

# **UNDERSTANDING AND UTILIZING USER PREFERENCES**

by

**YU PENG**

A Thesis Submitted to  
The Hong Kong University of Science and Technology  
in Partial Fulfillment of the Requirements for  
the Degree of Doctor of Philosophy  
in Computer Science and Engineering

June 2012, Hong Kong

## Authorization

I hereby declare that I am the sole author of the thesis.

I authorize the Hong Kong University of Science and Technology to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the Hong Kong University of Science and Technology to reproduce the thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

---

YU PENG

11 June 2012

# UNDERSTANDING AND UTILIZING USER PREFERENCES

by

YU PENG

This is to certify that I have examined the above Ph.D. thesis  
and have found that it is complete and satisfactory in all respects,  
and that any and all revisions required by  
the thesis examination committee have been made.

---

DR. RAYMOND CHI-WING WONG, THESIS SUPERVISOR

---

DR. WILFRED NG, ACTING HEAD OF DEPARTMENT

Department of Computer Science and Engineering

11 June 2012

## ACKNOWLEDGMENTS

First of all, I thank my supervisor, Dr. Raymond Chi-Wing Wong. He is a wonderful supervisor who has helped me and inspired me for the past four years. He showed me the way to become an independent researcher: to identify a research problem and solve it independently. The most unforgettable moment was, when I was rejected by conferences consecutively, I felt so depressed and almost lost my faith. He wrote me a long email and told me how to put my heart into peace and encouraged me to restart again. I felt enormous relief and realized the importance of faith, courage, and hope.

I would like to express my appreciation to the members of my thesis committee, Prof. Nevin L. Zhang, Prof. Lei Chen, Prof. Weichuan Yu and Prof. Hua Lu, for their kindness in sitting on this committee and reading through my thesis. I would also like to thank Prof. Wilfred Ng and Prof. Dik Lun Lee for being committee members for my thesis proposal defense.

I thank all my co-authors, namely Qian Wan, Liangliang Ye, Prof. Philip Yu., Prof. Ihab F. Ilyas, Prof. Tamer Özsu and all my colleagues Lian Liu, Cheng Long, Peng Peng and Bin Zhang. Without their help, these papers would never have come into being. I am very lucky to meet my dear friends in HKUST. Life together with them has been very enjoyable. Special gratitude is given to Xiaoheng Xie, Jun Luo and Ang Li. They have provided generous help to me in various ways.

Finally, I would like to thank my parents and my grandmother, for their endless love, care and support to me. To them I dedicate this thesis.

# TABLE OF CONTENTS

<b>Title Page</b>	<b>i</b>
<b>Authorization Page</b>	<b>ii</b>
<b>Signature Page</b>	<b>iii</b>
<b>Acknowledgments</b>	<b>iv</b>
<b>Table of Contents</b>	<b>v</b>
<b>List of Figures</b>	<b>ix</b>
<b>List of Tables</b>	<b>xi</b>
<b>Abstract</b>	<b>xii</b>
<b>Chapter 1 Introduction</b>	<b>1</b>
1.1 Introduction to User Preferences	2
1.2 Understanding User Preferences	4
1.3 Utilizing User Preferences	5
1.4 Contribution	7
1.5 Organization	9
<b>Chapter 2 Related Work</b>	<b>10</b>
2.1 Subsequence Matching and Mining	10
2.2 Skyline	12
<b>Chapter 3 Attribute-based Subsequences Matching and Mining</b>	<b>15</b>
3.1 Motivation	15

3.2 Problem Definition and Preliminaries	17
3.3 Modular-based Algorithm	22
3.3.1 Requirement Value Matching	24
3.3.2 Requirement Sequential Order	26
3.3.3 Comparison	34
3.3.4 Putting Two Requirements Together	35
3.3.5 Phase Preprocessing	35
3.3.6 Phase Query	36
3.4 Frequent Attribute-based Subsequence Mining	37
3.5 Experiment Results	39
3.5.1 Effects of $n, \mu, k, d$ and $m$ on Synthetic Datasets	41
3.5.2 Results on Real Datasets	42
3.5.3 Results for FASM	46
3.6 Conclusions and Future Directions	48
<b>Chapter 4 Finding Top-<math>k</math> Profitable and Top-<math>k</math> Popular Products</b>	<b>50</b>
4.1 Preliminaries	50
4.1.1 Background: Skyline	50
4.1.2 Notations	52
4.2 Finding Top- $k$ Profitable Products over Static Datasets	53
4.2.1 Motivation	53
4.2.2 Problem Definition	55
4.2.3 Finding Optimal Price Assignment	57
4.2.4 Dynamic Programming Approach	62
4.2.5 Greedy Algorithms	65
4.2.6 Extension with the $h$ -dominance Constraint	68
4.3 Finding Top- $k$ Profitable Products over Dynamic Datasets	69
4.3.1 Insertion into $P$	70

4.3.2	Deletion from $P$	72
4.4	Finding Top- $k$ Popular Products	72
4.4.1	Motivation	73
4.4.2	Problem Definition	74
4.4.3	Our Method	76
4.5	Experimental Settings	77
4.5.1	Synthetic Datasets	77
4.5.2	Real Datasets	78
4.6	Results for TPP over Static Datasets	79
4.6.1	Results over Small Synthetic Datasets	80
4.6.2	Results over Large Synthetic Datasets	80
4.6.3	Results over Real Datasets	83
4.7	Results of TPP over Dynamic Data	86
4.8	Results for Finding Top- $k$ Popular Products	87
4.9	Conclusions	89
<b>Chapter 5</b>	<b>Finding Competitive Price</b>	<b>90</b>
5.1	Motivation	90
5.2	Problem Definition	94
5.3	Spatial Approach	98
5.3.1	Notations	99
5.3.2	Properties	100
5.3.3	Algorithm	101
5.3.4	Detailed Steps and Theoretical Analysis	102
5.4	Discussion	110
5.4.1	Handling Multiple Non-spatial Attributes	110
5.4.2	How to set $K$	113
5.5	Empirical Studies	116

5.5.1	Scalability	119
5.5.2	Case Study	123
5.5.3	Experiments for Determining An Appropriate Value of $K$	125
5.5.4	Summary	126
5.6	Conclusions	126
<b>Chapter 6</b>	<b>Conclusions and Future Plans</b>	<b>127</b>
<b>References</b>		<b>130</b>
<b>Appendix A</b>	<b>Proofs of Lemmas/Theorems</b>	<b>141</b>



# LIST OF FIGURES

3.1	Invocation diagrams of Straightforward Algorithm and our Modular Algorithm	37
3.2	Effect of $n$ (dataset size)	43
3.3	Effect of $\mu$ (average length of a sequence)	43
3.4	Effect of $k$ (the length of a query sequence)	44
3.5	Effect of $d$ (the domain size of each attribute)	44
3.6	Effect of $m$ (the number of attributes of each object)	45
3.7	Results for FASM on BookX by <i>SPAMMA</i>	47
4.1	Execution time of all algorithms (small dataset)	81
4.2	Execution time of GR1, GR2 and DP (small dataset)	81
4.3	Profits of all algorithms (small dataset)	82
4.4	Memory costs of all algorithms (small dataset)	82
4.5	Effect of $ Q $ (the number of potential new tuples)	83
4.6	Effect of $ P $ (the number of tuples in the existing market)	84
4.7	Effect of $k$ (the size of the final selection set)	84
4.8	Effect of $l$ (the number of attributes)	85
4.9	Effect of $h$ (the minimum number of tuples dominated by each tuple in the selection set)	85
4.10	Effect of $h$ (the minimum number of tuples dominated by each tuple in the selection set)	86
4.11	Effect of $k$ (the size of the final selection set)	86
4.12	Effect of $ O $ (the size of update operation sets)	87
4.13	Effect of $ CP $ and $r$ on large synthetic datasets	87
4.14	Effect of $ CP $ on a real dataset	88
4.15	Effect of $r$ on a real dataset	88

5.1	A running example	90
5.2	Hotels in the map with a new hotel $h_f$	91
5.3	Region	100
5.4	Convex Hull	100
5.5	R*-tree and the MBRs	106
5.6	Effect of $K$ (the size of the final selection set)	120
5.7	Effect of $n$ (the number of attraction-sites)	121
5.8	Effect of $m$ (the number of service-sites)	121
5.9	Effect of $q$ (the number of non-spatial attributes)	122
5.10	Effect of $K$ (the number of final selection set)	123
5.11	Attractions and hotels in Manhattan	124

## LIST OF TABLES

3.1	An example showing the movie rental application	16
3.2	Notations	20
3.3	Labels	24
3.4	A table showing the bulky version of $X_s$	28
3.5	Default values	40
3.6	Statistics of real datasets	45
3.7	Execution time on real datasets	45
4.1	Packages in the existing market	51
4.2	Potential packages in the new travel agency	51
4.3	Notation table	57
4.4	The $f$ value of each tuple $p \in Q$	60
4.5	Experimental settings on large synthetic datasets	81
4.6	Experiment settings on real dataset	83
5.1	A decision-making table $T$	91
5.2	The running example with 2 non-spatial attributes	112
5.3	A decision-making table $T$	112
5.4	Default values	117
5.5	Value of $p_{max}$ under different star rates	124

# UNDERSTANDING AND UTILIZING USER PREFERENCES

by

YU PENG

Department of Computer Science and Engineering

The Hong Kong University of Science and Technology

## ABSTRACT

With the rapid growth of web-based applications, mining personalized preferences for promotion becomes a hot topic. In this thesis, we focus on two problems related to user preferences: understanding user preferences and utilizing user preferences.

In understanding user preferences, we propose two sub-problems when we consider *temporal user preferences*. The first sub-problem is called attribute-based subsequence matching (ASM) : given a query sequence and a set of sequences, considering the attributes of elements, we want to find all the sequences which are matched by this query sequence. We propose an efficient algorithm for problem ASM by applying the *Chinese Remainder Theorem*. The second sub-problem is to find all the attribute-based frequent subsequences. We adapt an existing efficient algorithm for this second sub-problem to show that we can use the algorithm developed for the first sub-problem. Experimental results show that frequent subsequences reflect user preferences, and our algorithms are scalable in large datasets. This work can stimulate a lot of existing data mining problems which are fundamentally based on subsequence matching.

In utilizing user preferences, we identify and tackle three sub-problems, finding top- $k$  profitable products, finding top- $k$  popular products, and finding  $K$ -dominating competitive price. The former two sub-problems are about designing new products,

while the latter one is about pricing new products.

In finding top- $k$  profitable products, we consider generalized user preferences, derived from the *skyline* concept. We propose a dynamic programming approach which can find the optimal solution when there are two attributes to be considered. We show that this problem is NP-hard when there are more than two attributes. Two greedy algorithms are proposed and one of them has theoretical bounds. We also consider this problem on dynamic datasets and propose update algorithms for different update operations. We extend this problem by considering another form of customer preferences, namely tolerant customer preferences in finding top- $k$  popular products. We prove that this problem is NP-hard and propose a 0.63-approximate algorithm for this problem. Extensive experiments show the effectiveness and efficiency of our approaches on both synthetic and real datasets.

In finding  $K$ -dominating competitive price in which we consider generalized user preferences only, we propose an efficient algorithm. We utilize spatial properties for pruning to speed up our algorithm. An extensive performance study using both synthetic and real datasets is reported to verify its effectiveness and efficiency. We also provide a real case study to show how our algorithm works in reality.

# CHAPTER 1

## INTRODUCTION

With the rapid growth of web-based applications, data mining plays a more and more important role to users and customers. On one side, commercial companies are eager to know personalized preferences for promotion instead of flooding customers with advertisements, especially when they realize that understanding user preferences<sup>1</sup> greatly helps deliver promotions to customers accurately and effectively, which in return rewards them with a large increase in sales and profit. On the other side, enormous business-related data has been recorded and stored in web-based applications and contains undermined information or knowledge about user preferences. Without effective data mining techniques, those undermined information or knowledge about user preferences cannot be found directly from raw data. One successful example of data mining methods for promotion is the introduction of recommendation systems in many large-scaled web-based applications such as Netflix [56] and Amazon [2]. In this thesis, we deal with two main issues about user preferences in this area. One is how to *understand* user preferences based on historical data which is recorded in web-based applications. The other is how to *utilize* user preferences to design new products for profit earning.

In Section 1.1, we introduce how user preferences are defined in the literature and what kinds of user preferences are focused and used in this thesis. Section 1.2 and Section 1.3 introduce our work about understanding user preferences and our work about utilizing user preferences, respectively. Section 1.4 summarizes our contribution for both understanding user preferences and utilizing user preferences. Section 1.5 provides the road map of the rest of this thesis.

---

<sup>1</sup>In this thesis, we use the terms “*user preference*”, “*customer preference*” and “*consumer preference*” interchangeably.

## 1.1 Introduction to User Preferences

User preferences are very important to companies because they can express the tendency of users to choose a list of objects/products. In the literature of data mining, user preferences have various forms. In collaborative filtering recommendation systems [39, 45], a “score” given by a user on a product shows the extent of his/her interest on it. In different scenarios, this “score” is referred to as “rating” or “ranking”. Different scores given to different products by the same user mean that the user prefers some products to others. In some previous studies [34], each score is simply a binary value denoting whether a user likes or dislikes a product. In personalized web-based applications, *strict partial orders* are commonly used to express user preferences in form of a *pairwise* comparison (e.g., “A is better than B”) since first proposed in [54]. [54] describes that some inconsistencies of user preferences on products are allowed in practice and should not be considered as a problem. Obviously, the user preferences based on scores can be transformed to the user preferences based on pairwise comparisons. Therefore, pairwise comparisons, or formally strict partial orders, are frequently used to express user preferences in the literature [55, 47, 51, 94]. In this thesis, we study three forms of user preferences based on pairwise comparisons. The first form is called a *generalized user preference*, the second form is called a *tolerant user preference*, and the third form is called a *temporal user preference*.

The first form can be described clearly when people make decisions as follows. When people make decisions, they have some “rules” in their minds. One obvious rule is: If product  $A$  is “worse” than product  $B$ , then we should not choose “A”. In order to give a more concrete and precise meaning of “worse”, we assume that each product is associated with a set of attributes. With this assumption, the rule becomes: If product “A” is worse than product “B” in at least one attribute, and has the same values as  $B$  in all the rest, then product  $A$  is “worse” than product  $B$ . Products like  $A$  should never be

considered. This is because we have a better choice  $B$  compared with  $A$ . Therefore, we regard this kind of rules for decision-making as *generalized user preferences*. With these generalized user preferences, we can determine which products are competitive and which are not. Products like  $A$  can be considered as *uncompetitive products*, while the rest that are not worse than any other products can be regarded as *competitive products*. This observation is captured in the concept of the *Skyline Set*, commonly used in the mathematics and theory community. This concept will be defined formally in Section 4. The *competitive products* compose the *Skyline Set*. Apparently, products in the *Skyline Set* is preferred to those not in the *Skyline Set* by any customer.

The second form of user preferences is different from the first form. The second form allows that different users can specify different preferences. But the first form represents that different users have the same preferences following the rule described. The first form is helpful when we have no detailed information about customers. However, some customers are aware of the differences between those s/he likes and those s/he dislikes. Therefore, we would like to define user preferences by the *tolerance* of the product set s/he likes. For example, if a customer wants to buy a notebook, s/he can specify his/her preference by  $\langle 2.5\text{ kg}, 14\text{ inches} \rangle$ , which means that s/he prefers notebooks with the weight of at most 2.5 kilograms and with the size of at most 14 inches. It looks like dividing the whole product set into two subsets and s/he prefers products satisfying his/her customer preference in form of  $\langle 2.5\text{ kg}, 14\text{ inches} \rangle$  to the other. This form of user preferences is similar to the *binary value expression* used in [34], but it is easier to model since customer preferences can be directly interpreted with this form. We call this form as *tolerant user preferences*. In the following chapters, if without specification, any user preference refers to *tolerant user preference* by default.

The third form is related to the temporal behavior of a user. This is called a tempo-



ral user preference. Temporal user preferences can be found from customers' purchase or rental records during a period. An example of these preferences is that a user who first buy a MacBook will buy its accessories later. It is easy to see that there is a time domain here which cannot be found in the previous two forms of user preferences. More examples will be given later in this thesis.

## 1.2 Understanding User Preferences

In this section, we describe the problem of "Understanding User Preferences". The user preferences studied here are temporal user preferences. As introduced, we want to understand temporal user preferences by analyzing users' interests from their historical (or temporal) behavior. In this thesis, most user temporal behaviors can be expressed as sequences, for example, transactions in a supermarket and movie rental records in Netflix [56]. So, in this section, we briefly introduce previous studies on sequences and then our work.

Sequence analysis is very important in our daily life. Typically, each sequence is associated with an ordered list of elements. For example, in a movie rental application, a customer's movie rental record containing an ordered list of movies is a sequence example. Most studies [81, 38, 77, 17] about sequence analysis focus on subsequence matching which finds all sequences stored in the database such that a given query sequence is a subsequence of each of these sequences. Since subsequence matching is very useful, it has been commonly adopted in many data mining problems such as frequent subsequence mining [68, 22, 71, 18, 102], sequence classification [96, 80, 40], sequence clustering [81, 14] and motif detecting [84, 63].

In many applications, elements are associated with properties or attributes. For example, each movie is associated with some attributes like "Director" and "Actors". Unfortunately, to the best of our knowledge, all existing studies about sequence anal-

ysis do not consider the attributes of elements. Motivated by this, we propose two problems. The first problem is: given a query sequence (a potential temporal user preference) and a set of sequences, considering the attributes of elements, we want to find all sequences which are matched by this query sequence. This problem is called attribute-based subsequence matching (ASM). All existing applications for the traditional subsequence matching problem can also be applied to our new problem provided that we are given the attributes of elements. We propose a novel algorithm to solve ASM. With this algorithm, we can calculate how many customers have the temporal user preference expressed by the query. The second problem is to find all frequent attribute-based subsequences. These frequent subsequences can be considered as some popular temporal user preferences. We also adapt an existing efficient algorithm for this second problem to show we can use the algorithm developed for the first problem. We also analyze the found frequent subsequences, which results in interesting findings.

### 1.3 Utilizing User Preferences

In this section, we elaborate the problem of “Utilizing User Preferences”. Here we focus on the first two forms of user preferences, namely generalized user preferences and tolerant user preferences. In this thesis, we are interested in utilizing generalized user preferences for designing and pricing new products. As we described, generalized user preferences are related to the *skyline* concept. In recent ten years, the *skyline* concept has received great attention since proposed in 2001 [24] in the database community. Initially, it is referred to as a set of competitive products in the market. Since the *skyline* concept tells us a basic rule of selecting products from a pool of different products, it can be regarded as a generalized user preference.

Since proposed, the importance of dominance and skyline analysis (which will be defined formally in Section 4.1) has been well recognized in multi-criteria decision

making applications. Therefore, most previous works study how to help customers select a set of possible interesting products from a pool of given products. Some representative methods based on numerical attributes include a bitmap method [86], a nearest neighbor (NN) algorithm [58], and a branch and bound skylines (BBS) method [69]. Other methods based on categorical attributes include [26, 25, 94, 79]. Instead of finding the *skyline* set from one table, [52, 85] focus on finding the *skyline* set from the table which is a result of the natural join of multiple tables. Different from the above work, we apply the *skyline* concept from the business perspective. Since *skyline* is related to the generalized user preferences, we utilize them for designing and pricing new products. Our work is mainly on three aspects.

Firstly, we study the problem of finding top- $k$  profitable products when we consider generalized user preferences. Given a set of products in the existing market, we want to find a set of  $k$  “best” possible products such that these new products are not dominated by the products in the existing market and the total profit is maximized. The dominance constraint is related to generalized user preferences. We follow our previous work [90] to generate a set of possible products from existing products. So the input possible new product set is the output of the algorithm we proposed in [90]. We proved that this problem of finding top- $k$  profitable products is NP-hard when the number of attributes is more than 2. We also proposed a dynamic programming method which finds an exact solution when the number of attributes is no more than 2, and two greedy algorithms which find approximate solutions with approximation guarantees when the number of attributes is more than 2 [91]. We also consider the problem of finding top- $k$  profitable products when the data changes over time. Three update algorithms are proposed for insertion, deletion and modification operations.

Secondly, when we additionally consider tolerant user preferences, the problem turns to finding top- $k$  popular products from a set of possible products such that these

new products are not dominated by the products in the existing market and the number of user preferences satisfied is maximized. The first dominance constraint is related to generalized user preferences while the second maximization constraint is related to tolerant user preferences. We propose a greedy algorithm to solve this problem and show that all the returned products are in the skyline set [75].

Thirdly, when we consider creating products or services in *spatial* databases, in our recent work, we propose an interesting problem, *finding competitive price*, which has not been studied before, based on generalized user preferences. Given a set of existing services, for a new service, we want to find the price of the new service such that the new service is not worse than any existing services in the spatial database (which corresponds to generalized user preferences). The price found refers to a competitive price. We also generalize the problem of finding competitive price to the problem of finding  $K$ -dominating competitive price which considers that the new product or service should dominate at least  $K$  other products or services when  $K$  is a positive integer given by a user. We propose an approach which makes use of some spatial properties in the spatial database and thus runs efficiently. Finally, we conducted experiments to show the efficiency of our proposed method.

## 1.4 Contribution

This thesis contribute to two research areas, namely *Understanding User Preferences* and *Utilizing User Preferences*. The major contributions for *Understanding User Preferences* are as follows when we consider temporal user preferences.

1. To the best of our knowledge, we are the first to propose problem ASM which considers the properties of objects. This problem has a lot of convincing applications. This work can stimulate a number of existing data mining problems which

are fundamentally based on subsequence matching such as the frequent subsequence mining problem, the longest common subsequence alignment problem and the discriminative subsequence mining problem.

2. We propose a novel algorithm for problem ASM based on the Chinese Remainder Theorem. We propose a data mining problem to find all frequent subsequences based on ASM to illustrate how ASM can be used in other data mining problems. These frequent subsequences are regarded as temporal user preferences which can be found from a lot of users.
3. We conduct some experiments to show the efficiency of our proposed algorithms. The key idea to the efficiency of this algorithm is to compress each whole sequence with potentially many associated attributes into just a triplet of numbers. By dealing with these very compressed representations, we greatly speed up the attribute-based subsequence matching.

The major contributions for *Utilizing User Preferences* are as follows when we consider generalized user preferences and tolerant user preferences.

1. To the best of our knowledge, we are the first to study how to find top- $k$  profitable products in which generalized user preferences are considered. Finding top- $k$  profitable products can help the effort of companies to find a subset of products together with their corresponding profitable prices, which cannot be addressed by existing methods. We propose a dynamic programming approach which can find an optimal solution when there are two attributes to be considered. We show that this problem is NP-hard when there are more than two attributes. Thus, we propose two greedy approaches to solve the problem efficiently. We also propose an incremental approach when datasets change over time.
2. We are the first to study the problem of finding top- $k$  popular products in which

both generalized user preferences and tolerant user preferences are considered. We prove that this problem is NP-hard and propose a 0.63-approximate algorithm for this problem. We present a systematic performance study using both real and synthetic datasets to verify the effectiveness and the efficiency of our proposed approach.

3. We are the first to study the problem of finding  $K$ -dominating competitive price in which generalized user preferences are considered. We propose an effective spatial approach by using some spatial properties. We conducted experiments to show the efficiency of our proposed approach and illustrate the process with a real case study.

## 1.5 Organization

The rest of this thesis is organized as follows. Chapter 2 provides the related work of *understanding user preferences* and *utilizing user preferences*. Chapter 3 describes our problem ASM, related to *understanding user preferences*, and elaborates our work in this new area. The differences between our sequence matching and existing works are also explained in Chapter 3. Our work about *utilizing user preferences* is presented in Chapter 4 and Chapter 5. Chapter 4 studies two problems, namely *finding top- $k$  profitable products* and *finding top- $k$  popular products*. Chapter 5 proposes an efficient algorithm to price new services when spatial databases are considered. Finally, Chapter 6 concludes the thesis and summarizes our plans for future work. All the proofs of lemmas and theorems in this thesis are presented in the Appendix unless otherwise stated.

## CHAPTER 2

### RELATED WORK

In this chapter, we introduce previous studies in the literature of two problems. As we described before, sequences are related to temporal user preferences defined in Chapter 1 for the problem of *understanding user preferences*. Section 2.1 summarizes representative studies for the problem of subsequence matching and mining. Since we consider generalized user preferences and tolerant user preferences for the problem of *utilizing user preferences*, and these preferences are related to the skyline concept, Section 2.2 summarizes various studies about the skyline concept.

#### 2.1 Subsequence Matching and Mining

Subsequence matching attracted a lot of attention in the database community, the data mining community and the bioinformatic community [38, 17, 43, 81, 14, 77]. Subsequence matching can be classified into two types: *accurate matching* and *approximate matching*. Given a query sequence  $q$ , accurate (subsequence) matching [81, 14, 77] is to find all sequences such that  $q$  is a subsequence of each of these sequences. On the other hand, approximate (subsequence) matching [38, 17, 43] is to find all sequences such that each of these sequences contains a subsequence  $s'$ , and the *distance* between  $s'$  and a query sequence  $q$  is at most a given *tolerance threshold*.

Due to its usefulness in biological sequences, subsequence matching has been studied extensively in the literature of bioinformatics. Algorithm SW [81] is known as the first algorithm solving subsequence matching problem in biological sequences. Besides, [14] and [77] proposed more efficient algorithms for accurate matching. [14]

finds matches by first locating short subsequences of the query in a sequence, and then assembles neighborhood matches of short subsequences to be matches of the query. [14] can also answer approximate queries given a threshold. The authors of [14] developed an open online application which is one of the most widely used bioinformatics programs.

There are a lot of studies about approximate matching [38, 17, 43] requiring users to define a distance metric. [38] used the Euclidean distance as a metric and presented an algorithm for approximate matching by using some indexing techniques. [17] adopted the Dynamic Time Warping (DTW) distance as a metric and proposed an algorithm. [43] proposed to find  $k$  sequences with their smallest distances from a given sequence  $q$  where the distance metric adopted is the DTW distance.

Frequent subsequence mining, also known as sequential pattern mining, is an important problem in sequence data mining, which has been widely studied in the literature. Based on searching strategy, we can also categorize previous methods into two major types [36]. One is Apriori-like, breadth-first search methods, while the other is based on pattern-growth, depth-first search methods. In the literature, [82, 67, 101] belong to the former category, while [42, 72, 93, 18] belong to the latter one.

The *Apriori* property was first proposed in [13] to mine association rules in large databases. [82] utilizes the *Apriori* property [13] for pruning candidate subsequences. [67] improves the GSP algorithm in [82] by replacing the hash tree with a prefix tree for storing candidates. The SPADE algorithm proposed in [101] utilizes a lattice structure to speed up the searching process. [42] finds frequent itemsets at first, and then generates frequent sequential patterns by combining frequent itemsets found previously. [72] follows the pattern-growth strategy in [42], and improves it by considering prefix subsequences and the postfix subsequences of prefix subsequences in projected databases only. [93] extends [72] to mine closed sequential patterns. The SPAM



algorithm proposed in [18] takes the advantage of the bitmap representation to mine frequent subsequences efficiently.

To the best of our knowledge, no existing studies about subsequence matching and mining consider the properties of objects which are studied in our paper [76]. Our work discussed in Chapter 3 belongs to accurate matching. Since the properties of objects are not considered in previous work [14] and [77], they are not applicable to our problem.

## 2.2 Skyline

Skyline queries have been studied since 1960s in the theory field where skyline points are known as Pareto sets and admissible points [37] or maximal vectors [21]. However, earlier algorithms such as [21, 20] are inefficient when there are many data points in a high dimensional space. Skyline queries in database was first studied by Börzsönyi [24] in 2001.

After that, various techniques were proposed to accelerate the computation of skyline and its variations. Here, we briefly summarize some of them. Some representative methods include a bitmap method [86], a nearest neighbor (NN) algorithm [58], and a branch and bound skylines (BBS) method [69].

Some representative studies in various problems related to skyline are summarized as follows. [100, 74, 97, 70] find the skylines with respect to subsets of attributes instead of all attributes. [25, 27, 94, 79] handle categorical attributes in skyline computing. [25, 27, 79] consider partial ordered domain on categorical attributes. [94] finds personalized skyline sets given preferences on categorical attributes. Algorithms to compute skyline sets on dynamic data, such as data streams, time series and moving objects, are proposed in [88, 48, 103, 50, 30, 32]. [28, 31] proposed methods to handle

very high dimensional data. Another hot topic is the complementary problem called *Reverse Skyline*, which has been studied in [60, 61, 35]. Recently, skyline on uncertain data has been discussed in [73, 23, 16]. A more recent work [64] studies how to upgrade products not in the skyline to be in the skyline with the minimum cost. However, our work are related to top- $K$  queries and representative skyline queries, which are different from these studies.

Top- $K$  queries about skyline were studied in [69, 62, 87]. [69] discussed *ranked skyline* and *K-dominating queries*. Given a set of points in  $d$ -dimensional space, *ranked skyline* specifies a monotone ranking function, and returns  $k$  tuples in the  $d$ -dimensional space which have the smallest (or greatest) scores according to an input function. Given a set of points in  $d$ -dimensional space, *K-dominating queries* retrieve  $K$  points that dominate the greatest number of points. [28] also considered *K-dominating queries*, but in high dimensional data.

[62, 87] studied *representative skyline queries*. The problem is to select  $k$  points among all skyline points according to a pre-defined objective function. The  $k$  points in the output are said to be representative. [62] was the first to introduce *representative skyline queries*. [62] finds a set of  $k$  points among all skyline points such that the number of points dominated by this set is maximized. However, the method in [62] cannot be applied in our problem because we consider both the profitability (or popularity) of products and the dominance relation of products, but [62] considers the dominance relation only. Besides, the price of each product is to be found in our problem.

Another definition of representative skyline queries was proposed by [87]. In [87], representative skyline queries are to find  $k$  points (or  $k$  representative points) among all skyline points such that the sum of the distances between each skyline point and its “closest” representative point is minimized.

All of the above studies are to find  $k$  points or tuples given a *single* table where all

attribute values of each tuple in the table are *given*. Our work described in Chapter 4 and Chapter 5 has the following differences. Firstly, we want to find  $k$  tuples from one table ( $Q$ ), given *two* tables (one is  $P$  and the other is  $Q$ ) where one of the attribute values of each tuple in one table ( $Q$ ) is *not* given and is to be found. Secondly, the concept of *profit* is considered in this thesis but not in the above studies.

There also exist some existing studies focus on profit optimization, since they do not consider the skyline techniques, they are different from our work. [15] and [59] are two representative studies. [15] uses an existing economic model, Rosen's hedonic price model, to find customer preferences from reviews given by customers in the web and then find profitable products. Recent work [59] proposes a regression model to find profitable products. But, since [15] and [59] do not consider the skyline techniques, some products found by [15, 59] can dominate some other products and thus these studies cannot guarantee that the products found are in the skyline, which is one of the goals studied in our work.

# CHAPTER 3

## ATTRIBUTE-BASED SUBSEQUENCES MATCHING AND MINING

In this chapter, we present our work in a new defined subsequence matching problem called ASM and an extension problem called FASM for *understanding user preferences*. The motivation is elaborated in Section 3.1. We give a formal definition of ASM and preliminaries in Section 3.2. An efficient algorithm for ASM is proposed in Section 3.3. We apply the proposed algorithm to the frequent attribute-based subsequence mining problem (FASM) in Section 3.4. In Section 3.5, we show the experimental results of the proposed algorithms in solving two problems. In Section 3.6, we conclude this chapter and give some future directions.

### 3.1 Motivation

A *sequence* is an ordered list of *elements* where each element is drawn from a given *element domain set*. For example, in the movie rental application, each movie in the rental store corresponds to an element and a set of all movies corresponds to the element domain set. Each customer rents a list of movies. This (ordered) list corresponds to a sequence. Table 3.1(a) shows a movie rental record table. Each element in the element domain set is associated with a set of *properties* or *attributes*. For instance, in the movie rental application, each movie is associated with some attributes like “Release Year”, “Director” and “Actors”. The properties of some movies are shown in Table 3.1(b). This table is called a *property table*. In particular, the director of movie “Titanic” is “James Cameron” and one of the actors is “Leonardo DiCaprio”.

Customer	List of movies
Alice	Titanic, The Aviator, Inception
Bob	Titanic, The Aviator
Clement	The Departed, The Dark Knight
...	...

(a) Movie rental record table

Movie Name	Release Year	Director	Actor 1	Actor 2	...
Titanic	1997	James Cameron	Leonardo DiCaprio	Kate Winslet	...
The Aviator	2004	Martin Scorsese	Leonardo DiCaprio	Cate Blanchett	...
The Departed	2006	Martin Scorsese	Leonardo DiCaprio	Matt Damon	...
The Dark Knight	2008	Christopher Nolan	Christian Bale	Heath Ledger	...
Avatar	2009	James Cameron	Sam Worthington	Zoe Saldana	...
Inception	2010	Christopher Nolan	Leonardo DiCaprio	Joseph Gordon-Levitt	...
...	...	...	...	...	...

(b) Movie property table

Table 3.1: An example showing the movie rental application

Sequence analysis has received a lot of interest from not only database and data mining communities but also bioinformatics communities. Database researchers study subsequence matching [38, 17, 43, 22, 102] and similarity search [65] while data mining researchers study frequent subsequence mining [18] and sequence prediction [80]. On the other hand, bioinformatics researchers study DNA sequence alignment [77], motif discovery [84, 63] and sequence classification [96, 40].

All of the above sequence analysis applications depend on a fundamental operator called *subsequence matching*. Given two sequences  $s$  and  $s'$ , sequence  $s$  is said to be a *subsequence* of  $s'$  if for any two elements  $e$  and  $e'$  in  $s$  where  $e$  occurs before  $e'$  in  $s$ , both elements  $e$  and  $e'$  occurs in  $s'$  and  $e$  occurs before  $e'$  in  $s'$ . We also say that  $s$  *matches*  $s'$  (or  $s'$  is matched by  $s$ ). For example, if  $s$  is “Titanic, Inception” and  $s'$  is “Titanic, The Aviator, Inception”, then  $s$  is a subsequence of  $s'$  and thus  $s$  matches  $s'$ . *Subsequence matching* is formulated as follows: given a set of sequences and a query sequence  $q$ , we want to find all sequences such that  $q$  is a *subsequence* of each of these sequences. In our running example, if  $q$  is “Titanic, Inception”, Alice’s sequence in Table 3.1 is one of the answers for subsequence matching.

Unfortunately, the traditional subsequence matching problem fails to answer a lot of interesting questions related to the properties of elements. Consider that we want to study how movie “Titanic” creates a star “Leonardo DiCaprio”. In order to do this,

we want to know how many customers are interested in watching movies acted by “Leonardo DiCaprio” after they watched movie “Titanic”. We regard it as a temporal user preference since most people know “Leonardo DiCaprio” only after they watched “Titanic”. Note that “Leonardo DiCaprio” is not a movie (or more formally not an element in the element domain set) but is a property value of a movie. The traditional subsequence matching problem cannot achieve this goal because the original problem is based on the elements but not the property values. Or in other words, traditional studies can find temporal user preferences on the element level. However, our problem is to find temporal user preferences on the property values, which are more diverse and comprehensive. One may adapt the traditional problem and generate all *possible* queries in form of “Titanic,  $x$ ” where  $x$  is a movie acted by “Leonardo DiCaprio”. If there are  $M$  movies acted by “Leonardo DiCaprio”, then this adapted approach will issue  $M$  queries, which is quite inefficient.

Motivated by this, we study a new problem called *attribute-based subsequence matching* (ASM). Informally speaking, the problem is described as follows: given a query sequence which contains some elements and some property values, we want to find all sequences which are *matched* by this sequence query.

## 3.2 Problem Definition and Preliminaries

We are given a set  $\mathcal{E}$  of *elements*.  $\mathcal{E}$  is called an *element domain set*. Each element  $e$  is associated with a set  $\mathcal{A}$  of  $m$  *properties* or *attributes*, namely  $A_1, A_2, \dots, A_m$ . The value of attribute  $A_i$  of an element  $e$  is denoted by  $e.A_i$  where  $i = 1, 2, \dots, m$ . In our running example, a set of movies corresponds to the element domain set  $\mathcal{E}$ . Attribute “Year of Release” and attribute “Director” are two examples of the properties of a movie. For the sake of discussion, we assume that one of the attributes in  $\mathcal{A}$  can uniquely identify an element  $e$ . This attribute is called an *identifying attribute*. In our example, attribute

“Movie Name” is an identifying attribute. Attribute “Director” and attribute “Release Year” are non-identifying attributes.

We define the *domain* of attribute  $A_i$ , denoted by  $D_i$ , to be the set of all possible values in attribute  $A_i$  where  $i = 1, 2, \dots, m$ . For instance, all directors like “James Cameron” and “Martin Scorsese” form the domain of attribute “Director”. We define the *value domain set*, denoted by  $\mathcal{V}$ , to be the union of the domains of all attributes. That is,  $\mathcal{V} = \cup_{i=1}^m D_i$ . Note that  $\mathcal{V}$  is a set of all possible *values*. The table (like Table 3.1(b)) containing all attribute values of each element is called the *property table*.

A value  $v$  is said to be an *identifying value* if  $v$  is a value of an identifying attribute. Note that this value  $v$  can be used to uniquely identify an element  $e$ . For the sake of clarity, we simply say that  $v$  identifies  $e$  (or  $e$  is identified by  $v$ ). For example, both “Titanic” and “The Aviator” are identifying values but “James Cameron” and “Leonardo DiCaprio” are not. We define  $\mathcal{U}$  to be a set of all identifying values. Note that  $\mathcal{U} \subseteq \mathcal{V}$ . If value  $v$  is an identifying value, we define the *attribute value set* of  $v$ , denoted by  $\alpha(v)$ , to be the set of all possible attribute values of the element identified by  $v$ . For example, if  $v$  is “Titanic” (an identifying value), then  $\alpha(v) = \{\text{“Titanic”, 1997, “James Cameron”, “Leonardio DiCaprio”, “Kate Winslet”, ...}\}$ .

A *sequence* is an ordered list of values where each value is drawn from  $\mathcal{V}$ . Suppose that there are  $k$  values in the sequence, a sequence is represented in form of “ $v_1, v_2, \dots, v_k$ ” where  $v_i \in \mathcal{V}$  for  $i = 1, 2, \dots, k$ . In this representation, for any two values  $v_i$  and  $v_j$  where  $i < j$ ,  $v_i$  appears before  $v_j$ . “Titanic, The Aviator, Inception” and “Titanic, Leonardo DiCaprio” are two examples of sequences. Consider that  $s$  is in form of “ $u_1, u_2, \dots, u_l$ ”. Value  $u_i$  in  $s$  is defined to have the *temporal position* equal to  $i$  for  $i \in [1, l]$ . For example, if  $s = \text{“Titanic, Leonardo DiCaprio”}$ , then “Titanic” has the temporal position equal to 1 and “Leonardo DiCaprio” has the temporal position equal to 2.

A sequence is said to be an *identifying sequence* if all values in the sequence are identifying values (i.e., all values in the sequence are drawn from  $\mathcal{U}$ ). For example, “Titanic, The Aviator, Inception” is an identifying sequence but “Titanic, Leonardo DiCaprio” is not.

In a lot of applications, we are given a set  $S$  of *identifying sequences*. Let  $n$  be the total number of identifying sequences in  $S$ . In our running example, we are given a set of identifying sequences and each identifying sequence corresponds to the movie rental record of a customer. Table 3.1(a) shows the set of identifying sequences. In the application of finding a researcher, an identifying sequence corresponds to the academic research background of a researcher while in the biology application, it corresponds to a protein sequence.

Let  $q$  be a query in form of “ $v_1, v_2, \dots, v_k$ ” where  $v_i \in \mathcal{V}$  for  $i \in [1, k]$ . Given a value  $v \in \mathcal{V}$  and a value  $u \in \mathcal{U}$ ,  $v$  is said to *match*  $u$  if  $v \in \alpha(u)$ . For example, if  $u$  is “Titanic” and  $v$  is “Leonardo DiCaprio”, then “Leonardo DiCaprio” matches “Titanic” since “Leonardo DiCaprio”  $\in \alpha(\text{“Titanic”})$ .

**Definition 3.2.1 (Match)** Consider a query  $q$  in form of “ $v_1, v_2, \dots, v_k$ ” where  $v_i \in \mathcal{V}$  for  $i \in [1, k]$ . Consider an (identifying) sequence  $s \in S$  in form of “ $u_1, u_2, \dots, u_l$ ” where  $u_i \in \mathcal{U}$  for  $i \in [1, l]$ . Query  $q$  is said to match  $s$  (or  $s$  is matched by  $q$ ) if there exist  $k$  integers, namely  $j_1, j_2, \dots, j_k$ , such that (1) for each  $i \in [1, k]$ ,  $v_i$  matches  $u_{j_i}$ , and (2)  $1 \leq j_1 < j_2 < \dots < j_k \leq l$  □

Consider that  $q$  is “Titanic, Leonardo DiCaprio” and  $s$  is “Titanic, The Aviator, Inception”. Since “The Aviator” was acted by “Leonardo DiCaprio” and “Titanic” occurs before “The Aviator” in  $s$ , it is easy to verify that  $q$  matches  $s$ . In this example,  $k = 2$  and  $l = 3$ . We also have  $j_1 = 1$  and  $j_2 = 2$ . Note that the first requirement in the above definition holds (i.e., “Titanic” matches itself and “Leonardo DiCaprio”



Table 3.2: Notations

Notation	Description
$S$	sequence set
$\mathcal{E}$	element domain set
$A$	attribute set of elements
$e.A_i$	value of element $e$ on attribute $A_i$
$u_i$	the $i$ -th element of a sequence
$q$	a query sequence
$\mathcal{V}$	union of domains of all attributes
$\alpha(v)$	identified attribute values of $v$
$P(v)$	label of $v$
$V_s$	value matching number of $s$
$L_s$	lower-bound sequential order number of $s$
$U_s$	upper-bound sequential order number of $s$
$\Delta_i$	query-aware lifespan of $v_i$ of $q$ in $s$
$I(v)$	inverted list of $v$
$C(q)$	a set of sequence IDs that satisfy Requirement Value Matching w.r.t. $q$
$n$	number of sequences
$\mu$	average length of sequences
$k$	length of queries
$d$	domain size of each attribute
$m$	number of attributes

matches “The Aviator”), and the second requirement also holds (i.e.,  $1 \leq j_1 < j_2 \leq l$ ).

Most of the notations used in this chapter are listed in Table 3.2.

In this section, we are studying the problem called *Attribute-based Subsequence Matching (ASM)*: Given a query sequence  $q$ , we want to find all sequences in  $S$  which are matched by  $q$ . In our running example, if  $q$  is “Titanic, Leonardo DiCaprio”, we want to find all sequences in  $S$  which are matched by  $q$ . In Table 3.1(a), Alice’s sequence is one of the sequences which are matched by  $q$ .

We give some preliminaries in the following. The *remainder* for a division of a positive integer  $N$  by a positive integer  $d$  is denoted by “ $N \bmod d$ ”. Notation “mod” is called a *modular* operator. Suppose that  $N$  is a large number and can be represented by  $\mathcal{N}$  4-byte integers, and  $d$  is a small number and can be represented by a 4-byte integer. The time complexity of the modular operation is  $O(\mathcal{N})$  [44].

Given a positive integer  $N$  and a positive integer  $d$ ,  $d$  is said to be a *divisor* of  $N$  if

$(N \bmod d) = 0$ . Given three positive integers  $N$ ,  $M$  and  $d$ ,  $N$  and  $M$  are said to have a *common divisor*  $d$  if  $d$  is a divisor of both  $N$  and  $M$ . Two integers  $N$  and  $M$  are said to be *relatively prime* if their greatest common divisor is equal to 1.

The theorems to be introduced are based on the equation using this modular operator in form of  $(N \bmod d) = r$  where  $N$ ,  $d$  and  $r$  are three positive integers. This equation is called a *congruence equation*. Now we introduce Unique Factorization Theorem. Unique Factorization Theorem is a fundamental theorem in number theory and is widely used in cryptography and security field.

**Theorem 3.2.1** (*Unique Factorization Theorem [44]*) Any positive integer  $N \geq 1$  can be uniquely expressed as a product of one or more prime numbers called factors of  $N$ .

□

**Property 1 (Factorization Property [44])** Given a positive integer  $N$  and a positive integer  $f$ ,  $f$  is a factor of  $N$  if and only if  $(N \bmod f)$  is equal to 0.

Next, we introduce *Chinese Remainder Theorem*.

**Theorem 3.2.2** (*Chinese Remainder Theorem [44]*) Let  $m$  be the number of congruence equations. Let  $r_1, r_2, \dots, r_m$  be  $m$  positive integers. Suppose that there are  $m$  pairwise relatively prime numbers:  $n_1, n_2, \dots, n_m$ . Let  $N = n_1 n_2 \dots n_m$ . There exists a unique integer  $x \in [0, N - 1]$  solving the system of  $m$  congruence equations each of which is in form of  $(x \bmod n_i) = r_i$  for  $i \in [1, m]$ .

□

In the literature, we can compute  $x$  by *Extended Euclidean Algorithm* [44] in  $O(m(\log n_{max})^2)$  time where  $n_{max} = \max_{i \in [1, m]} n_i$ . Proofs and other details of Theorem 3.2.1 and Theorem 3.2.2 can be found in [44].

---

**Algorithm 1** Straightforward Algorithm for problem ASM

---

**Require:** a query  $q$  and a set  $S$  of identifying sequences

**Ensure:** a set of identifying sequences in  $S$  which are matched by  $q$

```
1:  $\mathcal{O} \leftarrow \emptyset$ 
2: for each  $s \in S$  do
3:    $\text{isMatch} \leftarrow \text{checkMatch}(q, s)$ 
4:   if  $\text{isMatch} = \text{true}$  then
5:      $\mathcal{O} \leftarrow \mathcal{O} \cup \{s\}$ 
6: return  $\mathcal{O}$ 
```

---

---

**Algorithm 2** Algorithm  $\text{checkMatch}(q, s)$  (Naive Implementation)

---

**Require:** a query  $q$  and an identifying sequence  $s$

**Ensure:** whether  $q$  matches  $s$

```
1: let  $s$  be a sequence in form of " $u_1, u_2, \dots, u_l$ "
2: let  $q$  be a query in form of " $v_1, v_2, \dots, v_k$ "
3:  $j \leftarrow 1$ 
4: for  $i = 1$  to  $k$  do
5:   find the smallest integer  $r \in [j, l]$  such that  $v_i \in \alpha(u_r)$ 
6:   if there exists such a value  $r$  then
7:      $j \leftarrow r + 1$ 
8:   else
9:     return false
10: return true
```

---

### 3.3 Modular-based Algorithm

A possible approach for problem ASM is shown in Algorithm 1. In this algorithm, method  $\text{checkMatch}(q, s)$  is to return a boolean value indicating whether a query  $q$  matches an identifying sequence  $s$ . The efficiency of this algorithm depends on how to implement method  $\text{checkMatch}$ . One naive implementation is shown in Algorithm 2 which takes  $O(lm)$  time where  $l$  is the maximum length of a sequence in  $S$  and  $m$  is the total number of attributes. However, the time complexity of Algorithm 1 is  $O(nlm)$  where  $n$  is the total number of sequences in  $S$ . Apparently, Algorithm 1 is time-consuming.

We design an algorithm called *Modular Algorithm* with two major requirements which are simple to understand and are used to ease the understanding on how we use Chinese Remainder Theorem for problem ASM. They are *Requirement Value Matching* and *Requirement Sequential Order*.

**Definition 3.3.1** Given a value  $v \in \mathcal{V}$  and a sequence  $s \in S$ ,  $p(v, s)$  is defined to be

a set of all temporal positions such that each of the values at these positions in  $s$  is matched by  $v$ . □

Consider Alice's sequence  $s$ . If  $v$  is equal to "Leonardo DiCaprio", then  $p(v, s) = \{1, 2, 3\}$ . If  $v$  is equal to "Martin Scorsese", then  $p(v, s) = \{2\}$ .

Consider query  $q$  in form of " $v_1, v_2, \dots, v_k$ ".

**Definition 3.3.2 (Requirement Value Matching)** Let  $s$  be a sequence and  $q$  be a query in form of " $v_1, v_2, \dots, v_k$ ". If for each  $i \in [1, k]$ ,  $p(v_i, s) \neq \emptyset$ , then  $s$  is said to satisfy Requirement Value Matching. □

Intuitively, Requirement Value Matching requires that each value in  $q$  matches some of the values in a sequence  $s$  (without considering the temporal ordering of values). If this requirement is satisfied, we proceed to check the second requirement, Requirement Sequential Order, considering the temporal ordering of values.

**Definition 3.3.3 (Requirement Sequential Order)** Let  $s$  be a sequence. If there exist  $k$  integers, namely  $j_1, j_2, \dots, j_k$ , such that  $j_i \in p(v_i, s)$  for  $i \in [1, k]$  and  $j_1 < j_2 < \dots < j_k$ , then  $s$  is said to satisfy Requirement Sequential Order. □

Intuitively, Requirement Sequential Order requires that these "matched" values in  $s$  have the same temporal ordering as the correspondence values in  $q$ .

This algorithm involves two major phases. The first phase is called *Phase Preprocessing* and the second phase is called *Phase Query*. In Phase Preprocessing, given a set  $S$  of identifying sequences and the property table, we generate not only some synopsis of sequences but also some data structures which will be used in Phase Query. In Phase Query, given a query  $q$ , we find all sequences in  $S$  which are matched by  $q$  using the information generated in Phase Preprocessing.

Value	Label	Value	Label
Titanic	2	The Aviator	13
1997	3	2004	17
James Cameron	5	Martin Scorsese	19
Leonardo DiCaprio	7	Cate Blanchett	23
Kate Winslet	11	...	...
...	...	...	...

Table 3.3: Labels

Consider a *particular* sequence  $s$  in  $S$  and an *arbitrary* query  $q$ . We want to create a synopsis for  $s$  such that we can determine whether  $q$  matches  $s$  efficiently using the synopsis. In order to achieve this goal, we should create this synopsis which can help to determine whether the two requirements are satisfied *efficiently*. In particular, the synopsis contains two separate components. The first component, denoted by  $V_s$ , is used for Requirement Value Matching and corresponds to the first number in this triplet (Section 3.3.1). The other, denoted by  $X_s$ , is used for Requirement Sequential Order and corresponds to the last two numbers in this triplet (Section 3.3.2).

Before we create a synopsis, we first assign each *value* in  $\mathcal{V}$  with a unique prime *number* called the *label* of  $v$ . The label of  $v$  is denoted by  $P(v)$ . Table 3.3 shows the labels of some values in  $\mathcal{V}$  in our running example.

### 3.3.1 Requirement Value Matching

Given a sequence  $s \in S$ ,  $V_s$  is the *value matching number* of  $s$  which is defined as follows.

**Definition 3.3.4 (Value Matching Number)** *Given a sequence  $s \in S$  where  $s$  is in form of “ $v_1, v_2, \dots, v_l$ ”, the value matching number of  $s$  is defined to be the product of the labels of all the values in  $s$  and their property values. Formally,  $V_s = \prod_{i=1}^l \prod_{v \in \alpha(v_i)} P(v)$ .* □

---

**Algorithm 3** Algorithm **valueMatchCheck**( $q, V_s$ ) for Requirement Value Matching

---

**Require:** a query  $q, V_s$

**Ensure:** whether the temporal order of the matching values in  $s$  is consistent with the temporal order of all values in  $q$

- 1: let query  $q$  be in the form of “ $v_1, v_2, \dots, v_k$ ”
  - 2: **for**  $i = 1$  to  $k$  **do**
  - 3:   compute  $(V_s \bmod P(v_i))$  and store the answer as  $a_i$
  - 4: **if**  $a_i = 0$  for each  $i \in [1, k]$  **then**
  - 5:   **return** true
  - 6: **else**
  - 7:   **return** false
- 

**Phase Preprocessing:** In Phase Preprocessing, for each sequence  $s \in S$ , we compute  $V_s$ .

It is easy to verify that the running time of this phase is  $O(nlm)$  where  $l$  is the greatest length of a sequence.

**Phase Query:** Recall that Requirement Value Matching is that given a query  $q$  and a sequence  $s$ , each value in  $q$  matches one of the values in  $s$ . By using  $V_s$ , we can perform  $k$  modular operations to check whether  $s$  satisfies this requirement or not where  $k$  is the length of the query sequence. Algorithm 3 shows the algorithm.

With the following lemma, we know that the algorithm is correct.

**Lemma 3.3.1** *Let  $s$  be a sequence and  $q$  be a query in form of “ $v_1, v_2, \dots, v_k$ ”. If  $(V_s \bmod P(v_i)) = 0$  for each  $i \in [1, k]$ , then  $s$  satisfies Requirement Value Matching.*

□

We know that we can use  $V_s$  to check for Requirement Value Matching. Now, we want to introduce a stronger requirement called *Requirement Duplicate Value Matching*. This requirement states that given a sequence  $s$  and a query sequence  $q$ , for each value  $v$  in  $q$ , if  $v$  appears  $\gamma$  times in  $q$  and there exists  $\gamma$  values in  $s$  such that  $v$  matches each of these  $\gamma$  values, then  $s$  is said to satisfy *Requirement Duplicate Value Matching*. For example, consider Alice’s sequence  $s$ . If  $q = \text{“Titanic, Titanic”}$ , then  $s$  does not satisfy Requirement Duplicate Value Matching. If  $q = \text{“Leonardo DiCaprio, Leonardo DiCaprio”}$ , then  $s$  satisfies this requirement.

---

**Algorithm 4** Algorithm **valueMatchCheck**( $q, V_s$ ) for Requirement (Duplicate) Value Matching

---

**Require:** a query  $q, V_s$

**Ensure:** whether the temporal order of the matching values in  $s$  is consistent with the temporal order of all values in  $q$

```
1: let query  $q$  be in the form of " $v_1, v_2, \dots, v_k$ "
2:  $V \leftarrow V_s$ 
3: for  $i = 1$  to  $k$  do
4:   compute  $(V \bmod P(v_i))$  and store the answer as  $a_i$ 
5:   if  $a_i = 0$  then
6:      $V \leftarrow V/P(v_i)$ 
7: if  $a_i = 0$  for each  $i \in [1, k]$  then
8:   return true
9: else
10:  return false
```

---

By using  $V_s$ , we can also check for Requirement Duplicate Value Matching efficiently. By doing this, we modify Algorithm 3 to Algorithm 4.

Now, we analyze the storage size of  $V_s$ . Consider a particular sequence  $s$ . According to Definition 3.3.4, we need to multiply  $ml$  prime numbers. In all of our experiments, each prime number can be represented by a 4-byte integer. The storage size of  $V_s$  is *at most*  $4ml$  bytes. In case that a prime number needs more bits for storage, we can use some libraries [1] over large numbers which contain a lot of efficient bitwise operations.

Let the storage size of  $V_s$  be  $\mathcal{N}$ . In the above analysis,  $4ml$  is a loose upper bound on  $\mathcal{N}$ . In other words, in most cases,  $\mathcal{N} \ll 4ml$ . In method **valueMatchCheck**, it involves  $k$  modular operations where the divisor of each operation is a prime number. Note that in all our experiments, each prime number can be represented by a 4-byte integer. It is easy to verify that the running time of **valueMatchCheck** is  $O(k\mathcal{N})$ .

### 3.3.2 Requirement Sequential Order

In Section 3.3.2A, we first describe a *bulky* version of component  $X_s$ . Then, in Section 3.3.2B, we describe a *compressed* version of component  $X_s$  by just a pair of two numbers based on Chinese Remainder Theorem.

#### A. Bulky Version of $X_s$

**Phase Preprocessing:** Before we describe this component, we give some definitions first.

An *interval* is defined to be in form of  $(l, u)$  where  $l$  and  $u$  are two positive integers and  $l \leq u$ .

**Definition 3.3.5 (Appear After/Before)** Let  $\Delta_i$  and  $\Delta_j$  be two intervals  $(l_i, u_i)$  and  $(l_j, u_j)$ , respectively.  $\Delta_i$  appears before  $\Delta_j$  (or  $\Delta_j$  appears after  $\Delta_i$ ) if  $u_i < l_j$ .  $\square$

**Definition 3.3.6 (Lifespan)** Given a value  $v \in \mathcal{V}$  and a sequence  $s \in S$  where  $p(v, s) \neq \emptyset$ , the lifespan of  $v$  in  $s$ , denoted by  $LS_{v,s}$ , is defined to be an interval in form of  $(l, u)$  where (1)  $l = \min p(v, s)$  and (2)  $u = \max p(v, s)$ . We define  $LS_{v,s}.l$  to be  $l$  and  $LS_{v,s}.u$  to be  $u$ .  $\square$

Consider Bob's sequence  $s$ . If  $v$  is "Titanic", then  $p(v, s) = \{1\}$  and thus the lifespan of  $v$  in  $s$  is  $(1, 1)$ . If  $v$  is "Leonardo DiCaprio", then  $p(v, s) = \{1, 2\}$  and thus the lifespan of  $v$  in  $s$  is  $(1, 2)$ .

We are ready to describe the bulky version of  $X_s$ . Consider a sequence  $s$  in  $S$ . Let  $Y$  be the set of all values in  $s$  and their property values. Let  $h$  be the total number of possible values in  $Y$ . For each value  $v \in Y$ , we create an entry in form of  $(v, LS_{v,s})$ . The *bulky version* of  $X_s$  is equal to the table storing all these entries.

**Example 1** Consider Bob's sequence  $s$ , "Titanic, The Aviator". Note that "Titanic" has the temporal position equal to 1 and "The Aviator" has the temporal position equal to 2. The attribute values of "Titanic" are

- "Titanic" (with label = 2),
- "1997" (with label = 3),
- "James Cameron" (with label = 5),



Value $v$	Lifespan of $v$ in $s$
“Titanic”	(1, 1)
“1997”	(1, 1)
“James Cameron”	(1, 1)
“Leonardo DiCaprio”	(1, 2)
“Kate Winslet”	(1, 1)
“The Aviator”	(2, 2)
“2004”	(2, 2)
“Martin Scorsese”	(2, 2)
“Cate Blanchett”	(2, 2)

Table 3.4: A table showing the bulky version of  $X_s$

- “Leonardo DiCaprio” (with label = 7) and
- “Kate Winslet” (with label = 11)

The attribute values of “The Aviator” are

- “The Aviator” (with label = 13),
- “2004” (with label = 17),
- “Martin Scorsese” (with label = 19),
- “Leonardo DiCaprio” (with label = 7) and
- “Cate Blanchett” (with label = 23)

Thus, we have the set  $Y$  equal to {“Titanic”, “1997”, “James Cameron”, “Leonardo DiCaprio”, “Kate Winslet”, “The Aviator”, “2004”, “Martin Scorsese”, “Cate Blanchett”}. Then, we calculate the lifespan of each value  $v \in Y$  in this sequence as shown in Table 3.4. For example, if  $v = \text{“Titanic”}$ , then the lifespan of  $v$  in  $s$  is (1, 1). Similarly, if  $v = \text{“Leonardo DiCaprio”}$ , then the lifespan of  $v$  in  $s$  is (1, 2). Table 3.4 corresponds to the bulky version of  $X_s$ .  $\square$

Let  $l$  be the greatest length of a sequence in  $S$ . There are  $O(lm)$  possible values in  $Y$ . Thus, the size of the bulky version of  $X_s$  is  $O(lm)$ . In Section 3.3.2B, we present

a compressed version of  $X_s$  which contains only two positive numbers. Similarly, we can easily derive that the complexity of this phase is  $O(lm)$ .

**Phase Query:** Suppose that we are given a query sequence  $q$  and a sequence  $s$  in  $S$ . We want to check whether  $q$  matches  $s$  using the bulky version of  $X_s$ .

**Definition 3.3.7 (Query-Aware Lifespan)** *Let  $s$  be a sequence in  $S$  and  $X_s$  be the bulky version of  $X_s$  for  $s$ . Given a query sequence  $q$  in form of  $(v_1, v_2, \dots, v_k)$ , if  $s$  satisfies Requirement Value Matching with respect to  $q$ , then the query-aware lifespan of  $s$  with respect to  $q$ , denoted by  $QA-LS(s, q)$ , is defined to be  $(\Delta_1, \Delta_2, \dots, \Delta_k)$  where  $\Delta_i$  is the lifespan of  $v_i$  in  $s$  for  $i \in [1, k]$ .*  $\square$

Our strategy is to create the query-aware lifespan of  $s$  with respect to  $q$  according to the bulky version of  $X_s$ . This can be done in  $O(k)$  time if the bulky version of  $X_s$  is indexed with a hash data structure. Then, according to the query-aware lifespan, we can determine whether  $q$  matches  $s$  efficiently, which will be described next.

**Definition 3.3.8 (Non-Overlapping)** *Consider a sequence  $s$  and a query sequence  $q$ . Let the query-aware lifespan of  $s$  with respect to  $q$  be  $(\Delta_1, \Delta_2, \dots, \Delta_k)$ . The query-aware lifespan is said to be non-overlapping if and only if for each  $i, j \in [1, k]$  where  $i < j$ ,  $\Delta_i$  appears before  $\Delta_j$ .*  $\square$

**Definition 3.3.9 (Invalid)** *Consider a sequence  $s$  and a query sequence  $q$ . Let the query-aware lifespan of  $s$  with respect to  $q$  be  $(\Delta_1, \Delta_2, \dots, \Delta_k)$ . The query-aware lifespan is said to be invalid if and only if there exist any two integers  $i, j \in [1, k]$  such that  $i < j$  and  $\Delta_i$  appears after  $\Delta_j$ .*  $\square$

With the above definitions, we have the following lemma about how to determine whether  $q$  matches  $s$  or not.

---

**Algorithm 5** Algorithm **timespanCheck**( $q, s, QA-LS(s, q)$ )

---

**Require:** a query  $q$ ,  $s$  and  $QA-LS(s, q)$

**Ensure:** whether  $s$  satisfies Requirement Sequential Order

```
1: if  $QA-LS(s, q)$  is non-overlapping then  
2:   return true  
3: else  
4:   if  $QA-LS(s, q)$  is invalid then  
5:     return false  
6:   else  
7:     isMatch  $\leftarrow$  checkMatch( $q, s$ )  
8:     return isMatch
```

---

**Lemma 3.3.2** Consider a sequence  $s$  and a query sequence  $q$ . If the query-aware lifespan of  $s$  with respect to  $q$  is non-overlapping, then  $q$  matches  $s$ . If the query-aware lifespan of  $s$  with respect to  $q$  is invalid, then  $q$  does not match  $s$ .  $\square$

Suppose that we are given the query-aware lifespan of  $s$  with respect to  $q$ . Algorithm 5 shows the steps of checking whether  $q$  matches  $s$  according to the query-aware lifespan only. It is easy to verify that checking the conditions on whether the query-aware lifespan is non-overlapping (or invalid) takes  $O(k)$  time. If these conditions are not satisfied, we need to execute the statements in lines 7-8 involving **checkMatch** which takes  $O(lm)$  time. In our experiments, on average, there are about 90% cases that the query-aware lifespan is either non-overlapping or invalid. Thus, in most cases, the running time of **sequentialOrderCheck** is  $O(k)$ .

Let us analyze the storage complexity of the bulky version of  $X_s$ . Consider a particular sequence  $s$ . Let  $|Y|$  be the average size of  $Y$  (i.e., the average number of possible values in a sequence  $s$  and their property values). For each value  $v \in Y$ , we need to store entry  $(v, LS_{v,s})$ . In our implementation,  $v$  is stored in form of a prime number and  $LS_{v,s}$  is stored in form of two temporal positions. Since in all our experiments, the greatest possible values of each prime number and each temporal position can be represented by a 4-byte integer, each prime number and each temporal position are stored in a 4-byte integer. Thus, each entry occupies  $4 \times 3 = 12$  bytes. Since there are  $|Y|$  entries, the storage size of the bulky version of  $X_s$  for a particular

sequence  $s$  is equal to  $12|Y|$  bytes.

## B. Compressed Version of $X_s$ Based on a Pair of Numbers

Consider a sequence  $s$  in  $S$ . The bulky version contains  $h$  entries and each entry contains a value and its lifespan in  $s$ . This bulky version occupies a lot of space. Interestingly, the *compressed* version of  $X_s$  to be described contains only two positive numbers, which is quite space-efficient.

Specifically, given a sequence  $s \in S$ , the *compressed* version of  $X_s$  is defined to be equal to a pair of two numbers. The first number is called the *lower-bound sequential order number* of  $s$ , denoted by  $L_s$ , and the second number is called the *upper-bound sequential order number* of  $s$ , denoted by  $U_s$ . In Phase Preprocessing, these two numbers are to be found.

**Phase Preprocessing:** In Phase Preprocessing, for each sequence  $s \in S$ , we compute  $L_s$  and  $U_s$  as follows.

Let  $Y$  be the set of all values in  $s$  and their property values. Let  $h$  be the total number of possible values in  $Y$ . Recall that in the *bulky* version, for each value  $v \in Y$ , we create an entry in form of  $(v, LS_{v,s})$  where  $LS_{v,s}$  is the lifespan of  $v$  in  $s$ . Note that  $LS_{v,s}$  is in form of  $(LS_{v,s}.l, LS_{v,s}.u)$ . However, in the *compressed* version, for each value  $v \in Y$ , we *conceptually* create a pair of congruence equations as follows. The following equations are in the format of Chinese Remainder Theorem. Note that  $P(v)$  is the label of value  $v$ .

$$(L_s \bmod P(v)) = LS_{v,s}.l \quad (3.1)$$

$$(U_s \bmod P(v)) = LS_{v,s}.u \quad (3.2)$$

Since we have  $h$  values in  $Y$ , we conceptually generate  $h$  congruence equations in

form of (3.1) and  $h$  congruence equations in form of (3.2).

We first consider the  $h$  equations for  $L_s$  in form of (3.1) and describe how to determine  $L_s$ , one of the two numbers stored in the compressed form of  $X_s$ . Specifically, since the labels of all values in  $Y$  are prime numbers, they are pairwise relatively prime. Note that  $P(v)$  and  $LS_{v,s}.l$  are given in Equation (3.1) where  $v \in Y$ . This is the equation format of Chinese Remainder Theorem. By using the Extended Euclidean algorithm, we can find a unique integer  $L_s \in [0, N - 1]$  where  $N$  is the product of the labels of all values in  $Y$ .

We can use a similar technique to find  $U_s$  by considering the  $h$  equations for  $U_s$  in form of (3.2). Thus, the final compressed version of  $X_s$  are two numbers, namely  $L_s$  and  $U_s$ .

**Example 2** Consider Bob's sequence again. According to Example 1, we can obtain the bulky version of  $X_s$  (Table 3.4).

In the compressed version, since we have 9 values in  $Y$ , we can conceptually formulate 9 congruence equations for  $L_s$  and 9 congruence equations for  $U_s$ . Take  $v = \text{"Titanic"}$  for illustration. Since its label is equal to 2 and its  $LS_{v,s}.l$  is equal to 1, according to Equation (3.1), we create

$$(L_s \bmod 2) = 1$$

The other 8 congruence equations for  $L_s$  are:

$$(L_s \bmod 3) = 1 \quad (L_s \bmod 11) = 1 \quad (L_s \bmod 19) = 2$$

$$(L_s \bmod 5) = 1 \quad (L_s \bmod 13) = 2 \quad (L_s \bmod 23) = 2$$

$$(L_s \bmod 7) = 1 \quad (L_s \bmod 17) = 2$$

Similarly, we can construct the 9 congruence equations for  $U_s$ . By the Extended Eu-

---

**Algorithm 6** Algorithm **sequentialOrderCheck**( $q, L_s, U_s$ ) for Requirement Sequential Order

---

**Require:** a query  $q$ ,  $L_s$  and  $U_s$

**Ensure:** whether  $s$  satisfies Requirement Sequential Order

- 1: let query  $q$  be in the form of “ $v_1, v_2, \dots, v_k$ ”
  - 2: **for**  $i = 1$  **to**  $k$  **do**
  - 3:    $l_i \leftarrow (L_s \bmod P(v_i))$
  - 4:    $u_i \leftarrow (U_s \bmod P(v_i))$
  - 5:    $\Delta_i \leftarrow (l_i, u_i)$
  - 6: call **timespanCheck**( $q, s, (\Delta_1, \Delta_2, \dots, \Delta_k)$ )
- 

clidean Algorithm, we obtain  $L_s = 134,918,071$  and  $U_s = 7,436,431$ . □

We know that the time complexity of finding a solution for  $L_s$  (and  $U_s$ ) with  $m$  congruence equations is  $O(m(\log n_p)^2)$  time [44] where  $n_p$  is the largest prime numbers we use. Since each sequence  $s$  is associated with  $L_s$  (and  $U_s$ ), the running time of this phase considering all sequences is equal to  $O(nm(\log n_p)^2)$ .

**Phase Query:** Suppose that sequence  $s$  satisfies Requirement Value Matching. Recall that Requirement Sequential Order is that the “matched” values in  $s$  have the same temporal ordering as the correspondence values in  $q$ . Algorithm 6 shows how we check whether  $s$  satisfies Requirement Sequential Order using  $L_s$  and  $U_s$ . It is easy to see Algorithm 6 returns a correct solution with the following lemma.

**Lemma 3.3.3** Consider a sequence  $s$ . Let  $L_s$  and  $U_s$  be the lower-bound sequential order number and the upper-bound sequential order, respectively. Let  $Y$  be the set of all values in  $s$  and their property values. Given a value  $v \in Y$ , the lifespan of  $v$  in  $s$  is equal to  $(l, u)$  where  $l = (L_s \bmod P(v))$  and  $u = (U_s \bmod P(v))$ . □

We analyze the storage complexity of the compressed version of  $X_s$  as follows. Consider a particular sequence  $s$ . For this sequence  $s$ , we need to store two numbers, namely  $L_s$  and  $U_s$ . Consider number  $L_s$  which is computed based on  $|Y|$  congruence equations. Note that  $L_s$  is at most the multiplication of the divisors of all congruence

equations (i.e.,  $\prod_{v \in Y} P(v)$ ). In all of our experiments, each divisor (or each prime number) can be represented by a 4-byte integer. Thus,  $L_s$  can be represented by  $|Y|$  4-byte integers and thus the size of  $L_s$  is *at most*  $4|Y|$  bytes. Note that  $4|Y|$  is a *upper bound* of the size of  $L_s$ . In most cases, the exact size is smaller than  $4|Y|$ . Similarly, we can derive that the size of  $U_s$  is *at most*  $4|Y|$  bytes. The storage size of the compressed version of  $X_s$  given a particular sequence is at most  $4|Y| + 4|Y| = 8|Y|$  bytes. Since  $|Y|$  is at most  $ml$ , the storage size is at most  $8ml$  bytes.

Let  $\mathcal{N}'$  be the storage size of  $L_s$  (or  $U_s$ ). Similarly, in the above analysis,  $4|Y|$  is a loose upper bound on  $\mathcal{N}'$ . In other words,  $\mathcal{N}' << 4|Y| (< 4ml)$ . Similarly, it is easy to verify that the running time of **sequentialOrderCheck** is  $O(k\mathcal{N}')$  time if the running time of **timespanCheck** is  $O(k)$  in most cases.

### 3.3.3 Comparison

We compare the storage of the compressed version with the storage of the bulky version. Consider  $X_s$  of both versions. The compressed version of  $X_s$  occupies at most  $8|Y|$  bytes and the bulky version of  $X_s$  occupies  $12|Y|$  bytes. Thus, the storage size of the compressed version of  $X_s$  is at most  $2/3$  of the bulky version of  $X_s$ . Now, we consider the storage size of the compressed/bulky *synopsis* containing not only  $X_s$  but also  $V_s$ . Note that  $V_s$  is the common component used by the compressed version and the bulky version. Note that since the storage size of  $X_s$  is equal to  $4ml$ , the compressed synopsis occupies at most  $4ml + 8|Y| \leq 12ml$  bytes and the bulky synopsis occupies  $4ml + 12|Y| \leq 16ml$ . Thus, the storage size of the compressed synopsis is *at most*  $3/4$  ( $= 12/16$ ) of the storage size of the bulky synopsis. In the experimental results (Section 3.5), the real compression effect is more significant. On average, the storage size of the compressed synopsis is about  $1/4$  of the storage size of the bulky synopsis. The above theoretical analysis is based on the *upper bound* of the storage

size of the compressed synopsis (instead of the *exact* storage size) and thus the bound of 3/4 is not quite tight.

### 3.3.4 Putting Two Requirements Together

In this section, we present algorithms to combine the two requirements together in addition to introducing an indexing technique called *inverted list*.

### 3.3.5 Phase Preprocessing

In addition to the steps we discussed previously, we describe an indexing technique called *inverted list*. Suppose that each sequence  $s \in S$  is given a unique sequence ID. Given a value  $v \in \mathcal{V}$ , the *inverted list* of  $v$ , denoted by  $I(v)$ , is defined to be a set of sequence IDs such that one of the values in each of these sequences has its property values equal to  $v$ . Given a query  $q$ , we define  $C(q)$  to be a set of sequence IDs where each of the sequences with these IDs satisfies Requirement Value Matching. Thus,  $C(q)$  is equal to  $\cap_{i=1}^k I(v_i)$ .

So, there are two major steps in Phase Preprocessing. The first step is to generate the inverted list of  $v$  for each value  $v \in \mathcal{V}$ . The second step is to generate the synopsis of each sequence  $s$  (where the synopsis is in form of a triplet  $(V_s, L_s, U_s)$ ).

Note that both component  $V_s$  described in Section 3.3.1 and inverted lists are used for Requirement Value Matching. However, there are some differences. Firstly, inverted lists are used to locate sequences satisfying Requirement Value Matching by sequence IDs. Secondly,  $V_s$  can be used for Requirement Duplicate Value Matching but inverted lists cannot.

Note that the complexity of synopsis generation is  $O(nlm)$  where  $l$  is the greatest length of a sequence. Generating inverted lists also takes  $O(nlm)$  time. The overall complexity of this phase is equal to  $O(nlm)$ .



---

**Algorithm 7** Modular Algorithm for problem ASM

---

**Require:** a query  $q$  and a set  $S$  of identifying sequences

**Ensure:** a set of identifying sequences in  $S$  which are matched by  $q$

```
1:  $\mathcal{O} \leftarrow \emptyset$ 
2: for each  $s \in C(q)$  do
3:    $\text{isMatch} \leftarrow \text{checkMatch-Synopsis}(q, s)$ 
4:   if  $\text{isMatch} = \text{true}$  then
5:      $\mathcal{O} \leftarrow \mathcal{O} \cup \{s\}$ 
6: return  $\mathcal{O}$ 
```

---

---

**Algorithm 8** Algorithm **checkMatch-Synopsis**( $q, s$ )

---

**Require:** a query  $q$  and an identifying sequence  $s$

**Ensure:** whether  $q$  matches  $s$

```
1:  $\text{isValueMatch} \leftarrow \text{valueMatchCheck}(q, V_s)$ 
2: if  $\text{isValueMatch} = \text{true}$  then
3:    $\text{isSeqOrder} \leftarrow \text{sequentialOrderCheck}(q, L_s, U_s)$ 
4:   if  $\text{isSeqOrder} = \text{true}$  then
5:     return true
6: return false
```

---

### 3.3.6 Phase Query

With the inverted list, we can modify Algorithm 1 to Algorithm 7. The differences come from the statements in Line 2 and Line 3. Firstly, in Line 2 of Algorithm 7, instead of processing all sequences in  $S$ , we process the sequences in  $C(q)$  using the inverted list. Secondly, in Line 3 of Algorithm 7, instead of calling the original method **checkMatch** without using any synopsis, we call the new method **checkMatch-Synopsis** using the synopsis.

Algorithm 8 shows the algorithm for **checkMatch-Synopsis**. With Lemmas 3.3.1, 3.3.2 and 3.3.3, it is easy to verify the following theorem.

**Theorem 3.3.1** *Algorithm 7 returns all sequences which are matched by  $q$ .*

The proof of Theorem 3.3.1 is omitted as it can be easily derived from the proofs of Lemmas 3.3.1, 3.3.2 and 3.3.3.

Consider Algorithm 7. There are  $O(n)$  sequences in  $C(q)$ . Consider a sequence in  $C(q)$ . We need to execute **checkMatch-Synopsis** (Algorithm 8). In this algorithm, we know that **valueMatchCheck** takes  $O(k\mathcal{N})$  time and **sequentialOrderCheck**( $q, L_s, U_s$ ) takes  $O(k\mathcal{N}')$  time in most cases. Thus, **checkMatch-Synopsis** takes  $O(k(\mathcal{N} + \mathcal{N}'))$

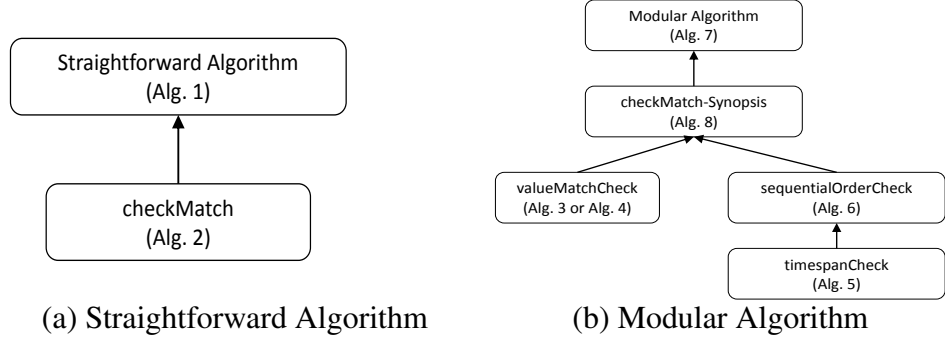


Figure 3.1: Invocation diagrams of Straightforward Algorithm and our Modular Algorithm

time. Since  $\mathcal{N} \geq \mathcal{N}'$ , the time complexity of **checkMatch-Synopsis** becomes  $O(k\mathcal{N})$ . In conclusion, the overall time complexity of Algorithm 7 is  $O(nk\mathcal{N})$ .

We show the invocation diagrams of Straightforward Algorithm and our Modular Algorithm in Figure 3.1. Each rounded-corner rectangle in Figure 3.1 represents a module in the corresponding algorithm and each module can be implemented by the algorithms specified in the brackets.

### 3.4 Frequent Attribute-based Subsequence Mining

In this section, we introduce a data mining problem, *frequent attribute-based subsequence mining*, which frequently makes use of the efficient operator for ASM (i.e., checking whether a query sequence matches a sequence in the dataset). Traditional frequent subsequence mining has been studied extensively in the literature [68, 18, 71, 102, 22]. It is useful to find frequent patterns in order to study customers' behaviors and temporal patterns. As introduced at the beginning of this chapter, ASM focuses on finding temporal user preferences. In this section, we propose frequent *attribute-based subsequence mining* (FASM), to find some popular temporal user preferences. It is the same as the traditional mining except that we consider the property table. These popular temporal user preferences are useful to companies. For example, with popular temporal user preferences as features, we can categorize customers into different

groups. Then we can make different promotion strategies for different groups according to their temporal user preferences.

Given a sequence  $p$  in form of “ $v_1, v_2, \dots, v_x$ ” where  $v_i \in \mathcal{V}$  for  $i \in [1, x]$ , the *frequency* of  $p$  is defined to be the total number of sequences in  $S$  which are matched by  $p$ . Given a parameter  $\theta$  which is a positive integer and a user parameter, a sequence  $p$  is said to be *frequent* if the frequency of  $p$  is at least  $\theta$ . The problem of finding frequent attribute-based subsequence mining (FASM) is: given a parameter  $\theta$ , we want to find all possible frequent sequences in  $S$ .

There are at least two categories of finding frequent subsequence mining in the literature. The first category is *singleton-based mining* while the second category is *set-based mining*.

In singleton-based mining, a sequence is represented in form of “ $u_1, u_2, \dots, u_l$ ” where  $u_i \in \mathcal{V}$  for  $i \in [1, l]$ . At each timestamp, there is only at most one value  $\in \mathcal{V}$  in the subsequence [36]. Singleton-based mining is to find all frequent subsequences in the dataset which have their frequencies at least a given threshold  $\theta$ .

In set-based mining, a sequence is represented as a *set-sequence* in form of “ $G_1, G_2, \dots, G_l$ ” where  $G_i \subseteq \mathcal{V}$  for  $i \in [1, l]$  [82, 18]. At each timestamp, there can be more than one value  $\in \mathcal{V}$  in the set-sequence (which is represented by a set  $G_i$  instead of a value in  $\mathcal{V}$ ) [36]. In this category, the concept of subsequence (or set-subsequence) is defined differently as follows. Given a set-sequence  $g$  in form of “ $G_1, G_2, \dots, G_l$ ” where  $G_i \subseteq \mathcal{V}$  for  $i \in [1, l]$  and another set-sequence  $h$  in form of “ $H_1, H_2, \dots, H_{l'}$ ” where  $H_i \subseteq \mathcal{V}$  for  $i \in [1, l']$ ,  $g$  is said to be a *set-subsequence* of  $h$  if there exist  $l$  integers, namely  $j_1, j_2, \dots, j_l$ , such that (1) for each  $i \in [1, l]$ ,  $G_i \subseteq H_{j_i}$ , and (2)  $1 \leq j_1 < j_2 < \dots < j_l \leq l'$ . If  $g$  is a set-subsequence of  $h$ , then  $h$  is said to *contain*  $g$ . In set-based mining, each set-sequence in the dataset is in form of “ $G_1, G_2, \dots, G_l$ ”. The frequency of a set-sequence  $g$  is equal to the total number of set-sequences in the

dataset containing  $g$ . Set-based mining is to find all frequent set-subsequences in the dataset which have their frequencies at least a given threshold  $\theta$ .

Note that our FASM problem is a special case of the set-based mining. That is, all sequences found in our FASM problem can be found in the set of set-sequences found in the set-based mining. Since our FASM problem is a special case of the set-based mining, we adapt an existing algorithm in the literature of set-based mining for our FASM problem, by using our efficient query operator in ASM. In the literature, most algorithms for set-based mining requires to enumerate some potential candidates as the output and count the total number of sequences in the dataset which are matched by each candidate (query sequence). Our operator for ASM can be used in the counting step. Whenever we need to obtain the count for each candidate, we can perform our operator. Since these algorithms involves a large set of candidates and their counting step is not optimized, if their counting step is replaced by our operator, the efficiency of the algorithms can be improved a lot. In the experiment, we use the algorithm in [18] to illustrate how the operator can improve the performance of the algorithm. Any other choice is also possible.

Besides, our operator can also be used to solve a more general problem, the set-based mining, using the above approach. For each candidate in form of “ $G_1, G_2, \dots, G_l$ ” where  $G_i$  is a set of values for  $i \in [1, l]$ , we generate all possible sequences in form of “ $v_1, v_2, \dots, v_l$ ” where  $v_i \in G_i$  for  $i \in [1, l]$ . Each generated sequence can be regarded as a query sequence in ASM.

### 3.5 Experiment Results

We implemented above three algorithms in C/C++, namely *Naive*, *MA* and *MAI*. In Section 3.3, we described two possible straightforward approaches for problem ASM. Since the first approach takes an exponential time with respect to the length of the query

Number of sequences	$n$	250k, 500k, 750k, 1M
Average length of sequences	$\mu$	20, <b>40</b> , 60, 80, 100
Length of queries	$k$	5, <b>10</b> , 15, 20
Domain size of each attribute	$d$	<b>100</b> , 200, 30, 40
Number of attributes	$m$	5, 10, 15, 20

Table 3.5: Default values

length which is not scalable, we implemented the second straightforward approach (Algorithm 1), called *Naive*. *MA* is Modular Algorithm *without* inverted list which is Algorithm 7 where  $C(q)$  in line 2 is replaced by  $S$ . *MAI* is Modular Algorithm *with* inverted list which is Algorithm 7. For *MA* and *MAI*, we adopt the compressed version of the synopsis because it occupies less storage and have nearly the same execution time in Phase Preprocessing and Phase Query compared with the bulky version. We do not compare with existing sequence matching algorithm because our problem is a new problem and defined on a new kind of sequences. The existing algorithms for the traditional sequence matching problems are not applicable for our problem.

All the experiments were performed on a 2.4GHz PC with 4.0GB RAM, on a Linux platform. We did experiments on both synthetic and real datasets. For the synthetic datasets, we first generate the length of the sequence following a given Gaussian distribution with its mean equal to  $\mu$  and its standard derivation equal to 5 where  $\mu$  is a user parameter representing the average length of a sequence. In addition to the dataset size  $n$  and the average length of each sequence  $\mu$ , the synthetic data generator also simulates the number of attributes  $m$ , the size of each attribute domain  $d$  and the length of each query  $k$ . We assume these three values are fixed for all the sequences in a single dataset. In order to find at least one matching in the whole dataset, we extract each query sequence from an arbitrary sequence in the dataset. The values of each parameter used in the experiments are given in Table 3.5, where the default values are in bold. Finally, we generate the synthetic datasets according to every distinct parameter setting in Table 3.5.

In the experiments, we evaluate the algorithms with four measurements: (1) *Pre-processing Time*, (2) *Execution Time*, (3) *Storage* and (4) *Compression Ratio*. (1) Pre-processing time of *MA* and *MAI* corresponds to the time cost in Phase Preprocessing. *Naive* has no preprocessing step. So, we do not consider it. (2) Query time refers to the time an algorithm takes to answer 100 queries. Since the query time of *MA* (*MAI*) using the compressed synopsis is similar to the query time of *MA* (*MAI*) using the bulky synopsis, we only report *MA* (*MAI*) using the compressed synopsis. (3) Storage is the total memory consumption used for each data structure and the original dataset. The storage of *Dataset Size* is the memory occupied by the sequence data. The storage of *Inverted List* is the memory occupied by the inverted list. The storage of *Compressed Synopsis* is the memory occupied by the compressed version of  $X_s$  and  $V_s$ , while the storage of *Bulky Synopsis* is the storage occupied by the bulky version of  $X_s$  and  $V_s$ . (4) *Compression Ratio* is the ratio of the storage of Compressed Synopsis to that of Bulky Synopsis.

### 3.5.1 Effects of $n$ , $\mu$ , $k$ , $d$ and $m$ on Synthetic Datasets

We study the effects of  $n$ ,  $\mu$ ,  $k$ ,  $d$  and  $m$  as follows.

*Effect of database size  $n$* : Figure 3.2(a) shows that the preprocessing in *MAI* is slightly larger than that of *MA*. It is because that *MAI* needs to generate the inverted list, but *MA* does not. In Figure 3.2(b), when the execution time of *Naive* increased sharply when  $n$  increases, the execution time of *MA* and *MAI* increased slightly. As expected, the storage of *Bulky Synopsis* is much larger than that of *Dataset Size*, *Inverted List* and *Compressed Synopsis* in Figure 3.2(c). The compression ratio is around 24% as shown in Figure 3.2(d).

*Effect of sequence length  $\mu$* : Figures 3.3(a), (b) and (c) have similar trends as Figures 3.2(a), (b) and (c). In Figure 3.3(c), each data structure increases with  $\mu$ . Note that

the compressed synopsis also increases. In the compressed synopsis, each sequence is compressed into a 3-number synopsis which is independent of the sequence length. So, apparently, it seems that it should not increase with the sequence length. However, when the sequence becomes longer, the compressed synopsis needs much larger prime numbers and thus the synopsis representation needs more storage. Notice that, in Figure 3.3(d), as  $\mu$  increased, the compression ratio decreased slowly, which means the longer the sequence is, the smaller storage the compressed synopsis occupies compared with the bulky synopsis.

*Effect of  $k$ :* As expected, the length of the query sequence does not affect the processing time and the storage of every data structure, as shown in Figure 3.4(a), Figure 3.4(c) and Figure 3.4(d). From Figure 3.4(b), we can see that the execution time of *MA* and *MAI* remained unchanged while that of *Naive* increased slightly.

*Effect of  $d$ :* As  $d$  increases, the storage of the compressed synopsis increased slightly, so that the compression ratio also increased, as shown in Figure 3.5(c) and Figure 3.5(d). When  $d$  increased, the diversity of the sequences in the database also increased sharply, each generated sequence is much more dissimilar to other sequences. Consequently, the execution time of *MA* and *MAI* decreased in Figure 3.5(b).

*Effect of  $m$ :* When  $m$  increased, the processing time and storage increased, as shown in Figure 3.6(a) and Figure 3.6(c). But the compression ratio remained around 24% in Figure 3.6(d). But in Figure 3.6(b), the execution time of *MAI* remained when  $m$  increases. However, the execution time of *Naive* increased significantly. It means that *MAI* can deal with sequences that have a large number of attributes efficiently.

### 3.5.2 Results on Real Datasets

Besides the synthetic datasets, we also did experiments on three real datasets: Netflix [56], BookX [5], and Genealogy [3]. (1) Netflix is a famous movie rental com-

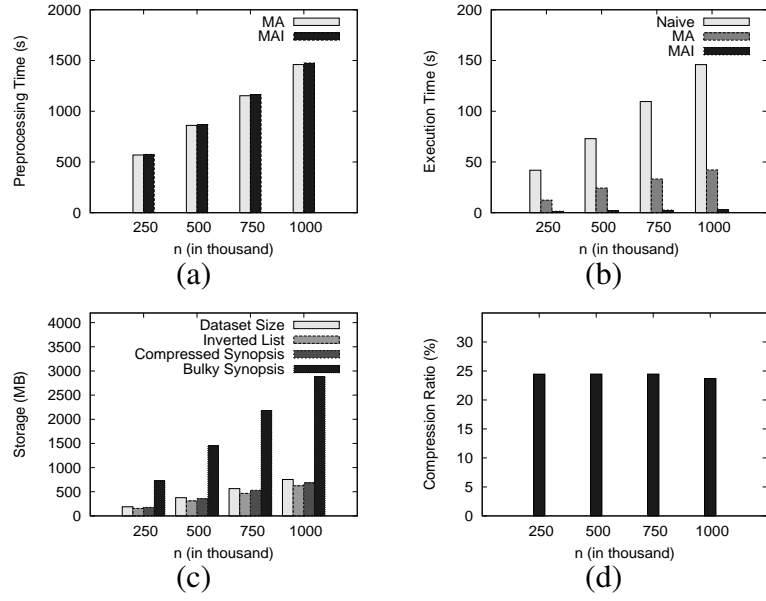


Figure 3.2: Effect of  $n$  (dataset size)

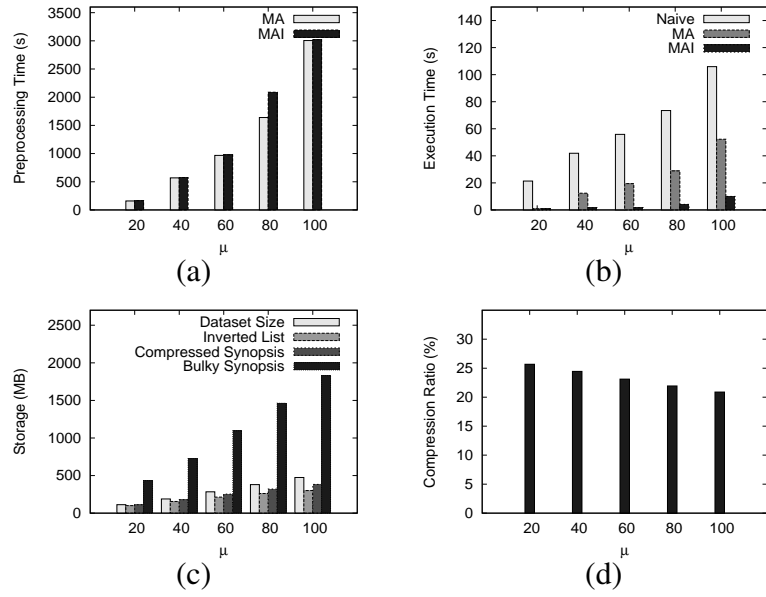


Figure 3.3: Effect of  $\mu$  (average length of a sequence)



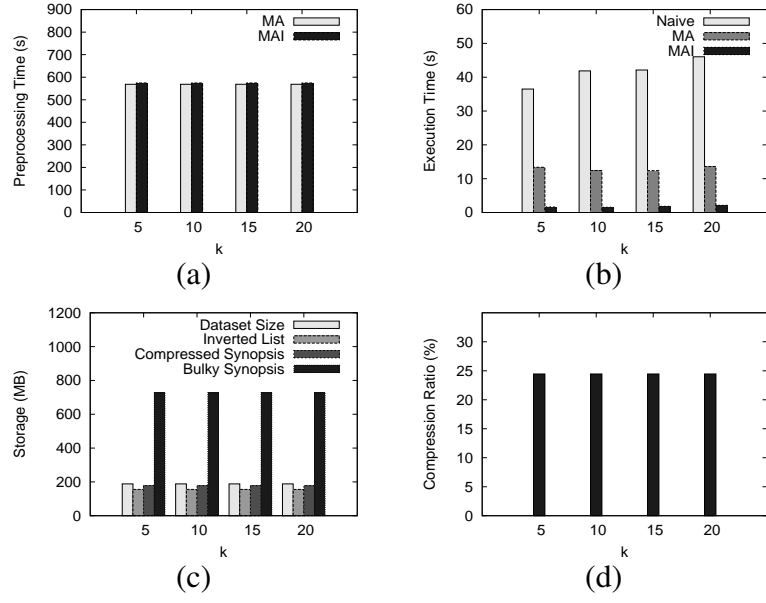


Figure 3.4: Effect of  $k$  (the length of a query sequence)

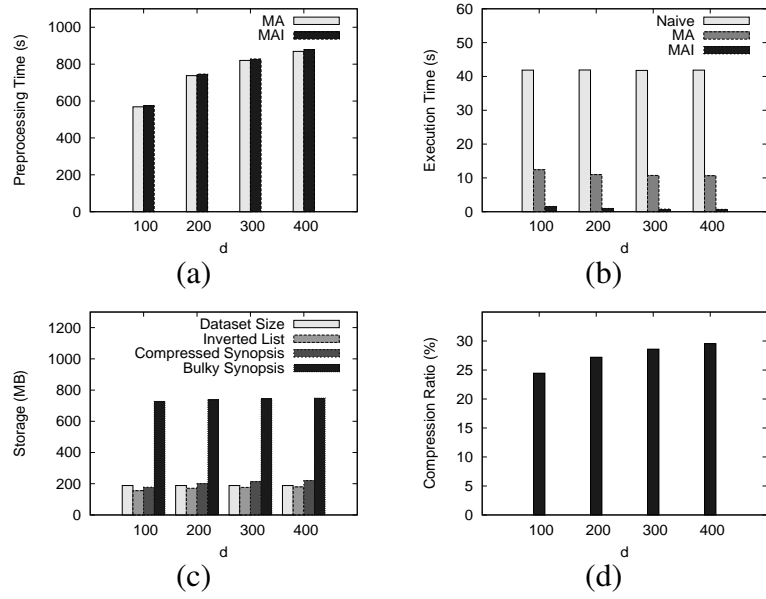


Figure 3.5: Effect of  $d$  (the domain size of each attribute)

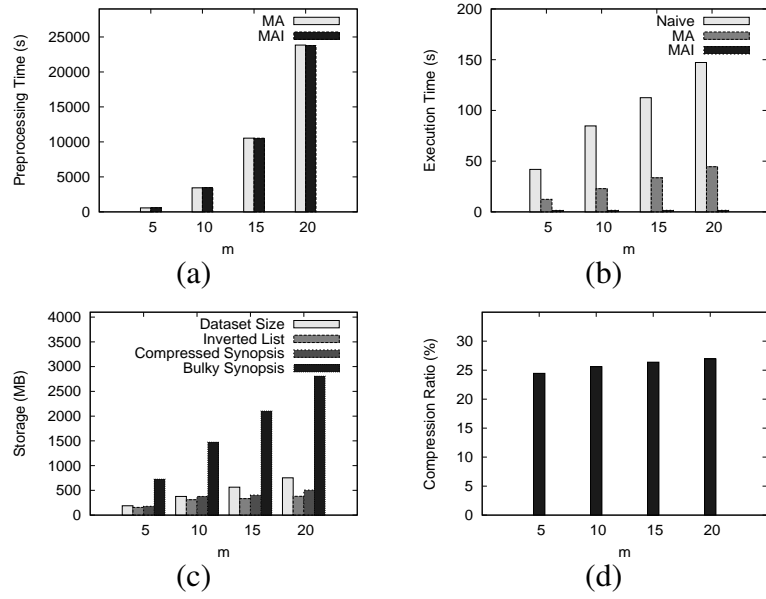


Figure 3.6: Effect of  $m$  (the number of attributes of each object)

Datasets	$n$	$m$	$\mu$	No. of Elements	Avg. Duplicates
Netflix	478905	3	157	7653	11.2%
BookX	91400	3	11.7	262554	4.2%
Genealogy	1000	3	2.5	1553	13.3%

Table 3.6: Statistics of real datasets

Dataset	Execution Time (s)			Compression Ratio (%)
	Naive	MA	MAI	
Netflix	205.939	85.446	6.744	31.382
BookX	2.489	2.034	0.681	30.978
Genealogy	0.011	0.004	0.003	21.699

Table 3.7: Execution time on real datasets

pany. We process the rating record dataset provided by Netflix to generate a rating sequence dataset through grouping the ratings by *customerID* (the identification of a customer) and sorting them by the rating date. (2) BookX (BookCrossing) is an online book searching and rating website. We download the ratings dataset and use a similar method to generate a sequence for every reader. (3) Genealogy dataset is collected by ourselves, which contains biographic sequences of 1000 researchers. Some statistics of the three datasets are shown in Table 3.6. The queries are generated by randomly selecting subsequences of sequences in each real dataset with a given length. In this table, *No. of Elements* is the number of elements appearing in this dataset, and *Avg. Duplicates* is the average proportion of duplicate attribute values in one sequence.

The first four columns in Table 3.7 shows the execution time on the three real datasets. The execution time of *MAI* is much smaller than that of *Naive* in every real dataset. The last column in Table 3.7 shows that the compression ratio of Netflix and BookX is around 30%, a little higher than that of the synthetic datasets. We summarize the other statistics of the experiments on real datasets. The greatest prime numbers used in encoding the three real datasets is 4,863,427. The greatest number used in the compressed synopsis contains 2100 digits. Although this number is large, the modular operation over this number can be done efficiently with the GMP library [1].

### 3.5.3 Results for FASM

#### A. Performance

We conducted experiments for the FASM problem on the BookX dataset. We insert *MA* as an operator in SPAM [18] for FASM. Due to the bitmap representation of database, SPAM cannot process attribute-based sequences of length more than 64, which are common in Netflix and BookX. The *SPAM* using our operator *MA* is denoted by *S-PAMMA*. We compare *SPAMMA* with the *SPAM* algorithm without using our *MA* op-

erator, denoted by *ASPAM*. The experimental results can be found in Figure 3.7 where  $\theta/n$  is the frequency threshold in fraction.

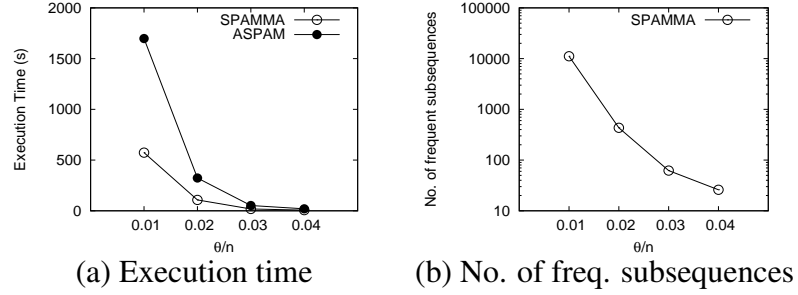


Figure 3.7: Results for FASM on BookX by *SPAMMA*

In Figure 3.7(a), the execution time of *SPAMMA* is much smaller than *ASPAM*. When  $\theta$  is larger, fewer subsequences are checked whether they are frequent according to the given threshold  $\theta$ . So, the resulting frequent subsequence set is also smaller, as shown in Figure 3.7(b).

## B. Case Study

By running *SPAMMA*, we found some interesting frequent subsequences. For example, when  $\theta/n$  is set to 0.01, we can find “<Deutscher Taschenbuch Verlag>, <Piper>” as a frequent subsequence. “Deutscher Taschenbuch Verlag” is a publisher, while “Piper” is a book. Note that “Deutscher Taschenbuch Verlag” and “Piper” belong to different attributes. We can interpret this frequent subsequence to a popular temporal user preference that people who prefer books from <Deutscher Taschenbuch Verlag> also prefer the book <Piper>. Another interesting case is a frequent subsequence, “<She’s Come Undone>”. It is a book with two versions published by two different publishers: one is Washington Square Press, and the other is Pocket Books. In the result, “<She’s Come Undone, Washington Square Press>” is also a frequent subsequence, but “<She’s Come Undone, Pocket Books>” is not. It means that compared with the Pocket Books version, readers prefer the Washington Square Press version. When we checked these two versions on Amazon [2], we found that the Washington Square Press version of this book has a higher paper quality. Maybe this is the reason why more peo-

ple choose the Washington Square Press version. We can see that, by comparing two related popular temporal user preferences, we can find more useful information on customer behaviors.

### 3.6 Conclusions and Future Directions

In this chapter, we proposed a new problem called *Attribute-based Subsequence Matching Problem* which has many applications. The main difference between our problem and traditional subsequence matching problems is that we consider properties of elements of sequences. We proposed an efficient algorithm for this problem using the Chinese Remainder Theorem to compress each sequence into a triplet of numbers. We also illustrated how our algorithm can be used for mining frequent subsequences. Finally, we conducted experiments to show that our algorithm is very efficient, nearly two orders of magnitude better than the straightforward method on synthetic datasets. We also embedded our algorithm into an existing algorithm to mine frequent attribute-based subsequences. Experimental results showed that our algorithm can be used in many applications which have subsequence matching operations. We had interesting findings in the frequent subsequences found, which show that the temporal user preferences found by our algorithms make sense in the reality.

In FASM, we did not consider how to find a proper threshold  $\theta$  as this is a given parameter. But users may have problems in selecting a proper  $\theta$  so that meaningful and reasonable subsequences are in the result set while redundant, meaningless and short subsequences are not. It is obvious that setting  $\theta$  to a too large value will result in missing a lot of meaningful and reasonable subsequences; Setting  $\theta$  to a too small value will generate an enormous result set with lots of redundant subsequences. Besides, for some of the subsequences, their frequencies are not so significant in the whole set, however, they may be discriminative in different groups of sequence owners. We have

an observation from real life, different groups have different interests in common. For example, young people are more likely to prefer movie “Avatar” to movie “City Lights” or “Modern Times”; Animation movies like “Ice Age” are preferred by children, but not so popular in other groups. This observation tells us that setting a fixed threshold to mine frequent subsequences in the whole dataset may miss some meaningful subsequences only significant in some groups of owners. In order to capture these missing meaningful subsequences, we measure the difference between the frequency of a subsequence with respect to a special group of sequence owners and that of the whole dataset and return sequences with a large difference only. We would like to leave it as a future work.

Another possible direction is to use our method to analyze protein sequences. Protein sequences are composed of twenty amino acids. Each amino acids may have different functions when appearing with some other amino acids as functional pairs or triples. It is interesting to apply our technique to find those functional combinations of amino acids and also analyze other common features of proteins with common functional combinations of amino acids.

## CHAPTER 4

# FINDING TOP- $K$ PROFITABLE AND TOP- $K$ POPULAR PRODUCTS

In this chapter, we study problems about *utilizing user preferences* when we consider generalized user preferences and tolerant user preferences. We first describe the preliminaries in Section 4.1. Then, we study the problem of *Finding Top- $k$  Profitable Products over Static Datasets* with solutions in Section 4.2, the problem of *Finding Top- $k$  Profitable Products over Dynamic Datasets* with solutions in Section 4.3 and the problem of *Finding Top- $k$  Popular Products* with solutions in Section 4.4, respectively. Section 4.5 gives the experimental settings. Extensive experimental results are given in Section 4.6, Section 4.7 and Section 4.8. Finally, we concludes this chapter in Section 4.9.

## 4.1 Preliminaries

### 4.1.1 Background: Skyline

The skyline analysis involves multiple attributes. The values in each attribute can be modeled by a partial order on the attribute. A *partial order*  $\preceq$  is a reflexive, asymmetric and transitive relation. A partial order is also a total order if for any two values  $u$  and  $v$  in the domain, either  $u \preceq v$  or  $v \preceq u$ . We write  $u \prec v$  if  $u \preceq v$  and  $u \neq v$ .

By default, we consider tuples in an  $w$ -dimensional<sup>1</sup> space  $\mathbb{S} = x_1 \times \cdots \times x_w$ . For each dimension  $x_i$ , we assume that there is a partial or total order. For a tuple  $p$ ,  $p.x_i$  is

---

<sup>1</sup>In this section, we use the terms “*attribute*” and “*dimension*” interchangeably.

Package	Distance-to-beach (km)	Price
$p_1$	7.0	200
$p_2$	4.0	350
$p_3$	1.0	500
$p_4$	3.0	600

Table 4.1: Packages in the existing market

Package	Distance-to-beach (km)	Price	Cost
$q_1$	5.0	?	100
$q_2$	4.5	?	200
$q_3$	0.5	?	400

Table 4.2: Potential packages in the new travel agency

the projection on dimension  $x_i$ . For dimension  $x_i$ , if  $p.x_i \preceq q.x_i$ , we also simply write  $p \preceq_{x_i} q$ . We can omit  $x_i$  if it is clear from the context.

For tuples  $p$  and  $q$ ,  $p$  *dominates*  $q$  with respect to  $\mathbb{S}$ , denoted by  $p \prec q$ , if for any dimension  $x_i \in \mathbb{S}$ ,  $p \preceq_{x_i} q$ , and there exists a dimension  $x_{i_0} \in \mathbb{S}$  such that  $p \prec_{x_{i_0}} q$ . If  $p$  dominates  $q$ , then  $p$  is more preferable than  $q$ . The set of tuples dominated by  $p$  is denoted by  $D(p)$ .

**Definition 4.1.1 (Skyline)** *Given a dataset  $\mathcal{D}$  containing tuples in space  $\mathbb{S}$ , a tuple  $p \in \mathcal{D}$  is in the Skyline of  $\mathcal{D}$  (i.e., a skyline tuple in  $\mathcal{D}$ ) if  $p$  is not dominated by any tuples in  $\mathcal{D}$ . The skyline of  $\mathcal{D}$ , denoted by  $SKY(\mathcal{D})$ , is the set of skyline tuples in  $\mathcal{D}$ .*

□

**Example 3 (Skyline)** *Consider that a customer is looking for a vacation package to Hannover using some travel agencies like Expedia.com [4] and Priceline.com [6]. The customer uses two criteria for choosing a package, namely price and distance-to-beach, where price is the price of a package and distance-to-beach is the distance between a hotel in a package and a beach. Table 4.1 shows four packages:  $p_1, p_2, p_3$  and  $p_4$ . The skyline set of  $P$  contains  $p_1, p_2$  and  $p_3$ .*

□



### 4.1.2 Notations

The *skyline* of a given dataset  $\mathcal{D}$  is denoted by  $SKY(\mathcal{D})$ . We have a set  $P$  of  $m$  tuples in the existing market, namely  $p_1, p_2, \dots, p_m$  (for example, packages in Table 4.1). Each tuple  $p$  has  $l$  attributes, namely  $A_1, A_2, \dots, A_l$ . The domain of each attribute is  $\mathbb{R}$ . The value of attribute  $A_j$  for tuple  $p$  is given and is denoted by  $p.A_j$  where  $j \in [1, l]$ . In particular, the last attribute  $A_l$  represents attribute price and all other attributes represent the attributes other than price.

Besides, we have a set  $Q$  of  $n$  potential new tuples, namely  $q_1, q_2, \dots, q_n$  (for example, packages in Table 4.2). Similarly, each tuple  $q$  has the same  $l$  attributes, namely  $A_1, A_2, \dots, A_l$ . The value of attribute  $A_j$  for tuple  $q$  is denoted by  $p.A_j$  where  $j \in [1, l]$ . However, the value of attribute  $A_l$  for tuple  $q$  is not given and the value of each of the other attributes is given. We assume that no two potential new tuples in  $Q$  are identical (i.e., no two tuples in  $Q$  have the same attribute values for  $A_1, A_2, \dots, A_{l-1}$ ). In addition to these  $l$  attributes, each tuple  $q$  is associated with one additional cost attribute  $C$ . The value of attribute  $C$  for  $q$  is denoted by  $q.C$ . We assume that for any two tuples in  $P \cup Q$ , they have at least one attribute value different among the first  $l - 1$  attributes. This assumption allows us to avoid several complicated, yet uninteresting, “boundary” cases. If this assumption does not hold, the proposed algorithms can be modified accordingly.

In our running example of Table 4.1 and Table 4.2,  $P$  contains 4 tuples, namely  $p_1, p_2, p_3$  and  $p_4$  (Table 4.1), and  $Q$  contains 3 tuples, namely  $q_1, q_2$  and  $q_3$  (Table 4.2). Attribute  $A_1$  and attribute  $A_2$  are “Distance-to-beach” and “Price”, respectively. Attribute  $C$  is “Cost”.

Let  $price_{max}$  be the greatest possible price of a tuple in  $P$ . We assume that the price of each tuple in  $Q$  should be set to a value at most  $price_{max}$ . This assumption makes sense since we do not want the price of each package too high compared with all

existing packages. Besides, since there are an infinite number of possible values in  $\mathbb{R}$ , we assume the domain of attribute “Price” (i.e.,  $A_l$ ) is defined to be  $D = \{i \cdot \sigma \mid i \text{ is a non-negative integer and } i \cdot \sigma \leq price_{max}\}$  where  $\sigma$  is a real number and a user parameter. If we want to have a finer granularity, we should set  $\sigma$  to a smaller value. This assumption makes sense in real applications where attribute Price involves discrete values instead of continuous values.

## 4.2 Finding Top- $k$ Profitable Products over Static Datasets

To differ from the problem studied in the next subsection, we specify that the problem studied in this subsection is over static datasets. In most part of this thesis, we call it *Finding Top- $k$  Profitable Products* (TPP) for short without any ambiguity.

### 4.2.1 Motivation

Consider that a new travel agency wants to start some new packages from a pool of potential packages as shown in Table 4.2, given the existing packages in the market as shown in Table 4.1. Table 4.2 shows three potential packages, namely  $q_1$ ,  $q_2$  and  $q_3$ . In this table, attribute distance-to-beach and attribute cost of each package are given. However, attribute price is to be determined by the agency.

**Example 4 (Profitable Price with One Package)** Suppose that we select only *one* new package, say  $q_1$ . What price should we set for package  $q_1$ ? If we set the price of  $q_1$  to be \$100, since the cost of  $q_1$  is \$100, the *profit* of  $q_1$  is equal to  $\$100 - \$100 = \$0$ . In other words, we cannot earn any profit. If we set the price to be \$400, although we can earn  $\$400 - \$100 = \$300$ , this new package  $q_1$  is dominated by  $p_2$  in the existing market. In other words, it is likely that no customer will select  $q_1$  since  $p_2$  is better than  $q_1$ .

However, if we set the price to be \$300, not only can we earn  $\$300 - \$100 = \$200$  but also  $q_1$  is not dominated by any packages in the existing market. We say that \$300 is a *profitable* price of  $q_1$  but \$100 and \$400 are not profitable prices of  $q_1$ .

Let us consider another example that we want to select only one new package  $q_2$  (instead of  $q_1$ ). Similarly, if we set the price to be \$200, the profit is \$0. If we set the price to be \$400,  $q_2$  is dominated by  $p_2$ . However, if we set the price to be \$300, we can earn \$100 and  $q_2$  is not dominated by any packages in the existing market. Thus, \$300 is a profitable price of  $q_2$  but \$200 and \$400 are not.  $\square$

Unfortunately, how we set the price of a new package may affect how we set the price of another new package.

**Example 5 (Profitable Price with Two Packages)** Suppose that we are interested in selecting *two* new packages, says  $q_1$  and  $q_2$ , instead of only one new package. From Example 4, if we set both the price of  $q_1$  and the price of  $q_2$  to be \$300 *separately*, then we can earn some profits and they are not dominated by any packages in the existing market. However, after we set these prices, the new package  $q_1$  is dominated by another new package  $q_2$ . An alternative price setting/assignment is that the prices of  $q_1$  and  $q_2$  are set to \$250 and \$300, respectively. In this assignment, it is easy to verify that  $q_1$  ( $q_2$ ) is not dominated by not only any packages in the existing market but also another new package  $q_2$  ( $q_1$ ). Besides, the profits of  $q_1$  and  $q_2$  are \$150 and \$100, respectively. The sum of these profits is equal to \$250.  $\square$

From the above example, we learn that how we set the price of a new package may affect how we set the price of another new package. Let  $Q$  be the set of potential new packages. In general, we want to select  $k$  packages from  $Q$  where  $k$  is a positive integer and is an input parameter. For example,  $k$  is equal to 2 in Example 5. We denote the set of these selected packages by  $Q'$ . Let  $F(Q')$  be a utility function on  $Q'$  which returns

a real number. Different sets for  $Q'$  can give different values of  $F$ . If the value of  $F$  is larger, then the set for  $Q'$  is more *preferable*. One example of  $F$  is a function which returns the sum of the profits of all packages in  $Q'$  as illustrated in Example 5.

In this section, we study the following problem: Given a set  $P$  of packages in the existing market and a set  $Q$  of potential new packages, we want to select a set  $Q'$  of  $k$  packages from  $Q$  such that the sum of the profits of the selected packages is maximized (Here,  $F(Q')$  is specified to be the sum of the profits of the selected packages) and each selected package is not dominated by any packages in the existing market and any selected new packages. We call this problem *finding top- $k$  profitable products (TPP)*.

Finding top- $k$  profitable products is common in many real life applications. Other applications include finding profitable laptops in a new laptop company, finding profitable delivery services in a new cargo delivery company and finding profitable e-advertisements in a webpage.

#### 4.2.2 Problem Definition

In Example 5, we selected  $q_1$  and  $q_2$  from  $Q$  when  $k = 2$ . However, when  $k$  is large, There exist many possible subsets containing  $k$  tuples from  $Q$ . Let us consider one *particular* subset  $Q'$ . The price of each of these  $k$  tuples (represented by attribute  $A_l$ ) in  $Q'$  is to be assigned with a value in  $D$ . Given a tuple  $q$  in  $Q$ , after we set  $q.A_l$  to a value  $v$ , the *profit* of  $q$ , denoted by  $\Delta(q, v)$ , is defined to be  $v - q.C$ .

We define a *price assignment vector* of  $Q'$ , denoted by  $\mathbf{v}$ , in form of  $(v_1, v_2, \dots, v_n)$ .  $v_i$  is said to be the  $i$ -th entry of  $\mathbf{v}$ . If  $q_i \in Q'$  where  $i \in [1, n]$ , then  $v_i$  is assigned with a value in  $D$ . Otherwise,  $v_i$  is set to 0.

A price assignment vector  $\mathbf{v}$  is said to be *feasible* if after we set the price of each  $q_i \in Q'$  to  $v_i$ , each  $q_i \in Q'$  is in the skyline with respect to  $P \cup Q'$ .

**Definition 4.2.1 (Profit of Selection)** Let  $Q'$  be a set of  $k$  tuples selected from  $Q$ . Let  $\mathbf{v}$  be the price assignment vector of  $Q'$  in form of  $(v_1, v_2, \dots, v_n)$ . The profit of  $Q'$  with  $\mathbf{v}$ , denoted by  $Profit(Q', \mathbf{v})$ , is defined to be  $\sum_{q_i \in Q'} \Delta(q_i, v_i)$ .  $\square$

**Definition 4.2.2 (Optimal Price Assignment Vector)** Let  $Q'$  be a set of  $k$  tuples selected from  $Q$ . Let  $\mathcal{V}$  be a set of all possible feasible price assignment vectors for  $Q'$ . The optimal price assignment vector of  $Q'$  is defined to be the price assignment vector  $\mathbf{v}_o$  for  $Q'$  such that

$$Profit(Q', \mathbf{v}_o) = \max_{\mathbf{v}' \in \mathcal{V}} Profit(Q', \mathbf{v}')$$

The optimal profit of  $Q'$ , denoted by  $Profit_o(Q')$ , is defined to be  $Profit(Q', \mathbf{v}_o)$  where  $\mathbf{v}_o$  is the optimal price assignment vector of  $Q'$ .  $\square$

In Section 4.2.3, we describe an efficient algorithm to find the optimal price assignment vector given a set  $Q'$  of  $k$  selected tuples.

We just learnt that given a *particular* set  $Q'$ , we can determine the optimal profit of  $Q'$ . However, there are many possible subsets of  $Q$  containing  $k$  tuples. The company wants to find a selection containing  $k$  tuples from  $Q$  such that the total profit is maximized.

**Problem 1 (Finding Top- $k$  Profitable Products)** Let  $\mathcal{Q}$  be the set of all possible subsets containing  $k$  tuples from  $Q$ . We want to select a set  $Q'$  of  $k$  tuples from  $Q$  such that  $Profit_o(Q') = \max_{Q'' \in \mathcal{Q}} Profit_o(Q'')$ .  $\square$

This problem is called *finding top- $k$  profitable products (TPP)*. Important notations used are in Table 4.3.

A naive way for this problem is to enumerate all possible subsets of size  $k$  from  $Q$ , calculate the optimal profit of each possible subset, and choose the subset with the

Notation	Description
$P$	a set of tuples in the existing market
$Q$	a set of potential new tuples
$m$	the size of $P$
$n$	the size of $Q$
$k$	the total number of tuples in $Q$ to be selected
$q.A_j$	the $j$ -th attribute value of tuple $q$
$l$	the number of the attributes of tuples in $P$
$Q'$	the set of top- $k$ profitable products
$Profit(Q', \mathbf{v})$	total profit of $Q'$ when the price vector is $\mathbf{v}$
$Profit_o(Q')$	the optimal profit of $Q'$
$\gamma(X, q)$	the set of all tuples in $X$ which quasi-dominate $q$
$SKY(P)$	the skyline of $P$
$\sigma$	the granularity of attribute price
$S(i, t)$	the set $Q'$ of size $t$ where $i \in [1, n]$ and $t \in [0, k]$ such that $Profit_o(Q') = \max_{Q'' \in \mathcal{Q}} Profit_o(Q'')$ where $\mathcal{Q}$ is the set of all possible subsets containing $t$ tuples from $Q(i)$
$\mathbf{v}(i, t)$	the optimal price assignment vector of set $S(i, t)$
$T(i, t)$	the (optimal) profit of set $S(i, t)$
$N$	$N$ is a positive integer where $N \ll m + n$

Table 4.3: Notation table

greatest profit. However, this approach is not scalable because there are an exponential number of all possible subsets. This motivates us to propose efficient algorithms for problem TPP which will be described in Section 4.2.4 and Section 4.2.5. Before that, we first introduce an algorithm embedded in both the dynamic programming approach and greedy approaches in Section 4.2.3.

### 4.2.3 Finding Optimal Price Assignment

We present an algorithm for finding the optimal price assignment called *AOPA* in  $O(k(\log(m + n) + N))$  time given a set  $Q'$  of size  $k$  where  $N \ll (m + n)$ .

Suppose that  $Q'$  is a selection set. Our objective is to find the optimal price assignment vector of  $Q'$ . After setting the prices of all tuples in  $Q'$  according to this vector, the tuples in  $Q'$  are in the skyline with respect to  $P \cup Q'$ .

Let  $X = P \cup Q'$ . Given  $p \in X$  and  $p' \in X$ ,  $p$  is said to *quasi-dominate*  $p'$  if (1)  $p$  dominates  $p'$  with respect to the first  $l - 1$  attributes, namely  $A_1, A_2, \dots, A_{l-1}$ , or (2)  $p$  has the same  $l - 1$  attribute values as  $p'$ . In our running example,  $l = 2$ . Suppose that  $Q' = \{q_1, q_2\}$ ,  $p_2$  quasi-dominates both  $p_1$  and  $q_1$  since  $p_2$  dominates both  $p_1$  and  $q_1$  with respect to attribute “Distance-to-Beach”.  $p_2$  also quasi-dominates  $q_2$ . Let  $\gamma(X, q_i)$  be a set containing all tuples in  $X$  which quasi-dominate  $q_i$ . For example, suppose that  $Q' = \{q_1, q_2\}$ .  $\gamma(X, q_1) = \{q_2, p_2, p_3, p_4\}$  and  $\gamma(X, q_2) = \{p_2, p_3, p_4\}$ .

The following lemma gives us an intuition of how to design an algorithm to find the optimal price assignment vector of  $Q'$ .

**Lemma 4.2.1** *Suppose that  $p \in X$  and  $q_i \in Q'$ . Consider that we are given a price assignment vector of  $Q'$  equal to  $\mathbf{v} = (v_1, v_2, \dots, v_n)$  such that we set the price of each  $q_j$  (i.e.,  $q_j.A_l$ ) in  $Q'$  to  $v_j$ ,  $j = 1, 2, \dots, n$ . If  $p$  dominates  $q_i$ , then  $p \in \gamma(X, q_i)$ .  $\square$*

According to the above lemma, we divide the tuples in  $X$  into two groups.

- **Group 1 (Outside  $\gamma$ ):** Group 1 is the set of all tuples not in  $\gamma(X, q)$  (more specifically, all tuples in  $X - \gamma(X, q)$ ). The tuples in this group do not dominate  $q$  regardless of any price assignment vector of  $Q'$ .

In our running example, let  $Q' = \{q_1, q_2\}$ . Consider  $q_1$ . Since  $p_1$  is not in  $\gamma(X, q_1)$ , we know that  $p_1$  does not dominate  $q_1$ .

- **Group 2 (Inside  $\gamma$ ):** Group 2 is the set of all tuples in  $\gamma(X, q)$ . For a particular price assignment vector of  $Q'$ , some tuples in this group may dominate  $q$  while for another particular price assignment vector of  $Q'$ , they may not dominate  $q$ .

For example, consider  $q_1$  again. Consider a price assignment vector  $\mathbf{v} = (300, 300, 0)$ .

Note that  $q_2$  is in  $\gamma(X, q_1)$ . After we set the prices of  $q_1$  and  $q_2$  with vector  $\mathbf{v}$ ,  $q_2$  dominates  $q_1$ .

Consider another price assignment vector  $\mathbf{v}' = (250, 300, 0)$ . After we set the prices of  $q_1$  and  $q_2$  with vector  $\mathbf{v}'$ ,  $q_2$  does not dominate  $q$ .

Our objective is to make sure that each tuple  $q \in Q'$  is in the skyline with respect to  $X (= P \cup Q')$ . That is, each tuple  $q$  in  $Q'$  is not dominated by any tuple in  $X$ . This is our goal. Consider Group 1 (Outside  $\gamma$ ). We can achieve the goal because all tuples in this group do not dominate  $q$ . Consider Group 2 (Inside  $\gamma$ ). It is possible that some tuples in  $\gamma(X, q)$  dominate  $q$  for a particular price assignment vector. For another price assignment vector, they do not dominate  $q$ .

In the above, we learn that if we want to determine the price of  $q_i$  in  $Q'$  such that  $q_i$  is in the skyline, we only need to consider the tuples in  $\gamma(X, q)$ .

Given a tuple  $q \in Q'$ , we know that only the tuples in  $X$  quasi-dominating  $q$  affect the price of  $q$ . Note that the prices of all tuples in  $P$  are given and the prices of all tuples in  $Q'$  are to be found. Thus, according to the quasi-dominance relationship, we design a *progressive* algorithm which finds the price of each tuple  $q$  in  $Q'$  by the following principle.

**Principle 1** *Whenever we want to find the price of  $q_i$  in  $Q'$ , we make sure that the prices of all tuples in  $Q'$  quasi-dominating  $q_i$  have already been determined.*

Next, we need to determine the ordering of processing tuples in  $Q'$  which follows the above principle. We define the following monotonically increasing function  $f$  which can determine the ordering. Given a tuple  $q$  in  $Q$ , function  $f$  is defined as follows.

$$f(q) = \sum_{i=1}^{l-1} q \cdot A_i$$

In our running example,  $l = 2$ . The  $f$  value of each tuple  $q \in Q$  can be found in Table 4.4. The ordering of tuples in  $X$  is  $q_3, q_2$  and  $q_1$ .



Tuple	$f$
$q_1$	5.0
$q_2$	4.5
$q_3$	0.5

Table 4.4: The  $f$  value of each tuple  $p \in Q$

The  $f$  function can be used to evaluate tuples. According to our setting for each attribute, the smaller value the better it is. Therefore, a smaller value of  $f$  function gives a better tuple. Different attributes have different physical meanings. It is hard to directly align the  $f$  function with some physical meaning, but we can use  $f$  function to sort tuples, which results in a reasonable sorting that is related to *quasi-dominate*. With this function  $f$ , we know the following lemma.

**Lemma 4.2.2** *Suppose  $p$  and  $p'$  are in  $X$ . If  $p$  quasi-dominates  $p'$ , then  $f(p)$  is smaller than or equal to  $f(p')$ .*  $\square$

For example, since  $p_2$  quasi-dominates  $p_1$ ,  $f(p_2)(= 4)$  is smaller than  $f(p_1)(= 7)$ .

With the above lemma, we can first compute the  $f$  values of all tuples in  $Q'$ . We sort the tuples in  $Q'$  in ascending order of these  $f$  values. Then, we determine the price of each tuple  $q$  in  $Q'$  according to this ordering, which follows Principle 1.

After we obtain the ordering of processing the tuples in  $Q'$ , we present an algorithm to determine the optimal price assignment vector of  $Q'$  incrementally.

Without loss of generality, we assume that  $q_1, q_2, \dots, q_k$  are the tuples in  $Q'$  sorted in ascending order of the  $f$  values. Let  $Q_0 = \emptyset$ . Let  $Q_i = Q_{i-1} \cup \{q_i\}$  where  $i = 1, 2, 3, \dots, k$ .

**Lemma 4.2.3** *Suppose that  $p \in X$  and  $q_i \in Q'$ . Consider that we are given the optimal price assignment vector of  $Q_{i-1}$  equal to  $\mathbf{v}_{i-1} = (v_1, v_2, \dots, v_n)$  such that we set the price of each  $q_j$  (i.e.,  $q_j.A_l$ ) in  $Q_{i-1}$  to  $v_j$ . Suppose that  $\gamma(X, q_i) \neq \emptyset$ . Let  $\mathbf{v}_i$  be a price assignment vector equal to  $\mathbf{v}_{i-1}$  except that the  $i$ -th entry of  $\mathbf{v}_i$  is set to  $(\min_{p \in \gamma(X, q_i)} p.A_l) - \sigma$ .  $\mathbf{v}_i$  is the optimal price assignment vector of  $Q_i$ .*

By Lemma 4.2.3, we can derive a progressive algorithm as shown in Algorithm 9.

---

**Algorithm 9** Algorithm **AOPA**( $Q'$ )

---

**Require:** A set  $Q'$  of tuples in  $Q$

**Ensure:** the optimal profit assignment vector of  $Q'$

- 1:  $Q'' \leftarrow \emptyset$
  - 2:  $\mathbf{v} \leftarrow (0, 0, \dots, 0)$
  - 3: **for** each  $q_i \in Q'$  (which is processed in the sorted ordering) **do**
  - 4:    $\mathbf{v} \leftarrow \text{findOptimalIncrementalPrice}(q_i, Q'', \mathbf{v})$  (See Algorithm 10)
  - 5:    $Q'' \leftarrow Q'' \cup \{q_i\}$
  - 6: **return**  $\mathbf{v}$
- 

---

**Algorithm 10** Algorithm **findOptimalIncrementalPrice**( $q_i, Q_{i-1}, \mathbf{v}_{i-1}$ )

---

**Require:** A set  $Q_{i-1} (= \{q_1, q_2, \dots, q_{i-1}\})$ , tuple  $q_i$  in  $Q'$  and the optimal price assignment vector  $\mathbf{v}_{i-1}$  of  $Q_{i-1}$

**Ensure:** the optimal price assignment vector  $\mathbf{v}_i$  of  $Q_i$

- 1:  $\mathbf{v}_i \leftarrow \mathbf{v}_{i-1}$
  - 2: find a set  $Y$  containing all tuples in  $P \cup Q'$  which quasi-dominate  $q_i$
  - 3: **if**  $Y \neq \emptyset$  **then**
  - 4:    $v \leftarrow (\min_{p \in Y} p.A_l) - \sigma$
  - 5: **else**
  - 6:    $v \leftarrow \infty$
  - 7: set the  $i$ -th entry in  $\mathbf{v}_i$  to  $v$
  - 8: **return**  $\mathbf{v}_i$
- 

With Lemma 4.2.3, it is easy to verify the following theorem.

**Theorem 4.2.1** *Given a set  $Q'$  of  $k$  tuples selected from  $Q$ , Algorithm AOPA returns the optimal price assignment of  $Q'$ .* □

**Implementation and Time Complexity:** In Algorithm 10, the most time-consuming operation is the step of finding all tuples in  $P \cup Q'$  which quasi-dominate  $q_i$ . One possible implementation is to build an R\*-tree index on dataset  $P \cup Q'$  according to the first  $l - 1$  attributes. If we perform a range query with the range equal to “ $A_i \leq q_i.A_i$ ” for each  $i \in [1, l - 1]$ , then we can find all tuples in  $P \cup Q'$  which quasi-dominate  $q_i$ . However, with this implementation, we have to build different indexes on  $P \cup Q'$  for different selection sets  $Q'$ , which is not efficient.

Another possible implementation is to build an R\*-tree index on dataset  $P \cup Q$  (instead of  $P \cup Q'$ ) according to the first  $l - 1$  attributes. Similarly, we perform a

range query with the same range as above and find all tuples in  $P \cup Q$  which quasi-dominate  $q_i$ . In this implementation, since  $Q' \subseteq Q$ , we can do a post-processing step to select all tuples in  $P \cup Q'$  which quasi-dominate  $q_i$ . This implementation has its advantage that we only need to build an index once for any selection set. We adopt this implementation in our experiment.

Suppose that we are given an R\*-tree index on dataset  $P \cup Q$  in the second implementation. We want to analyze the time complexity of the step of finding all tuples in  $P \cup Q'$  which quasi-dominate  $q_i$ . In most cases, the cost of a range query is  $O(\log(|P| + |Q|) + N)$  where  $N$  is the total number of tuples returned in a range query. Typically,  $N$  is extremely small compared with  $(|P| + |Q|)$ . That is,  $N \ll |P| + |Q|$ . The post-processing step takes  $O(N)$  time. Thus, the step of finding all tuples in  $P \cup Q'$  which quasi-dominate  $q_i$  takes  $O(\log(|P| + |Q|) + N)$  time.

After we analyze the time complexity of this time-consuming operation, it is easy to verify that Algorithm 10 takes  $O(\log(|P| + |Q|) + N)$  time.

Consider Algorithm 9. Since there are  $|Q'|$  iterations (in lines 3-5) and each iteration calls Algorithm 10 (which takes  $O(\log(|P| + |Q|) + N)$  time), the overall time complexity of Algorithm 9 is  $O(|Q'|(\log(|P| + |Q|) + N))$ . Since  $m = |P|$ ,  $n = |Q|$  and  $k = |Q'|$ , the time complexity becomes  $O(k(\log(m + n) + N))$ .

#### 4.2.4 Dynamic Programming Approach

In this section, we present a dynamic programming approach which finds an optimal solution for problem TPP when  $l = 2$ .

Consider  $l = 2$ . Without loss of generality, we assume that  $q_1, q_2, \dots, q_n$  are sorted in ascending order of the  $f$  values.  $Q(i)$  is defined to be a set of tuples in  $Q$  such that these tuples are quasi-dominated by  $q_i$ . Let  $S(i, t)$  denote the set  $Q'$  of size  $t$  where  $i \in [1, n]$  and  $t \in [0, k]$  such that  $Profit_o(Q') = \max_{Q'' \in Q} Profit_o(Q'')$  where  $Q$

is the set of all possible subsets containing  $t$  tuples from  $Q(i)$ . Let  $\mathbf{v}(i, t)$  denote the optimal price assignment vector of set  $S(i, t)$ . Let  $T(i, t)$  denote the (optimal) profit of set  $S(i, t)$ .

Let us use a symbol  $\alpha(q_i, S, \mathbf{v})$  to represent **findOptimalIncrementalPrice** $(q_i, S, \mathbf{v})$ .

In the following, we describe how to find three variables, namely  $\mathbf{v}(i, t)$ ,  $T(i, t)$  and  $S(i, t)$ . Consider two cases.

- **Case 1:**  $q_i$  is included in the final selection of size  $t$ .

By Lemma 4.2.3, the optimal price assignment vector of  $S(i, t)$ , denoted by  $\mathbf{v}(i, t)$ , can be obtained from the optimal price assignment vector of  $S(i - 1, t - 1)$ , denoted by  $\mathbf{v}(i - 1, t - 1)$ .

Thus, we have the following equation.

$$\mathbf{v}(i, t) = \alpha(q_i, S(i - 1, t - 1), \mathbf{v}(i - 1, t - 1)) \quad (4.1)$$

Thus,

$$T(i, t) = T(i - 1, t - 1) + v \quad (4.2)$$

where  $v$  is the  $i$ -th entry in  $\mathbf{v}(i, t)$ . Let  $T_{select} = T(i - 1, t - 1) + v$ .

Similarly,  $S(i, t)$  can be obtained as follows.

$$S(i, t) = S(i - 1, t - 1) \cup \{q_i\} \quad (4.3)$$

- **Case 2:**  $q_i$  is not included in the final selection.

We have

$$\mathbf{v}(i, t) = \mathbf{v}(i - 1, t) \quad (4.4)$$

We have

$$T(i, t) = T(i - 1, t) \quad (4.5)$$

and

$$S(i, t) = S(i - 1, t) \quad (4.6)$$

Let  $T_{notSelect} = T(i - 1, t)$ .

Note that we want to maximize the profit of the selection set of size  $t$ . Obviously, if  $T_{select} \geq T_{notSelect}$ , we should select  $q_i$  in the selection set (which corresponds to Case 1). Otherwise, we should not select  $q_i$  (which corresponds to Case 2).

The pseudo-code of the dynamic programming approach is shown in Algorithm 11.

---

**Algorithm 11** Dynamic programming approach

---

**Require:**  $P, Q$  and  $k$

**Ensure:** the final selection  $Q'$  of size  $k$  and the optimal price assignment vector  $\mathbf{v}$  of  $Q'$

```

1: for  $i = 1$  to  $n$  do
2:    $T(i, 0) \leftarrow 0$ 
3:    $S(i, 0) \leftarrow \emptyset$ 
4:   for  $t = 1$  to  $k$  do
5:      $T(1, t) \leftarrow \alpha(q_1, \emptyset, (0, 0, \dots, 0))$ 
6:      $S(1, t) \leftarrow \{q_1\}$ 
7:   for  $t = 1$  to  $k$  do
8:     for  $i = 1$  to  $n$  do
9:        $\mathbf{v}_{select} \leftarrow \alpha(q_i, S(i - 1, t - 1), \mathbf{v}(i - 1, t - 1))$ 
10:       $v \leftarrow$  the  $i$ -th entry in  $\mathbf{v}_{select}$ 
11:       $T_{select} \leftarrow T(i - 1, t - 1) + v$ 
12:       $T_{notSelect} \leftarrow T(i - 1, t)$ 
13:      if  $T_{select} \geq T_{notSelect}$  then
14:        // Case 1:  $q_i$  is selected
15:         $\mathbf{v}(i, t) \leftarrow \alpha(q_i, S(i - 1, t - 1), \mathbf{v}(i - 1, t - 1))$ 
16:         $T(i, t) \leftarrow T_{select}$ 
17:         $S(i, t) \leftarrow S(i - 1, t - 1) \cup \{q_i\}$ 
18:      else
19:        // Case 2:  $q_i$  is not selected
20:         $\mathbf{v}(i, t) \leftarrow \mathbf{v}(i - 1, t)$ 
21:         $T(i, t) \leftarrow T_{notSelect}$ 
22:         $S(i, t) \leftarrow S(i - 1, t)$ 
23:  $Q' \leftarrow Q(n, t)$ 
24:  $\mathbf{v} \leftarrow \mathbf{v}(n, t)$ 
25: return  $Q'$  and  $\mathbf{v}$ 

```

---

**Theorem 4.2.2** *Algorithm Dynamic Programming returns an optimal solution  $Q'$  of size  $k$  for problem TPP (i.e., the set  $Q'$  of size  $k$  with the greatest profit) when  $l = 2$ .*

**Time Complexity:** Consider Algorithm 11. Statements from line 1 to line 3 takes  $O(n)$  time while statements from line 4 to line 6 takes  $O(k \cdot k(\log(m + n) + N)) = O(k^2(\log(m + n) + N))$  time. Consider statements from line 7 to line 22. There are  $O(kn)$  iterations where the statements from line 9 to line 22 correspond to an iteration. It is easy to verify that each iteration takes  $O(k(\log(m + n) + N) + n)$  time. Thus, statements from line 7 to line 22 takes  $O(kn(k(\log(m + n) + N) + n)) = O(k^2n(\log(m + n) + N) + kn^2)$  time.

#### 4.2.5 Greedy Algorithms

In the previous section, we described a dynamic programming approach which finds an optimal solution when  $l = 2$ . However, when  $l > 2$ , we show that the problem is NP-hard as follows.

**Theorem 4.2.3** *When  $l > 2$ , problem TPP is NP-hard.* □

Since the problem is NP-hard, we propose two greedy algorithms for this problem. As we described in Example 5, the price of a new selected tuple may affect the price of another new selected tuple. We call this phenomenon a *price correlation*. The first version of the greedy algorithm is the algorithm which selects tuples in  $Q$  iteratively without considering the price correlation. The first greedy algorithm returns a solution with a theoretical guarantee on the profit. The second version is the algorithm which selects tuples in  $Q$  iteratively considering the price correlation. The second greedy algorithm performs well empirically.

---

**Algorithm 12** Greedy algorithm (Version 1)

---

```
1:  $\mathbf{v} \leftarrow (0, 0, \dots, 0)$ 
2: for each  $q \in Q$  do
3:   find the standalone profit of  $q$ 
4:  $Q' \leftarrow$  a set of the  $k$  tuples which have the greatest standalone profits
5: return  $Q'$ 
```

---

**Greedy Based Algorithm I**

The first version of the greedy algorithm is the algorithm which selects tuples in  $Q$  iteratively without considering the price correlation.

For each tuple  $q$  in  $Q$ , we first define the optimal profit of the selection set containing  $q$  only. We call this profit the *standalone profit* of  $q$ .

**Definition 4.2.3 (Standalone Profit)** *Given a tuple  $q$  in  $Q$ , the standalone profit of  $q$ , denoted by  $SP(q)$ , is defined to  $Profit_o(\{q\})$ .*

The first version of the greedy algorithm is described as follows. Specifically, for each tuple  $q$  in  $Q$ , we find the standalone profit of  $q$ . Then, we choose  $k$  tuples which have the greatest standalone profits. This version is shown in Algorithm 12.

Although this greedy approach is a heuristical approach, it has theoretical guarantees on the profit returned by the algorithm.

Suppose that  $O$  is the *optimal selection set* for problem TPP (i.e., the selection set which has the greatest profit). Note that the optimal profit of  $O$  is equal to  $Profit_o(O)$ . Recall that we want to maximize the profit, due to the heuristical nature of the greedy algorithm, this algorithm may return a selection  $Q'$  which has a lower profit (which is equal to  $Profit_o(Q')$ ). It is easy to verify that

$$Profit_o(Q') \leq Profit_o(O)$$

In the following, we give two theoretical results about the error guarantee on the profit

returned by the algorithm. The first result corresponds to an *additive error guarantee* while the second one corresponds to a *multiplicative error guarantee*.

**Theorem 4.2.4** *Let  $O$  be the optimal selection set and  $Q'$  be the selection set returned by Algorithm 12.*

$$Profit_o(O) - \epsilon_{add} \leq Profit_o(Q')$$

where

$$\epsilon_{add} = \frac{k(k-1)}{2}\sigma$$

□

**Theorem 4.2.5** *Let  $O$  be the optimal selection set and  $Q'$  be the selection set returned by Algorithm 12. Suppose that  $Profit_o(Q') > 0$ . Let  $\Delta = \sum_{q_i \in Q'} SP(q_i)$ . Algorithm 12 is a  $(1 - \epsilon_{mult})$ -approximate algorithm. That is,*

$$Profit_o(Q') \geq (1 - \epsilon_{mult})Profit_o(O)$$

where  $\epsilon_{mult} = \frac{k(k-1)\sigma}{2\Delta}$ .

□

**Time Complexity:** Consider Algorithm 12. We need to calculate the standalone profit of  $q$  for each tuple  $q \in Q$ . This step takes  $O(k(\log(m+n) + N))$  time. Then, we need to choose the  $k$  tuples which have the greatest standalone profits, which can be done in  $O(k \log k)$  time. Thus, the time complexity of Algorithm 12 is  $O(k(\log(m+n) + N) + k \log k)$ . Since  $k = O(m+n)$ , the complexity becomes  $O(k(\log(m+n) + N)) = O(k \log(m+n) + kN)$ .

## Greedy Based Selection II

In the previous subsection, we describe the first version of the greedy algorithm which does not consider the price correlation. In this subsection, we describe the second



---

**Algorithm 13** Greedy algorithm (Version 2)

---

```
1:  $Q' \leftarrow \emptyset$ 
2: while  $|Q'| \leq k$  do
3:   for each  $q_i \in Q$  do
4:      $x_i \leftarrow \text{AOPA}(Q' \cup \{q_i\})$ 
5:   find the tuple  $q_i$  in  $Q$  such that  $q_i$  has the greatest value of  $x_i$ 
6:    $Q' \leftarrow Q' \cup \{q_i\}$ 
7: return  $Q'$ 
```

---

version of the greedy algorithm which select tuples in  $Q$  iteratively considering the price correlation.

The second version of our greedy algorithms is shown in Algorithm 13.

**Time Complexity:** Consider Algorithm 13. There are  $O(k)$  iterations where statements from line 3 to line 6 correspond to an iteration. Consider an iteration. Statements from line 3 to line 4 take  $O(n \cdot k(\log(m+n) + N)) = O(nk(\log(m+n) + N))$  time. Statements from line 5 to line 6 takes  $O(n \log n)$  time. Thus, each iteration takes  $O(nk(\log(m+n) + N) + n \log n) = O(nk(\log(m+n) + N))$  time. The overall time complexity of Algorithm 13 is  $O(k \cdot nk(\log(m+n) + N)) = O(nk^2(\log(m+n) + N))$ . Note that compared with the time complexity of Algorithm 12 (i.e.,  $O(k \log(m+n) + kN)$ ), the time complexity of Algorithm 13 is higher.

#### 4.2.6 Extension with the $h$ -dominance Constraint

In problem TPP, after we set the price of each tuple in the selection set  $Q'$ , we know that each of these tuples is in the skyline with respect to  $P \cup Q'$ . In other words, after we set the price of each tuple in  $Q'$ , each of these tuples is *one* of the best choices for the customer to choose (because there may be more than one tuple in the skyline). In order to make sure that each tuple in  $Q'$  will be chosen by a customer in the market with a higher probability, we would like to set the price of each of these tuples such that not only each of these tuples is in the skyline but also each of these tuples dominates at least  $h$  tuples in the existing market  $P$  where  $h$  is an input parameter. This problem is

called Finding top- $k$  profitable products (TPP) with the  $h$ -dominance constraint. The  $h$ -dominance constraint corresponds to that each of these tuples dominates at least  $h$  tuples in the existing market  $P$ . If we set  $h=0$ , then the new problem becomes problem TPP without the  $h$ -dominance constraint. The algorithm proposed for the original problem can be adapted easily for the new problem. Details can be found in [91].

### 4.3 Finding Top- $k$ Profitable Products over Dynamic Datasets

In the above section, we discussed how to find top- $k$  profitable products over *static* datasets  $P$  and  $Q$ . However, in some cases, datasets are *dynamic* and change from time to time. In this section, we study the problem of finding top- $k$  profitable products on *dynamic* datasets. Since there are two kinds of datasets, namely  $P$  and  $Q$ , which can change over time, in the following, we focus on studying how to find top- $k$  profitable products when  $P$  changes. We do not discuss the case when  $Q$  changes because similar conclusions can also be drawn.

We study three kinds of *operations* on  $P$ , namely insertion, deletion and modification. Suppose that  $o$  is one of the three operations. After  $o$  is executed on  $P$ , we obtain a new dataset denoted by  $P_{new}$ . Specifically, we have the following operations and the corresponding  $P_{new}$ .

1. (Insertion) A tuple  $p_{new}$  is inserted into  $P$ . Then,  $P_{new} = P \cup \{p_{new}\}$ .
2. (Deletion) A tuple  $p$  is removed from  $P$ . Then,  $P_{new} = P - \{p\}$ .
3. (Modification) Some of the attribute values of a tuple  $p$  in  $P$  are changed and  $p$  becomes  $p'$ . Then,  $P_{new} = (P - \{p\}) \cup \{p'\}$ .

Note that a modification operation can be regarded as a sequence of the other two

operations (i.e., a deletion operation and then an insertion operation). It is sufficient to describe how we execute the deletion operation and the insertion operation. We introduce how we execute the insertion operation and deletion operation in Section 4.3.1 and Section 4.3.2, respectively.

**Problem 2 (Dynamic TPP)** *Let  $o$  be an operation and  $P_{new}$  be the resulting set  $P$  after  $o$  is executed on  $P$ . We want to find a set  $Q'_{new}$  of top- $k$  profitable products based on  $P_{new}$  and  $Q$ .* □

A straightforward approach is to run one of the algorithms in Section 4.2.5 on the new datasets  $P_{new}$  and  $Q$  from *scratch* whenever there is an operation. However, it is very costly because this approach does not make use of some useful results computed before the operation is executed. Instead, we propose an *incremental* algorithm to find a set  $Q'_{new}$  of top- $k$  profitable products based on not only the new datasets  $P_{new}$  and  $Q$  but also the previous result  $Q'$  (computed based on the previous datasets  $P$  and  $Q$ ).

Since  $P$  changes over time, it is desirable to design an efficient algorithm. In the following, we give an *incremental* version of Greedy1 because Greedy1 is more efficient compared with Greedy2. This incremental version which is based on not only the new datasets  $P_{new}$  and  $Q$  but also the previous result  $Q'$ , and returns the same selection set as the original Greedy1 which is based on the new datasets only.

Now, we focus on describing how the insertion operation is executed.

### 4.3.1 Insertion into $P$

Suppose that a new tuple  $p_{new}$  is inserted into  $P$  and then  $P_{new}$  is formed. Before we discuss our incremental algorithm, we first give two lemmas or properties for the insertion operation.

**Lemma 4.3.1 ( $P_{new}$ -Based Property)** *If  $p_{new} \notin SKY(P_{new})$ , then  $Q'_{new} = Q'$ .* □

---

**Algorithm 14** Incremental Algorithm for Insertion

---

```
1: // Checking  $P_{new}$ -Based Property (specified in Lemma 4.3.1)
2: if  $p_{new} \notin SKY(P_{new})$  then
3:    $Q'_{new} \leftarrow Q'$ 
4: else
5:   // Checking  $Q'$ -Based Property (specified in Lemma 4.3.2)
6:   if there does not exist  $q \in Q'$  such that  $p_{new}$  dominates  $q$  then
7:      $Q'_{new} \leftarrow Q'$ 
8:   else
9:     // need to calculate  $Q'_{new}$  incrementally
10:    for each  $q \in Q$  which is quasi-dominated by  $p_{new}$  do
11:      re-compute the standalone profit of  $q$ 
12:     $Q'_{new} \leftarrow$  a set of the  $k$  tuples in  $Q$  which have the greatest standalone profits
13: return  $Q'_{new}$ 
```

---

**Lemma 4.3.2 ( $Q'$ -Based Property)** *There does not exist  $q \in Q'$  such that  $p_{new}$  dominates  $q$  if and only if  $Q'_{new} = Q'$ .* □

According to Lemma 4.3.1 and Lemma 4.3.2, we design an incremental algorithm as shown in Algorithm 14.

**Theorem 4.3.1** *Algorithm 14 returns a selection set  $Q'_{new}$  which is equal to the selection set returned by Greedy1.* □

The complexity of checking the  $P_{new}$ -Based Property (Line 2 to Line 3) is  $O(m)$ . Otherwise, we can check the  $Q'$ -Based Property (Line 6 to Line 7), which takes  $O(k)$  time. If both of the properties are not satisfied, then we need to recompute  $SP(q)$  for each  $q \in Q$  which is quasi-dominated by  $p_{new}$ . For other  $q$  which is not quasi-dominated by  $p_{new}$ , their standalone profits are the same as before. Similar to the time complexity analysis in Section 4.2.5, the step (Line 10 to line 11) takes  $O(u \cdot ((m + n + N)))$  time where  $u$  is the total number of tuples in  $Q$  quasi-dominated by  $p_{new}$ . Finally, we choose  $k$  tuples which have the greatest standalone profits. It can be done in  $O(k \log n)$  time. So the total complexity is  $O(m + u \cdot ((m + n + N)) + k \log n)$ .

---

**Algorithm 15** Incremental Algorithm for Deletion

---

```
1: // Checking  $P_{new}$ -Based Property (specified in Lemma 4.3.3)
2: if  $p_{new} \notin SKY(P_{new})$  then
3:    $Q'_{new} \leftarrow Q'$ 
4: else
5:   // need to calculate  $Q'_{new}$  incrementally
6:   for each  $q \in Q$  which is quasi-dominated by  $p_{new}$  do
7:     re-compute the standalone profit of  $q$ 
8:    $Q'_{new} \leftarrow$  a set of the  $k$  tuples in  $Q$  which have the greatest standalone profits
9: return  $Q'_{new}$ 
```

---

### 4.3.2 Deletion from $P$

Suppose that tuple  $p$  is removed from  $P$ . Similarly, we have the following lemma for a deletion operation and have the corresponding algorithm as shown in Algorithm 15.

**Lemma 4.3.3 ( $P_{new}$ -Based Property)** *If  $p_i \notin SKY(P)$ , then  $Q'_{new} = Q'$ .* □

**Theorem 4.3.2** *Algorithm 15 returns a selection set  $Q'_{new}$  which is equal to the selection set returned by Greedy Algorithm (Version 1).* □

The complexity of checking the  $P_{new}$ -Based Property (Line 2 to Line 3) is  $O(\log(m+n) + N)$ . Otherwise, we need to recompute  $SP(q)$  for each  $q \in Q$  which is quasi-dominated by  $p_i$ . Similarly, this step (line 6 to line 8) takes  $O(u \cdot (\log(m+n) + N) + k \log n)$  time. So, the total complexity is  $O(\log(m+n) + N + u \cdot (\log(m+n) + N) + k \log n)$ .

## 4.4 Finding Top- $k$ Popular Products

In this subsection, we consider finding popular products when *tolerant user preferences* are given, where the popularity of an product is measured by the number of tolerant user preferences on the product. We focus on the problem of *finding top- $k$  popular products*. In the following, Section 4.4.1 introduces the motivation of this

problem. Section 4.4.2 defines the problem formally. Section 4.4.3 elaborates our algorithm to solve this problem.

#### 4.4.1 Motivation

As introduced in Chapter 1, we consider tolerant user preferences here. Suppose a customer would like to choose a package which is not too expensive and the hotel in the package is not too far away from a beach. For example, s/he gives his/her *preference* that the price is at most \$450 and the distance to the beach is at most 4.0km. \$450 and 4.0km are said to be the *greatest possible acceptable values* of attribute price and attribute distance-to-beach, respectively. Since \$450 and 4.0km are the greatest acceptable values for this customer, they are called *tolerant user preferences*. We will define *tolerant user preference* formally in the next subsection.

The tolerant user preferences can be collected by conducting surveys where customers can provide their preferences on products. We can also find tolerant user preferences by extracting customers' preferences from their past histories [51]. Besides, tolerant user preferences can also be obtained directly by some online systems such as "Name Your Own Price" in Priceline.com where customers can provide their preferable prices directly.

Formally, each *tolerant user preference*  $cp$  is represented by a set of  $l$  values, namely  $g_1, g_2, \dots, g_l$ , where  $l$  is the total number of attributes and  $g_j$  is the greatest possible acceptable value of attribute  $A_j$  for  $j \in [1, l]$ . If a customer does not have any special preference on a particular attribute  $A_j$ , s/he can simply specify the greatest possible acceptable value of attribute  $A_j$  to be  $\infty$ . In addition, each tolerant user preference  $cp$  is associated with a *weight*, denoted by  $w(cp)$ , denoting the total number of customers who give this tolerant user preference  $cp$ . Let  $CP$  be a set of tolerant user preferences.

In the following, in order to simplify the discussion, following the spatial database

literature [53, 57, 83, 98, 99], we assume that each potential tuple can satisfy as many tolerant user preferences as possible.

Recall that in Section 4.2.1, we can define various utility function  $F(Q')$  on  $Q'$ . Here, we change the utility function in TPP to another utility function that returns the total number of customers who are interested in some packages in  $Q'$  when tolerant user preferences are available. It makes sense that when new products come out, some companies may care more about the popularity of the new products. Therefore, we have a new problem, called *finding top- $k$  popular products*.

There are a lot of applications of this problem. Generating popular laptops is an example since the components in the market for assembling laptops is abundant and we can assume that a laptop can meet as many tolerant user preferences as possible. Finding popular delivery services in a new cargo company is another example where a delivery service can serve a lot of customers. Finally, finding popular cell phone plans in a new phone company is one example where a plan can be subscribed by a lot of customers. Considering the *capacities* of potential packages (i.e., how many units of potential packages which are available) is left as a future work.

#### 4.4.2 Problem Definition

Given a tuple  $q$  in a final selection set  $Q'$ , we have to set the price of  $q$ . Since we do not want to lose any money, we should set the price of  $q$  at least the cost of  $q$ . Besides, we want to guarantee that  $q$  is in the skyline with respect to  $P \cup Q'$  after we set the price of  $q$ . Given a tuple  $q \in Q$ , we define the set of all possible prices of  $q$ , denoted by  $PS(q)$ , which satisfy the above conditions to be  $\{v | v \geq q.C \text{ and } q \in SKY(P \cup \{q\}) \text{ if we set } q.A_l = v\}$ .

Note that  $Q'$  is the selection set. Suppose that for each tuple  $q \in Q'$ , we set the

price of  $q$  to a positive real number  $v \in PS(q)$ . Given  $q \in Q'$  and a tolerant user preference represented by  $\{g_1, g_2, \dots, g_l\}$ ,  $q$  is said to *satisfy cp* if for each  $j \in [1, l]$ ,  $q.A_j \leq g_j$ .

Given a tuple  $q \in Q$  and a value  $v \in PS(q)$ , the *influence set* of  $q$  with respect to  $v$ , denoted by  $IS(q, v)$ , is defined to be the set of tolerant user preferences which are satisfied by  $q$  if we set the price of  $q$  to  $v$ .

**Definition 4.4.1 (Influence Set and Influence Value)** *Given a tuple  $q \in Q$ , the influence set of  $q$ , denoted by  $IS(q)$ , is defined to be  $\cup_{v \in PS(q)} IS(q, v)$ . Given  $q \in Q$ , the influence value of  $q$ , denoted by  $IV(q)$ , is defined to be  $\sum_{cp \in IS(q)} w(cp)$ .*

*Let  $Q'$  be a subset of  $Q$ . The influence set of  $Q'$ , denoted by  $IS(Q')$ , is defined to be  $\cup_{q \in Q'} IS(q)$ . The influence value of  $Q'$ , denoted by  $IV(Q')$ , is defined to be  $\sum_{cp \in IS(Q')} w(cp)$ .* □

**Problem 3 (Finding Top- $k$  Popular Products)** *Let  $\mathcal{Q}$  be the set of all possible subsets containing  $k$  tuples from  $Q$ . We want to select a set  $Q'$  of  $k$  tuples from  $Q$  such that (1)  $IV(Q') = \max_{Q'' \in \mathcal{Q}} IV(Q'')$  and (2) each tuple in  $Q'$  is in the skyline with respect to  $P \cup Q'$ .* □

The tuples in the output of the above problem are called *top- $k$  popular products*. Note that in the above problem, setting different values of  $k$  gives different influence values of the final selection set. Let  $IV_i$  be the optimal influence value of the final selection set of size  $i$ . It is easy to verify that  $IV_i$  is *monotonically* increasing with  $i$  since more customers are interested in the tuples in the final selection set when more tuples are included in the final selection set. It is also easy to see that when the final selection set contains a certain number of tuples, says  $k_{max}$ , the influence value keeps unchanged even if the selection set contains more tuples. We define the greatest possible influence value denoted by  $IV_{max}$  to be  $\max_{i \in [1, |Q|]} IV_i$ . We also define  $k_{max}$  to be



---

**Algorithm 16** Algorithm for Finding Top- $k$  Popular Products

---

```
1:  $Q' \leftarrow \emptyset$ 
2: while  $|Q'| < k$  do
3:   find  $q \in Q$  such that  $IV(Q' \cup \{q\})$  is the greatest
4:    $X \leftarrow IS(Q' \cup \{q\}) - IS(Q')$ 
5:   set the price of  $q$  to be  $optPrice(q, X)$ 
6:    $Q \leftarrow Q - \{q\}$ 
7:    $Q' \leftarrow Q' \cup \{q\}$ 
8: return  $Q'$ 
```

---

the smallest possible number of tuples in the final selection set such that its influence value is equal to  $IV_{max}$ . That is,  $k_{max} = \min\{i | IV_i = IV_{max}\}$ . In the following, we assume  $k \leq k_{max}$ . If  $k > k_{max}$ , then the additional  $k - k_{max}$  tuples are redundant for influencing customers.

**Theorem 4.4.1** *Problem Finding Top- $k$  Popular Products is NP-hard.* □

### 4.4.3 Our Method

We propose a greedy approach to find top- $k$  popular products. Before we give the algorithm, we define the concept of “optimal price” as follows. Given a tuple  $q \in Q$  and a set  $X \subseteq IS(q)$ , the *optimal price* of  $q$  satisfying tuples in  $X$ , denoted by  $optPrice(q, X)$ , is the greatest possible price  $v \in PS(q)$  we can set such that  $X \subseteq IS(q, v)$ .

The algorithm is shown in Algorithm 16. In line 3 of Algorithm 16, we find  $q \in Q$  such that  $IV(Q' \cup \{q\})$  is the greatest. In some cases, there are ties. That is, there are at least two tuples  $q$  and  $q'$  in  $Q$  which give the same greatest values of  $IV(Q' \cup \{q\}) (= IV(Q' \cup \{q'\}))$ . Let  $Y$  be the set of tuples  $q$  in  $Q$  which give the same greatest values of  $IV(Q' \cup \{q\})$ . In this case, we choose  $q$  in  $Y$  such that  $f(q)$  is the smallest where  $f$  is the function defined in Section 4.2.3.

Note that there are two criteria in Problem 3. The first criterion is to maximize the influence value of a selection set while the second criterion is to make sure that

each tuple in the selection set is in the skyline with respect to  $P \cup Q'$ . Interestingly, although Algorithm 16 finds  $q$  iteratively according to the influence value of a selection set  $Q \cup \{q\}$  but not the criterion on whether  $q$  can be in the skyline with respect to  $P \cup Q'$ , it also returns a set  $Q'$  such that each tuple in  $Q'$  is in the skyline with respect to  $P \cup Q'$ . This result can be found in the following lemma.

**Lemma 4.4.1** *Let  $Q'$  be the selection set returned by the algorithm for finding top- $k$  popular products (i.e., Algorithm 16). Each tuple in  $Q'$  is in the skyline with respect to  $P \cup Q'$ .* □

This algorithm not only can satisfy the second criterion but also can give a theoretical guarantee for the first criterion (even though the problem is NP-hard).

**Theorem 4.4.2** *The algorithm for finding top- $k$  popular products (i.e., Algorithm 16) is 0.63-approximate. Let  $Q'$  be the selection set returned by the algorithm and  $O$  be the optimal set (which gives the greatest influence value). We have  $IV(Q') \geq 0.63 \cdot IV(O)$ .* □

## 4.5 Experimental Settings

We have conducted extensive experiments on a Pentium IV 2.4GHz PC with 4GB memory, on a Linux platform. The generation of synthetic datasets is elaborated in Section 4.5.1, while the collection of real datasets is introduced in Section 4.5.2.

### 4.5.1 Synthetic Datasets

For synthetic datasets, we adapt the dataset generator of [24]. We observe from the real dataset that some attributes have large cardinalities but some have small cardinalities.

For example, in the real dataset, *price* may have thousands of possible values, but *no-of-stops* can have only 2 or 3 possible values. We divide the attributes into two groups of nearly equal size. Note that  $P$  has  $l$  attributes only while  $Q$  has an additional attribute  $C$  in addition to the  $l$  attributes. The first group contains the first half of attributes (or more specifically,  $A_1, \dots, A_{\lfloor l/2 \rfloor}$ ) each of which has the cardinality of 10. The second group contains the second half of attributes (or more specifically,  $A_{\lfloor l/2 \rfloor + 1}, \dots, A_l$ ) and attribute  $C$  where each attribute in this group has the cardinality of 10k.

We generate  $P$  and  $Q$  in the same way except that generating  $P$  involves  $l$  attributes but generating  $Q$  involves the first  $l - 1$  attributes and attribute  $C$ . Note that attribute  $A_l$  of  $Q$  is not considered because in our problem definition, attribute  $A_l$  is to be found. The dataset generation process is described as follows. Firstly, we used the dataset generator provided by [24] to generate an anti-correlated dataset where each attribute value is a real number in a range between 0.0 and 1.0. Secondly, we perform a postprocessing step so that each attribute in the first group has the cardinality of 10 and each attribute in the second group has the cardinality of 10k. For an attribute in the first group, it can be easily done by multiplying a value in this attribute by 10 and rounding it to be an integer. We can also do a similar step for an attribute in the second group.

### 4.5.2 Real Datasets

For the real datasets, same as [90], we obtain real datasets from Priceline.com and Expedia.com. For the website of Priceline.com, we obtained all packages on Jan 15, 2009 for a round trip traveling from San Francisco to New York for a period from March 1, 2009 to March 7, 2009. We have 149 packages. These packages form the set  $P$  of existing tuples. Each package has 6 attributes, namely *quality-of-room*, *customer-hotel-grading*, *hotel-class*, *hotel-price*, *class-of-flight*, *no-of-stops* and *price*.

For the website of Expedia.com, we obtained all flights and all hotels on the same

day (i.e., Jan 15, 2009) for the same round trip with the same travel period. We have 1014 hotels and 4394 flights. According to these hotels and these flights, we adopt the method proposed by [90] to generate all *competitive packages*. Details can be found in [90]. These competitive products form set  $Q$ . In this dataset, we have 4787 competitive packages. Similarly, each package in  $Q$  has 6 attributes (including attribute *price*). Note that each package in  $Q$  is associated with an additional cost attribute. In order to generate the cost attribute, for each package  $q$  in this package set, we set  $q.C$  to be the price of this package multiplied by a discount rate  $d$  where  $d$  is a user parameter. Note that although there are values in attribute *price* in this set  $Q$ , we discard all these values in the dataset because our problem is to find these values.

## 4.6 Results for TPP over Static Datasets

We implemented all algorithms we proposed for problem TPP, namely *DP*, *GR1*, *GR2*. *DP* corresponds to our dynamic programming approach while *GR1* and *GR2* correspond to the first version and the second version of the greedy algorithms for problem TPP. We also implemented a naive (or brute-force) algorithm described in Section 4.2.2. We name it as *BF*. All the program are implemented in C++. In the following, we consider problem TPP with the  $h$ -dominance constraint discussed in Section 4.2.6 since it is more general than problem TPP without the  $h$ -dominance constraint.

We measured the algorithms with four measurements, namely (1) *Execution Time*, (2) *Preprocessing Time*, (3) *Memory Cost* and (4) *Profit*. (1) The execution time of an algorithm corresponds to the time it takes to find the final selection. (2) The preprocessing of an algorithm corresponds to the time it builds a  $R^*$ -tree index for quasi-dominance checking. (3) The memory cost of an algorithm is the memory occupied by the algorithm. (4) The profit of an algorithm corresponds to the profit returned by

the algorithm. The experimental results for TPP over small synthetic datasets, large synthetic datasets and real datasets are shown in Section 4.6.1, Section 4.6.2 and Section 4.6.3, respectively.

#### 4.6.1 Results over Small Synthetic Datasets

It is known that a dynamic programming approach is not scalable to large datasets. Besides, this dynamic programming approach solves problem TPP when  $l = 2$ . So, we conducted some experiments to compare all proposed algorithms over a small two-dimensional synthetic dataset where  $|P| = 10,000$  and  $|Q| = 10,000$ . We set the default parameters as  $h = 5$ ,  $d = 0.5$  and  $\sigma = 200$ .

We vary  $k$  to study the performance of the proposed algorithms. Figures 4.1, 4.2, 4.3 and 4.4 are the results for execution time, profit and the memory cost of each algorithm, respectively.

In Figure 4.1, the execution time of *BF* is very large and is very unscalable. Note the *PREP* is the preprocessing time of *GR1* and *GR2*. It is nearly equal to the time for *GR1* and *GR2* to find the selection set for problem TPP. In Figure 4.2, we compare *PREP+GR1*, *PREP+GR2* and *DP*, in which *DP* runs faster than the former two. In Figure 4.3, the profit of *DP* and *BF* is the greatest but the profit of *GR1* and *GR2* is also high. In most cases, *GR2* returns higher profit than *GR1*. In Figure 4.4, as expected, *DP* occupies much more memory than other approaches.

Since *BF* is not scalable, in the following, we do not compare all algorithms with *BF*.

#### 4.6.2 Results over Large Synthetic Datasets

We conducted experiments over large synthetic datasets to study the scalability of *GR1* and *GR2*. We varied  $|P|$ ,  $|Q|$ ,  $d$ ,  $l$ ,  $k$ ,  $\sigma$  and  $h$  in our experiments. The values of each

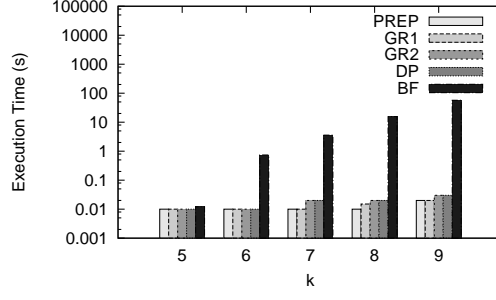


Figure 4.1: Execution time of all algorithms (small dataset)

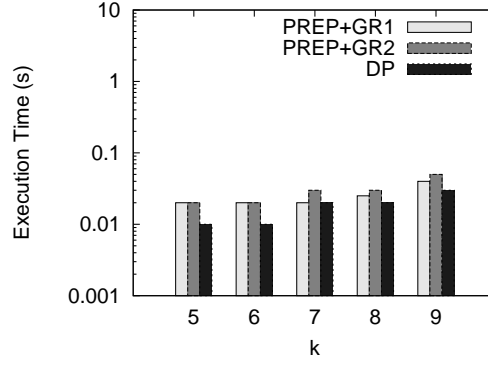


Figure 4.2: Execution time of GR1, GR2 and DP (small dataset)

Parameter	Values
$ P $	0.5M, <b>1M</b> , 1.5M, 2.0M
$ Q $	0.5M, 1M, <b>2M</b> , 3M
$d$	0.25, <b>0.5</b> , 0.75, 1
$l$	2, <b>5</b> , 10, 15, 20
$k$	10, <b>20</b> , 50, 100
$\sigma$	50, <b>100</b> , 150, 200
$h$	0, <b>10</b> , 20, 30

Table 4.5: Experimental settings on large synthetic datasets

parameter used in the experiments are given in Table 4.5 where the default values are in bold.

Figures 4.5, 4.6, 4.7, 4.8 and 4.9 show some selected results.

**Execution time:** Figures (a) show the measurement of execution time. In all figures, *GR2* runs slower than *GR1*. As we discussed in Section 4.2.5, the time complexity of *GR2* is higher than that of *GR1*. For factor  $k$  (Figure 4.7(a)), when  $k$  increases, the execution time of *GR2* increases exponentially but the execution time of *GR1* does not

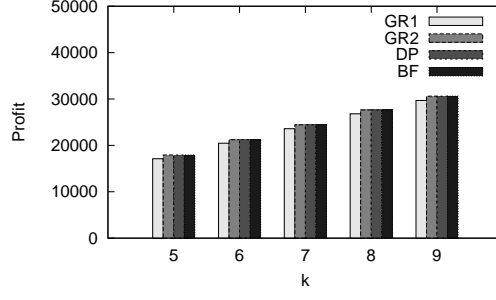


Figure 4.3: Profits of all algorithms (small dataset)

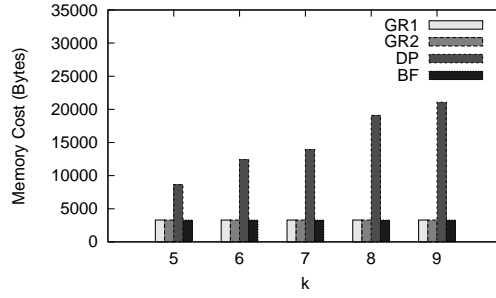


Figure 4.4: Memory costs of all algorithms (small dataset)

change much. This is because the time complexity of *GR2* is quadratic with respect to  $k$  but the time complexity of *GR1* is not.

In Figure 4.8(a), the execution times of *GR1* and *GR2* increase rapidly when  $l$  increases. This is because when  $l$  is larger, the computation cost of finding the quasi-dominance in the algorithm is higher.

**Preprocessing time:** Figures (b) show the preprocessing time of the algorithms. This involves the step of building the index. When  $|P|$ ,  $|Q|$  and  $l$  increase, the preprocessing times of *GR1* and *GR2* increase.

**Memory cost:** Figures (c) show the memory cost of the algorithms. Since the memory cost of both *GR1* and *GR2* is the memory occupied by the spatial index R\*-tree on dataset  $P \cup Q$ , when  $|Q|$  increases and  $|P|$  increases, the memory cost increases, as shown in Figures 4.5.

**Profit:** Figures (d) show the profit returned by the algorithms. In most cases, *GR1* and *GR2* gives similar profits. For factor  $k$  (Figure 4.7(d)), when  $k$  increases, the profits

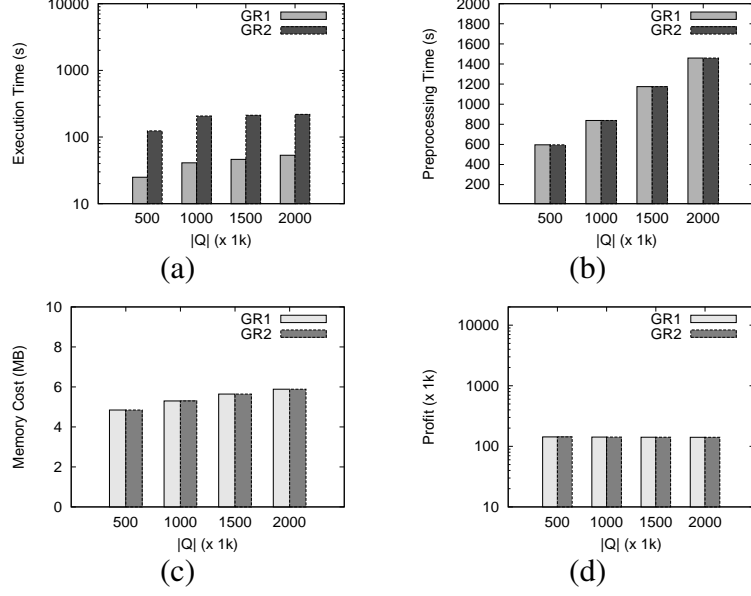


Figure 4.5: Effect of  $|Q|$  (the number of potential new tuples)

Parameter	Values
$h$	15, <b>20</b> , 25, 30
$\sigma$	1000, 2000, <b>5000</b> , 10000
$d$	0.4, <b>0.6</b> , 0.8, 1.0
$k$	100, <b>150</b> , 200, 250

Table 4.6: Experiment settings on real dataset

of  $GR1$  and  $GR2$  increase because more tuples are selected to contribute the profit of the final selection. For factor  $h$  (Figure 4.9(d)), when  $h$  increases, the profits of both algorithms decreases. This is because if  $h$  is larger, then the price of each selected tuple in the final selection should be set lower in order that each of tuples dominates at least  $h$  tuples in the existing market.

### 4.6.3 Results over Real Datasets

We show the experimental results for TPP on real datasets. We varied four factors, namely  $h, k, d$  and  $\sigma$ . But here we only show the results with two factors  $h$  and  $k$  as shown in Figures 4.10 and 4.11, respectively. The default setting configuration is:  $k = 150, h = 20, d = 0.6$  and  $\sigma = 50$ . The results for real datasets are similar to those for synthetic datasets.



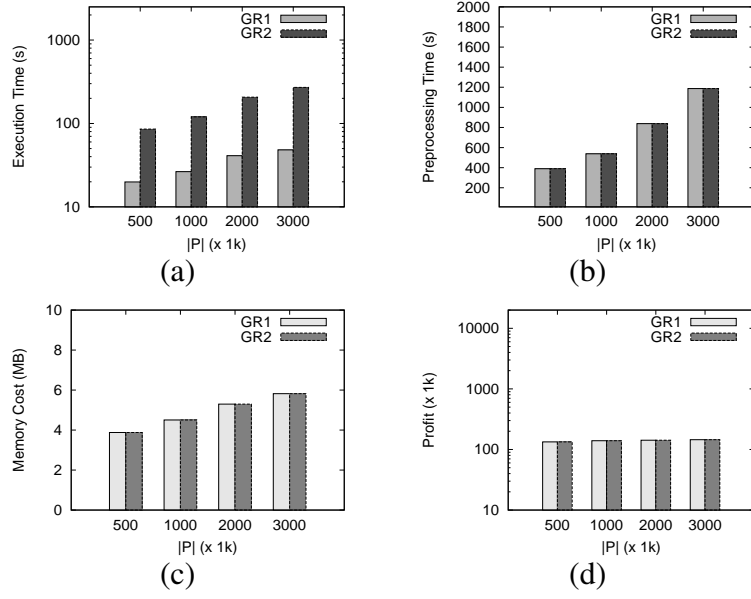


Figure 4.6: Effect of  $|P|$  (the number of tuples in the existing market)

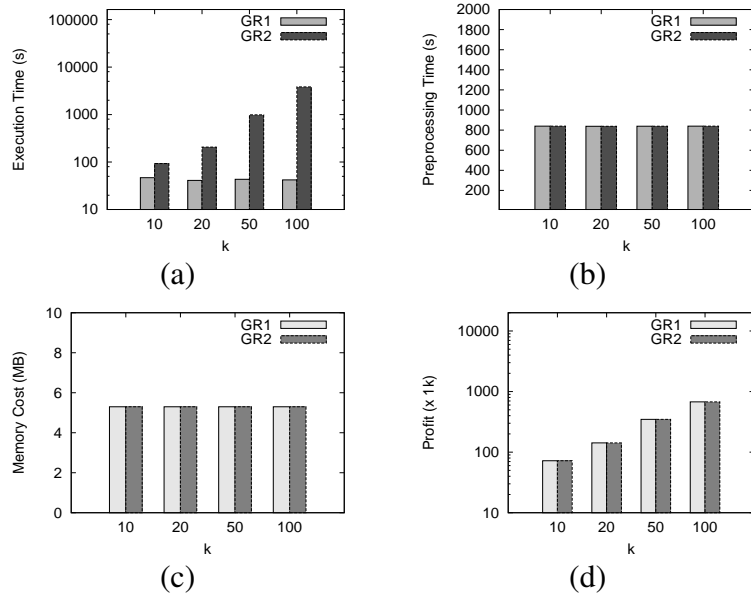


Figure 4.7: Effect of  $k$  (the size of the final selection set)

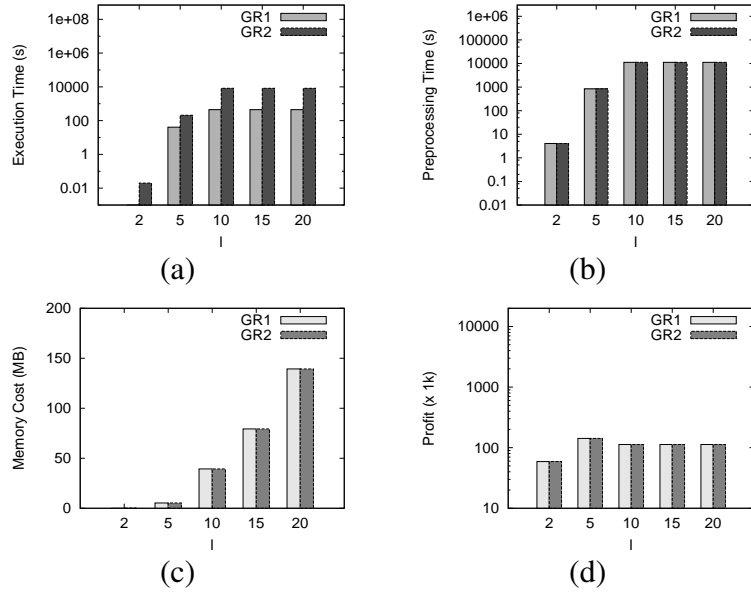


Figure 4.8: Effect of  $l$  (the number of attributes)

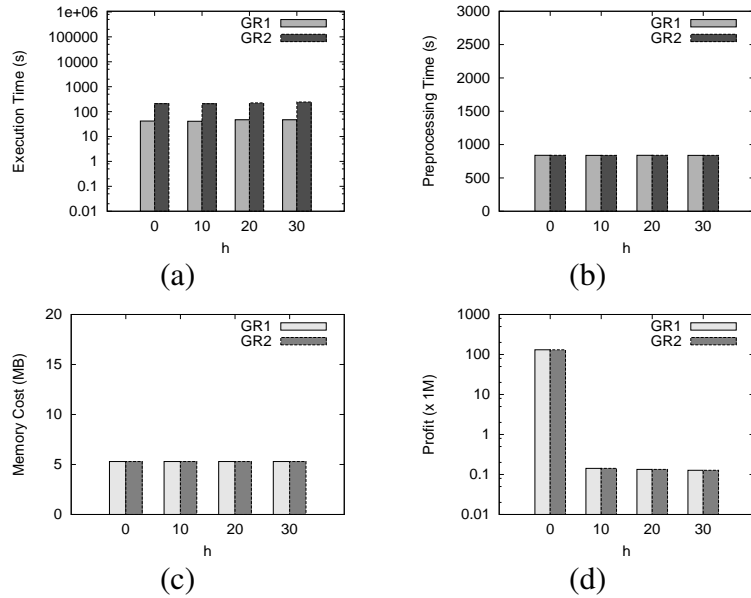


Figure 4.9: Effect of  $h$  (the minimum number of tuples dominated by each tuple in the selection set)

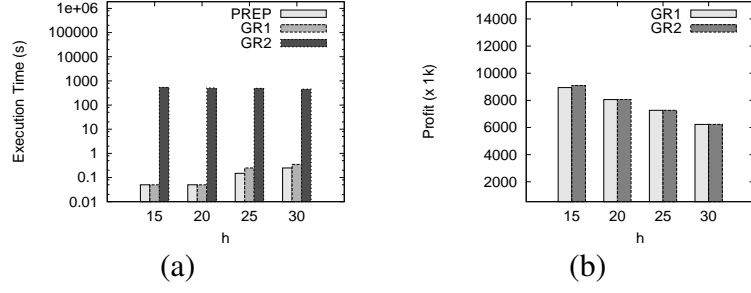


Figure 4.10: Effect of  $h$  (the minimum number of tuples dominated by each tuple in the selection set)

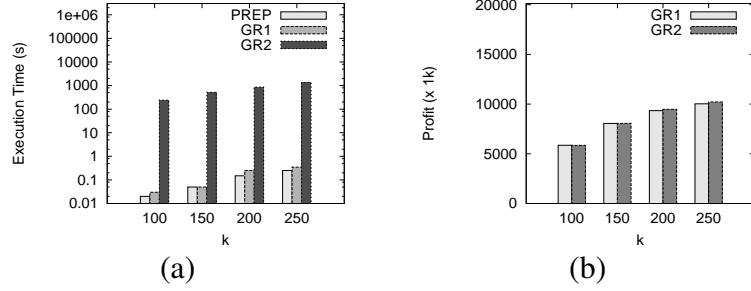


Figure 4.11: Effect of  $k$  (the size of the final selection set)

## 4.7 Results of TPP over Dynamic Data

We show the performance of our proposed incremental algorithm proposed in Section 4.3. We denote this algorithm by *IA*. We compare it with an algorithm which finds top- $k$  profitable products from *scratch*. Since *IA* is similar to *GRI* which finds the products from scratch, in this experiment, we choose *GRI* for comparison. Since *IA* and *GRI* have the same preprocessing time, in the following, we do not show *PREP* in the figure.

We conducted experiments over synthetic datasets and real datasets. The default values for the experiments over these datasets are the same as in Section 4.5.1 and Section 4.5.2.

In this dynamic case, we have three types of operations, namely insertion, deletion and modification. As we discussed before, we focus on the former two operations. In the experiments, we generate operations in this dynamic case as follows. Consider a dataset  $D$ . If the operation to be generated is an insertion, we randomly generate a

tuple by the method of generating tuples in  $P$  and regard it as the tuple to be inserted. If the operation to be generated is a deletion, we randomly pick one of the tuples in  $D$  and regard it as the tuple to be removed. In the experiment, for each dataset, we create a batch  $O$  of operations. We did three types of batches for experiments. The first type is the batch containing all insertion operations, the second type is the batch containing all deletion operations and the third type is the batch containing 50% insertion operations and 50% deletion operations. In our experiments, we varied the size of  $O$  (denoted by  $|O|$ ) from 100k to 400k. We evaluate the algorithms with their execution times. We show the experimental results over the real dataset as shown in Figure 4.12 when the third batch type is considered. The execution times of both algorithms increase with  $|O|$ . Besides,  $IA$  is much more efficient than  $GR1$ .

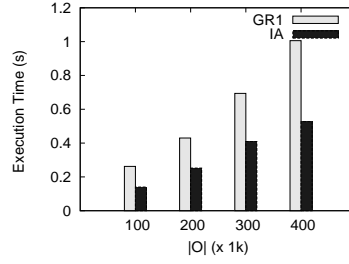


Figure 4.12: Effect of  $|O|$  (the size of update operation sets)

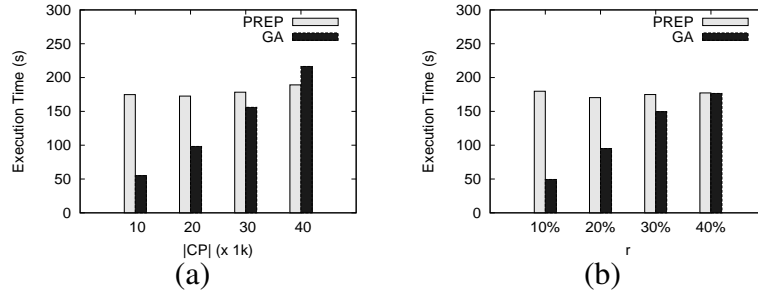


Figure 4.13: Effect of  $|CP|$  and  $r$  on large synthetic datasets

## 4.8 Results for Finding Top- $k$ Popular Products

We study how our proposed algorithm in Section 4.4.3 performs when we want to find top- $k$  popular products. The preprocessing time of this algorithm (which includes

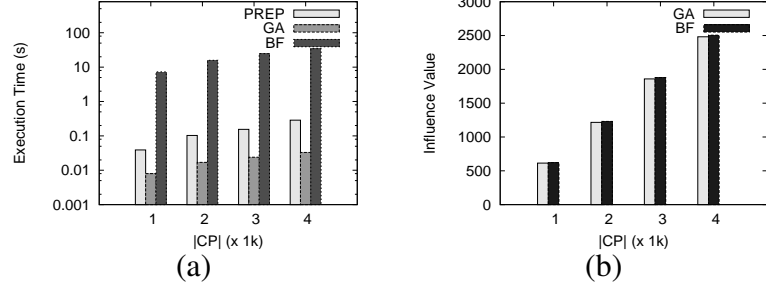


Figure 4.14: Effect of  $|CP|$  on a real dataset

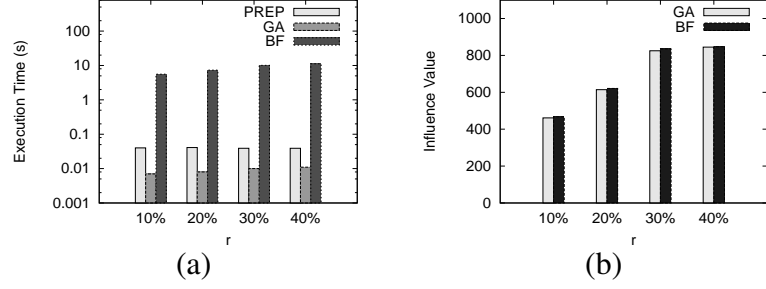


Figure 4.15: Effect of  $r$  on a real dataset

the time to build an R\*-tree) is denoted by *PREP* while the main processing time of this algorithm is denoted by *GA*. We compare it by a brute-force algorithm which enumerates all possible subsets of size  $k$  and finds the subset which gives the greatest influence value. We denote it by *BF*.

As before, we conducted experiments over the real dataset. The default values of parameters except  $k$  and  $h$  are the same as in Section 4.5.2. Besides, since running *BF* is time-consuming, we set  $k = 3$  in our experiments in order that the execution time of *BF* is shorter. Since finding top- $k$  popular products does not have any  $h$ -dominance constraint described in Section 4.2.6,  $h$  is set to 0.

In the problem of finding top- $k$  popular products, tolerant user preferences are generated as follows. Suppose that we want to generate a tolerant user preference  $cp$ . Firstly, we generate a tuple  $c$  by the method of generating a tuple in  $P$ . Secondly, for each attribute  $A_i$  of tuple  $c$  where  $i = 1, 2, \dots, l$ , we set  $g_i$  to  $c_i + r \times C(A_i)$  where  $r$  is a positive real number and a user parameter (set to 0.1 by default), and  $C(A_i)$  is the cardinality of attribute  $A_i$ . We varied the number of tolerant user preferences from 10k (1k) to 40k (4k) where its default value is 10k (1k) in synthetic (real) datasets.

We evaluate the algorithms with two measurements, namely the *execution time* of the algorithms and the *influence value* of a selection set returned by the algorithms.

Figure 4.13 shows the experiments on large synthetic datasets. Since *BF* is not scalable, we do not include it in the figure. In the figures, the execution time of *GA* increases with the size of *CP* and *r*.

Figures 4.14 and 4.15 show the experimental results on a real dataset. In Figure 4.14(a), *GA* runs at least 2 orders of magnitude faster than *BF*. In Figure 4.14(b), the influence values of the selection sets returned by *GA* and *BF* are nearly the same. Similar results can be found in Figure 4.15. All the results are consistent with our theoretical result about the 0.63-approximation.

## 4.9 Conclusions

In this chapter, we identified and tackled two problems, finding top- $k$  profitable products and finding top- $k$  popular products, which have not been studied before. In finding top- $k$  profitable products, we considered generalized user preferences. In finding top- $k$  popular products, we considered both generalized user preferences and tolerant user preferences. For the problem of finding top- $k$  profitable products, we proposed a dynamic programming approach which can find the optimal solution when there are two attributes to be considered. We showed that this problem is NP-hard when there are more than two attributes and two greedy algorithms were proposed. We also presented incremental algorithms for finding top- $k$  profitable products when the dataset  $P$  changes. For the problem of finding top- $k$  popular products, we proved that this problem is NP-hard and proposed a 0.63-approximate algorithm. An extensive performance study using both synthetic and real datasets was reported to verify the effectiveness and efficiency of our algorithms.

## CHAPTER 5

### FINDING COMPETITIVE PRICE

The previous chapter studied how to launch new products from a pool of potential products. This chapter presents how to price new products when *spatial databases* are considered. Section 5.1 introduces the motivation of finding competitive price. Section 5.2 formulates our proposed problems, namely finding simple competitive price and finding  $K$ -dominating competitive price. Section 5.3 proposes a spatial approach. Section 5.4 discusses some extensions of our problem. Section 5.5 evaluates the proposed technique through extensive experiments with both real and synthetic datasets and illustrates the process with a real case study. Section 5.6 concludes this chapter.

#### 5.1 Motivation

Dominance analysis is important in many multi-criteria decision making applications. Recently, dominance analysis [86, 58, 95, 51, 49, 104] has received a lot of interest from both research and applications.

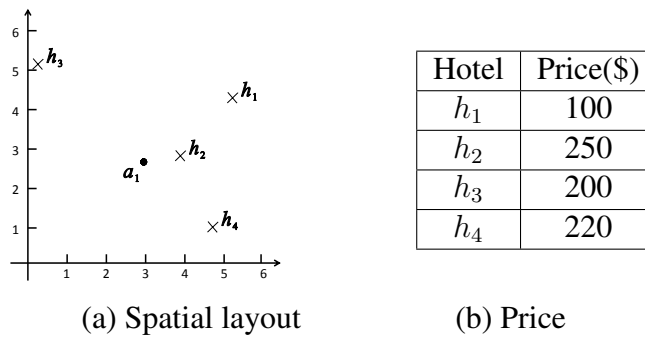
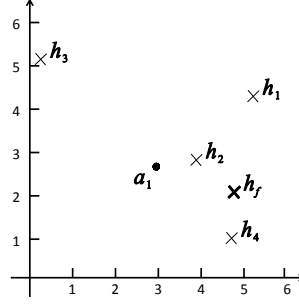


Figure 5.1: A running example

Table 5.1: A decision-making table  $T$ 

Hotel	Distance-to-beach(km)	Price(\$)
$h_1$	3.0	100
$h_2$	1.0	250
$h_3$	4.0	200
$h_4$	2.5	220

Figure 5.2: Hotels in the map with a new hotel  $h_f$ 

**Example 6 (Skyline)** Suppose there are 4 hotels, namely  $h_1, h_2, \dots, h_4$ , which are near to a beach  $a_1$  in Sydney as shown in Figure 5.1. Figure 5.1(a) shows the spatial layout of these 4 hotels and beach  $a_1$ , and Figure 5.1(b) shows the price of each hotel.

Consider that a customer wants to look for a hotel in Sydney using two factors/criteria: distance-to-beach and price. We transform the spatial layout in Figure 5.1(a) and the price of each hotel in Figure 5.1(b) into a new table  $T$  with two attributes, namely distance-to-beach and price, as shown in Table 5.1. This table is called a *decision-making table*. For example, consider hotel  $h_1$ . In Figure 5.1(a), we find the distance between  $h_1$  and  $a_1$ , denoted by  $d(h_1, a_1)$ , equal to 3.0 kilometers (km). Besides, in Figure 5.1(b), the price of  $h_1$  is \$100. Then, we construct a tuple for  $h_1$  in table  $T$  with (distance-to-beach, price) equal to (3.0, 100).

According to  $T$ , we want to determine the best possible choices for the customer. Recall the definition of *skyline* in Chapter 4. It is easy to find out that  $h_1, h_2$ , and  $h_4$  are in the *skyline* set. □

The reason why skyline analysis is popular is that we do not need to know any concrete *user preferences*, like *strict partial order user preferences* or *tolerant user*



preferences specified by users, and can obtain all possible “best” hotels which will be selected by users even with their preference functions. This is because it has been proved in [86, 58, 95, 51, 49, 104] that for any monotonic preference function  $g$  on the factors/criteria, the hotel which has the smallest value of  $g$  is in the skyline. For example, let us denote “Distance-to-beach” by  $X_1$  and “Price” by  $X_2$ . If we set  $g$  to be  $X_1 + X_2$  where the *weight* (or *coefficient*) of  $X_1$  and the *weight* of  $X_2$  are equal, then the hotel with the smallest value of  $g$  is  $h_1$  which is in the skyline. If we set  $g$  to be  $100X_1 + X_2$  where the weight of  $X_1$  is more important than that of  $X_2$ , then the hotel with the smallest value of  $g$  is  $h_2$  which is also in the skyline. If we set  $g$  to be a more complicated function like  $g = \frac{2}{7-2X_1} + \frac{100}{300-X_2}$ <sup>1</sup>, then  $h_4$  (which is in the skyline) is the hotel with the smallest value of  $g$ . So, we call the skyline-related user preferences *generalized user preferences*.

**Example 7 (Application)** Consider that a travel agency wants to open a new hotel  $h_f$  at location indicated in Figure 5.2. The travel agency has to find a suitable price for  $h_f$  called a *competitive price* of  $h_f$  so that  $h_f$  is competitive in the existing market (which includes hotels  $h_1, h_2, \dots, h_4$ ). From Figure 5.2, we find that  $d(h_f, a_1) = 2.0$ . If we set the price of  $h_f$  to \$300, according to the decision-making table  $T$ ,  $h_f$  will be dominated by  $h_2$ . We say that \$300 is not a competitive price of  $h_f$ . However, if we set the price of  $h_f$  to \$230,  $h_f$  will not be dominated by any hotels in the existing market. We say that \$230 is a competitive price of  $h_f$ . □

From the above example, we observe that  $h_f$  may or may not be in the skyline with different prices. In this paper, we are studying to find a competitive price of  $h_f$  such that  $h_f$  is competitive in the existing market. This problem is called *finding simple competitive price*.

---

<sup>1</sup>This function is also monotonic with respect to  $X_1$  and  $X_2$ .

Finding a competitive price of  $h_f$  means that, after we set the price of  $h_f$ ,  $h_f$  is *one* of the best choices for the customer to choose (because there may be more than one hotel in the skyline). In order to make sure that  $h_f$  will be chosen by a customer in the market with a higher probability, we would like to set the price of  $h_f$  such that not only  $h_f$  is in the skyline but also  $h_f$  dominates at least  $K$  existing hotels where  $K$  is an input parameter. This problem is called *finding  $K$ -dominating competitive price*. The above problem makes sense since it is assumed that each hotel in the existing market must be currently chosen by some customers and thus still exists in the market. If this assumption does not hold, it is very likely that the hotels do not exist in the market because no customers choose these hotels. Thus, if  $h_f$  dominates these existing hotels, the customers who originally choose these hotels will choose  $h_f$  finally. We regard the  $K$ -dominating requirement as another kind of generalized user preference.

**Example 8 ( $K$ -dominating Competitive Price)** If we set the price of  $h_f$  to \$230, according to  $T$ ,  $h_f$  does not dominate any hotels. However, if we set it to \$210,  $h_f$  dominates one hotel, namely  $h_4$ . \$210 is a price for problem finding 1-dominating competitive price but \$230 is not. □

Note that our two problems are not limited to one attraction. Instead, we consider multiple attractions. In addition to beach, Opera House and Sydney Aquarium are two other possible attractions in Sydney. In this chapter, we will describe later how we consider multiple attractions.

Setting a competitive price is common in our daily life applications. One example is setting the selling price of an apartment for sale or rental where attractions can be railway stations and shopping malls. Another example is setting a parking fee of a car park where attractions can be shopping malls and museums.

## 5.2 Problem Definition

We have a set  $H$  of  $m$  objects, namely  $h_1, h_2, \dots, h_m$ , in the Euclidean space, each of which represents a *service-site* (e.g., a hotel in Figure 5.2). We also have another set  $A$  of  $n$  objects, namely  $a_1, a_2, \dots, a_n$ , in the same space, each corresponding to an attraction-site (e.g., a beach). For each service-site  $h \in H$ , we use  $h.p$  to denote its *price*<sup>2</sup>. For each  $h \in H$  and  $a \in A$ , the distance between  $h$  and  $a$  is denoted by  $d(h, a)$ .

We consider a general situation where each pair of objects in each Cartesian product  $H \times \{a_j\}$  has a distinct distance for each  $j \in [1, n]$ . That is, for each attraction-site  $a_j$ , any 2 service-sites  $h$  and  $h'$  have distinct distances to  $a_j$  (i.e.,  $d(h, a_j) \neq d(h', a_j)$ ). This assumption allows us to avoid several complicated and uninteresting “boundary cases”. When the assumption is not satisfied, an infinitesimal perturbation to the positions of some service-sites or attraction-sites can always be applied, to break the tie of the distances of two object pairs. Due to the tininess of perturbation, results obtained from the perturbed datasets should be as useful as those from the original datasets.

In order to analyze which service-site  $h$  in  $H$  is better than other service-sites in  $H$ , we define a table called a *decision-making table*  $T$  with  $n + 1$  attributes, namely  $X_1, X_2, \dots, X_{n+1}$ , as follows. For each object  $h \in H$ , we construct a tuple in form of  $(x_1, x_2, \dots, x_{n+1})$  where  $x_j$  is equal to  $d(h, a_j)$  for each  $j \in [1, n]$  and  $x_{n+1}$  is equal to  $h.p$ . We denote each value  $x_j$  by  $h.X_j$  for  $j \in [1, n + 1]$ . Table 5.1 shows an example of the decision-making table  $T$ . In our running example, since there are 4 hotels and one attraction,  $m = 4$  and  $n = 1$ . Thus, there are two attributes in  $T$  where  $X_1$  = “Distance-to-beach” and  $X_2$  = “Price”. In  $T$ , there are 4 correspondence tuples. Let  $\mathcal{X} = \{X_1, X_2, \dots, X_{n+1}\}$ .

Consider two service-sites  $h$  and  $h'$  according to table  $T$ . Following our definition

---

<sup>2</sup>Here, we assume that each service-site is located in the Euclidean space and is associated with *only one* attribute, namely price, for the sake of illustration. In general, each service-site can be associated with more attributes. For example, in our motivating example, hotels can have other attributes like star-rate. In Section 5.4, we will discuss how we extend our problem to this general scenario.

of *Skyline* in Section 4.1, in Table 5.1, it is easy to verify that  $h_1, h_2$  and  $h_4$  are in the skyline. Besides,  $D(h_1) = \{h_3\}, D(h_2) = \emptyset, D(h_3) = \emptyset$  and  $D(h_4) = \emptyset$ .

Consider that a company wants to start a new service-site  $h_f$  and wants to find a competitive price of  $h_f$ , denoted by  $h_f.p$ , such that this new service-site will not be worse than any existing service-site. Suppose that  $h_f.p$  is set to a non-negative value. We say that  $h_f$  meets the *skyline requirement* if  $h_f$  is in the skyline  $SKY(H \cup \{h_f\})$ .

Consider a scenario that the price of each existing service-site is non-zero. A trivial solution is to set the price of  $h_f$  equal to \$0. However, the company wants to earn as much profit as possible. In Example 7, we learn that if we set the price of  $h_f$  too high (e.g., \$300), then  $h_f$  is dominated by other existing service-sites. Apparently, we should choose a suitable value for the price of  $h_f$  which is not too low and too high. The following *monotonicity* property helps to determine a suitable value.

**Property 2** (*Monotonicity for Skyline Requirement*) Consider two possible non-negative real numbers  $p_1$  and  $p_2$  where  $p_1 \geq p_2$ . If  $h_f$  meets the skyline requirement when  $h_f.p$  is set to  $p_1$ , then  $h_f$  meets the skyline requirement when  $h_f.p$  is set to  $p_2$ .

Let  $p_s$  be a non-negative real number such that  $h_f$  meets the *skyline requirement* when  $h_f.p$  is set to  $p_s$ . This monotonicity property suggests that  $h_f$  meets the *skyline requirement* when  $h_f.p$  is set to *any* value at most  $p_s$ . This means that  $h_f.p$  can be set to *many* possible values such that it satisfies the skyline requirement. In this chapter, we are studying to return a *price range* of  $h_f$  (instead of a particular price value) such that  $h_f$  satisfies the skyline requirement. Let  $p_{max,s}$  be the *maximum* possible price for the *skyline requirement*. Formally, we define a *price range*  $\mathbf{R}_s$  of  $h_f$  in form of “ $0 \leq h_f.p < p_{max,s}$ ” such that after we set  $h_f.p$  to be any possible value in this range  $\mathbf{R}_s$ ,  $h_f$  satisfies the skyline requirement. Note that, in order to avoid discussing the complicated and uninteresting boundary case, we assume that  $h_f.p \neq p_{max,s}$  in our problem setting.

**Problem 4 (Finding Simple Competitive Price)** Given a new service-site  $h_f$ , we want to find a price range  $\mathbf{R}_s$  of  $h_f$  in form of “ $0 \leq h_f.p < p_{max,s}$ ” where  $p_{max,s}$  is a non-negative real number such that (1)  $p_{max,s}$  is maximized and (2)  $h_f$  is in the skyline  $SKY(H \cup \{h_f\})$  if we set  $h_f.p$  to be any possible value in  $\mathbf{R}_s$ .

After we set the price of  $h_f$  to a value within  $\mathbf{R}_s$ ,  $h_f$  is one of the best choices for the customer to choose (because there may be more than one service-site in the skyline). In order that  $h_f$  becomes more competitive among all the service-sites in  $SKY(H \cup \{h_f\})$ , we want that  $h_f$  can attract more customers who originally chose other service-sites. This motivates us to propose another problem in which not only  $h_f$  is not worse than any existing service-site but also  $h_f$  dominates at least  $K$  existing service-sites. Intuitively, if  $K$  is larger, then  $h_f$  dominates more service-sites. Consequently, more customers will choose  $h_f$ .

We say that  $h_f$  meets the  $K$ -dominating requirement if  $h_f$  dominates at least  $K$  existing service-sites (i.e.,  $|D(h_f)| \geq K$ ). Similarly, let  $p_{max,d}$  be the maximum possible price for the  $K$ -dominating requirement. We define the price range  $\mathbf{R}_d$  of  $h_f$  in form of “ $0 \leq h_f.p < p_{max,d}$ ” such that after we set  $h_f.p$  to be any possible value in this range  $\mathbf{R}_d$ ,  $h_f$  satisfies the  $K$ -dominating requirement.

A service-site  $h_f$  is said to meet requirement  $\mathcal{R}$  if  $h_f$  satisfies both the skyline requirement and the  $K$ -dominating requirement. Note that  $p_{max,s}$  is the maximum possible price for the skyline requirement and  $p_{max,d}$  is the maximum possible price for the  $K$ -dominating requirement. Thus, it is easy to verify that the maximum possible price for requirement  $\mathcal{R}$ , denoted by  $p_{max}$ , is equal to  $\min\{p_{max,s}, p_{max,d}\}$ . It can be seen that the requirement  $\mathcal{R}$  can be regarded as an advanced *generalized user preference*, since it is a mix of two kinds of *generalized user preferences*. Intuitively,  $p_{max,s}$  should be no more than  $p_{max,d}$ . But in the experiments, it does not always hold. When  $K$  is relatively small, it is possible that there are several hotels in a partial order

relationship, so that  $p_{max,s}$  is higher than  $p_{max,d}$ .

Similarly, requirement  $\mathcal{R}$  satisfies the monotonicity property.

**Property 3** (*Monotonicity for  $\mathcal{R}$* ) Consider two possible non-negative real numbers  $p_1$  and  $p_2$  where  $p_1 \geq p_2$ . If  $h_f$  meets requirement  $\mathcal{R}$  when  $h_f.p$  is set to  $p_1$ , then  $h_f$  meets requirement  $\mathcal{R}$  when  $h_f.p$  is set to  $p_2$ .

**Problem 5** (*Finding  $K$ -dominating Competitive Price*) Given a non-negative integer  $K$  and a new service-site  $h_f$ , we want to find a price range  $\mathbf{R}$  of  $h_f$  in form of “ $0 \leq h_f.p < p_{max}$ ” where  $p_{max}$  is a real number such that (1)  $p_{max}$  is maximized, (2)  $h_f$  is in the skyline  $SKY(H \cup \{h_f\})$  and (3)  $|D(h_f)| \geq K$  if we set  $h_f.p$  to be any possible value in  $\mathbf{R}$ .

In the above problem formulation, Condition (2) and Condition (3) correspond to the skyline requirement and the  $K$ -dominating requirement, respectively. In the above problem, we want to maximize  $p_{max}$ . In the following, when we say the *optimal* solution, we refer to this maximized value.

Some alternative problem formulations are using some existing prediction models such as regression and decision tree models to *estimate* the price of  $h_f$  based on the existing service-sites in  $H$ , which are good if we are given some *tolerant user preferences* or *strict partial order user preferences* specified by customers. However, in our case that we do not know any specified user preferences, the estimated price obtained by these formulations may not meet the two requirements specified in Problem 5.

Note that problem Finding  $K$ -dominating Competitive Price is more general than problem Finding Simple Competitive Price. This is because when  $K$  is equal to 0, problem Finding  $K$ -dominating Competitive Price becomes problem Finding Simple Competitive Price. In the following, we focus on solving problem Finding  $K$ -dominating Competitive Price.

A naive approach to this problem is described as follows. For each possible non-negative real number  $v$ , it tries to set the price of  $h_f$  to be  $v$  and test whether  $h_f$  satisfies the skyline requirement and the  $K$ -dominating requirement. If yes,  $v$  is a possible solution for  $p_{max}$ . Finally, it selects the greatest value such that  $h_f$  satisfies the requirements. However, since there are a large (or an infinite) number of possible values, this approach is infeasible. In the following, we propose an approach which avoids testing the requirements with a large number of possible values.

### 5.3 Spatial Approach

In this section, we propose a spatial approach which makes use of some spatial properties and runs efficiently in large datasets.

Problem Finding  $K$ -Dominating Competitive Price has two requirements, namely the skyline requirement and the  $K$ -dominating requirement. We propose a spatial approach which meets the above two requirements. Specifically, it involves the following three major phases.

- **Phase 1 (for Skyline Requirement):** We find the maximum possible price for the skyline requirement, denoted by  $p_{max,s}$ .
- **Phase 2 (for  $K$ -Dominating Requirement):** We find the maximum possible price for the  $K$ -dominating requirement, denoted by  $p_{max,d}$ .
- **Phase 3 (for Requirement  $\mathcal{R}$ ):** We compute the maximum possible price for requirement  $\mathcal{R}$  (which combines the above two requirements), denoted by  $p_{max}$ , to be  $\min\{p_{max,s}, p_{max,d}\}$ .

In the following, we describe how we make use of some spatial properties to perform the above three phases efficiently. Section 5.3.1 first describes some notations.

Then, Section 5.3.2 describes the spatial properties used in our spatial approach. Then, Section 5.3.3 proposes a spatial method for problem Finding  $K$ -Dominating Competitive Price.

### 5.3.1 Notations

Given an attraction-site  $a_j$  and a new service-site  $h_f$ , we define the *critical region* for attraction-site  $a_j$ , denoted by  $R_j$ , to be the region occupied by the circle centered at  $a_j$  with radius equal to  $d(h_f, a_j)$  (with the boundary included). For example, consider our running example. In Figure 5.3(a) which has the same objects as Figure 5.2, the shaded region is  $R_1$ .

The critical region for attraction-site  $a_j$  is used to efficiently determine whether a service-site  $h_i \in H$  is nearer to attraction-site  $a_j$  compared with  $h_f$ . Specifically, if a service-site  $h_i$  is inside  $R_j$ , then we know that  $h_i$  is nearer to attraction-site  $a_j$  compared with  $h_f$ . Otherwise, we know that  $h_i$  is farther from  $a_j$ . For example, in Figure 5.3(a), since  $h_2$  is in  $R_1$ ,  $h_2$  is nearer to attraction-site  $a_1$  compared with  $h_f$ . On the other hand,  $h_3$  is farther from  $a_1$  since  $h_3$  is outside  $R_1$ .

We also define  $\bigcap_{j=1}^n R_j$  ( $\bigcup_{j=1}^n R_j$ ) to be the intersection (union) among all regions represented by  $R_1, R_2, \dots, R_n$ . Let  $\mathcal{I} = \bigcap_{j=1}^n R_j$  and  $\mathcal{U} = \bigcup_{j=1}^n R_j$ . For example, Figure 5.3(b) shows the same objects as Figure 5.3(a) with two additional attraction-sites, namely  $a_2$  and  $a_3$ . In Figure 5.3(b), the shaded region corresponds to  $\mathcal{I}$ . In Figure 5.3(c) showing the same objects as Figure 5.3(b), the shaded region corresponds to  $\mathcal{U}$ .

Given a set  $A$  of  $n$  objects in a Euclidean space, the *convex hull* of  $A$  [41] is a minimal set of objects in  $A$  such that these objects form a convex polygon and all objects in  $A$  are inside the region occupied by this convex polygon. Let the *region* occupied by the convex polygon for the convex hull of  $A$  be  $CH(A)$ . For example,



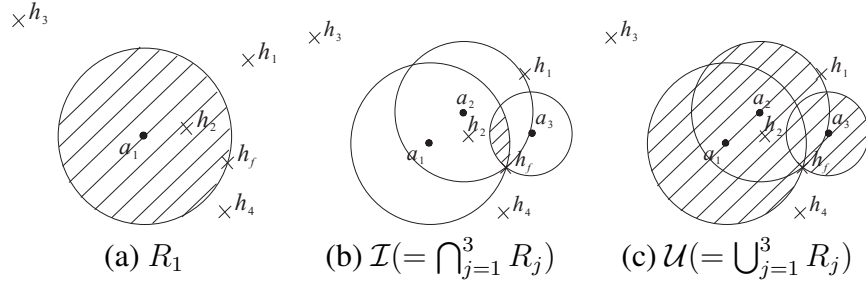


Figure 5.3: Region

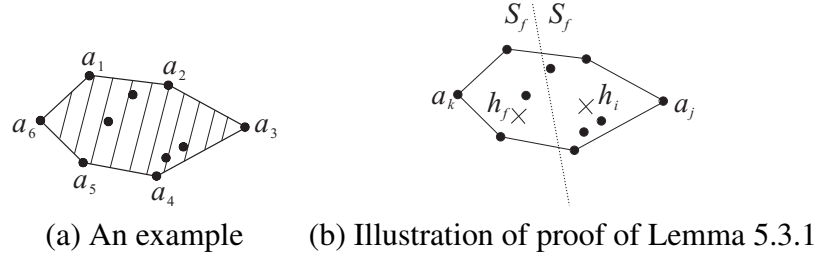


Figure 5.4: Convex Hull

Figure 5.4(a) shows some objects represented by black dots where  $A$  is equal to a set of all black dots. The convex hull of  $A$  is  $\{a_1, a_2, a_3, a_4, a_5, a_6\}$ . The convex polygon in the figure corresponds to the polygon for the convex hull of those objects. The shaded region in the figure corresponds to  $CH(A)$  (i.e., the region occupied by the polygon).

### 5.3.2 Properties

In this subsection, we give some *spatial properties* for problem Finding  $K$ -Dominating Competitive Price which can be used to speed up the computation.

This problem has two requirements. Consider the first requirement, the skyline requirement. In order to determine  $p_{max,s}$  for the skyline requirement, we have the following two lemmas.

**Lemma 5.3.1** Suppose  $h_f$  is inside  $CH(A)$ .  $h_f$  is in the skyline  $SKY(H \cup \{h_f\})$  no matter what value  $p_{max,s}$  is. □

**Lemma 5.3.2** Suppose  $h_f$  is not inside  $CH(A)$ . If we set  $p_{max,s}$  to be  $\min_{h \in \mathcal{I}} h.p$ , then  $h_f$  is in the skyline  $SKY(H \cup \{h_f\})$ . □

From Lemma 5.3.1 and Lemma 5.3.2, we learn that, if  $h_f$  is inside region  $CH(A)$ ,  $p_{max,s}$  can be set to any value and thus  $h_f$  is in the skyline  $SKY(H \cup \{h_f\})$ . Note that we do not need to scan any service-sites in this case. If  $h_f$  is not inside region  $CH(A)$ , then  $p_{max,s}$  is set to be  $\min_{h \in \mathcal{I}} h.p$ . In this case, we only need to scan the service-sites in region  $\mathcal{I}$ .

Consider the second requirement, the  $K$ -dominating requirement. In order to determine  $p_{max,d}$  for the  $K$ -dominating requirement, we have the following lemma.

**Lemma 5.3.3** *Suppose that there are at least  $K$  service-sites not in  $\mathcal{U}$ . If we set  $p_{max,d}$  to be the  $K$ -th greatest price (i.e.,  $h.p$ ) among all service-sites  $h$  not in  $\mathcal{U}$ , then  $h_f$  dominates at least  $K$  service-sites.*  $\square$

With the above lemmas, in order to meet the  $K$ -dominating requirement, we have to set  $p_{max,d}$  to be the  $K$ -th greatest price among all service-sites not in  $\mathcal{U}$ . There are two issues related to the above lemma. The first issue is how we set  $p_{max,d}$  when there are less than  $K$  service-sites not in  $\mathcal{U}$ . In this case, we report to the user that  $h_f$  cannot dominate at least  $K$  service-sites with the current value of  $K$  and suggest the user should provide a smaller value of  $K$ . The second issue is how we set  $p_{max,d}$  when  $K$  is set to 0. In this case, we set  $p_{max,d}$  to  $\infty$ .

In order to satisfy requirement  $\mathcal{R}$  (which combines the above two requirements), the final price is set to  $p_{max} = \min\{p_{max,s}, p_{max,d}\}$ .

### 5.3.3 Algorithm

Algorithm 17 shows the algorithm for finding  $K$ -dominating competitive price. With the lemmas in Section 5.3.2, it is easy to verify the following theorem.

**Theorem 5.3.1 (Correctness)** *Algorithm 17 returns the optimal solution of problem Finding  $K$ -Dominated Competitive Price.*  $\square$

---

**Algorithm 17** Algorithm for Finding  $K$ -Dominating Competitive Price

---

```
1: find  $\mathcal{I}$  and  $\mathcal{U}$ 
2: // Phase 1: To meet the skyline requirement, we find a price  $p_{max,s}$ 
3: find  $CH(A)$ 
4: if  $h_f$  is inside  $CH(A)$  then
5:    $p_{max,s} \leftarrow \infty$ 
6: else
7:    $p_{max,s} \leftarrow \min_{h \text{ is inside } \mathcal{I}} h.p$ 
8: // Phase 2: To meet the  $K$ -dominating requirement, we find a price  $p_{max,d}$ 
9:  $p_{max,d} \leftarrow$  the  $K$ -th greatest price among all service-sites not in  $\mathcal{U}$ 
10: // Phase 3: Finding  $K$ -Dominating Competitive Price  $p_{max}$ 
11:  $p_{max} \leftarrow \min\{p_{max,s}, p_{max,d}\}$ 
12: return  $p_{max}$ 
```

---

Some readers may notice that  $p_{max,s}$  can be equal to  $\infty$  (which can be found in line 5 of Algorithm 17) when  $h_f$  is inside  $CH(A)$ . However, if  $K$  is greater than 0, then it is easy to verify that  $p_{max,d}$  is equal to a value at most the price of one of the service-sites in  $H$  and thus is not equal to  $\infty$ . Finally, the competitive price  $p_{max}$  (which is equal to  $\min\{p_{max,s}, p_{max,d}\}$ ) is not equal to  $\infty$ . If  $K$  is equal to 0, then we set  $p_{max,d}$  to  $\infty$ . In this case, it is possible that  $p_{max}$  is  $\infty$  (because  $p_{max,s}$  can be equal to  $\infty$  and  $p_{max} = \min\{p_{max,s}, p_{max,d}\}$ ). Since it is not reasonable to return  $\infty$  as the answer for  $p_{max}$  in real-life applications, in our implementation, we set  $p_f$  to be the price of the nearest service-site of  $h_f$ .

We know that the lemmas in Section 5.3.2 can help us to find the competitive price efficiently since we do not need to scan all service-sites in  $H$ . In the next subsection, we will introduce some *indexing techniques* (e.g., R\*-tree) to further speed up the computation. Moreover, we will give a theoretical time complexity analysis.

### 5.3.4 Detailed Steps and Theoretical Analysis

In Algorithm 17, we need to determine two regions  $\mathcal{I}$  and  $\mathcal{U}$ . Besides, we also need to find the service-sites in  $\mathcal{I}$  and find the service-sites not in  $\mathcal{U}$ . In the following, we describe how we achieve the above steps *efficiently* by using some *spatial index techniques*.

Now, we give the detailed steps of Algorithm 17 and analyze its complexity. There are five major steps in the algorithm.

- *Step 1 (Finding  $\mathcal{I}$  and  $\mathcal{U}$ ):* Firstly, for each  $a_j \in A$ , we construct  $R_j$  which is a circle centered at  $a_j$  with radius  $d(h_f, a_j)$ . Since the circle construction takes  $O(1)$  time and there are  $n$  attraction-sites in  $A$ , this step takes  $O(n)$  time. Secondly, we construct  $\mathcal{I}(\mathcal{U})$  by performing an intersection (union) operation over all  $R_j$ 's. In our implementation, we conceptually represent  $\mathcal{I}(\mathcal{U})$  by storing a list  $L$  of  $R_j$ 's. Note that  $L$  contains  $n$  elements for  $R_j$ . This step takes  $O(n)$  time. Thus, Step 1 takes  $O(n)$  time.
- *Step 2 (Finding Convex Hull):* Step 2 finds the convex hull over set  $A$ . Let  $\alpha(N)$  be the running time to find the convex hull over a set of size  $N$ . Step 2 takes  $O(\alpha(n))$  time. We adopt the algorithm from [41] to find the convex hull where the running time of this algorithm is  $O(N \log N)$  time. Thus, Step 2 requires  $O(n \log n)$  time.
- *Step 3 (Checking whether  $h_f$  is inside  $CH(A)$ ):* It is easy to verify that checking whether  $h_f$  is inside  $CH(A)$  requires  $O(n)$  time. If  $h_f$  is inside  $CH(A)$ , then we assign  $p_{max,s}$  with  $\infty$ , which takes  $O(1)$  time. Otherwise, we do the following step. We find all service-sites in region  $\mathcal{I}$ . This step can be done by performing range queries over set  $H$ . Specifically, for each  $R_j$  in  $L$  (representing  $\mathcal{I}$ ), we perform a range query over set  $H$  with a circle centered at  $a_j$  with radius  $d(h_f, a_j)$ . Let  $\beta(N)$  be the running time of a range query over the dataset of size  $N$ . Since  $L$  contains  $n$  elements and each range query takes  $O(\beta(m))$  time, the running time of this sub-step is  $O(n \cdot \beta(m))$ . Then, we perform intersection operations among all results obtained from the above range queries. With the bitwise implementation, an intersection operation with two sets can be done in  $O(1)$  time.

Since there are  $O(n)$  intersection operations, this step takes  $O(n)$  time. Among all service-sites in region  $\mathcal{I}$ , we find the smallest price. The overall running time for Step 3 is  $O(n + n \cdot \beta(m) + n) = O(n \cdot \beta(m))$ . If  $\mathcal{I}$  is empty or there is no service-site in  $\mathcal{I}$ , then we assign  $p_{max,s}$  with  $\infty$  immediately.

In the literature,  $\beta(m)$  is theoretically bounded [29]. Let  $J$  be the greatest result size of a range query (i.e., the greatest number of service-sites in the range). In our problem setting, since a range query can be executed in  $O(J + \log m)$  time [29], Step 3 takes  $O(n(J + \log m))$  time.

In our implementation, we adopt an R\*-tree [19] to support range queries. It has been shown that the R\*-tree performs efficiently in real cases and is commonly adopted for range queries although it does not have good worst-case asymptotic performance. Specifically, we build an R\*-tree over all service-sites in  $H$  and then perform a range query over this R\*-tree.

- *Step 4 (Finding the  $K$ -th Greatest Price Among all Service-sites not in  $\mathcal{U}$ ):* Firstly, we find a set  $R$  of all service-sites in  $\mathcal{U}$ . Similar to Step 3, this step can be done in  $O(n \cdot \beta(m))$  time. Secondly, we find a set  $S$  of all service-sites not in  $\mathcal{U}$  by  $H - R$ , which takes  $O(1)$  with the bitwise implementation. Thirdly, we sort all service-sites in  $S$  in descending order of their prices, which takes  $O(m \log m)$  time. Fourthly, we find the service-site  $h$  with the  $K$ -th greatest price, which takes  $O(1)$  time. This value corresponds to  $p_{max,d}$ . The running time of Step 4 is  $O(n \cdot \beta(m) + 1 + m \log m + 1) = O(n \cdot \beta(m) + m \log m)$  time.

With the method used in [29], Step 4 can be done in  $O(n(J + \log m) + m \log m)$  time.

- *Step 5 (Finding the minimum value from  $\{p_{max,s}, p_{max,d}\}$ ):* We find  $p_{max}$  with  $\min\{p_{max,s}, p_{max,d}\}$ , which takes  $O(1)$  time.

Thus, the running time of Algorithm 17 is  $O(n + \alpha(n) + n \cdot \beta(m) + n \cdot \beta(m) + m \log m + 1) = O(n + \alpha(n) + n \cdot \beta(m) + m \log m)$  time.

**Theorem 5.3.2** *The complexity of Algorithm 17 is  $O(n + \alpha(n) + n \cdot \beta(m) + m \log m)$ .*

□

*Enhancement:*

In Algorithm 17, there are three phases. The first two phases involve the searching process of finding  $p_{max,s}$  and  $p_{max,d}$ . In order to find  $p_{max,s}$ , we need to find all service-sites in  $\mathcal{I}$ . Finding all service-sites in  $\mathcal{I}$  without any index is time-consuming. Similarly, in order to find  $p_{max,d}$ , we need to find all service-sites not in  $\mathcal{U}$  (or all service-sites in  $\mathcal{U}$ ). It is also time-consuming to do this operation without any index. In this enhancement, we adopt an indexing technique called an *aggregate R\*-tree* [89] to speed up these two operations.

First of all, we present the indexing structure of an *aggregate R\*-tree*. Then, we describe how we use this tree to speed up the two operations.

We adopt an aggregate R\*-tree [89] which is built on the set of all service-sites. In this R\*-tree, each leaf node corresponds to a service-site and each non-leaf node corresponds to a *rectangular region* (usually called a *minimum bounding rectangle (MBR)*) that all its descendants lie inside it. Each leaf node is associated with the price of the corresponding service-site while each non-leaf node is associated with two *aggregate* values of the prices of the service-sites which appear in all of its descendant nodes, namely *minp* and *maxp*. Given a non-leaf node  $N$ , the aggregate value *minp* of  $N$ , denoted by  $N.minp$ , is defined to be the smallest price of the service-sites which appear in all descendant nodes of node  $N$ . That is,  $N.minp$  is defined to be  $\min\{h_i.p | h_i \text{ is inside } N\}$ . Similarly, given a non-leaf node  $N$ , the aggregate value *maxp* of  $N$ , denoted by  $N.maxp$ , is defined to be the greatest

price of the service-sites which appear in all descendant nodes of node  $N$ . That is,  $N.maxp = \max\{h_i.p | h_i \text{ is inside } N\}$ . These two aggregate values of each non-leaf node can be computed directly when the R\*-tree is built. Details can be found in [89].

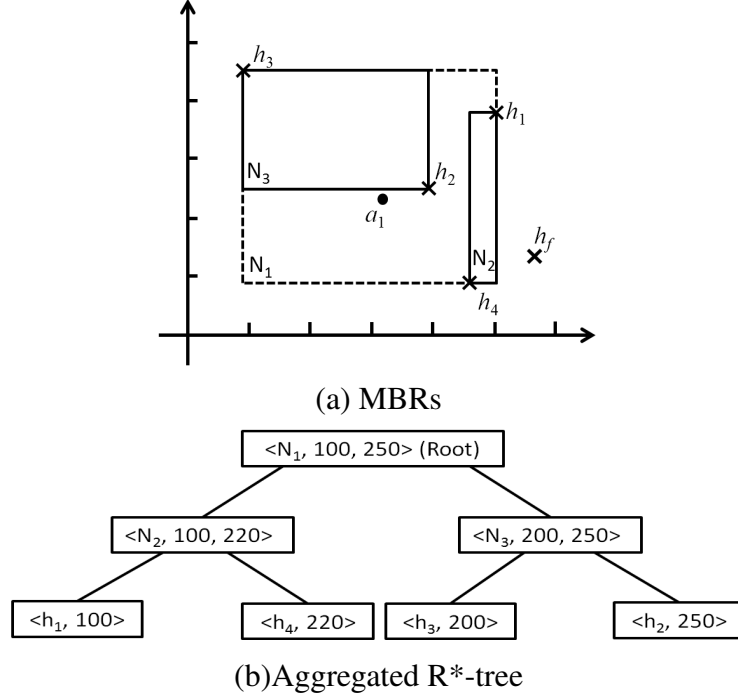


Figure 5.5: R\*-tree and the MBRs

**Example 9 (Aggregate R\*-tree)** Consider our running example as shown in Figure 5.1. According to 4 hotels, we build an aggregate R\*-tree as shown in Figure 5.5. Figure 5.5(a) shows all MBRs in the tree. For example, node  $N_3$  contains two hotels, namely  $h_2$  and  $h_3$ , while node  $N_2$  contains two hotels, namely  $h_1$  and  $h_4$ . Besides, a higher-level node  $N_1$  contains two nodes, namely  $N_2$  and  $N_3$ .

Figure 5.5(b) shows the structure of the R\*-tree. Each leaf node corresponds to a hotel and is associated with its price. For example, the leftmost leaf node corresponds to hotel  $h_1$  and is associated with its price equal to 100. Each non-leaf node  $N$  is associated with two aggregate values, namely  $minp$  and  $maxp$ . We represent this node in form of  $(N, minp, maxp)$  in the figure. For instance, consider node  $N_2$  in form of  $(N_2, 100, 220)$ , which means that the  $minp$  value of  $N_2$  is equal to 100 (which is the smallest price value among all of its descendant nodes) and the  $maxp$  value

of  $N_2$  is equal to 220 (which is the greatest price value among all of its descendant nodes).  $\square$

After we present the indexing structure of the R\*-tree, we are ready to describe how we speed up the two operations related to  $p_{max,s}$  and  $p_{max,d}$  by using the R\*-tree. Before we give the mechanism on how to speed up the operations, we first introduce two variables, namely  $p'_{max,s}$  and  $p'_{max,d}$ . Variable  $p'_{max,s}$  is the variable storing the value of  $p_{max,s}$  which is found to be the best during the execution of the algorithm using the R\*-tree. Note that we want to find the *smallest* price value among all service-sites in  $\mathcal{I}$ . Initially,  $p'_{max,s}$  is set to  $\infty$ . During the execution of the algorithm, when we find a service-site  $h$  in  $\mathcal{I}$ , we update  $p'_{max,s}$  with  $h.p$  if  $h.p$  is smaller than  $p'_{max,s}$ . Similarly, variable  $p'_{max,d}$  is the variable storing the value of  $p_{max,d}$  which is found to be the best during the execution. Note that we want to find the  $K$ -th *greatest* price value among all the service-sites not in  $\mathcal{U}$ . Initially,  $p'_{max,d}$  is set to 0. When the algorithm is executed, when we find a service-site  $h$  not in  $\mathcal{U}$ , we update  $p'_{max,d}$  with  $h.p$  if  $h.p$  is larger than  $p'_{max,d}$ .

Now, we describe how we make use of these two variables for speeding up the two operations. We first present two lemmas related to this speedup. The first lemma is related to  $p'_{max,s}$  while the second lemma is related to  $p'_{max,d}$ .

**Lemma 5.3.4** (Pruning Rule in Computing  $p'_{max,s}$ ) *If node  $N$  in the aggregate R\*-tree satisfies  $p'_{max,s} \leq N.minp$ , then all the descendants of  $N$  can be pruned.*  $\square$

**Lemma 5.3.5** (Pruning Rule in Computing  $p'_{max,d}$ ) *If node  $N$  in the aggregate R\*-tree satisfies  $p'_{max,d} \geq N.maxp$ , then all the descendants of  $N$  can be pruned.*  $\square$

Lemma 5.3.4 is used to speed up the computation of  $p_{max,s}$  while Lemma 5.3.5 is used to speed up the computation of  $p_{max,d}$ .



In the algorithm to be explained later, we will describe that we process a certain number of nodes in the  $R^*$ -tree one by one in order to compute  $p_{max,s}$  and  $p_{max,d}$ . Now, we first assume that the nodes in the tree are processed in a pre-defined ordering.

Consider that we want to compute  $p_{max,s}$ . Suppose that we process a node  $N$  during the execution of the algorithm. If  $N$  satisfies  $p'_{max,s} \leq N.minp$ , then all the descendants of  $N$  can be pruned. In other words, we do not need to process all descendant nodes of  $N$ .

Consider that we want to compute  $p_{max,d}$ . Suppose that we process a node  $N$  during the execution of the algorithm. If  $N$  satisfies  $p'_{max,d} \geq N.maxp$ , then all the descendants of  $N$  can be pruned. In other words, we do not need to process all descendant nodes of  $N$ .

**Example 10 (Pruning in the Aggregate  $R^*$ -tree)** *We can compute  $p_{max,s}$  with pruning by Lemma 5.3.4. Suppose currently  $p'_{max,s} = 80$  and we process node  $N_1$ . Note that  $N_1.minp = 100$  and  $N_1.maxp = 250$ . Since  $p'_{max,s} < N_1.minp$ , we can prune the subtree rooted at  $N_1$  immediately and continue processing the next node in the predefined ordering.*

*Similarly, we can compute  $p_{max,d}$  with pruning by Lemma 5.3.5. Suppose currently  $p'_{max,d} = 270$  and we process node  $N_1$ . Since  $p'_{max,d} > N_1.maxp$ , we can prune the subtree rooted at  $N_1$  immediately and continue processing the next node in the predefined ordering.* □

After describing how we can do the pruning operation, we present how we process the nodes in the  $R^*$ -tree in a pre-defined order. In the following, we describe how we can do the pruning operation for computing  $p_{max,s}$ . How we can do the pruning for computing  $p_{max,d}$  is similar. We do not discuss here. Specifically, we have a variable  $L$  which denotes a list of nodes to be processed in the  $R^*$ -tree. The nodes in  $L$  are sorted

in the ascending order of their *minp* values during the execution of the algorithm. Initially,  $L$  is  $\emptyset$ . The algorithm has the following steps.

- **Step 1 (Initialization):** Insert the root node of the R\*-tree into  $L$  and  $p'_{max,s}$  is set to  $\infty$ .
- **Step 2 (Iterative Step):**
  - We remove the first node  $N$  (i.e., the node with the smallest *minp* value in  $L$ ) from  $L$ .
  - If  $N$  is an internal node, we perform the following operations.
    - \* If  $p'_{max,s} \leq N.minp$  or  $N$  does not overlap with  $\mathcal{I}$ , then  $N$  is pruned.
    - \* Otherwise, we expand all child nodes of  $N$  and insert them into  $L$ .
  - If  $N$  is a leaf node, we perform the following operations.
    - \* If  $N$  is inside  $\mathcal{I}$ , then we update  $p'_{max,s}$  accordingly. (Suppose that  $N$  corresponds to the service-site  $h$ . If  $h.p < p'_{max,s}$ , then  $p'_{max,s}$  is updated to  $h.p$ .)
    - \* Otherwise,  $N$  is discarded and we continue the process by jumping to Step 1 for execution.
- **Step 3 (Termination):** This process terminates when  $p'_{max,s} \leq N.minp$  or  $L$  is empty.

**Example 11 (Processing  $L$ )** Suppose the R\*-tree is given in Figure 5.5(b). The process of computing  $p_{max,s}$  is as follows. Initially,  $L = \langle N_1 \rangle$ ,  $p'_{max,s} = \infty$ .

- Iteration 1, we remove  $N_1$  from  $L$ . As  $p'_{max,s} > N_1.minp$  and  $N_1$  overlaps with  $\mathcal{I}$ , we expand all child nodes of  $N_1$  (i.e.,  $N_2$  and  $N_3$ ) and insert them into  $L$ . Since  $N_2.minp < N_3.minp$ ,  $N_2$  is inserted before  $N_3$ . Currently,  $L = \langle N_2, N_3 \rangle$ .

- Iteration 2,  $N_2$  is removed from  $L$ . Similar to  $N_1$ ,  $N_2$  is an internal node that satisfies  $p'_{max,s} > N_2.minp$  and  $N_2$  overlaps with  $\mathcal{I}$ . Thus,  $N_2$  is expanded and all the child nodes are inserted into  $L$  where the nodes are sorted in the ascending order of the  $minp$  value (or the price value). Currently,  $L = \langle h_1, N_3, h_4 \rangle$ .
- Iteration 3,  $h_1$  is removed from  $L$  and  $h_1$  is a leaf node inside  $\mathcal{I}$ . Therefore,  $p'_{max,s}$  is updated to  $h_1.p$ , which is 100. Currently,  $L = \langle N_3, h_4 \rangle$ .
- Iteration 4,  $N_3$  is removed from  $L$ . Note that  $p'_{max,s} < N_3.minp$ . The process terminates and  $p_{max,s} = 100$  finally. □

## 5.4 Discussion

In this section, we focus on three issues related to our problem. The first one is how to apply our method when multiple non-spatial attributes are considered (Section 5.4.1). The second one is how to find a reasonable value of  $K$  for the  $K$ -dominating requirement (Section 5.4.2).

### 5.4.1 Handling Multiple Non-spatial Attributes

In this section, we will extend our problems and our proposed techniques to a general scenario when there are multiple non-spatial attributes. In previous sections, we study one single non-spatial attributes, namely attribute Price, in order to simplify our discussion. In our running example, each service-site can have multiple non-spatial attributes. For example, in addition to attribute Price, it can have non-spatial attributes such as star rate. Under this general scenario, the two problems studied in this paper are still the same except the definition of dominance relationship among service-sites for the decision making table.

Formally, suppose each service-site  $h$  has  $q$  non-spatial attributes, namely  $\gamma_1, \gamma_2, \dots, \gamma_q$ . Without loss of generality, let the last attribute  $\gamma_q$  be attribute Price. For  $k \in [1, q]$ , the value of each attribute  $\gamma_k$  for  $h$  is represented by  $h.\gamma_k$ . In this problem, for each existing service-site  $h$  and each non-spatial attribute  $\gamma_j$ ,  $h.\gamma_j$  is given. For the new service-site  $h_f$ , each non-spatial attribute  $\gamma_j$  other than  $\gamma_q$  is given. We want to find  $h_f.\gamma_q$  to satisfy the skyline requirement and the  $K$ -dominating requirement. That is, we study Problem 5 but there are multiple non-spatial attributes (instead of a single non-spatial attribute).

We adapt the construction of the decision-making table with  $n + q$  attributes as follows. For each object  $h \in H$ , we construct a tuple in form of  $(x_1, x_2, \dots, x_n, x_{n+1}, x_{n+2}, \dots, x_{n+q})$  where  $x_j$  is equal to  $d(h, a_j)$  for each  $j \in [1, n]$  and  $x_j$  is equal to  $h.\gamma_{j-n}$  for each  $j \in [n + 1, n + q]$ . We denote each value  $x_j$  by  $h.X_j$  by  $j \in [1, n + q]$ . Let  $\mathcal{X}' = \{X_1, X_2, \dots, X_{n+q}\}$ . The dominance relationship is defined on  $\mathcal{X}'$  instead of  $\mathcal{X}$ . With this adapted dominance relationship, we can define the two problems accordingly under this general scenario.

Our proposed approach (Algorithm 17) can be adapted to this general scenario. All changes in the algorithm for this scenario are related to the adapted dominance relationship which involves the *additional* non-spatial attributes. Specifically, there are two changes in the algorithm. The first change is related to the skyline requirement. Let  $H'$  be a set of service-sites  $h$  inside  $\mathcal{I}$  such that, for each non-spatial attribute  $\gamma$  other than attribute Price (i.e.,  $\gamma_q$ ),  $h.\gamma \leq h_f.\gamma$ . In Line 7 of Algorithm 17, we modify it to “ $p_{max,s} \leftarrow \min_{h \in H'} h.p$ ”. The second change is related to the  $K$ -dominating requirement. Let  $H''$  be a set of service-sites  $h$  not in  $\mathcal{U}$  such that, for each non-spatial attribute  $\gamma$  other than attribute Price (i.e.,  $\gamma_q$ ),  $h_f.\gamma \leq h.\gamma$ . In Line 9 of Algorithm 17, we modify it to “ $p_{max,d} \leftarrow$  the  $K$ -th greatest price among all service-sites  $h$  in  $H''$ ”. Let us call the modified algorithm *Improved Algorithm*. It is easy to verify the following

Table 5.2: The running example with 2 non-spatial attributes

Service-site	Star rate ( $\gamma_1$ )	Price ( $\gamma_2$ )
$h_1$	3	100
$h_2$	4	250
$h_3$	3	200
$h_4$	4	220

Table 5.3: A decision-making table  $T$ 

Hotel	Distance-to-beach(km)	Star rate	Price(\$)
$h_1$	3.0	3	100
$h_2$	1.0	4	250
$h_3$	4.0	3	200
$h_4$	2.5	4	220

theorem (Since the *Improved Algorithm* is similar to Algorithm 17, the correctness of Theorem 5.4.1 can be directly derived from the proof of Theorem 5.3.1. Therefore, we omit the proof).

**Theorem 5.4.1** *Improved Algorithm returns the optimal solution of problem Finding  $K$ -Dominating Competitive Price when there are multiple non-spatial attributes.*  $\square$

**Example 12 (Multiple Non-Spatial Attributes)** *Let us illustrate the algorithm with our running example. We use the example as shown in Figure 5.1 but the price table is changed from the table in Figure 5.1 (which contains one non-spatial attribute, namely “price”) to the table in Table 5.2 (which contains two non-spatial attributes, namely “star rate” and “price”). Here, attribute “star rate” corresponds to  $\gamma_1$  while attribute “price” corresponds to  $\gamma_2$ . Note that different from attribute “price” in which a smaller value is more preferable, in attribute “star rate”, a larger value is more preferable.*

*Similarly, we have the corresponding decision-making table as shown in Table 5.3.*

*In this example, suppose that we set the star rate of the new hotel  $h_f$  to 3 and the location of  $h_f$  to the location as shown in Figure 5.2. Let  $K$  be 2.*

*In Phase 1, we find all the hotels in  $\mathcal{I}$ . Only one service-site,  $h_2$ , locates inside  $\mathcal{I}$ . Since  $h_2.\gamma_1 = 4 > 3 = h_f.\gamma_1$ ,  $H' = \{h_2\}$ . Therefore,  $p_{\max,s}$  is set to the price of  $h_2$ ,*

\$250.

*In Phase 2, we find all the hotels not in  $\mathcal{U}$ . There are three service-sites,  $h_1$ ,  $h_3$  and  $h_4$ , not in  $\mathcal{U}$ . And all of them have a value on  $\gamma_1$  no less than  $h_f$ . Thus,  $H'' = \{h_1, h_3, h_4\}$ . The 2-nd greatest price among all the hotels in  $H''$  is the price of  $h_3$ , \$200. Therefore,  $p_{max,d}$  is set to be \$200.*

*In Phase 3, we can compute  $p_{max} = \min\{\$250, \$200\} = \$200$ . Thus, the 2-dominating price of  $h_f$  is \$200.* □

### 5.4.2 How to set $K$

How to set a proper  $K$  is essential for the  $K$ -dominating requirement. In some cases, users have their mind to set the value of  $K$  because they want that  $h_f$  must dominate at least  $K$  service-sites in the existing market. However, in some other cases, users do not have any idea about how to set the value of  $K$ . In this section, we propose a method to help users to determine the value of  $K$  in this case.

As we know, different values of  $K$  result in different prices, but which price among those is more reasonable and thus benefits the new service-site is still not known. In this subsection, we propose a model to suggest a way to find an appropriate value of  $K$  based on the well-studied field of economy and business [33].

There are some existing models in economical and business research studying *customer retention/attrition* [33]. In this subsection, we borrow the concept of the traditional demand-and-supply model [33] in the literature to find an appropriate value of  $K$ .

In this model, each service-site  $h$  is associated with a *demand*, denoted by  $h.d$ , representing the total number of customers that would like to choose this service-site.

In our running example, if there are 50 customers who want to accommodate in hotel  $h$ , the demand of this hotel  $h.d$  is 50. In our problem, for each  $h \in H$ ,  $h.d$  is given. However,  $h_f.d$  is not given. In the following, we propose a model to estimate  $h_f.d$ .

The demand of each service-site can be obtained from some external sources of information about the proportion of the usage of the service-sites. In our motivating example, this information can be the proportion of using the hotels [9]. Similar kinds of information are also available in other applications like setting a selling price of an apartment [8] and setting a parking fee of a car park [10].

**Definition 5.4.1 (Potential Loser)** *Given a service-site  $h$  in  $H$ ,  $h$  is said to be a potential loser if and only if there exists a non-negative real number  $p$  such that when  $h_f.p$  is set to  $p$ ,  $h_f$  dominates  $h$ .*

We define the set of all potential losers in  $H$  to be  $PL$ . Without loss of generality, we assume that  $PL$  contains  $l$  service-sites. Note that  $h_f$  dominates at most  $l$  service-sites if we set  $h_f.p$  to a particular non-negative real number. Thus, the greatest possible value of  $K$  that we can set is  $l$ . Without loss of generality, we assume that  $PL$  contains  $h_1, h_2, \dots, h_l$  where  $h_i.p \geq h_{i+1}.p$  for  $i = 1, 2, \dots, l - 1$ .

**Definition 5.4.2 (Real Loser)** *Given a potential loser  $h$  and a non-negative real number  $p$ ,  $h$  is said to be a real loser with respect to  $p$  if and only if  $h_f.p$  is set to  $p$  and  $h_f$  dominates  $h$ .*

Given a non-negative real number  $p$ , we define the set of all real losers with respect to  $p$  to be  $RL(p)$ . If  $h$  is a real loser with respect to  $p$ , then we know that  $h_f$  dominates  $h$ . Similar to  $PL$ , we also assume that  $h_i.p \geq h_{i+1}.p$  for  $i = 1, 2, \dots, l - 1$  without loss of generality. Thus, we know that some of the current customers choosing  $h$  may change their preference and finally choose  $h_f$  instead of  $h$ . However, in the real-life

applications, not all customers choosing  $h$  are eager to making this change. In order to capture this, we define an input parameter  $\alpha$  which is a real number between 0 and 1 and denotes the *transfer rate* representing the fraction of customers who originally want to choose  $h$  (before  $h_f$  is set up) and finally choose  $h_f$  (after  $h_f$  is set up). Similarly, the transfer rate  $\alpha$  can be obtained from some external sources of information about customers' behaviors. There are a lot of studies related to customers' behaviors in the literature of psychology, sociology, social anthropology and economics. This information can be found in the studies about customers' behaviors [78, 66].

Suppose that  $h_f.p$  is set to  $p$ . According to this transfer rate, given a real loser  $h$  with respect to  $p$ , there are  $\alpha \times h.d$  customers who originally want to choose  $h$  (before  $h_f$  is set up) and finally choose  $h_f$  (after  $h_f$  is set up). Thus, by considering all real losers in  $RL(p)$ , we deduce that the total number of customers who originally want to choose some real losers with respect to  $p$  and finally choose  $h_f$  is equal to

$$\sum_{h \in RL(p)} \alpha \times h.d$$

Let us denote the above equation by a function  $f$  as follows.

$$f(p) = \sum_{h \in RL(p)} \alpha \times h.d$$

Now, we are ready to define a formula for  $h_f.d$  as follows.

$$h_f.d = f(p)$$

Next, we describe how to determine the value of  $K$  by using a new concept of *utility*. Given a non-negative real number  $p$ , the *utility* of  $h_f$  with respect to  $p$ , denoted



by  $U(p)$ , is defined as follows.

$$U(p) = p \times f(p)$$

Note that  $f(p)$  corresponds to  $h_f.d$ . We conclude that  $p \times f(p)$  corresponds to the total income for  $h_f$ .

Let  $u_i = U(h_i.p)$  for  $i = 1, 2, \dots, l$ . We deduce that

$$u_i = h_i.p \times f(h_i.p)$$

Note that in both PL and RL(p),  $h_i.p$  is monotonically decreasing when  $i$  increases. Besides,  $f(h_i.p)$  is monotonically increasing when  $i$  increases. Thus, we do not know whether  $u_i$  is larger than  $u_{i+1}$  or not where  $i = 1, 2, \dots, l - 1$ .

According to  $u_1, u_2, \dots, u_l$ , we want to determine the value of  $K$  as follows. Firstly, we calculate all values  $u_1, u_2, \dots, u_l$ . Secondly, we find an integer  $i_o$  which gives the greatest value of  $u_{i_o}$  among  $u_1, u_2, \dots, u_l$  (i.e.,  $i_o = \operatorname{argmax}_i u_i$ ). Thirdly, we set  $K$  to  $i_o$ .

**Example 13 (Finding  $K$ )** *In the running example of Figure 5.2, the set of potential losers is  $\{h_1, h_3, h_4\}$ . Thus,  $l$  is 3 and the greatest possible value of  $K$  we can set is 3. Suppose that  $h.d$  is set to 100 for each  $h \in H$  and  $\alpha$  is set to 0.1. We obtain that  $u_1 = 2200$ ,  $u_2 = 4000$  and  $u_3 = 3000$ . Thus, the value  $K$  is 2 and the maximum utility is 4000. We will apply this model in the case study described in Section 5.5.  $\square$*

## 5.5 Empirical Studies

In this section, we verify the scalability of our proposed algorithms. The algorithms were implemented in C/C++. All the experiments were performed on a 2.4GHz PC

Table 5.4: Default values

$m$	200000
$n$	20
$K$	20
$x$	150
$\sigma$	27
$q$	1

with 4.0GB RAM, on a Linux platform. We ran experiments on both real and synthetic datasets.

The synthetic datasets were generated as follows. Firstly, we collect the locations of objects in North American (e.g., roads, populated places and cultural landmarks) from Digital Chart of the World [11]. Let  $W$  be the set of objects. Then, from  $W$ , we randomly select  $m$  objects as service-sites and  $n$  objects as attraction-sites. These service-sites and these attraction-sites form set  $H$  and set  $A$ , respectively. The location of a new service-site  $h_f$  is randomly generated. Since there is no attribute related to price in set  $H$ , we generate the price of each service-site in  $H$  as follows. For each service-site  $h$  in  $H$ , we find its nearest attraction-site  $a$  and set  $h.p$  to be a value which is randomly picked from a normal distribution with mean equal to  $x/(1 + d(h, a)^2)$  and standard derivation equal to  $\sigma$ , where  $x$  and  $\sigma$  are two input parameters. Intuitively,  $x$  is the (expected) greatest possible price of a service-site when we consider the nearest attraction-site. By default, we adopt  $x = 150$  and  $\sigma = 27$  since the real dataset (to be described next) has such a distribution. The default values of some parameters in our experiment are shown in Table 5.4. In the following, we use the default settings unless specified otherwise.

The real dataset was obtained from Surfy Hotel [12] which provides information about hotels in North America, including price and location. We select 20,000 hotels for our experiment. We chose four attraction-sites, namely Status of Liberty, Empire State Building, Museum of Art and Wall Street. Similarly, the location of a new service-site  $h_f$  is randomly generated.

We implemented four algorithms, namely (a) *Blind-Naive*, (b) *Guided-Naive*, (c) *3-Phase(No Index)* and (d) *3-Phase(Index)*. (a) *Blind-Naive* is the naive algorithm we described at the end of Section 5.2. In our implementation, *Blind-Naive* tries to find a set of possible prices starting from 0 with an incremental count of 0.01 (i.e., 0.00, 0.01, 0.02, ...) such that these prices meet the skyline requirement and the  $K$ -dominating requirement. It continues the process until the largest price which meets the requirements is found. This largest price corresponds to the answer of *Blind-Naive*. (b) *Guided-Naive* is similar to *Blind-Naive*. Instead of trying all possible prices in an increment count of 0.01, *Guided-Naive* tries to find all possible prices of existing service-sites. These prices can be considered as candidates for the competitive price. Similarly, it tries to find the largest price among these prices which meet the requirements as the final answer. (c) *3-Phase(No Index)* corresponds to Algorithm 17 (which is a three-phase algorithm) but it is not equipped with any index. Specifically, it has to find  $\mathcal{I}$  and  $\mathcal{U}$  without using any index in order to determine  $p_{max,s}$ ,  $p_{max,d}$  and  $p_{max}$ . (d) *3-Phase(Index)* corresponds to Algorithm 17 and it is equipped with an index, namely an aggregate R\*-tree [89], for computation. We adopted an aggregate R\*-tree [89] for the range query in *3-Phase(Index)* where the maximum number of entries in a node is equal to 20 and the minimum number of entries in a node is equal to 10.

In the following, for clarity, we simply denote *Blind-Naive* and *Guided-Naive* as *Blind* and *Guided*, respectively.

Since problem Finding  $K$ -Dominating Competitive Price is a more general problem than Finding Simple Competitive Price, in the following experimental results, we study the former problem only. In Section 5.4, we described how we extend our algorithm for the general scenario which involves  $q$  non-spatial attributes. In the experiment, we also conducted experiments by varying  $q$  from 1 to 4.

Section 5.5.1 studies the scalability of our proposed technique, Section 5.5.2 gives

a case study of our problem and Section 5.5.3 gives some results on how to find an appropriate value of  $K$ .

### 5.5.1 Scalability

We evaluate the algorithms with four measurements: (1) *Price*, (2) *Query Time*, (3) *Storage* and (4) *Proportion of Pruned Points*. (1) Price corresponds to three different types of prices, denoted by  $p_{max,s}$ ,  $p_{max,d}$  and  $p_{max}$ , respectively. Since  $p_{max,s}$  may be equal to  $\infty$  in some cases, in our experiment, we only report any value not equal to  $\infty$  for  $p_{max,s}$ . Besides, note that  $p_{max} = \min\{p_{max,s}, p_{max,d}\}$ . (2) Query time refers to the time of executing the algorithm to find the price. (3) Storage is the total memory consumption used for the algorithm. The storage of *3-Phase(Index)* is the memory occupied by the aggregate R\*-tree. The storage of other three algorithms is the memory occupied by the decision-making table. (4) Proportion of pruned points is the ratio of points (or objects) that the algorithm does not need to read. Specifically, since *3-Phase(Index)* has an indexing structure, some objects need not be read. For each measurement, each experiment was conducted 1,000 times and the average of the results was reported. We studied the effects of  $K$ ,  $n$ ,  $m$  and  $q$  as follows.

*Effect of  $K$* : Figure 5.6(a) shows when  $K$  increases,  $p_{max,d}$  decreases and  $p_{max,s}$  remains the same.  $p_{max,s}$  is not affected by the number of dominated service-sites, while  $p_{max,d}$  decreases since the  $K$ -th greatest price of the service-sites outside  $\mathcal{U}$  decreases when  $K$  increases. In Figure 5.6(b), the query time of each algorithm increased slightly except *Guided* when  $K$  increases. This is because each algorithm has to find more existing service-sites which are dominated by  $h_f$  when  $K$  increases. Note that the query time of *3-Phase(Index)* is two orders of magnitude less than *3-Phase(No Index)*, and much less than the other two algorithms. Besides, the query times of the four algorithms decrease in the order of *Blind*, *Guided*, *3-Phase(No Index)*, *3-Phase(Index)*.

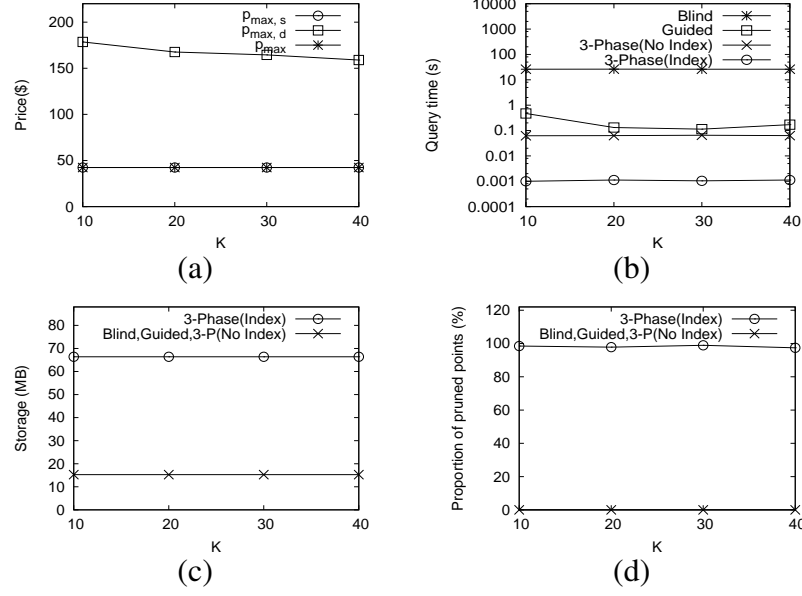


Figure 5.6: Effect of  $K$  (the size of the final selection set)

This order also appears in all other figures of query time measurement (which will be shown later). Figure 5.6(c) shows the storage of the algorithms does not change significantly when  $K$  increases. However, in Figure 5.6(d), as  $K$  increases, the proportion of pruned points of *3-Phase(Index)* decreases slightly. This is because when  $K$  increases, there are more service-sites which should be dominated by  $h_f$ . Thus, more service-sites should be considered and thus fewer service-sites (or points) are pruned when  $K$  increases.

*Effect of  $n$ :* In Figure 5.7(a), when  $n$  increases,  $p_{max,s}$  increases. This is because, if the total number of attraction-sites increases, then it is less likely that a service-site dominates the new service-site  $h_f$ . However,  $p_{max,d}$  decreases slightly when  $n$  increases. This is because, if there are more attraction-sites, then similarly, it is also less likely that the new service-site  $h_f$  dominates other service-sites. However, in Figure 5.7(b), the query time of *3-Phase(Index)* remains nearly the same when  $n$  increases, but the query time of other three algorithms increases as  $n$  increases. Figure 5.7(c) and 5.7(d) shows that  $n$  does not affect the storage and the proportion of pruned points of *3-Phase(Index)*. In Figure 5.7(c), the storage of the other three algorithms increases

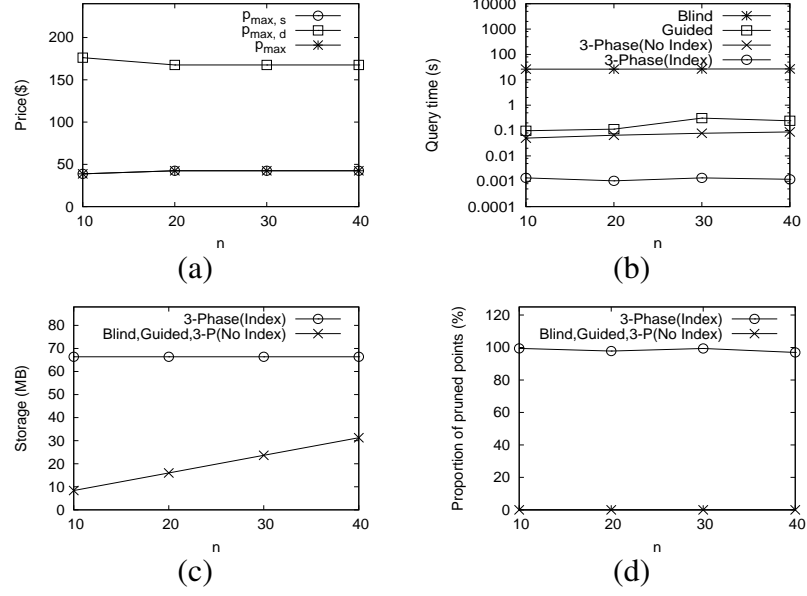


Figure 5.7: Effect of  $n$  (the number of attraction-sites)

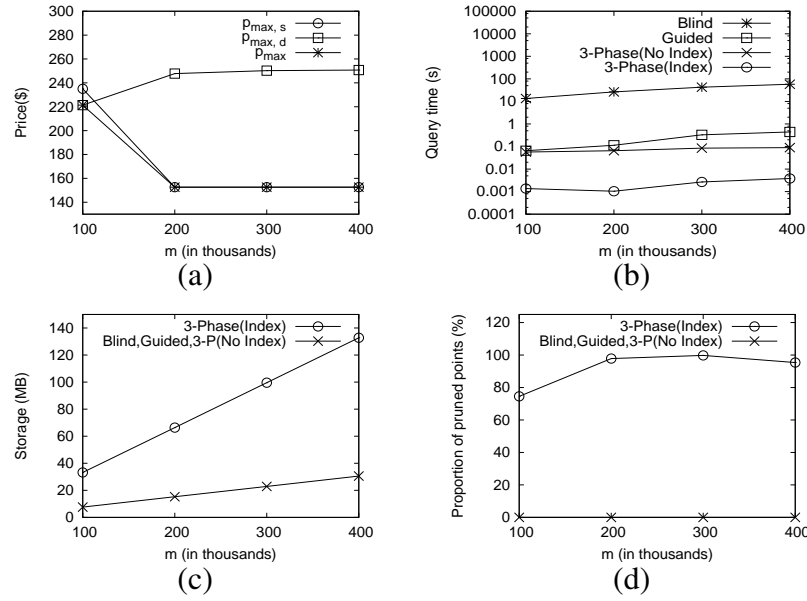


Figure 5.8: Effect of  $m$  (the number of service-sites)

with  $n$ , because the decision-making table used by the three algorithms becomes larger when  $n$  increases.

*Effect of  $m$ :* Figure 5.8(a) shows that, when the number of service-sites increases,  $p_{max,s}$  decreases. This is because, if there are more service-sites, then it is more likely that a service-site dominates the new service-site  $h_f$ . However, in the figure,  $p_{max,d}$  increases with the number of service-sites because it is more likely that  $h_f$  can dominate other service-sites. Thus,  $p_{max,d}$  can be higher. In Figure 5.8(b), the query times

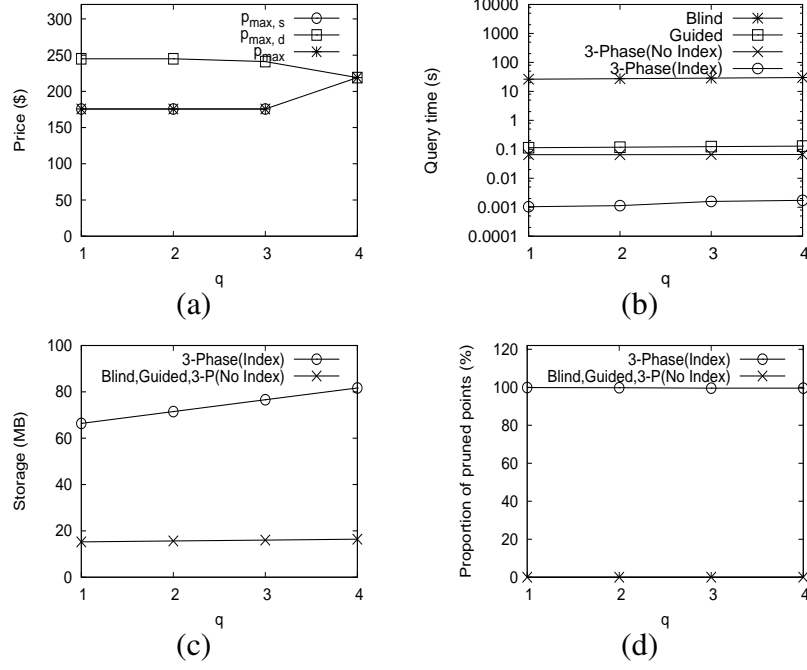


Figure 5.9: Effect of  $q$  (the number of non-spatial attributes)

of both all algorithms increase with  $m$ . The storage of all the algorithms increases with  $m$ , as shown in Figure 5.8(c). In Figure 5.8(d), the proportion of pruned points of *3-Phase(Index)* increases slightly when  $m$  increases.

*Effect of  $q$ :* We considered four non-spatial attributes, namely Price, Star Rate, No. of Transportation Types, and Customer Confidence. In the experiments, the number of non-spatial attributes, denoted by  $q$ , increased from 1 to 4 according to the above-mentioned order. Figure 5.9(a) shows that  $p_{max,s}$  increases and  $p_{max,d}$  decreases when  $q$  increases. The explanation is similar to the one described in “Effect of  $n$ ” because, when  $q$  increases, it is less likely that a server-site dominates a new service-site  $h_f$  and it is less likely that  $h_f$  dominates other service-sites. Figures 5.9(b) and (c) shows that the query times and the storage of all algorithms increases with  $q$ . In Figure 5.9(d), the proportion of pruned points for *3-Phase(Index)* remains nearly unchanged when  $q$  changes.

*Effect of  $K$  on Real Dataset:* We conducted experiments on real datasets, and the experimental results with the variation of  $K$  are shown in Figure 5.10. The trends are

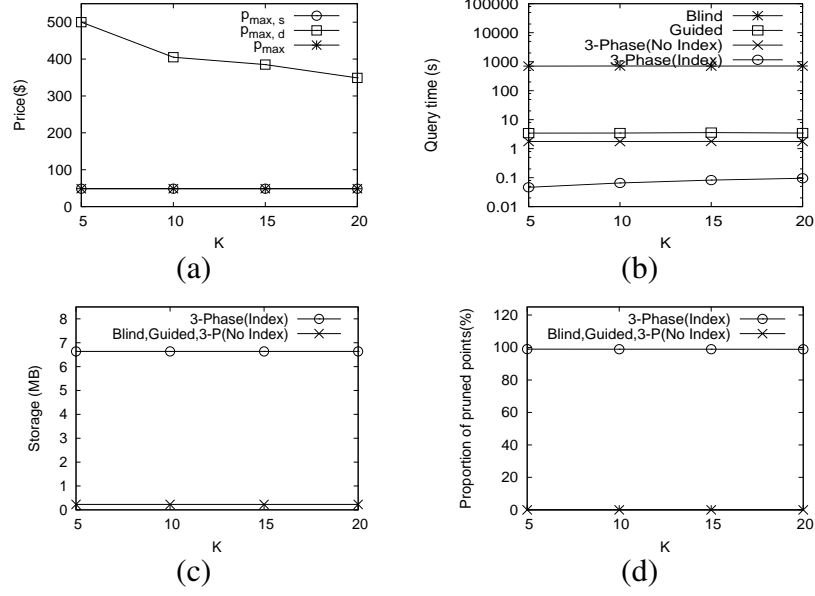


Figure 5.10: Effect of  $K$  (the number of final selection set) similar to those for the synthetic dataset.

### 5.5.2 Case Study

In order to illustrate the practicality of our algorithm, we use a real dataset with Manhattan hotels to show how it can be used in reality. In this dataset, hotels correspond to service-sites and attractions correspond to attraction-sites in our problem setting. We choose two attractions, namely Empire State Building and Bull Sculpture in the Wall Street, denoted by two black triangles as shown in Figure 5.11. Besides, suppose the new location where we want to set up a new hotel  $h_f$  is denoted by the black square box near Empire State Building as shown in Figure 5.11. In this dataset, there are 154 hotels. For the sake of illustration, 30 hotels out of 154 hotels are shown in Figure 5.11. In the dataset, the prices of these 154 hotels ranges from \$70 to \$463. The average price and the standard deviation of these 157 hotels are \$153 and \$67, respectively. There are 13 1-star hotels, 65 2-star hotels, 48 3-star hotels, 27 4-star hotels and 1 5-star hotel. We conducted two sets of experiments for the case study. The first set is that each hotel (or service-site) is associated with its location and its price only. The second set is that each hotel is associated with its star-rate (one additional non-spatial



Table 5.5: Value of  $p_{max}$  under different star rates

Star Rate of $h_f$	$K = 1$	$K = 2$
1	81	81
2	81	80
3	99	96
4	149	141
5	284	209

attribute) in addition to its location and its price.

For the first set, we find that  $p_{max,s} = \$81$ . If we set  $K = 1$ , then  $p_{max,d} = \$126$ . Thus,  $p_{max} = \min\{p_{max,s}, p_{max,d}\} = \$81$ . Similarly, if we set  $K = 2$ , we have  $p_{max,s} = \$81$  and  $p_{max,d} = \$119$ . Thus, we have  $p_{max} = \$81$ . For the second set,  $p_{max}$  has different values if we set the star rate of the new hotel to different star rates. Table 5.5 shows the results of  $p_{max}$  with different star rates of the new hotel  $h_f$ . In general, if the star rate of  $h_f$  is higher, then  $p_{max}$  is higher. This is because usually, an existing hotel with a higher star rate has a higher price. When  $K$  is higher,  $p_{max}$  decreases. This is because in order that  $h_f$  can dominate more hotels,  $p_{max}$  is set to a smaller value.



Figure 5.11: Attractions and hotels in Manhattan

### 5.5.3 Experiments for Determining An Appropriate Value of $K$

In the following, the objective is to determine the value of  $K$  by using the method introduced in Section 5.4.2. Here, we also use the real dataset in Section 5.5.2. There are the following three major steps. The first step is to obtain the demands of all the hotels in the existing market. The second step is to obtain the transfer rate  $\alpha$ . The final step is to find an appropriate value of  $K$  according to the information obtained in the previous steps.

The first step can be done by gathering the information about the *hotel occupation rate* (which corresponds to the proportion of the number of rooms occupied by customers). According to the NYC statistical report [7], on average, the occupation rate of a hotel in New York City is around 80%. With this occupation rate, we can calculate the demand of each hotel by multiplying the total number of rooms in the hotel with this occupation rate.

The second step can be done similarly by gathering the information about customers' behaviors. We derive the transfer rate  $\alpha$  from two sources of information. According to the first source, a statistical report from The Harvard Business Review, a company loses 50% of their customers every five years on average [78]. In other words, 10% of customers are lost every year. According to the second source, the report from the U.S. Small Business Administration and the U.S. Chamber of Commerce [66], about 82% of the customers who do not continue choosing the original company choose the other better companies finally because they are upset with the treatment they have received from the original company. By combining the above two sources, we conclude that about 8.2% of all customers who originally choose a particular company will probably choose other better companies finally. Thus, we obtain  $\alpha = 0.082$ .

The third step is to find an appropriate value of  $K$  according to the information

obtained in the previous steps. In this experiment, we find that the set of potential losers contain 36 hotels. Suppose that we set the star rate of the new hotel to be 2. According to the method introduced in Section 5.4.2, we first find  $u_1, u_2, \dots, u_{36}$ . In this experiment,  $u_1 = 894.26$  and  $u_2 = 3345.34$ . We also compute  $u_i$  for  $i = 3, 4, \dots, 36$ . Finally, we find that  $u_{35}$  has the highest value among all values  $u_1, u_2, \dots, u_{36}$  and it is equal to 26834.46. Thus, we find the appropriate value of  $K$  as 35. After we set  $K$  to be 35, we can find the corresponding competitive price of  $h_f$  as \$79, which is really competitive compared to other prices.

#### 5.5.4 Summary

We find that *3-Phase(Index)* finds the competitive price of a new service-site more efficiently compared with algorithms *Blind*, *Guided* and *3-Phase(No Index)*. Generally, *3-Phase(Index)* performs faster than other algorithms at least two orders of magnitude.

### 5.6 Conclusions

In this chapter, we identified and tackled two interesting data mining problems, finding simple competitive price and finding  $K$ -dominating competitive price, considering the skyline concept as generalized user preferences, the same as the previous chapter. A spatial approach was proposed to the problem of finding  $K$ -dominating competitive price by using the spatial properties. The problem of finding simple competitive price can also be solved by this approach with proper parameter settings. We conducted experiments to show the efficiency of our proposed approach and illustrated the process with a real case study.

## CHAPTER 6

### CONCLUSIONS AND FUTURE PLANS

In this thesis, we report our work with two problems in the context of user preference analysis. One is *understanding user preferences* when temporal user preferences are considered. The other is *utilizing user preferences* when generalized user preferences and tolerant use preferences are considered.

Consider the first problem of *understanding user preferences*. We proposed a new problem called *Attribute-based Subsequence Matching Problem* which has many applications. This problem considers customer preferences related to the property table which has not been studied extensively in the literature. We propose an efficient algorithm for this problem using Chinese Remainder Theorem to compress each sequence into a triplet of numbers. We also illustrate how this problem can be used for mining frequent attribute-based subsequences. Finally, we conduct experiments to show that our algorithm is very efficient, nearly two orders of magnitude better than the straightforward method. We also apply the proposed MA algorithm into *Frequent Attribute-based Subsequence Mining Problem*. The experimental results show some unexpected interesting frequent subsequences.

Consider the second problem of *utilizing user preferences*. In this problem, we study two kinds of preferences, namely generalized user preferences and tolerant user preferences. Firstly, we identify an interesting sub-problem when generalized user preferences are considered, *finding top-k profitable products*, which has not been studied before. Given a set of products in the existing market, we want to find a set of  $k$  “best” possible products such that these new products are not dominated by the

products in the existing market. We need to set the prices of these products such that the total profit is maximized, and we assume that the user preferences are based on the *skyline* concept which is related to generalized user preferences. We refer such products as top- $k$  profitable products. Secondly, we consider additionally tolerant user preferences. We want to find  $k$  products such that these  $k$  products can attract the greatest number of customers. That is, we assume that the popularity of a product can be measured as the number of tolerant user preferences satisfied by this product. We refer these products as top- $k$  popular products. So, the second sub-problem we focused on is *finding top- $k$  popular products*. In above two problems, a straightforward solution is to enumerate all possible subsets of size  $k$  and find the subset which gives the greatest profit (for the first sub-problem) or attracts the greatest number of customers (for the second sub-problem). However, there are an exponential number of possible subsets. In this thesis, we report our solutions to find the top- $k$  profitable products and the top- $k$  popular products efficiently. An extensive performance study using both synthetic and real datasets is reported to verify the effectiveness and the efficiency of proposed algorithms.

In the third sub-problem, we also utilize generalized user preferences to find the competitive price for a new service when spatial databases are considered. We identify and tackle an interesting problem, finding  $K$ -dominating competitive price by considering the *skyline* concept related to generalized user preferences. Although setting price comes naturally in many real life applications, we are the first to propose to find the price of a new service-site with the *skyline* concept.

For the future plans, we are extending our work in the first problem when the hierarchical taxonomy of attributes are given. Meanwhile, we want to take more practical factors into consideration when delivering new products to the market. For example, we are now working on combining both profit and popularity as a goal in our problem.

Besides, we plan to formalize new problems for other underlying factors which may affect our problem setting according to real scenarios.

## REFERENCES

- [1] <http://gmplib.org/>.
- [2] <http://www.amazon.com/>.
- [3] <http://www.cse.ust.hk/~raywong/genealogy/>.
- [4] <http://www.expedia.com/>.
- [5] <http://www.informatik.uni-freiburg.de/~chiegler/bx/>.
- [6] <http://www.priceline.com/>.
- [7] Nyc statistics. In <http://www.nycgo.com/?event=view.article&id=78912>.
- [8] Realfacts. In <http://realfacts.com/>.
- [9] Smith travel research. In <http://www.strglobal.com/News/News.aspx>.
- [10] Town of riverhead parking management workshop. In <http://www.riverheadli.com/07.09.parking.management.workshop.pdf>.
- [11] R-tree portal. In <http://www.rtreeportal.org/spatial.html>, 2009.
- [12] Surfy hotel. In <http://www.surfy.com/>, 2009.
- [13] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *Proc. of VLDB Conference*, pages 487–499, 1994.
- [14] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. In *Journal of molecular biology*, 1990.

- [15] N. Archak, A. Ghose, and P. G. Ipeirotis. Show me the money!: deriving the pricing power of product features by mining consumer reviews. In *Proc. of ACM SIGKDD Conference*, pages 56–65, 2007.
- [16] M. J. Atallah and Y. Qi. Computing all skyline probabilities for uncertain data. In *Proc. of ACM PODS Conference*, pages 279–287, 2009.
- [17] V. Athitsos, P. Papapetrou, M. Potamias, G. Kollios, and D. Gunopulos. Approximate embedding-based subsequence matching of time series. In *Proc. of ACM SIGMOD Conference*, 2008.
- [18] J. Ayres, J. Flannick, J. Gehrke, and T. Yiu. Sequential pattern mining using a bitmap representation. In *Proc. of ACM SIGKDD Conference*, pages 429–435, 2002.
- [19] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R\*-tree: An efficient and robust access method for points and rectangles. In *Proc. of ACM SIGMOD Conference*, volume 19, pages 322–331, 1990.
- [20] J. L. Bentley, K. L. Clarkson, and D. B. Levine. Fast linear expected-time algorithms for computing maxima and convex hulls. In *Proc. of ACM-SIAM SODA Conference*, 1990.
- [21] J. L. Bentley, H. T. Kung, M. Schkolnick, and C. D. Thompson. On the average number of maxima in a set of vectors and applications. In *Journal of ACM*, 25(4), 1978.
- [22] L. Boasson, P. Cegielski, I. Guessarian, and Y. Matiyasevich. Window-accumulated subsequence matching problem is linear. In *Proc. of ACM PODS Conference*, 1999.



- [23] C. Böhm, F. Fiedler, A. Oswald, C. Plant, and B. Wackersreuther. Probabilistic skyline queries. In *Proc. of ACM CIKM Conference*, pages 651–660, 2009.
- [24] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *Proc. of IEEE ICDE Conference*, pages 421–430, 2001.
- [25] C. Chan, P. Eng, and K. Tan. Stratified computation of skylines with partially-ordered domains. In *Proc. of ACM SIGMOD Conference*, pages 203–214, 2005.
- [26] C.-Y. Chan, P.-K. Eng, and K.-L. Tan. Efficient processing of skyline queries with partially-ordered domains. In *Proc. of IEEE ICDE Conference*, pages 190 – 191, 5 2005.
- [27] C.-Y. Chan, P.-K. Eng, and K.-L. Tan. Efficient processing of skyline queries with partially-ordered domains. In *Proc. of IEEE ICDE Conference*, 2005.
- [28] C.-Y. Chan, H. V. Jagadish, K.-L. Tan, A. K. H. Tung, and Z. Zhang. Finding k-dominant skylines in high dimensional space. In *Proc. of ACM SIGMOD Conference*, pages 503–514, 2006.
- [29] B. Chazelle. New upper bounds for neighbor searching. In *Information and Control*, 1986.
- [30] L. Chen and X. Lian. Efficient processing of metric skyline queries. *IEEE Trans. on Knowl. and Data Eng.*, 21(3), Mar. 2009.
- [31] B. Cui, L. Chen, L. Xu, H. Lu, G. Song, and Q. Xu. Efficient skyline computation in structured peer-to-peer systems. *IEEE TKDE*, 21(7):1059–1072, July 2009.
- [32] A. Das Sarma, A. Lall, D. Nanongkai, and J. Xu. Randomized multi-pass streaming skyline algorithms. *Proc. of VLDB Conference*, 2(1):85–96, 2009.

- [33] R. B. David Besanko. *Microeconomics*. Wiley, 2004.
- [34] J. Delgado and N. Ishii. On-line learning of user preferences in recommender systems. *International Journal of KnowledgeBased Intelligent Engineering Systems*, 3(3), 2000.
- [35] E. Dellis and B. Seeger. Efficient computation of reverse skyline queries. In *Proc. of VLDB Conference*, pages 291–302. VLDB Endowment, 2007.
- [36] G. Dong and J. Pei. *Sequence Data Mining (Advances in Database Systems)*. Springer-Verlag, New York, Inc., 2007.
- [37] O. B.-N. et al. On the distribution of the number of admissible points in a vector random sample. In *Theory of Probability and its Application*, 11(2), 1966.
- [38] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. In *Proc. of ACM SIGMOD Conference*, 1994.
- [39] M. D. Gemmis, L. Iaquinta, P. Lops, C. Musto, F. Narducci, and G. Semeraro. Preference learning in recommender systems. In *Preference Learning (PL-09) ECML/PKDD-09 Workshop*, 2009.
- [40] P. Geurts, A. B. Cuesta, and L. Wehenkel. Segment and combine approach for biological sequence classification. In *Proc. of IEEE Symposium on Computational Intelligence in Bioinformatics and Computational Biology*, 2005.
- [41] R. L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Information Processing Letters*, 1(4), 1972.
- [42] J. Han, J. Pei, B. Mortazavi-Asl, Q. Chen, U. Dayal, and M.-C. Hsu. Freespan: frequent pattern-projected sequential pattern mining. In *Proc. of ACM SIGKDD Conference*, pages 355–359, 2000.

- [43] W.-S. Han, J. Lee, Y.-S. Moon, and H. Jiang. Ranked subsequence matching in time-series databases. In *Proc. of VLDB Conference*, 2007.
- [44] G. H. Hardy and E. M. Wright. *An Introduction to the Theory of Numbers*. Oxford, England: Clarendon Press, 1979.
- [45] J. L. Herlocker, J. A. Konstan, L. G. Terveen, and J. T. Riedl. Evaluating collaborative filtering recommender systems. *ACM TOIS*, 22(1), 2004.
- [46] D. S. Hockhbaum. *Approximating covering and packing problems: Set cover, vertex cover, independent set, and related problems in Approximation algorithms for NP-hard problems*. PWS Publishing Company, Boston, 1997.
- [47] S. Holland, M. Ester, and W. Kießling. Preference mining: A novel approach on mining user preferences for personalized applications. In *Proc. of PKDD Conference*, 2003.
- [48] Z. Huang, H. Lu, B. C. Ooi, and A. K. H. Tung. Continuous skyline queries for moving objects. *IEEE TKDE*, 18(12):1645–1658, December 2006.
- [49] Z. Huang, S. Sun, and W. Wang. Efficient mining of skyline objects in subspaces over data streams. *KAIS*, 2010.
- [50] B. Jiang and J. Pei. Online interval skyline queries on time series. In *Proc. of IEEE ICDE Conference*, pages 1036–1047, march 2009.
- [51] B. Jiang, J. Pei, X. Lin, D. W. Cheung, and J. Han. Mining preferences from superior and inferior examples. In *Proc. of ACM SIGKDD Conference*, pages 390–398, 2008.
- [52] W. Jin, M. Ester, Z. Hu, and J. Han. The multi-relational skyline operator. In *Proc. of IEEE ICDE Conference*, 2007.

- [53] J. M. Kang, M. F. Mokbel, S. Shekhar, T. Xia, and D. Zhang. Continuous evaluation of monochromatic and bichromatic reverse nearest neighbors. In *Proc. of IEEE ICDE Conference*, 2007.
- [54] W. Kießling. Foundations of preferences in database systems. In *Proc. of VLDB Conference*, pages 311–322, 2002.
- [55] W. Kießling and G. Köstler. Preference sql: design, implementation, experiences. In *Proc. of VLDB Conference*, pages 990–1001, 2002.
- [56] Y. Koren. Collaborative filtering with temporal dynamics. In *Proc. of ACM SIGKDD Conference*, pages 447–456, 2009.
- [57] F. Korn and S. Muthukrishnan. Influence sets based on reverse nearest neighbor queries. In *Proc. of ACM SIGMOD Conference*, 2000.
- [58] D. Kossmann, F. Ramsak, and S. Rost. Shooting stars in the sky: An online algorithm for skyline queries. In *Proc. of VLDB Conference*, pages 275–286, 2002.
- [59] B. Li, A. Ghose, and P. G. Ipeirotis. Towards a theory model for product search. In *Proc. of WWW Conference*, pages 327–336, 2011.
- [60] X. Lian and L. Chen. Monochromatic and bichromatic reverse skyline search over uncertain databases. In *Proc. of ACM SIGMOD Conference*, pages 213–226, 2008.
- [61] X. Lian and L. Chen. Reverse skyline search in uncertain databases. In *ACM TODS*, volume 35, pages 1–49, 2010.
- [62] X. Lin, Y. Yuan, Q. Zhang, and Y. Zhang. Selecting stars: the  $k$  most representative skyline operator. In *Proc. of IEEE ICDE Conference*, pages 86–95, 2007.

- [63] X. Liu, D. L. Brutlag, and J. S. Liu. Bioprospector: Discovering conserved dna motifs in upstream regulatory regions of co-expressed genes. In *Pacific Symposium on Biocomputing*, 2001.
- [64] H. Lu and C. S. Jensen. Upgrading uncompetitive products economically. In *Proc. of IEEE ICDE Conference*, 2012.
- [65] S. Iyong Lee, S. ju Chun, D. hwan Kim, J. hong Lee, and C.-W. Chung. Similarity search for multidimensional data sequences. In *Proc. of IEEE ICDE Conference*, 2000.
- [66] W. Maguire. Six reasons we lose customers. August 2007.
- [67] F. Masseglia, F. Cathala, and P. Poncelet. The psp approach for mining sequential patterns. In *Proc. of PKDD*, pages 176–184, 1998.
- [68] R. T. Ng, L. V. S. Lakshmanan, J. Han, and A. Pang. Exploratory mining and pruning optimizations of constrained associations rules. In *Proc. of ACM SIGMOD Conference*, 1998.
- [69] D. Papadias, Y. Tao, G. Fu, and B. Seeger. Progressive skyline computation in database systems. In *ACM TODS*, volume 30, 2005.
- [70] J. Pei, A. W.-C. Fu, X. Lin, and H. Wang. Computing compressed multidimensional skyline cubes efficiently. In *Proc. of IEEE ICDE Conference*, 2007.
- [71] J. Pei and J. Han. Can we push more constraints into frequent pattern mining? In *Proc. of ACM SIGKDD Conference*, 2000.
- [72] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M. Hsu. Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *Proc. of IEEE ICDE Conference*, pages 215–314, 2001.

- [73] J. Pei, B. Jiang, X. Lin, and Y. Yuan. Probabilistic skylines on uncertain data. In *Proc. of VLDB Conference*, pages 15–26, 2007.
- [74] J. Pei, W. Jin, M. Ester, and Y. Tao. Catching the best views of skyline: A semantic approach based on decisive subspaces. In *Proc. of VLDB Conference*, 2005.
- [75] Y. Peng, R. C.-W. Wong, and Q. Wan. Finding top-k preferable products. In *IEEE TKDE (Preprint)*, 2012.
- [76] Y. Peng, R. C.-W. Wong, L. Ye, and P. Yu. Attribute-based subsequence matching and mining. In *Proc. of IEEE ICDE Conference*, 2012.
- [77] A. Pol and T. Kahveci. Highly scalable and accurate seeds for subsequence alignment. In *Proc. of IEEE BIBE Conference*, 2005.
- [78] F. F. Reichheld. Learning from customer defections. *Harvard Business Review*, September 1996.
- [79] D. Sacharidis, S. Papadopoulos, and D. Papadias. Topologically-sorted skylines for partially-ordered domains. In *Proc. of IEEE ICDE Conference*, 2009.
- [80] R. She, F. Chen, K. Wang, M. Ester, J. L. Gardy, and F. S. L. Brinkman. Frequent-subsequence-based prediction of outer membrane proteins. In *Proc. of ACM SIGKDD Conference*, 2003.
- [81] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. In *Journal of Molecular Biology*, 1981.
- [82] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *Proc. of EDBT Conference*, 1996.
- [83] I. Stanoi, M. Riedewald, D. Agrawal, and A. E. Abbadi. Discovery of influence sets in frequently updated databases. In *Proc. of VLDB Conference*, 2001.

- [84] G. D. Stormo. DNA binding sites: representation and discovery. In *Bioinformatics*, 2000.
- [85] D. Sun, S. Wu, J. Li, and A. K. Tung. Skyline-join in distributed databases. In *IEEE ICDE Workshop*, 2008.
- [86] K.-L. Tan, P.-K. Eng, and B. C. Ooi. Efficient progressive skyline computation. In *Proc. of VLDB Conference*, pages 301–310, 2001.
- [87] Y. Tao, L. Ding, X. Lin, and J. Pei. Distance-based representative skyline. In *Proc. of IEEE ICDE Conference*, pages 892–903, 2009.
- [88] Y. Tao and D. Papadias. Maintaining sliding window skylines on data streams. *IEEE TKDE*, 18(3):377–391, 2006.
- [89] Y. Tao, D. Papadias, and J. Zhang. Aggregate processing of planar points. In *Proc. of EDBT Conference*, pages 682–700, 2002.
- [90] Q. Wan, R. C.-W. Wong, I. F. Ilyas, M. T. Özsu, and Y. Peng. Creating competitive products. *Proc. of VLDB Conference*, 2(1):898–909, August 2009.
- [91] Q. Wan, R. C.-W. Wong, and Y. Peng. Finding top-k profitable products. In *Proc. of IEEE ICDE Conference*, pages 1055–1066, 2011.
- [92] Q. Wan, R. C.-W. Wong, and Y. Peng. Finding top-k profitable products (technical report). In <http://www.cse.ust.hk/~raywong/paper/createTopKProfitableProduct-technical.pdf>, 2011.
- [93] J. Wang and J. Han. Bide: Efficient mining of frequent closed sequences. In *Proc. of IEEE ICDE Conference*, pages 79–90, 2004.

- [94] R. C.-W. Wong, A. W.-C. Fu, J. Pei, Y. S. Ho, T. Wong, and Y. Liu. Efficient skyline querying with variable user preferences on nominal attributes. In *Proc. of VLDB Conference*, 2008.
- [95] R. C.-W. Wong, J. Pei, A. W.-C. Fu, and K. Wang. Mining favorable facets. In *Proc. of ACM SIGKDD Conference*, 2007.
- [96] C. Wu, M. Berry, S. Shivakumar, and J. McLarty. Neural networks for full-scale protein sequence classification: Sequence encoding with singular value decomposition. In *Machine Learning*, 1995.
- [97] T. Xia and D. Zhang. Refreshing the sky: The compressed skycube with efficient support for frequent updates. In *Proc. of ACM SIGMOD Conference*, 2006.
- [98] T. Xia, D. Zhang, E. Kanoulas, and Y. Du. On computing top- $k$  most influential spatial sites. In *Proc. of VLDB Conference*, 2005.
- [99] C. Yang and K.-I. Lin. An index structure for improving nearest closest pairs and related join queries in spatial databases. In *Proc. of IEEE IDEAS Conference*, 2002.
- [100] Y. Yuan, X. Lin, Q. Liu, W. Wang, J. X. Yu, and Q. Zhang. Efficient computation of the skyline cube. In *Proc. of VLDB Conference*, 2005.
- [101] M. J. Zaki. Spade: An efficient algorithm for mining frequent sequences. In *Journal of Machine Learning*, volume 42, pages 31–60, January 2001.
- [102] M. Zhang, B. Kao, D. Cheung, and K. Yip. Mining periodic patterns with gap requirement from sequences. In *Proc. of ACM SIGMOD Conference*, 2005.
- [103] W. Zhang, X. Lin, Y. Zhang, W. Wang, and J. X. Yu. Probabilistic skyline operator over sliding windows. In *Proc. of IEEE ICDE Conference*, pages 1060–



1071, 2009.

- [104] Z. Zhang, L. Lakshmanan, and A. K. Tung. On domination game analysis for microeconomic data mining. In *TKDD*, 2009.

## APPENDIX A

### PROOFS OF LEMMAS/THEOREMS

**Proof of Lemma 3.3.1:** We prove it by contradiction. Suppose  $s$  is in form of “ $u_1, u_2, \dots, u_l$ ”. Assume that  $(V_s \bmod P(v_i)) = 0$  for each  $i \in [1, k]$ , but there exists an integer  $j \in [1, k]$  such that  $v_j$  in  $q$  that does not match any values in  $s$ . Since  $v_j$  does not match any values in  $s$ , we deduce that  $v_j \notin \alpha(u_x)$  for each  $x \in [1, l]$ . Note that  $V_s = \prod_{i=1}^l \prod_{u \in \alpha(u_i)} P(u)$ .  $V_s$  is a product of the labels of all values in  $\cup_{x=1}^l \alpha(u_x)$ . Since  $v_j \notin \alpha(u_x)$ , we deduce that  $(V_s \bmod P(v_j)) \neq 0$ . It contradicts to our premise that  $V_s \bmod P(v_i) = 0$  for any  $i \in [1, k]$ .  $\square$

**Proof of Lemma 3.3.2:** 1) If the query-aware lifespan of  $s$  with respect to  $q$  is non-overlapping, then we can select an arbitrary integer  $p_i$  inside  $\triangle_i$ , for each  $i \in [1, k]$ . In other words, for each  $i \in [1, k]$ , we can find that  $v_i$  matches the value in  $s$  at the temporal position equal to  $p_i$ . Thus,  $q$  matches  $s$ . 2) We prove by contradiction. Suppose that  $q$  matches  $s$ . If the query-aware lifespan of  $s$  with respect to  $q$  is invalid, then there exist two integers  $i, j \in [1, k]$  such that  $i < j$  and  $\triangle_i$  appears after  $\triangle_j$ . Since  $\triangle_i$  appears after  $\triangle_j$ , we deduce that the smallest temporal position of a value in  $s$  which is matched by  $v_i$  is larger than the largest temporal position of a value in  $s$  which is matched by  $v_j$ . We derive that  $v_i$  appears after  $v_j$  in  $q$ . We conclude that  $i > j$ . This leads to a contradiction that  $i < j$ .  $\square$

**Proof of Lemma 3.3.3:** By definition of  $L_s$  and  $U_s$ ,  $l = (L_s \bmod P(v)) = LS_{v,s}.l$  and  $u = (U_s \bmod P(v)) = LS_{v,s}.u$ . So, the lifespan of  $v$  in  $s$  is equal to  $(l, u)$ .  $\square$

**Proof of Lemma 4.2.1:** If  $p$  dominates  $q_i$ , and  $p \notin \gamma(X, q_i)$ , then  $p$  does not quasi-dominate  $q_i$ . So, there exists at least one attribute on which  $q_i$  is better than  $p$  according

to the quasi-dominance definition. This conflicts with our premise that  $p$  dominates  $q_i$ .

□

**Proof of Lemma 4.2.2:** If  $p$  quasi-dominates  $p'$ , according to the definition of  $f$ , this indicates that (1)  $p$  dominates  $p'$  with respect to the first  $l - 1$  attributes, namely  $A_1, A_2, \dots, A_{l-1}$ , or (2)  $p$  has the same  $l - 1$  attribute values as  $p'$ . In case 1, obviously,  $f(p) < f(p')$ . In case 2,  $f(p) = f(p')$ . Thus,  $f(p) \leq f(p')$  □

**Proof of Lemma 4.2.3:** Suppose that we have a better assignment vector  $\mathbf{v}'$  of  $Q_i$  in form of  $(v'_1, v'_2, \dots, v'_n)$  such that  $Profit(Q_i, \mathbf{v}') > Profit(Q_i, \mathbf{v}_i)$ .

Consider two cases. *Case 1:*  $v'_i \leq v_i$ . In this case,  $v'_i - q_i \cdot C \leq v_i - q_i \cdot C$ . In other words,  $\Delta(q_i, v'_i) \leq \Delta(q_i, v_i)$ . In addition, we know that  $\mathbf{v}_{i-1}$  is the optimal price assignment vector of  $Q_{i-1}$ , which means  $Profit(Q_{i-1}, \mathbf{v}') \leq Profit(Q_{i-1}, \mathbf{v}_{i-1})$ . Thus, we have

$$\begin{aligned} Profit(Q_i, \mathbf{v}') &= Profit(Q_{i-1}, \mathbf{v}') + \Delta(q_i, v'_i) \\ &\leq Profit(Q_{i-1}, \mathbf{v}_{i-1}) + \Delta(q_i, v_i) \\ &= Profit(Q_i, \mathbf{v}_i) \end{aligned}$$

Thus,  $Profit(Q_i, \mathbf{v}') \leq Profit(Q_i, \mathbf{v}_i)$ . This leads to a contradiction.

*Case 2:*  $v'_i > v_i$ . It is easy to verify that  $\mathbf{v}'$  is not a feasible price assignment vector.

So, we do not include the details. □

**Proof of Theorem 4.2.1:** According to Lemma 4.2.3, in each call of Algorithm 10 in Algorithm 9, the optimal price assignment vector  $\mathbf{v}$  of  $Q''$  is returned. Thus, when iteration step ends, that is  $Q'' = Q'$ , the returned  $\mathbf{v}$  is the optimal price assignment vector of  $Q'$ . □

**Proof of Theorem 4.2.2:** We prove it by contradiction. Let  $\mathbf{v}'$  be the optimal assignment vector and  $\mathbf{v}' \neq \mathbf{v}$ . Without loss of generality, assume  $q_i$  is the first tuple

that is selected in the optimal solution but not selected in  $Q'$  returned by Algorithm Dynamic Programming. That is  $\mathbf{v}'_i > 0$  and  $\mathbf{v}_i = 0$ . Since  $\mathbf{v}'_i > 0$ ,  $T(i, k) > T(i - 1, k - 1) + \alpha(q_i, S(i - 1, k - 1), \mathbf{v}(i - 1, k - 1))$ . However, since  $\mathbf{v}_i = 0$ ,  $T(i, k) < T(i - 1, k - 1) + \alpha(q_i, S(i - 1, k - 1), \mathbf{v}(i - 1, k - 1))$ . Therefore, it is a contradiction. Therefore,  $Q'$  must contain all the tuples of the optimal solution. Thus, the solution returned by Algorithm Dynamic Programming is the optimal solution.  $\square$

**Proof of Theorem 4.2.3:** Note that problem TPP is a maximization problem. In order to show the NP-hardness of the problem we are studying, we first give a decision problem for problem TPP called *DTPP*: Given a non-negative real number  $X$ , does there exist a set  $Q'$  of  $k$  tuples from  $Q$  such that  $Profit_o(Q') \geq X$ ?

The NP-hardness proof can be achieved by transforming an NP-complete problem, the  $d$ -coverage problem [92], to the DTPP problem.

**$d$ -Coverage Problem:** Given a set  $U$  of elements, a collection  $J$  of sets containing elements in  $U$ , a positive integer  $d$  and a positive integer  $t$  where  $1 < d \leq |J|$  and  $t \leq |U|$ , does there exist a subset  $I \subseteq J$  such that  $|I| = d$  and  $|\cup_{C \in I} C| \leq t$ ?

Given an instance of the  $d$ -coverage problem. We want to construct an instance of the DTPP problem from the above instance. We define two positive real numbers,  $M$  and  $m$ , where  $M \gg m$  and  $\frac{M}{2} > m$ . We construct the instance as follows. We set  $\sigma = \frac{M}{2}$ ,  $k = d + |U|$ ,  $l = |U| + |J| + 1$  and  $X = d \cdot m + |U| \cdot M - t \cdot \frac{M}{2}$ . Next, we set  $P$  and  $Q$  according to  $U$  and  $J$ . Specifically, we set  $P$  to be the set containing only one tuple  $p$ , and  $Q$  to be the set containing  $|U| + |J|$  tuples.

Note that tuple  $p$  is associated with  $l$  attributes, namely  $A_1, A_2, \dots, A_l$ , where the last attribute  $A_l$  corresponds to attribute Price. All  $l$  attribute values of  $p$  are to be set. Besides, each tuple in  $Q$  is also associated with the same  $l$  attributes together with an additional cost attribute  $C$  where only the first  $l - 1$  attribute values and the cost attribute value of each tuple in  $Q$  is to be set for the problem instance construction.

Now, we describe in detail how we generate  $|U| + |J|$  tuples in  $Q$ . Initially,  $Q$  is an empty set. Then, for each set  $S$  in  $J$ , we create a tuple  $q_S$  and insert it into  $Q$ . There are  $|J|$  tuples generated from  $J$  and these tuples are called *type I tuples*. Each of these tuples is arbitrarily given a unique *label* which is a positive integer  $\in [1, |J|]$ . For each element  $e$  in  $U$ , we create a tuple  $q_e$  and insert it into  $Q$ . There are  $|U|$  tuples generated from  $U$  and these tuples are called *type II tuples*. Similarly, each of these tuples is arbitrarily given a unique label which is a positive integer  $\in [|J| + 1, |U| + |J|]$ . For each set  $S$  in  $J$  and each element  $e$  in  $S$ , we say that tuple  $q_S$  (of type I) is a *parent* of tuple  $q_e$  (of type II).

Next, we set the attribute values of the tuples in  $P$  and  $Q$ . Consider tuple  $p$  in  $P$ . For each  $j \in [1, l - 1]$ , we set  $p.A_j$  to  $m$ . We set  $p.A_l$  to  $M + m + \sigma$ . Consider tuples in  $Q$ . For each  $q_i \in Q$ , we set  $q_i.C$  to  $M$  if  $q_i$  is a type I tuple and set it to  $m$  if  $q_i$  is a type II tuple. Then, we set the first  $l - 1$  attribute values with the following three major steps. For Step 1, we initialize a variable  $i$  to 1. Let the tuple with label equal to  $i$  in  $Q$  be  $q_i$ . We set  $q_i.A_i$  to any real number  $b$  such that  $m < b < M$ . Note that  $q_i.A_j$  is not set in this step and will be set in later steps for  $j \in [i + 1, l - 1]$ . For Step 2, we increment the variable  $i$  by 1. Tuple  $q_i$  is another tuple with label equal to an updated value  $i$ . We execute three sub-steps. *Step 2a*: We set  $q_i.A_j$  to be  $M$  for  $j \in [1, i - 1]$ . *Step 2b*: We set  $q_i.A_i$  to be any real number  $b$  such that  $m < b < M$ . Similarly, note that  $q_i.A_j$  is not set and will be set in later steps for  $j \in [i + 1, l - 1]$ . *Step 2c*: For  $y \in [1, i - 1]$ , we set  $q_y.A_i$  to  $m$  if  $q_y$  is a parent of  $q_i$ , and we set it to  $M$  otherwise. *Step 3*: We repeat Step 2 until  $i$  is equal to  $|U| + |J|$ . After this construction of  $Q$ , we know that a tuple  $q'$  is a parent of another tuple  $q$  if and only if  $q'$  quasi-dominates  $q$ .

We have just defined the constructed problem instance for DTPP. Next, we analyze some properties from the final solution of this constructed problem instance. Note that in the final solution of DTPP, we have to set the attribute  $A_l$  value of each tuple  $q$  in

$Q'$ . In the following, we determine how to set the  $A_l$  value of each tuple in  $Q'$ .

It is easy to verify that for each tuple  $q \in Q$ ,  $q$  is quasi-dominated by tuple  $p$ . Now, we consider how to set the  $A_l$  value of a tuple  $q$  in  $Q'$ . Consider two cases: *Case 1*:  $q$  is a type I tuple.  $q.A_l$  must be set to  $M + m$  so that the profit of  $q$  is the greatest and  $q$  is in the skyline with respect to  $P \cup Q'$  (no matter what  $Q'$  is). This is because if  $q.A_l$  is set to a value greater than  $M + m$ , then  $q$  is dominated by  $p$ . Besides,  $q$  is not dominated by any other tuples in  $Q'$  no matter what the  $A_l$  values of these tuples in  $Q'$  are set. In Case 1, the profit of  $q$  is equal to  $(M + m) - M = m$ . *Case 2*:  $q$  is a type II tuple.  $q.A_l$  is set to different values according to two different sub-cases. *Case 2a*: There does not exist any type I tuple in  $Q'$  which quasi-dominates tuple  $q$ . This case is similar to Case 1.  $q.A_l$  must be set to  $M + m$  so that the profit of  $q$  is the greatest and  $q$  is in the skyline with respect to  $P \cup Q'$  (no matter what  $Q'$  is). In Case 2a, the profit of  $q$  is equal to  $(M + m) - m = M$ . *Case 2b*: There exists a type I tuple  $q'$  in  $Q'$  which quasi-dominates tuple  $q$ . Note that  $q'$  must be a parent of  $q$ . Besides, according to Lemma 4.2.1, since  $q'$  quasi-dominates  $q$ , the value of  $q'.A_l$  can be determined before  $q.A_l$  is to be set. According to Case 1,  $q'.A_l$  is set to  $M + m$ . On the other hand,  $q.A_l$  must be set to  $M + m - \sigma$  so that the profit of  $q$  is the greatest and  $q$  is in the skyline with respect to  $P \cup Q'$  (no matter what  $Q'$  is). This is because if  $q.A_l$  is set to a value greater than  $M + m - \sigma$ , says  $M + m$  (in Case 2a), then  $q$  is dominated by  $q'$ . Besides,  $q$  is not dominated by any other tuples in  $Q'$  no matter what the  $A_l$  values of the tuples in  $Q'$  other than  $q'$  are set. In Case 2b, the profit of  $q$  is equal to  $(M + m - \sigma) - m = M - \sigma = \frac{M}{2}$ .

According to the above strategy to set the  $A_l$  value of each tuple in  $Q'$  and  $\frac{M}{2} > m$ , we conclude that the profit of each type II tuple in  $Q'$  is greater than the profit of each type I tuple in  $Q'$  (no matter what  $Q'$  is). Since  $k = d + |U|$  and  $k$  is the total number of tuples in  $Q'$ , the final selection set  $Q'$  must contain all  $|U|$  type II tuples and

exactly  $d$  type I tuples. The total profit of all the type I tuples in  $Q'$  is equal to  $d \cdot m$ . Let  $f$  be the total number of type II tuples in  $Q'$  each of which is quasi-dominated by at least one type I tuple in  $Q'$ . The total profit of all  $|U|$  type II tuples is equal to  $(|U| - f) \cdot M + f \cdot \frac{M}{2} = |U| \cdot M - f \cdot \frac{M}{2}$ . Thus, the profit of  $Q'$ , denoted by  $Profit_o(Q')$ , is  $d \cdot m + |U| \cdot M - f \cdot \frac{M}{2}$ .

It is easy to verify that there exists a set  $Q'$  of  $k$  tuples from  $Q$  such that  $Profit_o(Q') \geq d \cdot m + |U| \cdot M - t \cdot \frac{M}{2}$  (and thus  $f \leq t$ ) if and only if there exists a set  $I \subseteq J$  such that  $|I| = d$  and  $|\cup_{C \in I} C| \leq t$ . Since the  $d$ -coverage problem is NP-complete, the DTPP problem is NP-hard.  $\square$

**Proof of Theorem 4.2.4:** Let  $P$  be the set of existing tuples, and  $Q$  be the set of the newly created tuples. Let  $O$  be the optimal selection and  $Q'$  be the selection returned by Algorithm 12. Given  $Q' \subseteq Q$  and  $q_i \in Q'$ , we define  $\Delta_o(q_i, Q') = \Delta(q_i, v_i)$  where  $v_i$  is the  $i$ -th entry of the optimal price assignment vector  $\mathbf{v}$  of  $Q'$ . Suppose  $O = \{o_1, o_2, \dots, o_k\}$  and  $Q' = \{q_1, q_2, \dots, q_k\}$ , where  $o_i$  and  $q_i$  are sorted in ascending order of the  $f$  values described in Section 4.2.3.

Let  $n_i$  be the greatest possible number of tuples in  $Q'$  quasi-dominating  $q_i$ . It is easy to verify that  $\Delta_o(q_i, Q') \geq SP(q_i) - n_i \sigma$ . Note that  $\sum_{i=1}^k n_i \leq \frac{k(k-1)}{2}$ . We derive that

$$\begin{aligned} \sum_{q_i \in Q'} \Delta_o(q_i, Q') &\geq \sum_{q_i \in Q'} (SP(q_i) - n_i \sigma) \\ &= \sum_{q_i \in Q'} SP(q_i) - \sum_{q_i \in Q'} n_i \sigma \\ &\geq \sum_{q_i \in Q'} SP(q_i) - \frac{k(k-1)}{2} \sigma \end{aligned}$$

Note that

$$\begin{aligned}
\sum_{o_i \in O} \Delta_o(o_i, O) &\leq \sum_{o_i \in O} SP(o_i) \\
&\leq \sum_{q_i \in Q'} SP(q_i)
\end{aligned}$$

Therefore, we have

$$\begin{aligned}
Profit_o(O) - \epsilon_{add} &= \sum_{o_i \in O} \Delta_o(o_i, O) - \frac{k(k-1)}{2}\sigma \\
&\leq \sum_{q_i \in Q'} SP(q_i) - \frac{k(k-1)}{2}\sigma \\
&\leq \sum_{q_i \in Q'} \Delta_o(q_i, Q') \\
&= Profit_o(Q')
\end{aligned}$$

□

**Proof of Theorem 4.2.5:** According to Theorem 4.2.4, we have that  $\Delta - \frac{k(k-1)}{2}\sigma \leq Profit_o(Q')$ . By the nature of optimality, we also have  $Profit_o(Q') \leq Profit_o(O)$ . For any tuple  $q \in Q'$ , the real profit cannot be larger than the standalone profit. Thus,  $\Delta - \frac{k(k-1)}{2}\sigma \leq Profit_o(Q') \leq Profit_o(O) \leq \Delta$ .

Therefore, if  $Profit_o(Q') > 0$ , we have

$$\begin{aligned}
\frac{Profit_o(Q')}{Profit_o(O)} &\geq \frac{\Delta - \frac{k(k-1)}{2}\sigma}{\Delta} \\
&= 1 - \frac{k(k-1)\sigma}{2\Delta}
\end{aligned}$$

Therefore,

$$Profit_o(Q') \geq (1 - \epsilon_{mult})Profit_o(O)$$



□

**Proof of Lemma 4.3.1:** Since  $p_{new} \notin SKY(P_{new})$ , we have  $SKY(P_{new}) = SKY(P)$ .

It is easy to verify  $Q'_{new} = Q'$ . □

**Proof of Lemma 4.3.2:** Suppose  $Q'_{new} = Q'$ . Since  $Q'_{new}$  is the answer of the problem, each tuple in  $Q'_{new}$  is in  $SKY(P_{new} \cup Q'_{new})$ . Since  $Q'_{new} = Q'$ , we deduce that each tuple in  $Q'$  is in  $SKY(P_{new} \cup Q')$ . Note that  $p_{new} \in P_{new}$ . We conclude that there does not exist  $q \in Q'$  such that  $p_{new}$  dominates  $q$ .

Suppose that there does not exist  $q \in Q'$  such that  $p_{new}$  dominates  $q$ . We want to show that  $Q'_{new} = Q'$  by dividing the proof into three parts.

Firstly, we show that each tuple  $q$  in  $Q'$  is in  $SKY(P_{new} \cup Q')$ . Before  $p_{new}$  is inserted into  $P$ , we know that each tuple  $q$  in  $Q'$  is in  $SKY(P \cup Q')$ . Consider a tuple  $q$  in  $Q'$ . We know that no tuples in  $P \cup Q'$  dominate  $q$ . Since there does not exist  $q \in Q'$  such that  $p_{new}$  dominates  $q$ , we deduce that no tuples in  $P \cup Q' \cup \{p_{new}\} (= P_{new} \cup Q')$  dominate  $q$ . We conclude that  $q$  is in  $SKY(P_{new} \cup Q')$ .

Secondly, we show that  $Q'$  is the set of  $k$  tuples from  $Q$  such that  $Profit_o(Q') = \max_{Q'' \in \mathcal{Q}} Profit_o(Q'')$  where  $\mathcal{Q}$  is the set of all possible subsets containing  $k$  tuples from  $Q$ . Let  $Profit_o(Q', P)$  be the optimal profit of  $Q'$  based on datasets  $Q$  and  $P$ . Thus,  $Profit_o(Q') = Profit_o(Q', P)$  before  $p_{new}$  is inserted into  $P$  while  $Profit_o(Q') = Profit_o(Q', P \cup \{p_{new}\})$  after  $p_{new}$  is inserted into  $P$ . Before  $p_{new}$  is inserted into  $P$ , we know that for each  $Q'' \in \mathcal{Q}$ , we have

$$Profit_o(Q', P) \geq Profit_o(Q'', P) \quad (\text{A.1})$$

Since there does not exist  $q \in Q'$  such that  $p_{new}$  dominates  $q$ , we deduce that

$$Profit_o(Q', P \cup \{p_{new}\}) = Profit_o(Q', P) \quad (\text{A.2})$$

Besides, for each  $Q'' \in \mathcal{Q}$ , we know that

$$Profit_o(Q'', P) \geq Profit_o(Q'', P \cup \{p_{new}\}) \quad (\text{A.3})$$

From (A.1), (A.2) and (A.3), we conclude that for each  $Q'' \in \mathcal{Q}$ ,  $Profit_o(Q', P_{new}) \geq Profit_o(Q'', P_{new})$ .

Lastly, since each tuple  $q$  in  $Q'$  is in  $SKY(P_{new} \cup Q')$  and  $Q'$  is the set of  $k$  tuples from  $Q$  such that  $Profit_o(Q') = \max_{Q'' \in \mathcal{Q}} Profit_o(Q'')$ , we conclude that  $Q'_{new} = Q'$ .  $\square$

**Proof of Theorem 4.3.1:** Let  $SP_{before}(q)$  be the standalone profit of tuple  $q \in Q$  calculated before  $p_{new}$  is inserted into  $P$ . Let  $SP_{after}(q)$  be the standalone profit of tuple  $q \in Q$  calculated after  $p_{new}$  is inserted into  $P$ . It is easy to verify that for each  $q \in Q$ ,  $SP_{after}(q)$  is smaller than or equal to  $SP_{before}(q)$ . Consider three cases. *Case 1:*  $p_{new} \notin SKY(P_{new})$ . We know that for each  $q \in Q$ ,  $SP_{before}(q) = SP_{after}(q)$ . Thus, the  $k$  tuples in  $Q$  which have the greatest standalone price after  $p_{new}$  is inserted are exactly the same as the tuples in  $Q'$ . Thus, since  $Q'_{new} = Q'$ , the selection set returned by Algorithm 14 is the selection set returned by Greedy Algorithm (Version 1). *Case 2:* There does not exist  $q \in Q'$  such that  $p_{new}$  dominates  $q$ . By using the techniques used in the proof of Lemma 4.3.2, we also derive that the  $k$  tuples in  $Q$  which have the greatest standalone price after  $p_{new}$  is inserted are exactly the same as the tuples in  $Q'$  (because for each  $q \in Q$ ,  $SP_{after}(q) \leq SP_{before}(q)$  and for each  $q \in Q'$ ,  $SP_{after}(q) = SP_{before}(q)$ ). *Case 3:*  $p_{new} \in SKY(P_{new})$  and there exists  $q \in Q'$  such that  $p_{new}$  dominates  $q$ . All the tuples in  $Q$  which standalone prices are changed are updated in Lines 10-11. The output of Algorithm 14 (i.e., the  $k$  tuples in  $Q$  which have the greatest (updated) standalone price (Line 13)) are the selection set returned by Greedy1.  $\square$

**Proof of Lemma 4.3.3:** The proof is similar to that of Lemma 4.3.1. Similar tech-

niques can be used in the proof of this lemma.  $\square$

**Proof of Theorem 4.3.2:** The proof is similar to that of Theorem 4.3.1. Similar techniques can be used in the proof of this theorem.  $\square$

**Proof of Theorem 4.4.1:** Note that Problem 3 is a maximization problem. We give a decision problem: Given a non-negative real number  $X$ , does there exist a set  $Q'$  of  $k$  tuples from  $Q$  such that (1)  $IV(Q') \geq X$  and (2) each tuple in  $Q'$  is in the skyline with respect to  $P \cup Q'$ .

The NP-hardness proof can be achieved by transforming an NP-complete problem, the *maximum coverage* problem, to our decision problem.

**Maximum Coverage:** Given a positive integer  $d$ , another positive integer  $t$ , a set  $U$  of elements and a collection  $J$  of sets each of which is a subset of  $U$ , does there exist a set  $I \subseteq J$  such that  $|I| \leq d$  and  $|\cup_{s \in I} s| \geq t$ ?

Given an instance of the maximum coverage problem. We want to construct an instance of our decision problem from the above instance. Recall that in the proof of Theorem 4.2.3, we construct an instance of the DTPP problem from a given instance of the  $d$ -coverage problem, by creating tuples of type I and type II (for the tuples in  $Q$ ). Note that these tuples are created with their attribute values for attribute  $A_i$  where  $i = 1, 2, \dots, l - 1$  and their attribute value for attribute Cost  $C$ . In this proof, we construct tuples for  $Q$  and customer preferences for  $CP$  similarly. Let  $M$  be a very large positive real number. For each set  $S$  in  $J$ , we create a type I tuple  $u$  and then create a customer preference  $cp_S$  in form of  $\{g_1, g_2, \dots, g_l\}$  by setting  $g_i$  to be  $u.A_i$  for  $i \in [1, l - 1]$  and setting  $g_l$  to  $M$ . All customer preferences generated form set  $CP$ . For each customer preference  $cp$  in  $CP$ , we set  $w(cp)$  to 1. For each element  $e$  in  $U$ ,

we create a type II tuple  $u$  and create a tuple  $q$  which is exactly equal to  $u$ . All tuples generated from  $U$  form set  $Q$ . We set  $P, k$  and  $X$  to  $\emptyset, d$  and  $t$ , respectively.

It is easy to see that this transformation can be constructed in polynomial time. It is also easy to verify that when the problem is solved in the transformed decision problem, the original maximum coverage problem is also solved. Since the maximum coverage problem is an NP-complete problem, our decision problem is NP-hard.  $\square$

**Proof of Lemma 4.4.1:** We prove by contradiction. Suppose there exists a tuple  $q$  in  $Q'$  which is not in the skyline with respect to  $P \cup Q'$ . This means that there exists a tuple in  $P \cup Q'$  which dominates  $q$ . Consider two cases.

*Case 1:* The tuple dominating  $q$  is a tuple from  $P$ . Since  $PS(q) = \{v | v \geq q.C \text{ and } q \in SKY(P \cup \{q\}) \text{ if we set } q.A_l = v\}$ , we know that  $q \in SKY(P \cup \{q\})$ . Thus, there does not exist any tuple in  $P$  dominating  $q$ . This leads to a contradiction.

*Case 2:* The tuple in  $P \cup Q'$  dominating  $q$  is a tuple in  $Q'$  other than  $q$ . Let this tuple be  $q'$ . Note that  $q'$  dominates  $q$ . We have  $q'.A_i \leq q.A_i$  for each  $i \in [1, l]$ . Thus,  $IS(q) \subseteq IS(q')$ . Let  $Q_i$  be the selection set  $Q'$  maintained by Algorithm 16 at the end of the  $i$ -th iteration for  $i = 1, 2, \dots, k$ . We define  $Q_0 = \emptyset$ . Since  $IS(q) \subseteq IS(q')$ , we deduce that  $IV(Q_i \cup \{q\}) \leq IV(Q_i \cup \{q'\})$  for each  $i \in [1, k]$ . We further consider two sub-cases. *Case 2(a):*  $IV(Q_i \cup \{q\}) < IV(Q_i \cup \{q'\})$ . We deduce that  $q'$  is selected and inserted into the selection set maintained by Algorithm 16 before  $q$  is selected and inserted. Consider the iteration of selecting  $q'$ , says the  $j$ -th iteration. We have  $Q_j = Q_{j-1} \cup \{q'\}$ . We conclude that for the  $l$ -th iteration where  $l \in [j + 1, k]$ ,  $IV(Q_l \cup \{q\}) = 0$ . This leads to the contradiction that  $IV(Q_l \cup \{q\}) > 0$  for each  $l \in [1, k]$  (This is because if  $k \leq k_{max}$ , we have  $IV(Q_l \cup \{q\}) > 0$  for each  $l \in [1, k]$ ). *Case 2(b):*  $IV(Q_i \cup \{q\}) = IV(Q_i \cup \{q'\})$ . Since  $q'$  dominates  $q$ , we deduce that  $f(q') < f(q)$ . Thus,  $q'$  is selected and inserted into the selection set maintained by Algorithm 16 before  $q$  is selected and inserted. We have a similar conclusion as Case

2(a). □

**Proof of Theorem 4.4.2:** We can transform our problem to the maximum coverage problem by mapping each customer preference and the influence set of each tuple in  $Q$  for our problem to an element and a set of elements in the maximum coverage problem, respectively. Note that the greedy algorithm for the maximum coverage problem which chooses the set containing the largest number of *uncovered* elements is 0.63-approximate [46]. Since Algorithm 16 follows the same framework as the above greedy algorithm, it is 0.63-approximate. □

**Proof of Lemma 5.3.1:** We prove by contradiction. Suppose  $h_f.p$  is set to a non-negative value such that  $h_f$  is not in the skyline  $SKY(H \cup \{h_f\})$ . That is, there exists a service-site  $h_i$  dominating  $h_f$ .

Since  $h_i$  dominates  $h_f$ , we deduce that, for all attribute  $X \in \mathcal{X}$ ,  $h_i.X \leq h_f.X$  and there exists an attribute  $X' \in \mathcal{X}$  such that  $h_i.X' < h_f.X'$ .

Consider that we draw a perpendicular bisector of a line segment joining  $h_i$  and  $h_f$ . Figure 5.4(b) shows an example that (1)  $h_f$  is inside  $CH(A)$  and (2) there exists a service-site  $h_i$  and an attraction-site  $a_j$  where  $d(h_i, a_j) < d(h_f, a_j)$ . The dashed line corresponds to the perpendicular bisector of a line segment joining  $h_i$  and  $h_f$ .

The bisector cuts the Euclidean space into two sides: one side  $S_f$  containing  $h_f$  and the opposite side  $\bar{S}_f$  not containing  $h_f$ . It is easy to verify that, for each attraction-site  $a$  which is inside  $S_f$ ,

$$d(h_f, a) < d(h_i, a) \tag{A.4}$$

Since  $h_f$  is inside  $CH(A)$ , we deduce that  $S_f$  contains a service-site  $a_k$  other than  $a_j$  in the convex hull of  $A$ . Since  $a_k$  is inside  $S_f$ , by Inequality (A.4), we deduce that  $d(h_f, a_k) < d(h_i, a_k)$ . That is,  $h_f.X_k < h_i.X_k$ , which leads to a contradiction that, for

all attribute  $X \in \mathcal{X}$ ,  $h_i.X \leq h_f.X$ . □

**Proof of Lemma 5.3.2:** Note that  $h_i$  is nearer to all attraction-sites compared with  $h_f$  if and only if  $h_i$  is inside  $\mathcal{I}$ . With notation  $\mathcal{I}$ , Lemma 5.3.2 can be re-written as follows: “ $h_i$  dominates  $h_f$  if and only if (1)  $h_i$  is inside  $\mathcal{I}$  and (2)  $h_f.p \geq h_i.p$ ”. If we set  $p_{max,s}$  to be  $\min_{h \in \mathcal{I}} h.p$ , since  $h_f.p < p_{max,s}$ , it is easy to verify that  $h_f$  is in the skyline  $SKY(H \cup \{h_f\})$ . □

**Proof of Lemma 5.3.3:** Note that  $h_i$  is farther from all attraction-sites compared with  $h_f$  if and only if  $h_i$  is not inside  $\mathcal{U}$ . With notation  $\mathcal{U}$ , we re-write Lemma 5.3.3 as follows “ $h_f$  dominates  $h_i$  if and only if (1)  $h_i$  is not inside  $\mathcal{U}$  and (2)  $h_f.p \leq h_i.p$ .” If we set  $p_{max,d}$  to be the  $K$ -th greatest price (i.e.,  $h.p$ ) among all service-sites  $h$  not in  $\mathcal{U}$ , since  $h_f.p < p_{max,d}$ , it is easy to verify that  $h_f$  dominates at least  $K$  service-sites. □

**Proof of Theorem 5.3.1:** According to the proofs of Lemma 5.3.1, Lemma 5.3.2 and Lemma 5.3.3, Algorithm 17 returns the optimal solution. □

**Proof of Theorem 5.3.2:** By using the methods used in [41] and [29], Algorithm 17 takes  $O(n + n \log n + n(J + \log m) + m \log m) = O(n \log n + n(J + \log m) + m \log m)$  time. □

**Proof of Lemma 5.3.4:** According to the definition of  $N.minp$ , any leaf node that is a descendant of  $N$  must have a price no less than  $N.minp$ . Without loss of generality, suppose  $t$  is the service-site corresponding to a random selected leaf node which is a descendant of  $N$ . If the current value  $p'_{max,s}$  is equal or smaller than  $N.minp$ , then  $p'_{max,s} \leq t.p$ . As  $p'_{max,s}$  is the current best value of  $p_{max,s}$ , there must exist a service-site  $\hat{h}$  in  $\mathcal{I}$  that  $p'_{max,s} = \hat{h}.p$ . Therefore, if we set  $p_{max,s}$  to  $t.p$ , then  $h_f$  will be dominated by  $\hat{h}$  since  $\hat{h} \in \mathcal{I}$  and  $\hat{h}.p = p'_{max,s} \leq t.p$ . So  $p'_{max,s}$  should not be updated to  $t.p$ . According to the generality of  $t$ ,  $p'_{max,s}$  should not be updated to the price of

any service-site that is a descendant of  $N$ . So all the descendants of  $N$  can be pruned without checking.  $\square$

**Proof of Lemma 5.3.5:** The proof is similar to the proof of Lemma 5.3.4. According to the definition of  $N.maxp$ , any leaf node that is a descendant of  $N$  must have a price no more than  $N.maxp$ . Without loss of generality, suppose  $t$  is the service-site corresponding to a random selected leaf node which is a descendant of  $N$ . If the current value  $p'_{max,d}$  is equal or greater than  $N.maxp$ , then  $p'_{max,d} \geq t.p$ . According to the definition of  $p_{max,d}$ , the best value of  $p_{max,d}$  should be at least  $p'_{max,d}$ . So  $p'_{max,d}$  should not be updated to  $t.p$ . According to the generality of  $t$ ,  $p'_{max,d}$  should not be updated to the price of any service-site that is a descendant of  $N$ . So all the descendants of  $N$  can be pruned without checking.  $\square$