The Hexagon Algorithm for Pareto Preference Queries

Timotheus Preisinger
University of Augsburg
Institute of Computer Science
Germany
preisinger@informatik.uni-augsburg.de

Werner Kießling
University of Augsburg
Institute of Computer Science
Germany
kiessling@informatik.uni-augsburg.de

ABSTRACT

Database queries expressing user preferences have been found to be crucial for personalized applications. Such preference queries, in particular Pareto preference queries, pose new optimization challenges for efficient evaluation. So far however, all known generic Pareto evaluation algorithms suffer from non-linear worst case runtimes. Here we present the first generic algorithm, called *Hexagon*, with linear worst case complexity for any data distribution under certain reasonable assumptions. In addition, our performance investigations provide evidence that *Hexagon* also beats competing Block-Nested-Loop style algorithms in the average case. Therefore *Hexagon* has the potential to become one key algorithm in each preference query optimizer's repertoire.

1. INTRODUCTION

Preferences have always been an important natural concept in real life. In computer science, this importance has been missed out a long time. But this has changed during the last years. Preferences in computer science form popular research topics not only in database technology – on which we will focus in this paper, but also in fields as AI ([15]) and constraint logic programming.

Preferences and their integration with databases have been in focus for some time by now, leading to diverse approaches. We will follow Kießling's way ([8, 9]) and look at a preference as $P = (A, <_P)$, where A is a set of attributes and $<_P$ is a *strict partial order* on the domain of A. One implementation of this preference model is an extension of SQL, *Preference SQL* ([10]).

Figure 1 shows the usage of user preferences in a database query, expressing some user preferences on a hotel after the keyword `PREFERRING`. It is a Pareto preference (`AND`) consisting of base preferences on the price (AROUND) and the hotel category (POS1), and a prioritization (`PRIOR TO`) stating that having a jacuzzi (POS2) is more important than a fitness room (POS3).

The result of a preference query consists of *best matches only* ([8]). For a preference $P = (A, <_P)$ on an input relation R these are all the input tuples that are not dominated wrt. $<_P$. This *best matches only* (BMO) result is computed by the *preference selection operator* $\sigma[P](R)$ (called *winnow* in [2]):

$$\sigma[P](R) := \{t \in R \mid \nexists\, t' \in R : t[A] <_P t'[A]\}$$

The Preference SQL query we have seen in figure 1 can be translated into the operator tree in figure 2. Together with the need for preference queries of course came the need for their efficient evaluation. For the scope of this paper we investigate the evaluation of Pareto preferences[1]. Many algorithms have been developed in the context of *skyline queries* introduced in [1]. As Pareto preferences form a superset of these, many skyline algorithms can be adapted for Pareto preference processing. Generally, there are two types of algorithms: those depending on index structures ([14, 11, 12]) and generic ones ([1, 2, 13]). Index based algorithms tend to be faster, but are less flexible - they are designed for flat query structures and have a high maintenance overhead associated with database updates. In general, they cannot

```
SELECT *
FROM   hotel h, features f
WHERE  c.id = f.hotel_id
PREFERRING
  c.price AROUND 80, 10 AND
  c.category IS '****'  AND
    (f.jacuzzi IS 'YES' PRIOR TO
     f.fitness_room IS 'YES')
```

Figure 1: Sample Preference SQL query

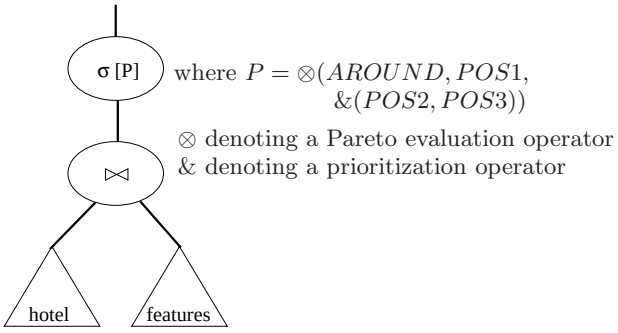

Figure 2: Operator tree for figure 1's preference

[1]We will not consider algebraic preference query optimization techniques here. For such, please refer to [7, 2].

VLDB '07, September 23-28, 2007, Vienna, Austria.

deal with algebraic representations of the *preferring* part of the Preference SQL queries, i.e. complex operator trees as we see one matching our query in figure 2. On the other hand, the generic algorithms show linear average case running times wrt. to the size of the input relation, in a worst-case scenario this may degenerate to quadratic time. As generic algorithms work without preparations on any kind of input relation, we will not distinguish between base relations and dynamically produced intermediate relations like join results.

Our goal is a completely new type of algorithm for evaluating Pareto preference queries in linear worst case time for any data distribution. As we will see, this goal can be reached if certain reasonable assumptions on the type of base preferences are satisfied.

In section 2, we will now give a brief introduction to our preference model. Then we will look at "better-than" graphs, a visualisation method for preferences, in section 3. Analyzing the structure of these graphs will afterwards lead us to our new *Hexagon* algorithm in section 4. In section 5, we will conclude with some theoretical and experimental performance analyses showing the power of *Hexagon*.

2. BACKGROUND

After having a look at our preference model and some sample preferences constructors on single attributes we will see how such preferences can be combined to form complex preferences. At the end of the section we will review the state of the art of Pareto preference evaluation algorithms.

2.1 Preference Constructors

The preference model of [8, 9] proposes to use intuitive preference constructors for preference definition. Before we have a look onto some of them, let's state an important subclass of strict partial orders. Given $<_P$, the *indifference relation* $\sim_P$ is defined as:

$$x \sim_P y \Leftrightarrow \neg(x <_P y \vee y <_P x)$$

► ***Weak Order Preference (WOP)***

Following [5], a *weak order preference* (*WOP*) is a strict partial order, in which indifference is transitive. For each WOP we can define a *level* function that can be used to determine dominance between two values:

LEMMA 1. *For a WOP $P = (A, <_P)$, we can find a level function to determine dominance between two values:*

$$level : dom(A) \to \mathbb{N}_0$$
$$x <_P y \Longleftrightarrow level_P(x) > level_P(y)$$

A proof of this lemma is given in [13]. Of course the definition of the level function depends on the type of preference it is used for. The maximum level value for a preference P is called $\max(P)$.

► ***Base Preference Constructors***

Preferences on single attributes are called *base preferences*. Usually they can be defined as WOPs. We will now have a look at some base preference constructors and their level functions. There are base preference constructors for continuous and for discrete domains. For both types we will now see some practically very important samples, all of them being WOPs.

a) Categorical base preference constructors

We will look at the *POS preference* as a sample for this type. It's constructor is *POS*(*A*, *POS-set*).

A POS-preference states that the user has a set of preferred values – the *POS-set* – in the domain of *A*. The level function reflects the preference of elements of the *POS-set* over other values:

$$level_{\mathrm{POS}}(x) := \begin{cases} 0 & \text{iff } x \in POS\text{-}set \\ 1 & \text{iff } x \notin POS\text{-}set \end{cases}$$

The sample query in figure 1 shows three POS preferences with singleton *POS-sets* for hotel category and flags indicating the existance of a jacuzzi and a fitness room.

The generalization of the POS preference is the LAYERED preference introduced in [9]. It enables users to specify any number of different sets. Each set is preferred more or less than all others (and so having a unique integer level value). For all categorical base preferences, the maximum level value is settled by the definition of the preference constructor. For POS preferences, this value clearly is 1.

b) Numerical base preference constructors

Continous numerical domains need a different type of preferences. In order to map the domain to discrete values, the *d-parameter* ($d > 0$) has been introduced in [9].

A preference $\mathrm{AROUND}_d(A, value)$ is used to specify a preferred numeric value. From two values, the one with less deviation from the specified value is preferred. We see this in the level function:

$$level_{\mathrm{AROUND}_d}(x) := \left\lceil \frac{|value - x|}{d} \right\rceil$$

Only a perfect match has the level value zero. The maximum level value is subject to the domain: it is defined by the extremal values of the domain. For product prices e.g. there is always a minimum of zero.

The query in figure 1 shows an AROUND preference for the price. It should be around 80 in the example; differences of up to 10 do not matter.

► ***Complex Preference Constructors***

There is the need to combine several of base preferences into more complex structures. One way is to list a number of preferences that all are equally important to the user. This is the concept of Pareto preferences.

DEFINITION 1. *Pareto preference constructor*
For WOPs $P_1 = (A_1, <_{P_1})$, ..., $P_m = (A_m, <_{P_m})$,
a Pareto preference

$$P = \otimes(P_1, \ldots, P_m) = (A_1 \times \cdots \times A_m, <_P)$$

is defined as:

$$(x_1, \ldots, x_m) <_P (y_1, \ldots, y_m) \text{ iff}$$
$$\exists i \in \{1, 2, \ldots, m\} : level_{P_i}(x_i) > level_{P_i}(y_i) \wedge$$
$$\forall j \in \{1, 2, \ldots, m\} : level_{P_j}(x_j) \geq level_{P_j}(y_j)$$

Note that we restrict our attention here to WOPs as input preferences for P. According to [9] strict partial order property is guaranteed. All values mapping to the same level value are considered as substitutable and are treated as one *equivalence class*.

Our preference model knows prioritized combination of preferences and numerical rankings as well ([8]). As evalua-

tions for such constructors lead to different problems, we do not regard them here.

At this point it is only important to observe that prioritizations over WOPs yield WOPs ([2]). Numerical rankings of course yield weak orders, too. So in figure 2, $\otimes$ is an instance of our Pareto preference constructor.

2.2 Pareto Preference Query Evaluation

Finding a Pareto preference's BMO set is a generalization of finding the best elements of a set of multi-dimensional values. This has been investigated for a long time and was brought in a database context with the *skyline* operator in [1]. Since then, algorithms are divided into the two classes of index-based and generic algorithms. We will give an outline on the evolution of both types.

Index-based algorithms offer very good performance under some circumstances. There is a large number of them, for instance *Index* by Tan et al. using B^+-trees ([14]), or Kossmann et al.'s *Nearest Neighbor* ([11]) or *Branch-and-Bound-Skyline* (Papadias et al., [12]) using R-trees.

But index-based algorithms cannot be used in all cases: when joins or other complex database operations are used in a query or the query is complex as seen in figure 1, usually all existing indexes fail. Even in cases they can be applied, many algorithms fall back on tuple-to-tuple comparisons at some point. So to find the best matching object, an algorithm based on tuple comparison has to be used.

The nested-loop based algorithms are the other approach. They do not need indexes but are based on tuple-to-tuple comparisons. The trivial version would be to keep all candidate tuples in memory and compare each tuple to all others. By introducing *Block-Nested-Loop* (*BNL*) in [1], Börzsönyi et al. offered a nested-loop using constant memory.

This algorithm since then has been the basis for other algorithms like Chomicki et al.'s *Sort-Filter-Skyline* (*SFS*, [3]) or Godfrey et al.'s *Linear Elimination Sort for Skyline* (*LESS*, [6]). These algorithms sort input sets to find killer tuples that dominate many of other tuples in a early evaluation phase. Another approach was BNL^{++} (Preisinger et al., [13]) which uses result set pruning mechanisms to filter out input tuples not belonging to the result set without having to compare them to other tuples.

A different kind of loop-based algorithm is *Bitmap* introduced by Tan et al. in [14]. Abandoning the "nested" in the loop, this algorithm compares each tuple to all others using (possibly very fast) binary operations.

3. THE "BETTER-THAN" GRAPH

Visualization of strict partial orders is often done using Hasse diagrams, graphs in which edges state domination ([4]).

DEFINITION 2. *"better-than" graph (BTG)*
The "better-than" graph *for a preference $P = (A, <_P)$ is the Hasse diagram of $<_P$ with the additional characteristics:*

a) Each equivalence class in dom(A) is represented by one node in the BTG.

b) The level value *of a node is the length of a longest path leading to it.*

We will now see the differences between BTGs for WOPs and for Pareto preferences. Then we will analyze the latter.

3.1 BTGs for WOPs

For WOPs like a base preference P_i, a BTG contains nodes for each value in the interval $[0, max(P_i)]$. As in a weak order, domination can be directly seen from the level value, this leads to BTGs as shown in figure 3.

As we see there, a BTG in this case is a total order of the values representing the different equivalence classes. The level value of a node in the BTG simply is the value the node shows ([13]). Best values have a level value of zero and therefore belong to the top node. The lowest node of the BTG is the one with the largest possible level value, $\max(P_i)$.

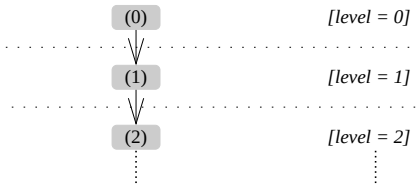

Figure 3: Sample BTG for a WOP

For the POS preference on the existance of a jacuzzi in figure 1, the BTG clearly consists only of the nodes for level 0 and 1.

3.2 BTGs for Pareto Preferences

The key point is that Pareto preferences do ***not*** create weak orders, even if they are applied to WOPs as input preferences ([2]). This leads to a more complex BTG.

EXAMPLE 1. *In figure 4, we see a BTG for a Pareto preference consisting of two WOPs with maximum level values of 2 and 3.*

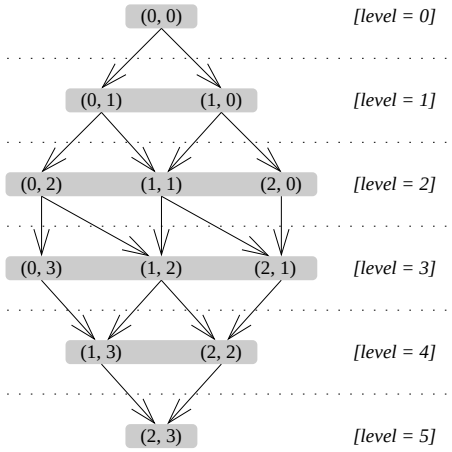

Figure 4: Sample BTG for a Pareto preference

The BTG for a Pareto preference $P = \otimes(P_1, P_2, \ldots, P_m)$ shows some typical properties we will see now. More details can be found in [13], where BTGs for Pareto preferences containing only WOPs have been analyzed for the first time. All aspects directly depend on the maximum level values of the P_i contained in P.

► ***Level of a node***
The level of a node $a = (a_1, a_2, \ldots, a_m)$ is computed as:

$$level_P(a) = \sum_{i=1}^{m} a_i$$

► ***Number of nodes***

The number of nodes is given by the number of different equivalence classes constructible by the different possible level values: $|BTG| = \prod_{i=1}^{m}(\max(P_i)+1)$

► ***Height and widths of different levels***

The height of the BTG can be derived from the maximum level value of P:

$$height(BTG) = 1 + \max(P) = 1 + \sum_{i=1}^{m} \max(P_i)$$

The width of the BTG depends on the level. The top level contains only one node: $(0,\ldots,0)$, level 1 contains m nodes. The levels grow wider until a maximum width is reached before or at level $1/2 \cdot \max(P)$. There may be more than one level with the greates width. The graph is symmetric and so starts to get "thinner" while increasing the overall level value and finally reaches one node in the highest level, $(\max(P_1),\ldots,\max(P_m))$. This leads to the characteristical *hexagon shape* of BTGs we see in figure 5.

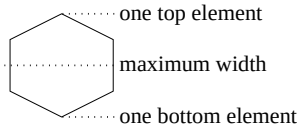

Figure 5: Hexagon shape of a BTG

So far we have recapitulated known knowledge from [13]. Now we will present our new results.

► ***Edge weights***

Coming from the top node, most nodes can be reached not only by one path but by a number of different paths. Nevertheless there is a method to select unique paths for each node. This path will be represented by an integer value that also can stand for the unique ID of the node. To supply each node of the BTG with a unique ID, we need to introduce the *weight* of edges in the BTG.

The nodes u and v are connected by a direct edge, iff they differ in exactly one position, expressing domination by some preference P_i. Thus the weight of an edge can be characterized by the weight of this P_i, which is given by:

$$weight(P_i) \quad := \quad \prod_{j=i+1}^{m} (\max(P_j)+1)$$

EXAMPLE 2. *In example 1 we have seen a Pareto preference* $P = \otimes(P_1, P_2)$ *with* $\max(P_1) = 2$, $\max(P_2) = 3$. *Therefore, in figure 4 we get the weights:*

$$weight(P_1) = 4, weight(P_2) = 1$$

E. g., one edge with weight 4 goes from $(0,0)$ *to* $(1,0)$, *another one from* $(1,1)$ *to* $(2,1)$.

► ***Unique node ID***

We will use these edge weights now to define an ID for each node $a = (a_1, a_2, \ldots, a_m)$ in the BTG.

THEOREM 1. *Uniqueness of a node ID*

Let $ID : (\mathbb{N}_0)^m \to \{0,1,2,\ldots,|BTG|-1\}$ *be a mapping such that*

$$ID(a) \quad := \quad \sum_{i=1}^{m} (weight(P_i) * a_i)$$

Then the following points hold:

a) ID is unique for every node in the BTG.

b) Every value in the set $\{0,1,2,\ldots,|BTG|-1\}$ *is a valid ID for one node in the BTG.*

c) ID is bijective. $ID^{-1}(n) = (a_1, a_2, \ldots, a_m)$ *maps a unique integer ID* n *to the corresponding level value combination* $(a_1, a_2, \ldots, a_m)$, *where the* a_i *are found the following way (DIV stands for* integer division *and MOD for the* remainder *of the division):*

$$\begin{aligned} n \ DIV \ weight(P_1) &= a_1 \ MOD \ r_1 \\ r_1 \ DIV \ weight(P_2) &= a_2 \ MOD \ r_2 \\ &\ldots \\ r_{m-1} \ DIV \ 1 &= a_m \end{aligned}$$

PROOF.

a) Let's start with two nodes $a = (a_1, a_2, \ldots, a_m)$ and $b = (b_1, b_2, \ldots, b_m)$. If their IDs are equal, we see:

$$\begin{aligned} ID(a) - ID(b) &= 0 \text{ iff} \\ \sum_{i=1}^{m}(weight(P_i) * a_i) - \sum_{i=1}^{m}(weight(P_i) * b_i) &= 0 \text{ iff} \\ \sum_{i=1}^{m}(weight(P_i) * (a_i - b_i)) &= 0 \end{aligned}$$

As all $weight(P_i)$ are positive values, $ID(a)$ and $ID(b)$ can only be equal if $\forall i \in \{1,2,\ldots,m\} : a_i = b_i$, and so a and b are equal.

b) There are exactly $|BTG|$ nodes in a BTG. From the definition of the ID, there are two extremal values:

- the top node $t = (0,\ldots,0)$ has an ID of 0
- the bottom node $b = (\max(P_1), \max(P_2), \ldots, \max(P_m))$ has the ID

$$ID(b) := \sum_{i=1}^{m}(weight(P_i) * \max(P_i))$$

Applying the construction formula for the edge weights leads to the following result:

$$ID(b) = |BTG| - 1$$

So IDs of all $|BTG|$ nodes belong to the set $\{0,1,2,\ldots,|BTG|-1\}$. Each node has a unique ID and the set has $|BTG|$ members.

c) We can find the inverse function ID^{-1} by deconstructing the ID function: $ID(a) = weight(P_1) * a_1 + r_1$, with r_1 being some remainder. Then, we can interpret r_1 as $weight(P_2) * a_2 + r_2$ and so on. □

EXAMPLE 3. *Using the Pareto preference from example 1 and its edge weights computed in example 2, we find the following unique IDs:*

$$\begin{aligned} ID((2,0)) &= 4*2+1*0 = 8 \\ ID((1,2)) &= 4*1+1*2 = 6 \end{aligned}$$

We now come back to the *unique path* to every node we heard of when introducing edge weights. It is found by a simple rule: in each node you visit, follow the edge with the highest weight leading to the target node. This path clearly is unambiguous. In each node, you have only one edge you can follow. Of course the path exists for every node.

EXAMPLE 4. *In figure 4, we will find the unique path for node* $(1,2)$. *Starting at* $(0,0)$, *the edge with the highest weight leading to the target node takes us to* $(1,0)$. *From this node on, we only follow edges with weight 1. By passing* $(1,1)$ *we finally reach* $(1,2)$. *Adding up weights of the edges we have walked, we get* $4+1+1=6$, *the ID of node* $(1,2)$.

4. THE HEXAGON ALGORITHM

Based on the results made until now, we have developed the *Hexagon* algorithm [2]. The algorithm consists of three parts, a construction phase for initialization, an adding phase when tuples are read, and a removal phase to remove dominated tuples. We will have a close look at each of them.

4.1 Construction Phase

The *construction phase* shown in procedure 1 is needed to initialize the data structures, in particular the BTG. It is constructed according to the WOPs that will be evaluated on the input relations.

We assume that we already know the maximum level values of the WOPs contained in the Pareto preference to be evaluated. With the maximum level values, we compute the edge weights and the BTG size.

In the next step, we initialize an array with the size of the BTG. This is the representation of the BTG in memory. Each position stands for one node and holds a list of tuples belonging to it. We then start to loop over the node IDs. For each ID, we determine to which level the corresponding node belongs. This is done by applying the function ID^{-1}.

Then we look if another node n belonging to this level has been found before. If so, the current node is n's *next* node and n is the current node's *previous* node. If no node in the same level has been found until now, the current node is the first node in this level. With all IDs processed, for each level the *next/prev*-relation has been constructed. The levels are then connected by using the first node in level k as the *next* node of the last node of level $k-1$. This is done for all levels. The *prev*-node of the first nodes of the levels are set accordingly. So we have constructed a *breadth-first walk* through the BTG. In each level, nodes are visited in ascending order of their ID. As a last step, the *next* node of the bottom node is set to -1, an invalid ID.

4.2 Adding Phase

In the *adding phase*, the input relation is read tuple-by-tuple. As procedure 2 shows, for each tuple all level values and the unique ID they are represented by are computed.

The new tuple is then added to the BTG node with the same ID. The ID is used as index in the BTG array.

4.3 Removal Phase

This phase, as shown in procedure 3, starts with a possible special case if perfect matches for the preference (i.e. tuples with ID 0) have been found. Then only these are the result. All other tuples can be discarded.

The ordinary case is to follow the *breadth-first* walk constructed as *next*-relation in the init phase of the algorithm, starting at ID 0.

Empty nodes just are removed from the *next/prev*-relation. For non-empty nodes, all dominated nodes have to be removed. The removal of the nodes dominated by a particular other node uses a *depth-first* tree-walk algorithm. Thereby, each of the dominated nodes of a node will only be visited only once. Let's assume we have found a non-empty node. Starting from this node, we begin a depth-first tree-walk down to the bottom node (see procedure 4). We will visit each dominated node exactly once. We follow each edge to nodes directly dominated. Then, we have a simple constraint: only follow edges with weights lower than or equal to the one's used to reach the current node.

So the paths from the non-empty node to the dominated nodes are found just like the unique paths to nodes as described in the end of section 3.2. Like then each node is reached and each node is reached exactly once.

We remove each node we reach from the *next/prev*-relation, so that we will not reach the node while following it and we can diagnose a node has been already visited (and therefore itself and its dominated nodes have been removed from *next/prev*). If the node is reached once more (possible only due

[2]It's name derives from the characteristical BTG shape as sketched in figure 5.

PROCEDURE 1 *Hexagon* – construction phase

Input: Pareto preference P
Global variables: arrays btg, lvl, $next$, $prev$, $weight$

1: **procedure** INIT(P)
2: // calculate edge weights
3: $weight[|P|] \leftarrow 1$
4: **for** $i \leftarrow |P|-1,\ldots,2,1$ **do**
5: $weight[i] \leftarrow weight[i+1] * (\max(P_i)+1)$
6: **end for**
7: // calculate the BTG size
8: $size \leftarrow \prod_{i=1}^{m}(\max(P_i)+1)$
9: $btg \leftarrow$ array with $size$ elements
10: $next, prev, lvl \leftarrow$ `int`$[size]$
11: // arrays needed for init computation:
12: // stores highest ID found for each level by now
13: $work \leftarrow$ `int`$[\max(P)+1]$
14: // stores first ID found for each level by now
15: $first \leftarrow$ `int`$[\max(P)+1]$
16: // initialize the arrays
17: $next[0] \leftarrow 1$
18: **for** $id \leftarrow 1,2,\ldots,size-1$ **do**
19: // compute level of the node
20: $curLvl \leftarrow 0$
21: **for** $i \leftarrow 0,1,2,\ldots,|P|$ **do**
22: $curLvl \leftarrow curLvl + \lfloor id/weight[i] \rfloor$
23: $id \leftarrow id - (\left\lfloor \frac{id}{weight[i]} \right\rfloor * weight[i])$
24: **end for**
25: **if** $first[curLvl] = 0$ **then**
26: $first[curLvl] = id$
27: **end if**
28: // set *next* node in the level
29: $next[work[curLvl]] \leftarrow i$
30: $work[curLvl] \leftarrow i$
31: $lvl[id] \leftarrow curLvl$
32: **end for**
33: // init *next* relation for last nodes in the levels
34: **for** $curLvl \leftarrow 1,2,\ldots,|P|-1$ **do**
35: $next[work[curLvl]] \leftarrow first[curLvl+1]$
36: **end for**
37: // set *next* node of bottom node
38: $next[size-1] \leftarrow -1$
39: **end procedure**

PROCEDURE 2 *Hexagon* – adding phase

Input: *tuple* the tuple to be added
Global variables: *btg*, *weight*

```
1: procedure ADD(tuple)
2:     // compute the node ID for the tuple
3:     id ← 0
4:     for i ← 1, 2, ..., m do
5:         id ← id + level_{P_i}(tuple) * weight(P_i)
6:     end for
7:     // add tuple to its node
8:     btg[id].append(tuple)
9: end procedure
```

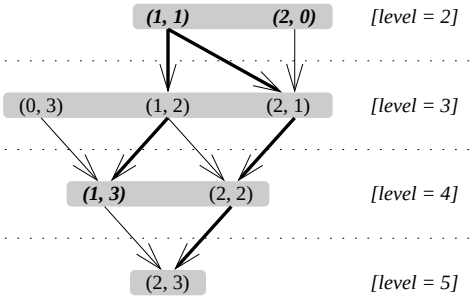

Figure 6: BTG in removal phase

to domination), the algorithm will stop at this point and continue the depth-first tree-walk at another node.

EXAMPLE 5. *Let's assume we are evaluating the Pareto preference from example 1. After construction and adding phase, there are tuples belonging to the nodes* $(1,1)$*,* $(2,0)$ *and* $(1,3)$ *(marked in bold italics in figure 6).*

We step into the algorithm the very moment when node $(1,1)$ *is reached in the* next*-relation. For clarity, we have not visualized the breadth-first* next/prev *relation but only show the nodes still belonging to it. (Note: all dominated nodes are removed from* next/prev *as soon as they are reached, tuples belonging to them are discarded.) The edges we use in our depth-first walk are marked bold in figure 6.*

At first, we go down the path from $(1,1)$ *to* $(1,3)$ *as it starts with the lowest edge going out from* $(1,1)$*. Doing this, we do not reach* $(2,2)$ *and* $(2,3)$ *as they would be reached by edges with higher weight than use in our path.*

Then we walk down over $(2,1)$*. We see that all followers of this node have not been visited. Each node dominated by* $(1,1)$ *is visited exactly once. After removing them, no nodes dominated by* $(1,1)$ *are in* next/prev *anymore.*

We can move on to $(2,0)$*. As the only follower already has been removed, nothing has to be done anymore. The last step is to remove the empty node* $(0,3)$ *from* next/prev*.*

At this point, all tuples have been read and all nodes that are empty or dominated by non-empty nodes in the BTG have been removed. The results now can be read by simply following the *next*-relation starting at the first existing node. Each node holds the set of tuples belonging to it.

Disc based implementation issues: So far we have described a main memory implementation of *Hexagon*. But as we only work on equivalence classes, keeping any input tuples in memory is not neccessary. For each equivalence class we only need to know if it is empty or not. The tuples themselves can be temporarily stored on disc after the computation of their IDs.

PROCEDURE 3 *Hexagon* – removal phase

Global variables: *btg*, *next*, *prev*, *lvl*

```
1: procedure FINDBMO
2:     // special case: top node is not empty
3:     if btg[0] ≠ ∅ then
4:         next[0] ← −1 return btg[0]
5:     end if
6:     cur ← 1
7:     while cur ≠ −1 do
8:         if btg[cur] ≠ ∅ then
9:             // non-empty node belonging to BMO set
10:            last ← cur
11:            // remove all nodes dominated by current
12:            for i ← 1, 2, ..., |P| do
13:                // check if there is an edge for P_i
14:                if lvl[cur + weight[i]] = lvl[cur] + 1 then
15:                    REMOVE(cur + weight[i], i)
16:                end if
17:            end for
18:        else
19:            // node is empty: remove from next/prev
20:            next[last] ← cur
21:            prev[next[cur]] ← last
22:        end if
23:    end while
24: end procedure
```

5. EFFICIENCY OF HEXAGON

We will first have a look at the theoretical complexity of the algorithm.

5.1 Worst-Case Complexity

To find out the theoretical complexity of the algorithm, we will have a look at the different parts of the algorithm.

► ***Construction Phase***

In the initialization phase of the algorithm, an array for the BTG nodes is constructed. The size of this array is determined by the maximum level values of the given user preferences and can be easily computed (see section 3.2). The $\max(P_i)$ have constant size and as we have stated are usually relative small leading to a small BTG. For the creation of the *next/prev*-relation, this array is traversed exactly one time. The level of each node has to be computed. This needs m steps. Apparently, the costs in this phase have the tight bound of $\Theta(m * |BTG|) = m * \Theta(\prod_{i=1}^{m}(\max(P_i)+1))$.

► ***Adding Phase***

When the input tuples are added, we have to do some computations for each of them. As seen in 3.2, calculating the unique ID of one tuple is done in $\Theta(m)$, where m is the number of WOPs contained in the Pareto preference query. For n input tuples, this clearly leads to a complexity linear in n: $\Theta(m * n) = m * \Theta(n)$.

► ***Removal Phase***

Removal of nodes from the BTG is again depending on the size of the BTG. Following the *next*-relation is done in $\mathcal{O}(|BTG|)$. For each node reached it is possible that it contains tuples and so that the nodes dominated have to be removed from the BTG. When a node is reached that

PROCEDURE 4 tree walk for node removal

Input: *id* the node to be removed
index edge weight index the node was reached with
Global variables: *btg*, *next*, *prev*, *lvl*

```
1: procedure REMOVE(id, index)
2:     // check if the node has already been removed
3:     if next[prev[id]]! = id then return
4:     else
5:         // remove the node from next/prev relation
6:         next[prev[id]] ← next[id]
7:         prev[next[id]] ← prev[id]
8:         // remove tuples in node
9:         btg[id] ← ∅
10:    end if
11:    // remove followers
12:    for i ← 1, 2, ..., index do
13:        // follow the edge for preference i
14:        if lvl[id + i] = lvl[id] + 1 then
15:            REMOVE(id + i, i)
16:        end if
17:    end for
18: end procedure
```

already has been removed, the algorithm will not follow the edges down from this node. This can be determined by the *next/pref*-relation: if the prev node's next isn't the same as the current node, the current node already has been removed. As a (very high) upper bound we could say that each node of the BTG is visited m times: once for every directly dominating node. Of course, this only can happen for a very small number of nodes: the maximum number of nodes belonging to the result set is given by the maximum width of the BTG. Even if all nodes of the BTG would be visited this often, it would be done in $\mathcal{O}(m * |BTG|) = m * \mathcal{O}(\prod_{i=1}^{m}(\max(P_i) + 1))$.

THEOREM 2. **Hexagon** ***is linear.***
Given a Pareto preference P ***with a fixed number*** m ***of WOPs on an input relation*** R ***of size*** n ***where the BTG size is at most linear in*** n***, then*** **Hexagon** ***evaluates Pareto preference queries in linear time*** $\Theta(n)$***.***

PROOF. We have the following complexities to add for the overall complexity of *Hexagon*:

- construction phase: $m * \Theta(\prod_{i=1}^{m}(\max(P_i) + 1))$
- adding phase: $m * \Theta(n)$
- removal phase: $m * \mathcal{O}(\prod_{i=1}^{m}(\max(P_i) + 1))$

Under the assumption of a BTG size in order of n, we find a *worst-case complexity of* $\Theta(n)$. □

The assumption of an at most linear-sized BTG is very frequently met in practice, in particular for B2C and B2B e-commerce. Categorical preferences never pose any problems, but numerical ones like e.g. AROUND_d potentially might do so. There, in the worst of all cases, each tuple of R could be mapped onto a different equivalence class, yielding a BTG size of $(n+1)^m$. For such extreme scenarios Hexagon is clearly useless in its present form. But by making semantically justified reasonable choices for the d-parameters, cutting down the problem to an at most linear-sized BTG is often possible in practice,.

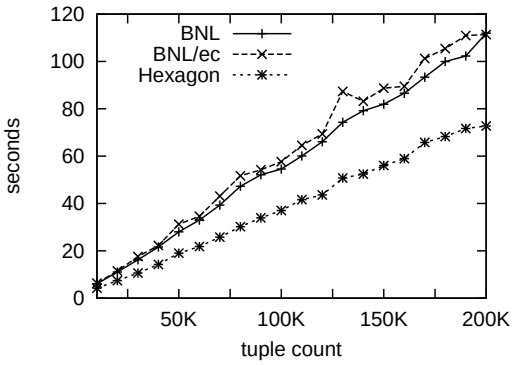

Figure 7: Avg. case performance (correlated data)

5.2 Performance Benchmarks

We will now have a look at the performance of the algorithm compared to standard BNL and a BNL derivate using equivalence classes we call BNL_{ec}[3]. Preliminary tests have been done for different types of input data distributions, different input sizes, and different Pareto preferences.

We have not included BNL style algorithms that use presorting techniques like SFS or LESS. Although the smart order of the input relation improves the performance of the BNL part, SFS suffers from the sorting costs of $\mathcal{O}(n * log\ n)$. LESS uses a combined sorting and discarding in linear time, but this is bound to specific features of the input tuples, in particular un-correlated and uniformly distributed data with mostly distinct values ([6]). Nevertheless, both as well show quadratic worst case performance.

All benchmarks have been executed on a Pentium 4 at 3.4GHz and 2GB RAM. The different test algorithms have all been implemented and executed under Java SE 6. To find our test preferences, we have constructed 10 WOPs with maximum level values between 1 and 10. We then have combined up to 7 of them to form 94 different Pareto preferences with BTGs with 10000 up to 20000 nodes. In our tests, this means the BTGs have sizes between $0.05 * n$ and $10 * n$ with n being the number of input tuples. We also have run the tests with the 561 Pareto preferences with less than 10000 nodes. As expected, the performance advantage of *Hexagon* is even greater due to the small amount of constant costs resulting from small BTGs.

► ***Average Case for BNL***

As known in literature ([6]), BNL based algorithms show linear average case running times. Therefore, we will examine different types of random distributed input sets, i. e. correlated and independently distributed data.

The number of tuples belonging to the BMO set of course depends on the distribution: correlated distributions will have smallest BMO sets, independent will have the bigger ones ([1]). As figures 7 and 8 show, the execution times of the BNL based algorithms correspond with the BMO set sizes. Correlated distributions show very good performance for both BNL style algorithms and *Hexagon*, independent distributions lead to worse performance as we see in figure 8. In all cases, the BNL based algorithms show far worse performance than *Hexagon* for large input sets.

[3] As far as we know, this algorithm is not known in literature. It is feasible only in our preference model due to equivalence classes formed by d-parameters and SV-relations.

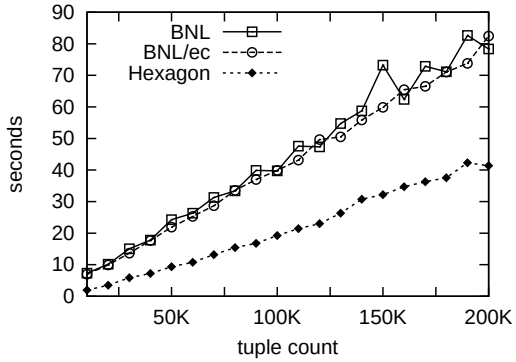

Figure 8: Avg. case performance (indep. data)

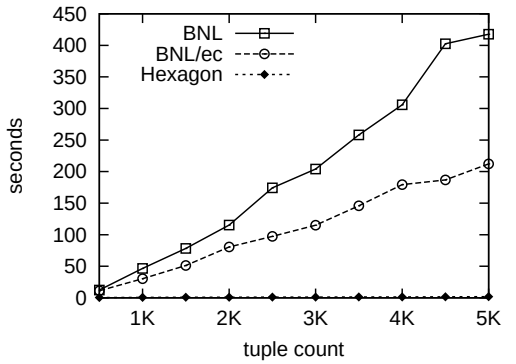

Figure 9: BNL$_{ec}$ worst case performance

► ***Worst Case for BNL***

The worst case for BNL$_{ec}$ we have examined is a sorting of the input relation in a way that worst tuples are first: no tuple will be dominated by a tuple coming before it in the input relation.

Figure 9 shows the superiority of *Hexagon* over the BNL algorithms. The running times of *Hexagon* are hardly to see: it evaluates 5,000 tuples in 1.8 seconds, compared to more than 200 (BNL$_{ec}$) or 400 (BNL) seconds. We did not continue these tests for more tuples as BNL already exceeds feasible running times by far. For *Hexagon*, required time seems to be constant. This is due to the initialization costs dominating input-based costs for such small inputs. As the diagram indicates, this difference will increase for bigger input relations.

6. CONCLUSION

We have presented the *HEXAGON* algorithm for evaluating Pareto preference queries over weak order preferences. *Hexagon* does not require the existance and costly maintenance of any special index structures, instead it gains its speed from the dynamic construction and careful analysis of the underlying "better-than" graph (BTG). Compared to competing algorithms, which usually work on a block-nested-loop paradigm, *Hexagon* does not suffer from a non-linear worst case complexity.

In fact, under absolutely reasonable practical assumptions, *Hexagon* is linear in time for any data distribution. Moreover, our preliminary performance benchmarks indicate that *Hexagon* is faster also for the average case. In summary, we consider *Hexagon* as an "algorithm for all practical seasons" for efficiently evaluating Pareto preference queries. As next steps we will do more performance experiments and also integrate Hexagon into our Preference SQL query optimizer. Finally, for extreme use cases generating overlinear-sized BTGs, we will investigate the behavior of our BNL^{++} algorithm, which likewise works on the principles of a detailled analysis of the "better-than" graph.

7. REFERENCES

[1] S. Börzsönyi, D. Kossmann, and K. Stocker. The Skyline Operator. In *ICDE*, pages 421–430. IEEE Computer Society, 2001.

[2] J. Chomicki. Preference formulas in relational queries. *ACM Trans. Database Syst.*, 28(4):427–466, 2003.

[3] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with Presorting. In U. Dayal, K. Ramamritham, and T. M. Vijayaraman, editors, *ICDE*, pages 717–816. IEEE Computer Society, 2003.

[4] B. A. Davey and H. A. Priestley. *Introduction to lattices and order*. Cambridge University Press, Cambridge, UK, 2nd edition, 2002.

[5] P. Fishburn. Intransitive indifference in preference theory: A survey. *Operations Research*, 18(2):207–228, 1970.

[6] P. Godfrey, R. Shipley, and J. Gryz. Algorithms and analyses for maximal vector computation. *VLDB J.*, 16(1):5–28, 2007.

[7] B. Hafenrichter and W. Kießling. Optimization of Relational Preference Queries. In H. E. Williams and G. Dobbie, editors, *ADC*, volume 39 of *CRPIT*, pages 175–184. Australian Computer Society, 2005.

[8] W. Kießling. Foundations of Preferences in Database Systems. In *VLDB*, pages 311–322, 2002.

[9] W. Kießling. Preference Queries with SV-Semantics. In J. R. Haritsa and T. M. Vijayaraman, editors, *COMAD*, pages 15–26. Computer Society of India, 2005.

[10] W. Kießling and G. Köstler. Preference SQL - Design, Implementation, Experiences. In *VLDB*, pages 990–1001, 2002.

[11] D. Kossmann, F. Ramsak, and S. Rost. Shooting Stars in the Sky: An Online Algorithm for Skyline Queries. In *VLDB*, pages 275–286, 2002.

[12] D. Papadias, Y. Tao, G. Fu, and B. Seeger. An Optimal and Progressive Algorithm for Skyline Queries. In A. Y. Halevy, Z. G. Ives, and A. Doan, editors, *SIGMOD Conference*, pages 467–478. ACM, 2003.

[13] T. Preisinger, W. Kießling, and M. Endres. The BNL^{++} Algorithm for Evaluating Pareto Preference Queries. In *Proceedings of the ECAI 2006 Multidisciplinary Workshop on Advances in Preference Handling*, pages 114–121, 2006.

[14] K.-L. Tan, P.-K. Eng, and B. C. Ooi. Efficient Progressive Skyline Computation. In P. M. G. Apers, P. Atzeni, S. Ceri, S. Paraboschi, K. Ramamohanarao, and R. T. Snodgrass, editors, *VLDB*, pages 301–310. Morgan Kaufmann, 2001.

[15] P. Viappiani, B. Faltings, and P. Pu. Preference-based Search using Example-Critiquing with Suggestions. *Journal of Artificial Intelligence Research (JAIR)*, 27:465–503, 2006.