Graph-based Algorithms for Pareto Preference Query Evaluation

Dissertation zur Erlangung des Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat)

der Fakultät für Angewandte Informatik
der Universität Augsburg

Timotheus Preisinger
University of Augsburg

Gutachter:

Erstgutachter:	Prof. Dr. Werner Kießling
Zweitgutachter:	Prof. Dr. Bernhard Möller
Mündliche Prüfung:	9. Juli 2009

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

Herstellung und Verlag
Books on Demand GmbH, Norderstedt

ISBN: 978-3-8391-2011-8

To my family and my friends.

I would like to thank everybody who supported me during the last years – may it have been by expertise or boosting morale or mood.

My friends who spent countless hours with me listening to bands in shabby clubs and at muddy music festivals (as well as shiny clubs and sunny music festivals).

All the unknown people entertaining me or letting me entertain them.

You all did what you could to distract me from doing research.
I would not have come this far without you!

Thank you all!

Contents

Abstract

Searching a database is one of the most common procedures in everyday life. Usually, the results of such a search match the query parameters perfectly. But if no perfect match is found, the user usually has to find out by himself how to change search parameters in order to get results.

To overcome this problem, Kießling has introduced a model of preferences in databases. This model is based on simple strict partial orders as given in expressions like "red is better than blue". For every query, the best-matching objects are returned, whether these are perfect matches or not. A best match is a tuple that matches the preference not worse than any other tuple – or as we say – that is not dominated by any other tuple. The specific problem we address is finding best matches for Pareto preferences, the combination of preferences with all of them being equally important. This problem is closely related to skyline queries.

Based on the better-than graph, a visualization of the strict partial orders constructed by Pareto preferences, we have found a novel type of optimization called *pruning* that can be applied to all existing generic algorithms. While common generic algorithms rely on tuple-to-tuple comparisons to identify dominated tuples, our optimization technique uses the structure of the better-than graph to identify elements in the order that are definitively dominated by some given tuple. This enables us to omit many comparisons.

By further analysis of the better-than graph, we were able to find a new kind of algorithm. This generic algorithm, *Hexagon*, is capable of finding the best matches in some previously unknown set of tuples in linear time with respect to the size of the better-than graph. Apart from the standard algorithm, we present a number of optimizations for it regarding its memory requirements. But *Hexagon* is not limited to standard preference queries. We also address *top-k* queries with a variant of *Hexagon*. These queries return the best k tuples of an input relation with respect to some rating function.

The performance benchmarks we have made show the superiority of algorithms using *pruning* and especially of *Hexagon*, although the latter cannot be used in all cases due to memory requirements. Moreover, *Hexagon* can be combined with existing algorithms that have been optimized by *pruning* to enable the cost-based algorithm selection for Pareto preference evaluation.

Acknowledgement

This work would not have been possible without the support of a number of people. First of all, I would like to thank Prof. Dr. Werner Kießling, my doctaral thesis supervisor, who provided me with the possibility of research in this fascinating field of computer science. Without the discussions we had and his helpful comments this work would hardly have been possible.

I am also very grateful for the support (especially on some problems concerning the theoretical parts of my work) received from Prof. Dr. Bernhard Möller.

Moreover, I would like to thank my colleagues Markus Endres, Alfons Huhn, Peter Höfner, Sven Döring, Stefan Fischer and Anna Schwartz. Apart from the beneficial talks we had, my work at the Department for Databases and Information Systems at the University of Augsburg, Germany, would not have been nearly as exciting and joyful without them. I'm also grateful to Wolf-Tilo Balke for sparking my interest in scientific work when I started my diploma thesis.

I would also like to thank my sister Priska for helping me proof-reading this book and pointing out some of my linguistic adventures.

Chapter 1

Introduction

In real life, it is absolutely common for people to express preferences in product searches or decision-making. Nevertheless, the subject of preferences has come into the focus of research only in recent years in computer science, especially in the contexts of artificial intelligence and of databases, with us dealing with the issue in the latter field. Still, most database search engines can only deal with hard constraints: a tuple either fulfills all given conditions or it does not belong to the search result. While this concept works fine in many use-cases, it definitively has some weak points. Starting with a small example, we will see how to solve this problem and what kind of new problems this entails. This will lead directly to the main objectives of this book.

1.1 Motivation

Tom is looking for a used car. Apart from some hard constraints he has, he wants both the mileage and the price of the car to be as low as possible. These two wishes are clearly contrarian. Considering two cars that only differ in price and mileage, the one with the lower mileage will probably be the one with the higher price.

When both the price and the mileage preference are interpreted as hard constraints, the search interface will not be able to present any result to Tom, although there may be lots of cars in the database. This phenomenon is called the *empty result effect*.

Kießling has described preferences as strict partial orders in [37] and outlined how to integrate such preferences into database systems. By using strict partial orders, he was able to overcome the problem of hard constraints. A value can be *better* than another value, and the *best matches* are selected as search results.

cars	id	color	price	mileage
	1	black	5,000	100,000
	2	blue	10,000	80,000
	3	black	12,000	150,000
	4	silver	20,000	10,000
	5	black	15,000	20,000
	6	silver	16,000	25,000
	7	silver	10,000	30,000

Table 1.1: A relation

For our car example, this means that one car is better than another if and only if it is cheaper and has a lower mileage. Table 1.1 holds some sample cars. When we evaluate Tom's preference on this relation, we find out that car 2 is worse than car 7, car 3 is worse than cars 1, 2, and 7, and car 6 is worse than car 5. The rest of the input is not worse than any other car. So it would be best to present cars 1, 4, 5, and 7 to Tom. Each of them is not more expensive and has not more mileage than any other car. They all match Tom's preference *best*.

When two or more preferences are combined with all of them being equally important, the result is called a *Pareto preference* ([37]). The concept of returning those tuples that are not worse than any other is called *best matches only* (*BMO*).

In a more formal way, we can say that, when evaluating a Pareto preference P on a database relation R, the best matches in R wrt P form the result set. These are tuples of R that are not dominated by other tuples. A tuple $r_1 \in R$ *dominates* a tuple $r_2 \in R$ if r_1 is better at least in one preference contained in the Pareto preference and at the same time r_1 is not worse than or incomparable to r_2 in all other preferences. While two values that are equally good with respect to a preference, but not actually equal, are classified as "incomparable" in common Pareto preferences, the SV-semantics introduced in [38] often render them as "substitutable".

Pareto preference queries are also known as *skyline* queries ([6]), although the latter form just a specialization of Pareto preferences. Only Kießling's very generic approach of strict partial orders has the capability of dealing with numerical domains as well as with non-numerical, categorical, domains. *Skyline* queries can only handle numerical domains, although there are often preferences on categorical domains such as colors or brands.

In the last few years, a number of different algorithms has been presented for efficient BMO (or *skyline*) computation. As we will see in detail when related work is discussed in chapter 2, all existing algorithms can generally be divided into two different types:

Generic algorithms: No information on the input data is needed. Usually, these algorithms use tuple-to-tuple comparisons to eliminate tuples that are dominated. As in the worst case each tuple has to be compared with every other tuple, the performance of such algorithms is rather poor, with runtimes quadratic with respect to the number of input tuples.

Index-based algorithms: The algorithm reads the input tuples according to their positions in one or more indexes. This way, only potentially matching tuples have to be read at all and fast execution times are possible. But to be able to use an index-based algorithm, a suitable index has to exist, which is difficult to guarantee for all potential queries.

The general goal of this book is to solve the problem of finding the *best matches* for a preference in some input relation by introducing completely new ways of dealing with Pareto preference queries. We want to overcome the problem of either slow or only scarcely useable algorithms.

1.2 Objectives of this Book

This book will describe different ways to improve the performance of Pareto preference evaluation. Before we come to this, existing algorithms and other related work will be discussed in chapter 2. Then Kießling's preference model of [37] and [38]) will be introduced in chapter 3. We will use this model throughout the whole paper. The *better-than* graph as a visualization for the strict partial orders induced by our preferences will be the subject of chapter 4, giving us the basis for the following chapters. The focus will of course lie on Pareto preferences. After these basics, we will focus on the main research issues of this book: algorithm optimization, design of new algorithms and integration of both concepts into existing database optimizers. We will give brief descriptions of these topics now.

1.2.1 Optimization

Based on the *better-than graph*, we will introduce a completely new kind of optimization strategy for algorithms for Pareto preference queries. Many existing algorithms, e.g. *BNL* or *LESS* ([6] and [31]), rely on tuple-to-tuple comparisons to identify the BMO set in some input relation. We will be able to reduce the number of comparisons that have to be made, and hence be able to cut down execution times by a notable factor. This technique is called *pruning* and its integration in different algorithms will be the subject of chapter 5.

1.2.2 Algorithms

The *better-than graph* will also form the basis for the new algorithm we will introduce in chapter 6, *Hexagon*. We will see that it can be applied on all kinds of input relations and does not need any indexes. While common algorithms with these characteristics have to rely on tuple-to-tuple comparisons, *Hexagon* only has to read every input tuple once. The input relation may be completely unknown, as is often the case for dynamically constructed relations such as join or set function results. The overwhelming performance of *Hexagon* in comparison to known generic algorithms will be shown in the benchmark results of chapter 7.

In chapter 6, we will also see that *Hexagon* can evaluate Pareto preferences containing preferences that define any type of strict partial order. For the first time ever, a generic algorithm can do this in linear time. Moreover, we will see that *Hexagon* is not only capable of handling common Pareto preference queries, but, with only slight modifications, can be used for related problems as well. *Hexagon* will be used to compute *multi-level BMO sets*, in which not only the best matches, but also the second best matches, third best matches and so on are relevant. Furthermore, *Hexagon* will be used to answer $top - k$ queries, in which each tuple is rated according to some function and the k tuples with best function values are returned.

1.2.3 Cost-based Algorithm Selection

The novel optimization technique and the new type of algorithms will be benchmarked of course. We will carry out a number of performance tests, varying preferences and size and data distribution of input relations. The performance benchmarks will be discussed in chapter 7. In the same chapter, we will use the results to define rules for *cost-based algorithm selection*. These rules will enable us to find the fastest algorithm for Pareto preference evaluation depending on factors like available memory or distribution of input relations. Query optimization in database systems is usually done in two phases. The first phase is the algebraic optimization of a query, which has been discussed in [15, 34]. In this book, for the first time, the second phase, the cost-based optimization can be addressed by our rules of algorithm selection.

Chapter 2

Related Work

Pareto preference query processing belongs to the field of multi-dimensional optimization. While we understand preferences to be capable of dealing with any type of strict partial order (see [22]), most effort has been put into handling numeric values combined in a Pareto way. This subject was examined wrt to special algorithm design and computational complexity for the first time for finding maximal elements in a set of vectors in [44]. In the database research community, *skylining* is the most common term for this type of problem. It came into play with Börszonyi et al.'s `skyline` operator in [6]. The size of *skylines* has been investigated in [30].

There have been a lot of algorithmic approaches to find the best matching tuples for a query, based on some database relation. We will give an overview over the most important of them and over recent research results. Generally, most algorithms were designed for *skyline* queries dealing with totally ordered numerical domains but work for weak orders (see [22]) as well. First, we will have a look at algorithms that use indexes, then we will see more generic approaches. Finally, we will give an overview on research on *top-k* and *skyline* queries and derivates of the latter.

2.1 Index-based Algorithms

This type of algorithms was introduced together with the *skyline* operator. Already on first appearance in [6], applicability was the main problem. After having a look at some index based algorithms, we will get back to this issue.

In [6], B-tree indexes on the simple attributes of the *skyline* query were used. The authors of the paper did not expect good performance due to some deficits in the design of the algorithm and therefore they did not further examine it. Börszonyi et al. also outlined the use of R-trees but did not pursue this as well. With *Index* ([55]), an algorithm using B$^+$-trees was introduced. Just as for the B-tree-algorithm of [6], there came a point after which tuple-to-tuple comparisons had to be used. As was the intention for so-called *progressive* (or *online*) algorithms, fast delivery of the first result tuple could be achieved, while it could happen that, for the complete result, all indexes had to be scanned completely. Together with the large number of tuple-to-tuple comparisons that occured in such cases, the algorithm did not run significantly faster than generic algorithms.

For some time then, R-tree based indexes were the focus of research on *skyline* algorithms. [42] introduced the R-tree based *Nearest-Neighbor* algorithm. It was designed as an *online algorithm*. One of its main features was that results were returned with respect to their quality in one or more of the contained dimensions: the better an undominated tuple would fulfill the query preference, the earlier it would be delivered to the user as a result. Using R-trees as well, *Branch-and-Bound-Skyline* (*BBS*, [49, 50]) proved to be even faster, outperforming generic algorithms by far. In [45], a new type of index was presented: the ZBTree. This data structure was specifically designed for *skyline* queries and made the *ZSearch* algorithms using it faster than *BBS* and less memory consuming.

While index-based algorithms promise (and some of them also show) very good performance, they suffer from a major problem: a large number of indexes on different attributes or even the combination of any attributes is not available in many cases. This leads to poor applicability for index-based algorithms if they are not used in well-defined static use cases. Another problem arises from the index maintenance costs. If the base data set of the index is constantly changing, index maintenance may generate more computational costs than the queries themselves. Additionally, index availability and maintenance costs always conflict in their objectives. The more indexes on different attribute combinations there are, the more often the algorithm based on the index type may be used. But at the same time, more indexes naturally mean more maintenance costs as well. Also, a database will not hold indexes for the results of table joins or set operations like `UNION` in most cases. Thus, if *skyline* computation on such dynamically constructed sets have to be made, index based algorithms will usually fail.

2.2 Generic Algorithms

Generic algorithms for *skylining* can work on any kind of data. All information they need is acquired directly when reading the data. Each input tuple has to be read and analyzed at least once, something a good index-based algorithm never has to do.

Hence, generic algorithms are never able to compete with progressive algorithms for the fastest delivery of the first element of the result set. We will distinguish between two types of generic algorithms: Those that are based on nested loops and those that are not. Both types will be introduced in the following sections.

2.2.1 Classical Nested Loops

Determining the *skyline* of some dataset by tuple-to-tuple comparisons in a loop is the most intuitive way and was already shown in [44]. For use in a database scenario, a general loop however was not useful due to memory restrictions.

So in [6], *Block-Nested-Loop* was introduced. During execution, a number of undominated tuples was kept in memory and compared to newly read tuples. A more detailed overview of *BNL* can be found in Section 5.2.1. *Sort-Filter-Skyline* was proposed in [16] and used presorting of the input relation wrt to a scoring function. Using a sophisticated scoring function, it was possible to keep only the tuples belonging to the result in the main memory buffer. Some optimizations and theoretical observations on it were published in [17]. *Linear Elimination Sort for Skyline* (*LESS*, [31]) combines sorting and removal during a dedicated first phase of the algorithm and so is capable of removing dominated objects even during sorting. *LESS* will be one of the topics of Section 5.2.2 and is described there in detail.

2.2.2 Other Approaches

Apart from the popular nested loop approach, there have also been some completely different ways of solving the problem. Although they have not been found to be as fast as other algorithms, we will outline them as well.

In both [40] and [6], the translation of a Pareto preference respectively a *skyline* query into standard SQL was introduced. In the former, good performance benchmark results were achieved for common e-commerce applications; in most cases three or fewer *skyline* dimensions were used. The latter paper found that query rewriting is scaling very badly wrt to the number of *skyline* dimensions and the size of the result set. Both [40] and [6] found the performance of query rewriting to be comparable to other algorithms for small *skyline* dimensions. But for five or more dimensions with huge result sets, performance completely collapses, leading to execution times longer than other algorithms in orders of magnitude.

The *Divide & Conquer* algorithm of [44] (enhanced in [6]) at first partitions the input into smaller sets. After *skyline*s have been computed for the smaller sets, these partial result sets are merged. In the merge phase, nodes from one set may dominate nodes. Of course, the dominated nodes have to be dismissed then. Using sophisticated strategies when partitioning, such cases of domination at merge time can be minimized (as described in [6]). Still, the algorithm was outperformed by *BNL* even then.

Another completely different algorithm exists with *Bitmap*, proposed in [55]. It was based on a mapping of domain values to binary numbers. Thus it was possible to rely on Boolean operations to determine *skyline* tuples. Compared to tuple-to-tuple comparisons, the mapping led to a much larger number of Boolean operations to determine *skyline* tuples. As such operations can be executed extremely fast, performance benefits were expected – and partially realized, too. But (mostly due to the necessary preprocessing phase), *Bitmap* was shown to be less efficient than the simultaneously presented *Index*. Hence, such an approach was never used again.

Most *skyline* algorithms basically address the problem of Pareto preferences over weak orders. To overcome this restriction, *BNL* was generalized in [9] to an algorithm called BNL^+, dealing with Pareto preferences over general strict partial orders. The elements of a strict partial order were mapped to two integer level values. Not all orders could be represented completely, so in addition to the complexity of *BNL*, in many cases the elements of the partial order had to be compared as well to determine domination. Still, performance gains could be realized over the general approach that only relies on element comparison. This kind of mapping was also used in [10], when it was integrated into *BBS*, forming BBS^+. As the mapping produces quite a number of false or missing dominance relations, the algorithms *SDC* and SDC^+ were introduced, using optimized handling strategies for mapping errors. In [53], elements of partial orders were once again represented by two integer level values. Although the mapping was more general than in [9], it was still not possible to cover all types of strict partial orders, as we will see in Section 4.5.2. The authors introduced the algorithm *TSS*, which was, as they would show in some performance benchmarks, multiple times faster than other algorithms dealing with *skyline*s on strict partial orders, among them BNL^+, too.

While usual algorithms only deal with centralized data storages such as a single database where all input is read from, in [19], *PaDSkyline* was presented, an algorithm for dealing with distributed data sources. Intelligent filtering rules are used to reduce network traffic by reducing the number of *skyline* candidates as early as possible.

With the presentation of *Lattice Skyline* in [47] and, simultaneously and independently, the first version of *Hexagon* in [51], another type of algorithm appeared. Both abstracted from tuple-to-tuple comparisons: Each tuple was mapped to a specific equivalence class and comparisons were made between those equivalence classes. Other generic algorithms were outperformed by up to magnitudes of orders, as for the first time, a generic algorithm had linear complexity. *Hexagon* is a major part of this book and we will introduce and analyze it in detail in chapters 6 and 7. Still, despite the similarity of *Lattice Skyline* and *Hexagon*, only the latter one is capable of dealing with any number of numerical domains.

2.3 Top-k and Skyline-related Queries

Another important concept in multi-dimensional optimization is *top-k* search. Basically, the input is sorted according to some scoring function and the k elements with the best score form the result. Special algorithms *top-k* queries have been proposed in [7], based on the proposed new SQL keyword `STOP AFTER`. A number of other algorithms for *top-k* query processing has been found, among them *Prefer* ([36]) and *Linear Programming Adaption of the Threshold Algorithm* ([21]), which both use materialization in views, and the index based algorithms *Onion* ([12]) and *Approximate Solution for Robust Index* ([59]). *Quick Combine* as introduced in [33] and *Threshold Algorithm* ([26]) enhanced Fagin's algorithm of [25] for *top-k* queries in multimedia databases.

In [60], correlations between tuples belonging to a *skyline* and a *top-k* query result are used when constructing the index structure *Dominant Graph*. Combining *top-k* and *skyline* query evaluation has been done in [56], where *SUBSKY* was proposed, an algorithm based on the transformation of multiple-dimensioned input-data into simple numerical values indexed by a B-tree. Vlachou et al. proposed *top-k* evaluation based on *skyline*s for peer-to-peer networks. A peer finds the k results by evaluating *skyline*s retrieved from other peers. For online analytical query processing (OLAP), *P-Cube* provides index structures and query algorithm for *skyline* and for *top-k* queries. The previously mentioned *BBS* (cf. Section 2.1) is adjusted to deal with *top-k* queries as well.

In [50], other related problems are discussed as well. Among those are *constrained skylines*, meaning *skylines* combined with hard constraints filtering out some input tuples. *Skyband* queries are used to find those tuples dominated by not more than a given number of other tuples, whereas *k-dominating* queries find the number of tuples that are dominated by some given tuple. All of these problems can be addressed by special versions of *BBS*.

Chapter 3

Background: the Preference Framework

The modeling and handling of user preferences forms a complex challenge for application designers seeking to integrate personalization in their software. There is the need for a sophisticated but intuitive, a simple but comprehensive way to express preferences. Kießling's preference model introduced in [37, 38] has these features on top of a well-founded theoretical base. Needless to say, different preference models have been introduced in the last ten years, but most of them lack some of the mentioned key features. For a discussion of those, see [3, 58, 14, 15, 27, 43, 40, 38].

Figure 3.1: It's a Preference World

"It's a Preference World" (cf. Figure 3.1) has been the mantra of the Chair for Databases and Information Systems for some years and perfectly shows the research group's affiliation to preferences. Several technologies based on Kießling's preference model have been developed at the institute, enabling the comfortable and productive use of preferences in e-commerce and other applications. We will describe the basics of Kießling's work in this section and consider its implementations as well.

3.1 Preference Modeling

"I like red better than blue" is a completely natural way of formulating preferences on the colors "red" and "blue". Every person, even every child understands such statements of preferences and is able to use similar phrases to express his or her own preferences.

It is quite straight-forward to associate such a "better-than" semantics with the mathematical concept of strict partial orders. Being intuitive and easily expressible in mathematics, "better-than" semantics form a perfect concept for preference modeling. This leads us to Kießling's general definition of preferences ([37]):

Definition 1. ***Preference***
Given a set of attribute names A, a preference P is a strict partial order $P := (A, <_P)$, where $<_P \ \subset dom(A) \times dom(A)$. □

"$x <_P y$" is interpreted as "I like y better than x". As a strict partial order, $<_P$ is irreflexive and transitive and thus asymmetric. If of two different objects none is better wrt a preference, we call them *indifferent* or *unranked*. This is formally described in Definition 2.

Definition 2. ***Indifference***
Consider a preference $P := (A, <_P)$ and two values $x, y \in dom(A)$ with $x \neq y$. If none of them is better than the other, they are indifferent*, $x \ \sim_P \ y$:*

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

Keep in mind that, in general, $\|_P$ is reflexive, symmetric, but not transitive ([38]). In many cases it may be useful to have indifference as a transitive relation. The *substitutable value semantics* has been introduced in [38] for this reason.

Definition 3. ***Substitutable Values***
Consider a preference $P = (A, <_P)$. $\cong_P$ is called P's substitutable values relation *(*SV-relation*) iff for all $x, y, z \in dom(A)$:*

a) $x \cong_P y \Rightarrow x \ \sim_P \ y$

b) $x \cong_P y \wedge \exists z : z <_P x \Rightarrow z <_P y$

c) $x \cong_P y \wedge \exists z : x <_P z \Rightarrow y <_P z$

d) $\cong_P$ *is reflexive, symmectric and transitive*

The relation $x \cong_P y \Leftrightarrow x = y$ *is called* trivial SV-semantics*.* □

Such a preference model obviously meets our demands for simplicity and flexibility. To enable intuitive use of preferences, different types of preference constructors for single attributes will be introduced in the next section. Afterwards, we will look at how to combine preferences intuitively to model complex user preferences.

3.1.1 Base Preferences

To express simple preferences like minimum or maximum for numerical values, there are diverse *base preference constructors*. In this section, we will focus on base preferences that are not only strict partial orders, but even weak orders. As we will see, the variety among those is big enough to model a huge number of preferences. (A strict partial order base preference will be introduced in Section 4.5, though.) Generally speaking, base preferences can be classified by the domains they are applied to. After discussing some basics, we will look at various constructors for preferences on different domains.

Definition 4. ***Weak Order Preferences***
A weak order preference *(*WOP*) is a preference with* $<_P$ *being a* weak order*, i.e. a strict partial order for which negative transitivity holds ([22]).* □

For WOPs, domination and indifference between different domain values can be determined with the use of a numerical value. Thus, better-than tests can be specified efficiently by a numerical score function. We will study WOPs that can be characterized by a *level* function which maps a domain value to an integer value.

Definition 5. ***WOP Level Function***
For a WOP, every domain value can be mapped to an integer level value to determine domination. The function level has the following signature:

$$level : dom(A) \rightarrow \mathbb{N}_0$$

Better domain values have lower level values. The maximum level value *for a specific WOP P is denoted by* $\max(P)$*, the* minimum level value *always is 0.* □

Lemma 1. *For every WOP there is an integer level function to determine domination between two values:*

$$x <_P y \Longleftrightarrow level_P(x) > level_P(y)$$

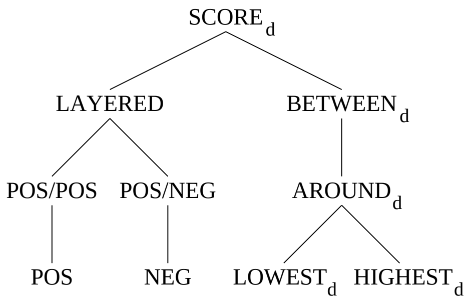

Figure 3.2: Taxonomy of weak order base preferences

Thus, substitutable values share the same level value. Every WOP we will discuss uses regular SV-semantics, *where* $x \cong_P y \Leftrightarrow x \sim_P y$*. We say* x *and* y *are* equivalent.

Proof. $\mathbb{N}_0$ *is a fully ordered set wrt "$>$". For every weak order, we can define a function mapping its elements to a total order. This function is* level *in our case.*

For each base preference constructor we will introduce now, there is a level function so that Lemma 1 holds for it – as a consequence, they all are WOPs. In the taxonomy of base preference constructors shown in Figure 3.2, the SCORE preference is the "super"-preference of all others, meaning that the others can be expressed by suitable SCORE preferences. It is reasonable to start an overview on base preference constructors with it.

A SCORE preference produces a numerical ranking. With our requirements for data structures and algorithms in coming chapters in mind, we will propose a definition of SCORE preferences slightly different from those presented in [37] and [38]. For our numerical values, smaller means better and zero means best:

Definition 6. ***SCORE$_d$* Preference**
Consider a scoring function $f : dom(A) \rightarrow \mathbb{R}_0^+$ *and a value* $d > 0$*.* P *is called a SCORE$_d$ preference, iff for* $x, y \in dom(A)$*:*

$$x <_P y \Leftrightarrow \left\lceil \frac{f(x)}{d} \right\rceil > \left\lceil \frac{f(y)}{d} \right\rceil$$

□

Please note that the original SCORE preference takes higher scores as "better" and has the value set $\mathbb{R}$ (see [37]). Both modifications can be simulated by proper modifications of f.

The so-called d-parameter partitions the value set of f. Different input values that may even have different function values for f are mapped to a single integer value and so eventually become substitutable. This partition helps to model user preferences better and more intuitive for a user as we will see later.

Theorem 1. *A $SCORE_d$ preference with a scoring function f is a WOP. Its level function is given by:*

$$level_{SCORE_d}(x) := \left\lceil \frac{f(x)}{d} \right\rceil$$

Proof. *It is enough to combine Definition 6 and Theorem 1:*

$$\begin{array}{rrcl} x <_P y & \Leftrightarrow \quad \left\lceil \frac{f(x)}{d} \right\rceil & > & \left\lceil \frac{f(y)}{d} \right\rceil \\ & \Leftrightarrow \quad level_{SCORE_d}(x) & > & level_{SCORE_d}(y) \end{array}$$

So we have shown that the SCORE preference is a WOP. For more convenience in preference modeling, we will now derive a number of different preference constructors from it. These constructors will consist of the preference *name*, the target *attribute*, and the required *parameter*s:

name(attribute; parameters)

A $SCORE_d$ preference is constructed by:

$$SCORE_d(A; f)$$

The d-parameter is given in the index, the attribute is denoted by A, and the scoring function f is the only parameter.

As already mentioned, there are different kinds of base preferences resulting from different kinds of domains. We distinguish between numerical and categorical domains. First, we will see some numerical base preferences, afterwards categorical ones.

Numerical Base Preferences

As the name implies, numerical base preferences deal with numerical domains such as physical measures or prices. We will start our overview of them with the most generic one, BETWEEN$_d$, a direct child of SCORE$_d$ in the taxonomy of Figure 3.2.

With a BETWEEN$_d$ preference, a user expresses that his favorite values lie between a lower bound *low* and an upper bound *up*:

Definition 7. ***BETWEEN$_d$* *Preference***
Consider a preference P on a domain A, and two values $low, up \in dom(A)$ with $low \leq up$, P is a BETWEEN$_d$ preference *iff it is a SCORE$_d$ preference with the following scoring function:*

$$f(x) := \begin{cases} low - x & \Leftrightarrow \quad x < low \\ 0 & \Leftrightarrow \quad low \leq x \leq up \\ x - up & \Leftrightarrow \quad up < x \end{cases}$$

The constructor of a BETWEEN$_d$ preference *is given by:*

$$BETWEEN_d(A; low, up)$$

□

Lemma 2. *For a preference $P := BETWEEN_d(A; low, up)$, the maximum level value* $\max(P)$ *can be found by*

$$\max(P) = \max(level_P(\min(dom(A))), level_P(\max(dom(A))))$$

with $\min(dom(A))$ *as minimal and* $\max(dom(A))$ *as maximal element of* $dom(A)$.

Proof. *Following Theorem 1, higher values for the scoring function f will lead to higher level values (with a fixed value of d).*
In Definition 7 we see that function values for f get bigger with:

a) $x < low$ and getting smaller (hence $low - x$ getting bigger)

b) $x < up$ and getting bigger (and with it $x - up$)

The highest function value for case a) is generated by the smallest value of x, the minimum of $dom(A)$ or $\min_A$. *Analogously, in case b) the maximum of $dom(A)$ yields the highest value for f. The overall maximum must be the maximum of these two values, which is given by:*

$$\max(f(\min_A), f(\max_A))$$

The maximum of the value for level that $\min_A$ *and* $\max_A$ *produce clearly is the maximum level value. Replacing f by level in the above formula, we see Lemma 2.*

In Figure 3.3, we can see the coherence for f and *level* for a BETWEEN$_d$ preference schematically. Due to rounding up all non-integer values, *level* is always equal to or higher than f. The biggest possible difference between both always is d. A brief example will show us when and how BETWEEN$_d$ preferences are used. Moreover, we will see how the d parameter enables us to model preferences in a more comprehensible and hence better way for users.

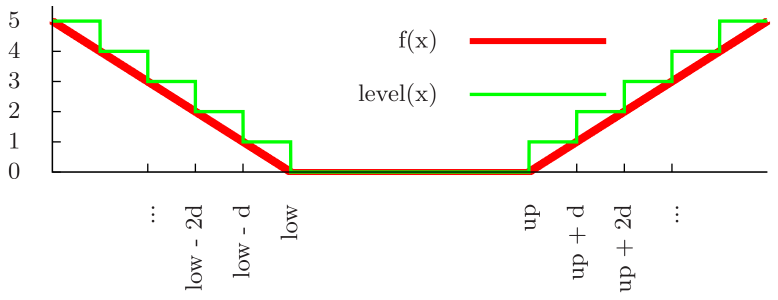

Figure 3.3: BETWEEN$_d$: $f(x)$ and $level_{BETWEEN_d}(x)$

Example 1. *John is looking for a rental car. He wants to spend €60 to €80 per day. He does not care about price differences of €5 or less. John's preference can easily be expressed with a BETWEEN$_d$ preference:*

$$P_{price} := BETWEEN_5(price; 60, 80)$$

If the price domain would not be partitioned by d, a rental car with a price of €58.99 would be rated worse than one for €59.00. The values are numerically different, but in a range that is not recognized by many users. Users would be rather surprised not to get both results. Of course, the value for d has to be defined carefully for each and every user and query.

In many cases, users will express preferences for precise values, not for intervals. Then we speak of AROUND$_d$ preferences.

Definition 8. ***AROUND$_d$ Preference***
Consider a preference P on a domain A, and a value $z \in dom(A)$, P is an AROUND$_d$ preference *iff it is constructed by:*

$$AROUND_d(A; z) := BETWEEN_d(A; z, z)$$
□

Two other preferences can be derived from the BETWEEN$_d$ preference: the extremal preferences HIGHEST$_d$ and LOWEST$_d$.

Definition 9. ***Extremal Preferences***
HIGHEST$_d$ and LOWEST$_d$ model preferences for extremal values. Consider a relation R with an attribute A and a supremum $sup \in dom(A) : \nexists x \in R : sup < x$ and an infimum $inf \in dom(A) : \nexists x \in R : x < inf$, they can be defined using an AROUND$_d$ preference:

a) HIGHEST$_d$(A; sup) := AROUND$_d$(A; sup)

b) LOWEST$_d$(A; inf) := AROUND$_d$(A; inf)

$\square$

The extremal preferences allow users to easily express their desire for values as high or as low as possible, in some domains a very common wish. The next example will present such a case:

Example 2. *John has another preference for rental cars. He wants the car to be as new as possible. The newest car in the database has an age of 0, which obviously is the domain's infimum value. John's preference is expressed as follows:*

$$P_{age} := LOWEST_1(age; 0)$$

We have seen a number of numerical preferences providing easy-to-use preference modeling. Using them, even unexperienced users can express preferences on numerical data, as they closely follow concepts known from everyday speech. As not all domains are numerical, we will get to know another type of base preferences for such domains in the following section.

Categorical Base Preferences

In many domains, numerical rankings cannot be applied, as, for example, for colors or brands. To offer users intuitive preference modeling, we have to define special constructors for non-numeric or – as we call them – categorical domains.

The first categorical base preference constructor we will look at is LAYERED$_m$, this being also the most univeral one. Analog to BETWEEN$_d$, in the last section we will see how to construct other preferences using the LAYERED$_m$ constructor.

Definition 10. ***LAYERED$_m$ Preference***
Consider an attribute A, with $L = (L_1, L_2, \ldots, L_{m+1})$ (with $m \geq 0$) being an ordered list of $m + 1$ sets forming a partition of dom(A). P is a LAYERED$_m$ preference iff it is a SCORE$_1$ preference with the following scoring function:

$$f(x) := i - 1 \Leftrightarrow x \in L_i$$

That means, $\forall x \in L_i : f(x) = i - 1$. A LAYERED preference on attribute A is constructed by:

$$LAYERED_m(A; L_1, \ldots, L_{m+1})$$

For convenience, one of the L_i may be named "others", representing the set dom(A) without the elements of the other subsets. $\square$

The level function for a LAYERED$_m$ preference is trivial. Level values are determined by the number of sets in the partition a domain value belongs to.

Lemma 3. *For a LAYERED$_m$ preference, it holds that $f(x) = level(x)$. So, for a LAYERED$_m$ preference P, it holds that: $\forall x \in L_i : level_P(x) = i - 1$.*

Proof. *According to Definition 10, the constructor for LAYERED$_m$ preferences is based on a SCORE$_1$ preference. With the score function f having only positive integer results for LAYERED$_m$ preferences and a d parameter of 1, Theorem 1 shows the lemma's content.*

So, in a LAYERED$_m$ preference, elements are ordered in sets wrt the preference. Elements of sets with lower indexes are preferred to others. Example 3 illustrates the use of a LAYERED$_m$ preference.

Example 3. *The color of diamonds is graded in special scales. One of them is issued by the Gemological Institute of America Inc[1]. It can be found in Table 3. Each linguistic variable (e.g. "colorless") is represented by some letters (e.g. D, E, and F). The domain of ratings is all letters from "D" as best rating to "Z" and finally "Z+" as worst rating and so forms a more fine-grained scale.*

colorless	near colorless	faint yellow	very light yellow	fancy yellow	fancy
D, E, F	G, H, I, J	K, L, M	N, O, P, Q, R	S to Z	Z+

Table 3.1: GIA Diamond color scale

In databases, the color is usually represented by its letter. To express a preference for less colorful diamonds based on the linguistic terms, a LAYERED$_m$ preference can be constructed as follows:

$$\begin{aligned} LAYERED_5\,(\,clarity;\ & \{D, E, F\},\\ & \{G, H, I, J\},\\ & \{K, L, M\},\\ & \{N, O, P, Q, R\},\\ & others,\\ & \{Z+\}) \end{aligned}$$

Of course, we could also specify the complete set $\{S, T, U, V, W, X, Y, Z\}$ instead of using "others" as a substitute for L_4.

[1]http://www.gia.edu/library/4286/6281/faq_detail_page.cfm

The maximum level value for LAYERED$_m$ preference is easy to determine, as we will see in the next theorem. In contrast to base preferences on numerical domains (see Lemma 2), only the definition of the preference is of interest.

Theorem 2. *LAYERED$_m$ preference P has a maximum level value of* $\max(P) = m$.

Proof. *Following the definition of LAYERED$_m$ preferences, the highest value found for its scoring function f is m for elements of L_{m+1}, and so it is for the level function (see Lemma 3).*

A number of special cases for LAYERED$_m$ will cover the most common cases of user preferences on categorical domains. The number of value sets a user can or wants to distinguish between regarding quality is often relatively small. We will have a look at these sub-constructors of LAYERED$_m$ now.

Definition 11. ***POS/POS Preference***
In a POS/POS preference *on an attribute A, a set of values $S_1 \in dom(A)$ is preferred to a set of values $S_2 \in dom(A)$. Both are preferred to other values in their domain. A* POS/POS preference *can be constructed on base of a $LAYERED_2$ preference:*

$$POS/POS(A; S_1, S_2) := LAYERED_2(A; S_1, S_2, others) \qquad \square$$

Definition 12. ***POS Preference***
A POS preference *on an attribute A specifies the preference of a set of values $S \in dom(A)$ over all other values of $dom(A)$. It is constructed as a special case of POS/POS preference:*

$$POS(A; S) := POS/POS(A; S, \emptyset) \qquad \square$$

Example 4. *Marilyn has a preference for only the brightest diamonds. According to the GIA color scale for diamonds in Table 3, the brightest color for those gemstones is called "colorless". Marilyn's preference can be defined as follows:*

$$P_{Marilyn} := POS(color; 'colorless')$$

It is common to explicitly express dislikes together with some preferences. The next two preference constructors make this easy to formulate on the base of LAYERED$_m$ preferences.

Definition 13. ***POS/NEG Preference***
With a set $S_1 \in dom(A)$ of liked and a set $S_2 \in dom(A)$ of disliked values, a POS/NEG preference *constructor on attribute A is defined as:*

$$POS/NEG(A; S_1, S_2) := LAYERED_2(A; S_1, others, S_2) \qquad \square$$

Example 5. *For a rental car, John prefers the colors red and blue above all others. He completely dislikes purple. This can easily expressed by a POS/NEG-preference:*

$$P_{col} := POS/NEG(color, \{red,\ blue\}; \{purple\}).$$

Red and blue are unranked among each other, but better than any other element in the domain of colors. Purple is worse than anything else.

In some cases, there may only be some domain values a user can specify as unfavored. This special type of POS/NEG preference is the last base preference we will discuss here.

Definition 14. ***NEG Preference***
With a set of disliked values $S \in dom(A)$, a NEG preference *on attribute A can be constructed as follows:*

$$NEG(A; S) := POS/NEG(A; \emptyset, S)$$

□

We have now seen a number of weak order base preferences offering a wide range of instruments for intuitive preference modeling. In many cases however, users will have more than just one base preference on a domain. Some mechanism to combine preferences in order to model complex preferences is therefore needed. This will be the subject of the next section.

3.1.2 Complex Preferences

Base preferences on simple attributes clearly form the basis of preference modeling. But when preferences on multiple attributes of one object or tuple are defined, we need ways to combine them. In [37, 38], three different types of complex preferences have been introduced: Pareto preferences, prioritization and numerical preferences. We will discuss all of them now.

Pareto Preferences

The most intuitive way of combining preferences works without any need of rating the importance of them. In a *Pareto preference*, all contained preferences are equally important. A tuple is better than another one, if it is not worse in any contained preference and better in at least one of them. This is called the *Pareto principle* (named after Italian sociologist and economist Vilfredo Pareto).

Definition 15. ***Pareto Preferences***
For WOPs $P_1 = (A_1, <_{P_1}), \ldots, P_m = (A_m, <_{P_m})$, a Pareto preference

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

on two tuples $x = (x_1, \ldots, x_m), y = (y_1, \ldots, y_m)$ is defined as:

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

□

As the domination relation for a Pareto preference is based on level values of the contained preferences, it seems to be the logical consequence to define a recursive level function for Pareto preferences as well.

Definition 16. ***Level function for Pareto preferences***
The level function of a Pareto preference $P = \otimes(P_1, \ldots, P_m)$ for an arbitrary input tuple $x = (x_1, \ldots, x_m)$ is given by:

$$level_P(x) = \sum_{i=1}^{m} level_{P_i}(x_i)$$

□

We will see the reasons for this definition of a level function for preferences when we will discuss *better-than graphs* in chapter 4. According to [15], Pareto preferences form strict partial orders, even when combining only weak orders. So the WOP coherence of domination and differences in level values (cp. Lemma 1) does not apply. In particular, domination requires a higher level value, but a higher level value is not synonymous with being dominated. We will see this in the following lemma.

Lemma 4. *For a Pareto preference P it holds that*

$$x <_P y \Rightarrow level_P(x) > level_P(y).$$

Proof. *Consider a Pareto preference $P := (A, <_P)$ and two tuples $s = (s_1, \ldots, s_m)$, $t = (t_1, \ldots, t_m) \in dom(A)$.*

We assume $s <_P t$. Following Definition 15, each level value for s is at least as high as the corresponding value for t and one of the values for s has to be higher. The minimum for the level value of s is $level_P(t) + 1$.

The next example will show the implication *bigger level $\Rightarrow$ worse value* does not hold for Pareto preferences (or strict partial orders in general). The determination of domination by comparing a single numerical value is a computational privilege only for WOPs.

Example 6. *We have a Pareto preference $P := \otimes(P_1, P_2)$ with maximum level values* $\max(P_1) = \max(P_2) = 4$ *and a number of tuples (attribute values are represented by their level values for improved readability):*

$$t_1 := (1,1), t_2 := (1,2), t_3 := (2,1), t_4 := (0,4)$$

t_1 has a level value of $1+1=2$ and dominates t_2 and t_3, both with level values of 3. But although t_4 has the highest level value of all, it is not dominated by any of the other tuples.

Prioritization

Preferences are not equally important in every case. *Prioritized preferences* or *prioritizations*, introduced in [37, 38], allow the modeling of combinations of preferences that have different importance. Although generally not restricted, we will only deal with prioritizations containing WOPs.

Definition 17. ***Prioritization***
For WOPs $P_1 = (A_1, <_{P_1})$, ..., $P_m = (A_m, <_{P_m})$, a prioritization

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

on two tuples $x = (x_1, \ldots, x_m), y = (y_1, \ldots, y_m)$ is defined as:

$$\begin{aligned}(x_1, \ldots, x_m) <_P (y_1, \ldots, y_m) \quad \Leftrightarrow \quad & \exists k \in \{1, \ldots, m\} : \forall i \in \{1, \ldots, k-1\} : \\ & x_i \cong_{P_i} y_i \wedge x_k <_{P_k} y_k\end{aligned}$$

□

A prioritization containing only WOPs itself is a WOP again ([15]). Hence there has to be a *level* function to calculate dominance.

Theorem 3. *The level function for a tuple $x := (x_1, \ldots, x_m)$ in a* prioritization *$P := \&(P_1, \ldots, P_m)$ with all the P_i being WOPs is given by:*

$$level_{\&(P_1,\ldots,P_m)}(x) := \sum_{i=1}^{m} \left(level_{P_i}(x_i) * \prod_{j=i+1}^{m} (\max(P_j) + 1) \right)$$

Proof. *Consider a prioritization $P := (A, <_P)$ and tuples $s = (s_1, \ldots, s_m)$, $t = (t_1, \ldots, t_m)$, both in dom(A). We assume $s <_P t$. Following Definition 17, there is a k for which $s_k <_{P_k} t_k$. For all $i < k$, the corresponding attributes s_i and k_i are* *substitutable and so (cp. Lemma 1) $level_{P_i}(s_i) = level_{P_i}(t_i)$. We will now look at $level_P(t) - level_P(s)$, which, with P being a WOP, has to be negative:*

$$\sum_{i=k}^{m} \left((level_{P_i}(t) - level_{P_i}(s)) * \prod_{j=i+1}^{m} (\max(P_j) + 1) \right) < 0$$

We know that $level_{P_k}(s_k) > level_{P_k}(t_k)$. So the above inequation is true, iff the amount the level value of s has to be higher than t's level value due to $s_k <_{P_k} t_k$ is bigger than the sum of products with the biggest possible level values for t for P_{k+1}, ..., P_m.

As $level_{P_k}(s_k)$ has to be at least bigger than $level_{P_k}(t_k)$ by 1, hence it is to be proved:

$$\prod_{i=k+1}^{m} (\max(P_i) + 1) > \sum_{i=k+1}^{m} \left(\max(P_i) * \prod_{j=i+1}^{m} (\max(P_j) + 1) \right)$$

For $k = m$ this leads to $1 > 0$ as base step of our induction. Assuming the statement holds for some value of k, we will now check if it holds for $k-1$ (for readability, we will abbreviate $\prod_{i=a}^{b}(\max(P_i) + 1)$ by X_a^b):

$$\begin{aligned}
X_{k+1}^{m} &> \sum_{i=k+1}^{m} \left(\max(P_i) * X_{i+1}^{m}\right) \quad \textit{(hypothesis)} \\
X_{k}^{m} &> \sum_{i=k}^{m} \left(\max(P_i) * X_{i+1}^{m}\right) \quad \textit{(induction step: } k \rightarrow k-1\textit{)} \\
(\max(P_k) + 1) * X_{k+1}^{m} &> \sum_{i=k+1}^{m} \left(\max(P_i) * X_{i+1}^{m}\right) + \max(P_k) * X_{k+1}^{m} \\
\max(P_i) * X_{k+1}^{m} + X_{k+1}^{m} &> \sum_{i=k+1}^{m} \left(\max(P_i) * X_{i+1}^{m}\right) + \max(P_i) * X_{k+1}^{m} \\
X_{k+1}^{m} &> \sum_{i=k+1}^{m} \left(\max(P_i) * X_{i+1}^{m}\right)
\end{aligned}$$

By reducing the formula to our induction hypothesis, we have proved it to be correct. Being able to reduce it as far back as to $k = m$, we even get the following equation:

$$\prod_{i=k+1}^{m} (\max(P_i)+1) = \sum_{i=k+1}^{m} \left(\max(P_i) * \prod_{j=i+1}^{m} (\max(P_j) + 1) \right) + 1 \qquad (*)$$

So all values from 0 to some $\max(P)$ are valid level values.

A prioritization defines a lexicographic order on a domain ([15]). The level function maps each possible value to its position in the “lexicon”.

Lemma 5. *The maximum level value for a* prioritization P *is given by:*

$$\max(P) := \prod_{i=1}^{m} (\max(P_i) + 1) - 1$$

Each integer value in the closed interval $[0, \max(P)]$ *is a valid level value.*

Proof. *Following Theorem 3, the function value of* $\max(P)$ *is computed by maximizing the* P_i*'s level values:*

$$\max(\&(P_1, \ldots, P_m)) := \sum_{i=1}^{m} \left(\max(P_i) * \prod_{j=i+1}^{m} (\max(P_j) + 1) \right).$$

The formula marked with $(*)$ *at the end of the proof of Theorem 3 shows that this is equal to the product given in Lemma 5.*

There are $\prod_{i=1}^{m}(\max(P_i) + 1)$ *different combinations of level values for contained WOPs. As shown in the proof of Theorem 3, each combination of them yields a different value. The interval* $[0, \max(P)]$ *holds exactly* $\max(P) + 1 = \prod_{i=1}^{m}(\max(P_i) + 1) - 1 + 1$ *values.*

We will now see a level value function for a prioritization in an example.

Example 7. *Consider preferences* P_1 *and* P_2 *with* $\max(P_1) = 3$ *and* $\max(P_2) = 2$. *For the simple case of* $P = \&(P_1, P_2)$, *this leads to the following level function for a parameter tuple* $x = (x_1, x_2)$:

$$\begin{aligned} level_{\&(P_1,P_2)}(x) \quad := \quad & level_{P_1}(x_1) * (\max(P_2) + 1) + level_{P_2}(x_2) = \\ & level_{P_1}(x_1) * 3 + level_{P_2}(x_2) \end{aligned}$$

The maximum level value for P *would be* $(\max(P_1) + 1) * (\max(P_2) + 1) - 1 = 11$.

Numerical Preference

Analogously to the SCORE preference, the *numerical preference* computes score values on the base of domain values. It uses a number of SCORE preferences (or subconstructors of SCORE as seen in Figure 3.2) as input and merges their (weighted) function values to an overall score.

Definition 18. ***Numerical Preference***
Given some SCORE$_d$ (or subconstructor) preferences $P_1, \ldots, P_m$ with scoring functions $f_1, \ldots, f_m$. Then a numerical ranking preference $\text{rank}_{F,d}$ *with an m-ary combining function $F : \mathbb{R}^m \to \mathbb{N}_0$ is defined as:*

$$(x_1, \ldots, x_m) <_P (y_1, \ldots, y_m) \Leftrightarrow \left\lceil \frac{F(f_1(x_1), \ldots, f_m(x_m))}{d} \right\rceil > \left\lceil \frac{F(f_1(y_1), \ldots, f_m(y_m))}{d} \right\rceil$$

$\square$

Theorem 4. *A* numerical preference *is a WOP with the following level function:*

$$level_{rank_{F,d}}(x) := \left\lceil \frac{F(f_1(x_1), \ldots, f_m(x_m))}{d} \right\rceil$$

Proof. *Using this function in Definition 18 yields Lemma 1.*

Weighting preferences according to their importance is not very intuitive. Hardly anybody will be able to express that one preference is 2.0 or 2.5 times more important than another. Nevertheless, *numerical preference* can be used to simulate prioritizations, as we will see in the next example.

Example 8. *Consider the prioritization found in Example 7 with SCORE functions f_1 and f_2 for P_1 and P_2. An equivalent numerical preference can be defined, using a d-parameter of 1 and the ranking function*

$$F(f_1(x_1), f_2(x_2)) := (\max(P_2) + 1) * f_1(x_1) + f_2(x_2).$$

The simulation is done by defining the ranking function exactly identical to a prioritization's level function. A generalization to more than two input preferences is straightforward.

3.2 Evaluation of Preferences

With the overview of Kießling's most important preference constructors in mind, we will now study how these preferences can be used to enable a user-centered approach to database search. This surely is a key issue on the way to personalized systems. After introducing a query model designed for using preferences, we will see how it can be integrated into the standard database query language SQL, forming *Preference SQL*.

3.2.1 The BMO Query Model

Common database search mechanisms (like SQL) only support hard constraints. The search conditions are matched perfectly by each tuple in the result. But often there a perfect match cannot be found for a user's particular wish. An empty result set is the logical consequence. This often frustrating outcome is known as the *empty result effect* ([37]). To avoid this notorious problem, preferences are evaluated as *soft constraints*. That means, when a database is queried on base of some user preferences, those tuples that best match the user's wishes are returned. They may match perfectly, or they may not – but only those tuples are returned for which no better tuple can be found. This semantics of returning *best matches only* are formally defined as follows:

Definition 19. ***Best Matches Only***
The result of a preference query are best matches only *(*BMO*). For a preference* $P := (A, <_P)$ *on an input relation* R*, the BMO set is given by a* preference selection*:*

$$\sigma[P](R) := \{x \in R |\ \nexists x' \in R : x <_P x'\}$$

□

Please note that there is another way of avoiding the *empty result effect*. The number of hard constraints may be reduced, or they may be changed from being conjunctive (linked with `AND`) to disjunctive (linked with `OR`). But then a different problem arises: the constraints may become too weak to provide reasonable filtering and the user is *flooded* with results. This is called the *flooding effect*. An example will show how the BMO is found:

Example 9. *In Example 1 and Example 5, we have heard of John's preferences on cars. We will combine these preferences now to a Pareto preference:*

$$P_{John} := \otimes(BETWEEN_5(price; 60, 80), POS/NEG(color; \{red,\ blue\}, \{purple\})$$

When this preference is evaluated on the data in Table 3.2, the BMO set consists of the tuples with IDs 1, 2, and 3. The other tuples are discarded, as they are dominated. Table 3.3 shows the level values for the WOPs in P_{John}*.*

For tuple 4, the level value for the price preference is 1, just as for the color. This tuple is dominated by tuple 2 and 3, both with level value of 0 for the price (and 1 for the color). Tuple 5, the Skoda, has a perfect color, but its price is worse than the one of tuple 1. Tuple 6 is worse than any of the other tuples.

We have seen the BMO semantics at work in the example. Obviously only a tuple worse than at least one other with respect to the given preference is discarded. All tuples in the BMO result set are either substitutable or indifferent.

RENTAL_CARS	id	manufacturer	color	price
	1	VW	red	50
	2	Audi	black	70
	3	BMW	brown	75
	4	Toyota	yellow	55
	5	Skoda	red	45
	6	Hyundai	purple	45

Table 3.2: Sample database

id	1	2	3	4	5	6
$level_{P_{col}}(t_{id})$	0	1	1	1	0	2
$level_{P_{price}(t_{id})}$	2	0	0	1	3	3

Table 3.3: Level values for Example 9

Example 10. *Let's have a look at the preference from Example 9 only using standard SQL:*

```
SELECT *
FROM rental_cars
WHERE price BETWEEN 68 AND 80
  AND color IN ('red', 'blue')
```

No tuple in the database matches these constraints. The empty result effect *occurs, although a number of cars nearly matches the constraints – the results of the preference query of Example 9.*

The next section will show how preferences and BMO semantics are integrated in common query languages.

3.2.2 Preference SQL

With its first steps made under the name of *SEQUEL* in 1974 ([8]), the *Standard Query Language* (*SQL*) has now been the standard query language for relational databases for more than 20 years. In order to make preference querying available to a large audience, the preference based BMO query model was integrated into SQL, leading to the first version of *Preference SQL* (*PSQL*) in [40].

In addition to hard constraints stated in the `WHERE` clause of a SQL statement, filtering is also carried out in the new `PREFERRING` clause. A PSQL query has the following schematical design:

```
SELECT ...
FROM ...
WHERE ...
PREFERRING ...
```

The preference is evaluated on the results of the hard constraint stated in the `WHERE` clause. It follows the BMO semantics of Definition 19. As a consequence, empty result sets can only occur in cases when all tuples are filtered out by the `WHERE` clause. With knowledge of the preference constructors of Section 3.1, the syntax of the preference extensions is straightforward. Table 3.4 shows the Preference SQL expressions for some of them. The following issues have to be kept in mind for the syntax:

- The keyword `REGULAR` at the end of each base preference expression indicates the use of regular SV semantics, which is a vital property of WOPs. (Non-weak-order base preferences will be discussed later in Section 4.5.)
- The sets needed for a LAYERED$_m$ preference and its sub-constructors are entered as comma-separated values in brackets:

  ```
  ('a', 'b', 'c', ..., 'z')
  ```

- A scoring function cannot be stated directly in Preference SQL. Preference SQL is implemented in Java, and there is the possibility to declare a Java class providing a certain interface as scoring function. In the syntax of Table 3.4, the name of the class providing the scoring function f would be `Fclass`, handed over to the query engine as a string constant. This technique is used for numerical preferences as well.

Example 11. *John's Pareto preference (as seen in Example 9) can easily be expressed in Preference SQL:*

```
SELECT *
FROM rental_cars
PREFERRING price BETWEEN 60 AND 80, 5 REGULAR
       AND color IN ('red', 'blue')
                 NOT IN ('purple') REGULAR
```

Now let's say John's company has a policy for renting cars. Renting cars from premium manufacturers like Audi or BMW is not permitted. This hard constraint has to be added to the query:

Preference constructor	Preference SQL expression
$SCORE_d(A; f)$	`A SCORE 'Fclass' d`
$BETWEEN_d(A; low, up)$	`A BETWEEN low AND up, d REGULAR`
$AROUND_d(A; z)$	`A AROUND z, d REGULAR`
$HIGHEST_d(A; sup)$	`A HIGHEST sup, d REGULAR`
$LOWEST_d(A; inf)$	`A LOWEST inf, d REGULAR`
$LAYERED_m(A; L_1, \ldots, L_{m+1})$	`A LAYERED (`L_1`, ..., `L_{m+1}`) REGULAR`
$POS/POS(A; S_1, S_2)$	`A IN `S_1` ELSE `S_2` REGULAR`
$POS(A; S)$	`A IN `S` REGULAR`
$POS/NEG(A; S_1, S_2)$	`A IN `S_1` NOT IN `S_2` REGULAR`
$NEG(A; S)$	`A NOT IN `S` REGULAR`
$\otimes(P_1, \ldots, P_m)$	P_1` AND ... AND `P_m
$\&(P_1, \ldots, P_m)$	P_1` PRIOR TO ... PRIOR TO `P_m
$rank_{F,d}(P_1, \ldots, P_m)$	`(`P_1`, ..., `P_m`) RANK 'Fclass' d`

Table 3.4: Preference SQL Syntax

```
SELECT *
FROM rental_cars
WHERE manufacturer <> 'Audi'
  AND manufacturer <> 'BMW'
PREFERRING price BETWEEN 60 AND 80, 5 REGULAR
       AND color IN ('red', 'blue')
                 NOT IN ('purple') REGULAR
```

The tuples with IDs 2 and 3 are discarded by the hard constraint even before the preference is evaluated. As a consequence, the query result changes. Tuple 1 stays in the BMO set, and tuple 4 becomes the second element of it, as it had only been dominated by tuples 2 and 3 (see Example 9).

The preference model and the preference algebra defined by Kießling can be seamlessly integrated in relational algebra and in particular into heuristical optimization algorithms. As shown in [34], this leads to great performance gains.

PSQL is not the only example of integration of this preference model into standard query languages. [39] presented an integration of it into the first version of W3C's XML query language *XPath*[2] under the name of *Preference XPath*. With *XPath* being less important than other XML query languages these days, and all of them playing only marginal roles in databases nowadays, it is omitted here.

[2]http://www.w3.org/TR/xpath

Chapter 4

The Better-Than-Graph

The preference model introduced in chapter 3 provides us with a powerful means to express user preferences. We will now see a graphical representation of such preferences. Visualization in general can be a powerful technique to improve comprehensibility, but in our case, this goes even further. *Better-than graphs* (*BTG*s) form the basis of further results in this book. After introducing *better-than graphs*, we will carefully examine their structure and their measures.

4.1 The BTG as a Graphical Representation of Orders

Basically, a *better-than graph* (BTG) is simply a visualization of the domination of domain elements for a preference. For each equivalence class it has a node, and for each direct domination of one node over another, it has an edge.

Definition 20. ***Better-Than Graph***
Consider a preference $P := (A, <_P)$ on an attribute or attribute set A with the domain $dom(A)$. The better-than graph *for P (*BTG_P*) than has:*

- *one node for each equivalence class in $dom(A)$*
- *a directed edge (a_1, a_2) from a_1 to a_2 for each pair of nodes a_1, a_2 for which holds:*

$$a_2 <_P a_1 \wedge (\neg \exists a_3 \in dom(A) : a_2 <_P a_3 <_P a_1)$$

□

Please note that we will use the terms *node* and *equivalence class* synonymously, as an equivalence class for a preference P is represented by exactly one node in the BTG for P and vice versa.

As Definition 20 shows, edges following from transitivity of domination are omitted. This kind of visualization for all kinds of partial orders is also known as *Hasse diagram* ([22]).

As a special characteristic of our graphs, we have a concept of levels of the nodes in the BTG:

Definition 21. ***Level***
The level of a node in a BTG is the length of the longest path leading to it from an undominated node. □

BTGs for WOPs and BTGs for Pareto preferences differ very much, as Pareto preferences form no weak orders. Hence, we will analyze them separately from each other, starting with the more simple BTGs for WOPs.

4.2 BTGs for WOPs

As we already know, for a WOP each level value represents exactly one equivalence class in the input domain. Therefore, the BTG for a WOP P contains exactly one node for each value in the value set of the function $level_P$, which is the closed interval of integer numbers $[0, max(P)]$. Domination of one node over another can be directly derived from the level value, which is a characteristic of WOPs. This leads to BTGs as shown in Figure 4.1. We can see that this is a total order of the level values. 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).

4.2.1 Structure

BTGs for WOPs show a chain structure. Each node is dominated by exactly one node (except for the top node) and is dominating exactly one node itself (except for the bottom node). The schematic Figure 4.1 illustrates this, while Example 12 shows the BTG for a given instance of the POS/NEG preference.

Example 12. *In Example 5, we heard about John's preference on colors:*

$$P_{col} := POS/NEG(color, \{red,\ blue\}; \{purple\}).$$

Figure 4.2 shows the BTG for this preference and the domain values the nodes are representing.

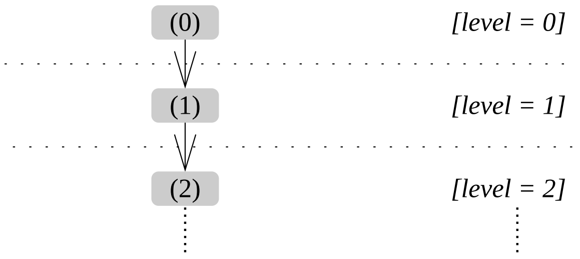

Figure 4.1: Structure of BTGs for WOPs

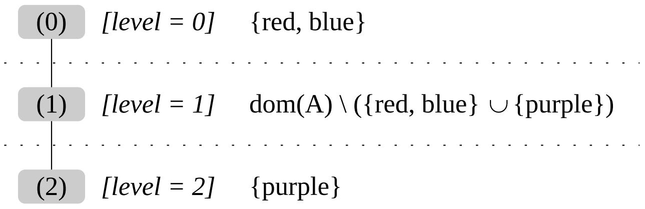

Figure 4.2: BTG for a POS/NEG preference

4.2.2 Measures

We will now discuss some measures of BTGs for WOPs. This will be width, height, and the number of nodes. First, we will look at the relation of the level of a node in the BTG and the level function value of domain values for the corresponding WOP.

Theorem 5. *The value of the level function of a WOP $P := (A, <_P)$ for any value $v \in dom(A)$ is equal to the level of v's corresponding node in the BTG for P.*

Proof. *Consider a WOP $P := (A, <_P)$ and a set of values $T = (t_1, \ldots, t_p) \in dom(A)$ that is not dominated. For each of the $t_i \in T$, $level_P(t_i) = 0$ (cp. Definition 4). In the BTG for WOP, all elements of T are mapped to the top node.*

In another set of values $V = (v_1, \ldots, v_q) \in dom(A)$, each element is dominated only by the elements of T (and no other values). As there is no other value w with $v_i <_P w <_P t$, $\forall v_i \in V : level_P(v_i) = 1$.

The node the different v_i belong to has a level value of 1 as it can be reached by a path with a length of 1 coming from the top node. If there were another node between *the one for T and the one for V, all v_i would be dominated by elements of $dom(A)$ mapped to this node. This is a contradiction to the selection of V.*

By induction, it is obvious that this correlation of BTG node level and preference level holds for the full BTG.

With the equivalence of a preference's level function and the level value in the corresponding BTG, we can now use level functions to form easily applicable functions to compute BTG measures. Nonetheless, we will start with the trivial value of BTG width.

Theorem 6. *The width of a BTG for a WOP is 1 in each level.*

Proof. *Each equivalence class is identified by a single integer value. Each of them therefore forms its own level.*

Theorem 7. *The height of the BTG of a WOP P is $height(BTG_P) = \max(P) + 1$.*

Proof. *Each equivalence class is in a separate level. As a consequence there are $\max(P) + 1$ equivalence classes.*

Theorem 8. *A BTG for a WOP P has $nodes(BTG_P) = height(BTG_P)$ different nodes.*

Proof. *For each WOP P, there are $height(BTG_P)$ different levels. The number of nodes is identical to the number of equivalence classes.*

The very simple chain structure of BTGs for WOPs makes their analysis very simple. In the next section, we will study BTGs for Pareto preferences, which are more complex and thus more interesting.

4.3 BTGs for Pareto Preferences

Just as for WOPs, the BTG of a Pareto preference has a node for every equivalence class wrt the preference. But with Pareto preferences containing WOPs forming true *strict partial orders* (cf. [15]), their BTGs are more complex.

As we will concentrate on Pareto preferences on WOPs, in our BTGs for Pareto preferences, domain values will be represented by the level values they are mapped to by the WOPs. As we have seen in Definition 15, the domain of a Pareto preference $P := \otimes(P_1, \ldots, P_m)$ is given by the Cartesian product of domains of the $P_i \in P$. As each value in the domain of each P_i can be represented by its level value, a node in a BTG for P will be a combination of level values for the P_i. Again, we will first explore the structure of the BTG and analyze its measures afterwards.

4.3.1 Structure

A BTG for a Pareto preference with only WOPs as input preferences is a complete distributive lattice. To be able to prove this, we first have to provide a definition for lattices (cp. [22]).

Definition 22. ***Lattice***
An ordered set S that has an infimum *and a* supremum *for any two of its elements is a* lattice. *If an* infimum *and a* supremum *is defined for all subsets of S, we have a* complete lattice. □

The set the lattice is defined on only needs to be partially ordered, just as the equivalence classes of a Pareto preference are (and hence the nodes of the BTG for such a preference). The next theorem will show that BTGs are lattices.

Theorem 9. *To find the* supremum *of nodes $x = (x_1, \dots, x_m)$ and $y = (y_1, \dots, y_m)$ from a BTG, we have to use the minimum level value of these nodes for each WOP:*

$$\min(x, y) = (\min(x_1, y_1), \dots, \min(x_m, y_m))$$

To find the infimum *of nodes x and y from a BTG, we have to use the maximum level value of these nodes for each WOP:*

$$\max(x, y) = (\max(x_1, y_1), \dots, \max(x_m, y_m))$$

Domination in the BTG can be computed with these functions min *and* max. *Thus all BTGs are lattices.*

Proof. *Consider two nodes of a BTG, $x = (x_1, \dots, x_m)$ and $y = (y_1, \dots, y_m)$. If x dominates y, clearly x has to be the supremum of both and y has to be the infimum. As x dominates y, no x_i can be bigger than the corresponding y_i and at least one x_i has to be smaller (see Definition 15). Hence $x_i = \min(x_i, y_i)$ and $y_i = \max(x_i, y_i)$.*

If x and y are indifferent, then $\exists j : \min(x_j, y_j) = x_j \wedge \exists k : \min(x_k, y_k) = y_k$.

A node $s = (\min(x_1, y_1), \dots, \min(x_m, y_m))$ is a candidate for the supremum of x and y as it dominates both. If there is no node z with $x <_P z <_P s$, we have shown that s must be the supremum.

Suppose there is such a z. We choose some j for which $x_j \leq y_j$. If $z_j > s_j$, (with all other values being equal), s would dominate z. But z would not dominate x as $x_j = s_j$. Thus there can be no such z. It is clear that this is true just as well for any k with $x_k \geq y_k$ and vice versa for the supremum.

Every pair of nodes has one infimum (or supremum), which is one node of the BTG. To prove completeness, we can split finding the infimum (or supremum) of a set of

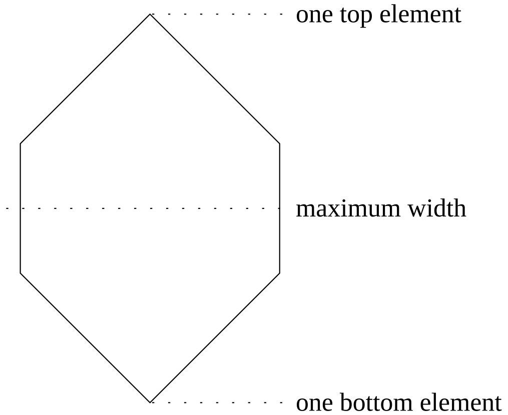

Figure 4.3: Structure of BTGs for Pareto preferences

nodes N into multiple problems of finding the infimum (or supremum) for pairs of nodes of this set. We combine the resulting nodes in the same way until only one node is left: the infimum (or supremum) of N.

The distibutive nature of min *and* max *leads to component-wise distributivity of the x_i and y_i and therefore to distributivity of the lattice.*

Please note that this theorem also holds for Pareto preferences built from WOPs with $m = 1$ and as a consequence for simple WOPs as well, although it is less relevant as the BTGs for WOPs describe total orders on intervals on $\mathbb{N}_0$. Following [22], the product of two lattices K and L produces a lattice with one instance of L for each node in K (i.e. each level combination of L is joined with each level combination in K), the BTG for a Pareto preference may also be interpreted as the product of the lattices for the $P_i \in P$.

A BTG for a Pareto preference $P := \otimes(P_1, \ldots, P_m)$ has a single top node (i.e. the combination of best values for the P_i) and a single bottom node (i.e. the combination of worst values for the P_i). The general structure of a BTG is a hexagon, as shown in Figure 4.3. In Example 13 we can see a Pareto preference and the BTG formed by it.

Example 13. *Remember John's preferences on the price (see Example 1) and color of rental cars (see Example 5). In Example 9, the two WOPs were combined in a Pareto preference:*

$$P_{John} := \otimes(BETWEEN_5(price; 60, 80), POS/NEG(color; \{red,\ blue\}, \{purple\}))$$

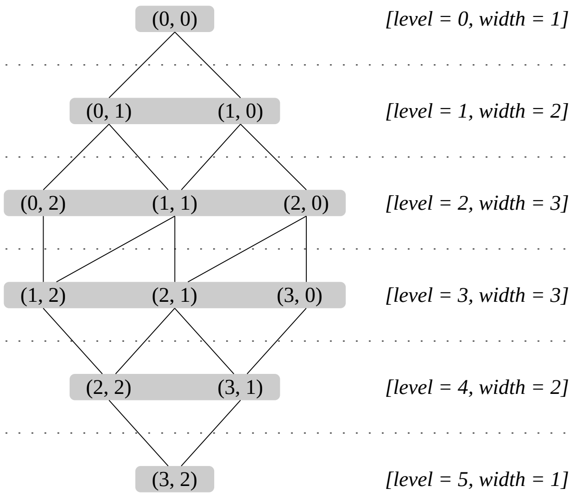

Figure 4.4: BTG for a Pareto preference

Using it on the car database of Table 3.2, we get a maximum level value of 3 for the price preference. POS/NEG preferences always have a maximum level value of 2 (cp. Theorem 2). So, $BTG_{P_{John}}$ is characterized by the values 3 and 2. Figure 4.4 shows $BTG_{P_{John}}$ and its typical hexagon shape.

In the next section, we will discuss the measures of the BTG, giving us clear explanations for the structure of the BTG for a Pareto preference.

4.3.2 Measures

Understanding the measures of a BTG for a Pareto preference is essential in understanding the BTG itself. The greater complexity (compared to BTGs for WOPs) leads to a more complex analysis – that means, we will see a greater range of different measures. This will give us the insight we need for the design of new algorithms in later parts of this book. We will begin with the number of nodes in a BTG for a Pareto preference.

Theorem 10. *For a Pareto preference $P := \otimes(P_1, \ldots, P_m)$, the number of nodes in its better-than graph BTG_P is:*

$$nodes(BTG_P) = \prod_{i=1}^{m}(\max(P_i) + 1)$$

Proof. *Each node holds a different combination of level values for the $P_i \in P$. For any P_i, the valid values are all* $\max(P_i)+1$ *integer numbers in the interval* $[0, \max(P_i)]$. *Combining all possible combinations of level values for all the P_i, we get the formula in Theorem 10.*

Closely connected with the number of nodes in a BTG is the number of edges, as we will see in the next theorem.

Theorem 11. *The BTG for a Pareto preference $P := \otimes(P_1, \ldots, P_m)$ has the following number of edges:*

$$edges(BTG_P) = \prod_{i=1}^{m}(\max(P_i)+1) * \sum_{i=1}^{m} \frac{\max(P_i)}{\max(P_i)+1}$$

Proof. *Have a look at those edges in the Pareto preference P's BTG resulting from domination because of one P_i. All nodes with a level value smaller than* $\max(P_i)$ *for P_i have such an outgoing edge. The number of such nodes is given by*

$$\frac{nodes(BTG_P)}{\max(P_i)+1} * \max(P_i).$$

These are all nodes with any possible level values for all $P_j \in P$ with $i \neq j$ and a level value in the interval of $[0, \max(P_i)-1]$ *for P_i. This leads to the following sum of edges for all preferences in P:*

$$\sum_{i=1}^{m}\left(\frac{nodes(BTG_P)}{\max(P_i)+1} * \max(P_i)\right) = nodes(BTG_P) * \sum_{i=1}^{m}\left(\frac{\max(P_i)}{\max(P_i)+1}\right)$$

Just as for WOPs, each node in a Pareto preference's BTG has a level value, although it is no longer a unique identifier of the node. In contrast to the simple BTGs for WOPs, the number of nodes in the BTG for a Pareto preference does not have a direct impact on the number of levels (which is equal to the height) of the BTG. The following theorem shows how to determine such a node's level value:

Theorem 12. *For the BTG of a Pareto preference $P := \otimes(P_1, \ldots, P_m)$, the level function can be computed as follows:*

$$level_P(a_1, \ldots, a_m) = \sum_{i=1}^{m} level_{P_i}(a_i)$$

Proof. *Consider some tuple $a \in dom(A)$ having the vector $v_0 = (0, \ldots, 0)$ as its equivalence class (or node in the BTG). By replacing exactly one of the 0's by a 1, we obtain the equivalence class $v_1 = (0, \ldots, 0, 1, 0, \ldots, 0)$.*

According to the Pareto definition of P we have $v_1 <_P v_0$. With level being an integer-valued function, v_1 characterizes a direct successor of v_0 in the BTG of P. Thus if $(\overline{a}_1, \ldots, \overline{a}_m)$ represents a value with level vector v_1 and $(a_1, \ldots, a_m)$ represents a value with level vector v_0, then

$$level_P(\overline{a}_1, \ldots, \overline{a}_m) = level_P(a_1, \ldots, a_m) \ + \ 1 \ = \ 0 \ + \ 1 \ = \ 1.$$

In general, consider

$$\begin{aligned} v_n &= (level_{P_1}(a_1), \ldots, level_{P_m}(a_m)) \text{ and} \\ v_{n+1} &= (level_{P_1}(a_1), \ldots, level_{P_{i-1}}(a_{i-1}), level_{P_i}(a_i) + 1, level_{P_{i+1}}(a_{i+1}), \ldots), \end{aligned}$$

differing from v_n by increasing exactly one level from $level_{P_i}(a_i)$ to $level_{P_i}(a_i) + 1$. Assuming that v_n represents a node at level

$$n = \sum_{i=1}^{m} level_{P_i}(a_i)$$

in the BTG of P, then v_{n+1} represents a direct successor of v_n at level

$$n + 1 = \sum_{i=1}^{m} level_{P_i}(a_i) + 1.$$

Thus by induction over the height of the BTG of P the proof is achieved.

With the level value being the sum of other values it is clear that there is more than a single node in most of the levels. Theorem 13 shows how to determine the height of the BTG for a Pareto preference.

Theorem 13. *The height of the BTG for a Pareto preference $P = \otimes(P_1, \ldots, P_m)$ is given by:*

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

Proof. *The height of a BTG is defined by the number of different levels it has. Theorem 12 describes how to find the level value for a single node. The minimum level value is 0, the maximum level value is $\sum_{i=1}^{m} (\max(P_i))$. As all integer numbers in these two limits (and they themselves as well) are valid level values, we get the number of levels given in the theorem.*

Nodes in a BTG for a Pareto preference are not equally distributed over the different levels of the graph. The width of such a BTG is not constant. While at the top of the BTG there is always exactly one node with a level combination of $(0, \dots, 0)$, we already have a number of nodes in level 1. There are m different nodes containing 0 as a WOP level value $m-1$ times and 1 exactly one time. Beginning at the bottom node and moving up one level, there are also m different nodes with one of the values for $level_{P_i}$ being smaller than its $\max(P_i)$. The BTG is symmetric to its middle axis:

Lemma 6. *A BTG for a preference $P := \otimes(P_1, \dots, P_m)$ is symmetric with respect to its middle axis. That means, in a level x of the BTG, there are exactly as many nodes as in level $\max(P) - x$.*

Proof. *Consider a node $a = (a_1, \dots, a_m)$ in level v. Without any problems, we can find a node $\overline{a} = (\overline{a_1}, \dots, \overline{a_m})$ with $\overline{a_i} := \max(P_i) - a_i$. So the level of $\overline{a}$ is given by:*

$$level_P(\overline{a}) := \sum_{i=1}^{m} (\max(P_i) - a_i) = \sum_{i=1}^{m} \max(P_i) - \sum_{i=1}^{m} a_i = \max(P) - level_P(a)$$

As we can find a node $\overline{a}$ for each node a in level x, $\overline{a}$'s level must have as many nodes as level $\max(P) - v$. This is also known as the duality principle *in lattices.*

In the next theorem, we will see how to find the width of a given BTG for each of its levels. The method is based on products of lattices as described in [22] (and already mentioned in Section 4.3.1).

Theorem 14. *Consider a Pareto preferences $P_K = \otimes(P_{K_1}, \dots, P_{K_m})$ with m WOPs and another Pareto preference $P_L = \otimes(P_{L_1}, \dots, P_{L_n})$ with n WOPs. We construct a preference $P = \otimes(P_{K_1}, \dots, P_{K_m}, P_{L_1}, \dots, P_{L_n})$ by integrating all WOPs contained in P_K and P_L.*
The width of the BTG for P in level v can be found using the following formula:

$$width(P, v) = \sum_{i=0}^{v} (width(P_K, i) * width(P_L, v - i)$$

Using this formula, we can split every Pareto preference down to its WOPs to compute the width of the BTG for any level.

Proof. *As we have seen in Theorem 9, a BTG is a lattice. In our case, the BTG for P is built as the product of the BTGs for P_K and P_L (with BTGs called K and L).*
For level v, the algorithm combines the nodes of K in level 0 with those of L in level v, those of K in level 1 with those of L in level $v-1$ and so on. So all possible combinations of nodes of K and L having v as combined sum of level values are counted. These are all nodes in level v of the BTG for P.

In some cases, there are shortcuts for this calculation. They are shown in Lemma 7.

Lemma 7. *For $m = 2$ (and therefore $P = \otimes(P_1, P_2)$) we can find a shortcut for the computation of the BTG width for a given level v to reduce computational effort. There are three different cases:*

1. $v \leq \min(\max(P_1), \max(P_2))$: $width(BTG_P, v) = v + 1$
2. $\min(\max(P_1), \max(P_2)) < v \leq \frac{1}{2}(\max(P_1) + \max(P_2))$:
$$width(BTG_P, v) = \min(\max(P_1), \max(P_2)) + 1$$
3. $\frac{1}{2}(\max(P_1) + \max(P_2)) < v$:
$$width(BTG_P, v) = width(BTG_P, height(BTG_P - v)$$

Proof. *We will prove the different parts of the lemma one by one.*

1. *Consider a BTG node $x = (x_1, x_2)$ in level $v \leq \min(\max(P_1), \max(P_2))$. So we get $v + 1$ possible level combinations for x: $(0, v)$, $(1, v - 1)$, ..., $(v, 0)$.*
2. *Consider $\max(P_1) < \max(P_2)$ and a node $x = (x_1, x_2)$ in the BTG so that $\max(P_1) < x_1 + x_2 \leq \max(P_2)$. Then, we have $\max(P_1)$ possible combinations in a given level $v \geq \max(P_1)$: $(0, v)$, $(1, v - 1)$, ..., $(\max(P_1), v - \max(P_1))$.*
3. *This follows from the symmetry of BTGs shown in Lemma 6 and holds for BTGs for Pareto preferences with any number of input WOPs.*

With Theorem 14 in mind, we can analyze the maximum width of a BTG. This maximum width is found in the middle of the BTG (see the BTG structure in Figure 4.3. Lemma 8 will show how to retrieve the correct value.

Lemma 8. *The maximum width of the BTG for a Pareto preference P is computed by the following function:*

$$width(P) = width(P, \lfloor height(P)/2 \rfloor)$$

The computation of the maximum width for a BTG can be seen in the following example:

Example 14. *Ringo has a Pareto preference consisting of 3 WOPs. The WOPs have the following maximum level values:*

$$\max(P_1) = 2, \max(P_2) = 4, \max(P_3) = 10$$

The number of nodes for the BTG for this preference is $(2+1)*(4+1)*(10+1) = 165$. *The BTG itself can be seen in Figure 4.5 (graphic created with the* BTG Visualizer *of [54]). Its height is* $2+4+10+1 = 17$. *So the BTG has the biggest width at level 8 (among others). The width for level 8 can now be computed by splitting the preference in two parts:* $P_K = P_1$ *(whose BTG always has a width of 1) and* $P_L = \otimes(P_2, P_3)$. *We find the following widths for the BTG for* P_K *and* P_L *using the shortcut formula:*

level	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
$width(P_K, level)$	*1*	*1*	*1*	*0*	*0*	*0*	*0*	*0*	*0*
$width(P_L, level)$	*1*	*2*	*3*	*4*	*5*	*5*	*5*	*5*	*5*

Please note that for level values bigger than 2 the BTG for P_K *has no nodes and therefore a width of 0. To find the width of* BTG_P, *we will summarize the widths for* BTG_{P_K} *of each level i with those of* BTG_{P_L} *for each level* $8-i$.

$$1*5+1*5+1*5+0*5+0*5+0*4+0*3+0*2+0*1 = 15$$

Complete measures for different BTGs can be explored in Example 15. It also shows that width and height of a BTG hardly interdepend: there is a BTG with many nodes that is wider than high and another BTG, having less nodes than the latter and being much higher.

Example 15. *Consider three Pareto preferences* P_A, P_B, *and* P_C *with the following maximum level values of the respectively contained WOPs:*

- P_A: *2, 2, 1*
- P_B: *2, 2, 100*
- P_C: *20, 20, 20*

The BTGs for the preferences can be found in Figure 4.6. Due to the number of nodes, only the shapes of BTG_{P_B} *and* BTG_{P_C} *can be seen. Displaying the nodes of* BTG_{P_C} *in the same size as those of* BTG_{P_A} *would lead to* BTG_{P_C} *being over three meters wide. The measures of the BTGs are:*

preference	*nodes*	*edges*	*height*	*width*
P_A	*18*	*33*	*6*	*3*
P_B	*909*	*2,112*	*105*	*9*
P_C	*9,261*	*26,460*	*61*	*331*

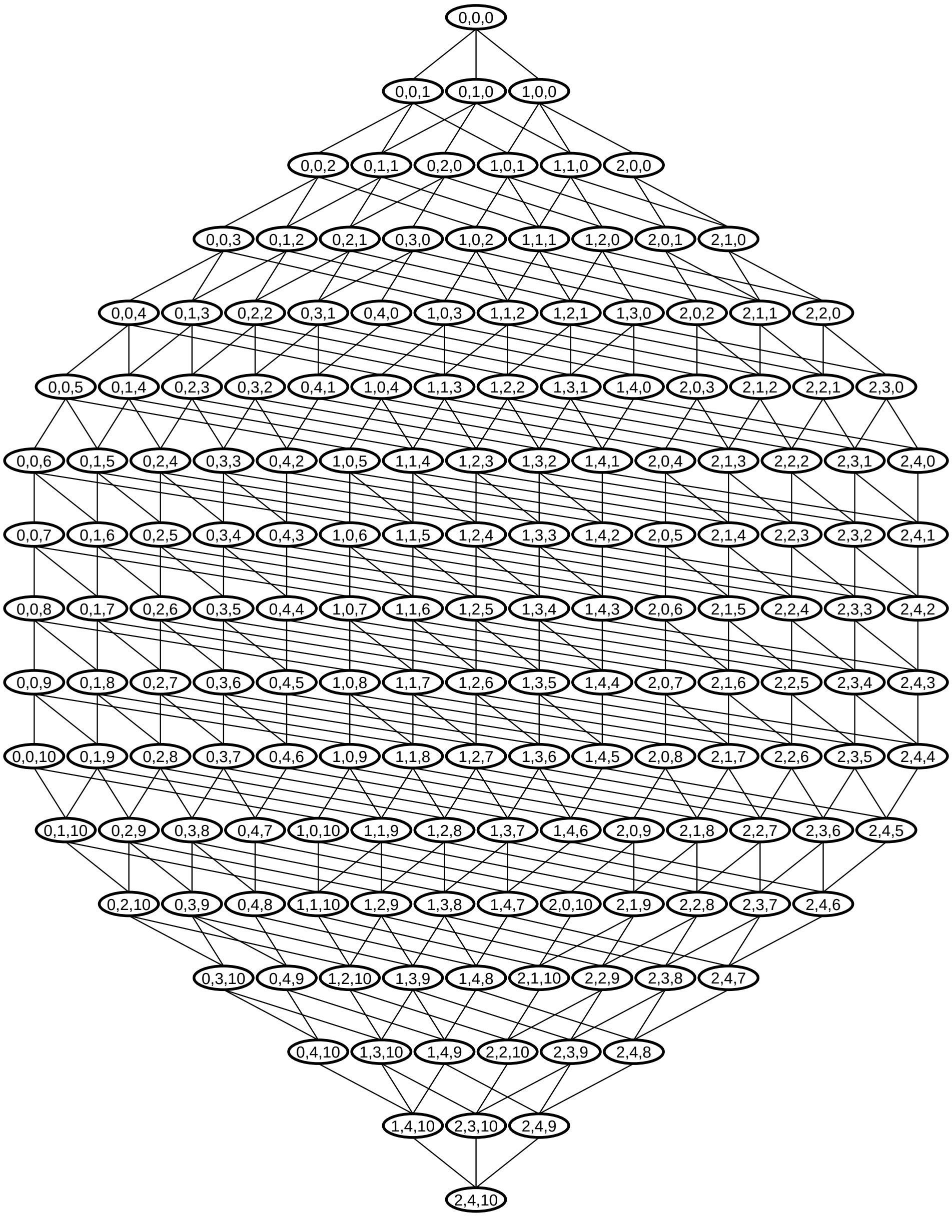

Figure 4.5: Sample BTG with maximum level values of 2, 4, and 10

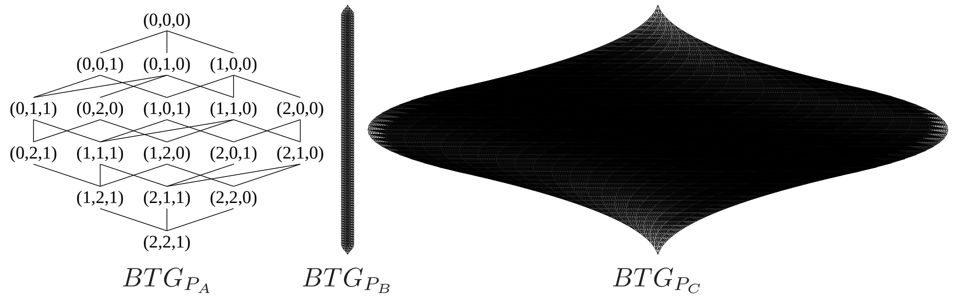

Figure 4.6: BTGs for Example 15

BTG_{P_A} shows a relatively good balance between height an width. In general, the graph is quite small. The other two BTGs differ from it considerably not only by the node count but also by their shapes. As we can see, BTG_{P_B} is much higher than BTG_{P_C} although the latter has ten times more nodes. When we look at the widths, we see a different image with a very broad BTG_{P_C} and a very thin BTG_{P_B}. The different shapes can also be nicely seen in Figure 4.6.

4.4 Unique Node IDs

We will now introduce a method to provide each of the nodes in a BTG with a unique integer number: the *node ID*. A unique node ID that is easy to generate is crucial for the algorithms in Section 6. Before we are able to find such IDs, we have to define edge weights in our BTG.

4.4.1 Edge Weights

Each edge in a BTG is expressing direct domination of one node over another as a result of domination in exactly one dimension of the level value combination. We classify all edges in a BTG according to the preference they state domination for.

Definition 23. *Edge Weight*
For the BTG of a Pareto preference $P := \otimes(P_1, \ldots, P_m)$, we define the weight of an edge expressing domination with respect to any $P_i \in P$ as follows:

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

□

So each edge in a BTG has one of m different edge weights. Please note that for WOPs for BTGs, all edges have a weight of 1. Example 16 shows edge weights for the BTG of Figure 4.4.

Example 16. *In Example 13, we have seen a Pareto preference $P = \otimes(P_1, P_2)$ with maximum level values 3 and 2. Therefore, for the edges of the BTG in Figure 4.4, we get the weights:*

$$weight(P_1) = 3, weight(P_2) = 1$$

The edges are classified into those with a weight of 3 (e.g. from $(0,0)$ to $(1,0)$ or from $(1,1)$ to $(2,1)$) specifying domination wrt P_1 and those with a weight of 1, specifying domination wrt P_2 (e.g. from $(0,0)$ to $(0,1)$).

4.4.2 From Edge Weights to Unique Node IDs

With the edge weights of Definition 23, we are now able to provide each node in a BTG with a unique ID. This unique *node ID* is generated as follows:

Theorem 15. *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 properties hold:

1. *The ID is unique for every node in the BTG.*
2. *Every value in the set $\{0, 1, 2, \ldots, |BTG| - 1\}$ is the ID of some node in the BTG.*

Proof. *We will prove the two parts of the theorem one after the other:*

1. *Consider the two nodes*

$$\begin{aligned} a &= (a_1, \ldots, a_{i-1}, a_i, 0, \ldots, 0) \text{ and} \\ b &= (a_1, \ldots, a_{i-1}, a_i - 1, \max(P_{i+1}), \ldots, \max(P_m)). \end{aligned}$$

Thus, we find $ID(a) - ID(b) =$

$$weight(P_i) - \sum_{j=i+1}^{m} (weight(P_j) * \max(P_j)).$$

Now we will have a look at the computation of edge weights:

$$\begin{aligned}
weight(P_m) &= 1 \\
weight(P_{m-1}) &= weight(P_m) * (\max(P_m) + 1) \\
&= weight(P_m) * \max(P_m) + weight(P_m) \\
weight(P_{m-2}) &= weight(P_{m-1}) * (\max(P_{m-1}) + 1) \\
&= weight(P_{m-1}) * \max(P_{m-1}) + weight(P_{m-1}) \\
&= \sum_{j=m-1}^{m} (weight(P_j) * \max(P_j)) + 1 \\
&\vdots \\
weight(P_i) &= \sum_{j=i+1}^{m} (weight(P_j) * \max(P_j)) + 1
\end{aligned}$$

We notice that $ID(a) = ID(b) + 1$. Increasing some level value at index i by 1 increases the ID more than increasing all level values for higher indexes to their maximum. All nodes with equal level values for $a_1, \ldots, a_{i-1}$ and values smaller than a_i at index i have smaller IDs than a.

Considering two nodes, the one with a greater level value at a lower index will always have the greater ID. Thus, there cannot be two nodes with the same ID as by definition two nodes always have different level combinations.

2. *The lowest ID of a node clearly is the one of the top node $t = (0, \ldots, 0)$. It is always 0. The greatest value for the ID of a node of the BTG is achieved by maximizing all values in the ID computation for node b:*

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

This node b is the bottom node of the BTG. Replacing the term $weight(P_i)$ by its definition, we get:

$$ID(b) = \left(\prod_{i=1}^{m} (\max(P_i) + 1) \right) - 1$$

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

Apart from finding the node ID to some level value combination, it is also possible to find the level value combination to a given node ID. The following lemma will show how to invert the node ID generation process.

Lemma 9. *ID is a bijective mapping. $ID^{-1}(n) = (a_1, a_2, \ldots, a_m)$ 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):*

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

In a more compact notation, we can calculate each a_i's value as follows:

$$a_i = \left\lfloor \frac{ID(a) - \sum_{j=1}^{i-1} a_j * weight(P_j)}{weight(P_i)} \right\rfloor$$

Proof. *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.*

As all edge weights for WOPs are 1, the ID function for them is trivial, just as well as ID^{-1}. The ID of a node is identical to the single level value its equivalence class is identified by. But now, we will see some more interesting node IDs for the BTG of a Pareto preference in the next Example:

Example 17. *We have computed the edge weights for the BTG for Example 13 in 16 ($weight(P_1) = 3$, $weight(P_2) = 1$. Using these weights, we can compute unique IDs for the nodes of the BTG, e.g. for the following:*

$$\begin{aligned} ID(0,2) &= 3 * 0 + 1 * 2 = 2 \\ ID(1,1) &= 3 * 1 + 1 * 1 = 4 \end{aligned}$$

Using the results of Lemma 9, we can find the level combination of any node with a given ID. We will demonstrate this for the node with an ID of 7:

$$\begin{aligned} a_1 &= \left\lfloor \frac{7-0}{3} \right\rfloor = \left\lfloor \frac{7}{3} \right\rfloor = 2 \\ a_2 &= \left\lfloor \frac{7 - a_1 * 3}{1} \right\rfloor = \lfloor 7-6 \rfloor = 1 \end{aligned}$$

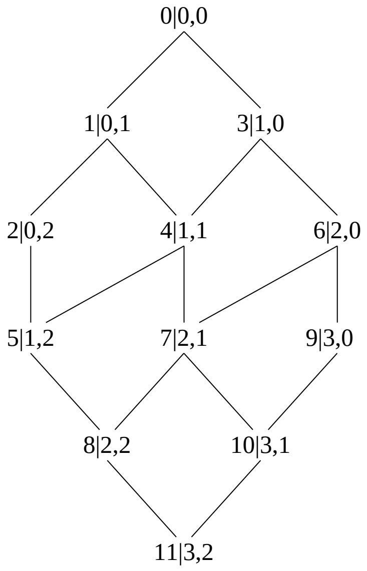

Figure 4.7: Sample BTG with node IDs for maximum level values $(3, 2)$

The BTG for this preference showing level combinations for the nodes and their IDs can be found in Figure 4.7. The nodes are labeled with the ID and their level combinations after the vertical line.

4.5 Integration of Strict Partial Orders

We have seen how to combine WOPs in a Pareto preference and thereby create strict partial orders. By abstracting from the preferences we started with, we arrived at BTGs in which integer level values are used to determine domination of nodes among each other. Now, we will use this abstract view to integrate general strict partial orders in our BTG. This will enable us to handle them just in the same way as we handle BTGs for Pareto preferences containing only WOPs. As a first step, we will show how to transform the orders defined by base preferences that use trivial instead of regular SV-semantics (and as a consequence know incomparable values) into BTGs as known from Section 4.3. Then, we will show how to embed all kinds of strict partial orders in BTGs – giving us the ability to treat them in the same way as Pareto preferences.

4.5.1 Base Preferences Without Regular SV-semantics

In their primary form in [37], all preferences were defined with trivial instead of regular SV-semantics of Definition 3. In general, this means that one value may be

- equal to
- better than
- worse than
- uncomparable to

another value. For categorical base preferences, all different values in the same set (i.e. a layer or a POS- or NEG-set) are *uncomparable* to each other (more detailed descriptions will be given in Theorem 16 and Theorem 17). Looking only at base preferences, this makes no semantical difference to those described in Section 3.1.1 ([38]). The BMO result holds the tuples with values from the best set found in the input. The difference occurs (and the problem arises for us) when such preferences are combined, for instance in a Pareto preference, as we will see in the following example:

Example 18. *Consider John's Pareto preference on cars from Example 9:*

$$P_{John} := \otimes(BETWEEN_5(price; 60, 80), POS/NEG(color; \{red,\ blue\}, \{purple\})$$

We will now evaluate it on the sample database of Table 3.2 without using regular SV-semantics. The level values as seen in Table 3.3 are no longer sufficient to check for domination. An equal level value in one WOP may lead to incomparability of two tuples.

The BMO result set without using regular SV-semantics holds tuple 4 as well, as for price *it is rated worse than tuples 2 and 3, but its* color *"yellow" is uncomparable to the "black" of car 2 and the "brown" of car 3.*

When, in a LAYERED preference, values of the same layer are *uncomparable*, the level value of this layer alone is not sufficient to determine domination. Two different attribute values may render two tuples *uncomparable*, despite having identical level values. We will see how to overcome this limitation for both categorical and numerical base preferences by replacing the single integer level value by two integers.

For this book, these base preferences with trivial SV-semantics are only exceptions from the regular WOPs of Section 3.1.1. Instead of giving some additional syntax to define them, we will derive them from their regular SV-semantics using counterparts as we have done in Example 18. In the syntax of *Preference SQL*, base preferences with trivial SV-semantics can be identified by the missing keyword `REGULAR` at the end of their definition.

Numerical WOPs With Trivial SV-Semantics

In the case of numerical base preferences the problem of trivial SV-semantics arises only for BETWEEN$_d$ and AROUND$_d$ preferences. With the latter being a special case of the former, it will be sufficient to analyze the problem only for BETWEEN$_d$ preferences.

Theorem 16. *Consider a preference $P := BETWEEN_d(A; low, up)$ and a preference P' derived from P by replacing regular by trivial SV-semantics.*

The level value each element of $dom(A)$ is mapped to in P is replaced by two values (l_1, l_2) in P'. A value x is represented by the integer combination (l_1, l_2) that is found as follows:

$$x \rightarrow (l_1, l_2) = \begin{cases} (level_P(x), level_P(x) - 1) & \Leftrightarrow \quad up < x \\ (level_P(x) - 1, level_P(x)) & \Leftrightarrow \quad x < low \end{cases}$$

So l_1 and l_2 can be initialized with $level_P(x)$ and then l_1 resp. l_2 is decremented iff $x < low$ resp. $up < x$.

Then P' models the same order wrt $dom(A)$ as P, but distinguishes between values lower and values higher than the interval borders.

Proof. *Consider P' as a preference derived from a BETWEEN$_d$ preference P by replacing regular by trivial SV-semantics, a value v with level combination (v_1, v_2), and a value w with level combination (w_1, w_2). The following cases may occur:*

- $level_P(v) = level_P(w) + 1$:
  - $v < low \quad \wedge \quad w < low \quad \Rightarrow \quad (w_1 = v_1 + 1) \quad \wedge \quad (w_2 = v_2 + 1)$
  - $v < low \quad \wedge \quad up < w \quad \Rightarrow \quad (w_1 = v_1 + 2) \quad \wedge \quad (w_2 = v_2)$

- $level_P(v) = level_P(w)$:
  - $v < low \wedge w < low \Rightarrow (v_1, v_2) = (w_1, w_2) \Rightarrow v \cong_{P'} w$
  - $v < low \wedge up < w$

$$\begin{aligned} \Rightarrow \quad (v_1, v_2) &= (level_P(v), level_P(v) + 1), \\ (w_1, w_2) &= (level_P(v) + 1, level_P(v)) \\ \Rightarrow \quad v &\sim_{P'} w \end{aligned}$$

All other possible cases can be derived from those above. So the level combination assigned to domain values fulfills the specification of the preference.

For extremal preferences like HIGHEST$_d$ (or LOWEST$_d$), the problem of incomparable values does not arise. The supremum (or infimum) denotes the best available value. Deviation from the optimum value is only possible in one direction – so no two values can differ by more than d when they have the same level.

id	1	2	3	4	5	6
price	50	70	75	55	45	45
$level_{P_{Paul}}(price)$	0	4	5	1	1	1
(l_1, l_2)	(0, 0)	(4, 3)	(5, 4)	(1, 0)	(0, 1)	(0, 1)

Table 4.1: Level combinations for non-WOP

Example 19. *Paul wants to rent a car for approximately €50 per day and does not care for differences of up to €5. So formally we get:*

$$P_{Paul} := AROUND_5(price; 50)$$

We derive a preference with trivial SV-semantics from P_{Paul} and create level pair mappings for the sample database in Table 3.2. The results can be found in Table 4.1. A perfect value of 50 is mapped to $(0, 0)$. 45 and 55 (with $level_P(45) = level_P(55) = 1$ are mapped to incomparable value combinations.

From a technical point of view, two WOPs are connected and used to model the strict partial order defined by the numerical base preference with trivial SV-semantics when such a preference is evaluated. A "virtual" Pareto preference is constructed by the numerical base preference. The size of the BTG for such a preference is given in the following lemma:

Lemma 10. *The size of the BTG P' defined by a $BETWEEN_d$ preference P by replacing regular by trivial SV-semantics is given by:*

$$(\max(level_P(\min(dom(A))), level_P(\max(dom(A)))) + 1)^2 = (\max(P) + 1)^2$$

Proof. *Looking at the computation of level combinations for values to be rated, the BTG that is constructed is identical to one for a Pareto preference containing two WOPs with maximum level values of* $\max(P) + 1$.

Not all nodes of such a BTG represent valid inputs. As we will see in Lemma 11, the number of useable nodes is growing linear wrt the maximum level value of the corresponding preference with regular SV-semantics:

Lemma 11. *Consider a preference P' which is defined as a $BETWEEN_d$ preference P with trivial instead of regular SV-semantics. The number of used nodes (i.e. the number of nodes that can be matched by values evaluated by P') in the BTG for P' is given by* $2 * \max(P) + 1$.

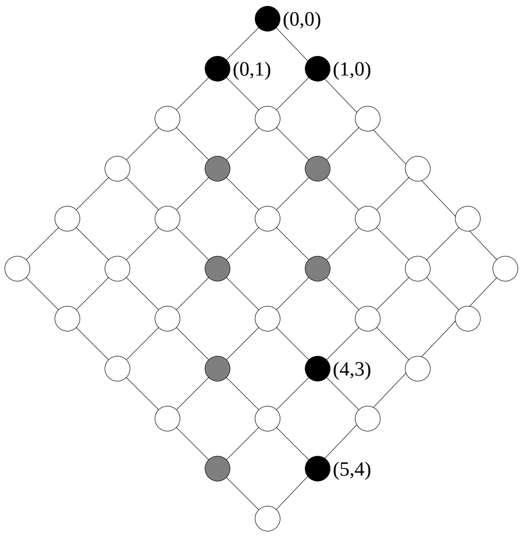

Figure 4.8: BTG for an AROUND_d preference with trivial SV-semantics

Proof. *The node* $(0,0)$ *is used for perfect matches. Other nodes used have level combinations of* $(x, x+1)$ *or* $(x+1, x)$*, as stated in Theorem 16. The minimum value for x is one, the maximum is* $\max(P)$*, leading to* $2 * \max(P) + 1$ *values in use.*

Example 20. *For* P_{Paul} *of Example 19 and the database of Table 3.2, we find* $\max(P) = 5$*. The BTG constructed by* P'_{Paul} *(i.e.* P_{Paul} *with trivial instead of regular SV-semantics) is shown in Figure 4.8.*

The black nodes have tuples belonging to them (given Table 3.2 as input), the gray nodes represent valid integers for l_1 *and* l_2*, while the white nodes are unused dummy nodes given by the graph structure. Just as Lemma 10 and Lemma 11 indicate, the BTG has* $(5+1)^2 = 36$ *nodes of which* $2 * 5 + 1 = 11$ *might be used.*

Categorical WOPs With Trivial SV-Semantics

Incomparability of values is straightforward for categorical base preferences as well. Considering some preference $LAYERED_m(A; L_1, \ldots, L_{m+1})$, all values in one of the L_i are incomparable (although having the very same level value) to each other (as mentioned at the beginning of Section 4.5.1).

In Theorem 17, we will see how to map each value of the domain of a LAYERED preference to a level combination to deal with these requirements.

Theorem 17. *Consider a preference $P := LAYERED_m(A; L_1, \ldots, L_{m+1})$ and a preference P' derived from P by replacing regular by trivial SV-semantics. Each value in dom(A) is mapped to a pair of integer level values.*
The elements of the L_i are labeled with indexes: $L_i := \{l_{i,1}, l_{i,2}, \ldots, l_{i,|L_i|}\}$. Every element of L_i has to get a unique index value. Then, the level combination for each $l_{i,j}$ can be found with the following formula:

$$l_{i,j} \to \left(\left| \bigcup_{x=1}^{i-1} L_x \right| - i + j, \left| \bigcup_{x=1}^{i} L_x \right| + 1 - (i+j) + |\{x \mid x \leq i \wedge |L_x| = 1 \wedge |L_{x-1}| = 1\}| \right)$$

Proof. *Consider a preference $P' = (A, <_{P'})$ with trivial SV-semantics derived from a LAYERED preference P (using regular SV-semantics) and three categorical values $l_{i,j}, l_{i,k}, l_{i+1,q} \in dom(A)$ with $j < k$. The level combination a value $l_{x,y}$ is mapped to is $(l_{x,y}[0], l_{x,y}[1])$. We have to prove that P' constructs the same order as P on elements of different layers and renders elements of the same layer indifferent. For readability, we will abbreviate $\left|\bigcup_{x=1}^{i-1} L_x\right|$ with* s *and $|\{x \mid x \leq i \wedge |L_x| = 1 \wedge |L_{x-1}| = 1\}|$ with* t(i).

- $l_{i,j} \sim_{P'} l_{i,k}$:
  - $$\begin{aligned} l_{i,j}[0] - l_{i,k}[0] &= (s - i + j) - (s - i + k) \\ &= j - k \end{aligned}$$
    $\Rightarrow l_{i,j}[0] < l_{i,k}[0]$
  - $$\begin{aligned} l_{i,j}[1] - l_{i,k}[1] &= (s + |L_i| + 1 - (i+j) + t(i)) - \\ &\quad\ (s + |L_i| + 1 - (i+k) + t(i)) \\ &= -j + k \end{aligned}$$
    $\Rightarrow l_{i,j}[0] > l_{i,k}[0]$

  With $l_{i,j}[0] < l_{i,k}[0] \wedge l_{i,j}[0] > l_{i,k}[0]$ it follows that $l_{i,j} \sim_{P'} l_{i,k}$.

- $l_{i+1,q} <_{P'} l_{i,j}$:
  - $$\begin{aligned} l_{i,j}[0] \leq l_{i+1,q}[0] \Leftrightarrow \quad s - i + j &\leq s + |L_i| - (i+1) + q \\ j &\leq |L_i| - 1 + q \end{aligned}$$
    This always holds as $j \leq |L_i| \wedge (-1+q) \geq 0$.
    $$\begin{aligned} \Rightarrow \quad & j = |L_i| - 1 - q \Leftrightarrow j = |L_i| \wedge q = 1 \\ & j < |L_i| - 1 - q \Leftrightarrow j < |L_i| \vee q > 1 \end{aligned}$$

- $l_{i,j}[1] \leq l_{i+1,q}[1] \Leftrightarrow$

$$\begin{array}{rcl} s + |L_i| + 1 - (i+j) + t(i) & \leq & s + |L_i| + |L_{i+1}| + 1 - \\ & & (i+1+q) + t(i+1) \\ -j + t(i) & \leq & |L_{i+1}| - 1 - q + t(i+1) \end{array}$$

  - *case 1:* $|L_i| = 1 \wedge |L_{i+1} = 1 \Leftrightarrow t(i+1) = t(i) + 1 \Rightarrow j = 1 \wedge q = 1$

$$\begin{array}{rcl} \Rightarrow -j + t(i) & \leq & |L_{i+1}| - 1 - q + t(i) + 1 \\ -j & \leq & |L_{i+1}| - q \\ -1 & \leq & 1 - 1 \end{array}$$

  - *case 2:* $|L_i| > 1 \vee |L_{i+1} > 1 \Leftrightarrow t(i+1) = t(i)$

$$\begin{array}{rcl} \Rightarrow -j + t(i) & \leq & |L_{i+1}| - 1 - q + t(i) \\ -j & \leq & |L_{i+1}| - 1 - q \\ j & \geq & 1 + q - |L_{i+1}| \qquad (*) \end{array}$$

    For $j = 1 \wedge q = |L_{i+1}|$*, both sides are equal.*
    As $(j \geq 1) \wedge (q - |L_{i+1}| \leq 0)$*, the inequation*
    holds in all other cases, too.

To sum up the preceding points, we can distinguish the following cases, showing that $l_{i+1,q} <_{P'} l_{i,j}$ *always holds:*

$$\begin{array}{ccccccc} j = 1 & \wedge & q = |L_{i+1}| & \Rightarrow & l_{i,j}[0] < l_{i+1,q}[0] & \wedge & l_{i,j}[1] = l_{i+1,q}[1] \\ 1 < j < |L_i| & \wedge & 1 < q < |L_{i+1}| & \Rightarrow & l_{i,j}[0] < l_{i+1,q}[0] & \wedge & l_{i,j}[1] < l_{i+1,q}[1] \\ j = |L_i| & \wedge & q = 1 & \Rightarrow & l_{i,j}[0] = l_{i+1,q}[0] & \wedge & l_{i,j}[1] < l_{i+1,q}[1] \end{array}$$

As we can see, all elements of the same layer are indifferent and better than all elements of (wrt their indexes) higher layers.

Please note that there are different ways of mapping domain values to level value combinations. Depending on the structure of the layers, easier mapping rules are possible as well. We focused on a generic approach which is capable of handling all possible LAYERED preferences.

Example 21. *John's color preference is well-known from Example 5, Example 12 and others:*

$$P_{col} := POS/NEG(color, \{red,\ blue\}; \{purple\})$$

Derived from it is a POS/NEG preference P'_{col} *with the same sets but trivial instead of regular SV-semantics. The same preference can be expressed by a LAYERED preference as well (as POS/NEG is a sub-constructor of LAYERED, cp. Section 3.1.1). The domain of the attribute, color shall be* $\{yellow,\ red,\ purple,\ blue,\ brown,\ black\}$*:*

$$P_{col_2} := LAYERED_2(color, \{red,\ blue\}; others; \{purple\})$$

So others $= \{yellow,\ brown,\ black\}$*. Figure 4.9 shows the BTG for a LAYERED preference* P'_{col_2} *with the same sets, but trivial SV-semantics. Nodes with invalid level*

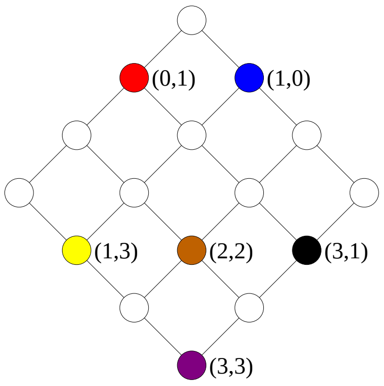

Figure 4.9: BTG for a LAYERED preference with trivial SV-semantics

combinations are white, nodes with other colors are labeled with the level combination the color is assigned to. The BTG for the original P_{col} can be found in Figure 4.2.

4.5.2 General Strict Partial Orders

Although it is a base preference, we have not yet mentioned the EXPLICIT preference introduced in [37] in Section 3.1.1. We will discuss it in this section. An EXPLICIT preference allows a user to express most of the possible strict partial order over a domain. In contrast to the base preferences of Section 3.1.1, it does not construct a *weak order*.

Definition 24. *EXPLICIT preference*
Consider a preference $P := (A, <_P)$ and set of pairs of values $E \subset dom(A) \times dom(A)$. With each of these pairs (a_1, a_2) stating domination of some a_2 over some other value a_1 and the order being transitive, P is an EXPLICIT preference.
The constructor for an EXPLICIT preference is given by:

$$EXPLICIT(A, \{(a_1, a_2), \ldots, (a_x, a_y)\})$$

Unmentioned values are considered worse than any value in some element of E. □

To be able to evaluate EXPLICIT preferences with algorithms of Section 6, we have to embed the strict partial order defined by it into a distributive lattice. The problem we are facing here is the general embedding of Hasse diagrams for arbitrary strict partial orders into distributive lattices.

For this embedding, we start with a Hasse diagram representing the strict partial order of the EXPLICIT preference. To each node, we will assign a so-called *signature*. This signature is a combination of integer values. We call the signature of a node a combination of integer values used to identify it. When the construction of the distributed lattice structure is complete, the signature of a node is identical to the level combination of the BTG node it is mapped to. The mapping is done in the following four steps:

1. Identify non-dominated nodes and generate an unlabeled virtual top node $\triangle$ for them. Add edges from $\triangle$ to the non-dominated nodes. Also add a virtual bottom node $\bigtriangledown$ that is dominated by all the nodes not dominating other values in the graph.
2. Do a depth-first search beginning at the top node. The algorithm used for the depth-first search is irrelevant, but the following issues have to be kept in mind:
    - Keep a counter. Each time the search finds a dead end, increase the counter by one.
    - Annotate each edge during the search with the counter value.
    - Do no follow annotated edges.
3. Do a breadth-first search on the graph, starting at the top node again. There are two possibilities in each node n:
    - n is directly dominated by exactly one node and reached by an edge with an annotated value of v. The signature value of n is the signature value of its dominating node increased by one at position v.
    - n is directly dominated by a number of nodes $d_1, d_2, \ldots, d_x$. The signature of n at each position i is given by the maximum value of the d_i at the same position.
4. Check the maximum values in use at each position of the node signature and remove those positions with a maximum value of zero.

The mapping of domain values mentioned in the original Hasse diagram to their level combination can be done using a hash table. As the values are a given subset of the (possibly the whole, finite) domain, a minimal perfect hash function can be found as described in [18, 20, 29] to make access as fast and memory requirements as low as possible. For an EXPLICIT preference, all domain values not mentioned in any better-than pairs given in its constructor are mapped to the virtual bottom node.

We have mentioned that EXPLICIT preferences are not capable of modeling any strict partial order. Such an order can be found in the following example:

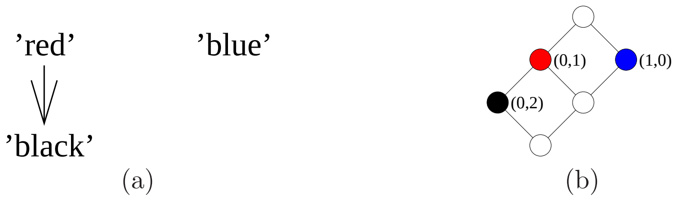

Figure 4.10: Hasse diagram for a strict partial order with corresponding BTG

Example 22. *Paul tells a car vendor about his preferences on colors:*

> *I like* `red` *more than* `black`*. And I like* `blue`*.*

Figure 4.10(a) shows the Hasse diagram of this preference. As "black" is a preferred value but not better than any other given value, we cannot model this preference with the EXPLICIT operator. Figure 4.10(b) shows the BTG with the order embedded into it. The colored nodes show the color they represent and are annotated with their level combinations.

Our algorithm has no problem in integrating any kind of strict partial order. Isolated nodes in a Hasse diagram belong to the set of top nodes that is linked to the virtual single top node created in step one of the algorithm. Example 23 will show the algorithm in action on a larger graph.

Example 23. *A Hasse diagram for a strict partial order is given in Figure 4.11(a). We add top and bottom nodes in Figure 4.11(b). Then, the depth-first marking of the edges begins. Figure 4.11(c) shows the moment when the first edges has been marked with "3". In Figure 4.11(d), all edges are marked.*

Then, we determine the node signatures, starting at $\triangle$*. As the highest number assigned to an edge is 4, the node signatures have four integer values. The top node has a signature of* $(0,0,0,0)$*. For a, we get the signature* $(1,0,0,0)$*, as it is dominated by the top node by an edge marked with* 1 *and so the signature value of* $\triangle$ *is increased by one at position* 1*. The next table shows signatures for all nodes after step 3 of the algorithm:*

node	a	b	c	d	e	f	g
signature	$(1,0,0,0)$	$(2,0,0,0)$	$(3,0,0,0)$	$(0,0,1,0)$	$(0,0,2,0)$	$(2,0,2,0)$	$(0,0,0,1)$

We see that the maximum value at position 2 is 0. It is therefore removed. So we keep the maximum signature values 3,2, and 1. The resulting BTG can be seen in Figure

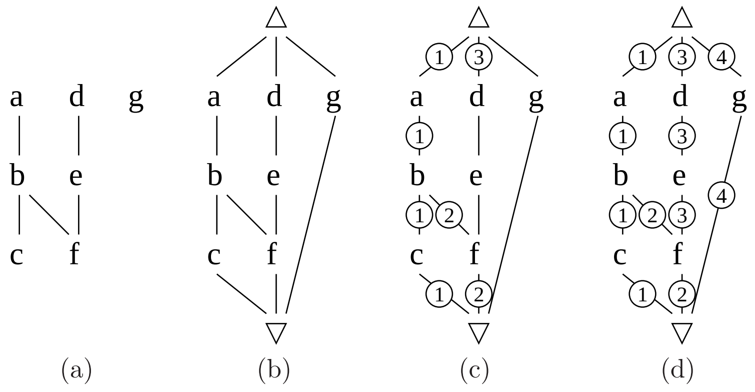

Figure 4.11: Embedding a strict partial order in a BTG

4.12 with part (a) showing all signature values and part (b) replacing those connected to a value of the original order.

Please note that the algorithm for embedding a strict partial order in a distributive lattice like a BTG is not neccessarily producing minimal BTGs, as we will see in Example 24. How to deterministically find the smallest BTG has to be further investigated in later works.

Example 24. *We will have another look at the strict partial order in Figure 4.11(a). Some depth-first search algorithm yields the edge annotation of Figure 4.13(b). Then, the maximum integer values for the embedding are 2, 1, 2, 1. With those maximum values, the BTG that is constructed has 36 nodes; it is 50% bigger than the one shown in Example 23.*

Another mapping of partial orders to distributive lattices has been presented in [53]. Using only two integer numbers to represent each value in a partial order, a BTG constructed according to the maximum signature levels tends to be smaller than when using our algorithm. But not all partial orders can be expressed when such a mapping is used. There are partial orders that cannot be expressed by only two integer values as we will see in Lemma 12.

Lemma 12. *We have some values $a, b, \dots, f$ and a partial order on them defined by $d < a$, $d < b$, $e < a$, $e < c$, $f < b$, and $f < c$. The Hasse diagram for this order is given in 4.14(a). Then a mapping of each node to a BTG node defined by two integer values is not able to preserve the original partial order.*

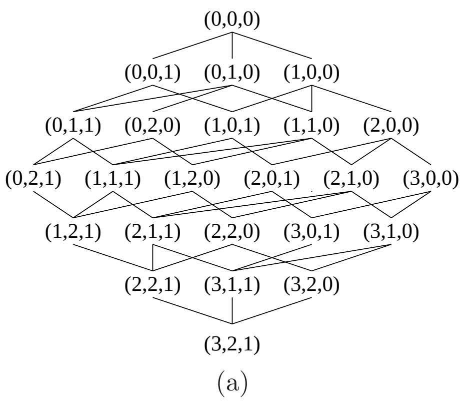

Figure 4.12: BTG for the preference expressed in Figure 4.11(a)

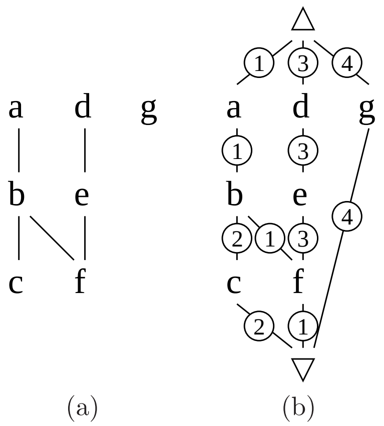

Figure 4.13: Embedding a strict partial order in a BTG (2)

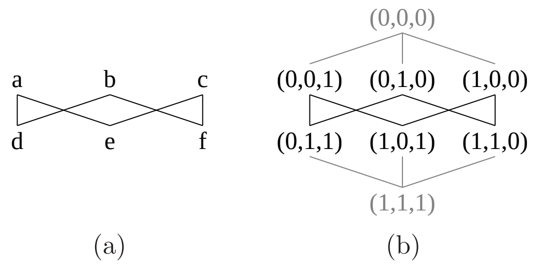

Figure 4.14: Strict partial order needing more than two integer values

Proof. *$a = (a_1, a_2)$, $b = (b_1, b_2)$, and $c = (c_1, c_2)$ are indifferent. We will assume $a_1 < b_1 < c_1$ and $a_2 > b_2 > c_2$. From the strict partial order we know:*

$$e < a \wedge e < c \Rightarrow a_1 \leq e_1 \wedge a_2 \leq e_2 \wedge c_1 \leq e_1 \wedge c_2 \leq e_2$$

Hence, the combination of smallest possible values for e is $(e_1, e_2) = (c_1, a_2)$. So e always is dominated by b, too, which is a contradiction to the original partial ordered set. There is no valid mapping with two integer level values.

Using the Hasse diagram structure of Figure 4.14(a) as a pattern, similar proofs for more than two values in a signature can be found. The number of integers needed to construct a BTG to embed a partial order depends on the partial order and does not have an upper border.

In Example 25, a possible embedding of the given partial order into a BTG using three integer values can be seen.

Example 25. *Consider the partial order in Figure 4.14(a). A possible embedding to a distributed lattice is shown in the BTG with maximum level values of $(1, 1, 1)$ in Figure 4.14(b). The mappings are:*

node	*a*	*b*	*c*	*d*	*e*	*f*
signature	$(0, 0, 1)$	$(0, 1, 0)$	$(1, 0, 0)$	$(0, 1, 1)$	$(1, 0, 1)$	$(1, 1, 0)$

4.5.3 Combining WOPs and General Strict Partial Orders

As we have seen, a single integer level value is not enough to express the semantics of base preferences with trivial SV-semantics or strict partial orders. We have overcome this limitation by using two or more integer values. Now we have to integrate these preferences in the standard Pareto preferences introduced in Definition 15.

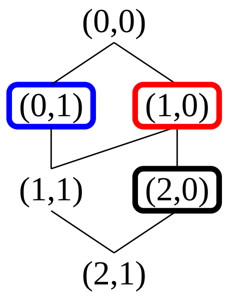

Figure 4.15: BTG for a strict partial order

Theorem 18. *Consider a strict partial order S embedded into a BTG G_S and a Pareto preference P and the corresponding graph BTG_P.*

The order constructed by the combination of P and S is visualized by the product of G_S and BTG_P.

Proof. *Both G_S and BTG_P are lattices. According to [22], the product of them yields a lattice with a combined order of both input lattices.*

The BTG for such a combination surely holds unused nodes (as the BTG for the strict partial order does already), but as we have not examined the space-complexity of the embedding of a strict partial order into a BTG, we cannot make profound statements about the percentage of used nodes at all. Nevertheless the embedding can be very useful as it enables us to evaluate base preferences that are strict partial orders just like Pareto preferences and Pareto preferences consisting of strict partial orders just as if as if they only used standard WOPs as input preferences. Example 26 defines a BTG that is the result of the combination of a WOP and a strict partial order.

Example 26. *Remember the strict partial order on colors of Example 22. The BTG this order is embedded in is shown in Figure 4.15, where the nodes representing nodes of the original order have a border of the respective color.*

Now we combine this order with John's price preference $BETWEEN_5(price; 60, 80)$ (cf. Example 1), with its maximum level value of 3 (when evaluated on the sample database of Table 3.2. The BTG for the combined order can be seen in Figure 4.16. Nodes representing no reachable level combination (due to the strict partial order) are printed in gray.

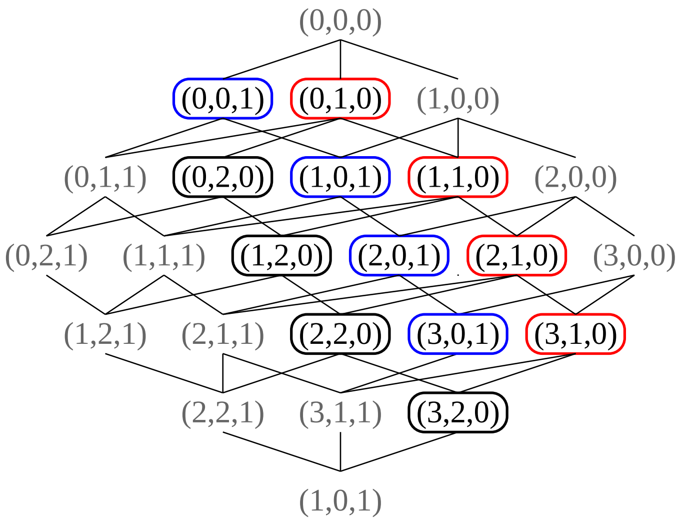

Figure 4.16: BTG for the combination of a WOP and a strict partial order

Chapter 5

Pruning and its Application

We will now use the knowledge of size and structure of BTGs we gained in the last chapter to improve the performance of algorithms for BMO set generation. After the definition and calculation of *pruning levels* for BTGs, we will see how this new principle can be used in existing algorithms.

5.1 Pruning Levels

For every node in every BTG, we can find a level of the graph wherein all nodes are dominated by the mentioned node. So, when we read a tuple t from the input relation, we can determine a level from which point onward all nodes are dominated by t. We call this t's *pruning level*. When evaluating a preference on some input relation, we speak of *pruning* the BTG there, because all tuples belonging to nodes in the *pruning level* (or higher levels) are definitely dominated by t and no longer need to be considered. As we will see later, this can lead to large performance improvements for some algorithms.

Definition 25. ***Pruning Level***
The pruning level $pl_P(n)$ *for a node n in the BTG for a preference P is the lowest level for which all nodes of the current and of higher levels are dominated by this node.*

$$\begin{array}{rcll} pl_P & : & dom(A) \rightarrow \mathbb{N}_0 & \\ pl_P(n) & := & w \;\Leftrightarrow & \nexists a \in dom(A) : (level_P(a) = w \quad \wedge \quad \neg(a <_P n)) \;\wedge \\ & & & \exists b \in dom(A) : (level_P(b) = w - 1 \quad \wedge \quad \neg(b <_P n)) \end{array}$$

The pruning level *of a value $a \in dom(A)$ is the* pruning level *of the node of a's equivalence class.* □

Just like the structure of BTGs for WOPs and for Pareto preferences, the computation of the *pruning level* of a node differs. Unsurprisingly, it is more complex for Pareto preferences. We will learn about algorithms to compute both *pruning levels* for nodes and pruning nodes for whole levels in the next sections, starting with the determination of *pruning levels* for WOPs. Unfortunately, as there is no efficient way to compute *pruning levels* for arbitrary strict partial orders as introduced in Section 4.5, we have to omit them.

5.1.1 Pruning Levels for WOPs

The *pruning level* for a BTG node in a WOP is easy to find. For a node in level x, we have to find the level y in which all nodes are dominated by our starting node. As there is exactly one node in each level, clearly $y = x + 1$. Theorem 19 outlines this more formally:

Theorem 19. *The* pruning level *for a node n in the BTG of a WOP P is given by $level_P(n) + 1$.*

Proof. *The BTG of a WOP is a chain. Each node dominates all nodes of higher levels.*

This simple formula for finding the *pruning level* can be very helpful for all kinds of algorithms. When evaluating a WOP on some input, at every point of time only the one tuple with the best level value seen so far is of interest. It will be dominated by a tuple with a lower level value, but it will dominate all tuples with higher level values. The *pruning level* mirrors the key feature of WOPs as given in Lemma 1.

5.1.2 Pruning Levels for Pareto Preferences

For Pareto preferences, pruning is not as trivial a task as it is for WOPs. A better level value does not cause domination, but is only a prerequisite as we will see in the next example:

Example 27. *Figure 5.1 shows the BTG for a Pareto preference with maximum level values of the WOPs in it of 2, 2, and 1 (which is BTG_{P_A} of Figure 4.6 with node IDs). In Figure 5.1, node 6 belongs to level 1. Although it is dominating some nodes in level 2, there are nodes indifferent to it as well (e.g. node 4). The same holds for level 3. Only from level 4 on, all nodes are dominated by node 6.*

Determining the *pruning level* of a node in a BTG for a Pareto preference is a complex task. It was shown for the first time in [52], where the following calculation rule was introduced:

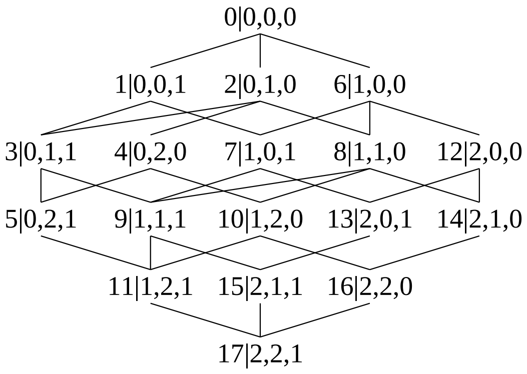

Figure 5.1: Sample BTG with node IDs

Theorem 20. *For a Pareto preference $P := \otimes(P_1, \ldots, P_m)$, a tuple $t = (t_1, \ldots, t_m)$ has the following* pruning level *with respect to P:*

$$pl_P(t) = \begin{cases} 1 & \Leftrightarrow \quad level_P(t) = 0 \\ \max(P) + 1 & \Leftrightarrow \quad level_P(t) = \max(P) \\ \sum_{j=1}^{m} max(P_j) - \min(\{max(P_i) - level_{P_i}(t_i) \mid & \\ \qquad \forall i : 1 \leq i \leq m \wedge level_{P_i}(t_i) > 0\}) & otherwise \end{cases}$$

Proof. *The first two cases are obvious and we will be able to discuss them in a few sentences. All level values of the top node are zero. According to the Pareto preference, it dominates all other nodes in the BTG. It is clear that a worst node cannot prune nodes from the BTG as it does not dominate any.*

After these trivial first two cases, we will now address the general and complicated third one. Consider a Pareto preference $P := (P_1, \ldots, P_m)$ and a tuple $a = (a_1, \ldots, a_m)$. We select a tuple $\overline{a} = (\overline{a}_1, \ldots, \overline{a}_m)$ with $a \ \sim_P \ \overline{a}$ and an overall level value as high as possible. So there must be one attribute of $\overline{a}$ with $level_{P_i}(\overline{a}_i) < level_{P_i}(a_i)$. The maximum value for this $level_{P_i}(\overline{a}_i)$ clearly is $level_{P_i}(a_i) - 1$, while the other attributes' level values should be as high as possible to maximize the overall level value of $\overline{a}$. This means the vector of level values for $\overline{a}$ is one of the following:

$$\begin{array}{ccc} [\ \overline{a}^{(1)}\ ] & := & (l_{P_1}(a_1) - 1, \max(P_2), \max(P_3), \ldots, \max(P_m)) \\ [\ \overline{a}^{(2)}\ ] & := & (\max(P_1), l_{P_2}(a_2) - 1, \max(P_3), \ldots, \max(P_m)) \\ \vdots & \vdots & \vdots \\ [\ \overline{a}^{(m)}\ ] & := & (\max(P_1), \max(P_2), \max(P_3), \ldots, l_{P_m}(a_m) - 1) \end{array}$$

For any $level_{P_j}(a_j) = 0$, the corresponding $[\overline{a}^{(j)}]$ clearly is not valid as level values smaller than 0 cannot occur. So no related $\overline{a}_j$ can exist. We will disregard them in the rest of the proof.

The overall levels of the different $\overline{a}_i$ are:

$$level_P(\overline{a}^{(i)}) = \sum_{j=1}^{m} \max(P_j) - (\max(P_i) - (level_{P_i}(a_i) - 1))$$

The node representing $\overline{a}^{(i)}$ itself only dominates one node in the next higher level: the one with a value of $level_{P_i}(a_i)$ for preference P_i and maximum values for all other preferences. So any value dominated by any of the $\overline{a}^{(i)}$ is also dominated by a. That means, when we find the highest level of a $\overline{a}^{(i)}$, we will find the highest overall level a value can have without being dominated by a.

In the algorithm, the sum of maximum level values is constant, so the result is maximized by minimizing the subtrahend, $\max(P_i) - (level_{P_i}(a_i) - 1)$. *The level computed is the highest overall level for which not all values belonging to it are dominated by a. When we increase this value by one and then choose the minimum for it upon all $\overline{a}^{(i)}$, we receive the formula introduced in Theorem 20. In this level, all values must be dominated by a. If all values having one specific overall level are dominated, of course all values with higher overall levels will be dominated, too. There are some cases when no $\overline{a}^{(i)}$ indifferent to a can be found:*

- $\forall\ i \in \{1, \ldots, m\} : level_{P_i}(a_i) = 0$*: This means, a is represented by the top node. We have already looked at this case.*
- $\forall\ i \in \{1, \ldots, m\} : level_{P_i}(a_i) = \max(P_i)$*: So a's level value is* $\max(P)$. *This special case has been discussed, too.*

So we have found a general formula for determining a tuple's pruning level *for Pareto preferences (and with the tuple just as well for the corresponding node of the BTG).*

Please note that there is no strict relation between a node's level and its *pruning level*. A node with a higher level may have a lower, equal, or higher level than another node. We will see this in Example 28.

Example 28. *The BTG of Figure 5.1 has the bottom node* $(2, 2, 1)$. *We will look at node* $(0, 0, 1)$ *in level 1 and node* $(1, 1, 0)$ *in level 2. Their* pruning level*s are computed as follows:*

$$\begin{aligned} pl_P(0,0,1) &:= 5 - \min(\{2-0, 2-0, 1-1\}) = 5 \\ pl_P(1,1,0) &:= 5 - \min(\{2-1, 2-1, 1-0\}) = 4 \end{aligned}$$

As we can see, although $level_P(0,0,1) < level_P(1,1,0)$*, the* pruning level *of the latter node is better.*

In some cases, it may be more viable not to look for the *pruning level* of a node, but find the *pruning node* for a level – i.e. the one node with the highest overall level that dominates all nodes in a given level. We will address this issue in the next theorem:

Theorem 21. *Consider a Pareto preference* $P := \otimes(P_1, \ldots, P_m)$*. The* pruning node $p^{(v)}$ *for all nodes in a level* v *of* P*'s BTG (except for level 0 and level* $\max(P)$*) can be found by:*

$$\begin{aligned} p^{(v)} := (p_1^{(v)}, \ldots, p_m^{(v)}) \;=\; (\; & \min(\{n_1 \mid \forall n := (n_1, \ldots, n_m) : \; (\textstyle\sum_{i=1}^{m} n_i) = v\}), \\ & \ldots, \\ & \min(\{n_m \mid \forall n := (n_1, \ldots, n_m) : \; (\textstyle\sum_{i=1}^{m} n_i) = v\}) \;) \end{aligned}$$

The two exceptions for which the formula does not hold are the levels containing only one node. The top node in level 0 has no pruning node*, and the bottom node in level* $\max(P)$ *is dominated by any other node.*

Proof. *This follows from the BTG being a lattice and the construction of the supremum of two or more nodes using the function* min *as described in Theorem 9.*

With Theorem 21, it is easy to find the pruning node for a given level v of some BTG. But the formula has the disadvantage of needing to know all nodes in the level. Lemma 13 will show an easier way of computing the pruning node, only taking the level v and the $\max(P_i)$ of the preference as an input.

Lemma 13. *For a preference* $P := \otimes(P_1, \ldots, P_m)$*, the level value* $level_{P_i}$ *of the pruning node of a level* v *of* P*'s BTG for each of the* P_i *is given by:*

$$p_i^{(v)} := \max(0, v - (\max(P) - \max(P_i)))$$

Proof. *For some given* i*, the algorithm tries to maximize the sum of level values for all* $j \neq i$*. If the possible maximum sum of this (given by* $\max(P) - \max(P_i)$*) is lower than* v*, it follows that* $p_i^{(v)}$ *must be higher than zero so the overall level of* v *can be matched.*

In Example 29, we will have a detailed look on the computation of *pruning nodes* for all levels of a given BTG. We will see that, especially for small BTGs, the top node is the *pruning node* for a huge majority of the different levels, leading to hardly any *pruning* possibilities.

Example 29. *Again, we will look at the BTG shown in Figure 5.1, whose preference has the maximum level values 2, 2, and 1. The* pruning nodes *for the different levels are:*

level	*1*	*2*	*3*	*4*
pruning node	*(0,0,0)*	*(0,0,0)*	*(0,0,0)*	*(1,1,0)*

As we can see, levels 1 throughout 3 are completely dominated by the top node only. Looking at level 3 in the BTG, we see that for each of the P_i it contains at least one node holding a value of 0 for it. For levels 1 and 2 this holds just as well. For level 4, we will see how the pruning node *is found using Lemma 13:*

$$\begin{aligned} p_1^{(4)} &:= \max(0, 4-(5-2)) = \max(0,1) = 1 \\ p_2^{(4)} &:= \max(0, 4-(5-2)) = \max(0,1) = 1 \\ p_3^{(4)} &:= \max(0, 4-(5-1)) = \max(0,0) = 0 \end{aligned}$$

The standard algorithms used for Pareto preference evaluation as *BNL*, *SFS*, or *LESS* have been developed for use in skyline queries and therefore were designed to deal with floating point numbers. Until now, we have seen pruning as a technique only capable of dealing with integer level values. The next theorem will show us how to apply pruning to floating-point algorithms. We only have to restrict the preferences to LOWEST and the domain to the interval of $[0,1]$, which (according to [16]) has no impact on the general validity of the approach.

Theorem 22. *For a Pareto preference P containing m lowest preferences on m floating-point domains in the interval of $[0,1]$, a value $x := (x_1, \ldots, x_m) \in [0,1]^m$ has the following* pruning characteristics*:*

$$\sum_{i=1}^{m}(y_i) > m - \min(\{1 - x_i | 1 \leq i \leq m\}) \Rightarrow y <_P x$$

That means, all tuples with a component sum higher than $m - \min(\{1-x_i|1 \leq i \leq m\})$ are dominated by x.

Proof. *The proof is analog to the one for Theorem 20, as is the given formula. The only difference in the formula is that for floating-point values, it is not possible to give a specific* pruning level *(which would be a specific sum of component values), as there is no discrete number of valid sums (in contrast to the discrete number of valid integer level values). So we can only give the maximum component sum for a node not dominated by x and state that all nodes with higher component sums are dominated.*

Using the results given in Theorem 22, *pruning* can be used in any algorithm design for the evaluation of *skyline queries* on floating-point domains as well. In the next section, we will concentrate on the integration of *pruning* in algorithms for BMO set generation, which is a more general case of *skylining*.

5.2 Using Pruning in Algorithms

The *pruning level* for given tuples (or their corresponding nodes in the BTG) can be used to omit comparisons of tuples in preference evaluation algorithms. We will now study the integration in *BNL*, one of the most popular algorithms for BMO queries. Subsequently we will analyze how to use *pruning* in other algorithms as well, even in some designed for *skyline queries*.

5.2.1 BNL^{++}

Block Nested Loop (*BNL*) is the most common algorithm for the evaluation of Pareto preferences since its presentation in [6]. Before we can outline how to integrate *pruning* in this algorithm, we will describe the workflow of *BNL* itself.

BNL

The basic *BNL* algorithm has been introduced in [6] as the first algorithm for multi-dimensional optimization problems in the context of relational databases.

The algorithm uses a main memory buffer called "window" to keep a set of incomparable and un-dominated tuples. When a tuple t is read from the input, it is compared to the tuples in the window. In doing so, one of the following three cases may occur:

- Tuple t is dominated by a tuple in the window and therefore is discarded immediately.
- One or more tuples within the window are dominated by t. All dominated tuples are removed from the window and the new tuple is inserted.
- t is incomparable with all tuples in the window. If there is enough space left in main memory, t is inserted. If not, t is written to some external memory.

When all tuples have been read from the input, those in the window that have been compared to all tuples from the input can be returned as results. To identify these, all tuples receive a timestamp when they are read. Those tuples with timestamps issued earlier than any timestamp for a tuple that has been written to external memory have been compared with all so far un-dominated tuples. As they have proven to be not dominated by any of them, they can be returned as query results and are removed from the window. Now the algorithm starts another iteration using the external memory used in the previous iteration as new input. This continues until there is an iteration without any tuple written to external memory. Then the whole content of the window is returned and the algorithm is finished.

The algorithm was presented together with some improvements for it. One of them is to order tuples wrt their potential to dominate other tuples. So-called *killer tuples* are moved to the beginning of the window so that new tuples are compared to them first (and hopefully dominated without the need for many comparisons). In our algorithm, we will exploit the characteristics of such killer tuples as well.

Integration of Pruning in BNL: BNL^{++}

Using *pruning* is not the only optimization we will apply to *BNL* to improve performance. *BNL*$^{++}$ makes use of three different methods to identify and omit redundant comparisons of input tuples:

- optimization of better-than tests by pruning the BTG
- omission of comparisons of nodes with the same overall level
- omission of multiple comparisons with tuples belonging to the same BTG node

To transform *BNL* to *BNL*$^{++}$, we have to add an additional variable to the algorithm: *pruning_lvl* is the current *pruning level*, i.e. the lowest value for the *pruning level* found in the tuples that have been read yet. This *pruning_lvl* is initialized with $\max(P)+1$, with P being the preference which is about to be evaluated on some input data.

When a new tuple t is read, its overall level value is compared to *pruning_lvl*. If t's overall level value is as high as or higher than *pruning_lvl*, it is pruned and therefore can be discarded without the need for any tuple-to-tuple comparison. Otherwise, t's *pruning level* is calculated and – if *pruning_lvl* is higher than the new value – *pruning_lvl* is reset to the new value.

The list of BMO set candidates is kept in memory sorted by their overall level, with the tuples belonging to the same level stored in lists. Within these lists, tuples are grouped by the node they belong to. So comparisons have to be made only once for each equivalence class.

Due to the sorting by level, t will be compared with tuples with lower levels first and thus can be discarded earlier (if neccessary). When the list of tuples with the same overall level is reached, we can add t to it, as t cannot be dominated by any tuple in the window.

If t was added to the candidate list and the main memory buffer had been full before reading t and no tuple had been removed, external memory access is needed. One of the tuples with the highest overall level value in main memory is written to external memory and removed from the buffer.

When all tuples have been read from the input, those in the main memory buffer that were compared to all other tuples, can be returned as part of the result set and removed from the window. Just as in the original version, tuples therefore get a timestamp when they are read. In addition to this, it is possible to return those tuples with overall levels lower than the lowest overall level of a tuple written to external memory. Due to their overall level value, it is obvious they cannot be dominated, although they have not been compared with all tuples. The latter is the reason why they have to remain in the memory buffer: they could still dominate some of the other tuples.

Afterwards, a new iteration of the algorithm starts. The external memory of the last iteration becomes the input of the current. The algorithm continues like this until there is one iteration when all BMO set candidates fit into the main memory buffer and no external memory is needed.

For ease of understanding, we will now see BNL^{++} in action. The window size chosen is large enough to make external memory access unnecessary:

Example 30. *John's preference on rental cars has been introduced in Example 9. He has a preference on the price and an equally important preference on the color:*

$$P_{John} := \otimes(BETWEEN_5(price; 60, 80), POS/NEG(color; \{red,\ blue\}, \{purple\})$$

In our domain, the maximum level value is 3 for the $BETWEEN_5$ preference and 2 for the POS/NEG preference. The BTG for this preference is shown in Figure 4.4 and has 12 nodes in 6 different levels.

We will use the sample database in Table 5.1. The level values and the pruning level *of course will not be stored in a database but be computed on the fly when reading a tuple. We will start with an empty candidate list. The tuples will be added in order of their ID.*

t_1 *is added to the candidate list. The* pruning level *is set to 5. As there are no other candidates, the tuple's node is added without any comparisons.*

t_2 *is simply added to its already existing node. No comparison is needed.*

t_3 *is – for lack of existence – not compared to tuples with better levels. After being added to the candidate list, worse nodes are searched – and one is found ($(1, 2)$ holding t_1 and t_2) and removed.*

t_4 *is dismissed as its level value is equal to the* pruning level.

t_5 *is added to the list of nodes with level 4. Of course, it was checked before that it is not dominated by the only node in the candidate list:* $(0, 2)$.

ID	hp	color	$level_{P_{hp}}$	$level_{P_{color}}$	$level_{P_{John}}$	$pl_{P_{John}}$
t_1	56	purple	1	2	3	5
t_2	81	purple	1	2	3	5
t_3	70	purple	0	2	2	5
t_4	46	purple	3	2	5	6
t_5	45	silber	3	1	4	5
t_6	95	red	3	0	3	5
t_7	84	blue	1	0	1	3
t_8	88	black	2	1	3	4

Table 5.1: Sample car database with *BNL*$^{++}$ evaluation data

t_6 *is added to the list of nodes with level 3 after being compared with* $(0, 2)$*. Then,* t_6 *dominates and dismisses* $(3, 1)$ *(*t_5*).*

t_7 *lowers the* pruning level *to 3. All nodes with this level or higher ones are removed from the candidate list.* t_7 *is added to level 1's node list. As* t_7 *does not dominate* $(0, 2)$*,* t_3 *stays in the candidate list.*

t_8 *is discarded as its level is equal to the* pruning level.

In this scenario, only 5 comparisons have to be made. The simple BNL *from [6] would have needed 11. So the comparisons were cut by half, although the* pruning level *was set to its final value only when reading the penultimate tuple – but it still helped to reduce the number of comparisons (by pruning* t_6 *at once and* t_8 *later).*

We have made some performance tests of *BNL*$^{++}$ in cooperation with MAN Roland[1], the world's second largest manufacturer of professional printing systems. The algorithm was also used in the prototype implementation of an MAN Roland internal online procurement shop described in [24]. Our tests were made on real data, in particular a set of 4,925 drills. Preferences on eight (four numerical, four categorical) different attributes were used in 256 preference queries. The result sizes where between 1 and 1,250 tuples. Compared to the execution time of *BNL*, *BNL*$^{++}$ reduced the mean query execution time by two thirds. In other words, it ran three times faster than BNL.

[1]http://www.man-roland.com

5.2.2 Integration in other Algorithms

Pruning can be integrated in any algorithm designed for the evaluation of Pareto preferences. We will examine this in the next section. But even more is possible: algorithms for *skyline queries* on floating-point domains can also benefit from *pruning*, as we will see later.

Pruning in Level-based Algorithms

Basically, using *pruning* in an algorithm for BMO set computation for Pareto preferences is quite straight-forward. We will concentrate on the general steps of integration. Special conditions arising from some given algorithm will not be considered.

Just as for BNL^{++}, we add an integer variable *pruning_lvl* to the algorithm. Again, *pruning_lvl* set to $\max(P) + 1$ at the beginning (with P as the preference to be evaluated). When a new tuple is read from the input, its overall level is compared to *pruning_lvl*. If the new tuple's level is higher than *pruning_lvl*, it can be discarded without any tuple-to-tuple comparisons. If the new tuple has a *pruning level* better than *pruning_lvl*, the variable is reset to the new value.

For testing and benchmarking this method of integration, we have created an implementation of Godfrey et al.'s *LESS* algorithm ([31]) using *pruning*, which we call $LESS^{++}$.

We will now describe the basic workflow of *LESS* to be able to understand the adjustments made to exploit *pruning*. *LESS*, just like *BNL*, consists of a number of passes, although the first pass differs very much from later ones. In pass zero, when the input relation is read for the first time, the memory buffer is split up into two parts, the EF window and the sort window. The EF window (which usually is about 1% of the size of the sort window ([31, 47])) is used to keep a small set of *killer tuples* which all are by now un-dominated and show good values wrt some quality measure. This quality measure has to provide a topological sorting of the tuples so that a tuple with good quality cannot be dominated by a tuple with worse quality ([31, 16]). A good quality measure in our context would be the overall level value of a tuple.

Each time a tuple is read, it is compared to all the tuples in the EF window. Hence, a lot of tuples can be filtered out during the first pass. A tuple is added to the EF window (and may replace another tuple in it), if its quality is higher than the quality of a tuple in the EF window. Dealing with our preference model, this would mean the new tuple has to have a lower overall level value. Another reason to add a tuple is space available in the EF window. This occurs after removal of dominated tuples (and of course at the beginning of the execution of the algorithm). All tuples that are not discarded are stored in the sort window. As soon as this sort window is full, the contained tuples are sorted by their overall level value and stored in some external memory. When all tuples are read from the input and have been filtered and sorted, pass zero ends.

Before pass one begins, a complete sorting of the input is carried out using some external sort algorithm. From now on, the complete memory buffer can be used to store BMO set candidates. Due to the sorting, the tuples are read in order of their quality (i.e. overall level value for us). Each time a tuple is not dominated when compared to the current window content, it can be written to the output as it is part of the result set. Keep in mind that no tuple can be dominated by a tuple that is read later as the later tuple has a worse quality (resp. a higher overall level value).

Beginning with pass one, all passes work very similar to *BNL*. If not all input tuples fit into the main memory buffer, the surplus ones are written to external memory again. The algorithm continues with more passes until no tuples have to be written to external memory anymore.

The changes in *LESS* that have to be made to form *LESS*$^{++}$ are primarily made in pass zero. There, the value of *pruning_lvl* has to be adjusted as new tuples are read. In pass one, we only have to filter out some tuples with worse overall levels that occasionally were read (and stored on disk) before the *pruning_lvl* was set to a value discarding them. From pass two on, *LESS*$^{++}$ behaves exactly like *LESS*.

Due to the very good filtering characteristics of the basic *LESS* algorithm, we could only recognize mean performance enhancements of some percents for *LESS*$^{++}$ over *LESS*. Nevertheless, we think that *pruning* is a reasonable improvement for the algorithm, especially in cases like the one described in the next example, where a huge performance boost was made possible by *pruning*.

Example 31. *Consider a Pareto preference of WOPs with the maximum level values 1, 1, and 100. The input relation starts with 100,000 tuples with overall levels from 5 to 102. Then, one of the last tuples t in the input relations has the level combination* $\{0, 0, 4\}$*, having a* pruning level *of 6. In the second phase of the evaluation with* LESS^{++}*, we only need to read the nodes from level 5 from external memory to compare it with t. As soon as the first tuple with an overall level of 6 is read, the algorithm is finished as none of the following tuples can be part of the BMO set.*

In our test system, LESS *took 45s for the evaluation, whereas* LESS^{++} *only needed 21s. A performance gain of more than 50% was achieved.*

As we have seen in our tests, the average case of a preference evaluation with *LESS* is hardly affected by integrating *pruning*. But Example 31 has shown how some worst cases for *LESS* can be bypassed by *pruning*.

Please note that although this basic method is relatively easy and yet leads to good results, as seen in Example 31, the complete integration leading to maximum performance benefits may be far more complex, as we could see for *BNL* in Section 5.2.1. Of course, this will entirely depend on the algorithm.

Pruning in Algorithms for Floating-Point Domains

As we have seen in Theorem 22, *pruning* may also be used for floating point domains, which are common in *skyline queries*. *Pruning* can be integrated into an existing algorithm by altering it in some minor, well-defined spots. In [16], a simplification of *skyline queries* on floating-point domains was described. Every part of the query can be reduced to the equivalent of a LOWEST preference on the interval of $[0, 1]$. So the *skyline* consists of tuples with attribute values as low as possible.

There has to be a variable *pruned_sum* that keeps track of the lowest sum of tuple components that cannot be part of the result set (analog the *pruning_lvl* of Section 5.2.1). The initial value of *pruned_sum* is m, with m being the number of *skyline dimensions* (which corresponds with the number of WOPs contained in a Pareto preference P). Each time a tuple is read, the sum of its components has to be compared to *pruned_sum*. If the new tuple's sum is higher than *pruned_sum*, it can be discarded at once, without any further need for tuple-to-tuple comparisons at all. If it is not discarded, the *pruning characteristics* of the new tuple have to be determined as described in Theorem 22 and – if lower than *pruned_sum* – can be used as new value for *pruned_sum*.

Chapter 6

The Hexagon Algorithm

In this chapter, we will learn how to ultimatively exploit the lattice BTG for a preference for BMO set generation. First, we will get to know the basic ideas of *Hexagon*, including considerations of computational complexity and memory requirements. Then we can have a closer look at different implementation strategies, that – apart from the external version of *Hexagon* of Section 6.2 – primarily differ from each other in their memory management. In Section 6.4 we will see how the algorithm not only works fine for standard Pareto preference query evaluation, but can also be successfully deployed to the related problems *multi-level Pareto preference* and *top-k* queries.

6.1 Layout and Characteristic Features

Hexagon is based on the BTG for a preference. Input tuples are only needed to know if a node has tuples belonging to it or not. After getting to know the characteristic workflow of a preference evaluation on some input relation using *Hexagon*, we will learn about the theoretical computational complexity of the algorithm and the memory consumption using straightforward programming techniques.

6.1.1 Principles of Hexagon Algorithms

Tuples are mapped to nodes of the BTGs and the computation of dominance and indifference is done for nodes. This abstraction from tuple-to-tuple comparison is the main advantage of *Hexagon* over other algorithms. The maximum number of comparisons depends on the BTG. The distribution of input tuples accounts for the actual number of comparisons that have to be made. As we will see later, different distributions of input tuples are cooperative to a different degree. *Hexagon* always consists of three phases:

1. Construction Phase
2. Adding Phase
3. Removal Phase

It would be possible to speak of a forth phase, a *returning phase*, in which the tuples belonging to the BMO result set are returned. But as this can be included in phase three without any negative influence on the overall performance, we will address this issue when reaching the removal phase. *Lattice Skyline* (*LS*), presented independently in [47] at the same time as *Hexagon* in [51], works the same way.

We will now have a closer look at these phases.

Construction Phase

At the beginning, we have to initialize the BTG. Therefore, we have to determine its size. As we have seen in Section 4.3.2, we need to know the maximum level values of all WOPs in the Pareto preference. Some memory structure for the whole BTG has to be iniatialized in main memory. As we have a unique integer ID for each node, an array-like structure seems to be favorable. Please note that the Pareto preference may contain strict partial orders in the way introduced in Section 4.5 as well. As each preference is reduced to the BTG constructed by it, only integer value combinations are important.

In the *removal phase*, we will have to do a breadth-first walk through the BTG. Nodes should be reached in order of the level they belong to. Within each level, they should be visited in ascending order of their IDs. Therefore, we construct a *next*/*prev* node relation that keeps information about the predecessor and the successor of each node in the breadth-first walk.

This breadth-first order relation can be computed by visiting each node of the BTG in order of the node IDs. For each node, we need one pointer on its *next* and one on its *previous* node, best initialized as arrays of pointers or ID values. In the following description, we will concentrate on the *next* relation. Pointers on *previous* nodes are set at the same time as a *next* pointer is set.

In an auxiliary array *last*, we keep the highest ID found belonging to each level so far and in an array *first*, we keep the lowest ID for a node in each level. The *next* node of the top node always is the one with ID 1. Now we visit each node in order of its ID.

We compute the level a node belongs to by constructing its level value combination from its ID as seen in Lemma 9 and then determining the level the node belongs to (see Theorem 12). We look up the (until we reached the current ID) highest ID h for a node in the same level in *last* and set the current ID as value for *next* of h (and x as *prev* ID for the current value). Then, we set the current ID as the new highest ID for its level in *last* (and if no node of this level has been found before), we also set it as lowest ID in *first*.

When all IDs have been processed, we use the information in *first* and *last* to connect the different levels in the breadth-first walk. For a level x, *last* holds the last node in the breadth-first walk. The next node can be found in *first* for level $x+1$. The value of *last* for a level x is used as *prev* value for the ID stored in *first* for level $x+1$. A complete breadth-first walk over the BTG is created.

This breadth-first walk does not necessarily need to be constructed in this phase of the algorithm. It is not needed before the *removal phase* and can also be constructed then. The next example will show the initalization process. We will use it as a running sample throughout the whole introduction of *Hexagon*.

Example 32. *Remember John's preference for cars (cp. Examples 9, 30):*

$$P_{John} := \otimes(BETWEEN_5(price; 60, 80), POS/NEG(color; \{red,\ blue\}, \{purple\})$$

Together with the input relation of Table 5.1, we get maximum level values of 3 for the BETWEEN$_5$ *preference and 2 for the POS/NEG preference. Following Theorem 10, the BTG has* $(3+1)*(2+1) = 12$ *nodes. In memory, we initalize an array to store the state of each node. At the beginning, each node of course is set to* empty. *The table shows how the BTG is represented in main memory after the construction phase:*

id	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*
state	*e*	*e*	*e*	*e*	*e*	*e*	*e*	*e*	*e*	*e*	*e*	*e*
next	*1*	*3*	*4*	*2*	*6*	*7*	*5*	*9*	*10*	*8*	*11*	*-1*
prev	*-1*	*0*	*3*	*1*	*2*	*6*	*4*	*5*	*9*	*7*	*8*	*10*

Adding Phase

In the adding phase, the input relation is read. For each tuple, we have to determine its equivalence class and so the node of the BTG it is represented by. We have to store the information that a node's state changes to *non-empty*. We will see different approaches to the storing of the node state later in Section 6.3.

Example 33. *After adding all tuples from Table 5.1, the node states have changed to the following ("n" stands for* non-empty*). The* next/prev *relations are omitted as they are not relevant for this state.*

id	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*
state	*e*	*e*	*n*	*n*	*e*	*n*	*e*	*n*	*e*	*n*	*n*	*n*

Removal Phase

The last phase of the algorithm is the removal of dominated nodes. There are different ways of walking through a BTG to visit each node and to determine if it is dominated by some other node(s). The primary version of *Hexagon* (cp. [51]) used a combination of breadth-first and depth-first walk. The nodes of a BTG are, in general, visited level-by-level in the breadth-first order built in the *construction phase*.

When an empty node is reached, it is removed from the *next/prev* relation so that in another breadth-first walk, it will not be visited anymore. Each time a dominated node is found, a depth-first walk is done marking all dominated nodes. In this walk, every node that is reached is marked as *dominated* and removed from the *next/prev* relation. That way, it will not be visited in the further progress of the breadth-first walk. The weight w of the edge a node is reached with during the depth-first walk is very important. From each node, only edges with weights lower or equal to w are followed. The depth-first walk reaches a dead end in a node, when no edge with a lower weight can be found that leads to a non-dominated node.

Hence, during the walk, each node is reached by only one edge. Most of the edges are not used and the computational complexity of the algorithm is based on the number of nodes in the BTG. In Example 34, we will see the state of the complete BTG in some phases during the *removal phase*.

Example 34. *In Figure 6.1, the removal phase is visualized in several states. Each part of the figure only shows those nodes still reachable by the* next/prev *relation, together with the edges between them. At the beginning, the full BTG can be reached following the breadth-first walk based on* next/prev*.* Non-empty *nodes are marked in gray (Figure 6.1(a)).*

Starting the breadth-first walk in the top node, the first interesting node is the first non-empty *node* $(1, 0)$*, having the ID 3. Nodes* $(0, 0)$ *and* $(0, 1)$ *were removed from* next/prev *as they are* empty*, as we see in part (b) of the figure.*

Then, the depth-first walk starts. First we go to node $(1, 1)$*, reached from* $(1, 0)$ *by an edge with weight 1. From this node, we can only use the edge with weight 1, leading to node* $(1, 2)$*. As no edge with weight 1 emerges from* $(1, 2)$*, a dead end is reached.*

This part of the walk is illustrated by the blue edges in Figure 6.1(c). Both visited nodes are marked as dominated *and removed from* next/prev *(see Figure 6.1 (d)).*

Then, we follow the edge with weight 3 (green edges in 6.1(e)) coming from $(1, 0)$*, reaching* $(2, 0)$ *first. From this node, we follow edges with weight 1 (blue edges, again) and edges with weight 3 (green). After removing the dominated nodes from* next/prev*, we reach the state shown by Figure 6.1(f).*

Continuing the breadth-first walk leads to node $(2, 0)$*. As all dominated nodes are already marked as* dominated*, no depth-first search has to be done. As no non-*dominated next *node exists, the removal phase ends here.*

After the removal phase*, the BTG in memory shows the following states (with "d" for the state* dominated*.* next/prev *only holds valid entries for the remaining* non-empty*,* non-dominated *nodes:*

id	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*
state	*e*	*e*	*n*	*n*	*d*	*d*	*d*	*d*	*d*	*d*	*d*	
next	*-1*	*-1*	*-1*	*2*	*-1*	*-1*	*-1*	*-1*	*-1*	*-1*	*-1*	*-1*
prev	*-1*	*-1*	*3*	*-1*	*-1*	*-1*	*-1*	*-1*	*-1*	*-1*	*-1*	*-1*

It is equally possible to walk through the BTG in order of the node IDs. Each time a non-empty or a dominated node is reached, the nodes directly dominated by it are marked as dominated. But, as a consequence, each edge in the BTG has to be followed, leading to $edges(BTG_P)$ node visits. This simpler approach is used in *LS* ([47]). Starting a depth-first walk as described for the *removal phase* in the top node leads to an order of visiting the nodes exactly equal to the node IDs.

6.1.2 Computational Complexity

Due to the very strict partitioning of the algorithm into three different phases, it is possible to split up the examination of the computational complexity in three parts as well. In the last part of this section, the results for the three phases of *Hexagon* will be merged and the overall theoretical complexity of the algorithm will be found.

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 Theorem 10). For the creation of the *next/prev*-relation, this array is traversed exactly once. The level of each node has to

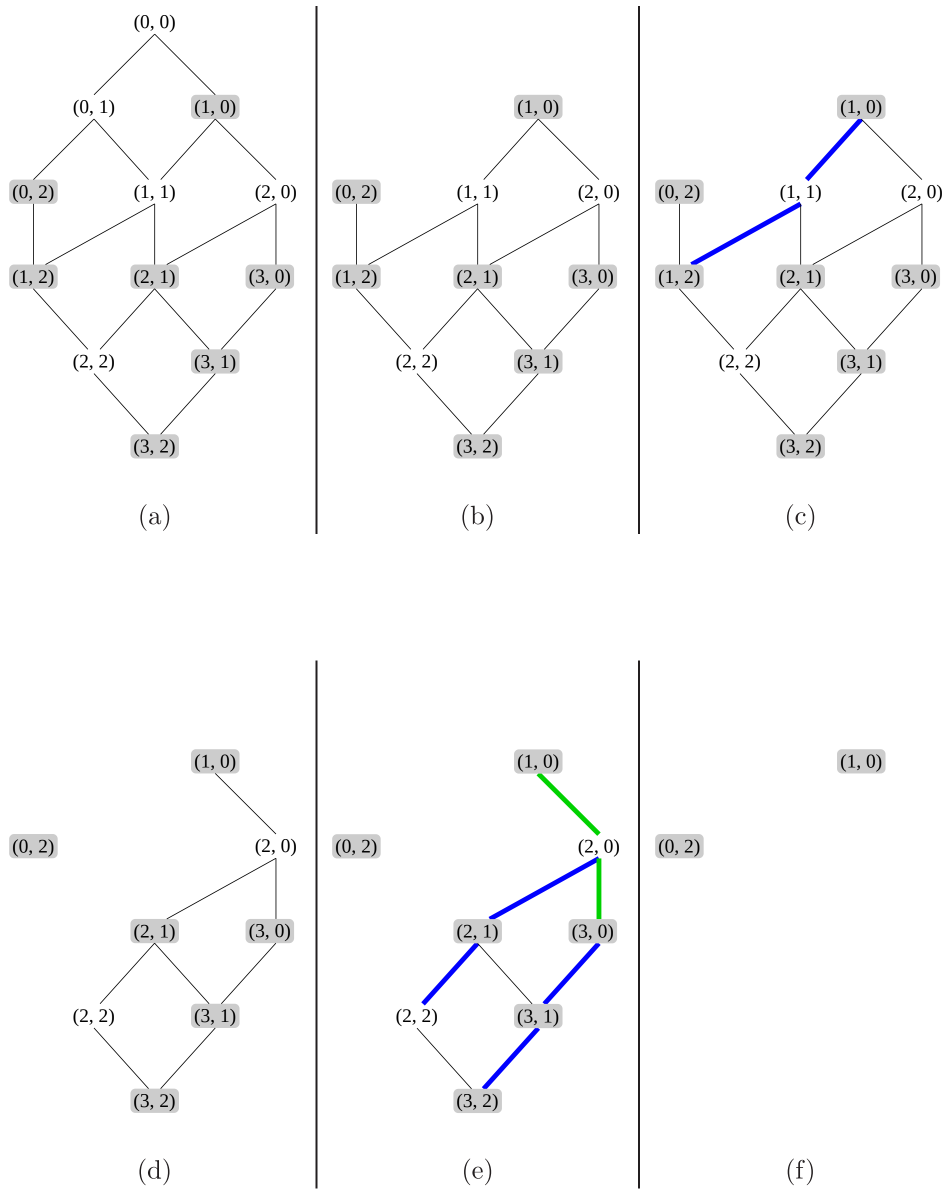

Figure 6.1: BTG during *Hexagon*'s removal phase

be computed. Using the method of Lemma 9, 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 carry out some computations for each of them. As seen in 15, 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 again depends on the size of the BTG. Doing the whole breadth-first walk defined by the *next*-relation is done in $\mathcal{O}(|BTG|)$. For each node reached, it is possible that it contains tuples and the nodes dominated will have to be removed. When a node is reached that has been removed from the *next*-relation, 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* is not the same as the current node, the current node has been removed already. 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 can only 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 were visited this often, it would be done in $\mathcal{O}(m * |BTG|) = m * \mathcal{O}(\prod_{i=1}^{m}(\max(P_i) + 1))$.

Complete Computational Complexity

Using the results of the previous parts of this section, we can now easily state the theoretical computational complexity of an execution of the *Hexagon* algorithm. Following the common principle of "space-time tradeoff", *Hexagon* usually uses more space but less time than other algorithms to evaluate a Pareto preference. Theorem 23 shows the results:

Theorem 23. Hexagon *is linear wrt the BTG size.*
For 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 parts to add to find the overall complexity of* Hexagon*:*

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

The overall complexity can be found by simply adding complexities of the different phases. As the adding phase is the only phase with a complexity depending on the size of the input relation, it clearly depends on the size of the BTG and the number of input tuples. As the BTG and especially m are fixed and usually very small compared to n, we find a worst-case complexity of $\Theta(n)$.

Small sized BTGs frequently occur in practice, in particular for e-commerce applications. Due to their limited maximum level values, categorical preferences never pose any problems, but numerical preferences like AROUND$_d$ might do so. In the worst case, each of the n tuples of an input relation could be mapped to a different equivalence class, yielding a BTG size of $(n + 1)^m$. For such extreme scenarios, *Hexagon*, of course, cannot be used in an efficient way. However, apart from the exponential complexity in such cases leading to long execution times, main memory requirements for the BTG would exceed the possibilities of existing computers even for very small values of n.

6.1.3 Memory Requirements

Each node of the BTG has one of three different states: *empty*, *non-empty*, and *dominated*. The easiest way to encode these three states is by using two bits with 0x00 standing for *empty*, 0x01 for *non-empty* and 0x10 for *dominated*. This enables us to use the extremely fast bit functions to check and change node states. The very same encoding has been proposed for *LS* for the same reasons as well ([47]).

Lemma 14. *The BTG for a Pareto preference $P := \otimes(P_1, \ldots, P_m)$ requires the following amount of memory:*

$$mem(BTG_P) := \left\lceil \frac{nodes(BTG_P)}{4} \right\rceil = \left\lceil \frac{1}{4} \prod_{i=1}^{m}(\max(P_i) + 1) \right\rceil$$

Proof. *One byte can hold four nodes using two bits each.*

So the memory requirement for *Hexagon* is linear wrt the size of the BTG. The next

	$\max(P_i)$	$mem(BTG_P)$
P_A	2, 2, 1	5 Bytes
P_B	2, 2, 100	228 Bytes
P_C	20, 20, 20	2316 Bytes
P_{John}	3, 2	3 Bytes
P_{CC}	20, 20, 20, 20, 20, 20	$\sim$ 20.5 MB

Table 6.1: Memory requirements for different BTGs

example shows memory requirements for some sample BTGs. We will see how to evade the problem of high memory consumption in the next two sections.

Example 35. *In Example 15, we got to know the Pareto preferences P_A, P_B, and P_C. We will also look at the BTG of the* Hexagon *Example 32 to Example 34 and a preference $P_{CC} := \otimes(P_1, \ldots, P_6)$. Table 6.1 holds the results. As we can see memory requirements can be very high for a Pareto preference holding only six WOPs with maximum level values of only 20 each.*

6.2 Hexagon with External BTG: Hexagon^{++}

Until now, we have always had to keep the complete BTG in main memory. As we have seen in the last section, memory requirements can be very high in some cases. The logical next step would be to keep only parts of the BTG in main memory while evaluating the preference.

We have developed a *Hexagon* derivate holding in memory not the complete BTG but only one of the BTG levels at a time. By marking dominated tuples in the BTG level by level, this version of *Hexagon* is capable of finding the best *pruning level* for any input relation. As we integrate pruning in *Hexagon*, analog to *BNL*$^{++}$, we call this algorithm *Hexagon*$^{++}$.

When reading the input relation tuple by tuple, do the following:

1. Keep in memory a sorted list of all unique IDs of the nodes for which tuples belonging to have been read. If a new tuple from a lower level is found, discard the IDs in memory and only keep the new tuple.
2. Store the tuple on external memory together with the other tuples from its level.

Note that there are no comparisons made in the first round of the algorithm. Step 2 requires a dedicated external memory space for each level in the BTG. No specific sorting has to be carried out as the external memory for one level works as a bucket for the tuples belonging to it. A bucket sort is done automatically. All tuples belonging to the lowest level can be written to the output right after all input tuples have been read. They cannot be dominated.

We then compute all nodes belonging to the lowest non-empty level and keep them in ascending order wrt their unique IDs. We mark the nodes with input tuples belonging to them as *non-empty*.

Now we walk through the BTG in a breadth-first order. In each node that we reach, we do the following:

1. Create the nodes connected to the current node in the spanning tree and store them in a list with the other nodes belonging to the next level.
2. If the node is marked as *dominated* or *non-empty*, mark its successors in the spanning tree as *dominated*.
3. Store other dominated nodes in *domination lists*. For each WOP (except for P_m), there is such a list.
4. Remove the current node from the current level.

After the new level has been constructed, tuples belonging to non-dominated nodes in this level can be returned. Therefore, we read all tuples belonging to this level from external memory. For each tuple we read, we have to check if the related node is marked as *dominated*. If a node is marked as *empty*, we change it to *non-empty* and return the tuple. So when constructing the next level, dominated nodes are marked as such.

For each WOP (except for P_m), we keep a list of nodes dominated by it. These lists keep nodes for which it is known they are dominated but that have not been reached by the breadth-first walk.

When walking through the BTG in a breadth-first manner, in each node create those nodes directly connected to the current node in the spanning tree of the BTG.

For non-empty and dominated nodes, we have to perform some additional tasks:

- mark nodes dominated in the spanning tree as dominated (as the spanning tree contains each edge for domination due to P_m, we do not need a list for this preference) and
- add other dominated nodes (those dominated by WOPs with greater edge weights) to the respective lists of nodes dominated for some WOP.

In each empty and non-dominated node, the lists of dominated nodes have to be checked. All elements of nodes dominated by the respective WOP with IDs lower than the current node's ID plus the WOP's edge weight are removed and all nodes that are dominated in the spanning tree by the current node are marked as dominated.

Note that the lists always keep the node IDs sorted. We only have to check for domination of the first node with a greater ID in the list (those with lower IDs are removed). If the current node would dominate another node in some list, the first node in the list would have been dominated because of the same WOP by a node with a lower ID than the current node.

Assume that the current node would dominate the second node in the list for some WOP, but not the first. Hence the ID of the first node is lower than the ID of the current node plus the edge weight for the WOP. As IDs are increasing while walking through a level, no node following the current node dominates this first node for this WOP. The relevant node has been passed already.

When we have finished computations for the current level, a sorted list of tuples belonging to the next level has to be produced. We read this list element by element to check for non-empty nodes. Each time we reach a non-empty, non-dominated node, the tuples belonging to it can be written to the output. The algorithm can stop as soon as one level only holds dominated objects. Clearly this level and all following levels are pruned.

For this algorithm, the memory buffer has to be at least the width of the BTG. During the construction of a new level, all old nodes are removed as soon as they have been used to construct their successors.

Example 36. *In Example 34, John's Pareto preference for cars was evaluated using* Hexagon. *We will now evaluate it with* Hexagon^{++}. *The BTG for this preference and its database (Table 5.1) can be found in Figure 6.1(a), where non-empty nodes are shaded in gray. For now, it is only relevant that (among others) nodes* $(1,0)$ *and* $(0,2)$ *have tuples belonging to them.*

The lowest level with non-empty nodes is level 1. After all tuples have been read, all nodes of level 1 are computed and we begin to walk through level 1. The first node we visit is $(0,1)$*, which is empty. It has one direct successor,* $(0,2)$*. This is the first node of level 2 in our breadth-first walk. The following table shows the relevant data for the algorithm. The node states are indicated by the indexes (*e *for* empty, n *for* non-empty *and* d *for* dominated*):*

current level:	$(0,1)_e \rightarrow (1,0)_n$
next level:	$(0,2)_e$
dominated by P_1:	$\{\}$

The non-empty node $(1,0)$ *is reached next. While processing it, we find two more nodes of level 2, both dominated by* $(1,0)$*:*

current level:	$(1,0)_n$
next level:	$(0,2)_e \rightarrow (1,1)_d \rightarrow (2,0)_d$
dominated by P_1:	$\{\}$

Now we are finished with level 1. Read tuples belonging to level 2 from external memory and return those belonging to the non-dominated node $(2,0)$*, which is marked as non-empty. Level 2 now becomes the current level.* $(2,0)$ *has one successor it dominates:*

current level:	$(0,2)_n \rightarrow (1,1)_d \rightarrow (2,0)_d$
next level:	
dominated by P_1:	$\{(1,2)\}$

When we reach $(2,2)$*, we find one of its successors in the list of nodes dominated for* P_1*. We remove it and add it as dominated node to the next level:*

current level:	$(1,1)_d \rightarrow (2,0)_d$
next level:	$(1,2)_d$
dominated by P_1:	$\{\}$

The last node in level 2, $(2,0)$ *completes the execution of the algorithm. As the current level has no more nodes and all nodes in the next level are dominated, we can stop now:*

current level:	$(2,0)_d$
next level:	$(1,2)_d \rightarrow (2,1)_d \rightarrow (3,0)_d$
dominated by P_1:	$\{\}$

Hexagon^{++} is capable of reducing main memory requirements to a very promising degree, but does this at the expense of performance – once again a typical case for the well-known time-space tradeoff in computer science: less memory requirements of an algorithm in comparison to another lead to more time consumption solving. In most of our tests, BMO set computation of *Hexagon*$^{++}$ was slower than even standard *BNL*. We have identified as main problem the sorting of input tuples according to their overall level value. But even in the case of a presorted input relation, the dynamic construction of the BTG produced loads of computational overhead. Therefore, it just could not compete with the common array-based approach of *Hexagon* as introduced in Section 6.1.1. Besides, *Hexagon*$^{++}$ could not outperform or at least approximate the benchmark results of *LESS* in any test.

As a consequence, we do not consider it as an appropriate approach for use cases with limited main memory. To reduce memory complexity, we will concentrate on optimizations for the standard version of *Hexagon*. The next section will show the results of these efforts.

6.3 Reducing Memory Requirements for Hexagon

For Pareto preferences containing many WOPs or a number of WOPs with large maximum level values, the memory available for keeping the BTG may become a problem. So it might occur that *Hexagon* cannot be used despite its superior performance.

In this section, we will describe various different ways to reduce *Hexagon*'s memory requirements. The wingspan of methods reaches from bit efficiency via presorting up to modeling issues. Afterwards, we will see if and how the introduced methods can be combined for maximum memory savings.

6.3.1 Enhancing Memory Efficiency

The straightforward implementation of *Hexagon* dististinguishes between three different states for each node: *empty*, *non-empty*, and *dominated*. For *LS*, a trivial approach of encoding the three states in two bits is used, leading to four nodes per byte. We will now see two methods to overcome this waste of memory induced by the unused forth state two bits can form.

Using “More” than 8 Bits per Byte

A single byte has $2^8 = 256$ different states. Using 2 bits to store the status information of one BTG node, we use $3^4 = 81$ states of one byte. So the following can be considered:

$$\begin{aligned} 3^x &\leq 2^8 \\ \text{iff } x &\leq \frac{\ln 2^8}{\ln 3} = 8 * \frac{\ln 2}{\ln 3} \approx 5.047 \end{aligned}$$

This is very much an encoding of the state of 5 nodes in a ternary number. That way we are able to store the states of 5 nodes in one byte of main memory, leading to 25% bigger BTGs that fit into main memory. Please note that reading and writing node states cannot be done using extremely efficient bit operations anymore. The disadvantage of this method is that we have to use the comparatively expensive operations division and multiplication to read and write node states. But the effort is worth the benefit, as the algorithm complexity of course remains linear.

As 5 nodes only use $3^5 = 243$ of the available 256 states of one byte, we could go further and use the whole memory to store one big ternary number for the BTG. In doing so, however, it would no longer be possible to easily process division and multiplication needed for access to single nodes. And with additional memory savings of less than 1%, we consider this method not to be cost-efficient.

Eliminating One State and All Unused Byte States

Another possibility to increase memory efficiency is based on the reduction of the number of relevant states for a node to two, *dominated* and *non-dominated*. Using only two states, memory requirements are cut by half compared to the basic version *Hexagon*. Then, each node can be represented by a single bit. Of course, some modifications have to be made to the algorithm.

While in the original version of the algorithm, *non-empty* nodes were marked as such, this is of course no longer possible or neccessary. However, *dominated* nodes still have to be marked. So, if a tuple is read that belongs to a node that would already have been marked *non-empty* in the original algorithm, the memory-enhanced version will just start the depth-first walk to mark all dominated nodes. As the node has been visited before, the walk ends very abruptly: all the directly dominated nodes are already marked as *dominated*.

Computational effort increases of course, but in the tests we made, this had hardly any impact on the overall performance of the algorithm. Using this method, we can store status information for eight nodes in one byte, which means we are doubling the BTG size fitting into a main memory buffer of some fixed size.

6.3.2 Storing the Best Value for one WOP

Another way of reducing memory requirements is breaking up the preference that is evaluated in two parts. We split $P = \otimes(P_1, \ldots, P_{m-1}, P_m)$ into sub-preferences $P' = \otimes(P_1, \ldots, P_{m-1})$ and P_m. Please note that due to commutativity, we can change the order of the $P_i \in P$ without changing the semantics of the preference. In fact, we can do this for any $P_i \in P$. For reasons of comprehensibility, the following is explained using P_m.

Node States and Best Values

In memory, we will construct a BTG for $P' = \otimes(P_1, \dots, P_{m-1})$. In addition to the node state, we keep a record on the best level value for P_m found for each node so far. This value may have been found in a tuple with the level combination of the node itself or a dominating node.

The workflow of the algorithm now has to change. Every time a tuple is read, the node it belongs to is marked as *non-empty*. Apart from this, the lowest value found for P_m for this node has to be stored. When a tuple is read with a higher level value for P_m than the lowest value read until now, it is discarded at once. When all tuples have been read, the BTG is walked through in breadth-first order. The best value for P_m is the lowest value found in the node itself and the dominating nodes.

When the algorithm walks through the input tuples to return them, it is checked which node they belong to. If a node is marked as dominated and a tuple's level value for P_m exceeds the one stored in the node, the tuple is dominated and discarded.

To store a level value for P_m, $\left\lceil \frac{\ln(\max(P_m))}{\ln 256} \right\rceil$ bytes are needed. In each byte, we store the state of x nodes (with $x = 4$ for the common implementation as described in Section 6.1.1). For the whole BTG, we get the following memory consumption:

$$\left\lceil \prod_{i=1}^{m-1} (\max(P_i) + 1) * \left(\left\lceil \frac{\ln(\max(P_m))}{\ln 256} \right\rceil + \frac{1}{x} \right) \right\rceil$$

Only for very small BTGs with memory requirements of 5 or fewer bytes this value may be bigger than the usual amount of memory needed. Comparing this formula to the usual BTG size introduced in Section 4.3.2, it is obvious that the best size reduction is achieved if P_m has the highest maximum level value of all P_i. The following example will show the great potential of this approach:

Example 37. *Let's analyze BTG memory consumption for some Pareto preferences with three WOPs. Their maximum level values can be found in the three first columns of Table 6.2. Column four shows the memory required using the storage method introduced in Section 6.3.1. In the fifth column, the amount of memory for the reduced storage is shown. In the last column, the reduction rate is shown. We see that the memory requirements are cut by up to 99%.*

This method is quite similar to the *Extended LS* proposed in [47], although Morse et al. use it to make *LS* capable of dealing with one unrestricted domain, not to reduce main memory requirements.

$\max(P_1)$	$\max(P_2)$	$\max(P_3)$	normal BTG	reduced BTG	reduction quota
10	10	10	267 B	146 B	45%
10	10	100	2,445 B	146 B	94%
100	100	100	206,061 B	12,242 B	94%
100	100	1000	2,042,241 B	22,443 B	99%

Table 6.2: Sizes of normal vs. reduced BTG

Best Values – and No Node States

The method described in the last section can be enhanced further. Basically, again one of the preferences P_i is extracted from P and the best values for it are kept separated. But this time, we omit storing node states completely.

The BTG is constructed for the sub-preference $P' := \otimes(P_1, \ldots, P_{m-1})$ of $P := \otimes(P_1, \ldots, P_m)$. For each node, we store one value representing the best value for P_m seen for the node itself or dominating nodes.

The function *enc* returns the encoded value for all values for which $level_{P_m}$ is defined:

$$\begin{array}{rcl} enc : dom(A_m) & \rightarrow & int \\ val(a_m) & := & \max(P_m) + 1 - level_{P_m}(a_m) \end{array}$$

The function values for *enc* range from 1 to $\max(P_m) + 1$. Note that for *enc*, better (in terms of P_m) means *higher* values. For *empty* nodes, the BTG in memory should hold a value of zero. As this is not a possible result of *enc* for a tuple, the status of the node is clear.

The definition of *enc* yields domination tests as simple comparisons of integer values. Empty nodes show the worst possible value wrt P_m and are practically ignored.

The algorithm then has the following workflow:

1. Construct the BTG as an array of $|BTP_{P'}|$ integer values. Node IDs are defined by those for $BTP_{P'}$.

2. Read the tuples from the input.

   For each tuple, store the *enc* value in the corresponding node. The new value for *enc* is the maximum of the old *enc* value and the new tuple's *enc* value.

   If the tuple has a higher value for *enc* than the previous value, it can be discarded at once. There must have been another tuple with equal level values for P_1 to P_{m_1} and a better level value for P_m.

   Otherwise the new tuple can be stored in external memory.

3. Walk through the BTG (e.g. in order of node IDs).

   In each *non-empty* node, do a depth-first walk to propagate the *enc* value increased by 1 downwards in the graph. The new *enc* value for a node is the maximum of the old and the propagated value.

4. Read the cached tuples from external memory. If the *enc* value kept for the corresponding node for P' in the BTG representation is lower than current tuple's *enc* value, the tuple is discarded.

Phases two and three may be combined. Each time a tuple is read (and not discarded), a depth-first walk is made to adjust the *enc* values for dominated nodes.

Due to the addition of 1 in step three, the maximal propagated value is $\max(P_m)+2$, representing the (impossible) level value of -1. This value is used if a tuple t has a level value of zero for P_m. A tuple that is dominated by t wrt P' (and hence reached in the depth-first walk starting at the node representing t) would need the hypothetical level value of -1 for P_m to be not dominated by t wrt P.

Please note that for small values of $\max(P_m)$, more than one value could be stored in a single byte, reducing the required memory even more. Example 38 will show this.

Example 38. *Consider a preference* $P := \otimes(P_1, P_2, P_3)$ *with maximum level values* $\max(P_1) = 2$, $\max(P_1) = 5$, *and* $\max(P_3) = 12$. *The BTG is constructed for* $P' := \otimes(P_1, P_2)$ *with 18 nodes. For each node, a value of up to 14 has to be stored for* P_3. *This can be done in 4 bits, so two values can be stored in one byte. The memory requirements then are given by* $18 * \frac{1}{2} = 9$ *bytes.*

6.3.3 Exploiting Presortings

In some cases, input data is sorted according to the level values for one or more WOPs of some Pareto preference P. For instance, this may be the case if the input is the result of a join operation (like e.g. sort-merge joins). We will now see how to exploit such prerequisites. Let's assume an input relation R that is sorted ascendingly according to the level values of a WOP P_1 in a Pareto preference P. Note that reordering the P_i does not change the semantics of P, so the following also holds for any other P_i.

We will detach P_1 from P and define $P' = \otimes(P_2, P_3, \ldots, P_m)$. R is split into a number of subsets R_v with $R_v = \{t | t \in R \wedge level_{P_1}(t) = v\}$. As R is already sorted by $level_{P_1}$, the R_v are automatically read one after each other (with ascending v).

The BTG of P' is constructed and kept in memory. For the first R_v (holding those tuples with the lowest occuring level value for P_1), the algorithm is executed just like the original *Hexagon* described in Section 6.1.1. The BMO set members are computed and can be returned, as they cannot be dominated by any subsequent tuple. Lemma 15 shows that every un-dominated tuple of one R_v cannot be dominated by an element of a subsequent R_w.

Lemma 15. *For dominance between partial BTGs and wrt the corresponding complete BTG, it holds that:*

a) A tuple $t \in R_v$ cannot be dominated by another tuple $t' \in R_w$ with $w > v$.

b) As all $t \in R_v$ are equivalent wrt P_1, we can ignore P_1 when evaluating P on R_v. Only P' needs to be evaluated.

c) For two values v and w, a node $(v, b_2, \ldots, b_m)$ dominates a node $(w, b_2, \ldots, b_m)$ iff $v < w$.

Proof. *Consider $t \in R_v$, $t' \in R_w$ with $w > v$:*

$$w > v \Rightarrow level_{P_1}(t') > level_{P_1}(t) \Rightarrow \neg(t <_P t')$$

This follows directly from the definitions of Pareto preferences (see Definition 15), just as items b) and c) do.

After one subset of R has been processed, we mark all *non-empty* nodes as *dominated*, as for a bigger value for v, these nodes are *dominated*. We then start reading the tuples of the next subset of R. Tuples belonging to dominated nodes can be discarded at once. Each time a subset has been processed completely, *Hexagon* is executed on the BTG that already holds information about dominated nodes and the BMO members of the previous R_v can be returned. The algorithm continues until either no more subsets of R need to be processed or the top node of P' is found being dominated. In the latter case, not even all input tuples would have to be read by *Hexagon*.

Example 39. *In the preference $P = \otimes(P_1, P_2, P_3)$, all P_i's are WOPs. They have maximum level values of 3, 1, and 2. Level values for tuples of a sample input relation that is sorted ascendingly by P_1's level value can be found in Table 6.3. The BTG for this preference has 18 nodes. Due to our optimization, we only have to keep the 6 nodes of the partial BTG for $P' = \otimes(P_2, P_3)$ in memory (see Figure 6.2 (a)).*

First, tuples t_1 and t_2 with a level value of 0 for P_1 will be inserted in the memory representation of the BTG. These tupels belong to R_0. The result can be seen in Figure 6.2 (b). Dominated nodes are omitted, non-empty nodes are shaded in gray.

The only (undominated) non-empty node is node $(0, 1, 1)$. Now the level value for P_1 increases and non-empty nodes are marked as dominated. The node $(0, 1, 1)$ is not represented in our new partial BTG anymore. We only find $(1, 1, 1)$, which is

r	id	$level_{P_1}(t)$	$level_{P_2}(t)$	$level_{P_3}(t)$
	t_1	0	1	2
	t_2	0	1	1
	t_3	1	0	2
	t_4	1	1	0
	t_5	2	0	0
	t_6	3	0	1

Table 6.3: Sample sorted input relation

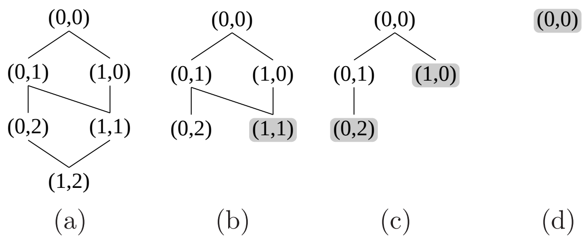

Figure 6.2: Partial BTGs during input processing

dominated by $(0,1,1)$*. There are only three empty, non-dominated nodes left. Figure 6.2 (c) shows the new partial BTG after processing all tuples with level value 1 for* P_1*. When* t_5*, represented by node* $(2,0,0)$ *has been read, the BTG looks like Figure 6.2 (d). Only the top node of the partial BTG is not dominated. Now, we do not have to continue the evaluation with more subsets of* R*. All nodes of* R_x *with* $x \geq 3$ *will be dominated by* t_5*.*

We will now generalize this strategy of exploiting presorted data to multiple attributes. After using data presorted in one dimension, we will now see how to utilize presortings in more than one dimension of an input relation.

First, we will discuss the case of two presorted dimensions P_1 and P_2. Let's assume the input relation is sorted wrt P_1, and tuples with equal level values for P_1 are sorted according to P_2. The BTG we keep in memory always represents the current partial BTG for one combination of level values for P_1 and P_2. While the level value for P_1 is constant, we follow exactly the algorithm for only one presorted domain, as it is irrelevant for evaluating P (see Lemma 15 (b)).

But when the level value for P_1 increases, a problem arises: no node in the current partial BTG will definitely dominate a node of the following one, as the following partial BTG will have a higher level value for P_1 and possibly a lower level value for P_2. In this case, corresponding nodes of the two partial BTGs will be indifferent. And from now on, we will always have to look for dominated notes at two previously handled partial BTGs: the one with the same level value for P_1 and a lower level value for P_2 and and the one with a lower level value for P_1 and the same level value for P_2.

Using more than two presorted dimensions would reduce memory requirements even more, but in this case, the problem of multiple relevant previous partial BTG scales up.

This leads the following theorem:

Theorem 24. *For a Pareto preference $P := \otimes(P_1, \dots, P_m)$ and an input relation R presorted according to $level_{P_1}$, ..., $level_{P_k}$, it holds:*

1. *We can reduce the number of nodes for the (then partial) BTG in memory to:*

$$\prod_{i=k+1}^{m} (\max(P_i) + 1) = \frac{nodes(BTG)}{\prod_{i=1}^{k} (\max(P_i) + 1)}$$

   Each of the partial BTGs can be identified by the combination of level values for the preferences $P_1, \dots, P_k$.

2. *When switching to the next partial BTG, we need to check k previously visited BTGs to identify dominated domains.*

Proof. *We will prove the parts of the theorem one by one.*

1. *For each of the level values for the $P_1, \dots, P_k$, there are $\max(P_i) + 1$ different level values. This is analogous to the size of the BTG.*
2. *Given a node $(a_{k+1}, \dots, a_m)$ of the partial BTG identified by the level combination $(a_1, \dots, a_k)$. The node is directly dominated by the top nodes of the partial BTGs identified by $(a_1 - 1, \dots, a_k)$, ..., $(a_1, \dots, a_k - 1)$. There are up to k such nodes.*

Managing the partial BTG for one presorted dimension is easy, as we have seen in Example 39. The first partial BTG that is looked at has no dominated nodes. In the next one, dominated and non-empty nodes from the last partial BTG are the dominated nodes. No additional computational or I/O costs arise.

Using two presorted dimensions, we have to provide two partial BTGs. One of them can be kept in memory. The other one has to be stored in external memory. Example 40 illustrates this problem.

Example 40. *Consider a Pareto preference P of four WOPs P_1 to P_4, all with maximum level values of 2. The input relation is presorted for P_1 and P_2, the partial BTG in memory consists of level values for P_3 and P_4. This means we have a BTG with 9 nodes in memory instead of a BTG with 81 nodes.*

A partial BTG is identified by the level values for P_1 and P_2. So $0,1$ *identifies the partial BTG for a level value of 0 for P_1 and 1 for P_2. We will have a look at the order of processed partial BTGs and the respectively required predecessors:*

partial BTG	*0,0*	*0,1*	*0,2*	*1,0*	*1,1*	*1,2*	*2,0*	*2,1*	*2,2*
req. for P_1	-	-	-	*0,0*	*0,1*	*0,2*	*1,0*	*1,1*	*1,2*
req. for P_2	-	*0,0*	*0,1*	-	*1,0*	*1,1*	-	*2,0*	*2,1*

We can draw some conclusions from Example 40. For P_2, the relevant previous partial BTG can be kept in main memory. Each time the level value for P_1 changes, the partial BTG in main memory is discarded. The partial BTGs required due to an increased level value for P_1 show exactly the same order as we use to visit the partial BTGs, although each of them is needed exactly $\max(P_2) + 1$ steps later. Each time a partial BTG (together with all tuples belonging to it) has been processed (and *non-empty* nodes have been marked as *dominated*), its complete state may be stored in serial order on a kind of external tape drive. Then, reading these partial BTGs one after each other later can easily be implemented.

When using k presorted dimensions, each time the partial BTG is switched, $k - 1$ previous BTGs have to be analyzed. With increasing k, this clearly becomes more and more inefficient. There is no way of storing them in the correct order as most of them are needed more than once. So sequential reading is not possible anymore. This is also shown in Example 41.

Example 41. *Consider the Pareto preference of Example 40, but this time with three presorted dimensions. The partial BTG identified by the level combination (1,1,1) requires information on the partial BTGs (0,1,1), (1,0,1), (1,1,0). The first of them is also needed when the partial BTGs identified by (0,2,1) and by (0,1,2) are processed. In general, each BTG will be needed three times and hence has to be kept in external memory at least twice.*

Efficient retrieval of node states of more than two previous partial BTGs can only be provided if the external memory provides random access performing as efficiently as serial access. For today's most common external memory type, hard disks, this is unfortunately not the case. With the increasing importance of *solid state drives*, this may change in the future.

cars	id	vendor	color	price	age	mileage
	1	Honda	green	5,000	14	100,000
	2	Dodge	blue	10,000	11	80,000
	3	Volkswagen	red	15,000	7	150,000
	4	BMW	red	20,000	7	100,000

Table 6.4: A sample car database

6.3.4 Data Structure and Query Modeling

In some cases, query modeling issues can change a query or the BTG relevant for its evaluation. We will study how to adjust the maximum level values for WOPs to the input data and how d-parameter manipulation can reduce BTG sizes. After that, we will see how to consider hard constraints given in the original query.

Reduction on Existing Values

In many cases, only a small number of domain values actually appear in a relation. For WOPs on numerical domains, this can have the effect that only some of the possible level values are met. Such a scenario is outlined in the following example:

Example 42. *Ringo is looking for a car. He has the following preference on the price:*

$$P_{Ringo} = AROUND_{1000}(price, 4000)$$

Applying P_{Ringo} to the car database of Table 6.4, we find that $\max(P_{Ringo}) = 16$, *even though there are only four different level values for the whole input relation: 1, 6, 11, and 16. A mapping of these values to 0, 1, 2, and 3 would reduce the size of every BTG for a preference containing P_{Ringo} to less than one fourth.*

One way to gain information of unused domain values could be the use of a histogram or a simple B-tree index. With this we can find existing level values for the corresponding WOP(s). We could efficiently reduce BTG sizes by "removing" (or more appropriate: preventing the existence of) unused nodes, leading to less memory needed and less effort for marking dominated empty nodes. It could be applied easily and with hardly any costs in cases in which histograms or B-tree indexes exist. The mapping of domain values to the few required level values could be done using minimal perfect hashing functions, leading to constant access times and minimum memory requirements ([18, 20, 29]).

d for P_1	$\Rightarrow \max(P_1)$	d for P_2	$\Rightarrow \max(P_2)$	BTG size
1	18000	1	50,000	$9.0 * 10^8$
50	360	100	500	$1.8 * 10^5$
100	180	1,000	50	$9.2 * 10^3$
200	90	10,000	5	550

Table 6.5: Impact of d-values on BTG sizes

Using the d-Parameter

Query composition is often not totally under the user's control. Many applications alter queries slightly to influence result set content and size. Setting d for some preference is done as a modeling decision for the application. One can reduce BTG sizes by slightly modifying user preferences, reducing the $\max(P_i)$.

If a d-parameter is very small, the range of input values is split into a large number of partitions. Increasing d decreases the number of partitions and hence the maximum level value. With some sharp maximum amount of main memory for the query evaluation, it could probably be reasonable to allow increases of d in a small range to enable usage of *Hexagon*. A case study in Example 43 gives an overview over the impact of the d-parameter.

Example 43. *George has the following preference on cars:*

$$\otimes(P_1, P_2) := \otimes(\; AROUND_d(price, 2,000), \; AROUND_d(mileage, 100,000))$$

With attribute ranges as seen in Table 6.4, different values for d strongly influence the BTG size. Table 6.5 shows different d-values and their impact on maximum level values and BTG sizes for the query. It becomes clear that in this use case, d values of 1 would lead to an incredibly big BTG. But already in line 2, when the d value flattens deviations of not more than 2.5% of the desired value, the BTG would fit into memory. Lines 3 and 4 show BTGs for d values of 5% resp. 10% of the query parameters. While such deviations still appear relatively small, the BTG gets as small as 550 nodes.

Another case of the potentially very high impact of very small changes in the values of d-parameters will be examined later in Example 50.

```
SELECT *
FROM cars
PREFERRING price AROUND 10000, 1000 AND
           color IN ('red', 'black')
BUT ONLY LEVEL(price) <= 5
```

Figure 6.3: PSQL statement with `BUT ONLY` clause

Quality Checks

In the syntax of Preference SQL, there is a `BUT ONLY` clause, evaluated after the preference selection. This clause can be used to remove elements from the BMO set with too great deviations from desired values. In Figure 6.3, we see such a PSQL statement. The maximum level value tolerated in the `BUT ONLY` clause for the preference on the attribute `price` is 5. Tuples with higher level values for `price` will not be in the result set even if they belong to the BMO set.

In *Hexagon*, this could be exploited to set the maximum level value for the preference on `price`. Tuples with higher level values could be discarded at once. Example 44 shows possible reductions of BTG sizes.

Example 44. *Consider the car database of Table 6.4 and the query defined in Figure 6.3. Without using the restrictions from the `BUT ONLY` clause, the maximum level value for the preference on `price` would be 10, leading to a BTG size of 22.*

Integrating the `BUT ONLY` clause into the execution of Hexagon*, the maximum level for `price` is only 5. The corresponding BTG has a size of 12, which is a reduction by approx. 45%.*

Please note that another but-only-filtering method has been described in [28]. There, BMO result set elements are enriched by a quality rating representing the degree of similarity to a possibly virtual perfect match. This practice follows methods of sales psychology. One key issue is the removal of tuples with too great a deviation from the desired value. This is called *but-only-filter*. When such a filter is known at query evaluation time in the database, the required quality could be translated in a maximum level value and exploited in the very same way as a PSQL `BUT ONLY` clause.

6.3.5 Combined Memory Saving Potential

We have seen different options to reduce memory consumption for *Hexagon*. A major advantage of the different optimization is that they are indepedent from each other and can therefore be combined for best results.

Without any preconditions, two of them can be applied in all cases: storing 5 nodes per byte and keeping the best value for P_m. Presorting in one dimension may be a by-product of a join of input data or some other database operation. They do not affect or even trade off each other's effect. An example of possible reductions of required memory will show the big potential of the different strategies and their combination.

Example 45. *We have a Pareto prefence of four WOPs with maximum level values (5, 10, 10, 100) and an input relation presorted wrt to P_1. Table 6.6 shows memory requirements following from the different applied optimization strategies from Section 6.3.1 (enhancing memory efficiency), Section 6.3.2 (storing the best value for one WOP), and Section 6.3.3 (exploitation of presortings). Without presorted data, we are able to reduce the required memory from approx. 18 KB to 872 B, i.e. to less than 5%. If we have presorted data, we can reduce it to 146 B, i.e. to less than 1%.*

Section	method description	nodes	memory req.
	usual BTG	73,326	18,332 B
6.3.1	memory efficient BTG	73,326	14,666 B
6.3.2	P_m detached	726	908 B
6.3.3	exploit presorting of P_1	12,221	3,056 B
combinations of methods from Sections			
6.3.1	& 6.3.2	726	872 B
6.3.1	& 6.3.3	12,221	2,445 B
6.3.2	& 6.3.3	121	152 B
6.3.1	& 6.3.2 & 6.3.3	121	146 B

Table 6.6: Results of the combination of memory saving techniques

6.4 Hexagon for Multi-Level BMO Sets and Top-k Queries

In some cases, not only the BMO set is of interest, but also the tuples directly dominated by those of the BMO set, the tuples directly dominated by the latters, and so on. Therefore, we extend the definition of the BMO set by a level value to form *multi-level BMO* sets, denoted as BMO_{ml} sets.

Definition 26. ***BMO set of level k***
The multi-level BMO set of level k, BMO_{ml}^{k}*, for a preference P and an input relation R is found by a preference selection* $\sigma^k[P](R)$*. It is defined by:*

$$\sigma^k[P](R) := \sigma[P](R \setminus \bigcup_{i=0}^{k-1}(\sigma^i[P](R)))$$

$\mathrm{BMO}_{ml}^{k_{max}}$ *denotes the non-empty* BMO_{ml} *set with the highest level. The size of* BMO_{ml}^{k} *is given by* $|\mathrm{BMO}_{ml}^{k}|$ *or* $|\sigma^k[P](R)|$*. For each tuple* $r \in R$*, the level of the BMO set it belongs to is denoted by the function value of* $ml_P : dom(A) \rightarrow \mathbb{N}_0$*.* □

Please note that $\sigma^0[P](R)$ is identical to the standard preference selection $\sigma[P](R)$ introduced in Definition 19. The concept of multi-level result sets for preference queries has also been described for Chomicki's *winnow* operator under the term of *iterated preferences* in [15]. It has been stated that all tuples in an input relation belong to a BMO_{ml} set of some level:

Theorem 25. *For each tuple in a relation R, there is exactly one* BMO_{ml}^{k} *set it belongs to:*

$$\forall r \in R : \big((\exists k : r \in \sigma^k[P](R)) \wedge (\nexists j \neq k : r \in \sigma^j[P](R))\big)$$

Proof. *A preference selection on a set of tuples never yields an empty result. For increasing values of k (starting with 0 and ending with* k_{max}*), the input relation diminishes as all selection results for smaller values for k are removed from the input:*

$$\begin{array}{rlcl} R \neq \emptyset \Rightarrow & \sigma[P](R) \neq \emptyset & & \textit{(see Definition 19)} \\ \Rightarrow & R \setminus \sigma[P](R) = R \setminus \sigma^0[P](R) & \subset & R \\ \Rightarrow & R \setminus \bigcup_{i=0}^{k}(\sigma^i[P](R)) \setminus \sigma[P](R) & \subset & R \setminus \bigcup_{i=0}^{k-1}(\sigma^i[P](R)) \end{array}$$

As R's size is finite, there has to be some k_{max} *for which the following holds:*

$$\bigcup_{i=0}^{k_{max}-1}(\sigma^i[P](R)) \subset R \wedge \bigcup_{i=0}^{k_{max}}(\sigma^i[P](R)) = R$$

So each tuple in R belongs a $\sigma^i[P](R)$*.*

So an order on R wrt a preference P is induced. The next theorem describes this order:

Theorem 26. *All elements of $\sigma^{k+1}[P](R)$ are dominated by elements of $\sigma^k[P](R)$ for all $k < k_{max}$:*

$$\forall a \in \sigma^{k+1}[P](R) : (\exists b \in \sigma^k[P](R) : a <_P b))$$

Proof. *Consider a tuple $a \in \sigma^{k+1}[P](R)$ that is not dominated by any element of $\sigma^k[P](R)$. Following Definition 26, $a \in \sigma^k[P](R)$. This is a contradiction. Theorem 26 holds.*

For every preference query on every non-empty relation, there is at least a BMO$_{ml}$ set of level 0. If it is the only one, no tuple in R is worse than any other wrt the preference. It is equally possible that all tuples in R belong to BMO$_{ml}$ sets of different levels. The preference then defines a total order on the elements of R.

For each node n in the BTG, we can determine the level of the BMO$_{ml}$ set it belongs to. Of course, all tuples in one equivalence class (which is represented by one node) are elements of the same BMO$_{ml}$ set. To find the BMO$_{ml}$ set for each node, we start at level 0 at the top node of the BTG. All tuples belonging to the standard BMO set have a BMO$_{ml}$ set level of 0. As the following lemma will show, the level of each tuple is the highest level of all tuples dominating it, increased by one:

Lemma 16. *BMO set level for a tuple*
For a tuple $r \in R$ (or the BTG node representing its level values), $ml_P(r)$ can be computed as follows:

$$ml_P(r) \quad := \quad \begin{cases} 0 & \Leftrightarrow \quad r \in \sigma^0[P](R) \\ 1 + \max(\{ml_P(s) | s \in R \wedge r <_P s\}) & \Leftrightarrow \quad r \notin \sigma^0[P](R) \end{cases}$$

Proof. *This follows directly from Definition 26 and Theorem 26.*

With these concepts for multi-level BMO sets in mind, we will see how to adjust *Hexagon* to be able to assign a BMO set level to each node of the BTG in the next section.

6.4.1 Multi-level Hexagon

We will now see how *Hexagon* can be adjusted to support BMO$_{ml}$ set queries. The algorithm we propose is called *Hexagon$_{ml}$*. Just as *Hexagon*, it consists of three phases, with the first two phases of the algorithm, *construction* and *adding*, remaining unchanged in *Hexagon$_{ml}$*. Modifications have to be done solely in the *removal phase*. Actually, as dominated nodes are not removed anymore, the *removal phase* is replaced by a *node classification phase*.

The only node states we need are *empty* and *non-empty* (enabling us to store one node state per bit), but we need to store a temporary value for the BMO$_{ml}$ set level a node belongs to.

The *classification phase* uses the same breadth-first walk as the original *Hexagon* does. When a node n is reached, we reset the temporary BMO$_{ml}$ set level tmp_{ml} of the nodes d_1, d_2, ..., that are directly dominated by n. The value $tmp_{ml}(d_i)$ for a node d_i is computed as follows:

$$tmp_{ml}(d_i) = \begin{cases} \max(tmp_{ml}(d_i), tmp_{ml}(n)) & \Leftrightarrow \quad n \text{ is empty} \\ \max(tmp_{ml}(d_i), tmp_{ml}(n)+1) & \Leftrightarrow \quad n \text{ is not empty} \end{cases}$$

When a node is visited, its value for tmp_{ml} is identifying the BMO$_{ml}$ set level it belongs to. For a more convenient and efficient access to each of the BMO$_{ml}$ sets after the removal phase, we generate a list of nodes belonging to each BMO$_{ml}$ set while walking through the BTG. This is analog to the single *next* relation for standard BMO evaluation after the removal phase. Each time a non-empty node is reached, it is appended to the respective list of nodes belonging to the same BMO$_{ml}$ set.

In Example 46, the evaluation of a preference on an input relation using *Hexagon$_{ml}$* is outlined.

Example 46. *We will revisit John's preference for cars (cp. Example 32):*

$$P_{John} := \otimes(BETWEEN_5(price; 60, 80), POS/NEG(color; \{red,\ blue\}, \{purple\})$$

For the input relation of Table 5.1, we find maximum level values of 3 and 2 for the contained WOPs. The BTG is shown in Figure 6.4(a), with non-empty nodes shaded in gray.

In the classification phase *of* Hexagon$_{ml}$*,* $(1,0)$ *is the first non-empty node we reach Hence it belongs to BMO$^0_{ml}$. We set the (temporary) BMO$_{ml}$ set level tmp$_{ml}$ of the dominated nodes* $(1,1)$ *and* $(2,0)$ *to 1. Getting to node* $(0,2)$*, we see that it also belongs to BMO$^0_{ml}$. Later, we reach node* $(1,2)$ *which has a tmp$_{ml}$ value of 1. So we set tmp$_{ml}$ for the dominated node* $(2,2)$ *to 2.*

Figure 6.4(b) shows the different BMO$_{ml}$ sets after the end of Hexagon$_{ml}$*'s breadth-first walk. Nodes belonging to the same BMO$_{ml}$ set are surrounded by gray areas.*

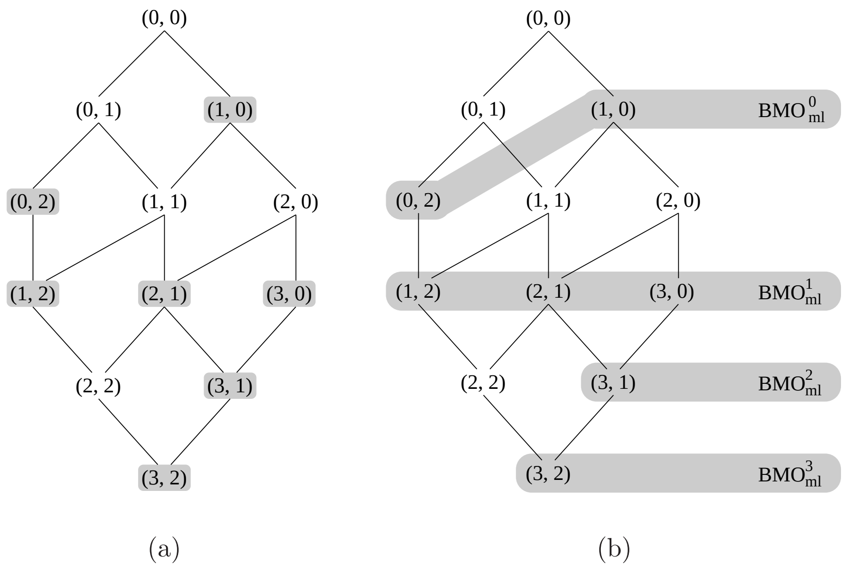

Figure 6.4: BTG for *Hexagon*$_{ml}$

Different approaches are possible to return the tuples. If only one specific BMO$_{ml}$ set is requested, it is most efficient to read the input relation from external memory and return only those tuples belonging to the specified set. To return more than one BMO$_{ml}$ set, it may be better to do a bucket sort on the input relation, with each BMO$_{ml}$ set level defining one bucket. Sorting and returning of the first requested set can easily be combined.

6.4.2 Top-k Queries

Although neither *Hexagon* nor *Hexagon*$_{ml}$ were designed to deal with top-k queries, we will now see that the latter seems to have been made for it. First, we have to see what the top-k approach, introduced in [7], is about.

Definition 27. ***Top-k Queries***
In a top-k *query on a data set D, each input value gets a quality rating of some kind. Only the k elements of D with the best quality values are returned as a query result.* □

We will now apply the principle of top-k queries to *Hexagon*$_{ml}$. Minor modifications will lead us to a new top-k algorithm.

Instead of dealing with node states, each node is represented by an integer counter, counting the number of tuples belonging to the node. During the *adding phase*, this counter is increased.

When all tuples are read, the classification is done just as described in Section 6.4.1. For each level of BMO$_{ml}$, we keep track of the number of tuples belonging to it. Each time the BMO set level for a non-empty node is determined, the number of tuples belonging to this BMO set is increased by the number of tuples belonging to the node.

After that, we start to determine what the *top k* results may be. It is reasonable to say that the top $|\sigma^0[P](R)|$ tuples are the best results for the query. We will then have to deal with one of the following three different cases in each query:

1. $|\sigma^0[P](R)| > k$:
   Not all elements of $\sigma^0[P](R)$ can be returned due to result set size limitations. When the tuples are re-read from external memory, the first k tuples belonging to $\sigma^0[P](R)$ are returned.

2. $|\sigma^0[P](R)| = k$:
   Probably the most unlikely case, but the easiest one: the result set is $\sigma^0[P](R)$, and all tuples belonging to it are returned as the result set. For this case, there is no difference between the BMO and the top-k result set.

3. $|\sigma^0[P](R)| < k$:
   The elements of $\sigma^0[P](R)$ are too few for an adequate answer. We have to find a value x which meets the following criteria:

   $$\left|\bigcup_{i=0}^{x-1} \sigma^i[P](R)\right| < k \leq \left|\bigcup_{i=0}^{x} \sigma^i[P](R)\right|$$

   That means, not only all elements of $\sigma^0[P](R)$ are returned, but also some of $\sigma^1[P](R)$ (if the number of result tuples is still lower than k, $\sigma^2[P](R)$ and so on may also be returned). Please note that this case is a generalization and covers cases 1 and 2 as well.

To return the correct number of results, we will loop through the different BMO$_{ml}$ set in order of their level and keep the sum of tuples belonging to them. We have to find the BMO$_{ml}$ sets that completely belong to the *top-k* results. In most cases, there will be one BMO$_{ml}$ set S only partially belonging to the *top-k* results. Of S, only so many tuples are to be returned that the total number of k results is matched.

After we have found S, we read the input relation once more. All tuples of BMO$_{ml}$ set with levels smaller than S are returned. For S, only the computed number of

i	0	1	2	3	4
nodes in $\sigma^i[P](R)$	(0,0,1) (0,1,0) (2,0,0)	(0,2,0) (1,0,1)	(2,0,1) (1,2,1) (2,2,0)	(2,1,1)	(2,2,1)
$\|\sigma^i[P](R)\|$	4	4	32	12	14

Table 6.7: Nodes and BMO$_{ml}$ set levels

tuples needed to get k results are returned. All other tuples are discarded. Example 47 will show the *Hexagon*$_{ml}$ with the *top-k* extension.

Example 47. *Consider a Pareto preference* $P := \otimes(P_1, P_2, P_3)$ *with maximum level values 2, 2, and 1. The BTG for this preference is given in Figure 6.5. The small index numbers next to each node show the number of tuples represented by each node. Nodes without index are empty. After reading all input tuples and classifying the nodes,* k *tuples should be returned. The different BMO*$_{ml}$ *set levels and their sizes can be found in Table 6.7. We will see what happens for different values of* k*. The three cases correspond to those described above:*

1. *k = 3:*
   Three of four tuples belonging to the nodes of $\sigma^0[P](R)$ *are returned.*

2. *k = 4:*
   $\sigma^0[P](R)$ *is returned.*

3. *k = 10:*
   $\sigma^0[P](R)$ *and* $\sigma^1[P](R)$ *are returned completely, leading to 8 tuples in the result set. Additionally,* $k - 8 = 2$ *tuples from* $\sigma^2[P](R)$ *are returned.*

Please note the proposed algorithm will not return tuples in a progressive way. A tuple with a higher overall level than another could be returned, just because of the order of the input relation. The next example will outline such a scenario:

Example 48. *We will work with the same preference and input as in Example 47. The top 1 result should be returned. Tuples belonging to the resp. nodes are read from external memory in the following order:*

$$(0,2,0),\ (2,1,1),\ (2,0,0),\ \ldots$$

The third tuple read is the first one in $\sigma^0[P](R)$*. As the top-1 query is looking for only one result, the algorithm will stop after reading (and returning)* $(2, 0, 0)$.

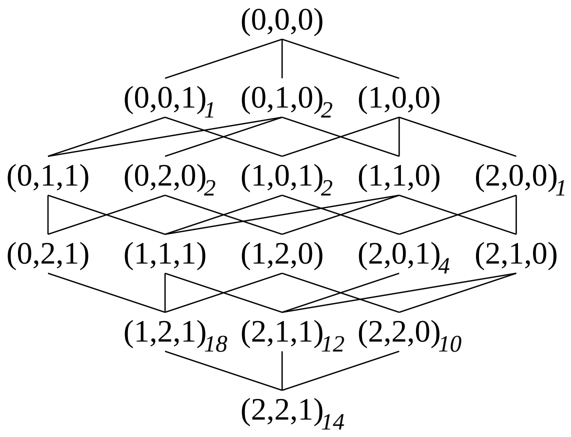

Figure 6.5: BTG for Example 47

Generally, tuples of higher level BMO sets may be returned earlier due to their position in the input relation. Result tuples could easily be ordered after they have been identified. In a reasonable query environment (where the k of the *top-k* query is much smaller than the size of the input relation), this sorting would take much less time than reading the complete input relation and therefore have hardly any impact on the overall performance.

Other approaches to pick the *top k* results from the different equivalence classes (i.e. node) are of course possible as well. Only those tuples belonging to some specific equivalence classes could be returned. The information on the number of tuples in the different equivalence classes may as well be another input to determine a weighting of the WOPs best suited to a user. By using this information, a number of different *top-k* queries could be executed with only a small need for computations. An additional weighting of the WOPs can be used to sort the nodes differently. Still, information on the BMO$_{ml}$ set level of a node can be used, taking it as some attribute result candidates are ordered by. Whichever additional conditions and characteristics are used, the *top k* results can be taken then from the nodes coming first in the new order, as Example 49 shows.

Example 49. *After the computations for answering the query in Example 47, some kind of presentation preference may induce a different weighting for the WOPs. P_1 is now more important than P_2 and P_3. Such presentation preferences often are added to user preferences in online shops ([28]).*

To answer this query, the set of non-empty nodes in the BTG has to be identified. Then, these nodes are ordered ascending wrt the level value for P_1. Nodes with equal level value for P_1 are ordered ascending wrt their BMO_{ml} set level. For the non-empty nodes of the BTG, this leads to the following order wrt the query (with best elements being on top):

$$\begin{gathered}
\{(0,0,1),\ (0,1,0)\},\\
\{(0,2,0)\},\\
\{(1,0,1)\},\\
\{(1,2,1)\},\\
\{(2,0,0)\},\\
\{(2,0,1),\ (2,2,0)\},\\
\{(2,1,1)\},\\
\{(2,2,1)\}
\end{gathered}$$

Nodes pooled in $\{,\}$ *remain unordered, due to Pareto preferences being strict partial orders ([15]).*

The nodes holding the top k *tuples than can now be identified by suming up the number of tuples belonging to each of the nodes. Using the values of Example 47 (and Figure 6.5), a top-5 query could be answered by returning all tuples of the nodes* $(0,0,1)$, $(0,1,0)$, *and* $(0,2,0)$.

Please note that a top-4 query would return all tuples of nodes $(0,0,1)$ *and* $(0,1,0)$, *but not all of* $(0,2,0)$.

As there are a lot of possibilies to impose an order on the nodes of the BTG, we omit the discussion of the effects of different order strategies here. We have seen that in spite of being developed for Pareto preference (or *skyline*) queries, *Hexagon$_{ml}$* can easily be applied to top-k queries as well. Its greatest advantage remains: the linear complexity on previously unknown input data.

Chapter 7

Performance

We have seen that *Hexagon* has a linear runtime behaviour with respect to the size of the input relation (for BTGs with fixed sizes, as stated in Theorem 23). While this is clearly superior to all other algorithms that do not need indexes, we have to investigate what this means in practice. We will now examine *Hexagon*'s performance experimentally compared to existing algorithms like *LESS* with and without *pruning* with a variety of Pareto preference queries and input relations. The results of the performance benchmarks will then be used to build a cost-based algorithm selector, promising the best performance for all kinds of Pareto preferences and input relations.

7.1 Benchmarks

In this section, we present the results of the experimental evaluation of our algorithms. We will use *LESS* as a yardstick for the performance of algorithms for *skyline* computations on generic data. The benchmark diagrams show the algorithm *LESS*$^{++}$. *LESS*$^{++}$ is *LESS* combined with the pruning techniques we have introduced in chapter 5. This implementation has constantly shown nearly equal or better runtimes than *LESS*. One use case with very unequal runtimes for the two algorithms has been presented in Example 31, where pruning enables a huge performance boost.

All algorithms were implemented in Java and run under Java 6. Our experiments have been run on a 1.7GHz Xeon running Windows 2003 Server. The system has a total amount of 12GB of RAM, although only small parts of it were used for each single test. As a result of the lack of direct control of memory usage in Java, we have restricted the number of tuples to be held in memory concurrently for nested-loop based algorithms. In our tests, 1,000 tuples were allowed. Assuming a typical size of 1 KB per tuple (as done in [6] and most of the following work) gives us virtually 1 MB of memory. This 1 MB is then used to determine the maximum size of a BTG we are able to keep in memory. Our tests have been carried out with the non-memory-optimized *Hexagon* with memory requirements as stated in Lemma 14. This means 2 bits are used to keep the current state of a node or up to four nodes can be stored in one byte. Thus, 1 MB of main memory would be capable of holding BTGs with over 4,000,000 nodes. This was the maximum size of BTGs used in our tests. Although we did the performance benchmarks without any of the memory optimizations of Section 6.3, the results hold for optimized versions of *Hexagon* as well, as the optimizations have hardly any effect on runtimes. Especially the detachment of one of the WOPs in a Pareto preference (cf. Section 6.3.2 has absolutely no effect on the algorithms overall performance. Also, runtimes for Java code are not directly comparable to runtimes for C code. Therefore, our test results show ratios rather than best achievable execution times using highly-optimized C code.

The data we have chosen for our experiments are synthetic. We use data with different distributions, in particular correlated, independent and anti-correlated. The test data domains and the test preferences were chosen to form two different BTG sizes:

- BTGs fitting into main memory: $\sim 10^6$ nodes
- BTGs too large for main memory: $\sim 10^9$ nodes

We have chosen different combinations of maximum level values to get 32 BTGs in each of the sizes for Pareto preferences with 3 up to 8 dimensions. We have created some Pareto preferences holding only WOPs with similar maximum level values and some with one WOP's maximum level value exceeding the others' maximum level values by far. As seen in BTG_{P_B} in Figure 4.6, the latter leads to BTGs with small widths and big heights. Note that these special preferences with one of the maximum level values being far bigger than the others is still different from the *LS* variant allowing one unrestricted attribute. Such an unrestricted attribute could still be integrated into our implementations.

Similar tests for *LS* and *LESS* have also been carried out in [47], where the experimental results clearly showed the superiority of *LS*. Due to the similarity of *LS* and *Hexagon*, we did not expect anything else.

We have carried out tests for BNL^{++}, *LESS*, $LESS^{++}$ (being *LESS* combined with the pruning mechanism also used in BNL^{++}), *Hexagon*, and the latter's external version using pruning, $Hexagon^{++}$. We have left out the performance data of *Hexagon* and $Hexagon^{++}$ for large BTGs for two reasons:

- BTGs with around 1 billion nodes require more main memory than we have used as buffer for the other algorithms.
- *Hexagon*$^{++}$'s performance is inadequate for BTGs in these sizes with input relations that are much smaller (in such cases, most of the BTG nodes are empty).

In general, the performance of *Hexagon*$^{++}$ has proven to be disappointing in all of our tests. The reason for this is that the algorithm needs input data to be sorted before it can start processing. Compared to other algorithms, the sorting procedure takes too long.

Apart from Figure 7.2, results for *LESS* are omitted as well, as they were found to be worse than those of *LESS*$^{++}$ by only some percentage points in nearly all test cases. Hence, plotting the execution times for both algorithms would only impair readability of those figures with a large range of values for execution times.

In Figure 7.1 and Figure 7.2, we see that *Hexagon* is running overwhelmingly faster than any other algorithm. Even the increasing number of domains does not really affect its execution times. With more domains, the number of incomparable tuples increases, just like the number of incomparable nodes in the BTG. For those algorithms based on tuple-to-tuple comparisons, this leads to an increase in the workload. As *Hexagon* only has to walk through the BTG once and this walk can be done in practically no time (compared to I/O costs), this is no surprise.

Please note that Figure 7.2 is showing the same results for *Hexagon* and *LESS*$^{++}$ as Figure 7.1. By omitting the "slow" algorithms *BNL*$^{++}$ and *Hexagon*$^{++}$, we are able to to add results for *LESS* that can be distinguished from those of *LESS*$^{++}$.

For the evaluation of preferences with BTGs that do not fit into main memory, *Hexagon* cannot be used. Hence our performance tests in these cases are restricted to *BNL*$^{++}$, *LESS*$^{++}$ and *Hexagon*$^{++}$. The results can be found in Figure 7.3, where *LESS*$^{++}$ shows the best results. For 8 dimensions, it takes only 10% of the runtime of the worst algorithm, *Hexagon*$^{++}$.

In our second test series (see Figure 7.4 to Figure 7.6), we focus on different input tuple distributions and input sizes. Please note that we have used logarithmic scale for both axes. With the linear runtime of nested-loop based algorithms for average cases like correlated data (cp. [31]), *Hexagon* has only slight performance advantages over *BNL*$^{++}$ and *LESS*$^{++}$ for correlated data. But anti-correlated data, *Hexagon* is even better by an order of magnitude. These relations have been shown to be quite similar for *LS* and *LESS* in [47]. The groundbreaking performance of *Hexagon* (and *LS*) obviously is not affected at all by any kind of data distribution.

We have also run some tests for input relations with 1 million and 10 million tuples.

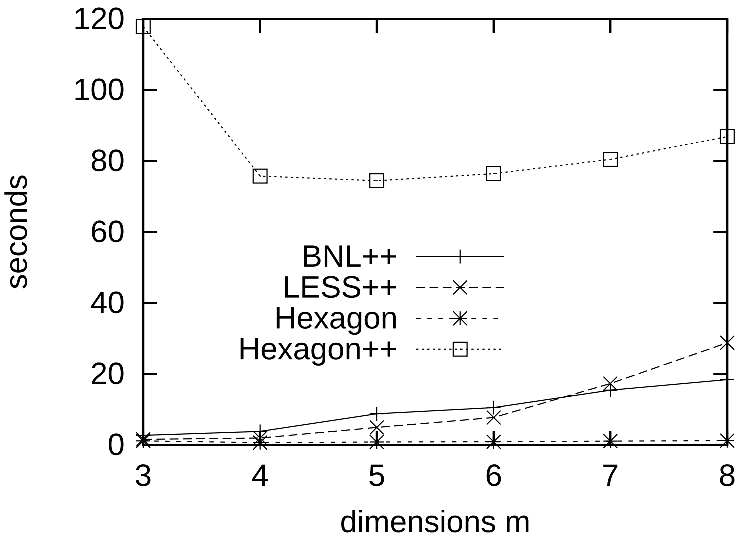

Figure 7.1: Performance for BTGs fitting in main memory (1)

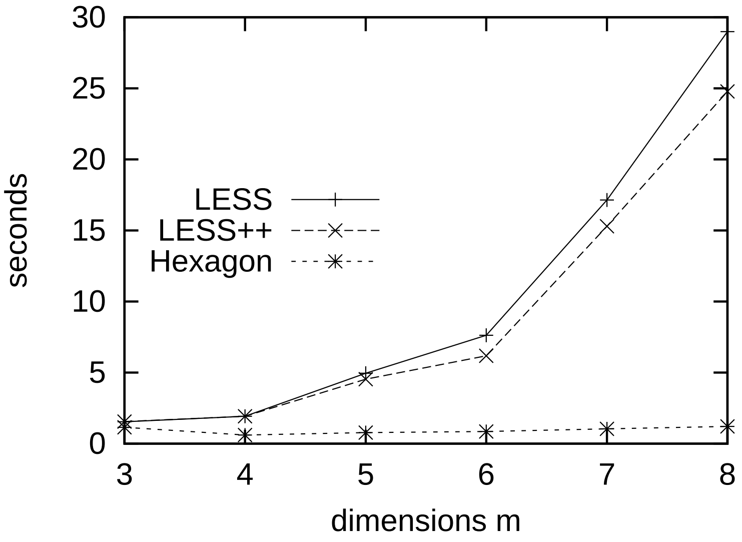

Figure 7.2: Performance for BTGs fitting in main memory (2)

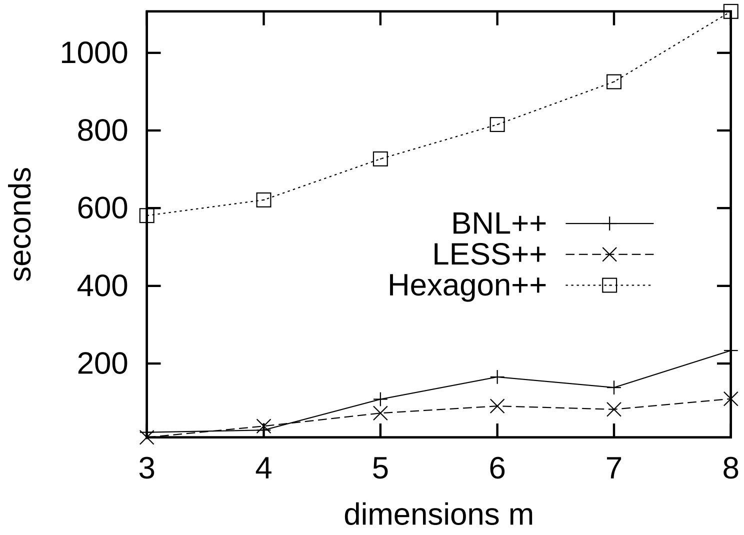

Figure 7.3: Performance for BTGs not fitting in main memory

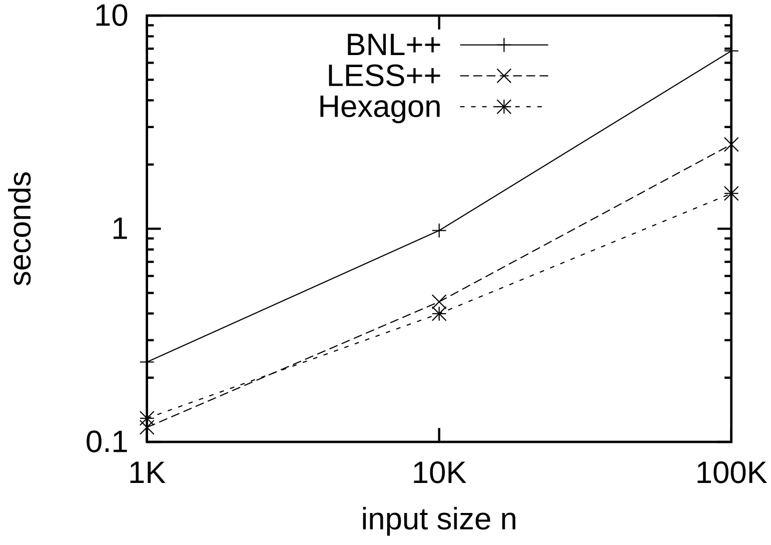

Figure 7.4: Performance for correlated data

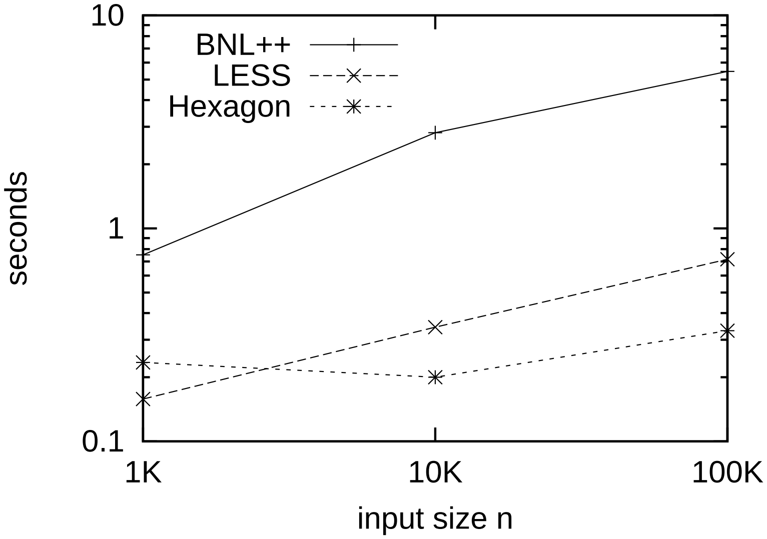

Figure 7.5: Performance for independent data

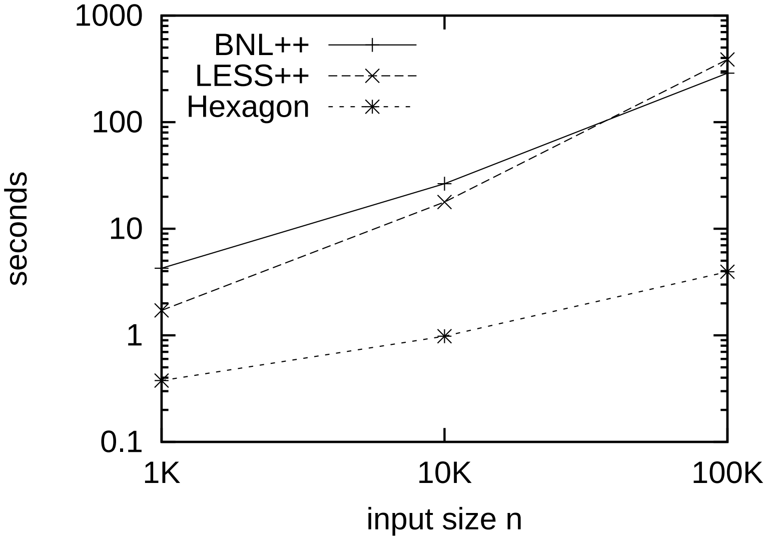

Figure 7.6: Performance for anti-correlated data

Hexagon needed 9.8 s to find the BMO set for 1 million tuples and 89.7 s for 10 million tuples. Still, in our test setup, *Hexagon* was able to process approximately 100,000 tuples per second. In other words, *Hexagon* is able to deal with half a million tuples in the reasonable time of 5 s. For comparison, *LESS*$^{++}$ took 89.2 s to process 1 million tuples and over 1,000 s for 10 million tuples, both times far beyond the latency a user accepts when querying a database.

In all our tests, *Hexagon* has proven to be the fastest algorithm available, sometimes by orders of magnitude. As we have also found out that up to 95% of *Hexagon*'s execution time was spent *reading* the input, we realized that finding the nodes belonging to the BMO took hardly any time at all. Even with the increasing computational effort resulting from the optimizations of Section 6.3.1 to Section 6.3.3, *Hexagon* will still be faster than any kind of other generic algorithm.

7.2 Cost-based Algorithm Selection

We will now use the knowledge we have gained in the performance benchmarks to find a formula helping a cost-based database optimizer to determine which algorithm to use for some (possibly unknown) input data.

Even for small input relations with only 1,000 tuples we have found *Hexagon* to be the fastest algorithm available. As a consequence, we propose to use *Hexagon* where possible, i.e. when there is enough available main memory. This seems especially reasonable as awkward input tuple distributions (e.g. anti-correlated data) are common in real applications. As we have already seen in the example used in Section 1.1, mileage and price of used cars are anti-correlated.

Another result from Section 7.1 is that *Hexagon*$^{++}$ has permanently performed worse than at least one of the other algorithms. This leads us to the following heuristics:

For a preference query on a relation R using a Pareto preference $P = \otimes(P_1, \ldots, P_m)$, we assume a dynamically allocated main memory buffer size of k bytes is available for query execution. We can formulate heuristics for algorithm selection in pseudo code as follows:

Preconditions: The P_i of P are sorted such that

- P_1 is the one P_i with the highest maximum level value that R is presorted for,
- P_m has the highest maximum level value of all P_i (except for P_1 iff R is sorted by $level_{P_1}$).

```
IF $R$ is sorted according to $level_{P_1}$
    LET $P^* = \otimes(P_2, \dots, P_{m-1})$
ELSE
    LET $P^* = \otimes(P_1, \dots, P_m)$
END IF
IF $k \geq \left\lceil \prod_{p \in P^*} (\max(p) + 1) * (\left\lceil \frac{\ln(\max(P_m))}{\ln 256} \right\rceil + \frac{1}{5}) \right\rceil$
   EVALUTE $P$ USING Hexagon
ELSE
   EVALUTE $P$ USING LESS$^{++}$
END IF
```

Note that the buffer size k is usually not a constant, but depends on the current state and workload of the database system and hence has to be identified for each query. The given decision formula applies the memory requirement formula from Section 6.3.2 together with the implementation technique from Section 6.3.1. This means that the preference with the highest maximum level value is detached. Each node of the reduced BTG holds the best value for this detached preference. Node states are stored as ternary numbers, leading to five node states per byte. The cost-based algorithm selection has been integrated into the *Preference Query Optimizer* of *Preference SQL* in [32]. There, it has shown its power working together with the algebraic optimization strategies of [34].

As we can see, the formula is completely independent of the number of tuples in the input relation R. The benchmarks of Section 7.1 showed that *Hexagon* is better than *LESS*$^{++}$ for all input sizes, so we need not take this feature into account.

Another case study will show how big the impact can be when, due to memory restrictions, a preference has to be evaluated using *LESS*$^{++}$ instead of *Hexagon*.

Example 50. *Consider a Pareto preference with* $m = 4$ *with all* $\max(P_i)$ *being 100. The available main memory buffer for query evaluation is 1 MB. The input relation* R *has 10,000 tuples that are not sorted according to any* $level_{P_i}$.

The algorithm selection yields $P^* = \otimes(P_1, \dots, P_{m-1})$*. The BTG we have to keep in main memory to use* Hexagon *has the following size:*

$$\left\lceil 101^3 * (1 + \frac{1}{5}) \right\rceil B = \frac{6 * 101^3}{5} B = 1,236,362 B = 1.18\, MB$$

This exceeds the available memory buffer of 1 MB, so LESS^{++} *has to be used. On our test system, evaluation of such a query took 35 s for anti-correlated data. If the input relation had been sorted wrt one* P_i *or the memory buffer had been bigger by only 20%,* Hexagon *could have been used – taking less than 3 s for the evaluation.*

As we have seen, small differences in BTG size may have a significant impact on the runtimes, in this case a 20% increase in main memory size would lead to a reduction in execution time of over 90%. As a result of this study, we can say that main memory extension can lead to significantly faster execution.

Chapter 8

Summary & Outlook

We will now summarize the results of this book, looking in particular at the main contributions to our field of research. The chapter, and with it this book, will conclude with proposals for more fields of applications for *Hexagon* and for possibilities that could be created by technological progress.

8.1 Summary

One of the main contributions of this book is the abstraction from the better-than relation defined by a preference to the *better-than graph*, being nothing more than its graphical representation. Thereby, all our further results were made possible. The most important achievements are:

- *BTG as a lattice*:
  We have shown that the BTG for a Pareto preference with nothing but WOPs as input preferences is a lattice. Together with our algorithm for embedding any kind of strict partial order in a distributive lattice, we can find a lattice-BTG for every kind of preference. While we have already exploited characteristics of lattices when we introduced *Hexagon*, this could lead to more unexpected optimizations coming from the field of lattice theory.
- *Pruning*:
  The identification of values that are definitively dominated by a given tuple without further need for comparisons is a completely new kind of optimization for Pareto preference evaluation and *skyline* algorithms. Moreover, it was found to be compatible with all the existing optimization techniques used in present algorithms.

- *Hexagon*:
  Not only has a new algorithm been introduced with *Hexagon*: it stands for a completely new kind of algorithm with groundbreaking performance on unknown data. For the first time ever, Pareto preference evaluation can be carried out in guaranteed linear time. This is obviously the optimum in the given case of previously unknown or even dynamically created input as e.g. table join results. The algorithm *Lattice Skyline* of [47], that was introduced independently and simultaneously, follows the same principles and it shows the same theoretical and practical performance power, but is not able to deal with more than one numerical domain. Still, in addition to the basic algorithm, we have not only presented a number of techniques to reduce the memory requirements of *Hexagon* that do not affect its computational complexity and effort, but also have shown how to adapt the algorithm to deal with multi-level BMO sets and *top-k* queries. The detachment of one of the WOPs to reduce memory complexity can also be used to make *Hexagon* capable of dealing with an unrestricted numeric element.

  But even without applying any of our optimizations, *Hexagon* can be used perfectly on common computer systems. A machine with 1 GB of free RAM is able to handle BTGs with up to 4 billion nodes. A Pareto preference query holding two categorical WOPs with maximum level values of 4 and two numerical WOPs with maximum level values of 100 leads to a BTG of under 65 KB. Thus, 1 GB of RAM is capable of holding over 16,000 BTGs of this size in parallel. Of course, BTG size and the number of queries handled in parallel trade off each other. But the above values clearly show that *Hexagon* is ready to be used with today's computers.

- *Algorithm selection*:
  Finally, we are the first to provide a simple but powerful heuristics for algorithm selection given constraints as the amount of available main memory or characteristics of data structure. Integration in a query optimizer like the one of *Preference SQL* (see [34]) can easily be achieved. In fact, optimization strategies for preference queries as in [34] and [15] only address algebraic issues, which usually form the first part of query optimization in database systems. We were – to our best knowledge – the first to deal with the second task of optimizers, the cost-based selection of the adequate algorithm for preference evaluation.

Summing up, we have used the abstraction provided by a graphical representation of a given strict partial order to form a new kind of optimization and a new type of algorithm for Pareto preference evaluation.

8.2 Future Work

The concepts behind *pruning* and *Hexagon* are very promising. The field of application for them seems to be incredibly wide and a complete analysis would be beyond the scope of this book. Hence, we can only outline some approaches that would be interesting to explore in the future.

In this book, we could only present *pruning* capabilities for single nodes. The ability to determine the *pruning* level for more than a single node could help to improve the performance of algorithms like BNL^{++} or $LESS^{++}$. Consider a Pareto preference consisting of two WOPs with maximum levels of 10. The BTG node $(0, 4)$ has a *pruning level* of 14 (and so has $(4, 0)$). Looking at $(0, 4)$ and $(4, 0)$ together, we find a *pruning level* of 7.

Hexagon has shown great performance when it was used for traditional Pareto preference or *skyline* queries. As we have also seen, multi-level BMO sets can also be computed with hardly any additional effort. There are a number of closely related problems, for which *Hexagon* could also be adapted:

- *Skyband queries* ([50]):
  The *k-skyband* of a query is the number of tuples that is not dominated by more than k other tuples. Hence this forms some combination of classical *skyline* and *top-k* queries.

- *K-dominating queries* ([50]):
  For each tuple, the number of tuples it dominates is computed as a kind of quality measure. Then, the k tuples dominating the largest numbers of other tuples is chosen as result set.

- *Skylines on data streams or temporary data* ([46, 48]):
  For finding the best matches for some preference in a data stream, the algorithm has to keep track of the last n tuples read from the stream. On demand, it has to find the BMO set of these n tuples. Temporary data is a generalization of this, as each tuple that is read has its own specific period of validity. When the BMO set is requested, only those tuples valid at the moment must be considered.

- *Convex hulls* ([4]):
  Interpreting a tuple in a database relation as a point in a vector space, *Hexagon* could be the base for a high-performance algorithm to determine the convex hull of some tuples. Nowadays, good algorithms are merely known for two or three dimensions (e.g. Chan's algorithm in [11]), not for the general problem.

Despite its very good performance as a generic algorithm, *Hexagon* cannot compete with progressive algorithms regarding the time elapsed to return the first result. Still, with the ability to process unknown input data in linear time, *Hexagon* also has some advantages over progressive algorithms that need some kind of index structure on the input data. And as we have seen, in case of presorted domains, *Hexagon* does not need to read through the whole set of input tuples to return the first query results. So in addition to memory savings, simple indexes on some attributes could be used to create a progressive version of *Hexagon*.

Apart from that, hardware development could lead to better applicability of the existing algorithm and the optimizations for it. The most straight-forward approach is about RAM. Clearly, more available main memory leads to bigger (or more) BTGs that can be held in memory. As the price per GB of RAM seems to follow *Moore's law*[1], we can expect future computers to have much more main memory at the same price. So technological progress alone will improve *Hexagon*'s applicability.

Another promising approach could be the removal of some of the processing work from the CPU. The main part of the computation is done walking the BTG to identify dominated nodes. This part only uses integer numerics, a field in which graphics processing units have huge performance advantages over common multi-purpose CPUs. Therefore, massive parallelizing of the algorithm would be neccessary. Experimental results for performance gains of up to 500% for numerical computations can be found in [5, 35, 57].

Exploiting presortings (Section 6.3.3) may become interesting for multiple domains as well once *solid state drives* become more and more common. At the moment, presorting can only be efficiently exploited for one or two dimensions, as for more dimensions we need external memory having random access times that are not significantly slower than sequential access times. Conventional hard disk drives cannot provide this at the moment, and due to their physical design probable never will. *Solid state drives*, based on flash memory, already show very good access times and hence could offer sufficient performance. With decreasing prices and increasing deployment (leading SSD manufacturer Toshiba expects market share growths of over 300% in the next few years, as stated in its presentation "Strategies for growth: 2008"[2]), multi-dimensional presorting could be the method of choice in some years. As we have seen, most of the evaluation time of *Hexagon* is used to read the input. So improving I/O times for external memory will lead directly to improved execution times of the algorithm.

With both memory and computational capacity of mobile devices increasing more and more, the evaluation of preference queries on mobile phones or PDAs becomes an option, too. Apple's IPhone for example comes with a 412 MHz processor and

[1]http://www.intel.com/pressroom/kits/events/moores_law_40th/

[2]http://www.toshiba.co.jp/about/ir/en/library/pr/pdf/tpr20080508e.pdf

128 MB of RAM[3]. With only half of it being addressable by non-operating-system applications, the remaining 64 MB are enough to handle BTGs in sizes up to 260 million nodes – without applying any of the optimizations of Section 6.3.

All in all, we have seen different strategies to improve the performance of classic Pareto preference queries. The guaranteed linear complexity is very big step forward in opening the doors for a broad support of Pareto preference queries in common database management systems.

[3]http://www.semiconductor.com/resources/reports_database/view_device.asp?sinumber=18016

Bibliography

[1] *Proceedings of the 21st International Conference on Data Engineering, ICDE 2005, 5-8 April 2005, Tokyo, Japan.* IEEE Computer Society, 2005.

[2] *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, México.* IEEE, 2008.

[3] Rakesh Agrawal and Edward L. Wimmers. A Framework for Expressing and Combining Preferences. In Chen et al. [13], pages 297–306.

[4] Christian Böhm and Hans-Peter Kriegel. Determining the Convex Hull in Large Multidimensional Databases. In *Data Warehousing and Knowledge Discovery*, volume Volume 2114/2001, pages 294–306. Springer Berlin / Heidelberg, 2001.

[5] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröder. Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid. *ACM Transactions on Graphics*, 22:917–924, 2003.

[6] Stephan Börzsönyi, Donald Kossmann, and Konrad Stocker. The Skyline Operator. In *Proceedings of the 17th International Conference on Data Engineering*, pages 421–430. IEEE Computer Society, 2001.

[7] Michael J. Carey and Donald Kossmann. On Saying “Enough Already!” in SQL. In Joan Peckham, editor, *SIGMOD Conference*, pages 219–230. ACM Press, 1997.

[8] Donald D. Chamberlin and Raymond F. Boyce. SEQUEL: A Structured English Query Language. In Randall Rustin, editor, *Proceedings of 1974 ACM-SIGMOD Workshop on Data Description, Access and Control, Ann Arbor, Michigan, May 1-3, 1974, 2 Volumes*, pages 249–264. ACM, 1974.

[9] Chee Yong Chan, Pin-Kwang Eng, and Kian-Lee Tan. Efficient Processing of Skyline Queries with Partially-ordered Domains. In *ICDE* [1], pages 190–191.

[10] Chee Yong Chan, Pin-Kwang Eng, and Kian-Lee Tan. Stratified Computation of Skylines with Partially-ordered Domains. In Fatma Özcan, editor, *SIGMOD Conference*, pages 203–214. ACM, 2005.

[11] Timothy M. Chan. Optimal Output-sensitive Convex Hull Algorithms in two and three Dimensions. *Discrete & Computational Geometry*, 16:361–368, 1996.

[12] Yuan-Chi Chang, Lawrence D. Bergman, Vittorio Castelli, Chung-Sheng Li, Ming-Ling Lo, and John R. Smith. The Onion Technique: Indexing for Linear Optimization Queries. In Chen et al. [13], pages 391–402.

[13] Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein, editors. *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*. ACM, 2000.

[14] Jan Chomicki. Querying with Intrinsic Preferences. In Christian S. Jensen, Keith G. Jeffery, Jaroslav Pokorný, Simonas Saltenis, Elisa Bertino, Klemens Böhm, and Matthias Jarke, editors, *Advances in Database Technology – 8th International Conference on Extending Database Technology*, volume 2287 of *Lecture Notes in Computer Science*, pages 34–51. Springer, 2002.

[15] Jan Chomicki. Preference Formulas in Relational Queries. *ACM Transactions on Database Systems*, 28(4):427–466, 2003.

[16] Jan Chomicki, Parke Godfrey, Jarek Gryz, and Dongming Liang. Skyline with Presorting. In Umeshwar Dayal, Krithi Ramamritham, and T. M. Vijayaraman, editors, *Proceedings of the 19th International Conference on Data Engineering (ICDE'03)*, pages 717–816. IEEE Computer Society, 2003.

[17] Jan Chomicki, Parke Godfrey, Jarek Gryz, and Dongming Liang. Skyline with Presorting: Theory and Optimizations. In Mieczyslaw A. Klopotek, Slawomir T. Wierzchon, and Krzysztof Trojanowski, editors, *Intelligent Information Systems*, Advances in Soft Computing, pages 595–604. Springer, 2005.

[18] Richard J. Cichelli. Minimal Perfect Hash Functions Made Simple. *Communications of the ACM*, 23(1):17–19, 1980.

[19] Bin Cui, Hua Lu, Quanqing Xu, Lijiang Chen, Yafei Dai, and Yongluan Zhou. Parallel Distributed Processing of Constrained Skyline Queries by Filtering. In *ICDE* [2], pages 546–555.

[20] Zbigniew J. Czech, George Havas, and Bohdan S. Majewski. An Optimal Algorithm for Generating Minimal Perfect Hash Functions. *Information Processing Letters*, 43:257–264, 1992.

[21] Gautam Das, Dimitrios Gunopulos, Nick Koudas, and Dimitris Tsirogiannis. Answering Top-k Queries Using Views. In Dayal et al. [23], pages 451–462.

[22] Brian A. Davey and Hilary A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, Cambridge, UK, 2nd edition, 2002.

[23] Umeshwar Dayal, Kyu-Young Whang, David B. Lomet, Gustavo Alonso, Guy M. Lohman, Martin L. Kersten, Sang Kyun Cha, and Young-Kuk Kim, editors. *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*. ACM, 2006.

[24] Sven Döring, Stefan Fischer, Werner Kießling, and Timotheus Preisinger. Evaluation and Optimization of the Catalog Search Process of E-Procurement Platforms. *Journal of Electronic Commerce Research and Applications*, 5(1):44–56, 2006.

[25] Ronald Fagin. Fuzzy Queries in Multimedia Database Systems. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 1–10. ACM Press, 1998.

[26] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal Aggregation Algorithms for Middleware. *Journal of Computer and System Sciences*, 66(4):614–656, 2003.

[27] Boi Faltings, Peal Pu, and Jiyoung Zhang. Agile Preference Models based on Soft Constraints. In *AAAI Spring Symposium - Technical Report*, volume SS-05-02, pages 23–28. American Association for Artificial Intelligence, Menlo Park, CA 94025-3496, United States, 2005. Artificial Intelligence Laboratory, Ecole Polytechnique Fédérale de Lausanne (EPFL), CH-1015 Lausanne, Switzerland.

[28] Stefan Fischer, Werner Kießling, and Timotheus Preisinger. Preference based Quality Assessment and Presentation of Query Results. In Gloria Bordogna and Giuseppe Psaila, editors, *Flexible Databases Supporting Imprecision and Uncertainty*, pages 91–121. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[29] Edward A. Fox, Lenwood S. Heath, Qi Fan Chen, and Amjad M. Daoud. Practical Minimal Perfect Hash Functions for Large Databases. *Communications of the ACM*, 35(1):105–121, 1992.

[30] Parke Godfrey. Skyline Cardinality for Relational Processing. In Dietmar Seipel and Jose Maria Turull Torres, editors, *Foundations of Information and Knowledge Systems, Third International Symposium, FoIKS 2004, Wilhelminenburg Castle, Austria, February 17-20, 2004, Proceedings*, volume 2942 of *Lecture Notes in Computer Science*, pages 78–97. Springer, 2004.

[31] Parke Godfrey, Ryan Shipley, and Jarek Gryz. Algorithms and Analyses for Maximal Vector Computation. *The VLDB Journal*, 16(1):5–28, 2007.

[32] Alexander Gribov. Implementierung und Evaluierung einer kostenbasierten Query-Optimierung in Preference SQL. Diploma Thesis, Universität Augsburg, Augsburg, Germany, February 2009.

[33] Ulrich Güntzer, Wolf-Tilo Balke, and Werner Kießling. Optimizing Multi-Feature Queries for Image Databases. In Amr El Abbadi, Michael L. Brodie, Sharma Chakravarthy, Umeshwar Dayal, Nabil Kamel, Gunter Schlageter, and Kyu-Young Whang, editors, *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, pages 419–428. Morgan Kaufmann, 2000.

[34] Bernd Hafenrichter and Werner Kießling. Optimization of Relational Preference Queries. In Hugh E. Williams and Gillian Dobbie, editors, *Database Technologies 2005, Proceedings of the Sixteenth Australasian Database Conference, ADC 2005, Newcastle, Australia, January 31st - February 3rd 2005*, volume 39 of *CRPIT*, pages 175–184. Australian Computer Society, 2005.

[35] Karl E. Hillesland, Sergey Molinov, and Radek Grzeszczuk. Nonlinear Optimization Framework for Image-based Modeling on Programmable Graphics Hardware. In *International Conference on Computer Graphics and Interactive Techniques: ACM SIGGRAPH 2005 Courses*, page 224, New York, NY, USA, 2005. ACM.

[36] Vagelis Hristidis, Nick Koudas, and Yannis Papakonstantinou. PREFER: A System for the Efficient Execution of Multi-parametric Ranked Queries. In *SIGMOD Conference*, pages 259–270, 2001.

[37] Werner Kießling. Foundations of Preferences in Database Systems. In *VLDB 2002, Proceedings of 28th International Conference on Very Large Data Bases, August 20-23, 2002, Hong Kong, China*, pages 311–322, 2002.

[38] Werner Kießling. Preference Queries with SV-Semantics. In Jayant R. Haritsa and T. M. Vijayaraman, editors, *Advances in Data Management 2005, Proceedings of the Eleventh International Conference on Management of Data, January 6, 7, and 8, 2005, Goa, India*, pages 15–26. Computer Society of India, 2005.

[39] Werner Kießling, Bernd Hafenrichter, Stefan Fischer, and Stefan Holland. Preference XPath: A Query Language for E-Commerce. In Hans U. Buhl, Andreas Huther, and Bernd Reitweisner, editors, *5th International Conference on Wirtschaftsinformatik – Information Age Economy, Augsburg, Germany*, pages 427–440, Heidelberg, 2001. Physica-Verlag.

[40] Werner Kießling and Gerhard Köstler. Preference SQL - Design, Implementation, Experiences. In *VLDB 2002, Proceedings of 28th International Conference on Very Large Data Bases, August 20-23, 2002, Hong Kong, China*, pages 990–1001, 2002.

[41] Christoph Koch, Johannes Gehrke, Minos N. Garofalakis, Divesh Srivastava, Karl Aberer, Anand Deshpande, Daniela Florescu, Chee Yong Chan, Venkatesh Ganti, Carl-Christian Kanne, Wolfgang Klas, and Erich J. Neuhold, editors. *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*. ACM, 2007.

[42] Donald Kossmann, Frank Ramsak, and Steffen Rost. Shooting Stars in the Sky: An Online Algorithm for Skyline Queries. In *VLDB 2002, Proceedings of 28th International Conference on Very Large Data Bases, August 20-23, 2002, Hong Kong, China*, pages 275–286. Morgan Kaufmann, 2002.

[43] Georgia Koutrika and Yannis E. Ioannidis. Personalized Queries under a Generalized Preference Model. In *ICDE* [1], pages 841–852.

[44] H. T. Kung, Fabrizio Luccio, and Franco P. Preparata. On Finding the Maxima of a Set of Vectors. *Journal of the ACM*, 22(4):469–476, 1975.

[45] Ken Lee, Baihua Zheng, Huajing Li, and Wang-Chien Lee. Approaching the Skyline in Z Order. In Koch et al. [41], pages 279–290.

[46] Xuemin Lin, Yidong Yuan, Wei Wang, and Hongjun Lu. Stabbing the Sky: Efficient Skyline Computation over Sliding Windows. *Data Engineering, International Conference on*, 0:502–513, 2005.

[47] Michael Morse, Jignesh M. Patel, and H. V. Jagadish. Efficient Skyline Computation over Low-Cardinality Domains. In Koch et al. [41], pages 267–278.

[48] Michael D. Morse, Jignesh M. Patel, and William I. Grosky. Efficient Continuous Skyline Computation. *Information Sciences*, 177(17):3411–3437, 2007.

[49] Dimitris Papadias, Yufei Tao, Greg Fu, and Bernhard Seeger. An Optimal and Progressive Algorithm for Skyline Queries. In Alon Y. Halevy, Zachary G. Ives, and AnHai Doan, editors, *SIGMOD Conference*, pages 467–478. ACM, 2003.

[50] Dimitris Papadias, Yufei Tao, Greg Fu, and Bernhard Seeger. Progressive Skyline Computation in Database Systems. *ACM Transactions on Database Systems (TODS)*, 30(1):41–82, 2005.

[51] Timotheus Preisinger and Werner Kießling. The Hexagon Algorithm for Evaluating Pareto Preference Queries. In *Proceedings of the 3rd Multidisciplinary Workshop on Advances in Preference Handling (in conjunction with VLDB 2007), September 23*, 2007.

[52] Timotheus Preisinger, Werner Kießling, and Markus 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.

[53] Dimitris Sacharidis, Stavros Papadopoulos, and Dimitris Papadias. Topologically sorted skylines for partially ordered domains. In *Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, March 29 2009 - April 2 2009, Shanghai, China*, pages 1072–1083. IEEE, 2009.

[54] Florian Schleuter. Entwicklung einer Anwendung zur Visualisierung und Analyse von Better-Than-Graphen. Bachelor Thesis, Universität Augsburg, Augsburg, Germany, September 2008.

[55] Kian-Lee Tan, Pin-Kwang Eng, and Beng Chin Ooi. Efficient Progressive Skyline Computation. In Peter M. G. Apers, Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Kotagiri Ramamohanarao, and Richard T. Snodgrass, editors, *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, pages 301–310. Morgan Kaufmann, 2001.

[56] Yufei Tao, Xiaokui Xiao, and Jian Pei. Efficient Skyline and Top-k Retrieval in Subspaces. *IEEE Transactions on Knowledge and Data Engineering*, 19(8):1072–1088, 2007.

[57] Chris J. Thompson, Sahngyun Hahn, and Mark Oskin. Using Modern Graphics Architectures for General-purpose Computing: a Framework and Analysis. In *MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 306–317, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[58] Riccardo Torlone and Paolo Ciaccia. Finding the Best when it's a Matter of Preference. In *Decimo Convegno Nazionale su Sistemi Evoluti per Basi di Dati (SEBD 2002)*, pages 347–360, 2002.

[59] Dong Xin, Chen Chen, and Jiawei Han. Towards Robust Indexing for Ranked Queries. In Dayal et al. [23], pages 235–246.

[60] Lei Zou and Lei Chen. Dominant Graph: An Efficient Indexing Structure to Answer Top-K Queries. In *ICDE* [2], pages 536–545.

List of Figures

List of Tables

Curriculum Vitae

Timotheus Preisinger
Universitätsstraße 6a
86159 Augsburg
Germany

Personal Data	
Date of birth	November 13, 1979
Citizenship	German
Education	
March 2004 – July 2009	Doctorate, University of Augsburg, Germany
January 2004	Diploma in Applied Computer Science University of Augsburg, Germany
Academic Positions	
March 2004 – June 2009	Researcher, University of Augsburg, Germany