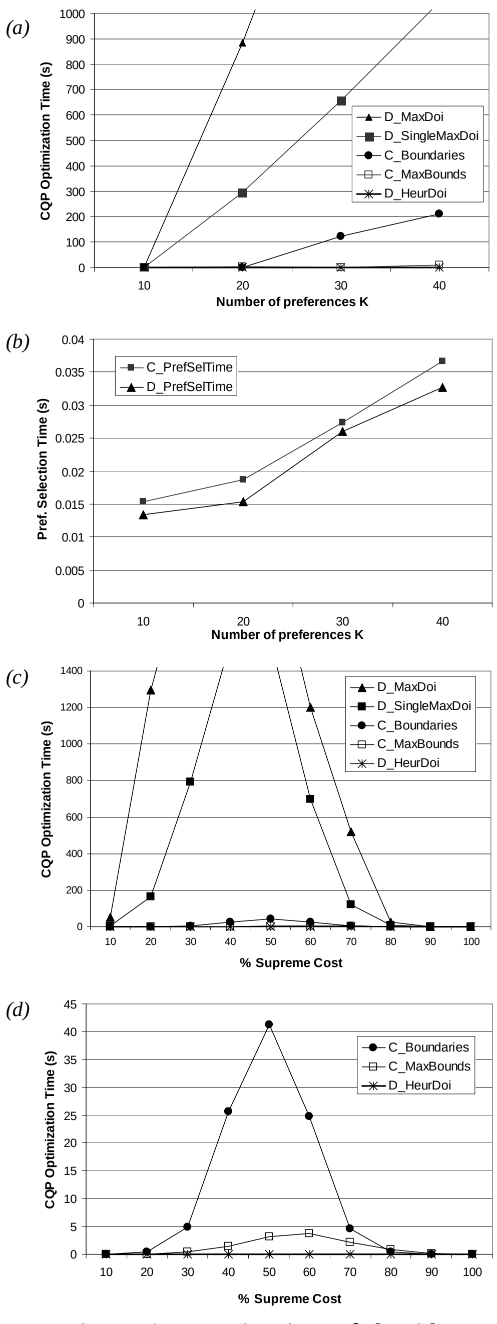

Figure 12. Execution times of algorithms

7.2.2 Memory Requirements

We have measured the maximum memory used by a CQP algorithm during its execution. Figure 13(a) and (b) present these measurements for different values of K and c_{max}, respectively. The memory requirements of all algorithms are commensurate with their execution costs for the exactly same reasons. D-MAXDOI and D-SINGLEMAXDOI are memory-hungry, C-BOUNDARIES is better but deteriorates relatively quickly with K, while C-MAXBOUNDS and D-HEURDOI have very low memory consumption and remain essentially unaffected by K. With respect to c_{max}, the diagrams are very similar to those for memory requirements for the same reasons analyzed there. Overall, we observe that even the worst algorithms have rather small memory requirements, so this is not an issue of concern for CQP.

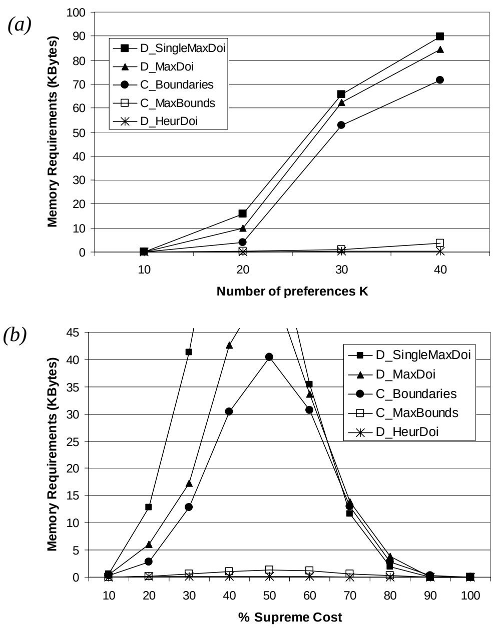

Figure 13. Memory requirements of the algorithms

7.2.3 Quality of Solution

Given that C-BOUNDARIES and D-MAXDOI are the only provably correct CQP algorithms, it is critical to experimentally evaluate the remaining, heuristic algorithms based on the doi of the solutions they find. For this, we use the difference

$$\text{Quality} = \text{doi}_{optimal} - \text{doi}_{found}$$

where $\text{doi}_{optimal}$ is the degree of interest of the optimal solution (found by D-MAXDOI), and doi_{found} is the degree of interest of the solution found by a heuristic CQP algorithm. Figure 14(a) and (b) compare CQP algorithms on Quality for different values of K and c_{max}, respectively. Based on both figures, we may conclude that all heuristic algorithms produce solutions of the highest quality, with miniscule differences from their deterministic counterparts. (Note that y-axes represent scalars multiplied by 10^7). Hence, in combination with their execution times, C-MAXBOUNDS and D-MAXDOI appear as excellent choices for CQP problems.

Note that, even for the lowest values of K or c_{max}, where there is some observable difference in quality among the algorithms, this is quite small. This is partly due to the model adopted for the doi of conjunctive preferences (Formula (10)), whose output value increases rapidly as more preferences are taken into account, which is the case when either K or c_{max} increases. Using a different model for conjunctive preferences would still exhibit the same growing trends but might have resulted in larger differences among approaches.

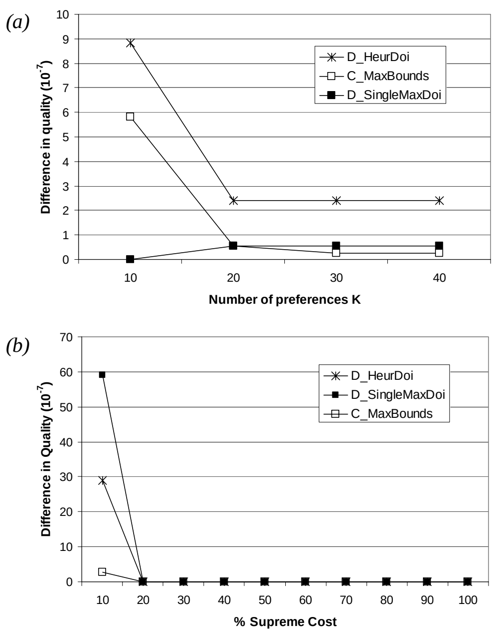

Figure 14. Comparison of solutions of algorithms

7.3 Personalized Query Cost Prediction

As a last aspect of our experimental effort, it is interesting to validate the simplified query cost model that we have adopted (Sections 4 and 7). Figure 15 shows that the costs returned by our formulas are very close to the actual observed ones when queries were executed, thus strengthening the usability of our approach.

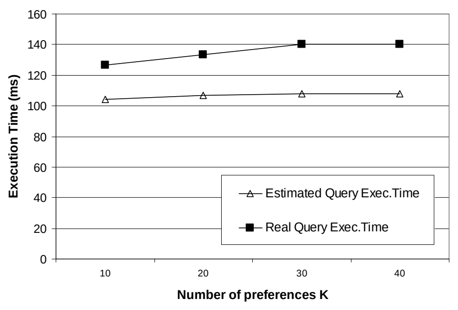

Figure 15. Cost Evaluation

8. CONCLUSIONS AND FUTURE WORK

In this paper, we have generalized earlier concepts of query personalization and have introduced Constrained Query Personalization (CQP) as a family of related optimization problems with constraints. We have formulated CQP as a state-space search problem and have devised state-space search algorithms that take advantage of particular characteristics of the CQP problems. Finally, we have demonstrated the effectiveness of our approach through experimental results that evaluate the algorithms proposed with respect to several important features. In ongoing work, we are concerned with policies mapping the search context onto the appropriate CQP problem. In that spirit, we are particularly interested in integrating CQP with location-based services so that richer forms of CQP may be devised. Finally, we are interested in studying query personalization as a multi-objective constrained optimization problem, where more than one query parameter may be optimized simultaneously.

9. REFERENCES

[1] Bruno, N., Chaudhuri, S., Gravano, L. Top- k Selection Queries over Relational Databases: Mapping Strategies and Performance Evaluation. *ACM TODS*, 27(2), 153-187, 2002.

[2] Chaudhuri, S., Gravano, L. Evaluating Top-k Selection Queries. Proc. of the 25th Int'l Conf. On VLDB, 1999.

[3] Ganguly, S., Hasan, W., Krishnamurthy, R. Query Optimization for Parallel Execution. Proc. Of the ACM Int'l Conf. On Management of Data, SIGMOD, 1992.

[4] Glover F. Tabu Search - Part I. *ORSA Journal on Computin*g, Vol. 1, pp. 190-206, 1989.

[5] Goldberg D. *Genetic Algorithms in Search and Machine Learnin*g. Reading, Addison Wesley, 1989.

[6] Ilyas, I., Aref, W. G., Elmagarmid, A. Supporting top-k join queries in relational databases. VLDB J. 13(3): 2004.

[7] Internet Movies Database. Available at www.imdb.com.

[8] Karypis, G. Evaluation of Item-Based Top-N Recommendation Algorithms. Proc. of CIKM, 247-254, 2001.

[9] Kellerer, H., Pferschy, U., Pisinger, D. Knapsack Problems. Springer-Verlag, 2003.

[10] Kirkpatrick, Gelatt, C. D., Vecchi, M. P. Optimization by Simulated Annealing, Science 220, 4598, 671-680, 1983.

[11] Kossmann, D., Stocker, K. Iterative Dynamic Programming: A New Class of Query Optimization Algorithms. ACM TODS, 25(1), 2000, 43–82.

[12] Koutrika, G. Ioannidis, Y. Personalization of Queries in Database systems. Proc. of ICDE, 2004.

[13] Liu F., Yu C., Meng W. Personalized Web Search by Mapping User Queries to Categories. Proc. of CIKM, 2002.

[14] Pitkow, J., Schutze, H., et al. Personalized Search. *Comm. of the ACM*, 45(9), 2002.

[15] Selinger, P.G., Astrahan, M.M., Lorie, R.A., Price, T. G. Access Path Selection in a Relational Database Management System. Proc. of SIGMOD, 1979.