EFFICIENT EXPLORATION TECHNIQUES ON LARGE DATABASES

by

SENJUTI BASU ROY

Presented to the Faculty of the Graduate School of
The University of Texas at Arlington in Partial Fulfillment
of the Requirements
for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT ARLINGTON
May 2011

To my mother who set the example, and who made me who I am.

ACKNOWLEDGEMENTS

I would like to express my deepest appreciation to my supervising professor and committee chair, Dr. Gautam Das, who continually and convincingly conveyed words of encouragement and a spirit of adventure in regard to research and scholarship. Without his guidance and persistent help, this dissertation would not have been possible. In addition, I extend my sincere thanks to all my committee members for showing interest in my research and serving on my committee; especially, I am extremely grateful to my external committee members, who have exhibited extraordinary enthusiasm and flexibility in their schedule for advising me and keeping this dissertation accessible.

I would also like to extend my gratitude to the department of CSE at UTA and my advisor, Dr. Das for providing me with financial supports during my entire graduate studies. I am especially grateful to my research collaborators, Dr. Sihem Amer-Yahia of Yahoo! Labs, and Dr. Cong Yu of Google Research for introducing me to some extraordinarily interesting problems and engaging me in exciting research discussions.

I am grateful to all my teachers who taught me during the years I spent in school, first in India, and then in the Unites States. In Particular, I would like to express my deepest thanks to my high-school mathematics teacher, Dr. Geetarashmi Basu, and my undergraduate thesis advisor, Dr. Jaya Sil, for encouraging and inspiring me to pursue research.

I would like to convey my heartfelt gratitude to my parents and family. I am especially indebted to my mother, who has been instrumental in shaping up my life. I feel extremely fortunate to be so blessed.

I would like to thank Arjun and Mahashweta, my very good friends and lab mates during my graduate studies. It would have been a lonely lab without them. Many thanks to all my

friends who helped me throughout my career; my research would not have been possible without their constant encouragement. I also acknowledge my UTA friends and other lab mates; without them my UTA life would not have been so much fun.

Finally, I would like to thank my husband Kausik, who always stood by me through thick and thin.

April 14, 2011

ABSTRACT

EFFICIENT EXPLORATION TECHNIQUES ON LARGE DATABASES

SENJUTI BASU ROY, Ph.D.

The University of Texas at Arlington, 2011

Supervising Professor: Gautam Das

Search, retrieval, and exploration of information have become some of the most intense and principal research challenges in many enterprize and e-commerce applications off late. The mainstay of this dissertation is to analyze and investigate different aspects of online data exploration, and propose techniques to accomplish them efficiently. In particular, the results in this dissertation widen the scope of existing *faceted search* and *recommendation systems* - two upcoming fields in data exploration which are still in their infancy.

Faceted search, the de facto standard for e-commerce applications, is an interface framework with the primary design goal of allowing users to explore large information spaces in a flexible manner. We study this alternative search and exploration paradigm in the context of structured and unstructured databases. More specifically, motivated by the rapid need of knowledge discovery and management in large enterprize organizations, we propose *DynaCet*, a minimum effort driven dynamic faceted search system on structured databases. In addition, we study the problem of dynamic faceted retrieval in the context of unstructured data using *Wikipedia*, the largest and most popular encyclopedia. We propose *Facetedpedia*, a faceted retrieval system which is capable of dynamically generating query-dependent facets for a set of Wikipedia articles.

The ever-expanding volume and increasing complexity of information on the web has made *recommender systems* essential tools for users in a variety of information seeking or e-commerce activities by exposing them to the most interesting items, and by offering novelty, diversity, and relevance. Current research suggests that there exists an increasing growth in online social activities that leaves behind trails of information created by users. Interestingly, recommendation tasks stand to benefit immensely by tapping into these latent information sources, and by following those trails. A significant part of this dissertation has investigated on how to improve the online recommendation tasks with novel functionalities by considering additional contexts that can be leveraged by tapping into social data.

To this end, this dissertation investigates problems such as, how to compute recommendation for a group of users, or how to recommend composite items to a user. Underlying models leverage on social data (co-purchase or browsing histories, social book-marking of photos) to derive additional contexts to accomplish those recommendation tasks. In particular, it focuses on techniques that enable a recommendation system to interact with the user in suggesting composite items - such as, bundled products in online shopping, or itinerary planning for vacation travel. We investigate the technical and algorithmic challenges involved in enabling efficient recommendation computation, both from the user (the interaction should be easy, and should converge quickly), as well as the system (efficient computation) points of view.

This dissertation also discusses extensive performance and user study results, which were conducted using the crowd-sourcing platform Amazon Mechanical Turk. We conclude by briefly describing other promising problems with future opportunities in this field.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Table Page

CHAPTER 1

INTRODUCTION

1.1 Overview

This dissertation has focussed in designing novel online data exploration techniques from underlying large data repositories (structured data and web), that extend existing ranked retrieval based query-answering paradigm. In particular, the results in this dissertation widen the scope of existing *faceted search*, and *recommendation systems* - two upcoming fields in data exploration which are still in their infancy.

Broadly speaking, data exploration aids a naive user to explore a large information space in an effective manner. A large number of emerging applications stand to benefit from such exploratory querying techniques; examples include almost any activity a user may perform online - potential customers shopping for regular commodities (e.g., car, electronics, home, clothes etc.), users looking for pertinent restaurants or movies of their interests, readers reading news, articles, reviews etc., or professionals in enterprize organizations searching for some records of their interest from vast data repositories. The mainstay of this dissertation is to analyze and investigate different aspects of online data exploration, and propose techniques to accomplish them efficiently.

Exploratory techniques are extremely useful in the cases, where the user queries are not *too selective* (e.g., *cheap cars, action movies, iPhone within* $200, *database research in graduate schools* etc.), and are not intended towards a single fact (e.g., *population in Texas*). A relevance-based ranked retrieval method may fail to effectively distinguish a smaller subset of results from there. Therefore, the query-answering task needs to be further enhanced to capture additional context and intent that the user may have in mind during querying. In addition, the increase in online social activity has given rise to additional opportunity and challenges in query-answering tasks. *Recommendation*, as a research topic has emerged to facilitate user in decision making in such cases. Furthermore, sometimes a user is unfamiliar about the domain, or unsure about her goal. Exploratory browsing techniques (such as *Faceted Search*) are of great assistance that facilitates and guides users to focus on the relevant aspects of her search results.

Consider a potential car buyer searching for a suitable used car listed in an online auto dealers database, where each car is described via numerous attributes such as Make, Model, Mileage, and so on. While the buyer is eventually interested in buying only one car, at the beginning of her search she may only have a few preferences in mind; thus an exploratory search is necessary to narrow down the choices.

Faceted search is a novel exploratory search technique to aid users in exploring items of interest within such vast data repository. It allows users to refine or navigate a collection of information by using a number of discrete attributes - the so-called *facets*. A facet represents a specific perspective on content that is typically clearly bounded and mutually exclusive (e.g., color , manufacturers or model of a car). The values within a facet can be a flat list (e.g. all possible manufacturers of cars) that allows only one choice or a hierarchical list that allows user to drill-down through multiple levels (e.g., car types → mid size car → compact executive car → D-segment). The combination of all facets and values is often referred to as a *faceted interface*. In the previous example, a user may have the option of drilling down through different facets, first by color (color → Black), then by car types (car types → mid size car → compact executive car → D-segment), and then by make (Make → Japanese Car → Honda), and so on.

The effectiveness of faceted search lies in the ability of users to create their own custom navigation by combining various perspectives rather than forcing them through a specific path. Lately, faceted search has become the de-facto standard for e-commerce and product- related web-sites, from big box stores to product review sites. Any content-heavy sites such as media publishers (e.g. Financial Times: ft.com), libraries (e.g. NCSU Libraries: lib.ncsu.edu/), and even non-profit organizations (e.g. Urban Land Institute: uli.org) are tapping into faceted search to make their vast repository content easily explorable. We analyze the opportunities of adopting principles of the faceted search paradigm for tuple search. However, unlike past works on images and unstructured data such as text, here, the challenge is to dynamically determine the facets that are best suited for enabling a faceted search interface. Our proposed faceted search framework is dynamic and completely user interaction dependent as compared to existing faceted search systems where the facets and hierarchies are predefined and static [1, 2, 3]. Our overall goal is to judiciously select the facet(s) dynamically based on user query, that the user employs further to drill down in the results, such that user's *navigational cost* during this exploration process is minimized. We investigate dynamic minimum effort driven faceted search in conjunction with structured and unstructured data.

In the cars database example above, a very simple faceted search interface is one where the user is prompted an attribute[1] (e.g., Make), to which she responds with a desired value (e.g., "Honda"), after which the next appropriate attribute (e.g., Model) is suggested to which she responds with a desired value (e.g., "Accord"), and so on. Our overall goal is to judiciously select the next facet(s) dynamically at every step, so that the user reaches the desired tuples with *minimum effort*. While the effort expended by a user during a search/navigation session may be fairly complex to measure, we focus on a rather simple but intuitive metric: the expected number of queries that the user has to answer in order to reach the tuples of interest.

In particular, we propose **DynaCet**, the minimum effort driven faceted search system that focuses upon two broad problem areas. *1. Faceted Search as an Alternative to*

[1] Henceforth in this dissertation *facets* and *attributes* will be used interchangeably.

Ranked-Retrieval and *2. Faceted Search that Leverages Ranking Functions*. We elaborate on these problems and propose several of their variants in this dissertation. Furthermore, we investigate fast implementation techniques of our proposed facet selection algorithms to achieve significant CPU speed up. This work has been published in [4, 5, 6].

Next, we investigate faceted search in conjunction with unstructured data. To that end, we propose **FacetedPedia**, a faceted retrieval system for information discovery and exploration in *Wikipedia*. Given the set of *Wikipedia* articles resulting from a keyword query, *Facetedpedia* generates a faceted interface for navigating the result articles. Compared with other faceted retrieval systems [1, 2, 3], *Facetedpedia* is fully automatic and dynamic in both facet generation and hierarchy construction, and the facets are based on the rich semantic information from *Wikipedia*. The essence of our approach is to build upon the collaborative vocabulary in *Wikipedia*, specifically the intensive internal structure (hyperlinks) and folksonomy (category system). Given the sheer size and complexity of this corpus, the space of possible choices of faceted interfaces is prohibitively large. We propose metrics for ranking individual facet hierarchies by user's navigational cost, and metrics for ranking interfaces (each with k facets) by both their average pairwise similarities and average navigational costs. We thus develop faceted interface discovery algorithms that optimize the ranking metrics. This work has been accepted in [7, 8].

This dissertation also discusses *recommendation*, another emerging information exploration technique that aims to support users in their decision-making while interacting with large information spaces. Broadly speaking, a recommendation system automatically predicts how much a user will like an item that is unknown to her. Recommended items of interest are based on preferences that a user has expressed, either explicitly or implicitly. The ever-expanding volume and increasing complexity of information on the web has therefore made such systems essential tools for users in a variety of information seeking or e-commerce activities. Recommender systems help overcome the information overload problem by exposing users to the most interesting items, and by offering novelty, surprise, and relevance. This dissertation also investigates novel problems in the area of recommender systems.

Single user recommendation has received significant attention in the past due to its extensive use in Amazon and Netflix. A significant number of single user recommender systems are already in use, including *launch.com* for online music, *entree* for restaurant recommendation, *alkindi, Moviefinder, Movielens* for movie recommendation etc. In addition, social networking sites like Facebook, content sites such as Yahoo! Travel, which have traditionally focused on managing content only, are beginning to encourage people to form social ties and share content. While there has been a recent trend in developing techniques for finding relevant content on social content sites [9], very little has been done to *help socially acquainted individuals* find content of interest to all of them together. The need for **group recommendation** arises in many scenarios: a movie for friends to watch together, a travel destination for a family to spend a holiday break, and a good restaurant for colleagues to have a working lunch. Intuitively, items that are ideal for recommendation to a group may be quite different from those for individual members.

In this dissertation, we analyze the desiderata of group recommendation and propose a formal semantics that accounts for both item relevance to a group and disagreements among group members. We design and implement algorithms for efficiently computing group recommendations. We further explore the impact of space constraints on maintaining per-user and pair-wise item lists and develop two complementary solutions that leverage shared user behavior to maintain the efficiency of our recommendation algorithms within a space budget. These results have been published in [10].

We further investigate how to extend the idea of single-item recommendation to **composite item recommendation**, and enable user interaction in recommendation computation. The overall objective is to recommend composite items by interacting with the user, and considering her preference. Several online applications such as shopping, travels, and so on benefit from such composite item recommendations. For example, a user shopping for an iPhone with a price budget can be presented with both the iPhone (central item) and a package of other items that match well with the iPhone (e.g., Belkin case, Bose sounddock, Kroo USB cable) as a composite item, whose total price is within the user's budget. Alternatively, a traveler, starting from a particular location in a city (central item) with certain budgets (time, money) can be presented with an itinerary (that consists of different POIs) that can be visited within her specified budget. The heart of this problem is the existence of different relationships between the individual items in a composite item. As an example, in the case of former example, the rest of the items in a package forms the shape of a *star* with the central item, whereas, in the latter example, the POIs in an itinerary form a *chain*. Different relationship between individual items gives rise to different modeling, and subsequently completely different solution.

In the case of **star model**, we define the problem as effective construction and exploration of large sets of packages associated with a central item, and design and implement efficient algorithms for solving the problem in two stages: summarization, a technique which picks k representative packages for each central item; and visual effect optimization, which helps the user diverse composite items quickly by minimizing overlap between packages. The challenge is to design efficient algorithms, since many of these problems are NP-complete in nature. We propose formal proof and principled solutions for these problems. This work has appeared in [11].

In the case of **chain model**, we formalize itinerary recommendation as an iterative process, where, at each step: (1) the user provides feedback on POIs selected by the system, (2) the system recommends the best itineraries based on all feedback so far, and (3) the system further selects a new set of POIs, with optimal utility, to solicit feedback for, at the next step. This iterative process stops when the user is satisfied with the recommended itinerary. We show that computing an itinerary is NP-complete even for simple itinerary scoring functions and that POI selection is NP-complete. We develop heuristics and optimizations for a specific case where the score of an itinerary is proportional to the number of desired POIs. This work has been accepted recently in [1].

In addition to formal modeling and theoretical proofs, my PhD dissertation extensively validates our proposed solutions empirically, by running rigorous experiments, and comprehensive user studies.

Our results in this dissertation can be divided into two broad categories: In the first category, we consider the problem of faceted search in the contexts of structured and unstructured data. In particular, we propose *DynaCet* that enables effective tuple search using faceted navigation on structured databases. At each step, *DynaCet* dynamically suggests facets based on user's response at previous step such that the expected number of such interactions with user is minimized. In addition, we consider the desiderata of faceted retrieval using unstructured data. Precisely, we propose *FacetedPedia*, a faceted retrieval system for information discovery and exploration in *Wikipedia*.

In the next category, we discuss the recommendation related problems. Broadly speaking, our contributions in this space are in solving *group recommendation* problems, and in investigating *composite item construction*. Our main results in *group recommendation* problem are - modeling the semantics of group recommendation and presenting algorithms to compute them efficiently. We further explore the impact of space constraints on maintaining per-user and pair-wise item lists. We propose *behavior factoring*, that factors out user agreements from disagreement lists, and *partial materialization*, that selectively materializes a subset of disagreement lists to accomplish this task. These results have appeared in [10].

In particular, we investigate two different models in the *composite item* space - the *star model*, and the *chain model*. For the *star model*, we aim to solve the problem of identifying all *valid* and *maximal* satellite packages, given a central item. We show that the number of valid and maximal packages associated with a central item is typically very large and presenting all of them to the user is impractical. Hence we investigate techniques to choose a set of k-representative maximal packages using *summarization* and returning those k-packages to the user using *visual effect* optimization.

For the *chain model*, we introduce and formalize the novel approach of interactive itinerary planning based on user feedback and itinerary expected scores. We formalize the *optimal itinerary construction problem*, and *optimal POI batch selection problem*, and prove the hardness. We design efficient algorithms for solving both these problems.

In the rest of this introductory chapter, we provide a brief synopsis of our *significant* contributions.

1.2 Category I: Faceted Search

1.2.1 DynaCet Results

1. We initiate research into the problem of automated facet discovery to enable minimum-effort driven faceted search in structured databases. We adopt a simple approximation algorithm, and show how this approach can be extended to incorporate the notion of facet uncertainty. We discuss how this approach is different from other attribute selection techniques.
2. We also extend our methods to work in conjunction with available ranking functions for tuples. We show how our methods are different from other attribute selection techniques in the presence of ranking functions such as [12].

3. We develop novel scalable implementation techniques of our algorithms. In particular, we leverage pipelining execution ranking models to avoid complete database scans at any time.
4. We describe the results of a thorough experimental evaluation of our proposed techniques.
5. We propose techniques to improve the performance of the facet selection algorithms by reducing CPU intensive computations. The main idea is a novel adaptation of the early stopping techniques used in the TA-family of algorithms for top-k computations [13, 14, 15]. Such techniques can attain early termination that avoid scanning and scoring the complete database in determining the next most promising facet.

1.2.2 FacetedPedia Results

1. Concept: Faceted *Wikipedia*: We propose an automatic and dynamic faceted retrieval system for *Wikipedia*. To the best of our knowledge, this is the first system of its kind. The key philosophy of our approach is to exploit collaborative vocabulary as the backbone of faceted interfaces.
2. Metrics: Facet Ranking: Based on a user navigation model, we propose metrics for measuring the “goodness” of facets, both individually and collectively.
3. Algorithms: Faceted Interface Discovery: We develop effective and efficient algorithms for discovering faceted interfaces in large search space.
4. System Evaluation: *Facetedpedia*: We conducted user study to evaluate the effectiveness of the system and to compare with alternative approaches. We also measured its quality and efficiency quantitatively.

1.3 Category II: Recommendation

1.3.1 Subcategory I: Composite Item Construction

We discuss star composite item recommendation, and chain composite recommendation problems to that end.

1.3.1.1 Star Composite Items

1. We propose the notion of composite item and compatible satellite package in the context of online data exploration. To help users effectively explore composite items, we formalize the problems of finding valid and maximal packages given a budget, finding representative packages through summarization, and reordering packages for visual effect optimization.
2. We design and implement a random walk algorithm to efficiently construct all valid and maximal packages.
3. We introduce a novel principle for summarizing a large set of maximal packages associated with one central item, and develop a max-k set coverage algorithm for efficient summarization. We further improve the efficiency of summarization by integrating it with the random walk package construction algorithm.
4. We formulate the problem of optimizing the visual effect of k packages associated with the same central item as that of finding an ordering of the packages that minimizes overlap between consecutive packages. We prove that this problem is NP-Complete, and design and implement a heuristic algorithm for solving it. In addition, we also prove that this algorithm is optimal when there is only one satellite type.

1.3.1.2 Chain Composite Items

1. We introduce and formalize the novel approach of interactive itinerary planning based on user feedback and itinerary expected scores.
2. We formally define the *optimal itinerary construction problem*, which is one of the two core problems in interactive itinerary planning. We prove NP-completeness of this problem and design an efficient real-time heuristic algorithm for computing itineraries based on user feedback and time budget.
3. We formally define the *optimal POI batch selection problem*, which is the other core problem, and propose a probabilistic model based on the notion of expected itinerary score given user feedback on a POI batch. We prove NP-completeness of this problem and design efficient heuristics for selecting a good batch of POIs.
4. Finally, we run extensive experiments validating our approach on real datasets. Quality experiments confirm the effectiveness of our algorithms for interactive itinerary planning and performance experiments demonstrate their efficiency.

1.3.2 Subcategory II: Group Recommendation

1. We formalize the problem of group recommendation and define its semantics as a consensus function that aims at maximizing item relevance and minimizing disagreements between group members.
2. We prove that the two important disagreement models being proposed satisfy the conditions required by the family of top-k threshold algorithms.
3. We design efficient algorithms based on one representative threshold algorithm, TA, to perform top-k group recommendation.
4. We formalize two optimizations: the problem of which disagreement lists to materialize given a space budget and the refinement of score bounds.

5. We conduct a comprehensive experimental evaluation, including a user study on Amazon Mechanical Turk to demonstrate the benefits of incorporating disagreements into group recommendation, and extensive experiments to demonstrate the efficiency of our algorithms.

To summarize, Chapter 2 contains results on *DynaCet*, Chapter 3 on *FacetedPedia*, Chapter 4 on *star composite item*, Chapter 5 on *chain composite item*, and Chapter 6 on *group recommendation* problem. We conclude by discussing ongoing and future research directions briefly in Chapter 7.

CHAPTER 2

DYNACET: MINIMUM-EFFORT DRIVEN DYNAMIC FACETED SEARCH IN STRUCTURED DATABASES

2.1 Introduction

One of the primary problems that many organizations face is that of facilitating effective search for data records within vast data warehouses. For example, consider the customer database of a large financial institution such as a bank. A data analyst or a customer service representative for the bank often has to search for records of a specific customer or a specific account in such databases. Of course, if the relevant tuple is uniquely identifiable by an identifier known to the user, this problem is trivial. But in most cases the user only has partial information about the tuple (e.g., perhaps the values of a few of its attributes) and thus it is necessary to enable an effective search procedure. As another example, consider a potential car buyer searching for a suitable used car listed in an online auto dealer's database, where each car is described via numerous attributes such as Make, Model, Mileage, and so on. While the buyer is eventually interested in buying only one car, at the beginning of her search she may only have a few preferences in mind (e.g., a late model family sedan with low mileage); thus a search is necessary to narrow down the choices.

One approach for enabling tuple search in databases is IR style *ranked retrieval* from databases. For the cars example above, a query such as "CarType=sedan, Age$<$5, Mileage$<$10k" can be specified via a form-based interface, and rather than simply executing the query using SQL - which will result in a flood of results since there are presumably many cars in the database that satisfy such broad query conditions - ranking-based systems will attempt to rank and retrieve the top-k most “relevant” tuples that satisfy these conditions (where k is usually a small number, such as 10-20). Much of the recent research has focused on the design of suitable ranking functions, as well as on the design of efficient retrieval algorithms [16, 17, 18].

However, recently, other search paradigms have gained popularity in certain specialized IR domains, including for searching over image and text data. In particular, it has been argued that *faceted search* interfaces can be extremely useful in user navigation and search [19, 2]. E.g., a user searching for a photograph of the Great Wall at a photo hosting website may have the option of drilling down via different facets of the dataset, e.g., first by geographical regions (such as Asia → China → Beijing), then via age (such as period → ancient), then via phototype (man made → historical monuments). While it remains to be seen if faceted search is a viable option for searching at the Web scale, it does offer a promising alternative in specialized domains such as these examples.

Main Goal of DynaCet - Investigate Faceted Search in Databases: The main goal of DynaCet is to explore the opportunities of adopting principles of the faceted search paradigm for tuple search in structured databases. However, unlike past works on images and text data, where the primary task is to design hierarchical meta-data and facets to enable faceted search, structured databases come with the tremendous advantage that they are already associated with rich meta-data in the form of tables, attributes and dimensions, known domain ranges, and so on. Instead, the challenge is to determine, from the abundance of available meta-data, which attributes of the tuples are best suited for enabling a faceted search interface. In the cars database example above, a very simple faceted search interface is one where the user is prompted an attribute[1] (e.g., Make), to which she responds with a desired value (e.g., “Honda”), after which the next appropriate attribute (e.g., Model) is suggested to which she responds with a desired value (e.g., “Accord”), and so on. In this thesis we fo-

[1] Henceforth in this thesis *facets* and *attributes* will be used interchangeably.

cus on two broad problem areas. We briefly elaborate on these problems and our solutions below.

1. FSNoRank - Faceted Search as an Alternative to Ranked-Retrieval: We first consider the problem where we don't assume any tuple relevance and ranking function as being available. Thus when a user poses an initial selection query, without any further information from the user we can only assume that all of the selected tuples are equally preferred by the user. Our task is then to develop a dialog with the user to extract more information from her on other desired attribute values - essentially initiate a facet-by-facet drill down procedure to enable her to zoom in on the tuple(s) of interest. Our overall goal is to judiciously select the next facets dynamically at every step, so that the user reaches the desired tuples with *minimum effort*. While the effort expended by a user during a search/navigation session may be fairly complex to measure, we focus on a rather simple but intuitive metric: *the expected number of queries that the user has to answer in order to reach the tuples of interest.*

Variants of this problem have been considered in [20] in the context of interactive question-answer applications. It was shown that the problem is intractable, and an approximation algorithm suggested with provably good performance. While we adopt the same cost metric, we extend the idea in several important ways. We propose a novel cost model for fast tuple search which assumes that attributes are associated with *uncertainties*, where the uncertainty of an attribute refers to the probability with which a user can provide a value that belongs to the domain of the attribute. We develop facet selection techniques that take into account such uncertainties.

Also, we formally show that the approximation algorithm for building minimal cost decision trees given in [20] generates trees different from those generated by other classical decision tree construction algorithms based on information gain, as well as other classical dimensionality reduction techniques such as principal component analysis (PCA).

2. FSRank - Faceted Search that Leverages Ranking Functions: We next ask whether faceted search procedures can work *in conjunction with* ranking functions. This is a novel problem area, and to the best of our knowledge, has not been investigated before. Recall that a ranked-retrieval system typically assigns relevance scores to all selected tuples and returns only the top-k tuples. From a faceted search perspective, we may view the ranking function as imposing a *skew* over the user preferences for the selected tuples, and thus would like to select the facet that directs the user towards the most preferred tuples as efficiently as possible. One interesting complication is that these tuple preferences (or scores) may change as the faceted search progresses; this is because as new attribute information is provided by the user, the ranking function may re-evaluate the scores of the remaining tuples still in contention. Thus a faceted search system in conjunction with a ranking function offers the benefits of focused retrieval as well as drill-down flexibility. We provide a formal definition of this problem, and offer a solution for facet selection that is based on minimum-effort driven principles.

The main contributions of our thesis may be summarized as follows:

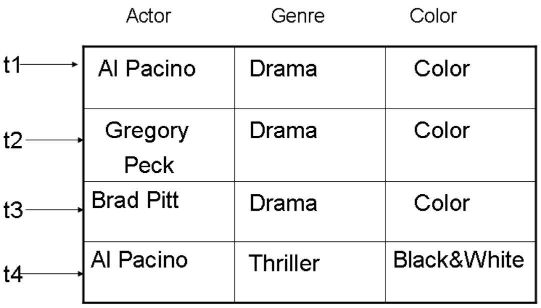
1. We present *DynaCet* [5]- a middle-ware system that sits between the user and the database and dynamically suggests facets for drilling down into the database. The facet suggestion model is driven by our intent to provide a *minimum-effort* database exploration solution for enterpriser users. We focus on a simple but intuitive metric for measuring effort: *the expected number of queries that the user has to answer in order to reach the tuples of interest*.
2. We adopt a simple approximation algorithm, and show how this approach can be extended to incorporate the notion of facet uncertainty. We discuss how this approach is different from other attribute selection techniques.

3. We also extend our methods to work in conjunction with available ranking functions for tuples. We show how our methods are different from other attribute selection techniques in the presence of ranking functions such as [12].
4. We develop novel scalable implementation techniques of our algorithms using a modified Rainforest framework [21]. Furthermore, we leverage pipelining execution ranking models to avoid complete database scans (referred to as the *Full Scan Algorithm* in this thesis) at any time.
5. We describe the results of a thorough experimental evaluation of our proposed techniques.

2.2 FSNoRank - Faceted Search as an Alternative to Ranked Retrieval

Let D be a relational table with n tuples $\{t_1, t_2, \ldots, t_n\}$ and m categorical attributes $A = \{A_1, A_2, \ldots, A_m\}$, each with domain Dom_i (for the rest of this thesis we only consider categorical data, and assume that numeric data has been suitably discretized). Assume that no two tuples are identical and that a user wishes to retrieve a tuple from this database. The faceted search system will prompt the user with a series of questions, where each question takes the form of an attribute name, and to which the user responds with a value from its domain. This drill-down process terminates when a unique tuple has been isolated. The task is to design a faceted search system which asks the minimum number of questions on the average, assuming that each tuple is equally likely to be preferred by the user (thus we do not assume the presence of a ranking function).

Essentially, the task is to build a *decision tree* which distinguishes each tuple by testing attribute values (asking questions). Each node of the tree represents an attribute, and each edge leading out of the node is labeled with a value from the attribute's domain. As an example consider Figure 2.1 which refers to a toy *movie* database with three attributes and four tuples. A decision tree for identifying each of the tuples in the tuple set $D = \{t_1, t_2, t_3, t_4\}$ is shown in Figure 2.2. The leaves of the tree represent the tuple set D and each tuple appears exactly once. A user reaches her tuple of interest by picking a path after each non-leaf node in the tree i.e. by assigning a value to each attribute query on the path leading to the tuple.



	Actor	Genre	Color
t1	Al Pacino	Drama	Color
t2	Gregory Peck	Drama	Color
t3	Brad Pitt	Drama	Color
t4	Al Pacino	Thriller	Black&White

Figure 2.1. A small movies database.

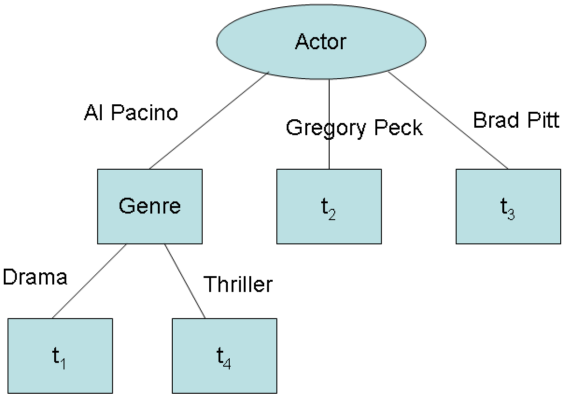


Figure 2.2. An optimal decision tree.

Given such a tree T, $cost(T)$ can be defined as the average tree height, $\sum_i ht(t_i)/n$ where $ht(t_i)$ is the height of leaf t_i. Equivalently, cost (i.e., effort) represents the expected number of queries that needs to be answered before the user arrives at a preferred tuple (assuming all tuples are equally likely to be preferred). It is easy to verify that the tree in Figure 2.2 is optimal (with minimum cost = $(2 + 2 + 1 + 1)/4 = 1.5$).

The problem of determining the minimum cost tree has been studied in the past in the context of question-answering dialog systems, and shown to be NP-complete (see [20] and references therein). A greedy approximation algorithm has been developed [20] which achieves an approximation factor of $O(\log d \log n)$ in the cost, where d is the maximum domain size of any attribute. Although the approximation factor appears large, it is the only theoretical approximation bound known for this problem. Moreover, as our experiments show, this algorithm performs quite well in practice. We describe this algorithm next as it forms the foundation for all our facet selection procedures.

The intuition is that any decision tree should distinguish every pair of distinct tuples. The approach is to make the attribute that distinguishes the maximum number of pairs of tuples as the root of the tree, where an attribute A_l is said to distinguish a pair of tuples t_i, t_j if $t_i[l] \neq t_j[l]$. Picking the attribute A_l as the root node partitions the database D into disjoint tuple sets $D_{x_1}, D_{x_2}, \ldots, D_{x_{|Dom_l|}}$, where each D_{x_q} is the set of tuples that share the same attribute value x_q of A_l. Using this intuition, we seek to select as root attribute A_l that minimizes the number of indistinguishable pairs of tuples. Hence, formally the function, Indg() seeks to minimize,

$$Indg(A_l, D) = \sum_{1 \leq q \leq |Dom_l|} |D_{x_q}|(|D_{x_q}| - 1)/2 \tag{2.1}$$

This process is recursively repeated for all sets D_{x_q}, until each set reduces to singleton tuples. Applying this algorithm to the database in Figure 2.1 gives the same resultant decision tree as shown in Figure 2.2. We see that $Indg(Actor) = 1$, while $Indg(Genre)$ = $Indg(Color)$ = 3. Thus Actor should be the root.

2.2.1 Comparing Against Other Attribute Selection Procedures

Comparing Against Information Gain: Decision tree construction is a very well understood process in machine learning and data mining, and several popular algorithms such as ID3 and C4.5 have been developed [22]. These algorithms are designed for the *classification* problem, and seek to maximize classification accuracy and avoid over-fitting. In contrast, our goal is not to solve a classification problem - rather our aim is to build full decision trees (where each leaf is a tuple) that minimizes average root-to-leaf path lengths. A popular heuristic used by these algorithms (e.g., ID3) for selecting the next feature, or “splitting” attribute, is the *information gain* measure. Since there is no class variable associated with the database, we may imagine that each tuple consists of its own unique class, and thus the information gain of an attribute A_l is equivalent to

$$InfoGain(A_l, D) = \log n - \frac{1}{n}\left(\sum_{1 \le q \le |Dom_l|} |D_{x_q}| \log |D_{x_q}|\right) \tag{2.2}$$

The selected facet may be the one with the largest information gain. Unlike the $Indg()$ based approach for which there are known approximation bounds, it is open whether similar approximation bounds exist for information gain based approaches. In fact, as we show now, the information gain heuristic produces different trees than the approach of minimizing $Indg()$.

Lemma 2.2.1. *Given a database D, the decision tree constructed by selecting facets that minimize $Indg()$ may be different from the decision tree constructed by selecting facets that maximize $InfoGain()$.*

Proof: Consider two attributes A and B of a database table D. Let A be a Boolean attribute with domain $\{a_1, a_2\}$. Let $n(x)$ represent the number of tuples with attribute value x. Let

$n(a_1) = n(a_2) = n/2$. Let the domain of B be $\{b_1, b_2, \ldots, b_{n/(2+\sqrt{2})+1}\}$ where $n(b_1) = n/\sqrt{2}$ and $n(b_2) = \cdots = n(b_{n/(2+\sqrt{2})+1}) = 1$. We then have

$$Indg(A, D) = \frac{n}{2}\left(\frac{\frac{n}{2}-1}{2}\right) + \frac{n}{2}\left(\frac{\frac{n}{2}-1}{2}\right) = \frac{n(n-2)}{4}$$
$$Indg(B, D) = \frac{n}{\sqrt{2}}\left(\frac{\frac{n}{\sqrt{2}}-1}{2}\right) = \frac{n(n-\sqrt{2})}{4}$$

Clearly $Indg(B, D) > IndgA, D)$, and thus A will be preferred over B during facet selection. We next consider the information gain heuristic. We then have

$$InfoGain(A, D) = \log n - \frac{1}{n}\left(\frac{n}{2}\log\left(\frac{n}{2}\right) + \frac{n}{2}\log\left(\frac{n}{2}\right)\right) = 1$$
$$InfoGain(B, D) = \log n - \frac{1}{n}(\frac{n}{\sqrt{2}}\left(\frac{n}{\sqrt{2}}\log\left(\frac{n}{\sqrt{2}}\right)\right))$$
$$= \log n - \frac{\left(\log n - \frac{1}{2}\right)}{\sqrt{2}}$$

Clearly $InfoGain(B, D) > InfoGain(A, D)$ and thus B will be preferred over A during facet selection. These arguments demonstrate that the tree produced by maximizing information gain may be different from the tree produced by minimizing $Indg()$.

Comparing Against Principal Component Analysis (PCA): We explore the popular technique of *principal component analysis* (PCA) [23] to see if it is applicable in facet selection. PCA has traditionally been developed for dimensionality reduction in numeric datasets, thus extending PCA to categorical databases such as ours requires some care. We illustrate these ideas by again considering the small movies database in Figure 2.1. Suppose we wish to reduce the dimension of this database from three to two and decide to retain the dimensions Genre and Color. In that case, the attribute Actor has to be homogenized (i.e., all values have to be transformed to a single common value) such that the number of values that are changed is minimized. It is easy to see that if we make all Actors as “Al Pacino”, this will require minimum number of changes (two changes, i.e., the Actor field in tuples t_2 and t_3). Hence the cost of the reduction is two in this case. On the other hand, if we decide to retain the dimensions Actor and Genre, only one value in the database needs to be changed (the Color field of t_4 has to be changed to “Color”). Thus, reducing the dimensions to Actor and Genre is cheaper than (and thus preferable to) reducing the dimensions to Genre and Color. More specifically, the best k attributes we retain are the ones that have the smallest modes. Mode of an attribute is defined as:

$$Mode(A_l, D) = \max\{|D_{x_q}|, |1 \leq q \leq |Dom_l|\}$$

Lemma 2.2.2. *Given a database D, the decision tree constructed by selecting facets that minimize $Indg()$ may be different from the decision tree constructed by selecting facets that minimize $Mode()$.*

The proof for above lemma can be derived using similar intuition as for prior lemma. The details are omitted in the interest of space. Among all three heuristics, only the $Indg()$ based approach has a known approximation factor associated with it and performs better in experimental evaluation.

2.2.2 Modeling Uncertainty in User Knowledge

The facet selection algorithm presented above assumes that the user knows the answer to any attribute that is selected as the next facet. In a practical setting, this is not very realistic. For example, a customer service representative of a bank searching for a specific customer may not know exactly the street address of the customer’s residence; likewise a user searching for a movie may not be sure of the director of the desired movie, and so on. One of the contributions of this thesis is to recognize that there are inherent uncertainties associated with the user’s knowledge of an entity’s attribute values, and accordingly to build decision trees that take such uncertainties into account.

In the simplest case, each attribute A_i of the database is associated with a probability p_i that signifies the likelihood that a random user knows the answer to the corresponding query. For example, in a cars database, the attribute Car Type may be associated with a probability of 0.8 (i.e., 80% of users know whether they want a sedan, hatchback, SUV, etc.) For simplicity we assume no correlations between attribute uncertainties (i.e., a user who does not know the car type is still assumed to specify heated seats with a finite probability) nor other more general uncertainty models. Estimating these probabilities require access to external knowledge sources beyond the database such as domain experts, user surveys, and analyzing past query logs.

In this thesis, we assume that the uncertainty models have already been estimated. In designing our decision trees to cope with uncertainty, we assume that users can respond to a question by either (a) providing the correct value of the queried attribute A_i, or (b) responding with a "don't know". In either case, the faceted search system has to respond by questioning the user with a fresh attribute. Consider Figure 2.3, which shows the decision tree of the same database of Figure 2.1. Assume each of the attributes has associated uncertainties. Consequently, each node in the decision tree also has an associated "don't know" link. As can be seen, the leaf nodes in this decision tree are either a single tuple, a set of tuples, or, at worst, the entire database. Moreover, note that the tuples of the database do not occupy unique leaves in the decision tree. For example, there are 7 different path instances of tuple t_1. This implies that when attempting to reach a tuple, different users may follow different paths through the tree.

At this context, we organized a small survey among 20 people selected from the students and faculty of our university.In that survey, each person assigned a value (between 0 to 1) for each attribute. This value denotes the likelihood (probability) with which she is able to answer the question corresponding to that attribute. The overall probability of each attribute is calculated by averaging all 20 values.

Algorithm 1: Single Facet Based Search(D, A')

```
Input: $D$, a set $A' \subset A$ of attributes not yet used
Global parameters: an uncertainty $p_i$ for each attribute $A_i \in A$
Output: A decision tree $T$ for $D$
begin
if $|D| = 1$ then
    Return a tree with any attribute $A_l \in A'$ as a singleton node
if $|A'| = 1$ then
    Return a tree with the attribute in $A'$ as a singleton node
Let $A_l$ be the attribute that distinguishes the maximum expected number of pairs
$A_l = argmin_{A_s \in A'}(1 - p_s) \times |D|(|D| - 1)/2 + p_s \times Indg(A_s, D)$
Create the root node with $A_l$ as its attribute label
for each $x_q \in Dom_l$ do
    Let $D_{x_q} = \{t \in D | t[l] = x_q\}$
    $T_{x_q}$ = Single-Facet-Based-Search$(D_{x_q}, A' - \{A_l\})$
    Add $T_{x_q}$ to $T$ by adding a link from $A_l$ to $T_{x_q}$ with label $x_q$
Create the “don’t know” link:
$T'$ = Single-Facet-Based-Search$(D, A' - \{A_l\})$
Add $T'$ to $T$ by adding a link from $A_l$ to $T'$ with label “don’t know”
Return $T$ with $A_l$ as root
end
```

Thus our challenge is to build such decision trees such that the expected path length through the tree is minimized. Our Single Facet based search algorithm is shown in Algorithm 1. However, we note that each node A_l now has $|Dom_l| + 1$ links, with one of the links labeled as “don’t know”. This link is taken with probability $1 - p_l$, whereas the

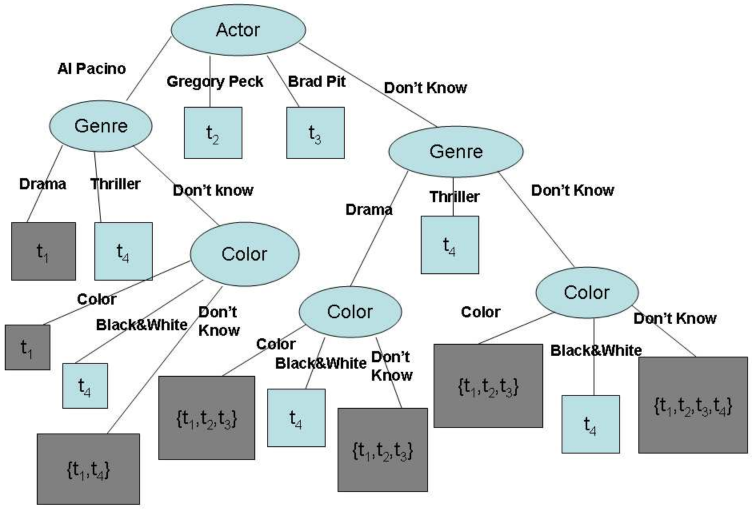


Figure 2.3. The decision tree of Figure 2.1 with uncertainty models.

rest of the links are taken with probability p_l. Thus, in the former case, the attribute A_l cannot distinguish any further pairs of tuples (the query was essentially wasted), whereas in the latter case, only $Indg(A_l, D)$ pairs were left indistinguishable. Thus, we can see that if we select A_l as the root node, then the *expected* number of tuple pairs that cannot be distinguished is $(1 - p_l) \times |D|(|D| - 1)/2 + p_l \times Indg(A_l, D)$. Consequently, an obscure attribute that has little chance of being answered correctly by most users, but is otherwise very effective in distinguishing attributes, will be overlooked in favor of other attributes in the decision tree construction.

2.2.3 Extending to k-Facets Selection

Next, we extend our model further by giving the user more flexibility at every step. As a practical consideration, a decision tree as shown in Figure 2.3 can sometimes be tedious to a user. It may be more efficient to present, at every step, *several* (say k) attributes to the user at the same time, with the hope that the user might know the correct value of one of the proffered attributes.

It may appear that for designing the root node of the decision tree for the k-facet case, instead of considering only m possible attributes as we did for the single-facet case, we will need to consider m_{C_k} sets of attributes of size k each, and from them, select the set that is the best at disambiguating tuple pairs. However, if we restrict the user to answering only one question at each iteration, the problem of determining best k-facets at any node in this decision tree has a much simpler solution - we order the unused attributes from the one that distinguishes most number of tuple pairs to the one that distinguishes the least number of tuple pairs, and select the top-k attributes from this sequence.

In this tree, the probability that a random user will follow “don’t know” links is much smaller than the single-facet case. For example, given the set of attributes A'' at the root, the probability that a random user will be unable to answer any of the k questions is $\prod_{A_l \in A''}(1 - p_l)$. Thus we expect such trees to be more efficient (i.e., shallower) than the trees in the single-facet case.

2.2.4 Designing a Fixed k-Facets Interface

In certain applications, it is disconcerting for the user to be continuously presented with new sets of attributes after every response. Such users would prefer to be presented with a single fixed form-like interface, in which a reasonably large (k) number of attributes are shown, and the user assigns values to as many of the preferred attributes as she can. If the space available on the interface is restricted such that only $k < m$ attributes can be shown, the task is then to select the best set of k attributes such that the expected number of tuples that can be distinguished via this interface can be maximized. We formalize this problem as follows: Given a database D, a number k, and uncertainties p_i for all attributes A_i, select k attributes such that the expected number of tuples that can be distinguished is maximized. If we assume that there are no uncertainties associated with attributes, this problem has similarities with the classical problem of computing minimum-sized *keys* of database relations, and with the problem of computing approximate keys of size k (see [24]).

However, in our case the problem is complicated by the fact that attributes are associated with uncertainties, thus such deterministic procedures [24] appear difficult to generalize to the probabilistic case. Instead, we propose a greedy strategy for selecting the k facets that is based on some of the underlying principles developed in our earlier algorithms. The overall idea is, if we have already selected a set A' of k' attributes, the task is then to select the next attribute A_l such that the expected number of pairs of tuples that cannot be distinguished by $A' \cup \{A_l\}$ is minimized.

Ignoring attribute uncertainties, the algorithm can be described as follows. Let $A' \cup \{A_l\}$ partition D into the sets $D_1, D_2, \ldots, D_d$ where within each set the tuples agree on the values of attributes in $A' \cup \{A_l\}$. Thus, we should select A_l such that the quantity $\sum_i |D_i|(|D_i| - 1)/2$ is minimized. Introducing attribute uncertainties implies that $A' \cup \{A_l\}$ does not always partition D into the sets $D_1, D_2, \ldots, D_d$. Rather, depending on the user interactions, the possible partitions could vary between finest possible partitioning, $P_{fine}(A' \cup \{A_l\}) = \{D_1, D_2, \ldots, D_d\}$, to the coarsest possible partitioning $P_{coarse}(A' \cup \{A_l\}) = \{D\}$ (the latter happens if the user responds to each attribute with a “don’t know”). Each intermediate partitioning occurs when the user responds with a “don’t know” to some subset of the attributes.

Consider any partitioning $P = \{U_1, U_2, \ldots U_u\}$. Let the quantity $IndgPartition(P)$ be defined as $\sum_i |U_i|(|U_i| - 1)/2$. This represents the number of tuple pairs that fail to be distinguished. Since each partitioning is associated with a probability of occurrence, we should thus select A_l such that the expected value of $IndgPartition(P)$ is minimized. However, this process is quite impractical since the number of partitionings are exponential in $|A' \cup \{A_l\}|$, i.e., exponential in $k' + 1$. We thus chose a simpler approach, by assuming that there are only two partitionings, the finest, as well as the coarsest. The probability of occurrence of the coarsest partitioning is $p(coarse) = \prod_{A_s \in A' \cup \{A_l\}} (1 - p_s)$. Thus, we select A_l that minimizes

$$IndgPartition(P_{coarse}(A' \cup \{A_l\}))p(coarse) + \\ IndgPartition(P_{fine}(A' \cup \{A_l\}))(1 - p(coarse))$$

2.2.5 Implementation

We have implemented our algorithms by modifying scalable decision tree frameworks Rainforest [21]. While Rainforest [21] aims to identify a class of tuples efficiently for a large data set, our task here is to identify each tuple. Since there is no class variable associated with the database, we may imagine that each tuple consists of its own unique class. Precisely, we can assume At every leaf node of the partially built tree, a single scan of the database partition associated with that node can be used to score each tuple and simultaneously and incrementally compute $Indg(A_l, D)$ for all facets A_l, and eventually the most promising facet is selected.

For the case where the database is static and the search queries are provided beforehand, our proposed approaches can simply pre-compute the decision trees. However, when the search queries are initiated on-the-fly with a regular SQL-like query, then building faceted search interface would require us to build the tree online (or in realtime). For such cases, instead of building the complete tree immediately, we can stay in sync with the user while she is exploring the partially constructed tree, and build a few “look ahead” nodes at a time. Finally, in the highly dynamic scenario where the database is frequently updated, a simple solution is to persist with the decision tree created at the start of the search, except that if a path through the tree terminates without a tuple being distinguished, the algorithm can then ask the remaining attributes in decreasing order of attribute probability until either the tuple gets distinguished or we run out of attributes. Thus, a fresh construction of the decision tree can be deferred to reasonable intervals, rather than after each update to the database.

2.3 FSRank - Faceted Search in Conjunction with Ranking Functions

In this section we develop faceted search procedures that can work in conjunction with ranking functions. Given a query Q, a ranking function typically assigns relevance scores $S(Q, t)$ to all selected tuples t, and a ranked-retrieval system will score and return only the top-n' tuples where $n' << n$. Developing ranking functions for database search applications is an active area of research, and ranking functions range from simple distance-based functions to probabilistic models (see [16, 25]). But in this thesis we shall treat such ranking functions as "black boxes"; thus our methods are aimed at very general applicability.

Our facet selection algorithm calls one such "black box" ranking function at every node in the decision tree during its construction and uses the ranked scores of the returned tuples as inputs to the facet selection algorithm. However, because the ranking function is a black box, it is challenging to develop methods for facet selection that are theoretically rigorous. In our approaches, we shall make one assumption: that the scores are normalized so that they are (a) positive, and (b) $\sum_{t\ selected\ by\ Q} S(Q, t) = 1$. In other words, the ranking function can be imagined as inducing a non-uniform "probability distribution" over the selected tuples, such that $S(Q, t)$ represents the probability that tuple t is preferred by the user. Of course, in the case that scoring functions are derived from probabilistic IR as well as language models, this assumption is justifiable. In the case of more ad-hoc ranking functions (such as distance-based, or vector-space models popular in IR), this assumption is perhaps a stretch. However, other than this specific assumption, we strive to be as principled as possible in our approaches.

From a faceted search perspective the task is to select the facet that directs the user towards the most preferred tuples (according to the ranking function) as efficiently as possible. One interesting complication is that these tuple preferences may change as the faceted search progresses; this is because as new attribute information is provided by the user, the ranking function may re-evaluate the scores of the remaining tuples still in contention. As an example, consider the car buyer who starts her search with an initial query Q = “Mileage = low AND Age = recent AND Car Type = sedan”. Suppose a ranking function when applied to such cars ranks cars with good reliability ratings the highest. After this initial query, a faceted search process starts which allows her to drill down further into the query results. But as the faceted search progresses, the buyer could select attribute values that may cause the ranking function to rank the remaining cars differently. For example, if the user also desires a “powerful engine” (i.e., the query has now been extended to Q = “Mileage = low AND Age = recent AND Car Type = sedan AND Engine Power = high”) then the ranking function may score cars with top speeds higher over good reliability. Thus a faceted search system in conjunction with a ranking function offers the benefits of focused retrieval as well as drill-down flexibility.

Defining the Cost of a Decision Tree: Given the above discussion, the cost of a specific decision tree T becomes more complicated than the corresponding definition in Section **??** where no ranking function was assumed. Consider a database D selected by an initial query Q, and consider a decision tree T with each tuple of D at its leaves. We will thus derive a formula for $cost(T, Q)$. Note that Q needs to be a parameter in the cost, as the ranking function uses Q to derive preference probabilities for each tuple. Note that in this cost definition we are not considering attribute uncertainties.

Let the root of the tree select the facet A_l. The root partitions D into the sets $D_{x_1}, \ldots, D_{x_{|Dom_l|}}$ where D_{x_q} is the set that satisfies the query $Q \wedge (A_l = x_q)$ for each $x_q \in Dom_l$. Let the corresponding subtrees for each of these partitions be $T_{x_1}, \ldots, T_{x_{|Dom_l|}}$. Clearly $cost(T_{x_q}, Q \wedge (A_l = x_q))$ is the (recursive) cost of each subtree. The quantity $\sum_{t \in D_{x_q}} S(Q,t)$ is the cumulative probabilities of all tuples in D_{x_q} and represents the probability that when the user is at the root, she will prefer any of the tuples in D_{x_q}. Thus we have

$$cost(T,Q) = \sum_{x_q \in Dom_l} \sum_{t \in D_{x_q}} S(Q,t) \times (cost(T_{x_q}, Q \wedge A_l = x_q) + 1) \tag{2.3}$$

It is easy to see that if no ranking functions are assumed, i.e., each tuple is uniformly preferred by the user, the cost of a tree reduces to the definition in Section 2, i.e., $\sum_{t \in D} ht(t)/n$. Our task is then the following: *Given an initial query Q that selects a set of tuples D, to determine a tree T such that $cost(T,Q)$ is minimized.* Since the problem is NP-Hard even without a ranking function, this problem too is intractable.

2.3.1 Facet Selection Algorithms

We develop a greedy heuristic that is motivated by our facet selection approaches presented in Section 2. Assume that we are at a particular node v of the decision tree. Let Q be the current query at that node. Thus Q is the initial query at the root, concatenated (i.e., AND'ed) with all conditions along the path from the root to v. Let D be the set of tuples of the database that satisfy Q. For any attribute A_l we can define a function $Indg(A_l, D)$ as follows:

$$Indg(A_l, D) = \sum_{x_q \in Dom_l} \left(\sum_{t_i, t_j \in D_{x_q}, i<j} S(Q,t_i) \times S(Q,t_j) \right) \tag{2.4}$$

The rest of the algorithm for selecting a single facet, even considering attribute uncertainty, is exactly the same as in Algorithm 1, except that Line 12 of Algorithm 1 is replaced selecting the attribute A_l that minimizes the expected value of Equation 2.4. The extensions to selecting k-facets, or building a fixed k-facet interface are similarly straightforward, and details are omitted from this version of the thesis.

2.3.2 Comparing Against Other Attribute Selection Procedures

In a recent thesis [12], algorithms were described that automatically select attributes of the results of a ranking query. Several selection criteria were examined, with the overall objective of attempting to select attributes that are most "useful" to the user. Attributes are consider most useful if, when the database is projected only on these attributes, the ranking function will re-rank the tuples in almost the same order. By listing the useful attributes, the motivation was to provide the end user the reasons why the top tuples were ranked so high. While such attribute selection algorithms can be used for faceted search, the following lemma shows that they do not necessarily achieve our goal of minimizing effort during the drill-down process.

Lemma 2.3.1. *Given a query Q that selects a set of tuples D, and a scoring function $S()$, the decision tree constructed by selecting facets that minimize $Indg()$ may be different from the decision tree constructed by selecting facets according to the* Score-Based *and* Rank-Based *attribute selection algorithms in [12].*

Proof (sketch): We sketch the proof by describing an example. Consider a cars database, and assume a ranking function exists, such that when a user poses an initial query for cars available in Texas, it ranks cars with air-conditioners very high. The ranking function assigns scores of 1 to the latter cars, and 0 to the rest. Both the Score-Based and Rank-Based algorithms in [12] will select the Boolean attribute AirCon as the most influential attribute. However, our minimum effort driven approach would not prefer to select the AirCon attribute. This is because all cars with air-conditioners will have high scores and will group together to produce a rather high value for $Indg(AirCon)$. In contrast, consider another attribute such as AutoTrans, which splits the total tuple set such that the highly ranked cars are evenly divided into each group. It is easy to see that $Indg(AutoTrans)$ is smaller than $Indg(AirCon)$ and hence more preferable.

Basically the attribute AirCon does not really help in further narrowing down the highly ranked tuples, because of the correlation with Texas cars via the ranking function. Offering some other facet such as AutoTrans will help the user narrow down the tuples more efficiently. Our experiments corroborate this observation in general.

2.3.3 Implementation

Although we assume that we are provided with a black box scoring function $S(Q, t)$, the way such a scoring function is implemented greatly affects the performance of our attribute selection algorithms. We define *single-result interface* for the ranking black box which is supported by previous works [12] on top-k computations. The *single-result interface* $S(Q, t)$ takes as input a query Q and a tuple t and outputs the score of the tuple. This interface incurs unit cost.

2.3.3.1 Facet Selection using Single Result Interface:

A scalable implementation of facet selection (Equation 2.4) using the single result interface is straightforward using ideas from the Rainforest framework [21]. We point out that even though the definition of $Indg()$ appears to require a quadratic-time algorithm, it can be computed in a single linear scan *Full Scan Algorithm*. The extensions to selecting k-facets as well as designing a fixed k-facet interface are straightforward. The extensions to include attribute uncertainties, k-facet selection, as well as designing a fixed k-facet interface are straightforward and omitted.

2.3.3.2 Implementation of Facet Selection using Pipelining Interface - Early Stoppage:

Furthermore, we explore interesting and novel techniques by which the performance of the facet selection algorithms in [4, 5] can be improved even further. To be truly effective, faceted search algorithms have to respond rapidly and without delay during an interactive session with an end user. The Full Scan algorithm presented in [4, 5], while better than any naive strategy, still suffered from high CPU cost and slow response time, as selecting the best attribute at each node required extensive calculations involving the database partition.[2] In this thesis, we propose techniques to improve the performance of the facet selection algorithms by reducing CPU intensive computations. The main idea is a novel adaptation of the early stopping techniques used in the TA-family of algorithms for top-k computations [13, 14, 15]. Such techniques can attain early termination that avoid scanning and scoring the complete database in determining the next most promising facet. In addition, as an even faster alternative, we propose an approximate facet selection technique that is guaranteed to stop after reading a fixed number of tuples and return the most promising facet discovered thus far.

Two types of faceted search problems on databases have been considered so far [4, 5]: (i) Faceted Search as an Alternative to Ranked-Retrieval and (ii) Faceted Search that Leverages Ranking Functions (referred to as FSNoRank and FSRank respectively in this theis). Essentially the second problem assumes that a ranking/scoring function is available that describes the preferences of the user for each tuple in a partition (see [4, 5] for more details). In this section, we focus on improving response time of the facet selection algorithms, by leveraging early stopping techniques from top-k algorithms. This part of our work has ap-

[2] The I/O cost is typically not an issue, as with the latest advent of semiconductor technology, even an inexpensive personal computer can often store an entire database partition associated with a decision tree node in main memory. It is the computational costs that are more critical to attain real-time response during interactions with an end user.

peared in [6].

Exact Indg() Calculation Based On Top-k Computation In general, top-k algorithms operate on index lists corresponding to a query's elementary conditions and aggregate scores monotonically for result candidates. The objective is to terminate the index scans as early as possible based on lower and upper bounds for the scores of result candidates. Motivated by such early stopping techniques employed in top-k algorithms, we wish to determine the best facet (or the best set of k facets) at every step of faceted navigation without performing a complete database partition scan.

In this thesis, we assume that the ranking function in the FSRank problem is accessible via a *pipelining interface*, which is natural and supported by previous works on top-k computation such as [13, 14, 15]. The *pipelining interface* $S(Q, D)$ takes as input a query Q and a database D and outputs a stream of tuples ranked descending according to $S(Q, t)$ along with their scores. The cost incurred in using this interface is the number of tuples retrieved (we can stop retrieving tuples at any time).

The high-level idea of early stopping is as follows: while scanning the database partition D_1, we consume tuples in some sequence and maintain a lower and upper bound for the value of $Indg(A_s, D_1)$ for each attribute A_s, and stop as soon as we discover an attribute A_l whose upper bound is no larger than the lower bound of all other attributes.

Lemma 2.3.2. *Given a database D with n tuples and m attributes, a CPU speedup of $n/(r + (n - r)/m)$ over the Full Scan algorithm can be obtained, if only r ($r \leq n$) tuples are consumed before the next best facet can be determined.*

Proof Sketch: The Rainforest implementation of Full Scan requires $n \times m$ update operations to compute $Indg()$ and return the best facet to the user (along with its domain information). Using the pipelining interface, if the best facet is determined after reading r tuples, then the total number of update operations required to suggest the best facet to the user (along with its domain information) is $(r \times m) + (n - r)$, where the first term refers to the update cost of processing the first r tuples, and the second term refers to the update cost of processing the remaining tuples, where only the counts for the selected attribute are updated. Therefore the speedup is $n/(r + (n - r)/m)$. □

As an illustrative example, for a database partition containing $200k$ tuples and 50 attributes, if only 20% of the tuples are consumed before termination, then the CPU speedup over Full Scan is $200k/(40k + 160k/50) = 4.6$.

We next discuss the FSRank case in detail. Assume that the pipelined interface has already scored r tuples, and let D_r be the set of tuples with the highest scores. Let the score of the rth tuple be S_r. For each attribute A_s, we maintain the following two quantities:

$$LowerIndg(A_s, D) = Indg(A_s, D_r) \tag{2.5}$$

$$\begin{aligned} UpperIndg(A_s, D) = LowerIndg(A_s, D) + \\ (n - r)S_r \times \max_{x_q \in Dom_s} \left\{ \sum_{t \in D_r, t[s] = x_q} S(Q, t) \right\} + \\ S_r \times S_r \times (n - r)(n - r - 1)/2 \end{aligned} \tag{2.6}$$

The lower bound of $Indg()$ reflects the minimum score that attribute A_s can get,i.e., it assumes that the rest $(n - r)$ tuples will not contribute anything to the score. This implies that each tuple that is not read yet has a unique domain value under attribute A_s. Therefore, $LowerIndg(A_s, D) = Indg(A_s, D_r)$.

On the other hand, the upper bound of $Indg()$ captures the maximum cumulative score attribute A_s can attain from the rest $(n - r)$ tuples by considering that the rest $(n - r)$ tuples have the same score S_r and can be paired with the largest subset of already read r tuples with the same domain value. This implies, if x_q is the largest domain value of attribute A_s so far, then the domain value of attribute A_s for the rest $(n - r)$ tuples is also x_q. Hence the $UpperIndg(A_s, D)$ formula contains the extra score accumulated by pairing these $(n - r)$ tuples with score S_r with each other and adding it up with the scores of the pairs formed by each (n-r) tuples with score S_r and each tuple in the largest subset of r tuples with domain value x_q and score $\geq S_r$. Quantity $S_r \times S_r \times (n-r)(n-r-1)/2$ in the $UpperIndg(A_s, D)$ formula captures the score accumulated by pairing each $(n-r)$ tuples with each other, whereas the quantity $(n-r)S_r \times \max_{x_q \in Dom_s} \left\{ \sum_{t \in D_r, t[s]=x_q} S(Q,t) \right\}$ contains the scores obtained by pairing each $(n-r)$ tuples with each tuple in the largest subset of r tuples with domain value x_q.

The pipelining interface consumes tuples and maintains these bounds, and stops when it discovers an attribute A_l whose upper bound is no larger than the lower bound of all other attributes. We refer to this as Exact FSRank Algorithm.

Algorithm 2: Exact FSRank (D, A')

```
Input: a database $D$ with $n$ tuples, a subset $A' \subset A$ attributes not yet used
Output: Attribute $A_1 \in A'$ that minimizes the $Indg()$ value.
**if** $|D| = 1$ **then**
    Return a tree with any attribute $A_l \in A'$ as a singleton node
**if** $|A'| = 1$ **then**
    Return a tree with the attribute in $A'$ as a singleton node
Read the first tuple.
Set $r = 2$;
**while** $r <= n$ **do**
    Read the $r$-th tuple.
    **for** each $A_l \in A'$ **do**
        $LowerIndg(A_1, D) = Indg(A_1, D_r)$
        $UpperIndg(A_1, D) =$
        $LowerIndg(A_1, D) +$
        $(n-r)S_r \times \max_{x_q \in Dom_s} \left\{ \sum_{t \in D_r, t[s]=x_q} S(Q,t) \right\} +$
        $S_r \times S_r \times (n-r)(n-r-1)/2$
    ChosenAttribute = Attribute with $argmin_{A_1 \in A'} UpperIndg(A_1, D)$
    **for** each $A_i \in A'$ and $A_i \neq ChosenAttribute$ **do**
        **if** $argmin_{A_1 \in A'} UpperIndg(A_1, D) \leq LowerIndg(A_i, D)$ **then**
            Continue;
        **else**
            $r = r + 1$
            Return to the while loop;
    return attribute with $argmin_{A_1 \in A'} UpperIndg(A_1, D)$ as root;
```

The Exact FSNoRank Algorithm is very similar, except that the upper and lower $Indg()$ for each attribute A_s is computed as follows:

$$LowerIndg(A_s, D) = Indg(A_s, D_r) \tag{2.7}$$

$$\begin{multline}UpperIndg(A_s, D) = LowerIndg(A_s, D) + \\ [(n-r) + \max_{x_q \in Dom_s} \{D_{x_q}\}]/2 \\ \left\{(n-r) + \max_{x_q \in Dom_s} \{D_{x_q}\} - 1\right\} \tag{2.8}\end{multline}$$

Here the pipelining interface outputs tuples in any arbitrary order (since there is no ranking function), and stops when it discovers an attribute A_l whose upper bound is no larger than the lower bound of all other attributes. We refer to this as the Exact FSNoRank Algorithm.

The $LowerIndg(A_s, D)$ calculation in the FSNoRank case follows similar explanation as for the FSRank case, except for the score of each tuple is assumed to be 1 here. Also, the score of each tuple is assumed to be 1 in the $UpperIndg(A_s, D)$ calculation, and the upper bound simply captures the score that can be attained by pairing each of the rest $(n-r)$ tuples with the largest subset of the r tuples with domain value x_q.

Although our attribute selection algorithms can work for any black box scoring function $S(Q, t)$, the score distribution across the tuples greatly affects the performance of our algorithms, since it determines which algorithms are feasible and efficient. A highly skewed scoring function - where the top few tuples have large scores, followed by a rapid degradation in score values for the remaining tuples - is most effective in making the Exact FSRank algorithm very efficient. This unfortunately does not apply in the case of Exact FSNoRank, because there is no ranking function to be leveraged. In fact as the lemma below shows, no matter what is the database, more than half of the database has to be *always* scanned by the FSNoRank algorithm before the best attribute can be determined.

Lemma 2.3.3. *For FSNoRank, even in the best case more than half of the database partition has to be scanned using the pipelining interface before the best facet can be determined.*

Proof Sketch: Consider a simple case, where a database D with n (n is even) tuples has only two attributes A and B. Let us assume that already $n/2$ tuples have been read, and the best scenario of early stopping has occurred so far in D, i.e., attribute A has returned $n/2$ different domain values $a_1, a_2, \ldots, a_{n/2}$, while attribute B repeats the same domain value b_1 in all $n/2$ tuples. Then we have, $LowerIndg(A, D) = 0$ and $UpperIndg(A, D) = n(n+2)/8$. Similarly, $LowerIndg(B, D) = n(n-2)/8$ and $UpperIndg(B, D) = n(n-1)/2$.

At this stage, no stopping decision can yet be made considering the upper and lower bounds of the $Indg()$ values of the attributes, and we must continue the scan of D. □

Approximate Indg() calculation As an even faster alternative to the above algorithms, we can simply stop reading further tuples after a small fixed number of iterations (i.e., bounded r), and use the most promising facet discovered thus far. Such an algorithm is of course guaranteed not to exhaust all tuples in the database partition, but may not necessarily produce the facet with the minimum $Indg()$ value. However, this is a good approximation if r is reasonably chose. It is easy to observe that such an approximate $Indg()$ calculation can be applied to both FSNoRank and FSRank.

The extensions to include attribute uncertainties, k-facet selection, as well as designing a fixed k-facet interface are straightforward and omitted.

2.4 The DynaCet System

The architecture of DynaCet and the flow of information through the system is illustrated in Figure 2.4. The front-end of the system is a web-based user interface which enables user to build queries and provides navigational access into the database. The back-end consists of two components, the *Facet Component* and the *Ranking Component*. DynaCet is domain independent and requires read-only access to the underlying database, thus making it implementable over any database system.

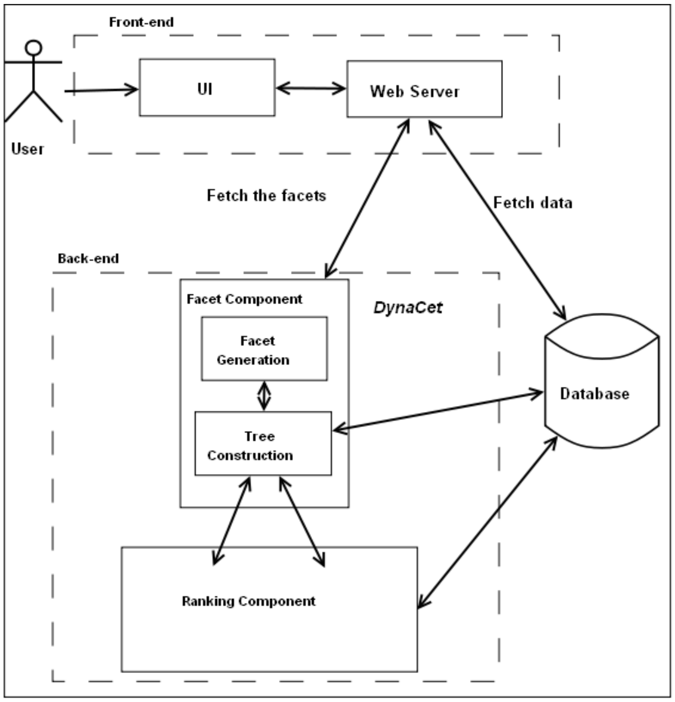
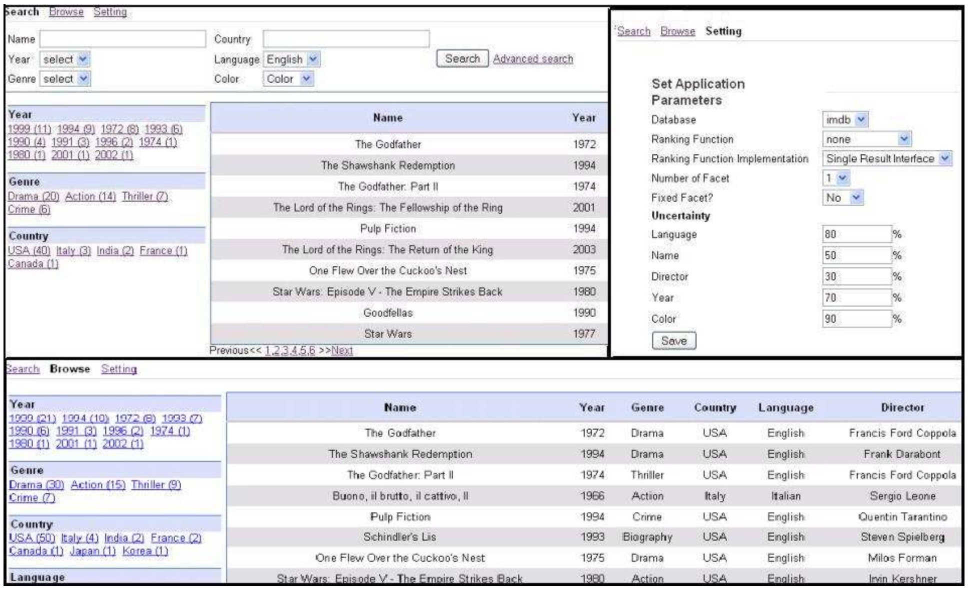


Figure 2.4. Architecture of DynaCet.

We have implemented our algorithms by leveraging the scalable decision tree framework Rainforest [21]. The *Facet Generation* module supports two modes of exploration over the facets - *Browse Only* and *Search and Browse*. In the *Browse Only mode*, a typical browsing session begins by showing suggested facets to the user. A user simply needs to select one of the facet values in order to move on to the next step in browsing. In this mode, the entire database is to be explored, hence the facet generation module uses pre-computed decision trees. However, for the *Search and Browse* mode, a more dynamic scenario is investigated. Here, a user can typically begin her search session by specifying one or more of her preferences in the form of a query. Next, the resultant tuple set is retrieved by DynaCet and faceted search is enabled on that set. Hence, in this case, decision trees are constructed



Name

Year select

Genre select

Country

Language English

Color Color

Year
1999 (11) 1994 (9) 1972 (8) 1993 (6)
1990 (4) 1991 (3) 1996 (2) 1974 (1)
1980 (1) 2001 (1) 2002 (1)

Genre
Drama (20) Action (14) Thriller (7)
Crime (6)

Country
USA (40) Italy (3) India (2) France (1)
Canada (1)

Name	Year
The Godfather	1972
The Shawshank Redemption	1994
The Godfather: Part II	1974
The Lord of the Rings: The Fellowship of the Ring	2001
Pulp Fiction	1994
The Lord of the Rings: The Return of the King	2003
One Flew Over the Cuckoo's Nest	1975
Star Wars: Episode V - The Empire Strikes Back	1980
Goodfellas	1990
Star Wars	1977

Previous<< 1,2,3,4,5,6 >>Next

Set Application Parameters

Database	imdb
Ranking Function	none
Ranking Function Implementation	Single Result Interface
Number of Facet	1
Fixed Facet?	No
Uncertainty	
Language	80 %
Name	50 %
Director	30 %
Year	70 %
Color	90 %

Year
1999 (21) 1994 (10) 1972 (8) 1993 (7)
1990 (6) 1991 (3) 1996 (2) 1974 (1)
1980 (1) 2001 (1) 2002 (1)

Genre
Drama (30) Action (15) Thriller (9)
Crime (7)

Country
USA (50) Italy (4) India (2) France (2)
Canada (1) Japan (1) Korea (1)

Language

Name	Year	Genre	Country	Language	Director
The Godfather	1972	Drama	USA	English	Francis Ford Coppola
The Shawshank Redemption	1994	Drama	USA	English	Frank Darabont
The Godfather: Part II	1974	Thriller	USA	English	Francis Ford Coppola
Buono, il brutto, il cattivo, Il	1966	Action	Italy	Italian	Sergio Leone
Pulp Fiction	1994	Crime	USA	English	Quentin Tarantino
Schindler's Lis	1993	Biography	USA	English	Steven Spielberg
One Flew Over the Cuckoo's Nest	1975	Drama	USA	English	Milos Forman
Star Wars: Episode V - The Empire Strikes Back	1980	Action	USA	English	Irvin Kershner

Figure 2.5. Screen shot of DynaCet GUI.

online over search results. Essentially, we build a partial tree with a few "look ahead" nodes and then stay in sync with the user while she is exploring the partially constructed tree. Each of these two above mentioned mode can also work in conjunction with a Ranking component, where the Ranking module imposes a *skew* over the user preferences for the selected tuples. Different problem variants of DynaCet are discussed in more detail in [**?**].

Browse Only Mode: In this model, the user does not initiate the search with a query - rather DynaCet will recommend facet(s) for her. Consequently, a user is shown m different facets, to which she responds by selecting a value from one of the facet domain (or a ”don't know"). Depending upon the user response, the next set of facets are dynamically suggested and the process repeats. The lower half of Figure 2.5 shows the interface from a typical browsing session over IMDB using DynaCet.

This model takes the advantage of a pre-computed decision tree and thus results in good response time. We will also provide a comparative evaluation of our proposed solutions with some existing attribute selection techniques in the demonstration. The user will be allowed to choose the m-Facets or the Fixed m-Facets algorithm in this mode.

Search and Browse Mode: In this mode the user will be able to start the exploration phase by providing a query through the form interface. The upper left part of Figure 2.5 shows the interface for querying. The query form shows only a few attributes from among the total set of attributes. A larger set can be seen by going to *Advanced Search*. In the figure, the user has asked for movies with Language="English" and Color="Color" which then results in only three dynamically generated facets being shown to the user.

Faceted Search in Conjunction with Ranking Function: The user will have an option of choosing an appropriate ranking function from the list of available ranking functions (provided as a drop-down list). A comparative cost evaluation between our proposed solution and a prior attribute ordering method [12] will also be shown.

2.5 Evaluation

In this section we describe our experimental setup, our different results of facet selection algorithms (without and in conjunction with ranking functions) and draw conclusions on the quality and performance of the techniques. We validated the quality of the our solutions by measuring *cost*, which is defined as the average number of user interactions (i.e., number of attributes or facets selected) before the desired tuple is identified. Experiments evaluating the time complexity of the node creation step of our tree building algorithms were also conducted. This measure is especially relevant for exploratory interactive users and hence a fast scalable implementation is desirable. In case the trees can be built in a preprocessing step, this measure is less critical. We also implemented several existing attribute selection techniques to compare against our approaches. Evaluation results clearly show that our solutions perform significantly better.

All implementation is done using Java and C# and the evaluations performed on a Windows XP machine with 3.0Ghz Intel Xeon processor and 2GB RAM.

Database Used: We evaluated our methods using two data sets, *IMDB movie database*[3] - a real world movie database accessible over the internet and *Yahoo Autos*[4], a online used-car listing database. Using the IMDB database, we generated a single movie database containing about $234,000$ tuples with 19 attributes including null values in some fields. Similarly, we built a car database with 43 attributes and more than $40,000$ tuples. We also generated a large synthetic dataset having nearly 10 million rows and 100 attributes from the car dataset by maintaining the original distribution of the dataset.

Uncertainty Model: As we discussed in Section 2.2.2, we use external knowledge about user uncertainty for ranking the attributes of our databases. For our evaluation, we organized a small survey among 20 randomly selected users comprising students and faculty members. In the survey, each person was asked to assign a value (between 0 to 1) for each attribute in the IMDB movie database. This value denotes the likelihood (probability) with which the user thinks she would be able to answer a question over that attribute. We took average probability scores for all attributes in our evaluation. Note that the question of whether the survey accurately reflects the true uncertainty model for the user population at large is an orthogonal problem, and is not extremely relevant for our purposes. The survey was conducted merely to obtain uncertainty values that are somewhat realistic for the related domain. Developing techniques for ascertaining uncertainty values is a future direction of research.

[3] http://www.imdb.com

[4] http://autos.yahoo.com

2.5.1 FSNoRank Experiments

In this set of experiments all tuples were considered equally desirable to the end user as no ranking function was assumed. We conducted evaluations to check the quality and robustness of the algorithms we developed.

2.5.1.1 Quality Evaluation

In this subsection, we briefly explain the three different quality experiments we performed and draw inferences. These experiments measure cost as defined in previous section, which is the average number of queries that needs to be answered before the user arrives at a desired tuple (i.e., effort).

Cost versus varying attribute probability: The intrinsic assumption in our decision tree modeling is the user's inability to answer all the questions. This experiment infers the influence of the probability of an attribute in determining cost.

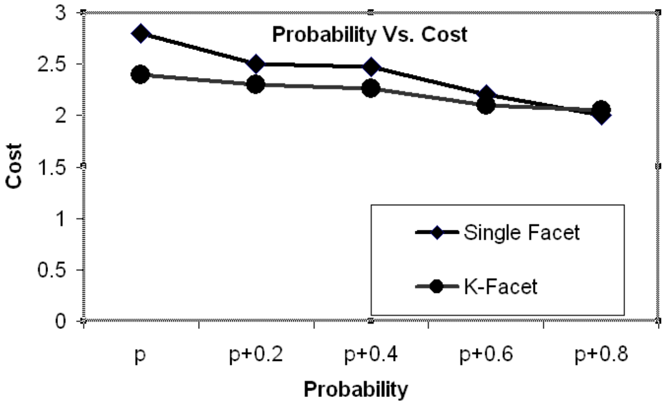
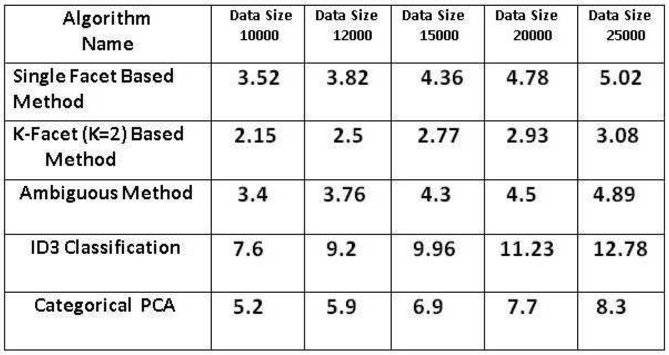


Figure 2.6. Change Of cost with varying probability.

As shown in Figure 2.6, we compare the cost of the Single-Facet search with the k-Facets based search algorithm by varying the uncertainty model. In our evaluation we set the value of k top $= 2$. We varied the probability of each attribute in increments of 0.2 in this experiment. As the graph suggests, with higher probability, the cost decreases for both the algorithms. This observation corroborates our basic intuition of considering probability of the attributes in the decision tree construction.

Cost versus varying database size: In this set of experiments, we vary the database size (auto database) and compare the costs of the Single-Facet and the k-Facets based search algorithms. As can be seen from Figure 2.7, the cost is more for the Single-Facet algorithm. Also, in both cases, costs increase with increasing database size. The reason being, with an increase in the number of tuples, more questions are needed to distinguish them.



Algorithm Name	Data Size 10000	Data Size 12000	Data Size 15000	Data Size 20000	Data Size 25000
Single Facet Based Method	3.52	3.82	4.36	4.78	5.02
K-Facet (K=2) Based Method	2.15	2.5	2.77	2.93	3.08
Ambiguous Method	3.4	3.76	4.3	4.5	4.89
ID3 Classification	7.6	9.2	9.96	11.23	12.78
Categorical PCA	5.2	5.9	6.9	7.7	8.3

Figure 2.7. Change of cost with varying database size.

Comparing against existing techniques: We compared the cost of facets suggested by our methods with that suggested using the *Indg()* method developed in [20] (named as Ambigous method in Figure 2.7, PCA for categorical data and the $ID3$ classification algorithm. Note that these three algorithms assume all values are known to the user and so do not have any provision for handling *uncertainty* in user response. Hence, these techniques are principally different from our facet selection algorithms. Howeve, from Figure 2.7, it is clear that the Indg() based method clearly outperforms the ID3 and PCA. Since, both Single-Facet and K-Facet algorithms are richer than Indg() and also show better performance, we can therefore claim that our techniques are better than existing facet (attribute) selection methods.

2.5.2 Performance Evaluation

We implemented the scalable Rainforest [21] framework to construct the decision tree. We vary two parameters (number of tuples and number of attributes) and measure the average node creation time. As seen from the Figure 2.13 and Figure 2.12, average node creation time increases with the increase of dataset size/ width. We point out that that the objective of our decision tree is to distinguish each tuple (in contrast to identifying a class of tuples). Hence, the depth of this tree is much larger than the normal decision trees used for classification problems. Consequently, this leads to a proportional increase in creation time.

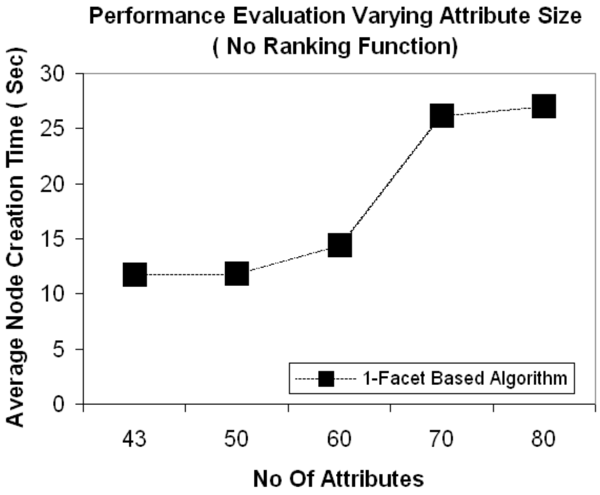


Figure 2.8. Change of average node creation time with varying number of attributes.

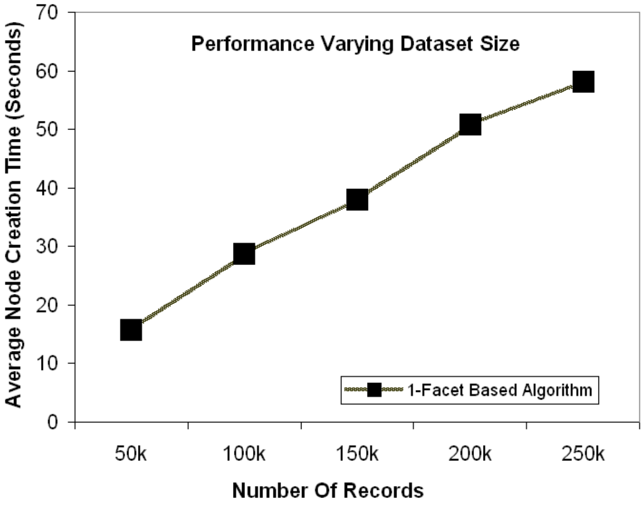


Figure 2.9. Change of average node creation time with varying number of tuples.

2.5.3 FSRank Experiments

In this section, we explain our experiments on facet selection algorithms in conjunction with ranking functions. We assume the presence of a ”black box” ranking function which simply contributes skewness towards the preference of tuples. Consequently, the solution cost is computed as described in Section 2.3.

Ranking Function: Design of an efficient and effective ranking function is an orthogonal research problem and is not our focus here. For practical purposes, however, we implement a simple ranking function where a tuple t gets a score equal to the square of its Euclidian distance from the centroid of the residual database partition. We further normalize this squared distance to a non-uniform probability distribution over the selected tuples, such that $S(Q,t)$ represents the probability that tuple t is preferred by the user, and that $\sum_{t\ selected\ by\ Q} S(Q,t) = 1$.

2.5.3.1 Quality Evaluation

In this experiment, we compared the cost of our Single Interface algorithm with the existing rank based attribute selection technique [12] on IMDB data. As seen from the Figure 2.10, our Single Interface facet selection technique performs better than existing approach.

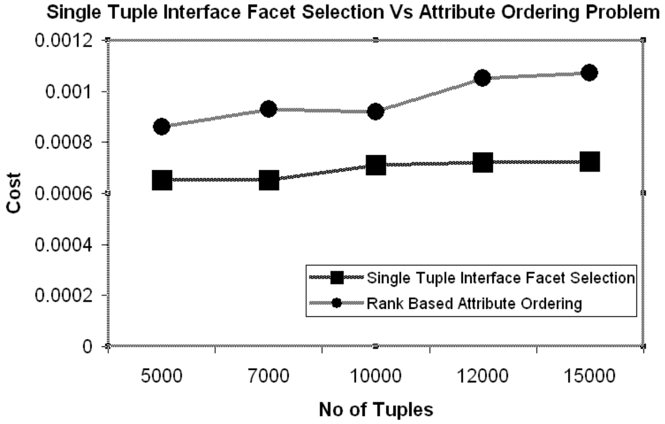


Figure 2.10. Comparison of Cost - Facet Selection and Attribute Ordering Problem.

Next, we evaluate how r (number of tuples are to be read before the pipelining interface is terminated to make the selection of the attribute) affects the quality of Approximate FSRank Algorithm.We vary the parameter r here. As expected, by deciding r in advance, we lose quality (i.e., increase the average navigation cost) as a trade off to the performance. However, the navigation cost decreases as r increases. An interesting problem here can be to find an optimal r value for a given dataset. This concludes our discussion on experiments.

2.5.3.2 Performance Evaluation

As discussed earlier, we implement the scalable Rainforest [21] framework to construct the decision tree. We vary two parameters (number of records and number of at-

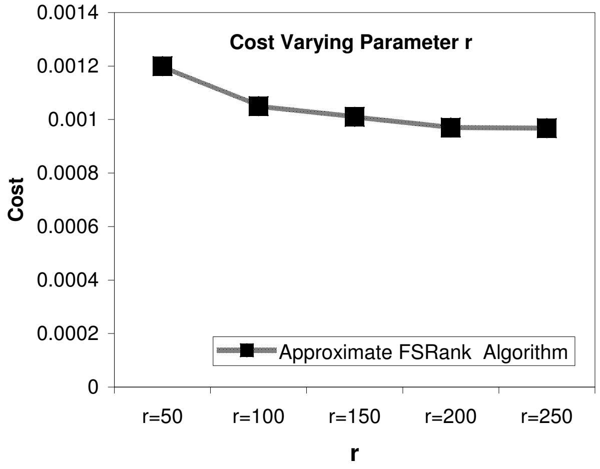


Figure 2.11. Change of Cost Varying r in Approximate FSRank Algorithm.

tributes) and measure the average node creation time. As seen from the Figures 2.13 and 2.12, average node creation time increases with the increase of dataset size/ width. We point out that that the objective of our decision tree is to essentially identify each tuple unambiguously (in contrast to identifying a class of tuples). Hence, the depth of this tree is much higher than the normal decision trees used for classification purposes in Machine Learning problems.

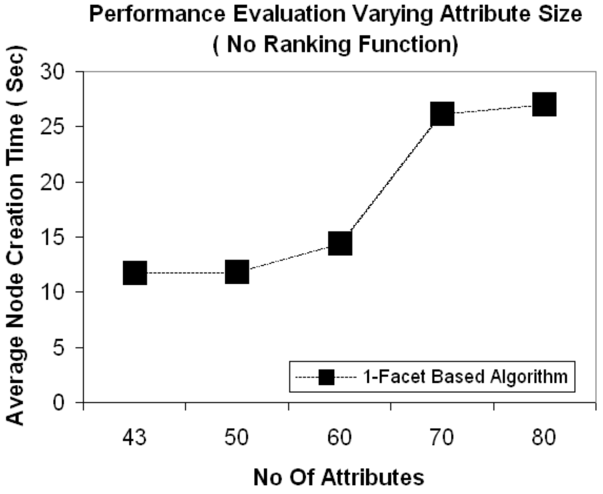


Figure 2.12. Change of average node creation time varying attribute size.

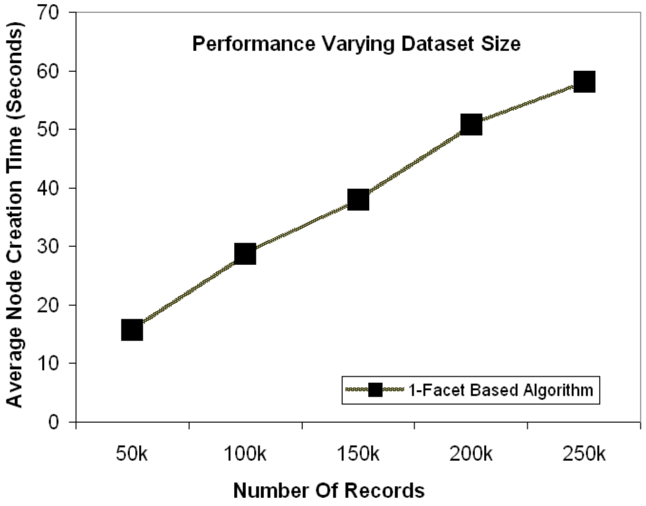


Figure 2.13. Change of average node creation time varying dataset size.

Next, we vary database size and observe the performance of three different algorithms. Performance is evaluated among the FSRank Full Scan Algorithm, the Exact FSRank Algorithm and the Approximate FSRank ($r = 100$) Algorithm.

Figure 2.14 corroborates our claim - the average node creation time can be significantly improved in Exact FSRank Algorithm compared to the FSRank Full Scan Algorithm. The Approximate FSRank is the fastest, but it comes with a loss in quality - the navigation cost is sometimes more.

2.6 Related Work

The traditional design goal of faceted search interfaces [19, 26, 2, 27, 28] is to offer users a flexible navigational structure, targeted towards text and/or image data. There have been recent efforts at creating a faceted search interface over structured database, e.g., [29], as well as heterogeneous collections [30]. The former is typically designed for specific applications by domain experts. In our work, we aim to propose a domain independent solution for automatically generating facets. In [30], the focus is on computing correlated

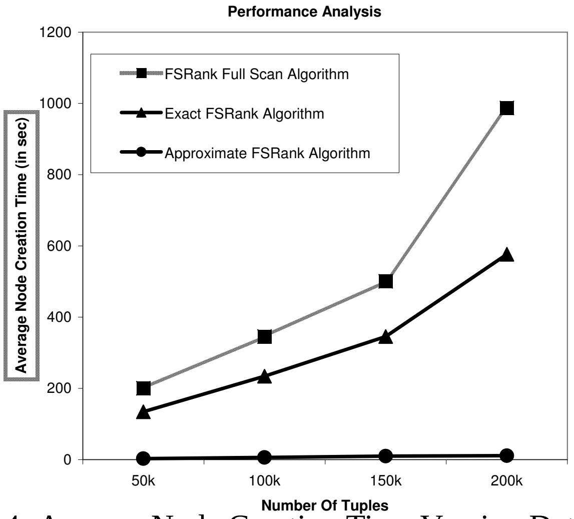


Figure 2.14. Average Node Creation Time Varying Dataset Size.

facets and using them to aggregate and present related information to the user. This appears to be different from our problem, where the focus is minimum effort drill-down.

Our work bears resemblance to the problem of generating automatic categorization of query results [31]. Our developed approach differs from this prior work along several key dimensions: (a) our proposed approach considers uncertainty models, (b) our approach is decision-tree based and depends on user interaction , and (c) our algorithms can work in conjunction with available ranking functions.

Decision trees and classical *Information Value Theory* [32] are widely studied class of techniques in machine learning [22]. However such models require explicit knowledge of each of the user decision models which is not present in our model. A recent work [20] uses decision trees for fast tuple identification in databases. Our proposed decision tree model captures user inability to answer certain attributes as well as the ability to incorporate ranking functions, which marks the intrinsic difference between our approach and [20].

In this thesis we have attempted a mapping of the key ideas of PCA [23] to categorical data, and have compared it against other approaches for selecting facets.

Ranked retrieval in structured databases is an active research area [16, 17, 18, 33]. Recent research effort address the problem of keyword-based search techniques in databases combined with the power of aggregation in Online Analytical Processing(OLAP) systems [34]. This ranking metric is based upon “interestingness” of attributes. We propose an effort-based strategy in our work for enabling tuple search, and leverage external knowledge in the form of uncertainty models and ranking functions. In [12], algorithms were described that automatically select attributes of the results of a ranking query. As discussed in this thesis, while such attribute selection algorithms can be used for faceted search, they do not necessarily achieve our minimum effort goals.

Selecting the next facet based on a ranking function has connections with automatic query expansion studies in IR ([35, 36]. At some level automatic facet selection may be viewed as a similar problem, however while AQE techniques are largely empirical and target text collections, we make several new and important contributions involving structured data, black box ranking functions, as well as scalable algorithms based on modern top-k concepts.

Our Fixed k-Facets Based Search Problem has similarities with the classical problem of computing minimum and approximate keys and functional dependencies of database relations (see [37, 24]). Most problem variants are NP-complete, and popular algorithms are based on level-wise methods from data mining ([24]). However, in our case the problem is complicated by the fact that attributes are associated with uncertainties, thus such deterministic procedures appear difficult to generalize to the probabilistic case.

2.7 Conclusion

In this thesis we tackle the problem of effective minimum-effort based faceted search within structured data warehouses of business organizations. Our proposed technique uses uncertainty models of attributes in the structured database, as well as leverages the existence of ranked-retrieval models, and our solution framework is based on the novel top-k approaches to efficient decision tree construction. As our future work, we like to extend these approaches to work for multi-table databases, design methods for obtaining reliable uncertainly models from external data sources, and leverage heterogeneous data (e.g., text as well as structured) as well as rich meta-data (e.g., "non-flat" hierarchies) that naturally occur as part of modern data warehouses.

We also aim to explore other techniques, such as sampling, that can assist in expediting the response time of facet selection algorithms. Such techniques may be useful in approximating domain information of the attributes in a principled way, thus guaranteeing a reduced CPU cost while suggesting facets to the user. We would also like to perform a comparative quality evaluation of these various proposed techniques on a variety of real world datasets. In the future, we would like to conduct user studies to obtain user evaluations on our proposed speedup techniques.

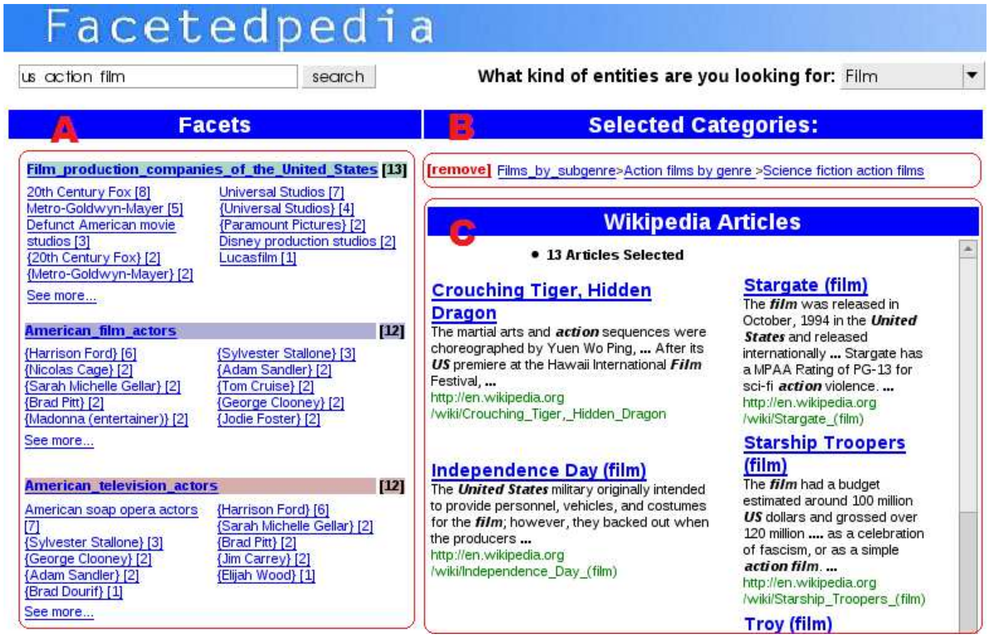
CHAPTER 3

FACETEDPEDIA: DYNAMIC GENERATION OF QUERY-DEPENDENT FACETED INTERFACES FOR WIKIPEDIA

3.1 Introduction

Wikipedia has become the largest encyclopedia ever created, with close to 3 million English articles by far. The prevalent manner in which the Web users access *Wikipedia* articles is keyword search. Keyword search has been effective in finding specific Web pages matching the keywords. Therefore it may well satisfy the users when they are causally interested in a single topic and use *Wikipedia* as a dictionary or encyclopedia for that topic. However, *Wikipedia* has now become a primary knowledge source for many casual users and even an integral component in the knowledge management systems of businesses for decision making. It is thus typical for a user to explore a set of relevant topics, instead of targeting a particular topic, for more sophisticated information discovery and exploratory tasks. With only keyword search, one would have to digest the potentially long list of search result articles, follow hyperlinks to connected articles, adjust the query and perform multiple searches, and synthesize information manually. This procedure is often time-consuming and error-prone.

One useful mechanism for information exploration is the *faceted interface*, or the so-called *hierarchical faceted categories* (HFC) [1]. A faceted interface for a set of objects is a set of category hierarchies, where each hierarchy corresponds to an individual *facet* (dimension, attribute, property) of the objects. The user can navigate an individual facet through its hierarchy of categories and ultimately a specific “property” value if necessary, thus reaching those objects associated with the categories and the value on that facet. The



us action film

What kind of entities are you looking for: Film

Film_production_companies_of_the_United_States [13]

20th Century Fox [8]
Metro-Goldwyn-Mayer [5]
Defunct American movie studios [3]
{20th Century Fox} [2]
{Metro-Goldwyn-Mayer} [2]
Universal Studios [7]
{Universal Studios} [4]
{Paramount Pictures} [2]
Disney production studios [2]
Lucasfilm [1]

American_film_actors [12]

{Harrison Ford} [6]
{Nicolas Cage} [2]
{Sarah Michelle Gellar} [2]
{Brad Pitt} [2]
{Madonna (entertainer)} [2]
{Sylvester Stallone} [3]
{Adam Sandler} [2]
{Tom Cruise} [2]
{George Clooney} [2]
{Jodie Foster} [2]

American_television_actors [12]

American soap opera actors [7]
{Sylvester Stallone} [3]
{George Clooney} [2]
{Adam Sandler} [2]
{Brad Dourif} [1]
{Harrison Ford} [6]
{Sarah Michelle Gellar} [2]
{Brad Pitt} [2]
{Jim Carrey} [2]
{Elijah Wood} [1]

[remove] Films_by_subgenre>Action films by genre >Science fiction action films

- 13 Articles Selected

Crouching Tiger, Hidden Dragon
The martial arts and ***action*** sequences were choreographed by Yuen Wo Ping, ... After its ***US*** premiere at the Hawaii International ***Film*** Festival, ...
http://en.wikipedia.org/wiki/Crouching_Tiger,_Hidden_Dragon

Independence Day (film)
The ***United States*** military originally intended to provide personnel, vehicles, and costumes for the ***film***; however, they backed out when the producers ...
http://en.wikipedia.org/wiki/Independence_Day_(film)

Stargate (film)
The ***film*** was released in October, 1994 in the ***United States*** and released internationally ... Stargate has a MPAA Rating of PG-13 for sci-fi ***action*** violence. ...
http://en.wikipedia.org/wiki/Stargate_(film)

Starship Troopers (film)
The ***film*** had a budget estimated around 100 million ***US*** dollars and grossed over 120 million .... as a celebration of fascism, or as a simple ***action film***. ...
http://en.wikipedia.org/wiki/Starship_Troopers_(film)

Troy (film)

Figure 3.1. The faceted retrieval interface of *Facetedpedia*.

user navigates multiple facets and the intersection of the chosen objects on individual facets are brought to the user's attention. The procedure hence resembles repeated constructions of conjunctive queries with selection conditions on multiple dimensions.

In this paper we propose *Facetedpedia*[1], a faceted retrieval system which is capable of dynamically generating query-dependent facets for a set of *Wikipedia* articles. We use the following example to further illustrate.

Example 1 (Motivating Example)**.** *Imagine that a user is exploring information about action films. The* Facetedpedida *system takes a keyword query, say, "us action film", as the input and obtains a ranked list of search result articles. It will create a faceted interface, as shown in Figure 3.1. The system dynamically derives k facets (region (A)) for covering the top s result articles. For instance, for "us action film", these dimensions (facets) can include Companies, Actors, and so on. Each facet is associated with a hierarchy of categories. Each article can be assigned to the nodes in these hierarchies, with each assignment*

[1] http://idir.uta.edu/facetedpedia/

representing an attribute value of the article. On each facet, the user can navigate through the category path which is formed by parent-child relationships of Wikipedia *categories.* [2] *The interface also shows the navigation paths (region (B)) and article titles (region (C)). When the user clicks one article title, the corresponding* Wikipedia *article would be shown. (This part of the interface is omitted.)*

Here in Figure 3.1 we only show three facets from the generated interface in region (A): (1) Film_production_companies_of_the_United _States; (2) American_film_actors; (3) American_television_actors. When the user selects any facet items for navigation in region (A), a user navigational path is added in region (B). Here we show only one path: Films_by_subgenre>Action_films_by_genre>Science_ fiction_action_films, which means the user selected facet root Films_ by_subgenre, then its subcategory Action_films_by_genre, and the subcategory of subcategory, Science_fiction_action_films. There are thirteen articles satisfying the chosen navigational paths, and they are shown in region (C). In this way, the user filters the large number of result articles and finds those matching her interests.

3.1.1 Overview of Challenges and Solutions

We study the problem of dynamic discovery of query-dependent faceted interfaces. Given the set of top-s ranked *Wikipedia* articles as the result of a keyword search query, *Facetedpedia* produces an interface of multiple facets for exploring the result articles.

We focus on *automatic* and *dynamic* faceted interfaces. The facets could not be precomputed due to the query-dependent nature of the system. In applications where faceted interfaces are deployed for relational tuples or schema-available objects, the tuples/objects are captured by prescribed schemata with clearly defined dimensions (attributes), therefore a query-independent static faceted interface (either manually or automatically generated)

[2] A *Wikipedia* article may belong to one or more categories. These categories are listed at the bottom of the article.

may suffice. By contrast, the articles in *Wikipedia* are lacking such pre-determined dimensions that could fit all possible dynamic query results. Therefore efforts on static facets would be futile. Even if the facets can be pre-computed for some popular queries, say, based on query logs, the computation must be automatic and dynamic. Given the sheer size and complexity of *Wikipedia* and its rapid growth, a manual approach would be prohibitively time-consuming and cannot scale to stay up-to-date. The main challenges in realizing *Facetedpedia* are summarized as follows:

Challenge 1: The facets and their category hierarchies are not readily available.

The concept of faceted interface is built upon two pillars: facets (i.e., dimensions or attributes) and the category hierarchy associated with each facet. The definition of "facet" itself for *Wikipedia* does not arise automatically, leaving alone the discovery of a faceted interface. Therefore we must answer two questions: (1) *facet identification*– What are the facets of a *Wikipedia* article?; and (2) *hierarchy construction*– Where does the category hierarchy of a facet come from?

Challenge 2: We need metrics for measuring the "goodness" of facets both individually and collectively.

We need to find facets useful for user navigation. A goodness metric for ranking the facets is needed. The problem gets even more complex because the utilities of multiple facets do not necessarily build up linearly– Since the facets in an interface should ideally describe diverse aspects of the result articles, a set of individually "good" facets may not be "good" collectively.

Challenge 3: We must design efficient faceted interface discovery algorithms based on the ranking criteria.

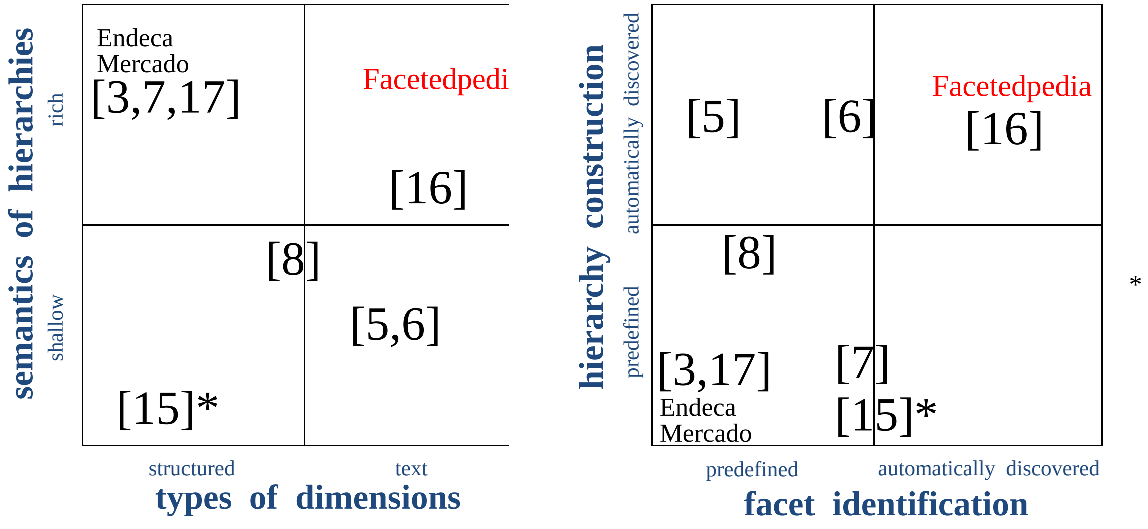
It is infeasible to directly apply the ranking metric exhaustively on all possible choices, due to the large search space. Furthermore, the interactions between the facets in a faceted interface make the computation of its exact cost intractable. Even computing the costs of individual facets without considering the interactions is non-trivial, given the size and the complexity of *Wikipedia*.

3.1.2 Summary of Contributions and Outline

- **Concept: Faceted *Wikipedia*.** We propose an automatic and dynamic faceted retrieval system for *Wikipedia*. To the best of our knowledge, this is the first system of its kind. The key philosophy of our approach is to exploit collaborative vocabulary as the backbone of faceted interfaces. (Section 3.3)
- **Metrics: Facet Ranking**. Based on a user navigation model, we propose metrics for measuring the "goodness" of facets, both individually and collectively. (Section 3.4)
- **Algorithms: Faceted Interface Discovery**. We develop effective and efficient algorithms for discovering faceted interfaces in the large search space. (Section 6.3)
- **System Evaluation: *Facetedpedia***. We conducted user study to evaluate the effectiveness of the system and to compare with alternative approaches. We also measured its quality and efficiency quantitatively. (Section 6.6)

3.2 Faceted Retrieval Systems: A Comparative Study

Faceted interface has become influential over the last few years and we have seen an explosive growth of interests in its application [38, 39, 1, 39, 1, 40, 41, 42, 43, 44, 45, 46, 47]. Commercial faceted search systems have been adopted by vendors (such as *Endeca*, *IBM*, and *Mercado*), as well as E-commerce Websites (e.g., *eBay.com*, *Amazon.com*). The utility of faceted interfaces was investigated in various studies [38, 1, 48, 39, 49, 48, 50, 1], where it was shown that users engaged in exploratory tasks often prefer such result grouping over simple ranked result list (commonly provided by search engines), as well as over alternative ways of organizing retrieval results, such as clustering [51, 52, 49].



(a) (b)

The work does not support hierarchy on facets.

Figure 3.2. Taxonomies of faceted retrieval systems, (a)Facet types and semantics, (b)Automation and dynamism.

In this section we present taxonomies to characterize the relevant faceted retrieval systems and compare them with *Facetedpedia*. Existing research prototypes or commercial faceted retrieval systems mostly cannot be applied to meet our goals, because they either are based on manual or static facet construction, or are for structured records or text collections with prescribed metadata. Very few have investigated the problem of dynamic discovery of both facet dimensions and their associated category hierarchies.

To the best of our knowledge, we are the first to propose a query-dependent faceted retrieval system for *Wikipedia*. *CompleteSearch* [53] supports query completions and query refinement in *Wikipedia* by a special type of “facets” on three dimensions that are very different from our notion of general facets: query completions matching the query terms; category names matching the query terms; and categories of result articles. Recently, a faceted *Wikipedia* search interface came out of the *DBPedia* [54] project around the same time as our work. The facets there appear to be query-independently extracted from common

Wikipedia infobox attributes, although the underlying method remains to be proprietary at this moment.

Figure 3.2(a): Taxonomy by Facet Types and Semantics

Previous systems roughly belong to two groups on this aspect. In some systems the facets are on relational data (e.g., *Endeca*, *Mercado*, [44]) or structured attributes in schemata (e.g., [39, 46, 47]) and the hierarchies on attribute values are predefined based on domain-specific taxonomies. The hierarchies could even be manually created, thus could contain rich semantic information. In some other systems a facet is a group of textual terms, over which the hierarchy is built upon thesaurus-based IS-A relationships (e.g., [40]) or frequency-based subsumption relationships between general and specific terms (e.g., [41, 42]). These systems cannot leverage as much semantic information. The work [45] is in the middle of Figure 3.2(a) since it has both structured dimensions and a subsumption-based topic taxonomy.

In contrast, *Facetedpedia* enables semantic-rich facet hierarchies (distilled from *Wikipedia* category system) over text attributes (hyperlinked *Wikipedia* article titles). In the absence of predefined schemata, it builds facet hierarchies with abundant semantic information from the collaborative vocabulary, instead of relying on IS-A or subsumption relationships.

Figure 3.2(b): Taxonomy by Degree of Automation and Dynamism

When building the two pillars in a faceted interface, namely the facet and the hierarchy, *Facetedpedia* is both automatic and dynamic, as motivated in Section 3.1.1. On this aspect, none of the existing systems could be effectively applied in place of *Facetedpedia*, because none is fully automatic in both facet identification and hierarchy construction.

In some systems (e.g., *Endeca*, *Mercado*, [44, 47, 39, 46]) the dimensions and hierarchies are predefined, therefore they do not discover the facets or construct the hierarchy. In [46, 44] a subset of interesting/important facets are automatically selected from the predefined ones. In [41, 42] the set of facets are predefined, but the hierarchies are automatically created based on subsumption. In [45] only one special facet (a topic taxonomy) is automatically generated and the rest are predefined.

With respect to the automation of faceted interface discovery, the closest work to ours is the *Castanet* algorithm [40]. The algorithm is intended for short textual descriptions with limited vocabularies in a specific domain. It automatically creates facets from a collection of items (e.g., recipes). The hierarchies for the multiple facets are obtained by first generating a single taxonomy of terms by IS-A relationships and then removing the root from the taxonomy.

3.3 Faceted Interface for Wikipedia by Collaborative Vocabulary

In discovering faceted interfaces for *Wikipedia,* the basis of our approach is to exploit its user-generated *collaborative vocabulary* such as the “grassroots” category system. Even internal *Wikipedia* hyperlinks are an instance of collaborative vocabulary in a broader sense, as they indicate the users’ collaborative endorsement of relationships between entities. The collaborative vocabulary represents the collective intelligence of many users and rich semantic information, and thus constitutes the promising basis for faceted interfaces. With regard to **the concept of facet dimension**, the *Wikipedia* articles hyperlinked from a search result article are exploited as its attributes. The fact that the authors of an article collaboratively made hyperlinks to other articles is an indication of the significance of the linked articles in describing the given article. This view largely enriches the semantic information associated with the result articles. With regard to **the concept of category hierarchy**, the *Wikipedia* category system provides the category-subcategory relationships between categories, allowing users to go from general to specific when specifying con-

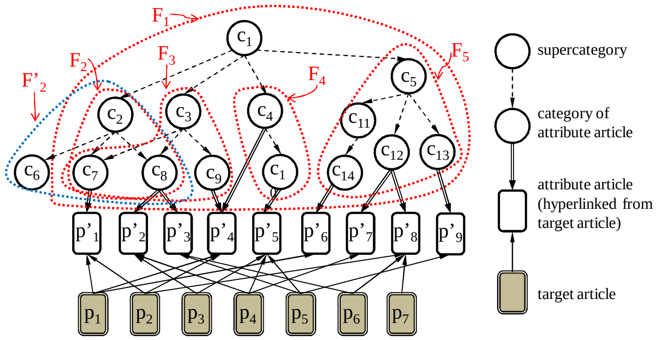


Figure 3.3. The concept of facet.

ditions. We now formally define the concepts in our framework and deliver the problem specification.

Definition 1 (Target Article, Attribute Article)**.** *Given a keyword query q, the set of top-s ranked* Wikipedia *articles, $\mathcal{T}=\{p_1, ..., p_s\}$, are the* target articles *of q. Given a target article p, each* Wikipedia *article p' that is hyperlinked from p is an* attribute article *of p. This relationship is represented as $p' \leftarrow p$. Given $\mathcal{T}$, the set of attribute articles is $\mathcal{A}=\{p'_1, ..., p'_m\}$, where each p'_i is an attribute article of at least one target article $p_j \in \mathcal{T}$.*

Definition 2 (Category Hierarchy)**.** Wikipedia category hierarchy *is a connected, rooted directed acyclic graph $\mathcal{H}(r_\mathcal{H}, \mathcal{C}_\mathcal{H}, \mathcal{E}_\mathcal{H})$, where the node set $\mathcal{C}_\mathcal{H}=\{c\}$ is the set of categories and the edge set $\mathcal{E}_\mathcal{H}= \{c \dashrightarrow c'\}$ is the set of category(c)-subcategory(c') relationships. The root category of $\mathcal{H}$, $r_\mathcal{H}$, is* Category:Fundamental. [3]

Definition 3 (Facet)**.** *A* facet *$\mathcal{F}(r, \mathcal{C}_\mathcal{F}, \mathcal{E}_\mathcal{F})$ is a rooted and connected subgraph of the category hierarchy $\mathcal{H}(r_\mathcal{H}, \mathcal{C}_\mathcal{H}, \mathcal{E}_\mathcal{H})$, where $\mathcal{C}_\mathcal{F} \subseteq \mathcal{C}_\mathcal{H}$, $\mathcal{E}_\mathcal{F} \subseteq \mathcal{E}_\mathcal{H}$, and $r \in \mathcal{C}_\mathcal{F}$ is the root of $\mathcal{F}$.*

Example 2 (Running Example)**.** *In Figure 3.3 there are 7 target articles (p_1, . . ., p_7) and 9 attribute articles (p'_1, . . ., p'_9). The category hierarchy has 14 categories (c_1, . . ., c_{14}). The figure highlights 6 facets ($\mathcal{F}_1$, . . ., $\mathcal{F}_5$, and $\mathcal{F}'_2$). For instance, F_2 is rooted at c_2 and consists of 3 categories (c_2, c_7, c_8) and 2 edges ($c_2 \dashrightarrow c_7$, $c_2 \dashrightarrow c_8$). There are many more*

[3] http://en.wikipedia.org/wiki/Category:Fundamental

facets since every rooted and connected subgraph of the hierarchy is a facet. Note that the figure may give the impression that edges such as $c_{11}\dashrightarrow c_{14}$ *and* $c_7\Rightarrow p'_1$ *are unnecessary since there is only one choice under* c_{11} *and* c_7*, respectively. The example is small due to space limitations. Such single outgoing edge is very rare in the real* Wikipedia *category hierarchy. We will use Figure 3.3 as the running example throughout the paper.*

The categories in the facet can "*reach*" the target articles $\mathcal{T}$ through attribute articles $\mathcal{A}$. That is, by following the category-subcategory hierarchy of the facet, we could find a category, then find an attribute article belonging to the category, and finally find the target articles that have the attribute. These target articles are called *reachable target articles*. A *facet* is a *safe reaching facet* if $\forall c{\in}\mathcal{C}_{\mathcal{F}}$, there exists a target article $p{\in}\mathcal{T}$ such that c reaches p, i.e., there exists $c\dashrightarrow ...\Rightarrow p'\leftarrow p$, a navigational path of $\mathcal{F}$, starting from c, that reaches p. In order to capture the notion of "reach", we formally define *navigational path* as follows.

Definition 4 (Navigational Path)**.** *With respect to the target articles* $\mathcal{T}$*, the attribute articles* $\mathcal{A}$*, and a facet* $\mathcal{F}(r,\mathcal{C}_{\mathcal{F}},\mathcal{E}_{\mathcal{F}})$*, a* navigational path *in* $\mathcal{F}$ *is a sequence* $c_1\dashrightarrow ...\dashrightarrow c_t\Rightarrow p'\leftarrow p$*, where,*

- *for* $1{\leq}i{\leq}t$*,* $c_i{\in}\mathcal{C}_{\mathcal{F}}$*, i.e.,* c_i *is a category in* $\mathcal{F}$*;*
- *for* $1{\leq}i{\leq}t{-}1$*,* $c_i\dashrightarrow c_{i+1}{\in}\mathcal{E}_{\mathcal{F}}$*, i.e.,* c_{i+1} *is a subcategory of* c_i *(in category hierarchy* $\mathcal{H}$*) and that category-subcategory relationship is kept in* $\mathcal{F}$*.*
- $p'{\in}\mathcal{A}$*, and* c_t *is a category of* p' *(represented as* $c_t \Rightarrow p'$*);*
- $p{\in}\mathcal{T}$*, and* p' *is an attribute article of* p *(i.e., there is a hyperlink* $p \rightarrow p'$*).*

Given a navigational path $c_1\dashrightarrow ...\dashrightarrow c_t\Rightarrow p'\leftarrow p$*, we say that the corresponding category path* $c_1\dashrightarrow ...\dashrightarrow c_t$ reaches *target article* p *through attribute article* p'*, and we also say that category* c_i *(for any* $1{\leq}i{\leq}t$*)* reaches p *through* p'*. Interchangeably we say* p *is* reachable *from* c_i *(for any* $1{\leq}i{\leq}t$*).*

Definition 5 (Faceted Interface)**.** *Given a keyword query* q*, a faceted interface* $I{=}\{\mathcal{F}_i\}$ *is a set of safe reaching facets of the target articles* $\mathcal{T}$*. That is,* $\forall\mathcal{F}_i{\in}I$*,* $\mathcal{F}_i$ *safely reaches* $\mathcal{T}$*.*

Example 3 (Navigational Path and Faceted Interface)**.** *Continue the running example. In Figure 3.3, $I{=}\{\mathcal{F}_2, \mathcal{F}_5\}$ is a 2-facet interface. Two examples of navigational paths are $c_2{\dashrightarrow}c_8{\Rightarrow}p'_3{\leftarrow}p_5$ and $c_5{\dashrightarrow}c_{13}{\Rightarrow}p'_9{\leftarrow}p_5$. However, $\{\mathcal{F}'_2, \mathcal{F}_5\}$ is not a valid faceted interface because $\mathcal{F}'_2$ is not a safe reaching facet, as category c_6 cannot reach any target articles.*

Based on the formal definitions, the **Faceted Interface Discovery Problem** is: Given the category hierarchy $\mathcal{H}(r_{\mathcal{H}}, \mathcal{C}_{\mathcal{H}}, \mathcal{E}_{\mathcal{H}})$, for a keyword query q and its resulting target articles $\mathcal{T}$ and corresponding attribute articles $\mathcal{A}$, find the "best" faceted interface with k facets. We shall develop the notion of "best" in Section 3.4.

3.4 Facet Ranking

The search space of the faceted interface discovery problem is prohibitively large. Given the set of s target *Wikipedia* articles to a keyword query, $\mathcal{T}$, there are a large number of attribute articles which in turn have many categories associated with complex hierarchical relationships. To just give a sense of the scale, in *Wikipedia* there are about 3 million English articles with hundreds of millions of internal links. The category system $\mathcal{H}$ contains close to half a million categories and several million category-subcategory relationships. By definition, any rooted and connected subgraph of $\mathcal{H}$ that safely reaches $\mathcal{T}$ is a candidate facet, and any combination of k facets would be a candidate faceted interface. Given the large space, we need ranking metrics for measuring the "goodness" of facets, both individually and collectively as interfaces.

Given that a faceted interface is for a user to navigate through the associated category hierarchies and ultimately reaching the target articles, it is natural to rank the interfaces by the user's navigational cost, i.e., the amount of effort undertaken by the user during

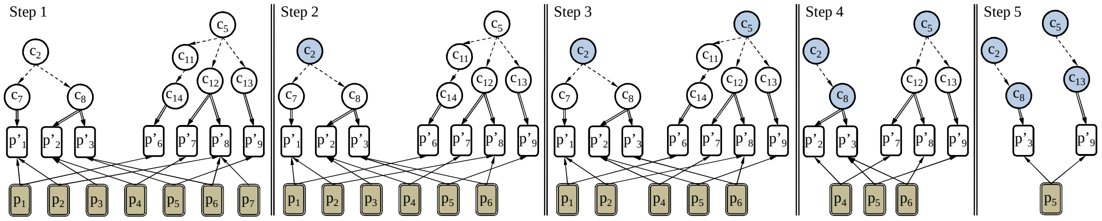


Figure 3.4. The navigation on a 2-facet interface $\mathcal{I} = \{\mathcal{F}_2, \mathcal{F}_5\}$.

navigation. [4] The “best” k-facet interface is the one with the smallest cost. Therefore as the basis of such ranking metrics, we model users’ navigational behaviors as follows.

User Navigation Model: A user navigates multiple facets in a k-facet interface. At the beginning, the navigation starts from the roots of all the k facets. At each step, the user picks one facet and examines the set of subcategories available at the current category on that facet. She follows one subcategory to further go down the category hierarchy. Alternatively the user may select one of the attribute articles reachable from the current category. The selections made on the k facets together form a conjunctive query. After the selection at each step, the list of target articles that satisfy the conjunctive query are brought to the user. The navigation terminates when the user decides that she has seen desirable target articles.

Example 4 (Navigation in Faceted Interface)**.** *Continue the running example in Figure 3.3. Consider a faceted interface I=$\{\mathcal{F}_2, \mathcal{F}_5\}$. A sequence of navigational steps on this interface are in Figure 3.4. At the beginning, the user has not selected any facet to explore, therefore all 7 target articles are available (step 1). Once the user decides to explore $\mathcal{F}_2$ which starts from c_2, p_7 is filtered out since it is unreachable from $\mathcal{F}_2$ (step 2). The user then selects c_5, which further removes p_3 from consideration (step 3). After the user further explores $\mathcal{F}_2$ by choosing c_8 (step 4), c_{11} is not a choice under c_5 anymore because no target articles could be reached by both $c_2 \dashrightarrow c_8$ and $c_5 \dashrightarrow c_{11}$. The user continues to explore*

[4] [44] also selects facets based on navigational costs, although their system is of a different nature, as discussed in Section 3.2.

$\mathcal{F}_5$ by choosing c_{13} (step 5), which removes p'_2 and also trims down the satisfactory target articles to $\{p_5\}$. The user may decide she has seen desirable articles and the navigation stops.

3.4.1 Single-Facet Ranking

In this section we focus on how to measure the costs of facets individually. Based on the navigational model, we compute the navigational cost of a facet as the average cost of its navigational paths. Intuitively a low-cost path, i.e., a path that demands small user effort, should have a small number of steps and at each step only require the user to browse a small number of choices. Therefore, we formally define the cost of a navigational path as the summation of the fan-outs (i.e., the number of choices) at every step, in logarithmic form. [5]

Definition 6 (Cost of Navigational Path)**.** *With respect to the target articles $\mathcal{T}$, the corresponding attribute articles $\mathcal{A}$, and a facet $\mathcal{F}(r, \mathcal{C}_{\mathcal{F}}, \mathcal{E}_{\mathcal{F}})$, the cost of a navigational path in $\mathcal{F}$ is*

$$cost(l) = log_2(fanout(p')) + \sum_{c \in \{c_1, \ldots, c_t\}} log_2(fanout(c)) \tag{3.1}$$

where $l{=}c_1 \dashrightarrow \ldots \dashrightarrow c_t \Rightarrow p' \leftarrow p$.

In Formula 3.1, $fanout(p')$ is the number of (directly) reachable target articles through the attribute article p',

$$fanout(p') = |\mathcal{T}_{p'}| \tag{3.2}$$

$$\mathcal{T}_{p'} = \{p | p \in \mathcal{T} \wedge p \rightarrow p' (\textit{i.e., } \exists \textit{ a hyperlink from } p \textit{ to } p')\} \tag{3.3}$$

In Formula 3.1, $fanout(c)$ is the fanout of category c in $\mathcal{F}$,

$$fanout(c) = |\mathcal{A}_c| + |\mathcal{C}_c| \tag{3.4}$$

[5] The intuition behind the logarithmic form is: When presented with a number of choices, the user does not necessarily scan through the choices linearly but by a binary search.

where $\mathcal{A}_c$ is the set of attribute articles belonging to c,

$$\mathcal{A}_c = \{p' | p' \in \mathcal{A} \wedge c \Rightarrow p'\} \tag{3.5}$$

and $\mathcal{C}_c$ is the set of subcategories of c in $\mathcal{F}$,

$$\mathcal{C}_c = \{c' | c' \in \mathcal{C}_\mathcal{F} \wedge c \dashrightarrow c' \in \mathcal{E}_\mathcal{F}\} \tag{3.6}$$

Note that we made several assumptions for simplicity of the model. The cost formula only captures the "browsing" cost. A full-fledged formula would need to incorporate other costs, such as the "clicking" cost in selecting a choice and the cost of "backward" navigation when the user decides to change a previous selection. Furthermore, we assume the user always completes the navigational path till reaching the target articles. In reality, however, the user may stop in the middle when she already finds desirable articles reachable from the current selection of category. We leave the investigation of more sophisticated models to future study.

Example 5 (Cost of Navigational Path)**.** *We continue the running example.*
Given l=$c_5 \dashrightarrow c_{12} \Rightarrow p'_8 \leftarrow p_6$, a navigational path of $\mathcal{F}_5$ in Figure 3.3,
$cost(l)$=$fanout(c_5)$+$fanout(c_{12})$+ $fanout(p'_8)$
=$log_2(3)$+$log_2(2)$+$log_2(3)$=4.17

Albeit the basis of our facet ranking metrics, the definition of navigational cost is not sufficient in measuring the goodness of a facet. It does not consider such a scenario that a facet cannot fully reach all the target articles, which presents an unsatisfactory user experience. In fact, low-cost and high-coverage could be two qualities that compete with each other. On the one hand, a low-cost facet could be one that reaches only a small portion of the target articles. On the other hand, a comprehensive facet with high coverage may tend to be wider and deeper, thus more costly. Therefore we must incorporate into the cost formula the notion of "coverage", i.e., the ability of a facet to reach as many target articles as possible. To combine navigational cost with coverage, we penalize a facet by associating

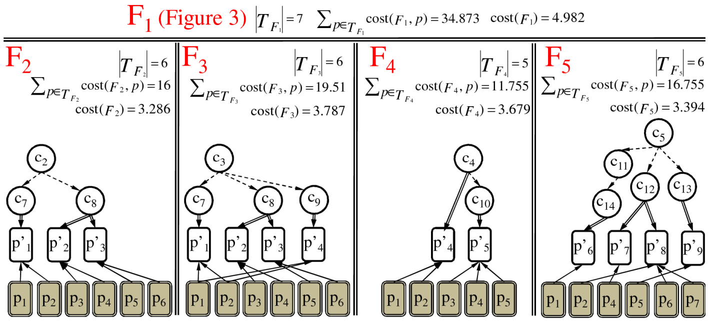

F1 (Figure 3) $|T_{F_1}|=7$ $\sum_{p\in T_{F_1}} cost(F_1, p) = 34.873$ $cost(F_1) = 4.982$
F2 $|T_{F_2}|=6$ $\sum_{p\in T_{F_2}} cost(F_2, p) = 16$ $cost(F_2) = 3.286$
F3 $|T_{F_3}|=6$ $\sum_{p\in T_{F_3}} cost(F_3, p) = 19.51$ $cost(F_3) = 3.787$
F4 $|T_{F_4}|=5$ $\sum_{p\in T_{F_4}} cost(F_4, p) = 11.755$ $cost(F_4) = 3.679$
F5 $|T_{F_5}|=6$ $\sum_{p\in T_{F_5}} cost(F_5, p) = 16.755$ $cost(F_5) = 3.394$
c2
c7
c8
p'1
p'2
p'3
p1
p2
p3
p4
p5
p6
c3
c7
c8
c9
p'1
p'2
p'3
p'4
p1
p2
p3
p4
p5
p6
c4
c10
p'4
p'5
p1
p2
p3
p4
p5
c5
c11
c12
c13
c14
p'6
p'7
p'8
p'9
p1
p2
p4
p5
p6
p7

Figure 3.5. Navigational costs of facets.

a high-cost *pseudo path* with each unreachable article. We then define the cost of a facet as the average cost in reaching each target article.

Definition 7 (Cost of Facet)**.** *With respect to the target articles $\mathcal{T}$, the cost of a safe reaching facet $\mathcal{F}(r, \mathcal{C}_{\mathcal{F}}, \mathcal{E}_{\mathcal{F}})$, $cost(\mathcal{F}_r)$, is the average cost in reaching each target article. The cost for a reachable target article is the average cost of the navigational paths that start from r and reach the target, and the cost for an unreachable target is a pseudo cost $penalty$.*

$$cost(\mathcal{F}_r) = \frac{1}{|\mathcal{T}|} \times (\sum_{p\in\mathcal{T}_r} cost(\mathcal{F}_r, p) + penalty \times |\mathcal{T} - \mathcal{T}_r|) \tag{3.7}$$

where $cost(\mathcal{F}_r, p)$ is the average cost of reaching p from r,

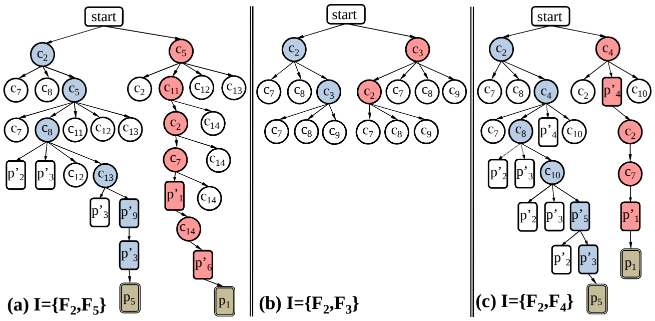
$$cost(\mathcal{F}_r, p) = \frac{1}{|l_p|} \times \sum_{l\in l_p} cost(l) \tag{3.8}$$

where l_p is the set of navigational paths in $\mathcal{F}$ that reach p from r,

$$l_p = \{l | l = r \dashrightarrow ... \Rightarrow p' \leftarrow p\} \tag{3.9}$$

In Formula 3.7, $penalty$ is the cost of the aforementioned expensive pseudo path that “reaches” the unreachable target articles, i.e., $\mathcal{T} - \mathcal{T}_r$, for penalizing a facet for not reaching them. Its value is empirically selected (Section 6.6) and is at least larger than the highest cost of any path to a reachable target article.

Example 6 (Cost of Facet)**.** *We continue the running example. Figure 3.5 shows the costs of the 5 highlighted facets in Figure 3.3, together with their category hierarchies and reach-*



(a) I={F$_2$,F$_5$} (b) I={F$_2$,F$_3$} (c) I={F$_2$,F$_4$}

Figure 3.6. The sequences of navigational steps.

able attribute and target articles. It does not show $\mathcal{F}_1$ *which is Figure 3.3 itself excluding* c_6. *The costs of facets are obtained by Formula 3.7, with* $penalty$=7. $cost(\mathcal{F}_2)$= $\frac{1}{7}\times(\sum_{p\in\{p_1,p_2,p_3,p_4,p_5,p_6\}} cost(\mathcal{F}_2, p)$ $+penalty\times|\mathcal{T}-\mathcal{T}_{\mathcal{F}_2}|)$= $\frac{1}{7}\times(16+7\times1)$=3.286. $\mathcal{F}_2$ *and* $\mathcal{F}_5$ *achieve lower costs than other facets. Even though the paths in* $\mathcal{F}_4$ *are cheap,* $\mathcal{F}_4$ *has higher cost due to the penalty for unreachable target articles (*p_6 *and* p_7*).* $\mathcal{F}_1$ *is even more costly due to its wider and deeper hierarchy, although it reaches all target articles.*

3.4.2 Multi-Facet Ranking

Even with the cost metrics for individual facets, measuring the "goodness" of a faceted interface, i.e., a set of facets, is not straightforward. This is because the best k-facet interface may not be simply the cheapest k facets. The reason is that when the user navigates multiple facets, the selection made at one facet has impact on the available choices on other facets, as illustrated by Example 4.

To directly follow the approach of ranking faceted interfaces by navigational cost, in principle we could represent the navigational steps on multiple facets as if the navigation is on one *"integrated" facet*. To illustrate, consider the navigation on a 2-facet interface $\mathcal{I}$=$\{\mathcal{F}_2, \mathcal{F}_5\}$ from Figure 3.3. Two possible sequences of navigational steps are shown in Figure 3.6(a). One is c_2, c_5, c_8, c_{13}, p'_9, p'_3, p_5, which are the steps taken by the user in

Figure 3.4, followed by choosing p'_9, p'_3, and finally p_5. (Remember, for simplification of the model, we assumed that the user will always complete navigational paths till reaching the target articles.) At each step, the available choices from both facets are put together as the choices in the "integrated" facet. Note that after c_8 is chosen, c_{12} and c_{13} are still valid choices but c_{11} is not available anymore because c_{11} cannot reach the target articles that c_8 reaches. For the same reason, after c_{13} is chosen, p'_3 is still a valid choice but p'_2 is not anymore. The other highlighted sequence is c_5, c_{11}, c_2, c_7, p'_1, c_{14}, p'_6, p_1. There are many more possible sequences not shown in the figure due to space limitations.

With the concept of "integrated" facet, one may immediately apply Definition 7 to define the cost of a faceted interface. That entails computing all possible sequences of interleaving navigational steps across all the facets in the interface. The interaction between facets is query- and data-dependent, rendering such exhaustive computation practically infeasible.

However, the "integrated" facet does shed light on what are the characteristics of good faceted interfaces. In general an interface should not include two facets that overlap much. Imagine a special case when two facets form a subsumption relationship, i.e., the root of one facet is a supercategory of the other root. Presenting both facets would not be desirable since they overlap significantly, thus cannot capture the expected properties of reaching target articles through different dimensions. As a concrete example, consider the navigational steps of $\mathcal{F}_2$ and $\mathcal{F}_3$ in Figure 3.6(b). After the user selects c_2 from $\mathcal{F}_2$ and then c_3 from $\mathcal{F}_3$, the available choices become $\{c_7, c_8, c_9\}$, which all come from the "dimension", $\mathcal{F}_3$. The same happens if the user selects c_3 and then c_2.

Based on the above observation, we propose to capture the overlap of the k facets by their *average pair-wise similarity*. The pair-wise similarity of two facets is the degree of overlap of their category hierarchies and associated attribute articles, defined below.

Definition 8 (Average Similarity of k-Facet Interface)**.** *The average pair-wise similarity of a k-facet interface is*

$$sim(\mathcal{I} = \{\mathcal{F}_1, ..., \mathcal{F}_k\}) = \frac{\sum_{1 \leq i < j \leq k} sim(\mathcal{F}_i, \mathcal{F}_j)}{k(k-1)/2}, \tag{3.10}$$

where $sim(\mathcal{F}_i, \mathcal{F}_j)$ is defined by the Jaccard coefficient,

$$sim(\mathcal{F}_i, \mathcal{F}_j) = \frac{|\mathcal{C}_{\mathcal{F}_i} \bigcap \mathcal{C}_{\mathcal{F}_j}| + |\mathcal{A}_{\mathcal{F}_i} \bigcap \mathcal{A}_{\mathcal{F}_j}|}{|\mathcal{C}_{\mathcal{F}_i} \bigcup \mathcal{C}_{\mathcal{F}_j}| + |\mathcal{A}_{\mathcal{F}_i} \bigcup \mathcal{A}_{\mathcal{F}_j}|} \tag{3.11}$$

where $\mathcal{C}_{\mathcal{F}_i}$ is the set of categories in $\mathcal{F}_i$ (Definition 3) and $\mathcal{A}_{\mathcal{F}_i}$ is the set of attribute articles reachable from $\mathcal{F}_i$,

$$\mathcal{A}_{\mathcal{F}_i} = \{p' | p' \in \mathcal{A} \wedge \exists c \in \mathcal{C}_{\mathcal{F}_i} \ s.t. \ c \Rightarrow p'\} \tag{3.12}$$

We choose Jaccard coefficient since it is one of the simplest set-similarity measures. While more complex measures that give different weights to nodes higher in the hierarchy are possible, we do not follow that in the interest of simplicity.

Example 7 (Similarity of Facets)**.** *Consider facets $\mathcal{F}_1, \ldots, \mathcal{F}_5$ in Figure 3.3.* $sim(\mathcal{F}_2, \mathcal{F}_3)$= $\frac{|\mathcal{C}_{\mathcal{F}_2} \bigcap \mathcal{C}_{\mathcal{F}_3}|+|\mathcal{A}_{\mathcal{F}_2} \bigcap \mathcal{A}_{\mathcal{F}_3}|}{|\mathcal{C}_{\mathcal{F}_2} \bigcup \mathcal{C}_{\mathcal{F}_3}|+|\mathcal{A}_{\mathcal{F}_2} \bigcup \mathcal{A}_{\mathcal{F}_3}|}$ $= \frac{|\{c_7,c_8\}|+|\{p'_1,p'_2,p'_3\}|}{|\{c_2,c_7,c_8,c_3,c_9\}|+|\{p'_1,p'_2,p'_3,p'_4\}|}$=$5/9$. *Other pair-wise similarities can be computed in the same way. The average pari-wise similarity of $\mathcal{I}$=$\{\mathcal{F}_2, \mathcal{F}_3, \mathcal{F}_5\}$ is* $sim(\mathcal{I}) = (sim(\mathcal{F}_2, \mathcal{F}_3) + sim(\mathcal{F}_2, \mathcal{F}_5) + sim(\mathcal{F}_3, \mathcal{F}_5))/3 = 5/27$.

We do not design a single function to combine the average pair-wise similarity of a faceted interface with its navigational cost, since they represent two measures with different natures. Instead, in Section 3.5.3 we discuss how to search the space of candidate interfaces by considering both measures.

3.5 Algorithms

A straightforward approach for faceted interface discovery is to enumerate all possible k-facet interfaces with respect to the category hierarchy $\mathcal{H}$ and apply the ranking metrics directly to find the best interface. Such a naïve method results in the exhaustive examination of all possible combinations of k instances of all possible facets, i.e., rooted and connected subgraphs of $\mathcal{H}$. Clearly it is a prohibitively large search space, given the sheer size and complexity of *Wikipedia*. The naïve technique would be extremely costly. Therefore finding the best k-facet interface is a challenging optimization problem.

Our k-facet discovery algorithm hinges on (1) reducing the search space; and (2) searching the space effectively and efficiently.

Reducing the Search Space: There are two search spaces in finding a good k-facet interface: the space of facets and the space of k-facet interfaces, which are sets of k facets. To reduce the space of candidate facets, we focus on a subset of the safe reaching facets, $\mathcal{RCH}$-induced facets, which are the facets that contain all the descendant categories of their roots (Section 3.5.1). To further reduce the space of faceted interfaces, we rank the facets individually by their navigational costs (Section 3.5.2) and only consider the top ranked facets that do not subsume each other (Section 3.5.3).

Searching the Space: Instead of exhaustively examining all possible interfaces, we design a hill-climbing based heuristic algorithm to look for a local optimum (Section 3.5.3). To further tackle the challenge of modeling the interactions of multiple facets in measuring the cost of an interface, the hill climbing algorithm optimizes for both the average navigational cost and the pair-wise similarity of the facets.

Our k-facet discovery algorithm is outlined as three steps: construction of relevant category hierarchy, ranking single facet, and searching for k-facet interface.

3.5.1 Relevant Category Hierarchy (Algorithm 3)

By Definition 5, the facets in a faceted interface must be safe reaching facets, i.e., they do not contain “dead end” categories that cannot reach any target articles. Therefore the categories appearing in any safe reaching facet could only come from the *relevant* *category hierarchy* ($\mathcal{RCH}$), which is a subgraph of the *Wikipedia* category hierarchy $\mathcal{H}$, defined below.

Definition 9 (Relevant Category Hierarchy)**.** *Given the category hierarchy $\mathcal{H}(r_{\mathcal{H}}, \mathcal{C}_{\mathcal{H}}, \mathcal{E}_{\mathcal{H}})$, the target articles $\mathcal{T}$, and the attribute articles $\mathcal{A}$, the* relevant category hierarchy *($\mathcal{RCH}$) of $\mathcal{T}$ is a subgraph of $\mathcal{H}$. Given any category in $\mathcal{RCH}$, it is either directly a category of some attribute article $p' \in \mathcal{A}$ or a super-category or ancestor of such categories. There exists an edge (category-subcategory relationship) between two categories in $\mathcal{RCH}$ if the same edge exists in $\mathcal{H}$. By this definition the root of $\mathcal{H}$ is also the root of $\mathcal{RCH}$.*

The procedural algorithm for getting $\mathcal{RCH}$ is in Algorithm 3. Based on definition, straightforwardly we could prove every safe reaching facet of the target articles $\mathcal{T}$ is a (rooted and connected) subgraph of $\mathcal{RCH}$. However, not every rooted and connected subgraph of $\mathcal{RCH}$ is a safe reaching facet. Therefore, even though $\mathcal{RCH}$ is much smaller than $\mathcal{H}$, the search space is still very large which needs us to further shrink the space by considering only one type of safe reaching facets, the *$\mathcal{RCH}$-induced facets*.

Definition 10 ($\mathcal{RCH}$-Induced Facet)**.** *Given the relevant category hierarchy $\mathcal{RCH}$ of the target articles $\mathcal{T}$, a facet $\mathcal{F}(r, \mathcal{C}_{\mathcal{F}}, \mathcal{E}_{\mathcal{F}})$ is $\mathcal{RCH}$-*induced *if it is a rooted induced subgraph of $\mathcal{RCH}$, i.e., in $\mathcal{F}$ all the descendants of the root r and their category-subcategory relationships are retained from $\mathcal{RCH}$.*

Example 8 ($\mathcal{RCH}$ and $\mathcal{RCH}$-Induced Facet)**.** *Continue the running example. In Figure 3.3, the $\mathcal{RCH}$ contains all the categories in the category hierarchy $\mathcal{H}$ except c_6 (and thus the edge $c_2 \dashrightarrow c_6$), since c_6 cannot reach any target article. $\mathcal{F}_2$ is an $\mathcal{RCH}$-induced facet, but would not be if it does not contain c_7 (or c_8).*

Note that every $\mathcal{RCH}$-induced facet is safe reaching, and the single-facet ranking and searching for k-facet are performed on it.

Algorithm 3: Construct RCH and Get Attribute Articles

```
Input: $\mathcal{T}$: target articles; $\mathcal{H}$: category hierarchy.
Output: $\mathcal{A}$:attribute articles; $\mathcal{RCH}$:relevant category hierarchy.
// get attribute articles.
$\mathcal{A}$←$\emptyset$; $\mathcal{C}_{\mathcal{RCH}}$←$\emptyset$; $\mathcal{E}_{\mathcal{RCH}}$←$\emptyset$
foreach $p \in \mathcal{T}$ do
    foreach $p \rightarrow p'$, i.e., a hyperlink from $p$ to $p'$ do
        $\mathcal{A}$←$\mathcal{A} \cup \{p'\}$
// start from the categories of attribute articles.
foreach $p' \in \mathcal{A}$ do
    foreach $c \Rightarrow p'$, i.e., a category of $p'$ do
        $\mathcal{C}_{\mathcal{RCH}} \leftarrow \mathcal{C}_{\mathcal{RCH}} \cup \{c\}$
// recursively obtain the supercategories.
$\mathcal{C}$←$\mathcal{C}_{\mathcal{RCH}}$; $\mathcal{C}' \leftarrow \emptyset$
while $\mathcal{C}$ is not empty do
    foreach $c \in \mathcal{C}$ do
        foreach $c' \dashrightarrow c \in \mathcal{E}_{\mathcal{H}}$ do
            $\mathcal{E}_{\mathcal{RCH}} \leftarrow \mathcal{E}_{\mathcal{RCH}} \cup \{c' \dashrightarrow c\}$
            if $c' \notin \mathcal{C}_{\mathcal{RCH}}$ then
                $\mathcal{C}_{\mathcal{RCH}} \leftarrow \mathcal{C}_{\mathcal{RCH}} \cup \{c'\}$; $\mathcal{C}' \leftarrow \mathcal{C}' \cup \{c'\}$
    $\mathcal{C} \leftarrow \mathcal{C}'$; $\mathcal{C}' \leftarrow \emptyset$
return $\mathcal{A}$ and $\mathcal{RCH}(r_{\mathcal{H}}, \mathcal{C}_{\mathcal{RCH}}, \mathcal{E}_{\mathcal{RCH}})$
```

Algorithm 4: Facet Ranking

```
Input: $\mathcal{T}$:targets;$\mathcal{A}$:attributes; $\mathcal{RCH}$:relevant category hierarchy.
Output: $\mathcal{I}_n$: top $n$ $\mathcal{RCH}$-induced facets with smallest costs.
// get reachable target articles for each attribute article.
foreach $p' \in \mathcal{A}$ do
    $\mathcal{T}_{p'} \leftarrow \{p | p \in \mathcal{T} \wedge \exists\, p \rightarrow p'$ (hyperlink from $p$ to $p'$) $\}$
    $fanout(p') \leftarrow |\mathcal{T}_{p'}|$
initialize $visited(r)$ to be $False$ for every $r \in \mathcal{C}_{\mathcal{RCH}}$.
$ComputeCost(r_{\mathcal{H}})$ // recursively compute the costs of all the $\mathcal{RCH}$-induced facets, starting
    from the root of $\mathcal{RCH}$.
$\mathcal{I}_n \leftarrow$ the top $n$ $\mathcal{RCH}$-induced facets with the smallest costs.
return $\mathcal{I}_n$
```

Algorithm 5: ComputeCost(r)

```
Input: $r$: the root of an $\mathcal{RCH}$-induced facet.
Output: $cost(\mathcal{F}_r)$: cost of $\mathcal{F}_r$; $cost(\mathcal{F}_r, p)$: average cost of reaching target article $p$ from $\mathcal{F}_r$;
        $pathcnt(\mathcal{F}_r, p)$: number of navigational paths reaching $p$ from $\mathcal{F}_r$; $\mathcal{T}_r$: reachable target
        articles of $r$.
if $visited(r)$ then
    return
$visited(r) \leftarrow True$;
$\mathcal{C}_r \leftarrow \{c | r \dashrightarrow c \in \mathcal{E}_{\mathcal{RCH}}\}$ // subcategories of $r$.
foreach $c \in \mathcal{C}_r$ do
    $ComputeCost(c)$
$\mathcal{A}_r \leftarrow \{p' | p' \in \mathcal{A} \wedge r \Rightarrow p'\}$ // attribute articles belong to $r$.
$fanout(r) \leftarrow |\mathcal{A}_r| + |\mathcal{C}_r|$
$\mathcal{T}_r \leftarrow (\cup_{p' \in \mathcal{A}_r} \mathcal{T}_{p'}) \bigcup (\cup_{c \in \mathcal{C}_r} \mathcal{T}_c)$ // reachable target articles.
foreach $p \in \mathcal{T}_r$ do
    $pathcnt(\mathcal{F}_r, p) \leftarrow |\{p' | p' \in \mathcal{A}_r, p \in \mathcal{T}_{p'}\}| + \sum_{c \in \mathcal{C}_r} pathcnt(\mathcal{F}_c, p)$
    $cost_1 \leftarrow \sum_{p' \in \mathcal{A}_r s.t. p \in \mathcal{T}_{p'}} (log_2(fanout(r)) + log_2(fanout(p')))$
    $cost_2 \leftarrow \sum_{c \in \mathcal{C}_r} (log_2(fanout(r)) + cost(\mathcal{F}_c, p)) \times pathcnt(\mathcal{F}_c, p)$
    $cost(\mathcal{F}_r, p) \leftarrow \frac{cost_1 + cost_2}{pathcnt(\mathcal{F}_r, p)}$
$cost(\mathcal{F}_r) \leftarrow \sum_{p \in \mathcal{T}_r} cost(\mathcal{F}_r, p) + penalty \times |\mathcal{T} - \mathcal{T}_r|$
return
```

Algorithm 6: Facet Selection

```
   Input: $\mathcal{I}_n$: the top $n$ $\mathcal{RCH}$-induced facets with the smallest costs.
   Output: $\mathcal{I}_k$: a discovered faceted interface with $k$ facets ($k$<$n$).
   // remove subsumed facets from $\mathcal{I}_n$
1  $\mathcal{I}_{n^-}\leftarrow\{\mathcal{F}_c|\nexists\mathcal{F}_{c'}\in\mathcal{I}_n$ s.t.$\mathcal{F}_c$ is subsumed by $\mathcal{F}_{c'}$, i.e., $c$ is a descendant category of $c'\}$
   // hill climbing
2  $\mathcal{I}_k\leftarrow$ a random $k$-facet subset of $\mathcal{I}_{n^-}$;  $\mathcal{I}'\leftarrow\mathcal{I}_{n^-}\backslash\mathcal{I}_k$
3  repeat
5      make $\mathcal{I}_k$=<$\mathcal{I}_k[1]$,...,$\mathcal{I}_k[k]$> sorted in increasing order of cost.
6      make $\mathcal{I}'$=<$\mathcal{I}'[1]$,...,$\mathcal{I}'[n-k]$> sorted in increasing order of cost
7      for $i=k$ to $1$ step $-1$ do
8          for $j=1$ to $n-k$ do
9              $\mathcal{I}_{new}\leftarrow(\mathcal{I}_k\backslash\{\mathcal{I}_k[i]\})\cup\{\mathcal{I}'[j]\}$
10             $S_1\leftarrow\sum_{\mathcal{F}_c,\mathcal{F}_{c'}\in\mathcal{I}_{new},\mathcal{F}_c\neq\mathcal{F}_{c'}} sim(\mathcal{F}_c,\mathcal{F}_{c'})$
11             $C_1\leftarrow\sum_{\mathcal{F}_c\in\mathcal{I}_{new}} cost(\mathcal{F}_c)$
12             $S_2\leftarrow\sum_{\mathcal{F}_c,\mathcal{F}_{c'}\in\mathcal{I}_k,\mathcal{F}_c\neq\mathcal{F}_{c'}} sim(\mathcal{F}_c,\mathcal{F}_{c'})$
13             $C_2\leftarrow\sum_{\mathcal{F}_c\in\mathcal{I}_k} cost(\mathcal{F}_c)$
14             if ($S_1\leq S_2$ and $C_1<C_2$) or ($S_1<S_2$ and $C_1\leq C_2$) then
15                 $\mathcal{I}_k\leftarrow\mathcal{I}_{new}$;  $\mathcal{I}'\leftarrow\mathcal{I}_{n^-}\backslash\mathcal{I}_k$
16                 go to line 5
17 until $\mathcal{I}_k$ does not change ;
18 return $\mathcal{I}_k$
```

3.5.2 Ranking Single Facet (Algorithm 4 and 5)

Among all the $\mathcal{RCH}$-induced facets, only the top n facets with the smallest navigational costs are considered in searching for a faceted interface. In ranking the facets by their costs, one straightforward approach is to enumerate all the $\mathcal{RCH}$-induced facets and to separately compute the cost of each facet by enumerating all of its navigational paths. This approach is exponentially complex due to repeated traversal of the edges in $\mathcal{RCH}$, because the $\mathcal{RCH}$-induced facets would have many common categories and category-subcategory relationships.

To avoid the costly exhaustive method, we design a recursive algorithm that calculates the navigational costs of all the $\mathcal{RCH}$-induced facets by only one pass depth-first search of $\mathcal{RCH}$. The details are in Algorithm 4. The essence of the algorithm is to, during the recursive traversal of $\mathcal{RCH}$, book-keep the number of navigational paths in a facet in addition to its navigational cost. The bookkeeping is performed for each reachable target article because the cost is averaged across all such articles by Definition 7. The cost of a facet rooted at r can be fully computed based on the book-kept information of the facets rooted at r's direct subcategories, without accumulating the individual costs of the facets rooted at r's descendants. Therefore it avoids the aforementioned repeated traversal of $\mathcal{RCH}$. More specifically, the lines 11-14 in Algorithm 5 are for computing $cost(\mathcal{F}_r, p)$ in Formula 3.7. However, the algorithm does not compute it by a direct translation of Formula 3.8 and 3.1, i.e., enumerating all the navigational paths that reach p. Instead, line 12 gets $cost_1$, the total cost of all the navigational paths $r \Rightarrow p' \leftarrow p$, i.e., the ones that reach p without going through any other categories; line 13 computes $cost_2$, the total cost of all the navigational paths that go through other categories, by utilizing $cost(\mathcal{F}_c, p)$ and $pathcnt(\mathcal{F}_c, p)$ of the subcategories c, but not other descendants. We omit the formal correctness proof.

3.5.3 Searching for k-Facet Interface (Algorithm 6)

Algorithm 6 searches for k-facet interface. To reduce the search space, our algorithm only considers $\mathcal{I}_n$, the top n facets from Algorithm 4. We further reduce the space by excluding those top ranked facets that are subsumed by other top facets (line 1). In other words, we only keep $\mathcal{I}_{n^-}$, the maximal *antichain* of $\mathcal{I}_n$ based on the graph (category hierarchy) subsumption relationship. This is in line with the idea of avoiding large overlap between facets (Section 3.4.2).

Given $\mathcal{I}_{n^-}$, instead of exhaustively considering all possible k-element subsets of $\mathcal{I}_{n^-}$, we apply a *hill-climbing method* to search for a local optimum, starting from a random k-facet interface $\mathcal{I}_k$. At every step, we try to find a better neighboring solution, where a k-facet interface $\mathcal{I}_{new}$ is a neighbor of $\mathcal{I}_k$ if they only differ by one facet (line 9). Given the $k \times (n - k)$ possible neighbors at every step, we examine them in the order of average navigational costs (line 5, 6, and 9). The algorithm jumps to the first encountered better neighbor. The algorithm stops when no better neighbor can be found. As the goal function to be optimized in hill-climbing, $\mathcal{I}_{new}$ is considered better if the facets of $\mathcal{I}_{new}$ have both smaller pair-wise similarities and smaller navigational costs than that of $\mathcal{I}_k$ (line 14). The idea of considering both similarity and cost is motivated in Section 3.4.2.

3.6 Experimental Evaluation

3.6.1 Experimental Settings

Facetedpedia is implemented in C++ and the dataset is stored in a *MySQL* database. The experiments are executed on a Dell PowerEdge 2900 III server running Linux kernel 2.6.27, with dual quad-core Xeon 2.0 GHz processors, $2$x6MB cache, 8GB RAM, and three 1TB SATA hard drivers in RAID5.

Dataset: We downloaded the *Wikipedia* dump of July 24, 2008 from http://download.wikimedia.org and loaded the data into a *MySQL* database. In particular, we used the tables *page.sql*, *pagelinks.sql*, *categorylinks.sql*, and *redirect.sql*, which provide all the relevant data including the hyperlinks between articles, categories of articles, and the category system. We performed several preprocessing tasks on the tables, including the detection and removal of cycles in the category hierarchy. Although cycles should usually be avoided as suggested by *Wikipedia*, the category system in *Wikipedia* contains

number of articles	$2,445,642$
number of hyperlinks between articles	$109,165,108$
average number of hyperlinks per article	45
number of distinct categories	$329,007$
average number of categories per article	3
number of category-subcategory relationships	$731,097$

Figure 3.7. Characteristics of the dataset.

a very small number (594 in the dataset) of elementary cycles [6] due to various reasons. We applied depth-first search algorithm to detect the elementary cycles. The category hierarchy is made acyclic by removing the last encountered edge in each elementary cycle during the depth-first search. Other performed preprocessing steps include: removing tuples irrelevant to articles and categories; replacing redirect articles by their original articles; removing special articles such as lists and stubs. We also applied basic performance tuning of the database, including creating additional indexes on *page_id* in various tables. The characteristics of the dataset are summarized in Figure 3.7. The total size of the tables is 1.2GB.

Queries: We experimented with 20 keyword queries that we designed (Figure 3.8), in addition to the open queries that the users came up with during user study (Section 3.6.2).

Parameters in algorithms: Each query was sent to *Google* with site constraint *site:en.wikipedia.org* to get the top 200 (s=200) English *Wikipedia* target articles. The relevant category hierarchy $\mathcal{RCH}$ was then generated by applying Algorithm 3 on the aforementioned *MySQL* database. By default, Algorithm 4 returns top 200 (n=200) facets and Algorithm 6 generates 10 facets (k=10). The value of $penalty$ in Definition 7 was set as 7. It was empirically selected by investigating the relationship between the number of unreachable target articles ($|\mathcal{T} - \mathcal{T}_r|$) and the total navigational costs of reachable targets ($\sum_{p\in\mathcal{T}_r} cost(\mathcal{F}_r, p)$).

[6] A cycle is elementary if no vertices in the cycle (except the start/end vertex) appear more than once.

$Q1$	action film	$Q2$	country singer
$Q3$	philosophers	$Q4$	Texas universities
$Q5$	Turing Award winner	$Q6$	missile
$Q7$	Ivy League schools	$Q8$	NBA players
$Q9$	historic landmarks	$Q10$	cartoon characters
$Q11$	Microsoft acquired game companies	$Q12$	stand up comedian
$Q13$	graph theorists	$Q14$	lakes in North America
$Q15$	American presidents	$Q16$	battle far east
$Q17$	waterfall national park	$Q18$	Chinese cuisine
$Q19$	premier league clubs	$Q20$	PS3 game

Figure 3.8. Experiment queries.

3.6.2 User Studies

We conducted user studies to evaluate the effectiveness of *Facetedpedia*, and to compare the quality of the faceted interfaces generated by *Facetedpedia* and *Castanet* [40]. We obtained the implementation of *Castanet* from its authors. Note that *Castanet* is intended for static, short, and domain-specific documents with limited vocabularies. Nevertheless, we applied *Castanet* on the dynamic keyword search results. Although not originally designed for such purposes, *Castanet* still appears to be possibly the closest related work. We use the same graphical user interface for both systems, to make the comparison irrelevant to interface design.

The user studies were conducted online. The users all have college degrees or are in college, including university students, faculty, staff, and financial and IT company workers. We believe these users are experienced with Web search and comfortable with more sophisticated access mechanisms, matching the target users of our system. To reduce the overhead of the user, we partitioned the 20 queries in Figure 3.8 into 4 equal-size groups and asked each user to only participate in the 5 queries of one group. For each query group, we sent user-study invitations to roughly equal number of people. Ultimately we were able to collect opinions from totally 36 users, 8 each for 2 groups, and 10 each for the other 2 groups.

For each query, we showed the query keywords and objective description to the user, and asked the user to explore two interfaces pre-generated by *Facetedpedia* and *Castanet*,

Choices– 1: useless; 2: not very useful; 3: useful to some extent; 4: useful; 5: very useful	
R1	My rating about usefulness of Facetedpedia.
R2	My rating about usefulness of Castanet.

Choices– Facetedpedia; Castanet	
R3	Which interface is better than the other?

Choices– 1: strongly disagree; 2: disagree; 3: neutral; 4: agree; 5: strongly agree	
R4	The facets in Facetedpedia conveys important concepts regarding the articles related to the query.
R5	Facetedpedia is useful for browsing and exploration purposes.
R6	I look forward to use this interface even in the future for exploratory browsing purposes.

Figure 3.9. User study questions and available answers.

respectively. At the end of each query, the user was asked to provide response to 3 questions, namely $R1$-$R3$ in Figure 3.9. The available choices for $R1$ and $R2$ are ratings from 1:“useless” to 5:“very useful”. The choices for $R3$ are “Facetedpedia” and “Castanet”. The same process iterated through the 5 queries in the group assigned to the user. After the 5 queries were done, the user was also provided opportunity to try arbitrary open queries on *Facetedpedia*, and provided answers to questions $R4$-$R6$ in Figure 3.9. The available choices are ratings from 1:“strongly disagree” to 5:“strongly agree”. The same open query study, however, was not possible for *Castanet* because the implementation we obtained from the authors takes about 5 minutes to process each query and therefore could not be used for dynamic queries. The reason is that it checks *WordNet* for each word in constructing category hierarchy. (Remember it was designed for static collection of short texts.)

In Figure 3.10, column 2 and column 3 records average user ratings per query on questions $R1$ and $R2$ respectively. Column 4 and 5 represent user’s absolute preference on one system over the other. Clearly, from the results, *Facetedpedia* receives much stronger feedback than *Castanet* on $R1$ and $R2$. Also, for absolute preference, user prefers *Facetedpedia* over Castanet almost unanimously. Figure 3.11 records average user ratings per group for $R4$, $R5$ and $R6$. As it can be seen, majority of the groups provide strong positive

	Average R1	Average R2	R3-Facetedpedia	R3-Castanet
$Q1$	3.5	2.5	7	1
$Q2$	3.5	2.625	5	3
$Q3$	3.5	2.875	5	3
$Q4$	3.625	2.5	7	1
$Q5$	3.375	2.5	7	1
$Q6$	3.625	3.375	6	2
$Q7$	4.0	3.625	5	3
$Q8$	3.75	3.625	4	4
$Q9$	4.125	3.25	7	1
$Q10$	3.5	3.875	4	4
$Q11$	4.2	3.1	9	1
$Q12$	3.8	3.2	8	2
$Q13$	3.8	3.5	6	4
$Q14$	3.7	3.5	6	4
$Q15$	3.7	3.7	6	4
$Q16$	3.9	2.9	9	1
$Q17$	4.1	3.1	9	1
$Q18$	4.2	2.9	9	1
$Q19$	3.7	2.7	7	3
$Q20$	3.6	3.1	6	4

Figure 3.10. Usefulness of *Facetedpedia* and *Castanet*.

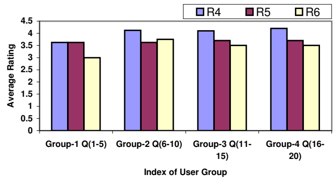


Figure 3.11. User experience with *Facetedpedia* for open queries.

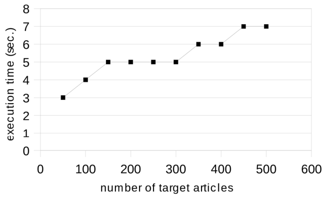
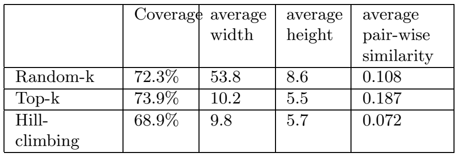


Figure 3.12. Execution time of *Facetedpedia* vs. number of target articles.

opinion the about usefulness of facets and the interface generated by *Facetedpedia* and they believe *Facetedpedia* interface is effective for exploration purposes.

3.6.3 Characteristics of Generated Facets

Our experiments compared the effectiveness of three algorithms: *hill-climbing* (Algorithm 6), *top-k*– selecting the top k facets ranked by Algorithm 4, and *random-k*– randomly choosing k facets. Figure 3.13 shows the average characteristics of the faceted interfaces generated by these methods. Although *hill-climbing* had a slightly worse target article coverage than the other two (5% less), it outperformed them in pair-wise similarity which means the k facets selected have smaller overlap of navigational paths. The detailed tracing results show that *hill-climbing* started from choosing top-k facets and gradually



	Coverage	average width	average height	average pair-wise similarity
Random-k	72.3%	53.8	8.6	0.108
Top-k	73.9%	10.2	5.5	0.187
Hill-climbing	68.9%	9.8	5.7	0.072

Figure 3.13. Compare the quality of faceted interfaces generated by various methods.

replaced similar facets by less similar ones. The final k facets selected by *hill-climbing* usually were still within the top 30%, while the ones selected by *random-k* were evenly distributed among the results from single-facet ranking. The average width and height of the facets generated by the three methods were about the same, except that *random-k* occasionally chose some much wider facets. Their average width and height were usually around 10 and 6, respectively. Therefore the fanout of internal nodes and the length of navigational paths are within a reasonable range for the users. Overall, *hill-climbing* helps us reducing overlapping facets without losing much coverage of target articles.

3.6.4 Efficiency Evaluation

We evaluated the scalability of our approach by measuring the average execution time of discovering k=10 facets for varying number of target articles (s from 50 to 500). As can be seen from Figure 3.12, *Facetedpedia* scales well since the execution time increases linearly by the number of target articles. It also shows that *Facetedpedia* already achieved fairly fast response without much performance optimization. In average it took 3 seconds to discover the facets for 50 target articles, and 5 seconds for 200 target articles.

3.7 Discussion

The faceted interfaces generated by *Facetedpedia* are certainly not perfect and could be improved on many aspects. The pitfalls and drawbacks of our system pose several open challenges which could possibly form new research directions. It is our plan to forward our investigation along the following lines:

First, hyperlinks in *Wikipedia* articles are not always good features of the target articles. In many cases the hyperlinked articles are important attribute articles that are strongly related to the target articles. However, there are also cases in which the authors of an article make hyperlinks to other articles not because they have strong relationships with the target articles. The author may believe that the readers would not be familiar with an entity mentioned in an article, therefore decides to make that mention an anchor text linking to the article describing the entity. The hyperlinked article is not necessarily highly related to the target article. For example, in *Wikipedia* article *Independence_Day_(film)*, hyperlinked articles such as *Will_Smith* and *20th_Century_Fox* are certainly valid attribute articles, while *Moon* and *Mexico* may not be. To assure the quality of the discovered facets, we plan to investigate data mining methods and NLP techniques in finding truly related articles.

Second, we found through experiments that a category hierarchy based on both rich semantics and strong IS-A relationships will provide more accurate facets than the current *Wikipedia* category system. This "grass-roots" folksonomy in *Wikipedia*, albeit containing user-generated categories with richer semantics than a thesaurus such as *WordNet* , is not always organized by rigorous IS-A relationships. For instance, it includes subcategories such as *People_from_Texas* and *History_of_Texas* under category *Texas*, which can be misleading to a user who plans to navigate through geographical concepts by choosing *Texas*. We plan to refine the category hierarchy for strong IS-A relationships.

Third, we need to design methods to improve the diversity of the top ranked facets generated by *Facetedpedia*. Since our ranking metric penalizes the facets that have small coverage, the top ranked ones may tend to come from relatively large concept domains such as people, organizations, etc. To avoid missing useful facets from small concept domains, one idea is to first cluster the attribute articles into several groups and then make sure that each group has at least a number of facets in the final results.

3.8 Conclusion

In this thesis we proposed *Facetedpedia*, a faceted search system over *Wikipedia*. This system provides a dynamic and automated faceted search interface for users to browse the articles that are the result of a keyword search query. Given the sheer size and complexity of *Wikipedia* and the large space of possible faceted interfaces, we proposed metrics for ranking faceted interfaces as well as efficient algorithms for discovering them. Our experimental evaluation and user study verify the effectiveness of our methods in generating useful faceted interfaces. Moreover, our findings pose several open problems for future study. It would also be interesting to further investigate if the proposed framework and methods can be applied to other applications, or even the generic Web.

CHAPTER 4

STAR COMPOSITE ITEMS

4.1 Introduction

While many online sites are still centered around facilitating a user's interaction with individual items (such as buying an iPod or booking a flight), an increasing emphasis is being put on helping users with more complex *search* activities, such as comparing similar products and determining which products are compatible with each other. For example, Amazon and Zappos offer the "Customers Who Viewed This Item Also Viewed" feature to engage users more effectively. Similarly, the "Explore by Destination" feature from Expedia invites users to examine related sights and activities in a given geographic location.

At the center of those activities is the notion of *composite item*. It consists of a *central item*, which is the main focus of the activity, and a *satellite package*, which is a set of *satellite items* of different *types* that are *compatible* with the central item. Compatibility can be either *soft* (e.g., other books that are often purchased together with the book being browsed) or *hard* (e.g., battery packs that must be compatible with the laptop or a travel destination that must be within a certain distance from the main destination). Composite items are often further constrained by certain criteria, such as a price budget on purchases and a time budget on travel itineraries.

Consider a user shopping for an iPhone with a price budget. In addition to the list of available iPhones within the budget, it is also desirable to present, along with each iPhone, a small set of packages, each of which consists of compatible items that can be purchased together with the iPhone and whose total price is within the budget. An example of such a package is $\{$*Belkin case, Bose sounddock, Kroo USB cable*$\}$. Here, compatibility between each item in the package and the returned iPhone, can be derived using item co-browsing and co-purchasing histories or absolute product compatibilities provided by manufacturers. Similarly, consider a user interested in discovering the northern and central parts of France. Typically, such a user will have a main destination (e.g., *Paris*) and a visit duration (akin to the price budget). In addition to the main destination, it is also desirable to present a set of small travel packages, each of which contains a few trips to nearby places (e.g., {*Normandy, Fontainebleau, Versailles*}), that can be completed within the indicated visit duration. Here, compatibility can be defined based on intrinsic properties of each location, such as the geographic distance between the central location and each satellite location. The goal of this work is to develop a principled approach for constructing such composite items and helping users explore them efficiently and effectively.

We address three main technical challenges. First, we aim to solve the problem of identifying all *valid* and *maximal* satellite packages given a central item. A valid package must satisfy a given budget such as a visit duration. A maximal package is the largest valid set of satellite items, where each item is compatible with the central item. A valid and maximal package is therefore *a set of compatible satellite items, such that, collectively with their central item, satisfy a budget and are not subsumed by another valid package*. We develop a random walk algorithm for that purpose.

The number of valid and maximal packages associated with a central item is typically very large and presenting all of them to the user is impractical. Hence, we tackle the challenge of *summarizing* the packages associated with a central item into k *representative* packages. Intuitively, the goal of summarization is to expose the user to as many satellite items as possible with as few as possible summary packages. Those packages can then be presented to the user, who can directly use them, or select a subset of satellite items to construct their desired composite items, *without worrying about checking the budget.*

We achieve this goal based on a principle called *maximizing k-set coverage* and explore a greedy algorithm and a randomized algorithm for efficient summarization.

Finally, when visualizing the satellite packages associated with a central item, the user experience is often affected by the diversity of satellite items encountered in sequential packages. Intuitively, given that most users explore ranked lists in a top-down fashion, there is an ordering of the packages associated with a central item, that *minimizes overlap* between any two consecutive packages and hence, *maximizes their visual diversity*. Our third challenge is therefore to efficiently identify an ordering of the k packages which *maximizes the visual effect of diversity.* We prove that this problem in its generality is NP-Complete and propose an efficient heuristic algorithm for solving it.

In summary, we make the following main contributions:

- We propose the notions of composite item and compatible satellite package in the context of online data exploration. To help users effectively explore composite items, we formalize the problems of finding valid and maximal packages given a budget, finding representative packages through summarization, and reordering packages for visual effect optimization (Section 4.2).
- We design and implement a random walk algorithm to efficiently construct all valid and maximal packages (Section 4.3).
- We introduce a novel principle for summarizing a large set of maximal packages associated with one central item, and develop a max-k set coverage algorithm for efficient summarization. We further improve the efficiency of summarization by integrating it with the random walk package construction algorithm (Section 4.4).
- We formulate the problem of optimizing the visual effect of k packages associated with the same central item as that of finding an ordering of the packages that minimizes overlap between consecutive packages. We prove that this problem is NP-Complete, and design and implement a heuristic algorithm for solving it (Sec-

tion 4.5). In addition, we also prove that this algorithm is optimal when there is only one satellite type.

We conduct extensive experiments on data sets from Yahoo! Shopping site to verify the effectiveness and efficiency of our algorithms (Section 4.6). Finally, we discuss related works and conclude in Sections 4.7 and 5.7, respectively.

4.2 Model and Problem Statement

We start by introducing our data model and some basic definitions, and then we formally state our exploration problem.

Let $\mathcal{C}$ denote the central type (e.g., *iPhone*) and $\mathcal{S} = \{\mathcal{S}_1, \ldots, \mathcal{S}_n\}$, the set of satellite types (e.g., *Case, Speaker*). We refer to an instance of a central (resp., satellite) type as a central (resp., satellite) item. Each item (central or satellite) has a unique identifier id and a set of attributes including a required application-dependent attribute, `cost`. For example, the cost of an item may represent the price for retail products or the visit duration for travel destinations. Compatibility between a central item c and a satellite item s, is provided using the predicate $\texttt{comp}(c, s)$, which is true if c and s are compatible. For example, for products, compatibility can be defined according to manufacturer specifications, based on co-purchasing histories gathered from millions of users, or a combination of the two. Each central type with its set of compatible satellite types form the *composite type*, denoted $[\mathcal{C}, \mathcal{S}_1, \ldots, \mathcal{S}_n]$.

4.2.1 Valid and Maximal Packages

Definition 11 (Satellite Package)**.** *A* satellite package, p, *for a given composite type,* $[\mathcal{C}, \mathcal{S}_1, \ldots, \mathcal{S}_n]$, *is a set of satellite items* $\{s_1, \ldots, s_n\}$, *where each* s_i *is either an item of satellite type* $\mathcal{S}_i$ *or a* `null` *item (shown as symbol "$-$") indicating that* p *does not contain an item of* $\mathcal{S}_i$*.*

A package p is said to be *compatible* with a central item c iff $\forall s \in p, \texttt{comp}(c, s) = true$, i.e., each satellite item s in package p is compatible with the central item c.

Definition 12 (Validity)**.** *Given a budget b, a* valid composite item, *denoted $(c, s_1, \ldots, s_n)$, is an instance of the composite type $[\mathcal{C}, \mathcal{S}_1, \ldots, \mathcal{S}_n]$, s.t. the satellite package $\{s_1, \ldots, s_n\}$ is compatible with the central item c and $(c.\texttt{cost} + \sum_i (s_i.\texttt{cost})) \leq b$. We refer to $\{s_1, \ldots, s_n\}$ as a* valid package.

Budget constraints are typically provided by the user at query time. Depending on the application, it may represent a price (e.g., for retail products), a time constraint (e.g., for travel itineraries), or a combination thereof.

As an example, consider a user shopping an iPhone for less than $\$350$. Assume we have the following table containing five iPhones as central items. Out of the five candidate iPhones, four qualify with price below $\$350$.

Table 4.1. Central Items

iPhone	**memory**	**price**
iPhone 3G	8GB	$99
iPhone 3G	16GB	$199
iPhone 3G S	8GB	$199
iPhone 3G S	16GB	$299
iPhone 3G S	32GB	$399

Also, consider the satellite items in Table 4.2, grouped by type for ease of exposition. There are 7 types in the table. Assume, for simplicity, that all satellite items in Table 4.2 are compatible with all available iPhones in Table 4.1. Table 4.3 then lists some of the valid packages along with their central items, given the budget of $\$350$.

As shown in the example, even with a small number of satellite items, the number of valid packages can quickly become overwhelming. Therefore, we define the notions of *valid and maximal package* (or simply *maximal package*) and *maximal composite item*.

Table 4.2. Satellite Items and their Price

type	**item**	**price**
Case	s^1_{case}: Kroo Case	$14.95
	s^2_{case}: Belkin Sport Case	$29.95
	s^3_{case}: Mesh Sport Case	$18.95
	s^4_{case}: Folio Leather Case	$39.95
Charger	$s^1_{charger}$: CarFM Charger	$59.95
	$s^2_{charger}$: Kensington Deluxe Charger	$99.00
	$s^3_{charger}$: Insipio Car Charger	$24.95
	$s^4_{charger}$: Wireless Car Charger	$14.95
Kit	s^1_{kit}: iKlear Spray Kit	$24.95
	s^2_{kit}: iPhone wipes	$9.95
Cable	s^1_{cable}: Dock 2ft Cable	$19.95
	s^2_{cable}: Belkin Stereo Cable	$14.95
	s^3_{cable}: Kroo USB Cable	$34.95
Speaker	$s^1_{speaker}$: Twin Speaker	$29.95
	$s^2_{speaker}$: Portable Bose Sounddock	$149.00
	$s^3_{speaker}$: Scosche Speaker Dock	$64.95
Screen	s^1_{screen}: AntiGlare Screen	$6.95
	s^2_{screen}: BodyGuardz Screen	$24.95
	s^3_{screen}: Macally Mirror Screen	$14.95
	s^4_{screen}: Zagg Invisible Shield	$66.00
Pen	s^1_{pen}: Touch Pen	$19.95
	s^2_{pen}: Kroo Stylus	$9.75

Table 4.3. Examples of Valid Satellite Packages

central item/capacity	**satellite packages**	**total price**
iPhone 3G/8GB	$\{s^1_{case}, s^1_{charger}, s^1_{kit}, s^1_{cable}, s^1_{speaker}, s^2_{screen}, s^1_{pen}\}$	$273.70
iPhone 3G/8GB	$\{s^4_{case}, s^2_{charger}, -, s^3_{cable}, -, s^1_{screen}, s^1_{pen}\}$	$299.80
iPhone 3G/8GB	$\{-, -, -, -, s^2_{speaker}, -, s^2_{pen}\}$	$257.75
iPhone 3G/8GB	$\{s^2_{case}, s^4_{charger}, -, s^2_{cable}, s^3_{speaker}, s^4_{screen}, s^1_{pen}\}$	$309.75
iPhone 3G/8GB	... ...	...
iPhone 3G/16GB	$\{s^2_{case}, s^4_{charger}, -, -, s^3_{speaker}, s^3_{screen}, s^1_{pen}\}$	$343.75
iPhone 3G/16GB	... ...	...
...	... ...	...

Definition 13 (Maximality)**.** *Given a central item and a budget constraint, a* maximal package *is a valid package, to which no further satellite item can be added without violating the validity. A maximal package, together with its associated central item, form a* maximal composite item*.*

For example, the two packages $\{s^4_{case}, s^2_{charger}, s^1_{kit}, s^4_{screen}, s^1_{pen}\}$ and $\{s^2_{speaker}\}$ form maximal composite items with the central item *iPhone 3G/8GB* and *iPhone 3G S/8GB*, respectively. Hence, any strict subset of those packages is not maximal. We now define our first technical problem of maximal package construction.

PROBLEM (Maximal Package Construction.)

Given a central item c, and a budget b, efficiently compute the maximal composite item set $\mathcal{M}_c$ formed by the set of valid composite items, which share the same central item c, s.t., the package within each composite item is maximal.

Examining maximal composite items, rather than enumerating all valid composite items, is useful to an end user because it drastically reduces the number of packages to be explored while preserving all compatible satellite items. At the end, users can always choose a subset of the items in the package to continue their transaction. We discuss our solution to the above problem in Section 4.3.

4.2.2 Summarization

While it is much smaller than the set of all valid packages, $\mathcal{M}_c$ can still become very large in practice. More importantly, different maximal packages associated with the same central item, may overlap significantly in their satellite items. For example, both $\{s^2_{case}, s^4_{charger}, s^3_{cable}, s^3_{speaker}\}$ and $\{s^2_{case}, s^4_{charger}, s^3_{speaker}, s^3_{screen}, s^1_{pen}\}$ are maximal packages w.r.t. the central item *iPhone 3G/16GB* (for a budget of $\$350$), but they overlap considerably. Hence, in addition to finding maximal packages, we further propose to *summarize*

$\mathcal{M}_c$ into a smaller set $\mathcal{I}_c$, containing k *representative* packages (typically $5 - 10$). We now define the summarization problem.

PROBLEM (Summarization.) *Given a maximal composite item set $\mathcal{M}_c$ and k, efficiently compute a set $\mathcal{I}_c$ of k representative packages from $\mathcal{M}_c$, s.t. the number of packages in $\mathcal{M}_c$ represented by the k packages in $\mathcal{I}_c$ is maximized.*

We refer to the output set $\mathcal{I}_c$ as the set of summary packages, or *summary set*. The motivation is to present to the user a short list of k maximal packages and yet represent as many valid packages as possible, thus offering the widest choice to the user. Table 4.4 shows two examples of maximal composite item sets containing four representative packages each associated with the iPhone 3G/8GB. We discuss our summarization solution in Section 4.4.

Table 4.4. Two Sets of Summary Packages for Central Item *iPhone 3G/8GB*

$p_1 =$	$\{s^1_{case},$	$s^1_{charger},$	$s^1_{kit},$	$s^1_{cable},$	$s^1_{speaker},$	$s^2_{screen},$	$s^1_{pen}\}$
$p_2 =$	$\{s^4_{case},$	$s^2_{charger},$	$-$,	$s^3_{cable},$	$-$,	$s^1_{screen},$	$s^1_{pen}\}$
$p_3 =$	$\{$ $-$,	$-$,	$-$,	$-$,	$s^1_{speaker},$	$-$,	$s^1_{pen}\}$
$p_4 =$	$\{s^4_{case},$	$s^2_{charger},$	$-$,	$s^3_{cable},$	$-$,	$s^1_{screen},$	$s^1_{pen}\}$

$p_1 =$	$\{s^1_{case},$	$s^1_{charger},$	$s^1_{kit},$	$s^1_{cable},$	$s^1_{speaker},$	$s^2_{screen},$	$s^1_{pen}\}$
$p_2 =$	$\{s^1_{case},$	$s^1_{charger},$	$-$,	$s^3_{cable},$	$-$,	$s^1_{screen},$	$s^1_{pen}\}$
$p_3 =$	$\{s^1_{case},$	$s^4_{charger},$	$-$,	$s^2_{cable},$	$s^3_{speaker},$	$-$,	$s^1_{pen}\}$
$p_4 =$	$\{s^2_{case},$	$s^4_{charger},$	$-$,	$s^2_{cable},$	$s^3_{speaker},$	$s^1_{screen},$	$s^1_{pen}\}$

4.2.3 Visual Effect

The next challenge after obtaining k summary packages for a given central item, is to effectively present them to the user, typically in a ranked list format. While ranking packages according to a particular attribute (such as price) is desirable in certain scenarios (e.g., when the user is looking for the cheapest package), it is not always applicable. For many users, once the package satisfies their budget, price is no longer a critical factor in their purchase decision, and many other factors come into play. One such factor is *diversity*, i.e., the user will like to explore many different packages associated with a given central item, quickly. Our summarization technique addresses diversity to a certain extent since it aims at returning representative packages. However, it may still return packages sharing satellite items. Hence, we introduce *visual effect*, a new principle which guides how a set of packages associated with the same central item, should be ranked in order to expose users to as many different satellite items as early as possible in their exploration process.

The visual effect principle aims to sort a set of packages $\mathcal{I}_c$ associated with a central item c, such that *presenting a package that is too similar to a package the user has just seen*, is avoided. This is particularly important for satellite types which matter to the user. Hence, to formally define the visual effect principle, we introduce the notion of *satellite type prioritization*, denoted $\mathcal{O} = \mathcal{S}_1 \prec \mathcal{S}_2 \prec \ldots \prec \mathcal{S}_m$, which indicates the *visual order of importance* of satellite types $\mathcal{S}_i$ to a user, meaning that it is more important to ensure diversity in $\mathcal{S}_1$ than in $\mathcal{S}_2$, and so on. Indeed, while one user looking for an iPhone may prefer seeing variety in chargers over seeing variety in speakers, another user may prefer variety in protective screens over variety in cables, etc. A default prioritization can often be set if it is not provided by the user. We can now define the notion of *penalty*.

Definition 14 (Penalty). *Given a satellite type prioritization, $\mathcal{O} = \mathcal{S}_1 \prec \mathcal{S}_2 \prec \ldots \prec \mathcal{S}_m$, and two packages p_1 and p_2 associated with the same central item, the* pair penalty *between p_1 and p_2 is a vector, $pv(p_1, p_2) = \langle v_1, v_2, \ldots, v_m \rangle$, where $v_i = 1$ if p_1 and p_2 share the same item on type $\mathcal{S}_i$, and $v_i = 0$ for all other scenarios, including the cases where one of the two packages does not have an item for type $\mathcal{S}_i$. Let $pv(p_1, p_2)[i]$ refer to v_i.*

Hence, we define the penalty *for an ordering of packages associated with the same central item c, $\mathcal{P}_c = [p_1, p_2, \ldots, p_k]$, as a vector, $pv(\mathcal{P}_c) = \langle a_1, a_2, \ldots, a_m \rangle$, where $a_i =$*

$\sum_{j=1}^{k-1}(pv(p_j, p_{j+1})[i])$. $pv(\mathcal{P}_c)$ *is an aggregation over the pair penalties of all consecutive packages in* $\mathcal{P}_c$*.*

Intuitively, the penalty vector of an ordering of packages associated with the same central item, keeps track of the number of times the same satellite item has appeared in consecutive packages. It is a good indicator of how *visually diverse* the ranked list of packages appears to the user. As an example, let us examine the two summary sets associated with iPhone 3G/8GB in Table 4.4. The first ordering $[p_1, p_2, p_3, p_4]$, has penalty $\langle 0, 0, 0, 0, 0, 0, 3\rangle$ which is computed by aggregating pairwise penalties in the ordering: $pv(p_1, p_2)$, $pv(p_2, p_3)$, and $pv(p_3, p_4)$. For example, given $p_1 = (s^1_{case}, s^1_{charger}, s^1_{kit}, s^1_{cable}, s^1_{speaker}, s^2_{screen}, s^1_{pen})$, $p_2 = (s^4_{case}, s^2_{charger}, \; - \;, s^3_{cable}, \; - \;, s^1_{screen}, s^1_{pen})$
we have $pv(p_1, p_2) = \langle 0, 0, 0, 0, 0, 0, 1\rangle$. The penalty for the second set of packages (in their listed order) is $\langle 2, 2, 0, 1, 1, 0, 3\rangle$.

We now formally define our third technical problem of finding a package ordering with the *optimal visual effect*.

PROBLEM (Visual Effect Optimization.) *Given a set* $\mathcal{I}_c$ *of* k *packages associated with the same central item* c *and a satellite type prioritization* $\mathcal{O} = \mathcal{S}_1 \prec \mathcal{S}_2 \prec \ldots \prec \mathcal{S}_m$*, find an ordering* $\mathcal{P}_c$ *of the packages s.t.,* $\forall \mathcal{P}'_c, \mathcal{P}_c \neq \mathcal{P}'_c$*:*

- $pv(\mathcal{P}_c)[1] < pv(\mathcal{P}'_c)[1]$*, or*
- $\forall i, 0 < i < m, pv(\mathcal{P}_c)[i] = pv(\mathcal{P}'_c)[i]$*, or*
- $\exists h, \forall i, 0 < i < h$*,* $pv(\mathcal{P}_c)[i] = pv(\mathcal{P}'_c)[i]$*,* $pv(\mathcal{P}_c)[h] \leq pv(\mathcal{P}'_c)[h]$.

Intuitively, the ordering with optimal visual effect incurs smaller penalties on higher priority types. We discuss the problem complexity and a heuristic algorithm in Section 4.5.

4.3 Maximal Package Construction

Recall from Section 4.2.1 that a maximal package is a set of satellite items associated with a central item where 1) each satellite item is compatible with the central item, 2) the total `cost` of the package and central item is within budget, and 3) there is no other valid package containing it as a proper subset. Given a central item, our first technical challenge is to construct its set of maximal packages, $\mathcal{M}_c$, efficiently.

This problem is closely related to frequent (maximal) itemset mining (FIM) [55, 56], where the goal is to identify (maximal) sets of items that co-occur frequently (i.e., above a certain *support* threshold) in a transaction database. There are two main differences, however, between this problem and our maximal package construction problem. First, the candidate itemsets in FIM are limited to items appearing within the database transactions, while the packages in our problem need to be constructed, subject to compatibility and budget constraints. Second, checking the satisfaction of an itemset against the *support* threshold requires scanning through the transaction database, while the *budget* constraint in our problem, can be checked using the cost of each item in the package itself, which makes our problem easier.

Given its resemblance to FIM, one straight-forward algorithm to solve our problem is to adapt the Apriori-style algorithms [55]. This algorithm simply iterates through packages level-wise (i.e., single-item packages first, then two-item packages, etc.), selecting compatible packages and eliminating those that no longer satisfy the budget or that can be subsumed by another larger package satisfying the budget. The result is the correct maximal composite item set $\mathcal{M}_c$.

Constructing the correct $\mathcal{M}_c$ using an Apriori algorithm is costly when the results have to be computed and returned to the user in real time. The number of valid packages to go through can be overwhelming when the number of satellite items is large, which is typically the case. As a result, we propose an alternative algorithm (adapted from [57]), `MaxCompositeItemSet`, that computes an approximate $\mathcal{M}_c$ based on random walks.

4.3.1 Algorithm MaxCompositeItemSet

Algorithm 7 illustrates our random walk algorithm. Intuitively, it constructs random maximal packages one at a time and stops after each current random maximal package has been generated at least twice. The routine `MaxCompositeItem` (Figure 4.1) accomplishes the random walk procedure. It starts from a random single item package and picks the next random item which is different from previously added items and which satisfies compatibility, and validity until the package is maximal.

Algorithm 7: `MaxCompositeItemSet`$(c, \mathcal{A}, b)$: computing maximal composite item set $\mathcal{M}_c$

```
Require:
    $c$, the central item,
    $\mathcal{A}$, the set of all satellite items compatible with $c$,
    $b$, the budget constraint
$\mathcal{M}_c = \{\}$
repeat
    $p =$ MaxCompositeItem($c, \mathcal{A}, b$)
    if $p \notin \mathcal{M}_c$ then
        $\mathcal{M}_c = \mathcal{M}_c \cup \{p\}$
        $count(p) = 1$
    else
        $count(p) + +$;
until $\{\forall p \in \mathcal{M}_c, count(p) \geq 2\}$
return $\mathcal{M}_c$ ;
```

We illustrate this algorithm with the running iPhone example from Section 4.2.

```
Function MaxCompositeItem($(c, \mathcal{A}, b)$) : Subroutine for computing one maximal package $p$
Require:
    $c$, the central item,
    $\mathcal{A}$, the set all satellite items compatible with $c$,
    $b$, the budget constraint
$p = \{\}$
pick a random $a \in \mathcal{A}$, add $a$ to $p$
repeat
    pick a random $a \in \mathcal{A}, a \notin p$, such that: (1) $\forall s \in \mathcal{A}$, $a$ and $s$ are of different types, (2) $a$ is compatible with $c$, and (3) $a.\texttt{cost} + \sum_{s_i \in \mathcal{A}}(s_i.\texttt{cost}) \leq b$
    add $a$ to $p$
until $\{$no new item can be added$\}$
return $p$
```

Figure 4.1. **Function** `MaxCompositeItem`.

Example 9. *Consider the central item, iPhone 3G/16GB (costing* $\$199$*), and a total price budget of* $\$300$*, which means a total of* $\$101$ *as the price budget for the satellite package. Assume there are* 5 *satellite items that are compatible with the central item:* $s^1_{kit}(\$24.95)$, $s^3_{cable}(\$34.95)$, $s^3_{speaker}(\$64.95)$, $s^4_{screen}(\$66.00)$*, and* $s^2_{pen}(\$9.95)$.

The set of maximal packages in this example are:

$\{s^1_{kit}, s^3_{cable}, s^2_{pen}\}$, $\{s^3_{cable}, s^3_{speaker}\}$, $\{s^3_{cable}, s^4_{screen}\}$,

$\{s^1_{kit}, s^3_{speaker}, s^2_{pen}\}$, $\{s^1_{kit}, s^4_{screen}, s^2_{pen}\}$.

The algorithm will randomly construct one of those five packages at each iteration, keep counts of the packages it has seen so far, and stop when the counts of every seen packages is at least two. Figure 4.2 depicts the random walk process as selecting random paths in the package lattice. Algorithm 7 may not generate the full $\mathcal{M}_c$. For example, it may construct each of the first four packages twice before seeing the last package, in which case, it will produce an approximate (i.e., incomplete) $\mathcal{M}_c$ instead. We discuss the algorithm termination condition and the probability of finding all of $\mathcal{M}_c$ next.

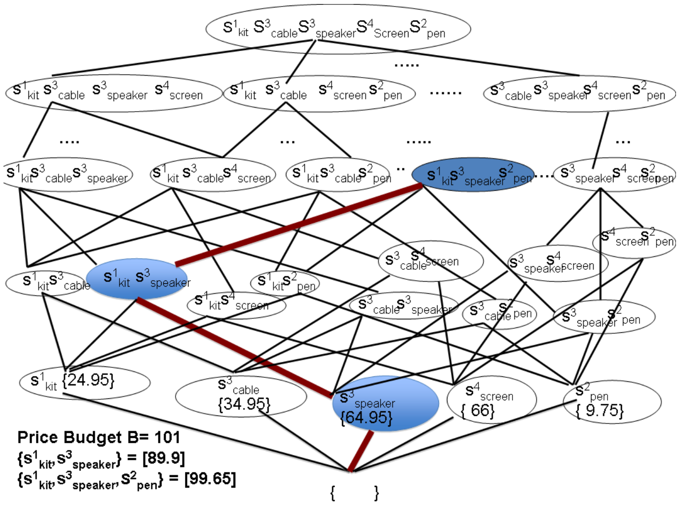

$S^1_{kit} S^3_{cable} S^3_{speaker} S^4_{Screen} S^2_{pen}$
$s^1_{kit} s^3_{cable} s^3_{speaker} s^4_{screen}$
$s^1_{kit} s^3_{cable} s^4_{screen} s^2_{pen}$
$s^3_{cable} s^3_{speaker} s^4_{screen} s^2_{pen}$
$s^1_{kit} s^3_{cable} s^3_{speaker}$
$s^1_{kit} s^3_{cable} s^4_{screen}$
$s^1_{kit} s^3_{cable} s^2_{pen}$
$s^1_{kit} s^3_{speaker} s^2_{pen}$
$s^3_{speaker} s^4_{screen} s^2_{pen}$
$s^4_{screen} s^2_{pen}$
$s^3_{cable} s^4_{screen}$
$s^3_{speaker} s^4_{screen}$
$s^1_{kit} s^3_{cable}$
$s^1_{kit} s^3_{speaker}$
$s^1_{kit} s^2_{pen}$
$s^1_{kit} s^4_{screen}$
$s^3_{cable} s^3_{speaker}$
$s^3_{cable} s^2_{pen}$
$s^3_{speaker} s^2_{pen}$
s^1_{kit} {24.95}
s^3_{cable} {34.95}
$s^3_{speaker}$ {64.95}
s^4_{screen} { 66}
s^2_{pen} { 9.75}
Price Budget B= 101
$\{s^1_{kit}, s^3_{speaker}\}$ = [89.9]
$\{s^1_{kit}, s^3_{speaker}, s^2_{pen}\}$ = [99.65]
{ }

Figure 4.2. Random Walk on Item Lattice.

4.3.2 Termination Condition

The termination condition used in Algorithm 7 is inspired by the *Good Turing Test* that is often used in population studies to determine the number of unique species in a large unknown population [58]. Consider a large population of individuals drawn from an unknown number of species with diverse frequencies, including a few common species, some with intermediate frequencies, and many rare species. Let us draw a random sample of N individuals from this population, which results in n_1 individuals that are the lone representatives of their species, and the remaining individuals belong to species that contain multiple representatives in the sample population. Then, P_0, which represents the frequency of all unseen species in the original population can be estimated using the following Lemma:

Lemma 4.3.1 (Good Turing Test)**.** $P_0 = n_1/N$.

The assumption here is that the overall probability of hitting one rare species is high while the probability of hitting the same rare species is low. Therefore, the more the sample hits the rare species multiple times, the less likely there are unseen species in the original population. We apply Lemma 4.3.1 to the maximal package construction problem, where the maximal packages map to the species and the probabilities of constructing each maximal package in `MaxCompositeItem` are the frequencies. The set of maximal packages constructed through the random walk process is the sample population. By ensuring this process visits each constructed maximal package at least twice, we are essentially ensuring that n_1 is 0. Thus, using Lemma 4.3.1, P_0 can be estimated to be 0, which means it is highly likely that all maximal packages have been discovered.

4.4 Summarization

Presenting the full set of maximal packages to the user directly has two main challenges as discussed in Section 4.2.2. First, the number of maximal packages can be extremely large for effective exploration by the user. Second, there can be significant overlaps between the maximal packages. The goal of summarization is therefore to find k *representative* maximal packages for further exploration by the user.

One commonly adopted approach for summarization is clustering. Specifically, a pair-wise distance measure can be defined to measure the distance between any two packages. Then, various clustering algorithms (e.g., k-means) can be used to group the packages into k clusters, and one package can be selected from each cluster to form the k representatives. However, defining a good distance measure in our case is difficult. For example, Jaccard distance can not tell the difference between a pair of single-item packages and a pair of multiple-item packages, as long as there is no overlapping item in either pair.

In this work, we explore a different approach to summarization by leveraging the principle of *maximizing coverage*. Specifically, we consider the goal of summarization as the following: maximizing the number of valid packages a user can construct with the k maximal packages, where we consider a package is *constructible* if it is subsumed by (i.e., is a subset of) one of the k maximal packages. Intuitively, this provides the user with

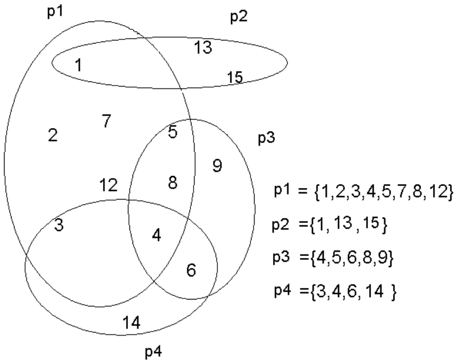


Figure 4.3. Example Maximal Packages to be Summarized..

the highest flexibility in creating a desired package without worrying about checking the budget constraints. Formally, we have:

Definition 15 (Set Coverage)**.** *Given a set of packages $\mathcal{M} = \{m_1, m_2, \ldots, m_n\}$, let $\mathcal{I} = 2^{m_1} \bigcup 2^{m_2} \bigcup \ldots \bigcup 2^{m_n}$ be the union of all powersets of the individual packages in $\mathcal{M}$, the* set coverage *of $\mathcal{M}$, denoted $Coverage(\mathcal{M})$, is $|\mathcal{I}|$, the number of unique sets in the union.*

The goal of summarization is therefore to compute a set of k representative maximal packages $\mathcal{I}_c$ such that $Coverage(\mathcal{I}_c)$ is maximized.

This principle is better illustrated in Figure 4.3, where the numbers indicate satellite items and the circles indicate maximal packages. (For simplicity, we adopt abstract items in this example and assume that these are all the possible valid maximal packages.) Assume we want to pick 2 packages out of the 4 total packages (i.e., $k = 2$). Selecting p_1 and p_3 (which turns out to be the best summary in this example) will allow the user to construct a total of 279 unique valid packages: 255 packages can be constructed from the 8-item package p_1 and 31 packages can be constructed from the 5-item package p_3, minus the 7 packages that are doubled-counted because of the 3-item overlap between the two packages. In contrast, selecting the two non-overlapping packages p_2 and p_3 will only give us 38 constructible packages.

Function `ComputeCoverage`($\mathcal{M}$) : Subroutine for Coverage Computation

```
Require:
    $\mathcal{M} = \{m_1, m_2, \ldots, m_n\}$, the set of packages
$M_1 = \sum_{i=1}^{n}(2^{|m_i|} - 1)$
$M_2 = \sum_{1\leq i<j\leq n}^{n}(2^{|m_i \bigcap m_j|} - 1)$
$M_3 = \sum_{1\leq i<j<k\leq n}^{n}(2^{|m_i \bigcap m_j \bigcap m_k|} - 1)$
$\ldots$
$M_n = 2^{|m_1 \bigcap \cdots \bigcap m_n|} - 1$
$C = M_1 - M_2 + M_3 - \ldots (-1)^{n-1} M_n$
return $C$
```

Figure 4.4. Function `ComputeCoverage`.

Intuitively, the coverage of a set of packages can be computed based on the *Inclusion-Exclusion Principle* [59] (a standard technique for deriving the cardinality of the union of a set of sets) using the procedure described in Figure 4.4. This naive way of coverage computation has an exponential complexity, since each M_i may require the summation of an exponential number of terms. As a result, summarization by maximizing coverage turns out to be a hard problem.

To address this performance challenge, in Section 4.4.1, we introduce a baseline greedy algorithm and a fast greedy algorithm for efficiently computing k summary packages, with a coverage that is within a bounded factor of the optimal coverage. Furthermore, in Section 4.4.2, we show that the performance can be further improved by generating summary packages directly from individual items, a process inspired by the random walk process in Section 4.3.

4.4.1 Greedy Summarization Algorithms with Bounded Approximation Factors

We first present the baseline `GreedySummarySet`, which is shown in Algorithm 15. The algorithm starts by selecting the largest package (i.e., the package with the largest number of items). At each iteration, it selects the package that, together with the previously chosen packages, produces the highest coverage (as computed by Function `ComputeCoverage` in Figure 4.4). The algorithm stops after k packages have been chosen. Consider again the example in Figure 4.3: when $k = 2$, Algorithm 15 produces the summary $\{p_1, p_3\}$, and when $k = 3$, it produces the summary $\{p_1, p_3, p_4\}$.

Algorithm 8: GreedySummarySet($\mathcal{M}_c, k$) : Algorithm for computing k summary packages

```
Require:
    $\mathcal{M}_c$, the set of maximal packages for central item $c$,
    $k$, desired number of summary packages
$\mathcal{I}_c = \{\}$
let package $p$ be the largest package in $\mathcal{M}_c$
remove $p$ from $\mathcal{M}_c$
add $p$ to $\mathcal{I}_c$
$iteration = 1$
while $iteration \leq k$ do
    $p = argmax_{p \in \mathcal{M}_c}$(ComputeCoverage($\mathcal{I}_c \bigcup \{p\}$))
    remove $p$ from $\mathcal{M}_c$
    add $p$ to $\mathcal{I}_c$
    $iteration++$
return $\mathcal{I}_c$
```

This baseline algorithm is directly adapted from a greedy approximate algorithm designed for the *Maximum k-Set Cover* problem [60], which is defined as follows. Given a set of sets X over a set of elements E, find k sets in X such that the union of the k sets is maximized. Our summarization problem can be mapped to the Maximum k-Set Cover problem by considering each subset of $\mathcal{M}_c$ as an element in E. The greedy approximate algorithm for the *Maximum k-Set Cover* problem is known to have a $(1-1/e)$ approximation ratio [60], therefore we have:

Lemma 4.4.1. *Given the set of maximal packages $\mathcal{M}_c$, let the optimal set of k packages be $\mathcal{I}_c^{opt}$ and the set of k packages returned by* `GreedySummarySet` *be $\mathcal{I}_c^{greedy}$, then $\frac{Coverage(\mathcal{I}_c^{greedy})}{Coverage(\mathcal{I}_c^{opt})} \geq (1 - 1/e)$, where e is the base of the natural logarithm.*

Because of the need to compute the coverage of multiple sets at each iteration, Algorithm 15 can still be quite expensive in practice. We describe `FastGreedySummarySet` (Algorithm 9) that improves upon the performance of `Greedy SummarySet`, while producing the same output (therefore maintaining the same approximation bound). The key idea within the fast greedy algorithm is to leverage Bonferroni upper and lower bounding techniques [59] to speed up the coverage calculations, and make sure the decision made in each iteration of `FastGreedySummarySet` is *exactly the same* as the decision made by `GreedySummarySet`.

Algorithm 9: `FastGreedySummarySet`$(\mathcal{M}_c, k)$: Algorithm for computing k summary packages

```
Require:
    $\mathcal{M}_c$, the set of maximal packages for central item $c$,
    $k$, desired number of summary packages
$\mathcal{I}_c = \{\}$
$iteration = 1$
while $iteration \leq k$ do
    $r = -1$
    repeat
        $r = r + 2$
        for $p \in \mathcal{M}_c$ do
            $p$.lower = BonferroniLower$(\mathcal{I}_c \bigcup \{p\}, r)$
            $p$.upper = BonferroniUpper$(\mathcal{I}_c \bigcup \{p\}, r)$
        $p_1 = argmax_{p' \in \mathcal{M}_c}(p'$.lower$)$
        $p_2 = argmax_{p' \in \mathcal{M}_c, p' \neq p_1}(p'$.upper$)$
    until $(p_1.lower \geq p_2.upper)$
    remove $p_1$ from $\mathcal{M}_c$
    add $p_1$ to $\mathcal{I}_c$
    $iteration++$
return $\mathcal{I}_c$
```

The algorithm estimates the coverage using the *Bonferroni Inequalities* [59] with a depth parameter r, an odd number between 1 and n where n is the total number of packages in $\mathcal{M}_c$. Specifically, the lower and upper bound estimates of the coverage can be computed as: `BonferroniLower`$(\mathcal{M}, r) = M_1 - M_2 + M_3 - \ldots + M_r - M_{r+1}$, and `BonferroniUpper`$(\mathcal{M}, r) = M_1 - M_2 + M_3 - \ldots + M_r$. When r is relatively small compared to n, those bounds can be computed efficiently. While `GreedySummarySet` computes the exact coverage of each candidate package at each iteration, `FastGreedySummarySet` considers the candidate packages in a round-robin manner and computes the (increasingly tighter) lower and upper bounds of the coverage by gradually increasing r. Furthermore, when r is incremented, the upper and lower bounds can be computed incrementally from those computed earlier with a smaller value of r. At each iteration, a package is chosen when its coverage lower bound is no smaller than the coverage upper bounds of the remaining candidates. The idea of leveraging the lower and upper bounds is motivated by the TA-style algorithms developed for top-k ranking problems [61], since the function `ComputeCoverage` exhibits a monotonic behavior with increasing r.

4.4.2 Randomized Summarization Algorithm

Both greedy algorithms described in Section 4.4.1 take as input the full set of maximal packages $\mathcal{M}_c$. As a result, their performance is constrained by the package construction time (i.e., Algorithm 7). In practice, the number of maximal packages can be large and therefore limits how fast the summary can be generated. In this section, we describe a randomized algorithm, `ProbSummarySet`, that produces k representative packages directly from the set of compatible satellite items, without generating the full set of maximal packages first.

Algorithm 10: `ProbSummarySet`$(\mathcal{V}_c, b, k)$: Randomized Algorithm for computing k summary packages

```
Require:
    $\mathcal{V}_c$, the set of all satellite items across all satellite types for the central item $c$,
    $b$, the budget,
    $k$, desired number of summary packages
$\mathcal{I}_c = \{\}$
$i = 1$
for $a \in \mathcal{V}_c$ do
    $a.\texttt{seenCnt} = 1$
while $i \leq k$ do
    $p = \texttt{SelectRepresentative}(\mathcal{V}_c, b)$
    if $p \notin \mathcal{I}_c$ then
        add $p$ to $\mathcal{I}_c$
        for $a \in p$ do
            $a.\texttt{seenCnt} + +$
    $i + +$
return $\mathcal{I}_c$ ;
```

As shown in Algorithm 10, `ProbSummarySet` has the same overall structure as `MaxCompositeItemSet` (Algorithm 7), i.e., it makes similar random walks to generate a set of maximal packages. There are two main differences. First, Algorithm 10 stops as soon as k packages are generated. Second, more importantly, each random walk (Function `Select Representative` in Figure 4.5) invoked from within Algorithm 10 is designed to generate a package that is as "different" as possible from the packages already discovered by the previous random walks, thus maximizing the potential coverage of the resulting set of maximal packages.

We now explain the rationale behind the computation of the probabilities of items being chosen (inside Function `SelectRepresentative`, lines 1-4). Consider the ith iteration and assume that $\mathcal{I}_c = \{m_1, m_2, \ldots, m_{i-1}\}$ is the current set of packages already chosen by the algorithm. For each item $a \in \mathcal{V}_c$, the algorithm keeps track of the number of packages in $\mathcal{I}_c$ that contain a ($a.\texttt{seenCnt}$). The algorithm then selects the next item with probability inversely proportional to its $a.\texttt{seenCnt}$. The intuition is that if an item has already appeared in many chosen packages, picking it again will not increase the coverage by much. The probability also inversely depends on the cost of the item. The intuition for this is that packages with items of lower costs can admit more items, hence, leading higher coverage.

As an example, consider Example 9 and the corresponding item lattice in Figure 4.2 and assume that Algorithm 10 discovers the maximal satellite package
$p_1 = \{s^1_{kit},\ s^3_{speaker}, s^2_{pen}\}$ during the first iteration of the random walk. In the second iteration, the probabilities of the items that appear in p_1 are reduced. For example, item s^1_{kit} now gets a 16% probability of being chosen, compared against its 20% probability in the first iteration, whereas items $s^3_{speaker}$ and s^2_{pen} now get 6% and 42% probabilities, respectively. On the other hand, the remaining items s^3_{cable} and s^4_{screen}, which have 14% and 7% probabilities, respectively, in the first random walk, are now given higher probabilities of 24% and 12%, respectively. (Note that, the cheaper item s^3_{cable} gains higher probability, although it appears in the same number of chosen packages as s^4_{screen}.)

While there is no approximation guarantee that can be provided for `ProbSummarySet`, it runs much faster than the greedy algorithms since it bypasses the computation of the full set of maximal packages. As shown in Section 4.6, we found this randomized summarization algorithm to work very well in practice.

Function SelectRepresentative$(\mathcal{V}_c, b)$: Subroutine for selecting one random package

```
Require:
    $\mathcal{V}_c$, the set of all satellite items across all satellite types for the central item $c$, with their seenCnts
    $b$, the budget,
$P = \sum_{a \in \mathcal{V}_c} \{\frac{1}{a.\texttt{seenCnt}} \times \frac{1}{a.\texttt{cost}}\}$
for $a \in \mathcal{V}_c$ do
    $a.\texttt{probiblity} = \frac{1}{a.\texttt{seenCnt} \times a.\texttt{cost} \times P}$
$p = \{\}$
repeat
    pick $a \in \mathcal{V}_c, a \notin p$, with probability $a.\texttt{probility}$, such that: (1) $\forall s \in \mathcal{A}$, $a$ and $s$ are of different types, (2) $a$ is compatible with $c$, and (3) $a.\texttt{cost} + \sum_{s_i \in \mathcal{A}}(s_i.\texttt{cost}) \leq b$
    add $a$ to $p$
until {no new item can be added}
return $p$
```

Figure 4.5. Function SelectRepresentative.

4.5 Visual Effect Optimization

While summarization drastically reduces the number of packages to be explored by the user, the challenge of presenting the final k packages to the user still remains. As discussed in Section 4.2.3, we propose a new principle called *visual effect* to guide how a set of packages should be ordered and presented to the user to achieve better visual diversity. Optimal visual effect is achieved when the cumulative penalty between consecutive packages (i.e., common satellite items) in the ordering is minimized at higher priority satellite types, given a satellite type prioritization. In this section, we consider how to solve the problem of *identifying the package ordering with optimal visual effect*. We begin by recalling the second set of packages in Table 4.4:

Example 10. *Consider the following four packages:*

$$p_1 = (s^1_{case}, s^1_{charger}, s^1_{kit}, s^1_{cable}, s^1_{speaker}, s^2_{screen}, s^1_{pen}),$$

$$p_2 = (s^1_{case}, s^1_{charger}, \; - \;, s^3_{cable}, \; - \;, s^1_{screen}, s^1_{pen}),$$

$$p_3 = (s^1_{case}, s^4_{charger}, \; - \;, s^2_{cable}, s^3_{speaker}, \; - \;, s^1_{pen}),$$

$p_4 = (s^2_{case}, s^4_{charger}, \; - \;, s^2_{cable}, s^3_{speaker}, s^1_{screen}, s^1_{pen}),$

Let the type priority be $\mathcal{O} = \mathcal{S}_{case} \prec \mathcal{S}_{charger} \prec \mathcal{S}_{kit} \prec \mathcal{S}_{cable} \prec \mathcal{S}_{speaker} \prec \mathcal{S}_{screen} \prec \mathcal{S}_{pen}$*. Among the* 24 *possible orderings,* $[p_1, p_4, p_2, p_3]$ *is one of the two optimal orderings, with penalty* $\langle 1, 0, 0, 0, 0, 0, 3\rangle$*. This penalty indicates that the ordering incurs one penalty point (i.e., same satellite item for one consecutive package pair) for type* $\mathcal{S}_{case}$ *(between* p_2 *and* p_3*), three penalty points for type* $\mathcal{S}_{pen}$ *(between all three consecutive pairs), and none for the other five types.*

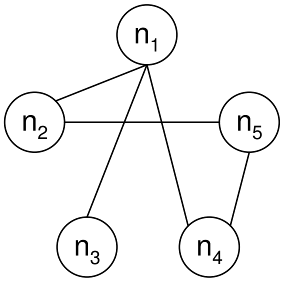
Identifying the ordering with the optimal visual effect turns out to be a hard problem. In Section 4.5.1, we give the proof sketch that the visual effect optimization problem is NP-complete. As a result, we design a heuristic algorithm in Section 4.5.2 and show that it is optimal when there is only one satellite type.

4.5.1 Visual Effect Optimization is NP-Complete

Lemma 4.5.1. *The* visual effect optimization *problem is NP-Complete for* m *satellite types, where* m *is bounded by* n*, the number of packages.*

Proof Sketch: To prove this, we use a reduction from the NP-complete Hamiltonian Path problem.

Consider the following problem: *Given a set of packages and a type priority ordering, check if an ordering* $\mathcal{P}$ *of the packages exists such that* $\forall i, pv(\mathcal{P})[i] = 0$. If we can solve the visual effect optimization problem in polynomial time, then this new problem can be solved in polynomial time by producing an ordering with the optimal visual effect, and checking whether the penalty vector of the result ordering contains all zeros. The process of checking can be accomplished in $O(mn)$, where n is the number of packages and m is the number of satellite types. Therefore, to prove that the visual effect optimization problem is NP-Complete, we just need to show this new problem is NP-Complete.



	S_1	S_2	...
p_1	$s^1{}_1$	$s^2{}_2$	
p_2	$s^2{}_1$	$s^3{}_2$	
p_3	$s^2{}_1$	$s^1{}_2$	
p_4	$s^2{}_1$	$s^4{}_2$	
p_5	$s^1{}_1$	$s^1{}_2$	

Figure 4.6. Transforming a graph into packages..

Given a graph G, we can transform it into a set of packages S in polynomial time and show that an optimal ordering of the packages with an all-zero penalty vector exists if and only if a Hamiltonian path exists for G.

Due to lack of space, we omit the details of the full transformation and only provide a brief description here. Basically, each node n_i in the graph G corresponds to one package p_i. For any edge (n_i, n_j) in the graph, the corresponding packages p_i and p_j are created such that they *do not* share any common satellite item on any type. For any non-edge pair of nodes (n_i, n_j), the packages p_i and p_j are created such that they share the same satellite item on at least one type. Figure 4.6 illustrates an example transformation from a graph to a set of packages. Thus, an ordering of the packages with an all-zero penalty vector exists if and only if a Hamiltonian path exists for G.

It can also be shown that the number of satellite types required for this transformation is bounded by the number of packages: $(n-1)$ satellite types are needed only when G contains a single node that is not connected to any other node in G and the rest of G is fully connected. The time complexity of the transformation is $O(n^3)$: we update a package p_i at most $(i-1)^2$ times. $\square$

4.5.2 Heuristic Visual Effect Optimization

In this section, we introduce a heuristic algorithm (Algorithm 11) for solving the visual effect optimization problem. The basic idea is to always select the next package from among the candidate packages that are optimized for the first satellite type (i.e., the one with the highest priority) and select the package in a greedy fashion by choosing the one that incurs the minimum penalty with the previously chosen package. Interestingly, we show later that, despite being heuristic in the general case, this algorithm is optimal when there is exactly one satellite type.

Algorithm 11: `EnhanceVE`$(\mathcal{P}, \mathcal{O})$: Heuristic algorithm for enhancing visual effect

```
Require:
    $\mathcal{P} = \{p_1, p_2, ...\}$: the set of satellite packages
    $\mathcal{O} = \mathcal{S}_1 \prec \mathcal{S}_2 \prec ... \prec \mathcal{S}_m$: the prioritization of $m$ satellite types
$PO = \{\}$ will maintain the ordered list of packages to be output
let $D_{\mathcal{S}_1}(\mathcal{P})$ be the set of distinct satellite items for type $\mathcal{S}_1$ within all the packages in $\mathcal{P}$;
let $D_{s_1^x}(\mathcal{P})$ be the set of packages ($\in \mathcal{P}$) with item $s_1^x \in D_{\mathcal{S}_1}$ for type $\mathcal{S}_1$;
while $|D_{\mathcal{S}_1}(\mathcal{P})| > 1$ do
    let $p_o$ be the last chosen package
    let $s_1^x$ be the satellite item for type $\mathcal{S}_1$ in $p_o$
    let $D_{s_1^y}(\mathcal{P})$ be the largest set of packages among all sets $D_{s_1^i}(\mathcal{P}), s_1^i \in D_{\mathcal{S}_1}$
    let $D_{s_1^z}(\mathcal{P})$ be the second largest such set
    if $s_1^y == s_1^x$ then
        $D = D_{s_1^z}(\mathcal{P})$
    else
        $D = D_{s_1^y}(\mathcal{P})$
    $p = $ PickBestCandidate$(p_o, D, \mathcal{O})$
    add $p$ to $PO$
    remove $p$ from $\mathcal{P}$
while $|\mathcal{P}| > 0$ do
    let $p_o$ be the last chosen package
    $p = $ PickBestCandidate$(p_o, D, \mathcal{O})$
    add $p$ to $PO$
    remove $p$ from $\mathcal{P}$
return $PO$
```

Function `PickBestCandidate`$(p_o, D, \mathcal{O})$: Subroutine for choosing the next best package

```
Require:
    $p_o$: the previously chosen package
    $D$: the set of candidate packages
    $\mathcal{O} = \mathcal{S}_1 \prec \mathcal{S}_2 \prec ... \prec \mathcal{S}_m$: the ordered list of $m$ satellite types
for $i = 2$ to $m$ do
    $C = \{\}$ will maintain the just eliminated candidate packages
    for $p_j \in D$ do
        if $p_o$ and $p_j$ share the same item for type $\mathcal{S}_i$ then
            add $p_j$ to $C$
            remove $p_j$ from $D$
    if $|D| == 1$ then
        return $p \in D$
    if $|D| == 0$ then
        return random $p \in C$
if $|D| > 1$ then
    return random $p \in D$
```

Figure 4.7. **Function** `PickBestCandidate`.

Intuitively, the algorithm starts by grouping all packages according to their satellite items of type $\mathcal{S}_1$. In choosing the next package, the algorithm always selects from the largest group for the remaining packages, unless the last package is also selected from that group, in which case the algorithm selects from the second largest group. Picking the exact package from within the group is accomplished by removing packages that share the same satellite item with the previously chosen package for each subsequent satellite type, until one package remains. We illustrate the algorithm with the simple example in Table 4.4:

Example 11. *Given the following four packages:*

$$p_1 = (s^1_{case}, s^1_{charger}, s^1_{kit}, s^1_{cable}, s^1_{speaker}, s^2_{screen}, s^1_{pen}),$$

$$p_2 = (s^1_{case}, s^1_{charger}, \; - \;, s^3_{cable}, \; - \;, s^1_{screen}, s^1_{pen}),$$

$$p_3 = (s^1_{case}, s^4_{charger}, \; - \;, s^2_{cable}, s^3_{speaker}, \; - \;, s^1_{pen}),$$

$$p_4 = (s^2_{case}, s^4_{charger}, \; - \;, s^2_{cable}, s^3_{speaker}, s^1_{screen}, s^1_{pen}),$$

We first separate them into two groups $G_{s^1_{case}} = \{p_1, p_2, p_3\}$ and $G_{s^2_{case}} = \{p_4\}$. Next, p_1 is randomly chosen from the group $G_{s^1_{case}}$ since it is a larger group. Although $G_{s^2_{case}}$ is still the smaller group, we need to choose a package from it because the last chosen package p_1 is from the larger group. Next, p_4 is chosen from the group $G_{s^2_{case}}$. Then, between the two remaining packages p_2 and p_3, p_3 is eliminated first because it shares item $s^4_{charger}$ with p_4, the last chosen package. The final ordering is therefore (p_1, p_4, p_2, p_3), which happens to be one of the two optimal orderings. Observe that, it is important to deterministically select the next package such that its addition incurs the least penalty with respect to the previously added package. Otherwise, a random selection between p_2 and p_3 in the third step may generate an ordering such as (p_1, p_4, p_3, p_2), which is worse than the ordering that our algorithm produces. In certain settings, where the packages share many common items with each other on lower priority satellite types, such a randomization may exacerbate the result drastically.

The algorithm is not guaranteed to find the optimal ordering. For example, if p_3 is chosen as the first package, the algorithm will fail to find one of the two optimal orderings. However, the time complexity of the algorithm is only $O(mn^2)$, where m is the number of types and n is the number of packages. As we will experimentally demonstrate in Section 4.6, this heuristic algorithm efficiently produces package orderings with close to optimal quality. Further, we prove that when $m = 1$, Algorithm 11 does produce the optimal ordering.

Lemma 4.5.2. *Algorithm 11 produces the ordering of packages with the optimal visual effect if $|\mathcal{O}| = 1$.*

Proof: Given n packages, let G_{big} be the largest group containing a single item with a total of x packages. Let the remaining groups have a total of y packages. Let the optimal ordering have penalty $\langle t \rangle$. If $x <= y + 1$, there will be enough packages that are not in G_{big} to separate packages in G_{big}, therefore $t = 0$. Otherwise, there will be $x - y - 1$ packages in

G_{big} that are followed or preceded by another package in G_{big}, leading to $t = x - y - 1$. The ordering produced by Algorithm 11 has exactly $\langle t \rangle$ as the penalty because each package in G_{big} is followed and/or preceded by a package containing a different item, until there is no more such package left, at which point, all $t + 1$ remaining packages in G_{big} are consecutively placed.□

4.6 Experiments

We conduct a set of comprehensive experiments using a data set obtained from Yahoo! Shopping site to evaluate the quality and performance of our proposed summarization and visual effect optimization algorithms. We assume that the list of central items can be retrieved efficiently (for example, using the TA-family of algorithms [62]) and focus our experiments primarily on efficiently summarizing and presenting satellite packages for a given central item.

Our prototype system is implemented in Java with JDK 5.0. All experiments were conducted on an Intel machine with dual-core 3.2GHz CPUs, 4GB Memory, and 500GB HDD, running Windows XP. The Java Virtual Memory size is set to 512MB. All numbers are obtained as the average of three runs.

4.6.1 Data Preparation

Online shopping is one of the main applications of composite item construction and exploration, so we naturally turn to Yahoo! Shopping that is available to us for data set generation. There are two main pieces of required data: *product listings* and *product compatibilities*. The product (i.e., item) listings are obtained from the site directly, and for each item, we obtain its *id*, *price*, and *type*. The items have wide ranging prices from 1 cent to several thousand dollars. We filter away items with extreme prices (price below $2 or price above $1000) because those are often spam listings. The items are organized into 10 high-level types. We choose one particular type, which contains a much higher concentration of items with prices from \$550 to \$1000, to be the central type, and the other 9 to be the satellite types. In the end, we have $101,271$ items, of which $2,222$ are considered as central items, and the rest are satellite items. On an average, we have $11,005$ items per satellite type.

Obtaining item compatibilities turns out to be a non-trivial task. Our initial thought is to use manufacturers' specifications. However, it is extremely hard to obtain a comprehensive list of compatibilities for such a large number of items. Instead, we turn to the history of transactions from the shopping site. Specifically, we compute the compatibilities between two items based on their pair-wise co-occurrences in various kinds of activities of the same user, such as browsing, rating, and purchasing. The resulting compatibility is a normalized score between 0 and 1, indicating how related two items are based on historical records. A threshold score is then selected to determine whether two items are compatible. In the experiments, tuning this threshold allows us to control how many satellite items are compatible with a central item on average.

The rest of this section is organized as follows. In Section 4.6.2, we demonstrate that our summarization algorithms, `FastGreedySummarySet` and `ProbSummarySet`, clearly outperform baseline algorithms in terms of speed while producing summaries of the same quality. Similarly, in Section 4.6.3, we show that the heuristic `EnhanceVE` algorithm can produce almost the optimal ordering of the summary packages while running much faster than its brute force counterpart.

4.6.2 Summarizing Maximal Packages

In this section, we experimentally evaluate both performance and quality aspects of the `FastGreedySummarySet` and `ProbSummarySet` algorithms in Section 4.4. We compare them against three baseline algorithms:

Table 4.5. # Maximal Packages Generated

#Comp. Size	10	50	100	150	200
#Max. Pckg.	71	320	$2,442$	$6,877$	$17,972$

% Price	5%	10%	15%	20%	25%
#Max. Pckg.	320	$1,060$	$4,375$	$11,470$	$14,805$

`Random`, where a set of k random packages are chosen to be in the summary;

`Deterministic`, where a set of k largest packages are chosen to be in the summary;

`GreedySummarySet` (Algorithm 15), where the coverage of a candidate set of summary packages are computed using the *Inclusion-Exclusion Principle* [59].

We begin by validating that summarization is a necessary technique to help users explore the results because the number of maximal packages is large in many reasonable scenarios.

4.6.2.1 Number of Maximal Packages is Large

Given a central item, the set of maximal packages are generated from individual items, which are compatible with the central item, using `MaxCompositeItemSet` (Algorithm 7). The number of generated maximal packages depends mainly on two factors: *compatibility size*, i.e., how many satellite items are compatible with the central item; and *price budget*, i.e., the total price the user is willing to pay. We vary both factors and examine the number of maximal packages generated.

Specifically, we control the compatibility size by tuning the threshold for the compatibility score, and we vary the price budget for the satellite package by setting it at various percentage levels compared to the price of the central item. A random sample of 100 central items are chosen, and we record the average number of maximal packages being generated for those items. The price budget is fixed at 5% when we vary the number of compatible satellite items, while the number of compatible satellite items is fixed at 50 when we vary the price budget. As shown in Table 4.5, the number of maximal packages grows quickly as the price budget goes up and as the number of compatible satellite items increases. More importantly, even at a modest level of 5% price budget and 50 compatible satellite items, the number of maximal packages reaches into the hundreds, which is clearly beyond what a normal user is willing to explore. This result clear indicates that obtaining a good summary of those maximal packages is a necessary step for exploration by the user.

Finally, we note that the number of maximal packages being generated by the randomized `MaxCompositeItemSet` algorithm is not an underestimate of the actual number that is generated by the Apriori-style optimal algorithms. In those settings where the optimal algorithms are able to finish within a reasonable amount of time (they don't always do), our heuristic algorithm generates exactly the same set of maximal packages (results omitted due to space limitation).

4.6.2.2 Summarization: Performance

Figure 4.8 shows the performance comparison between our two proposed algorithms, `FastGreedySummarySet` and `Prob SummarySet`, against the baseline algorithm, `GreedySummarySet`. For this experiment, we fix the compatibility size at 50 and the price budget at 5% (i.e., on an average 320 maximal packages), and vary the size of the summary (i.e., number of representatives) to be between 5 and 25. Not surprisingly, `FastGreedySummarySet` outperforms the baseline algorithm, especially for larger summaries. More importantly, `ProbSummarySet` significantly outperforms both across all summary sizes. The significant performance advantage of `ProbSummarySet` lies in the fact that it avoids producing the full set of maximal packages, while the other two algorithms have to generate all the maximal packages first. In fact, the process of generating the full set of maximal packages alone is quite time-consuming, as shown in Figure 4.8, where the cost

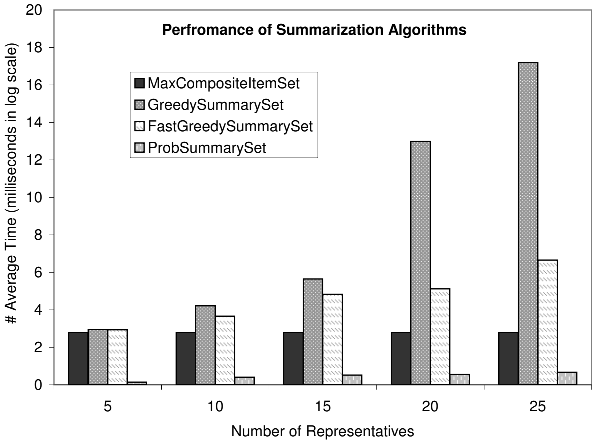


Figure 4.8. Summarization Algorithms Performance.

of `MaxCompositeItemSet` alone is more than the cost of `ProbSummarySet`. We note that the other two baseline algorithms: `Random` and `Deterministic` have essentially the same performance as `MaxCompositeItemSet` since they also require the generation of the full set of maximal packages, but the cost of picking random packages or largest packages are negligible. Finally, we note that only `ProbSummarySet` is able to produce the summary with interactive speed, which is critical in our goal of supporting users' exploration of the results.

4.6.2.3 Summarization: Quality

Having the best performance is of little importance if our algorithms fail to generate summaries of good qualities. We next verify that the summaries generated by our `FastGreedySummarySet` and `ProbSummarySet` are indeed comparable with the baseline algorithm and better than the two simple heuristic algorithms. The experiments are performed with the same settings as in the previous section. As shown in Figure 4.9, `FastGreedySummarySet` achieves exactly the same coverage as the baseline `GreedySummarySet`, which confirms our theoretical analysis in Section 4.4.1 that the for-

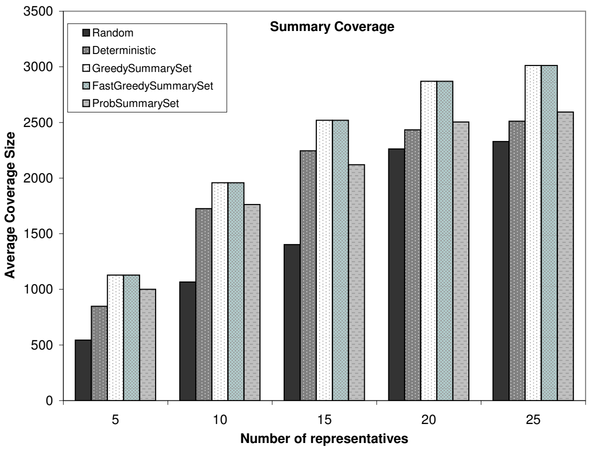


Figure 4.9. Summarization Algorithms Coverage.

mer faithfully mimics the behaviors of the latter, while having a substantially better performance. Furthermore, `ProbSummarySet`'s coverage number is within a reasonable range of the baseline coverage of `GreedySummarySet`, and it is comparable with `Deterministic` and significantly better than `Random`. Given the far superior performance of `ProbSummarySet` against all other algorithms as shown in Figure 4.8, we believe it is the best choice for summarization.

4.6.3 Visual Effect Optimization

In this section, we evaluate the quality and performance of the heuristics visual effect optimization algorithm `EnhanceVE` in Section 4.5. We compare this algorithm against the exponential brute force algorithm `BruteForceVE`, which computes the optimal ordering of packages by going through the types in their priority order and removing candidate orderings, which are no longer the best for the list of examined types so far, until only one ordering is left or all types are examined. We perform the experiments for 100 central items, and for each central item, we generate summaries of varying sizes (i.e., number of representatives), starting with 3, using both algorithms.

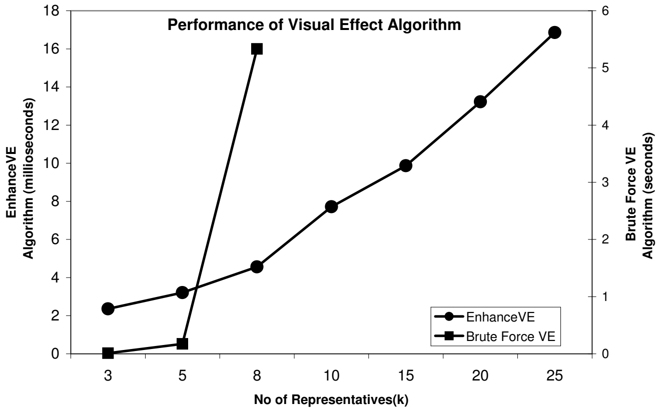


Figure 4.10. Performance of Visual Effect Algorithms.

Performance: Figure 4.10 illustrates that `EnhanceVE` significantly outperforms `BruteForceVE`. Note that the time cost of `BruteForceVE` is shown along the second y-axis on the right and is measured in seconds. As expected, `BruteForceVE` fails to produce an ordering within a reasonable amount of time (10 minutes) as soon as the summary size reaches 10, which is a reasonable number of packages to be shown to the user in practice. Meanwhile, `EnhanceVE` is able to produce an ordering in under 20 milliseconds, fast enough for the system to be interactive with the user.

Quality: Table 4.6 shows the aggregated penalty vectors for different values of k. (Note that `BruteForceVE` fails to produce results after 2 hours of running for summaries with $k > 10$.) The penalty vectors are of size 9 (the number of satellite types in the experiment), where the earlier entries correspond to higher priority satellite types. As the numbers illustrate, the penalty vector of the ordering produced by `EnhanceVE` matches exactly with the optimal penalty vector in higher priority types in all cases, and is only slightly higher in very few positions on the lower priority types. This indicates that `EnhanceVE` indeed produces realistically good solutions at a fraction of the cost of the brute force algorithm.

Table 4.6. Comparison of Penalty Vectors

k	EnhanceVE	BruteForceVE
3	$[0, 0, 0, 0, 0, 0, 2, 1, 0]$	$[0, 0, 0, 0, 0, 0, 2, 1, 0]$
5	$[1, 2, 0, 1, 3, 0, 4, 1, 0]$	$[1, 2, 0, 1, 2, 1, 4, 1, 0]$
8	$[2, 0, 2, 2, 1, 0, 4, 1, 0]$	$[2, 0, 2, 1, 1, 1, 4, 1, 0]$
10	$[2, 1, 2, 3, 1, 3, 5, 1, 0]$	$[2, 1, 2, 2, 1, 2, 5, 1, 0]$
15	$[2, 1, 2, 3, 1, 4, 7, 2, 1]$	N/A
20	$[2, 1, 2, 5, 3, 4, 7, 2, 1]$	N/A
25	$[2, 3, 2, 5, 3, 5, 7, 2, 2]$	N/A

4.7 Related Work

We organize our discussion on related works according to the three main technical problems of our work: maximal package generation, summarizing packages, and visual effect optimization. We also note that, to the best of our knowledge, the work described in this thesis is the first to propose and address the general problem of helping online users construct and explore composite items.

Generating Maximal Packages: Our maximal item set generation algorithm leverages random walk algorithms [57, 63] that are primarily designed for computing maximal frequent itemsets. Several other works have also investigated this problem [55, 64, 56]. Our solution is efficient since it leverages the fact that the budget constraint can be checked purely based on the item itself, and uses the *Good Turing Test* [58] as the stopping criterion.

Summarizing Packages: Our summarization problem can be mapped to an instance of the well-known NP-complete *Max k-Set Cover Problem* [60]. The main difference lies in counting the number of distinct subsets (not distinct items) of representative sets.

Although different from our problem statement, we note that schema summarization techniques based on information theory and statistical models were proposed recently in the context of relational [65] and XML databases [66].

Our proposed modeling of summarization bears resemblance to existing work on ranking skyline points based on dominance [67]. Each representative maximal package can be thought of as a skyline point which covers (dominates) a set of sub-packages. Thus the problem is to select k-representative maximal packages (points) such that the number of packages covered by at least one of them is maximized. However, our problem is more difficult, since we consider this problem in a high dimensional categorical space (as opposed to a low-dimensional numeric space) where the packages covered by a representative maximal package are not present explicitly in the data set.

Visual Effect Optimization: Our visual effect optimization problem definition uses a similar intuition as the diversity problem in [68]. However, while the latter solves the problem of evaluating k diverse query results, we aim at finding an optimal *ordering* of a set of representative packages which maximizes their visual diversity. This calls for a fundamentally different solution. The NP-complete *Hamiltonian Path Problem* [60] can be reduced to an instance of our visual effect optimization problem as discussed in Section 4.5.

4.8 Conclusion

A wide variety of online stores, from e-commerce sites such as Amazon, to online travel reservation sites such as Expedia offer features where a user is suggested a set of additional complementary items along with her main item of interest based on co-purchasing or co-viewing behavior. Broadly motivated by such applications, our approach helps users efficiently and effectively explore a large number of composite items formed by a central item, the item of interest, and compatible satellite packages subject to a budget constraint. To that effect, we propose *summarization* to reduce the large number of satellite packages associated with a central item, and *visual effect optimization* to leverage diversity and help users get a quick overview of available options within their budget. We design and implement efficient algorithms to address the technical challenges involved. Our extensive experiments on data obtained from Yahoo! Shopping site demonstrate the effectiveness and efficiency of our algorithms. As future research directions, we aim to explore more complex modeling of compatibility between satellite items and other variants of visual diversity.

CHAPTER 5

CHAIN COMPOSITE ITEMS

5.1 Introduction

Planning an itinerary is one of the most time-consuming travel preparation activities. For a popular touristic city, it involves painstakingly examining the hundreds of Points-of-Interest (POIs) to select the POIs that one likes, figuring out the order in which they are to be visited, and ensuring the time it takes to visit them, and to transit from one POI to the next, satisfies the user's time budget. Many online services such as Lonely Planet provide packaged itineraries to their users. However, those itineraries suffer from two main drawbacks. First, they are often not tailored to one's own interests. For example, a first-time NYC tourist is likely to be interested in a trip to the Statue of Liberty, while a NYC regular may prefer to check out the latest MoMA exhibit. Second, suggested itineraries may not fit one's particular time budget. Someone who visits a place for a very short time frame, e.g, in the case of a layover in a city, or a very long time frame, e.g., in the case of a month-long backpacking trip, is unlikely to find an itinerary suggested by those services, satisfactory.

Constructing a personalized itinerary for a user is a big challenge because, even with a relatively small number of POIs, the number of possible itineraries can be combinatorially large. In this paper, we adopt an interactive process where the user provides feedback on POIs suggested by our itinerary planning system and the system leverages those feedback to suggest the next batch of POIs, as well as to recommend the best itineraries so far. The process repeats until the user is satisfied. In other words, instead of asking the user to examine all the POIs before deciding on the itinerary, our goal is to ask the user to examine only a subset of those POIs in multiple steps, each with a small number of increasingly relevant POIs, thereby reducing the overall efforts required on the user to construct the itinerary. To the best of our knowledge, this work is the first to address the question of formalizing interactive itinerary planning and explore efficient solutions to this problem.

More specifically, the itinerary planning process involves the following interactions.

1. It starts with a user providing a time budget and a starting point of the itinerary (usually corresponding to the hotel where the user is staying);
2. At each step, the system presents the user with a small fixed number of POIs that are *most probably liked* by the user, based on feedback provided by the user so far;
3. The system also recommends *highly ranked itineraries* to the user based on the feedback;
4. The user provides her *feedback* on suggested POIs to indicate whether or not she is interested in them, and the process continues;
5. The user can also choose to pick one of the recommended itineraries, at which point, the process stops.

Designing such an interactive system is a non-trivial task and raises both semantics and efficiency challenges. We provide a brief overview of those challenges here.

First, we need to define the **POI Feedback Model**, which dictates how the user can specify her preference for the individual POIs. The most generic model is the *star model* where the user provides 5-star ratings for POIs she really wants to visit and 1-star ratings for POIs she does not want to see. Two simpler models are also common: the *ternary model*, where the user specifies 'yes' (i.e., positive), 'do_not_care', and 'no' (i.e., negative) for the POIs, and the *binary model*, where the user is provided with only two feedback options 'yes' and 'do_not_care'. We note that the star model can often be converted into the ternary model. We will discuss the impact of different feedback models on the complexity of itinerary planning, and focus on the binary model within this work.

Second, we need to define the **Itinerary Scoring Semantics**, which dictates how an itinerary should be scored based on the user feedback. Similarly, it can also be defined using multiple semantics. In the *set semantics*, the score of an itinerary positively correlates with the number of POIs with a 'yes' feedback and negatively correlates with the number of POIs with a 'no' feedback. In the strictest interpretation, a single POI with a 'no' feedback can render the entire itinerary ineligible. In the *chain* semantics, the score of an itinerary will further depend on how the positive and negative POIs are arranged in the itinerary. One such semantics could be to rank itineraries containing consecutive POIs marked with a 'yes' higher than ones containing more POIs marked with a 'yes' none of which being consecutive. Finally, an itinerary is only *valid* if it satisfies the budget constraint specified by the user. We focus on time budget in this paper and defer other kinds of budget for future work. We argue that during the interactive itinerary building process, previous user feedback has a direct impact on the score of a new itinerary. For example, when *Times Sq.* has been marked 'yes' by the user in previous steps, the score of an itinerary containing *Times Sq.* and *Madame Tussauds Wax Museum*, should increase, because those two POIs are frequently co-visited. In this work, we use a probabilistic model to compute the *expected score* of valid itineraries given user feedback using the set semantics. We leave the chain semantics to future work.

Third, we need to efficiently solve the **Optimal Itinerary Construction Problem**, i.e., how to construct the best scoring itinerary based on a given set of POIs, along with their feedback, and the user provided time budget. We argue that materialization of all itineraries is not practically feasible and design efficient algorithms for computing itineraries with the best expected scores *on the fly*.

Finally, we need to efficiently solve the **Optimal POI Batch Selection Problem**, i.e., how to select a fixed number of POIs to solicit future user feedback based on the feedback received so far. We argue that the best candidate POIs (to be suggested to the user next) are those which maximize the *expected scores* of the best itineraries. Any user feedback for those POIs is likely to lead to itineraries with high expected scores, and therefore satisfy user's needs sooner. We provide a formal definition of this problem and propose a probabilistic model to compute the expected score of a batch of POIs. There are two main efficiency challenges. First, selecting the optimal batch of k POIs according to the expected itinerary scores requires the system to go through all $m_{\mathcal{C}_k}$ sets of POIs, where m is the number of remaining POIs in the system, which can be large. We design a heuristic algorithm that selects POIs one by one to form partial batches, therefore significantly reducing the candidate POI sets to be examined. Second, the number of remaining POIs to be checked for each partial batch can still be large. In order to reduce that number, we design an efficient pruning strategy which accounts for the distance of the remaining POIs from the starting point and from POIs already in the batch.

Table 5.1 summarizes an example of a 2-step interactive itinerary planning for a user, whose starting location is *Ground Zero, NYC* and has a budget of 6 hours. At each step, the suggested batch of 5-POIs (column-2) is shown, the POIs for which user feedback is $`yes'$ (column-3), and the resulting top-1 itinerary based on her feedback (column-4) are also displayed. Note that, top-1 itinerary of step-2 considers both step-1 and step-2 feedback.

In summary, we make the following contributions.

The paper is organized as follows. Section 5.2 contains a formalization of the interactive itinerary planning approach. Section 5.3 describes the algorithms. Our experiments are reported in Section 6.6. The related work is summarized in Section 5.6. We conclude with future directions in Section 5.7.

5.2 Formalism and Problem Statement

In this section, we discuss the formal data model of interactive itinerary planning. We begin by describing different notations and their corresponding interpretations to be

Table 5.1. 3-step Iterative Itinerary Planning

Step	POI batch	‘yes' **feedback**	**Top-1 Itinerary**
1	*Trinity Church.*; *Brooklyn Bridge*; *NYC Stock Ex*; *Battery Park* ; *Statue of Liberty*	*Trinity Church.*; *NYC Stock Ex*; *Battery Park*	**1.** *Ground Zero - Trinity Church - NYC Stock Ex - Battery Park*
2	*Times Square* ; *Grand Central Terminal* ; *Chrysler Building* ; *UN Head Quarter* ; *Rockefeller Center*	*Times Square* ; *Grand Central Terminal*	**1.** *Ground Zero - Trinity Church - NYC Stock Ex - Battery Park - Times Square - Grand Central Terminal*

used throughout the paper. A summary of those notations is listed in Table 5.2 for easy reference.

Data Model: The underlying data model is a directed *complete graph* $G = (\mathcal{M}, E)$. Each node $m \in \mathcal{M}$ represents a POI and each edge (m_i, m_j) in E represents a transit between the two nodes and is annotated with an edge cost $\texttt{transit}(m_i, m_j)$. The edge cost is not always symmetric. For example, traveling time between two POIs can be different because it is downhill in one direction and uphill in another. Each POI m_i is also annotated with $\texttt{visit}(m_i)$, which represents the cost associated with visiting the POI. For example, it takes about 3 hours to visit the *Statue of Liberty.*

Itinerary: An itinerary is a path in the input graph starting from the start POI. Each itinerary τ has a total visit time $\texttt{totalVisit}(\tau) = \Sigma_{m_i \in \tau}\texttt{visit}(m_i)$, and a total transit time, $\texttt{totalTransit}(\tau) = \Sigma_{(m_i,m_j) \in \tau}\texttt{transit}(m_i, m_j)$. A *valid* itinerary is one such that $\texttt{totalVisit}(\tau) + \texttt{totalTransit}(\tau) \leq \mathcal{B}$, where $\mathcal{B}$ is a user provided budget constraint.

Table 5.2. Notations and Their Corresponding Interpretations

Notation	**Interpretation**
$\mathcal{M}$	set of all POIs in a city
$\mathcal{M}_{seen}$	set of POIs for which feedback has been received
$\mathcal{M}_{remain}$	$= \mathcal{M} - \mathcal{M}_{seen}$
$\texttt{transit}(m_i, m_j)$	transit time from POI m_i to m_j
$\texttt{visit}(m_i)$	time to visit POI m_i
$FeedbackOptions$	set of different feedback values a user can assign a POI (e.g., $\{`yes', `no', `do\_not\_care'\}$)
n	number of feedback options
$\langle id, feedback \rangle$	a POI as an ordered pair of id and feedback option
$\mathcal{I}$	a POI batch
k	number of POIs in a batch
$\mathcal{B}$	total budget
$AllFeedbacks(\mathcal{I})$	$= \{\mathcal{I}_1, \ldots, \mathcal{I}_{n^k}\}$, i.e., set of all possible feedback combinations of $\mathcal{I}$
$\mathcal{I}_j$	$= \{< id_1, feedback_1^j > , \ldots, < id_k, feedback_k^j > \}$, i.e., j-th feedback combination for the POI batch $\mathcal{I}$
τ	an itinerary, expressed as a sequence of POIs
$\tau_{\mathcal{I}_j}$	best itinerary corresponding to j-th feedback combination for the POI batch $\mathcal{I}$
$S_{\mathcal{I}_j}$	score of the best itinerary, given $\mathcal{I}_j$, $\mathcal{B}$ and $\mathcal{M}_{seen}$
$ExpScore(\tau \mid \mathcal{M}_{seen})$	expected score of itinerary, given feedback $\mathcal{M}_{seen}$
$ExpBatchScore(\mathcal{I} \mid \mathcal{M}_{seen})$	expected value (over all $\mathcal{I}_j$) of $S_{\mathcal{I}_j}$

5.2.1 System Overview

The input to the system is the graph G that obeys metric properties, a budget $\mathcal{B}$ (e.g., the user has 8 hours to spend in the city), and a starting POI (e.g., an airport or a hotel). The task of the system is to interact with the user and gather her preferences, and build the best possible itinerary for her via this iterative feedback process. In each iteration, the system suggests a batch of k POIs to the user, and the user provides feedback on these POIs, i.e., her preference for including them in her itinerary. Based on the feedbacks, the valid itineraries are re-ranked according to the *scoring semantics* and the top itineraries are suggested to the user. Since G is complete, therefore the POIs that the user has preferred to have included in the itinerary can always be connected with each other with direct edges based on their shortest transit time paths subject to the budget constraints, and does not need to involve any POI that she has not chosen. At each step, the user is shown the next batch of POI suggestions from the system. This interactive process ends when the user is satisfied with the top itineraries suggested by the system and decides not to proceed with the next batch.

Two computational problems form the heart of the system. The first is the *Optimal POI Batch Selection Problem*, where the system has to determine at every iteration the next batch of k POIs to be shown to the user. Once these POIs have been presented and user feedback collected and updated, the system then has to solve the *Optimal Itinerary Construction Problem*, which re-ranks all itineraries and presents the top-ranked ones to the user. In fact, the *Optimal POI Batch Selection Problem* also requires solutions to multiple instances of the *Optimal Itinerary Construction Problem*, as each candidate set of k POIs have to be considered and top itineraries have to be computed for each possible combination of user feedback. In the rest of this section we develop formal notations and definitions of both problems. We begin by describing the feedback models that we consider.

POI Feedback Model: When one (or more) POIs are shown to the user, the user expresses her preference for them according to a specific feedback model. Let *FeedbackOptions* be the set of different ways in which a user can show her preference to a POI. As an example, for the *ternary feedback model*, $FeedbackOptions = \{`yes', `no', `do\_not\_care'\}$. A simpler model *binary feedback model* has the options $\{`yes', `do\_not\_care'\}$. An alternate binary feedback model may have the options $\{`yes', `no'\}$.

Interestingly, since in this paper we consider recommending itineraries only for a *single user*, the specific feedback model is irrelevant. We only need to be concerned with the POIs marked as $`yes'$ by the user, as the POIs marked as $`no'$ or $`do\_not\_care'$ are never considered by the recommendation algorithm. This is because the underlying graph is a complete graph, and the recommended itinerary should try to visit as many $`yes'$ POIs as the budget allows, and will never need to visit any a POI marked as ‘no’ or ‘do_not_care’. The different feedback models only differ in their “user friendliness”, and do not impact the underlying solution. [1]

In our system, a POI may be regarded as an ordered pair $\langle id, feedback \rangle$, where id identifies the POI (e.g., ‘*Statue of Liberty*). Initially each POI’s feedback is set to the value ‘$unseen$’, and, after the POI is seen by the user, is set to a value from $FeedbackOptions$. At any stage during the interactive process, let $\mathcal{M}_{seen}$ (respectively, $\mathcal{M}_{remain}$) be the set of POIs that have currently been seen (respectively, remain to be seen) by the user; thus initially $\mathcal{M} = \mathcal{M}_{remain}$. At every step of the iteration, the system selects a batch $\mathcal{I}$ of k POIs from $\mathcal{M}_{remain}$ and shows them to the user. The user provides feedback for each POI in $\mathcal{I}$ indicating her preference for including the POIs in the output itinerary. Let

[1] However, if an itinerary has to be shared by a group of users (e.g., a set of people sharing a tour bus), then a POI marked as ‘no’ by some users may be marked as ‘yes’ by other users, and the recommendation algorithm will have to carefully consider the impact of visiting a POI with conflicting preferences by the user group. Recommending itineraries for user groups is left for future work.

$n = |FeedbackOptions|$. We note that there can be n^k feedback combinations, each of which represents a possible user feedback for POIs in $\mathcal{I}$. The following notation will be convenient: $AllFeedbacks(\mathcal{I}) = \{\mathcal{I}_1, \mathcal{I}_2, \ldots, \mathcal{I}_{n^k}\}$, where each $\mathcal{I}_j$ represents a specific combination of feedback by the user for each POI. Thus for the ternary model there are 3^k feedback combinations, whereas the simpler binary model leads to 2^k feedback combinations.

5.2.2 Probability Model

For any candidate set $\mathcal{I}$ of k POIs considered during an iteration, it is crucial that the system be able to derive the probability distribution of these n^k feedback combinations. Such a probability distribution will be useful in steering the system towards choosing the subset $\mathcal{I}$ that maximizes the chances of getting highly ranked itineraries. We adopt probabilistic models that are intended to combine users' general preferences (e.g., statistics derived from past query logs may reveal that most users who wish to visit the *Status of Liberty* would also like to visit the *Empire State Building*) with personalization (e.g., the specific feedback obtained from the user on previous batches of POIs may reveal that this particular user prefers art related places). We describe our models in more details below.

Generic Probability Model: A generic probability model can be used to compute the probability of j-th feedback combination: $Pr(\mathcal{I}_j|\mathcal{M}_{seen})$. This probability model can be learned from two training sources: the past activities (e.g., past itineraries accepted by other users of the system), and current ongoing activities (i.e., the POIs that have been seen and marked by the current user). Several classical machine learning solutions can be used for this purpose, e.g., graphical models such as Bayesian Networks or Markov Random fields [69].

Specific Probability Model: In this paper, however, instead of relying on complex solutions involving a generic probability model, we adopt a much simpler probability model using the assumption of a limited form of *conditional independence*[2]. We assume the POIs in $\mathcal{I}_j$ are not *totally independent* but rather are *conditionally independent*.

Under the conditional independence assumption, we have:

$$Pr(\mathcal{I}_j|\mathcal{M}_{seen}) = \prod_{m_i \in \mathcal{I}_j} Pr(m_i|\mathcal{M}_{seen})$$

Using Bayes' Theorem [70], this can be rewritten as:

$$Pr(\mathcal{I}_j|\mathcal{M}_{seen}) = \prod_{m_i \in \mathcal{I}_j} \frac{Pr(\mathcal{M}_{seen}|m_i) \times Pr(m_i)}{Pr(\mathcal{M}_{seen})}$$

Since $Pr(\mathcal{M}_{seen})$ is a constant for that particular iteration, we therefore have:

$$Pr(\mathcal{I}_j|\mathcal{M}_{seen}) \propto \prod_{m_i \in \mathcal{I}_j} Pr(\mathcal{M}_{seen}|m_i) \times Pr(m_i)$$

Applying conditional independence again:

$$Pr(\mathcal{I}_j|\mathcal{M}_{seen}) \propto \prod_{m_i \in \mathcal{I}_j} \prod_{m_l \in \mathcal{M}_{seen}} Pr(m_l|m_i) \times \; Pr(m_i)$$

Even though the probability formula is a proportionality formula, it suffices for our purpose as it is used in the scoring function for ranking itineraries, since all we need to know is whether one itinerary has a higher score than the other—the exact score is irrelevant. Computing the probability formula requires us to know the value of quantities such as $Pr(m_l|m_i)$ and $Pr(m_i)$ where m_i and m_l are POIs. However, singleton and pairwise probabilities can be computed in a preprocessing step from itineraries chosen by previous users. For example, $Pr(m_l|m_i)$ can be estimated as the fraction of previous itineraries containing m_i that also contain m_l, and $Pr(m_i)$ can be estimated as the fraction of itineraries that contain m_i.

5.2.3 Itinerary Scoring Semantics

An itinerary consists of two sets of POIs: the seen POIs for which user feedback has already been collected, and the remaining POIs for which we can only estimate the

[2] Conditional independence assumption is used in building Naive Bayes classifiers [70]

user feedback. Thus the score of an itinerary should be a combination of the score of the seen part, as well as the expected score of the remaining part, where the expectation is over the probability distribution of all possible user feedback. The probability model proposed earlier can be used to model the expected score of the unseen part.

Generic Scoring Function: Consider an itinerary τ as $\tau_{seen} \cup \tau_{remain}$. A generic scoring function has the form:

$$
\begin{aligned}
&ExpScore(\tau|\mathcal{M}_{seen}) = \\
&\qquad Combine(Score(\tau_{seen}), ExpScore(\tau_{remain}|\mathcal{M}_{seen}))
\end{aligned}
$$

where the two parts may be combined using any meaningful operation (such as addition, weighted or un-weighted). There can be numerous ways of defining reasonable forms of the function $Score(\tau_{seen})$. For example, a reasonable function is positively correlated with the number of 'yes' POIs, or a sophisticated scoring function may even consider the sequence of the 'yes' POIs in the overall itinerary score.

Specific Scoring Function: While we do not advocate for a specific scoring function in this paper, we illustrate several optimization opportunities in conjunction with a specific scoring function in Section 5.4. This scoring function is related to the binary feedback model, and has a simple but compelling form—the score of an itinerary is the expected number of POIs that will be marked as 'yes' by the user.

5.2.4 Problem Definitions

We are now ready to describe the two fundamental problems that our system needs to solve.

Optimal Itinerary Construction Problem: Given $\mathcal{B}$, $\mathcal{M}_{seen}$, and $\mathcal{I}_j$ (i.e., a specific batch of k POIs with their feedbacks from the user), compute the valid itinerary τ such that $ExpScore(\tau|\mathcal{M}_{seen} \cup \mathcal{I}_j)$ is maximized.

We next introduce some useful notation. Let $\tau_{\mathcal{I}_j}$ be the output of the *Optimal Itinerary Construction Problem*, i.e., the valid itinerary with the maximum expected score, and let its expected score be $S_{\mathcal{I}_j}$. Next, given $\mathcal{B}$, $\mathcal{M}_{seen}$, and a batch of k unseen POIs $\mathcal{I}$ (i.e., without any specific user feedback combination), let $ExpBatchScore(\mathcal{I}|\mathcal{M}_{seen})$ be the expected value (over all possible user feedback combinations $\mathcal{I}_j$) of the random variable $S_{\mathcal{I}_j}$.

Optimal POI Batch Selection Problem: Given $\mathcal{B}$ and $\mathcal{M}_{seen}$, compute the batch of k unseen POIs that maximizes $ExpBatchScore(\mathcal{I}|\mathcal{M}_{seen})$.

Intuitively, we wish to select a batch of k unseen POIs such that, no matter how the user responds with her preferences to these POIs, the expected score of the top ranked itinerary over all possible user feedback is maximized.

As will be discussed in the next sections, the choice of the itinerary scoring function as well as the probability model affects the efficiency of our solutions to these problems. We discuss a general solution framework for these problems in Section 5.3, and more efficient solutions tailored to a specific scoring function and the simpler probability model in Section 5.4. Our solutions are designed to solve one iteration step in the interactive itinerary planning problem.

5.3 General Algorithms for Itinerary Planning

In this section we shall develop the framework of a generic algorithm for solving the *Optimal POI Batch Selection Problem*. We refer to this as a “generic” algorithm because it is essentially a framework that assumes any arbitrary scoring function for itineraries, as well as any arbitrary probabilistic model for predicting user preferences for the remaining unseen POIs, given the current user feedback. We also develop a generic subroutine to solve the *Optimal Itinerary Construction Problem*. We analyze the computational complexity of the problems as well as the proposed algorithms. In the next section, we show how a specific probabilistic model (based on conditional independence), as well as a specific scoring function (based on user feedback restricted to only ‘yes’ and ‘do_not_care’ for POIs), can be leveraged, along with several algorithmic optimizations, to achieve extremely efficient approximate solutions to these problems.

5.3.1 A Generic Optimal POI Batch Selection Algorithm

Our generic algorithm for the *Optimal POI Batch Selection Problem* is shown in Algorithm 12. As can be seen, the main body consists of generating all possible k-sized batches of potential POIs from the remaining unseen POIs, and for each potential batch, computing the expected score of the optimal itinerary—where the expectation is over the probability distribution of all possible user feedback to those k POIs. This calculation is performed by the `ExpBatchScore` subroutine (which will be discussed next). The set of k POIs selected are those that maximize this expected optimal score. We next discuss the `ExpBatchScore` subroutine as described in Algorithm 13, which computes the expected score of the top itinerary given the POI batch ($\mathcal{I}$), conditioned upon the feedback of the seen POIs ($\mathcal{M}_{seen}$). For each of the n^k possible user feedback combinations $\mathcal{I}_j$, we need to recompute the scores of all valid itineraries, and determine the one with the highest score. This is achieved by repeated calls to the `OptItn` subroutine (which will be discussed next). Finally, the expected value of the score of the optimal itinerary is returned

Algorithm 12: Algorithm OptPOIBatchSelection

```
Require: $\mathcal{M}_{seen}$, $\mathcal{M}_{remain}$, batch size $k$, budget $\mathcal{B}$;
  $RS = \{\mathcal{I} \mid \mathcal{I} \subseteq \mathcal{M}_{remain}, |\mathcal{I}| = k\}$;
  $\mathcal{I}_{max} = argmax_{\forall \mathcal{I} \in RS} ExpBatchScore(\mathcal{I}|\mathcal{M}_{seen}, \mathcal{B})$;
  return $\mathcal{I}_{max}$;
```

Algorithm 13: Subroutine ExpBatchScore

```
Require: $\mathcal{M}_{seen}, \mathcal{I} \subseteq \mathcal{M}_{remain}$, budget $\mathcal{B}$;
$AllFeedbacks(\mathcal{I})$ = $\{\mathcal{I}_1, \mathcal{I}_2, \ldots \mathcal{I}_{n^k}\}$;
# each $\mathcal{I}_j$ is a possible feedback combination on $\mathcal{I}$
ExpBatchScore = $Pr(\mathcal{I}_j|\mathcal{M}_{seen})\times$
    $\Sigma_{1\leq j\leq n^k} ExpScore(OptIt(\mathcal{M}_{seen}, \mathcal{I}_j, \mathcal{B})|\mathcal{M}_{seen})$;
return ExpBatchScore;
```

Algorithm 14: Subroutine OptItn

```
Require: $\mathcal{M}_{seen}, \mathcal{I}_j$, budget $\mathcal{B}$;
$\mathcal{T} = \{\tau \mid \texttt{totalVisit}(\tau) + \texttt{totalTransit}(\tau) \leq \mathcal{B}\}$, where $\tau$ is an itinerary
$\tau_{max} = argmax_{\tau\in\mathcal{T}} ExpScore(\tau|\mathcal{M}_{seen} \cup \mathcal{I}_j)$;
return $\tau_{max}$;
```

(where the expectation is computed over the probability distribution of the user feedback $\mathcal{I}_j$).

The `OptItn` subroutine solves the *Optimal Itinerary Construction Problem*. It takes as input the user feedbacks from previous batches ($\mathcal{M}_{seen}$, along with a candidate user feedback combination $\mathcal{I}_j$), and computes the valid itinerary with the highest expected score. As can be seen from Algorithm 14, one straightforward (but inefficient) way of doing this is to first compute all valid itineraries, compute the expected scores of each of them (conditioned by the user feedback in previous batches and candidate user feedback combination), and return the one with the highest score.

In summary, the general algorithms discussed above do appear rather inefficient. However, in what follows, we show that the problems are NP-complete in general, and one may not be able to improve over such naive approaches in the generic case. To improve efficiency, one has to resort to specific scoring functions, approximation heuristics, and other optimizations—such approaches are discussed in Section 5.4.

5.3.2 Complexity Analysis

The generic *Optimal POI Batch Selection* algorithm described above is very inefficient. The inefficiency stems from three sources:

1. There are $\binom{|\mathcal{M}_{remain}|}{k} = O(|\mathcal{M}_{remain}|^k)$ possible batches of k POIs that need to be considered.
2. For a given batch $\mathcal{I}$, all possible n^k user feedback need to be considered.
3. For a given user feedback (i.e., a potential user feedback for a given batch, in conjunction with the user feedback for earlier batches), the itinerary with the highest expected score needs to be computed.

Thus, if we assume that the cost of a single optimal itinerary computation is T, then the total time taken by the `OptPOIBatchSelection` algorithm is $O(|\mathcal{M}_{remain}|^k \times n^k \times T)$. Unfortunately, as the following arguments show, it appears impossible to improve this in general, as even the third task in the list above, i.e., the problem of computing the itinerary with the optimal expected score for a given user feedback (essentially the Optimal Itinerary Construction Problem), is NP-complete.

Theorem 5.3.1. *The* Optimal Itinerary Construction Problem *is NP-complete.*

Proof. (sketch) We can reduce the NP-complete *Rooted Orienteering Problem* [71] to this problem. The rooted orienteering problem is defined as follows: Given a complete weighted graph (in a metric sense, i.e., satisfying the triangle inequality), a start node, and a length budget, determine a path from the start node that visits as many nodes as possible without going over the length budget.

The reduction proceeds as follows. Consider a very simple scenario where:

- the original POIs are connected by a complete weighted graph where each edge weight represents the transit time to go from one vertex to the other along the edge,
- the visit times of all POIs are 0,
- there is no prior probability model: thus all possible user feedback for the next batch are equally likely,
- the user feedback is restricted to 'yes'/'do_not_care' for each POI that is shown to her,
- the score of a valid itinerary is simply the number of POIs that have been marked as 'yes' by the user in her feedback, and
- we are considering the very first batch, i.e. user feedback has not been collected for any POI yet.

Let $\mathcal{I}$ be any subset of k POIs. Let $\mathcal{I}'$ be any subset of $\mathcal{I}$, representing a specific subset of the batch that the user may potentially mark as 'yes'. Consider the induced complete subgraph graph over $\mathcal{I}'$. Let this induced graph be isomorphic to the input graph of the rooted orienteering problem. It is easy to see that computing the valid itinerary with the highest score is equivalent to solving the rooted orienteering problem whose length does not exceed the budget. □

The above theorem shows that computing itinerary with the optimal expected score is NP-complete even for a simple scoring function. Moreover, since the *Optimal POI Batch Selection Problem* is more general than the *Optimal Itinerary Construction Problem*, the former is also easily seen to be NP-complete. Also, as can be seen, the `OptItn` subroutine is called inside the innermost loop of the overall `OptPOIBatchSelection` algorithm, and is therefore called numerous times, making the overall algorithm extremely inefficient. In the next section, we consider several ways to avoid these sources of intractability. In particular, we consider a simple but practical scoring function and a simple probabilistic model for scoring itineraries, and use fast heuristics to compute optimal itineraries approximately.

5.4 Efficient Algorithms for Itinerary Planning

In this section we discuss more efficient solutions to the itinerary planning problems, by focusing on the simple but practice scoring function (discussed in Section 5.2) based on the binary feedback model, and the simple probabilistic model for scoring itineraries based on the assumption of conditional independence. Our solutions are based on fast heuristics to compute optimal itineraries approximately, thus overcoming the intractability of `OptItn`. We also assume that the batch size k is reasonably small (which is true in practice as the value of k is limited by the screen size used to display the selected POIs to the end user), thus making the n^k factor in the running time of `ExpBatchScore` small. We also use a greedy approach to construct the k POIs, thus eliminating having to examine all $|\mathcal{M}_{remain}|^k$ subsets of POIs. Finally we develop several other algorithmic and data structure optimizations to achieve very efficient overall performance of `OptPOIBatchSelection` in practice. In the rest of this section we provide more details of our techniques.

5.4.1 Efficient Approximation Algorithm for POI Batch Selection

One of the main bottleneck in the `OptPOIBatchSelection` algorithm is that a large number of candidate POI batches need to be considered and the best one chosen from among them. Instead, we follow a greedy approach where we construct a POI batch one POI at a time, thus trading off batch quality (i.e., $ExpBatchScore(\mathcal{I}|\mathcal{M}_{seen})$) for efficiency, with the hope that small quality degradation can bring in huge performance improvements.

Consider the algorithm GreedyPOIBatchSelection shown in Algorithm 15. The first step is to prune from consideration those POIs in $\mathcal{M}_{remain}$ that are simply too far away from the start POI to be involved in valid itineraries. For tight budgets, this can be a very effective step in practice. Next, the batch of k POIs are constructed greedily in k iterations. In each iteration i, each of the remaining POIs in $\mathcal{M}_{pruned}$ are considered as candidate for adding to the batch, and the one that creates a batch with i POIs with the maximum batch score is selected for inclusion in the batch.

Thus, unlike the OptPOIBatchSelection algorithm in the previous section which makes

$O(|\mathcal{M}_{remain}|^k)$ calls to subroutine ExpBatchScore (which evaluates each candidate batch), the new GreedyPOIBatchSelection only makes at most $O(|\mathcal{M}_{remain}| \times k)$ calls to subroutine FastExpBatchScore (which itself is a more efficient subroutine than the earlier ExpBatchScore subroutine for evaluating each candidate batch, to be discussed later). Since the value of k is small is practice, the number of calls to FastExpBatchScore is acceptably small.

5.4.2 Efficient Computation of a Batch Score

We next discuss the subroutine FastExpBatchScore which is shown in Algorithm 16. This subroutine takes as input a candidate batch, and evaluates its “expected score”, i.e., for the distribution of all possible user feedback for the candidate batch, the expected score of the optimal itinerary according to the specific scoring function being used. The structure of this subroutine is very similar to that of the corresponding subroutine ExpBatchScore in Section 5.3, because it also enumerates all possible user feedback combinations to the candidate batch, and makes a total of $n^{|\mathcal{I}|}$ calls to another subroutine to determine the optimal itineraries for each possible user feedback combination (this sub-

Algorithm 15: Algorithm GreedyPOIBatchSelection

```
Require: $\mathcal{M}_{seen}$ consisting of '$yes$' and '$do\_not\_care$' feedback, $\mathcal{M}_{remain}$,
    batch size $k$, budget $\mathcal{B}$;
$\mathcal{M}_{pruned} = \{m | m \in \mathcal{M}_{remain}, \texttt{transit}(StartPOI, m) + \texttt{visit}(m) \leq \mathcal{B}\}$; {prune
    $\mathcal{M}_{remain}$ by removing POIs that are very far away from the start POI}
$\mathcal{I}_{max} = \{\}$;
$i = 0$;
{construct POI batch greedily by adding POIs one by one to initially empty batch}
while $i \neq k$ do
    $m = argmax_{m_j \in \mathcal{M}_{pruned}} FastExpBatchScore(\mathcal{I}_{max} \cup \{m_j\} | \mathcal{M}_{seen})$;
    $\mathcal{I}_{max}$ = $\mathcal{I}_{max} \cup \{m\}$;
    $\mathcal{M}_{pruned}$ = $\mathcal{M}_{pruned} - \{m\}$;
    $i$++;
return $\mathcal{I}_{max}$;
```

routine, called `ApproxItn`, will be discussed later). Since we assume that k is small, and $|\mathcal{I}| \leq k$, the total number of user feedback combinations will be reasonably small.

Hamiltonian Paths in Hypercubes: However, there is scope for optimizing `ExpBatchScore` even further. The crucial difference between `FastExpBatchScore` and the earlier generic `ExpBatchScore` is the *order in which* all the user feedback combinations are processed. `ExpBatchScore` processes the user feedback combinations in any arbitrary order, but we observe that certain specific orders can be leveraged to improve overall efficiency. Since we are considering the specific binary feedback model, for a given candidate batch $\mathcal{I}$, there are $2^{|\mathcal{I}|}$ different user feedback combinations. Consider any specific user feedback combination $\mathcal{I}_j$. If we consider $\mathcal{I}$ as an ordered set (in any order) of POIs, then $\mathcal{I}_j$ can be considered as a Boolean vector of length $|\mathcal{I}|$, in which a 1 implies that the corresponding POI has been potentially marked as 'yes', and a 0 implies that the corre-

Algorithm 16: Subroutine FastExpBatchScore

```
Require: $\mathcal{M}_{seen}$ consisting of '$yes$' and '$do\_not\_care$' feedbacks, a set $\mathcal{I}$ of $\leq k$ POIs from $\mathcal{M}_{remain}$;
$AllFeedbacks(\mathcal{I})$ = $\{\mathcal{I}_1, \mathcal{I}_2, \ldots \mathcal{I}_{2^{|\mathcal{I}|}}\}$
  {above sequence should correspond to Hamiltonian path in $\mathcal{I}$-dim hypercube}
FastExpBatchScore = $\Sigma_{1 \leq j \leq 2^{|\mathcal{I}|}} (Pr(\mathcal{I}_j | \mathcal{M}_{seen}) \times ExpScore(ApproxItn(\mathcal{M}_{seen}, \mathcal{I}_j) | \mathcal{M}_{seen}))$; {above calculation should be run in Hamiltonian path sequence to enable incremental computation of ApproxItn and $Pr(\mathcal{I}_j | \mathcal{M}_{seen})$}
return FastExpBatchScore;
```

sponding POI has been marked as a 'do_not_care'. Thus the set of $2^{|\mathcal{I}|}$ user combinations can be viewed as the vertices (i.e., corners) of a $|\mathcal{I}|$-dimensional hypercube.

The subroutine `FastExpBatchScore`'s order for processing all user feedback combinations is as follows: it finds a *Hamiltonian path* in the hypercube, and then processes each user feedback combination in the order in which it appears in this path. For example, consider the 3-dimensional hypercube in Figure 5.1, where a Hamiltonian path is shown traversing the 8 vertices.

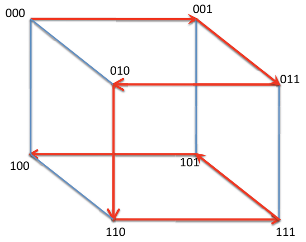


Figure 5.1. Hamiltonian paths in hypercubes..

The reason for using a Hamiltonian path for ordering the user feedback combinations is because the Hamming distance between any pair of consecutive vertices on this path is exactly 1, i.e., the corresponding subsets of 'yes' POIs differ by exactly one POI. This has important efficiency implications. For every user feedback combination $\mathcal{I}_j$, the subroutine `FastExpBatchScore` has to perform two computations (see line 2 in Algorithm 16) (a) it has to call a subroutine `ApproxItn`, and (b) it has to compute $Pr(\mathcal{I}_j|\mathcal{M}_{seen})$, i.e., the probability that the user will give this specific feedback combination, given her earlier feedbacks. We defer the details of `ApproxItn` till later. However, $Pr(\mathcal{I}_{j+1}|\mathcal{M}_{seen})$ can be incrementally computed very efficiently from $Pr(\mathcal{I}_j|\mathcal{M}_{seen})$ if they differ by only one 'yes' POI—as can be seen from the specific probability model formula in Section 5.2, $Pr(\mathcal{I}_{j+1}|\mathcal{M}_{seen})$ can be computed from $Pr(\mathcal{I}_j|\mathcal{M}_{seen})$ in $O(|\mathcal{M}_{seen}|)$ time rather than in $O(|\mathcal{M}_{seen}| \times |\mathcal{I}|)$ time, which would be required if $Pr(\mathcal{I}_{j+1}|\mathcal{M}_{seen})$ had to be computed from scratch.

The following lemma shows that all d-dimensional hypercubes have Hamiltonian paths, and moreover they are easy to construct.

Lemma 5.4.1. *Each d-dimensional hypercube has a Hamiltonian path, and such a path can be computed in $O(2^d)$ time.*

Proof. The proof is by induction. Assume that all hypercubes up to dimension d have Hamiltonian paths. Consider a Hamiltonian path $v_1, v_2, \ldots, v_{2^d-1}, v_{2^d}$. Now, consider a $(d+1)$-dimensional hypercube. It is easy to see that the path $0v_1, 0v_2, \ldots, 0v_{2^d-1}, 0v_{2^d}$, $1v_{2^d}, 1v_{2^d-1}, \ldots, 1v_2, 1v_1$ is a Hamiltonian path in the $(d+1)$-dimensional hypercube. Figure 5.1 illustrates this construction for the case $d = 2$, and $d + 1 = 3$. Clearly, this also implies a simple linear time recursive construction of such Hamiltonian paths. □

What if we did not use a Hamiltonian path ordering? If we use any arbitrary ordering, the changes between successive user feedback combinations may be quite large, thus making probability calculations expensive. For example, suppose we used a random ordering (i.e., a random permutation of all user feedback combinations). Then between successive user feedback combinations in such an ordering, it is easy to see that the expected Hamming distance may be $O(|\mathcal{I}|)$. Thus every time `ApproxItn` is called, the incremental probability computation may take $O(|\mathcal{M}_{seen}| \times |\mathcal{I}|)$ time rather than $O(|\mathcal{M}_{seen}|)$ time if the Hamiltonian path ordering was used.

The Hamiltonian path order is also crucial in the efficient execution of `ApproxItn`, which shall be discussed next.

5.4.3 Approximation Algorithm for Itinerary Construction

The `ApproxItn` subroutine solves the *Itinerary Construction Problem* using approximation heuristics. It takes as input a certain set of user inputs marked as ‘yes’ ($\mathcal{M}_{seen}$, enhanced with a candidate feedback combination $\mathcal{I}_j$), and computes the valid itinerary with the (approximate) highest expected score. Since this problem was shown to be NP-complete in Section 5.3, we use a “Best-Benefit” approximation heuristic to solve this problem approximately.

The subroutine is shown in Algorithm 17, which adopts a greedy approach. Starting from the start POI, at every iteration, the algorithm adds the POI (chosen from the remaining POIs, i.e., those not yet in the partially constructed itinerary) that has the best *benefit*, as defined in line 5. Intuitively, the benefit correlates positively with the probability that the user will mark the POI as ‘yes’, and negatively with the time needed (transit plus visit) to reach this POI from the last POI added to the itinerary.

Heap Data Structures for Maintaining Benefits: For the `ApproxItn` subroutine to be efficient, at every iteration it needs to be able to quickly determine, from the remaining

Algorithm 17: Subroutine ApproxItn

```
Require: $\mathcal{M}_{seen}$, and a candidate user feedback combination $\mathcal{I}_j$
$\mathcal{M}_{tempseen} = \mathcal{M}_{seen} \cup \mathcal{I}_j$
$\tau_{max} = StartPOI$
$RemainB = \mathcal{B} - \texttt{visit}(StartPOI)$
{Construct itinerary greedily using a “best benefit” heuristic}
while $RemainB > 0$ do
    $NextPOI =$
    $argmax_{m_i \in \mathcal{M}_{prune} - \tau_{max}} \frac{Pr(m_i.feedback=yes|\mathcal{M}_{tempseen})}{\texttt{transit}(m_i,\tau_{max}.LastPOI)+\texttt{visit}(m_i)}$;
    $RemainB = RemainB-$
    $\texttt{transit}(NextPOI, \tau_{max}.LastPOI) + \texttt{time}(NextPOI)$;
    if $RemainB > 0$ then
        $\tau_{max} = \tau_{max} \cup \{NextPOI\}$;
return $\tau_{max}$;
```

POIs in $\mathcal{M}$[3] that are not a part of the partially constructed itinerary, the POI with the best benefit with regard to the last POI added to the itinerary. A naive way of doing so is to pre-compute, before each execution of `ApproxItn`, for all pairs of POIs $m_x, m_y \in \mathcal{M}$ the benefit of reaching m_y from m_x. Then, while `ApproxItn` is executing, the benefit of reaching each POI in $\mathcal{M}$ from the last POI of the itinerary can be compared and the POI with the best benefit can be selected. Clearly this approach takes at least $O(|\mathcal{M}|^2)$ time, not accounting for the pre-computation time.

We can reduce the execution time of `ApproxItn` from $O(|\mathcal{M}|^2)$ to $O(|\mathcal{M}| \log(|\mathcal{M}|))$, using the data structuring techniques described below. Since `ApproxItn` is called in the

[3] Actually, this should be $\mathcal{M}_{pruned}$, but in this discussion we assume that in the worst case there may not be any pruning, and $\mathcal{M}_{pruned} = \mathcal{M}$.

innermost loop of our overall itinerary planning algorithms, this can be a substantial savings in practice.

Pre-Computation: Two data structures are prepared before each call to `ApproxItn`:

1. The first is $ProbOrder$, an ordered list of the POIs in $\mathcal{M}$, in decreasing order of $Pr(m_i.feedback = yes|\mathcal{M}_{tempseen})$ for each POI m_i. Note that these quantities are the numerators of the *benefit* of each POI (see line 5 in Algorithm 17). Instead of naively constructing $ProbOrder$ from scratch every time `ApproxItn` is called, we can leverage the fact that the calls are made in sequence along the Hamiltonian path ordering of the user feedback combinations. Thus for each POI m_i, we update $Pr(m_i.feedback = yes|\mathcal{M}_{tempseen})$ from its previous value in constant time, since $\mathcal{M}_{tempseen}$ has changed by only one POI since the last execution. Thus $ProbOrder$ can be updated and re-sorted in overall $O(|\mathcal{M}|\log(|\mathcal{M}|))$ time.
2. The second is a set of *priority queues/heaps* [72] $H_1, H_2, \ldots H_{\mathcal{M}}$, one for each POI in $\mathcal{M}$. For each POI, the corresponding heap contains the time (transit plus visit) needed to reach every other POI in $\mathcal{M}$. Note that these quantities correspond to the denominators of the benefit of each POI (see line 5 in Algorithm 17). These heaps allow the operation *find best time POI* to be performed in constant time, and the operations *delete best time POI* and *insert POI* to be performed in $O(\log(|\mathcal{M}|))$ time. Although it may appear that the total size of all the heaps is $O(|\mathcal{M}|^2)$, these heaps are constructed *only once* by the `FastExpBatchScore` subroutine. During each of the $2^{|\mathcal{I}|}$ executions of `ApproxItn`, these heaps change due to *delete best time POI* operations, but are restored to their original status before the next execution of `ApproxItn` by undoing the delete operations with corresponding *insert POI* operations, as shall be discussed next.

In-Computation: During the execution of `ApproxItn`, the main task at each iteration is to determine, for the last added POI, the POI from the remaining with the best benefit. However, as described above, we do not store the benefits of each POI directly in any data structure (since that will be expensive to maintain), but rather store the numerators and denominators in separate data structures. Thus to find the POI with the best benefit, we have to *simultaneously* scan both data structures in a round-robin manner, $ProbOrder$ as well as $H_{LastPOI}$ (the latter is done by repeated *delete best time POI* operations), until we determine the remaining POI with the best benefit. This is essentially an application of the popular *Threshold Algorithm* (TA) [61]. While in the worst case it can take $O(|\mathcal{M}|)$ if both data structures need to be completely scanned, in practice, it is expected to stop very early. Once the next best POI has been determined, then the heap $H_{LastPOI}$ can be restored by undoing the *delete best time POI* operations with corresponding *insert POI* operations. Thus the in-computation cost of each execution of `ApproxItn` takes $O(|\mathcal{M}|\log(|\mathcal{M}|))$ time, assuming that the TA algorithm only goes to a constant depth on each data structure on average.

In summary, in this section we presented efficient approximation heuristics for the *POI Batch Selection Problem* as well as the *Itinerary Construction Problem*. We leveraged a simple itinerary scoring function based on the binary feedback model, assumed that the batch size k is reasonably small, and applied a greedy strategy for constructing the batch of k POIs. This is facilitated by making calls to an approximation algorithm for itinerary construction that is based on the *best benefit* heuristic. Moreover, we employ interesting algorithmic and data structure optimizations, such as using the heap data structure for indexing the POI benefits, and maintaining the heaps as well as the probability quantities efficiently by following update strategies based on Hamiltonian path ordering in hypercubes.

5.5 Experiments

We conduct a set of comprehensive experiments for popular travel destinations using real world datasets extracted from *Lonely Planet*[4] and *Flickr*[5]. In this section, we describe our experimental set-up, data generation and explain our quality and performance results.

We implemented our prototype using JDK 5.0. All performance experiments were conducted on a 2.66GHz Intel Core i7 processor, 4GB Memory, and 500GB HDD, running OS X. The Java Virtual Memory size is set to 512MB. All numbers are obtained as the average of three runs.

5.5.1 Data Generation

City Names and POI Generation: We consider popular tourist destinations and their POIs for our itinerary planning problem. 12 geographically distributed cities are considered and the popular POIs of those cities are extracted using the *Lonely Planet* dataset. City names, corresponding number of POIs in each city and some example POIs are shown in table 5.3. For each POI, we used Wikipedia[6] to extract latitude and longitude information associated with it.

Transit Time, Visit Time Generation: Given a city, we generate the transit time between every pair of POIs in that city. We use *Google Maps*[7] to calculate the transit time *by car* between a pair of POIs using the underlying road network. This process gives rise to a POI graph, one for each city and each of these generated graphs are complete and directed. Note that, in general, the pairwise transit times generated in this process are asymmetric, which is usually true for any road network. Visit time of each POI is generated using the Flickr log.

[4] http://www.lonelyplanet.com/

[5] http://www.flickr.com/services/api/

[6] http://en.wikipedia.org/

[7] http://maps.google.com/

Atomic and Pair-wise Probability Generation using Flickr Log: We use the publicly available Flickr data[8] to generate atomic and pair-wise probabilities of POIs. Flickr data captures user itineraries in the form of photo streams, where the photos are tagged with corresponding POI names and the respective date/time associated with the photos define the set of possible itineraries (such as, a set of POIs visited on the same day). Given a Flickr log of a particular city, each row in that log corresponds to a user itinerary that is visited in a 12 hour window. We use this log to generate the atomic probabilities of the POIs, and the pair-wise probabilities of every POI pair for a particular city. Using three years worth of Flickr logs, the atomic probability of a POI is the fraction that a POI appears out of the total number of itineraries in the query log. The conditional pair-wise probabilities, $Pr(POI_i|POI_j)$ are calculated as the fraction that POI_i was also visited out of the total number of times POI_j was visited.

The Flickr log may be considered as a collection of itineraries selected by prior users. This enables us to perform quality experiments evaluating the effectiveness of interactive itinerary planning, without requiring a user study involving actual users. Our interactive approach chooses the next batch of POIs suggestions based on the probabilistic model learned from Flickr itineraries. User response is also simulated by the same probabilistic model.

5.5.2 Summary of Experimental Results

We conduct quality and performance experiments by varying the number of POIs, the budget and the size of the suggested POI batch. Each of these parameters impacts the running time and the score of the returned results. We consider a starting POI for each experiment that provides the starting point for the itinerary. All performance experiments are reported for a running time of a single batch. We argue that pre-computation of itineraries

[8] http://www.flickr.com/services/api/

Table 5.3. Example Cities and POIs

City Name	Number of POIs	Example POIs
Amsterdam	118	Diamond Museum, Museum Amstelkring, Oosterpark
Bangkok	48	Phayathai Palace, Siam Ocean World, Wat Traimit
Barcelona	73	Arc de Triomf, Museu Picasso, Plaza Reial
Chicago	91	Flat Iron Building, Lincoln Park, Soldier Field
London	163	Brick Lane, Buckingham Palace, Hyde Park
New Orleans	35	French Quarter, Pitot House, St Roch Cemetery
New York	119	Chelsea Art Museum, Lincoln Center, Russian & Turkish Baths
Paris	114	Bois de Vincennes, Jardin des Tuileries, Petit Palais
Rome	134	Arco di Costantino, Colosseum, Gianicolo
San Francisco	78	Alcatraz, Mission Dolores Park, Union Square
Sydney	96	Bondi Beach, Customs House, Taronga Zoo
Toronto	48	Cn Tower, Ontario Place, Spadina Museum

is not possible. We observe in our dataset, that, for a budget of 6 hours, any set of 5 POIs are permissible and can form a valid itinerary. Given a city that consists of about 150 POIs, roughly the number of valid itineraries that consist of all 5 POIs could be in the range of 0.5 billion (the total number of itineraries would be much more), which certainly is not feasible to pre-compute.

In short, our experimental results substantiate our claim that the greedy algorithm for interactive itinerary planning is a feasible solution for interactive itinerary planning, both quality and performance wise. In addition, we propose several optimizations of the greedy algorithm and our results accordingly corroborate our theoretical analysis, by generating faster running times for the optimized variants.

5.5.3 Quality Experiments

In this subsection, we discuss and report the results of our quality experiments.

Greedy Interactive Itinerary Planning Algorithm: In this experiment, we vary the budget and observe the expected score of the optimal itinerary in one step of the interactive itinerary planning process. We compare the optimal itinerary scores produced by `OptPOIBatchSelection` and `GreedyPOIBatchSelection`. Both of these algorithms use the greedy *best benefit* heuristic to obtain the best itinerary. Input to these algorithms is a set of user feedbacks ($`yes'$ to 3 different POIs) and a batch size (3). This experiment is run on New York City, which has 119 POIs.

Figure 5.2 shows the output of this experiment. We note that with increased budget, since more POIs can be added to the optimal itinerary, its expected score increases. The figure corroborates the fact that `GreedyPOIBatchSelection` is comparable in the quality of its optimal itinerary, to the more expensive `OptPOIBatchSelection`.

Effectiveness of Interactive Itinerary Planning: In this experiment, we select prior Flickr-based 25 static itineraries (we refer to this as `OfflineItinerary`) instead of

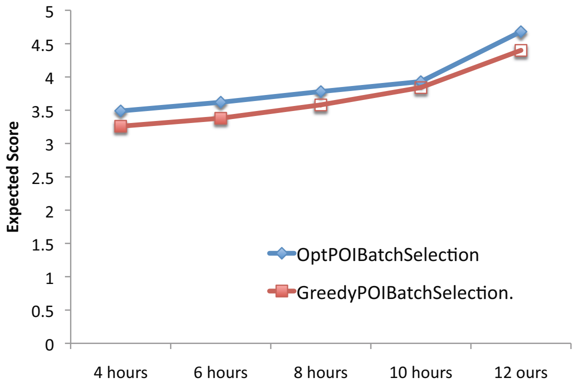


Figure 5.2. Expected Score Comparison.

actual users, where each itinerary consists of 10 POIs, and is visited in 12 hours. We consider a simpler scoring function to assign score in each of them - the score of an itinerary is the number of POIs in it. For each static itinerary, we apply our interactive itinerary planning algorithm (known as `InteractiveItnPlanning`), where the next batch of POIs suggestion is based on the probabilistic model learned from those Flickr itineraries. `InteractiveItnPlanning` calls `GreedyPOIBatchSelection` to select a POI batch at each iteration. In each batch, user response is akin to the actual POIs present in that itinerary,i.e., response is 'yes' for those POIs which actually surface in that static itinerary.

Figure 5.3 records the average itinerary score in each batch. It shows that `InteractiveItnPlanning` reaches the same score of offline itineraries in 4 batches on an average. Thus this result demonstrates that our interactive approach effectively generates itineraries that are liked by prior Flickr users .

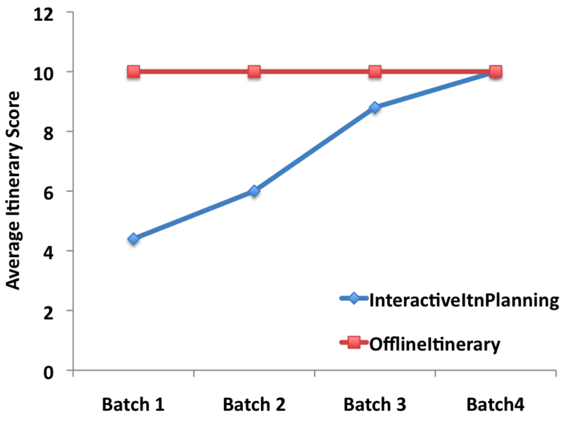


Figure 5.3. Effectiveness of Itinerary Planning Algorithm.

5.5.4 Performance Experiments

In this subsection, we discuss the efficiency aspects of the interactive itinerary planning algorithms, describe the running time attained by performing proposed optimizations and compare that with the optimal brute-force algorithm. Performance is recorded by mainly varying 3 parameters - budget, batch size and number of POIs.

Feasibility of the Optimal Algorithm: We record the running time of `OptPOIBatchSelection` and `GreedyPOIBatchSelection`, for varying batch sizes k in Figure 5.4. The number of POIs is set to 119 for this experiment, whereas the budget is fixed at 6 hours. `OptPOIBatchSelection` algorithm runs in seconds, whereas `GreedyPOIBatchSelection` runs in milliseconds. Also, beyond batch-size 4, `OptPOIBatchSelection` does not terminate within 10 hours, whereas `GreedyPOIBatchSelection` scales well with increasing batch size. This observation confirms that

`GreedyPOIBatchSelection` is an efficient solution for interactive itinerary planning.

Varying Batch Size: In this set of experiments, we vary the batch size k and profile the running time of the different optimizations performed in combination with `GreedyPOIBatchSelection`. This algorithm is compared with its optimized variants

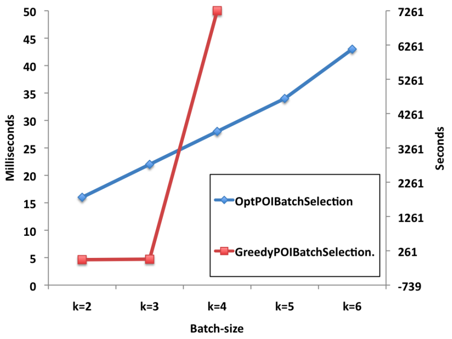


Figure 5.4. Running time Comparison.

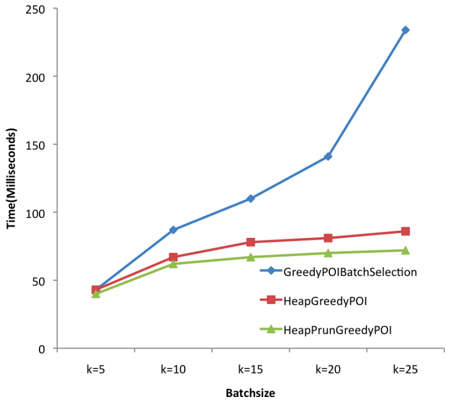


Figure 5.5. Running time Varying Batch Size.

- greedy that uses a heap to calculate *best time POI* and processes user feedback combinations in the heap following the Hamiltonian path computation(`HeapGreedyPOI`), and the the most optimized variant, (`HeapPrunGreedyPOI`), which in addition to efficient heap processing, also prunes the set of remaining POIs, subject to the budget.

The number of POIs is set to 119 for this experiment, while the budget is fixed at 6 hours. Figure 5.5 records the running time of this experiment. We observe that with an increasing batch size, the most optimized variant `HeapPrunGreedyPOI` performs substantially better than `GreedyPOIBatchSelection`. This confirms that our proposed optimizations are important to improve the overall performance.

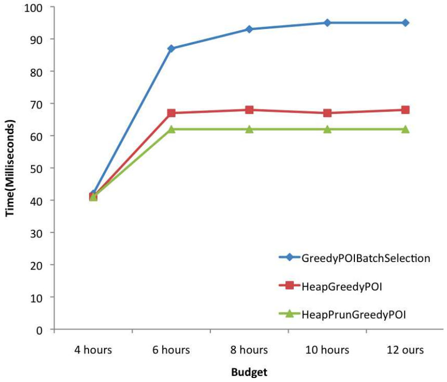


Figure 5.6. Running Time Varying Budget.

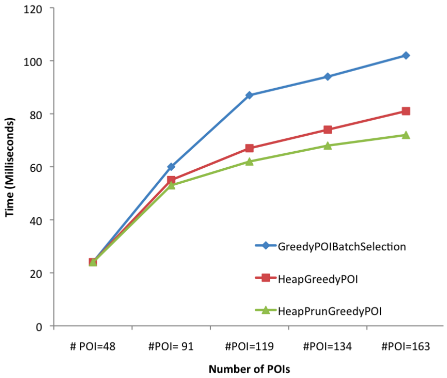


Figure 5.7. Running time Varying Number of POIs.

Varying Budget: We vary the budget constraints and keep the batch size and the number of POIs (10 and 119 respectively) fixed, and record the running time of different variants of the greedy algorithm in Figure 5.6. The figure shows that with the increasing budget, the running time increases in general for all variants. The most optimized variant `HeapPrunGreedyPOI` outperforms others in all cases. One interesting observation here is, the running time does not increase linearly with the budget. This phenomenon is due to the fact, that, for a large enough budget (while everything else is fixed), there cannot be any pruning based on budget and hence it does not impact performance anymore.

Varying Number of POIs: We vary the number of POIs for a fixed budget (6 hours) and batch size (10). The running time of different variants are recorded in Figure 5.7. With the increase in POIs, the running time increases in general. This can be explained since the greedy algorithm has to probe more POIs for selecting the k best POIs in each batch. The most optimized variant `HeapPrunGreedyPOI` outperforms the rest in all cases. One noteworthy observation here is, the running time of `HeapPrunGreedyPOI` increases the least with the increase in the number of POIs. The role of pruning becomes significant in this case, hence with the increase in the number of POIs, `HeapPrunGreedyPOI` effectively prunes the remaining POIs in a batch, and becomes the winner.

5.6 Related Work

Our work of interactive itinerary planning is an effort towards returning complex objects (i.e., an itinerary constructed of several POIs) to the user based on user interaction, subject to the constraints. In a recent work, we first propose the notion of composite items [73] towards that goal. However, an itinerary is *not any arbitrary* ordering of a set of POIs, but it renders a strict ordering between the POIs, subject to the constraints. Consequently this problem is significantly different from our earlier model [73].

The interactive itinerary planning facilitates effective navigation through the information space. Our interactive POI selection strategy is akin to exploratory browsing interfaces such as faceted search [74]. However, the interaction here is on the suggested set of POIs.

Existing work related to travel itineraries can be classified into touristic data analysis and touristic information synthesis. Regarding the former, there are a number of studies on analyzing POI visitation patterns from geo-spatial and temporal evidences left by travelers [75, 76, 77, 78, 79]. However, those works generally do not synthesize POIs into itineraries and instead focus solely on the analysis itself. In the context of touristic information synthesis, a number of works construct and recommend tourist itineraries at various granularities [80, 81, 82, 83] but none of them provides the ability to query constructed itineraries. Our work is tangentially related to other vast fields such as visualizing geo-spatial data, tracking movements based on sensor networks, and constraint optimization. The closest works to ours are [84] and [85] which merge touristic data analysis and synthesis to recommend itineraries based on user's input. However, none of them does so in an interactive manner.

A recent work proposes interactive route search in the presence of order constraints [86]. The proposed approach is different from our work in that it does not consider user budget, does not synthesize user's previous feedback to learn future probability of user preferences, and more importantly, tries to build an itinerary POI by POI, whereas we consider a navigational approach that starts with all possible valid itineraries, which is then iteratively narrowed by suggesting POIs in batches based on highest expected itinerary scores.

Our *optimal itinerary construction problem* is akin to the *vehicle routing problem* and *traveling salesman problem* [72, 87], widely studied in the field of Computer Science and Operation Research. These problems and several of their variants are known to be NP-complete. One variant of vehicle routing problem is the *Orienteering problem*, which and many of its variants are also known to be NP-complete [88, 89, 90]. In particular, we deal with the Rooted Orienteering problem in non-Euclidean asymmetric metric space. Efficient polynomial time approximation scheme is known for this problem problem in the plane [71]. Unfortunately, to the best of our knowledge, there are no known approximation algorithms with provable bounded factors for its non-Euclidean asymmetric variant.

Our greedy solution to the itinerary construction problem requires efficient searching for the next *best time* POI. We resort to a heap data structure [72] that facilitates efficient look up operation for the next best time node. The *next benefit* POI is retrieved by performing a threshold style [61] computation on *ProbOrder* lists and heaps.

Our greedy algorithm for POI selection problem processes feedback combinations in a current batch such that the heap requires only one update between subsequent combinations. We leverage on computing a Hamiltonian Path on a hypercube graph [72] to accomplish that task.

5.7 Conclusion

In this paper, we formalized interactive itinerary planning, showed that it is an NP-complete problem and developed intuitive optimizations for the case where the score of an itinerary is proportional to the number of Points Of Interest (POIs) desired by the user. In order to do so, we reduced our problem to the rooted orienteering problem. Our optimizations are based on computing a Hamiltonian path in a hypercube and on using an efficient heap-based data structure to efficiently prune POIs. In the future, we are planning to explore optimizations for more sophisticated itinerary scoring functions such as the chain semantics, and to consider more complex budget constraints which incorporate both time and price. Our algorithms would need to be revisited for that purpose.

CHAPTER 6

GROUP RECOMMENDATION: SEMANTICS AND EFFICIENCY

6.1 Introduction

Recommender systems have grown to become very effective in suggesting items of high relevance to individual users. Group recommendation, or the task of finding items that please a set of users, on the other hand, started to received attention relatively recently [91, 92, 93, 94, 95, 96, 97, 10]. We envision a system that a community of users can consult when planning an activity together such as looking for a book for a reading club, finding a restaurant to celebrate a project milestone with colleagues, or renting a movie to watch at a girls' night out. In this paper, we study this problem with a focus on time and space efficiency.

Even more so than in traditional individual recommendation, identifying items of high relevance to a group is challenging: What if group members disagree on their favorite items (e.g., people who prefer non-fiction books vs those who like fiction, in a book reading club)? What if there is a group member whose tastes highly differ from all others (e.g., a vegetarian going to a restaurant with non-vegetarians)? At its core, group recommendation necessitates the modeling of disagreements between group members and aims to find items with high predicted rating that also minimize disagreements between group members. In other terms, it is more desirable to return an item that each group member is happy with than to return an item that polarizes group members even if the latter has higher average ratings among them. In this work, we formalize the notion of *consensus functions* that capture such real-world scenarios.

Intuitively, the general form of consensus functions is a weighted combination of predicted rating and pair-wise disagreement. For a given user, her individual preference (i.e., predicted rating generated by an underlying recommender system) for items can be maintained in the so-called *predicted rating list*. We can then leverage Fagin-style merge algorithms [61] to generate items to be recommended to the group based on individual lists of items sorted by their predicted ratings to each group member. Unfortunately, while item disagreements between users can be computed from their predicted rating lists, they do not increase or decrease monotonically with the predicted ratings: two users who both think highly of an item may still disagree more on that item than on an item they both dislike. This drastically reduces the pruning power of the merge algorithms. To address this issue, we introduce *pair-wise disagreement lists* which are pre-computed from predicted rating lists and sorted in decreasing order of disagreements. Both predicted rating lists and pair-wise disagreement lists can then be merged, using Fagin-style algorithms, to find items to recommend to a group.

Without prior knowledge of what groups can be formed between users, a disagreement list has to be created for every user pair. In practice, this introduces enormous space requirements. A back-of-the-envelope computation shows that with a modest 70K-user, 10K-item database, a total of about 2TB space is needed to store the 14 trillion list entries in pair-wise disagreement lists. To address this concern, we develop space reduction strategies which exploit two key characteristics of disagreement lists. First, entries in those lists may be redundant due to shared user behavior. Our strategy which *factors out common entries* in disagreement lists without affecting I/O. Second, all lists do not contribute equally to processing time because of different rating distributions. We develop a *partial materialization* strategy which identifies which subset of lists to materialize in order to maximize space reduction and minimize processing time.

Intuitively, if two users (u and v) agree on many items, their disagreement lists with all other users will be the same for those items. In other terms, given any other user w, the entries corresponding to the items that u and v agree on in the (u, w) disagreement list will be the same as those in the (v, w) disagreement list. Hence, they can be stored only once, instead of being replicated in all lists. We call this *behavior factoring* in disagreement lists. We formalize the problem and devise an algorithm for efficiently factoring common entries in disagreement lists. This space saving strategy requires changes to the group recommendation algorithm to process factored lists.

Factoring comes for free and always saves space when at least two users agree on some items. Unfortunately, if a space budget is imposed, factoring alone does not always guarantee to produce a set of lists within that budget. We further explore *partial materialization* as a complementary space reduction strategy which selectively materializes a subset of the disagreement lists. In a nutshell, a disagreement list which does not significantly affect processing time and consumes too much space, should be dropped. Not surprisingly, partial materialization may negatively affect processing time since the benefit of non-materialized disagreement lists will be lost. We formulate partial materialization as a variant of the Knapsack problem and develop an algorithm which identifies the subset of lists to materialize.

6.1.1 Contributions and Outline

We make the following contributions in this thesis:

1. We formalize the problem of *top-k group recommendation* and introduce the notion of group consensus function that incorporates various predicted rating and disagreement models.

2. We propose the use of pair-wise disagreement lists, and design and implement efficient group recommendation algorithms based on the merging and effective pruning of individual predicted rating lists and pair-wise disagreement lists.
3. Given the potentially large number of disagreement lists, we exploit shared user behavior to reduce the space requirement of those lists. As a result, we extend the group recommendation algorithms to process factored lists. We show that factoring common entries in disagreement lists can drastically reduce storage space without incurring I/O overhead.
4. The factoring strategy does not always guarantee reaching a fixed space budget. To achieve a certain space budget, we develop a partial materialization strategy which exploits the size of each disagreement list and their impact on query processing: it skips disagreement lists in order to minimize space while incurring small processing time overhead. We formalize this question as an adaptation of the Knapsack problem and develop an algorithm to solve it.
5. We run an extensive set of experiments with different group sizes on MovieLens data sets. We perform extensive user-study in Amazon's Mechanical Turk to demonstrate the effectiveness of our group recommendation semantics and how satisfied users are with recommended group ratings compared to individual ones. Our elaborate performance experiments exhibit the efficiency of group recommendation computation. We also demonstrate the benefit of behavior factoring and partial materialization on space.

We note here that the group recommendation problem definition and the basic group recommendation algorithms were first introduced in the conference version [10] of this thesis. Furthermore, [10] also discussed partial materialization to a certain extent. However, the partial materialization algorithms described here address the problem in a more formal way. Behavioral factoring is introduced for the first time here. The rest of the thesis is organized as follows. Section 6.2 provides some background and formalism. It describes the family of consensus functions we tackle in this thesis and defines the group recommendation problem. Section 6.3 describes the group recommendation algorithm. Section 6.4 presents our behavior factoring strategy and a revision of the group recommendation algorithm to operate on factored lists. Section 6.5 discusses partial materialization in the presence of a space budget and develops our adaptation of the Knapsack problem to achieve partial materialization post factoring. Experiments are presented in Section 6.6. Section 6.7 contains the related work. We conclude in Section 6.8.

6.2 Background and Data Model

Let $\mathcal{U}$ denote the set of users and $\mathcal{I}$ denote the set of items (e.g., movies, travel destinations, restaurants) in the system. Each user u may have provided a rating for an item i in the range of 0 to 5, which is denoted as $\texttt{rating}(u, i)$. If the user has not provided a rating for an item, the `rating` is set to $\perp$. We further generate *predicted ratings* for each pair of user and item, denoted as $\texttt{predictedrating}(u, i)$. This predicted rating comes from two sources. If the user has provided a rating for the item, then it is simply the user provided rating. Otherwise, it is generated by the system using a recommendation strategy as outlined next.

6.2.1 Individual Recommendation Model

We review the two most popular families of recommendation strategies. These strategies rating on finding items similar to the user's previously highly rated items (item-based), or on finding items liked by people who share the user's interests (collaborative filtering) [98]. In both cases, missing ratings are assigned value 0.

6.2.1.1 Item-Based Strategies

These are the oldest recommendation strategies. They aim to recommend items similar to those the user preferred in the past. While different strategies use different approaches to compute the predicted rating, we present one common formulation. The rating of an item $i \in \mathcal{I}$ by a current user $u \in \mathcal{U}$ is estimated as follows:

$$\texttt{predictedrating}(u, i) = Avg_{i' \in \mathcal{I} \text{ \& } \texttt{rating}(u,i') \neq \perp} \texttt{ItemSim}(i, i') \times \texttt{rating}(u, i').$$

Here, $\texttt{ItemSim}(i, i')$ returns a measure of similarity between two items i and i'. Item-based strategies are very effective when the given user has a long history of rating activity. However, item-based strategy do not work well when a user first joins the system. To address that, collaborative filtering strategies have been proposed, which we briefly describe next.

6.2.1.2 Collaborative Filtering Strategies

These strategies aim to recommend items which are highly rated by users who share similar interests with or have declared relationship with the given user. The key of this method is to find other users connected to the given user. The rating of an item i by a user u is estimated as follows:

$$\texttt{predictedrating}(u, i) = Avg_{u' \in \mathcal{U} \text{ \& } \texttt{rating}(u',i) \neq \perp} \texttt{UserSim}(u, u') \times \texttt{rating}(u', i)$$

Here, $\texttt{UserSim}(u, u')$ returns a measure of similarity or connectivity between two users u and u' (it is 0 if u and u' are not connected). Collaborative filtering strategies broaden the scope of items being recommended to the user and have become increasingly popular.

We note that there are also so-called fusion strategies [99], which combine ideas from item-based and collaborative filtering strategies, and model based strategies, which leverage machine learning techniques. While we do not consider them in this thesis, we note that group recommendation does not consider rating on one specific strategy to generate recommendations for individual group members.

6.2.2 Group Recommendation Model

The goal of group recommendation is to compute a recommendation score for each item to reflect the interests and preferences of all the group members. In general, group members may not always have the same tastes and a *consensus score* for each item needs to be carefully designed. Intuitively, there are two main aspects to the consensus score. First, the score needs to *reflect the degree to which the item is preferred by the members*. The more group members prefer an item, the higher its score should be for the group. Second, the score needs to *reflect the level at which members disagree or agree with each other*. All other conditions being equal, an item that members agree most about should have a higher score than an item with a lower overall group agreement. We call the first aspect *group predicted rating* and the second aspect *group disagreement*.

Definition 16 (Group Predicted Rating)**.** *The predicted rating of an item i by a group $\mathcal{G}$, denoted as* $\texttt{rating}(\mathcal{G}, \texttt{i})$*, is an aggregation over the predicated rating of each group member,* $\texttt{predictedrating}\ (u, i)$ *where $u \in \mathcal{G}$. We consider two main aggregation strategies:*

1) Average:

$$\texttt{rating}(\mathcal{G}, i) = \frac{1}{|\mathcal{G}|} \sum (\texttt{predictedrating}(u, i))$$

2) Least-Misery:

$$\texttt{rating}(\mathcal{G}, i) = Min(\texttt{predictedrating}(u, i))$$

Average and Least-Misery aggregation models are considered because they are the most prevalent mechanisms being employed currently [91]. Alternative aggregations (e.g. Most-Happiness, i.e., taking the maximum over all individual predicted ratings) are also possible.

Definition 17 (Group Disagreement)**.** *The disagreement of a group $\mathcal{G}$ over an item i, denoted* $\texttt{dis}(\mathcal{G}, i)$*, reflects the degree of consensus in the predicted ratings for i among group members. We consider the following two main disagreement computation methods:*

1) Average Pair-wise Disagreements:

$\texttt{dis}(\mathcal{G}, i) = \frac{2}{|\mathcal{G}|(|\mathcal{G}|-1)} \sum (|\texttt{predictedrating}(u, i) - \texttt{predictedrating}(v, i)|)$*, where $u \neq v$ and $u, v \in \mathcal{G}$;*

2) Disagreement Variance:

$\texttt{dis}(\mathcal{G}, i) = \frac{1}{|\mathcal{G}|} \sum_{u \in \mathcal{G}} (\texttt{predictedrating}(u, i) - \texttt{mean})^2$*, where* $\texttt{mean}$ *is the mean of all the individual predicted ratings for the item.*

The average pair-wise disagreement function computes the average of pair-wise differences in predicted ratings for the item among group members, while the variance disagreement function computes the mathematical variance of the predicted ratings for the item among group members. Intuitively, the closer the predicted ratings for i between users u and v, the lower their disagreement for i. In Section 6.3.1, we will characterize the properties of both disagreement functions in detail.

Finally, we combine group predicted rating and group disagreement in the *consensus function*.

Definition 18 (Consensus Function)**.** *The consensus function, denoted* $\texttt{F}(\mathcal{G}, i)$*, combines the group predicted rating and the group disagreement of i for $\mathcal{G}$ into a single group recommendation score using the following formula:*

$\texttt{F}(\mathcal{G}, i) = w_1 \times \texttt{rating}(\mathcal{G}, i) + w_2 \times (1 - \texttt{dis}(\mathcal{G}, i))$*, where $w_1 + w_2 = 1.0$ and each specifies the relative importance of predicted rating and disagreement in the overall recommendation score.*

While one could design more sophisticated consensus functions (see [94] for an example), we adopt this general form of weighted summation of group predicted rating and group disagreement for its simplicity and the fact that the family of threshold algorithms can be easily applied for the computation. We note here that the commonly used least-misery model maps to the case where $w_1 = 1.0$ and group predicted rating is aggregated using the least-misery function.

6.2.3 Problem Statement

PROBLEM (Top-k Group Recommendation). *Given a user group $\mathcal{G}$ and a consensus function $\mathcal{F}$, identify a list $\mathcal{I}_\mathcal{G}$ of items such that:*

1. $|\mathcal{I}_\mathcal{G}| = k$
2. Items in $\mathcal{I}_\mathcal{G}$ are sorted on their decreasing group recommendation score as computed by the consensus function $\mathcal{F}$, and $\nexists j \in \mathcal{I}$ s.t. $\texttt{F}(\mathcal{G}, j) > \texttt{F}(\mathcal{G}, i), j \notin \mathcal{I}_\mathcal{G}, i \in \mathcal{I}_\mathcal{G}$.

6.3 Efficient Computation of Group Recommendation

In this section, we discuss efficient group recommendation algorithms. We first examine the applicability of existing top-k processing algorithms, then present our solution. We then discuss how to improve our algorithm with threshold tightening strategies that benefit from users' predicted rating lists.

6.3.1 Applicability of Top-K Threshold Algorithms

Many of the best algorithms for computing top-k items belong to the family of threshold algorithms [61]. Given an overall scoring function that computes the score of an item by aggregating scores from individual components, threshold algorithms consume sorted item lists that correspond to each component. Those input lists are scanned using sequential or random accesses, and the computation can be terminated earlier using stopping conditions based on score bounds (thresholds). Early stopping is possible when the scoring function is *monotone*, i.e., if component c is the only component in the scoring function and items i_1 and i_2 differ in their scores, the overall score of i_1 is no less than i_2's if i_1's score on c is no less than i_2's score on c.

Recall from Definition 18 that our consensus function is a weight summation of two components, *group predicted rating* and *group disagreement*. It is clear that the consensus function itself is monotone in the two individual components. In other words, if two items have the same group disagreement, the item with the higher group predicted rating will have at least the same group recommendation score, and vice versa.

It is also clear that the two group predicted rating functions proposed in Definition 16 are themselves monotone in the predicted ratings of individual members. If all group members, except u, rate items i_1 and i_2 the same, i_1 will have at least the same group predicted rating score as i_2 if u rates i_1 no less than i_2. This holds for both the average and the least-misery strategies.

It is, however, not clear whether the group disagreement functions proposed in Definition 17 are monotone. In this section, we prove that the two group disagreement functions proposed can be transformed into aggregations of individual pairwise disagreements and become monotone. This means we can apply threshold algorithms to compute the overall recommendation score with individual predicted rating lists and pair-wise disagreement lists as inputs, and take advantage of the pruning power that threshold algorithms give us.

6.3.2 Monotonicity of Group Disagreements

We use a simple example group of two users to show that computing group disagreement based on predicted ratings of individual members is not monotonic. Figure 6.1(a) illustrates the two sorted predicted rating lists for the two users (u_1 and u_2). It is clear that

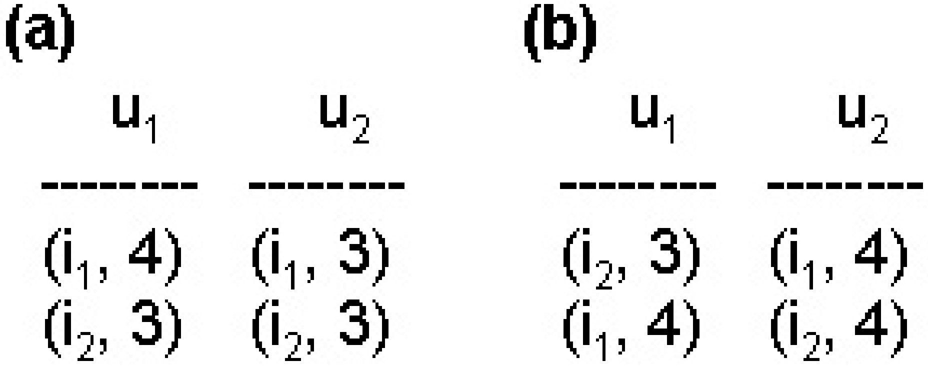


Figure 6.1. Group Disagreement is not monotonic w.r.t. predicted rating lists.

while i_1 has a higher predicted rating for u_1 than i_2 (4 versus 3), the group disagreement score for i_2 is in fact higher (1 instead of 0). The same non-monotonicity can be encountered when predicted rating lists are sorted in decreasing order (as shown in the example in Figure 6.1(b)). Hence, the problem of non-monotonicity of disagreement in predicted rating lists persists regardless of the order in which predicted rating lists are sorted.

To address this problem, we propose to maintain *pair-wise disagreement lists* instead and prove their monotonicity properties for the two group disagreement functions in Definition 17.

A pair-wise disagreement list (or simply disagreement list) for users u and v is a list of items which are sorted in the increasing order of the difference between their predicted rating scores for u and v. For an item i, we use $\Delta^i_{u,v} = |\texttt{predictedrating}(u,i) - \texttt{predictedrating}(v,i)|$ to denote this predicted rating difference.

Lemma 6.3.1. *The average pair-wise disagreement function in Definition 17 is monotonic w.r.t. pair-wise disagreement lists.*

Proof: Let us assume a group $\mathcal{G} = \{u_1, u_2, ..., u_p\}$ with all its $p(p-1)/2$ disagreement lists (one for each user pair). Also assume that there are a total of t items,

$\mathcal{I} = \{i_1, i_2, ..., i_t\}$. Note that we want to retrieve items with minimum disagreements first. Consider two items i_r and i_s within $\mathcal{I}$.

The group disagreement for i_r and i_s can be written as: $f \times \Sigma_{\forall j,k=1,2,...,p}(\Delta^{i_r}_{u_j,u_k})$ and $f \times \Sigma_{\forall j,k=1,2,...,p}(\Delta^{i_s}_{u_j,u_k})$, respectively, where $f = \frac{2}{p(p-1)}$ (see Definition 17).

Without loss of generality, assume we have

$\Delta^{i_r}_{u_x,u_y} < \Delta^{i_s}_{u_x,u_y}$, and $\forall j, k = 1, 2, \ldots, p, \Delta^{i_r}_{u_j,u_k} = \Delta^{i_s}_{u_j,u_k}$, where $(j, k) \neq (x, y)$. It is easy to see that

$f \times \Sigma_{\forall j,k=1,2,...,p}(\Delta^{i_r}_{u_j,u_k}) < f \times \Sigma_{\forall j,k=1,2,...,p}(\Delta^{i_s}_{u_j,u_k})$.

If the number of disagreement lists is restricted to m,[1] the monotonicity property can still be maintained by assuming the minimum disagreement values (0) for any unavailable user pairs during top-k computation. □

In the disagreement variance model in Definition 17, disagreement over an item is defined as the variance in predicted ratings among all group members. In other words, the predicted rating by each member is compared against the mean predicted rating of the group. We now show that this disagreement function can in fact be monotonically aggregated from pairwise disagreement lists.

Lemma 6.3.2. *The disagreement variance function in Definition 17 is monotonic w.r.t. pair-wise disagreement lists.*

Proof: Let us consider the group $\mathcal{G}$ and set of items $\mathcal{I}$ in Lemma 6.3.1. Consider two items i_r and i_s. The group disagreement of i_r and i_s can be written as:

$$\frac{\Sigma_{\forall j \in p}[\texttt{predictedrating}(u_j, i_r) - \frac{\Sigma_{\forall i \in p}\texttt{predictedrating}(u_i, i_r)}{p}]^2}{p}$$

and

$$\frac{\Sigma_{\forall j \in p}[\texttt{predictedrating}(u_j, i_s) - \frac{\Sigma_{\forall i \in p}\texttt{predictedrating}(u_i, i_s)}{p}]^2}{p}$$

[1] We discuss partial materialization of disagreements lists in Section 6.5.

We can transform this disagreement variance formula for i_r into (ignoring p):

$$[\Delta_{12}^{i_r}+\Delta_{13}^{i_r}+...+\Delta_{1p}^{i_r}]^2+[\Delta_{21}^{i_r}+\Delta_{23}^{i_r}+...+\Delta_{2p}^{i_r}]^2+...+[\Delta_{p1}^{i_r}+\Delta_{p2}^{i_r}+...+\Delta_{p(p-1)}^{i_r}]^2$$

which can be further expressed as:

$$[\Delta_{12}^{i_r}]^2 + ... + [\Delta_{1p}^{i_r}]^2 + ... + 2 \times [\Delta_{12}^{i_r}][\Delta_{13}^{i_r}] + 2 \times [\Delta_{12}^{i_r}][\Delta_{14}^{i_r}] + ...$$

It is clear that the above formula is a monotonic aggregation of $[\Delta_{jk}]\forall j,k \in p$. Without loss of generality, assume we have $\Delta_{u_x,u_y}^{i_r} < \Delta_{u_x,u_y}^{i_s}$, and $\forall j,k \in p, \Delta_{u_j,u_k}^{i_r} = \Delta_{u_j,u_k}^{i_s}$, where $(j,k) \neq (x,y)$. It is easy to see that the disagreement variance of i_r is less than the disagreement variance of i_s. Hence, we have proved that using pair-wise disagreement lists is sufficient to compute disagreement variance in a monotonic fashion. □

Materializing all possible pair-wise disagreement lists may not be practical since the number of such lists grows quadratically in the number of users. We discuss behavior factoring in Section 6.4 to save space and in Section 6.5, we discuss, given a fixed space constraint, which pairs to materialize in order to produce the best performance with threshold algorithms.

6.3.3 Group Recommendation Algorithms

Given a group $\mathcal{G}$, the goal, stated in Section 6.2.3, is to return the k best items according to a consensus function F (see Definition 18). We describe several algorithms for this problem; with each algorithm being a variant of the well-known TA [100] for top-k query processing.

The Full Materialization (FM) Algorithm: We start by describing Algorithm 18, which admits predicted rating lists $\mathcal{IL}$ of each user in the input group $\mathcal{G}$ and disagreement lists $\mathcal{DL}$ for every pair of users in $\mathcal{G}$. $\mathcal{IL}s$ are sorted in decreasing order of predicted rating and $\mathcal{DL}s$ are sorted in increasing order of disagreement. These predicted rating lists and disagreement lists of a group are akin to attributes on which the algorithm TA [100] works. We refer to Algorithm 18 as FM (Full Materialization).

Each $\mathcal{IL}$ is obtained using an individual recommendation strategy (as described in Section 6.2.1). Each $\mathcal{DL}$ is generated for a user pair and records the difference in scores for all items in their respective $\mathcal{IL}s$.

We showed in Section 6.3.1 that pairwise disagreement lists guarantee monotonicity for both pairwise and variance disagreements thereby allowing FM to rely on a threshold for early stopping. Our algorithm makes sequential access (SA) on each input lists (predicted rating and disagreement) in a round-robin fashion (lines 3 and 12) and reads an entry $e = (i, r)$, where i is the item-id and r is the predicted rating or disagreement value associated with it. There are two routines: ComputeExactScore which computes the score of the current item, and ComputeMaxScore which produces a new threshold value at each round. During the execution of the algorithm, we also maintain a bounded buffer(heap) which stores the top-k elements encountered thus far and their corresponding exact scores using the input consensus function F. If a new item is encountered during a sequential access (SA), ComputeExactScore performs a random access (RA) on all other predicted rating lists to compute the score of that item using the input consensus function F. The main difference between FM and TA is that while SAs are done on $\mathcal{IL}s$ and $\mathcal{DL}s$ interchangeably, RAs are only done on $\mathcal{IL}s$ (since disagreement can be computed from predicted ratings). In fact, $\mathcal{DL}s$ are not necessary to compute the final result. They are only there to compute the threshold and enable early termination.

`ComputeMaxScore` produces a new threshold value at each round. Its basic purpose is to provide an upper bound the score of any item that has not yet been seen by the algorithm. Thus, if r_u is the last predicted rating value read on list $\mathcal{IL}_u$ for all $u \in \mathcal{G}$, and $\Delta_{u,v}$ the last pairwise disagreement value read on disagreement list $\mathcal{DL}_{u,v}$ for all $u, v \in \mathcal{G}$, then the upper bound for the threshold (assuming the average pairwise disagreement model) is computed as follows:

$$\mathtt{F}(\mathcal{G}, i) \leq w_1 \times \frac{1}{|\mathcal{G}|} \sum_{u \in \mathcal{G}} r_u + w_2 \times (1 - \frac{2}{|\mathcal{G}|(|\mathcal{G}| - 1)} \sum_{u,v \in \mathcal{G}} \Delta_{u,v})$$

Algorithm 18: Group Recommendation Algorithm with Fully Materialized Disagreement Lists (`FM`)

```
Require: Group $\mathcal{G}$, consensus function F;
Retrieve predicted rating lists $\mathcal{IL}_u$ for each user $u$ in group $\mathcal{G}$;
Retrieve disagreement lists $\mathcal{DL}_{(u,v)}$ for each user pair $(u, v)$ in group $\mathcal{G}$;
Cursor $cur = \mathtt{getNext}()$ moves across each predicted rating and disagreement lists;
while ($cur <> $ NULL) do
  Get entry $e = (i, r)$ at $cur$;
  if $!(\mathtt{inHeap}(\mathtt{topKHeap}, e))$ then
    if ($\mathtt{ComputeMaxScore}(e.i, e.r, \mathtt{F}) > \mathtt{topKHeap}.k_{th}score$) then
      ComputeExactScore; Probe $\mathcal{IL}s$ to compute exact score $score$ of $e$ using F;
      if $score > \mathtt{topKHeap}.k_{th}score$ then
        $\mathtt{topKHeap.addToHeap}(e.i, score)$;
    else
      return topKList(topKHeap);
      Exit;
  $cur = \mathtt{getNext}()$;
return topKList(topKHeap);
```

The Ratings Only (RO) Algorithm: We next describe another variation of the algorithm, called `RO` (Ratings Only), which applies when only the predicted rating lists are present and none of the $\mathcal{DL}s$ are available. `RO` has the obvious benefit of consuming less space. As discussed above, the lack of disagreement lists does not have any impact on `ComputeExactScore`. However, it has an impact on how the `ComputeMaxScore` has to be modified to produce a (somewhat less tight) threshold value. More precisely, since disagreement lists are not available, we assume that the pairwise disagreement between each pair of users for any unseen item is 0. Thus the upper bound for the threshold value only comes from the last values read from each predicted rating list:

$$\mathtt{F}(\mathcal{G}, i) \leq w_1 \times \frac{1}{|\mathcal{G}|} \sum_{u \in \mathcal{G}} r_u$$

The Partial Materialization (PM) Algorithm: Finally, the most general variant is the case where only some disagreement lists are materialized, referred to as `PM` (Partial Materialization). As with `RO`, `PM` also has the obvious benefit of consuming less space than `FM`. In terms of processing, it differs from the others in how the threshold is computed. Let M be the set of all pairs of users for which disagreement lists have been materialized, the threshold can be computed as follows:

$$\mathtt{F}(\mathcal{G}, i) \leq w_1 \times \tfrac{1}{|\mathcal{G}|} \textstyle\sum_{u \in \mathcal{G}} r_u + \\ w_2 \times (1 - \tfrac{2}{|\mathcal{G}|(|\mathcal{G}|-1)} \textstyle\sum_{(u,v) \in M} \Delta_{u,v})$$

Intuitively, one may think that the more $\mathcal{DL}s$ are materialized, the tighter the score bound and hence, the faster the algorithm terminates. It turns out that it is not always the case. The basic intuition is that overall performance is a balance between the total number of distinct items which need to be processed before finding the best k items, referred to as `DIP` (Distinct Items Processed), and the number of sequential accesses, SAs, that result from the proliferation of disagreement lists. Consider the case of a 3-member group. The question we ask ourselves is when does using two materialized lists, $\mathcal{DL}_1$ and $\mathcal{DL}_2$, perform worse than when only one materialized list, say $\mathcal{DL}_1$, is used? If none of top items in $\mathcal{DL}_2$ is in the final output, each SA on $\mathcal{DL}_2$ is pure overhead. This is exacerbated if the the top items in $\mathcal{DL}_1$ and $\mathcal{DL}_2$, i.e., the ones with the least disagreement, are distinct. In both cases, if $\mathcal{DL}_2$ does not provide an opportunity to tighten the threshold, the number of SAs using $\mathcal{DL}_1$ and $\mathcal{DL}_2$ will be much higher than the number of SAs where only $\mathcal{DL}_1$ is used.

The `PM` variant raises an interesting question - which pair-wise disagreement lists should be materialized as a preprocessing step? This partial list materialization problem is discussed in the subsection 6.5. But first, in Section 6.3.4, we discuss interesting and novel techniques by which the threshold bounds can be sharpened even further.

6.3.4 Sharpening Thresholds

In this subsection we examine the different variants of the `TA` algorithm that we have developed thus far—`FM`, `RO` and `PM`—and suggest techniques by which their performance can be further improved, mainly by modifying the `ComputeMaxScore` function to compute sharper thresholds that enable earlier termination.[2]

Our approach is best illustrated by the following simple example. Consider a group consisting of two users $\mathcal{G} = \{u, v\}$. Recall that $\mathcal{IL}_u$ (resp. $\mathcal{IL}_u$) is the relevant list for user u (resp. v), and $\mathcal{DL}_{(u,v)}$ is the disagreement list of user pair u and v. Assume that the disagreement list has been materialized.

Consider a snapshot of the `FM` algorithm after a certain number of iterations. Let $r_u = 0.5$, $r_v = 0.5$ and $\Delta_{u,v} = 0.2$ be the last predicted rating and disagreement values read from each list respectively. The task of the `ComputeMaxScore` function is to provide an upper bound on the maximum possible value of the consensus function $\texttt{F}(\mathcal{G}, i)$ for any item i that has not yet been seen in any of the lists. Let the unseen item i's unknown predicted

[2] While these techniques appear very promising, we note that they are the subject of our ongoing investigations—we discuss them in this version of the paper primarily to illustrate their potential.

rating values be i_u and i_v for user u and v respectively. The consensus function is defined as:

$$\mathtt{F}(\mathcal{G}, i) = (i_u + i_v)/2 + (1 - |i_u - i_v|/1) \tag{6.1}$$

Since each list is sorted in decreasing order of predicted rating (increasing order of disagreement), it should be clear that the following inequalities hold:

$$0 \leq i_u \leq 0.5$$

$$0 \leq i_v \leq 0.5$$

$$0.2 \leq |i_u - i_v| \leq 1$$

As described in Section 6.3.3, our current approach provides a simple upper bound for $\mathtt{F}(\mathcal{G}, i)$ by substituting the upper bounds for i_u and i_v (and the lower bound for $|i_u - i_v|$) from the above inequalities, to arrive at the following threshold:

$$\mathtt{F}(\mathcal{G}, i) \leq (0.5 + 0.5)/2 + (1 - 0.2/1) = 0.5 + 0.8 = 1.3$$

However, a more careful examination of the inequalities reveals that this bound is not tight. Notice that i_u and i_v should be at least 0.2 units apart, thus both cannot be at 0.5. Since the upper bound of i_u is 0.5, i_v can be at most 0.3. Thus we can derive a sharper bound for $\mathtt{F}(\mathcal{G}, i)$ as follows:

$$\mathtt{F}(\mathcal{G}, i) \leq (0.5 + 0.3)/2 + (1 - 0.2/1) = 0.4 + 0.8 = 1.2$$

This example illustrates that due to the dependencies between the disagreement lists and the predicted rating lists, there are opportunities for deriving sharper thresholds for early termination after each iteration of the algorithm. More generally, after every iteration, we are faced with a formal *optimization problem* where we seek to maximize the consensus function over $|\mathcal{G}|$ real-valued variables, subject to various constraints on their values arising from the cursor positions on the predicted rating and disagreement lists. These optimization problems have seemingly complex formulations, because the consensus function as well the inequalities arising from disagreement lists are non-linear, involving absolute terms (e.g., of the form $|i_u - i_v|$) in the case of average pair-wise disagreement, as well as quadratic terms (e.g., of the form $(i_u - \mathtt{mean})^2$) in the case of variance based disagreement.

In this paper, we conduct a detailed investigation of the optimization problem involving the pair-wise disagreement model. Presence of absolute terms in the inequalities and consensus function makes the optimization problem non-linear; however we realize that the non-linear optimization problem can be reformulated as multiple linear optimization problems. Solution to this non-linear optimization can be achieved by solving each linear optimization problems individually and finally selecting the linear optimization solution that offers the maximum objective value.

Using LP-based reformulation technique, optimization problem in Equation 6.1 can be reformulated as two linear optimization problems:

a) Maximize

$$\mathtt{F}(\mathcal{G}, i) = (i_u + i_v)/2 + (1 - (i_u - i_v)/1)$$

s.t.

$$0 \leq i_u \leq 0.5$$

$$0 \leq i_v \leq 0.5$$

$$0.2 \leq (i_u - i_v) \leq 1$$

and

b) Maximize

$$\mathtt{F}(\mathcal{G}, i) = (i_u + i_v)/2 + (1 - (i_v - i_u)/1)$$

s.t.

$$0 \leq i_u \leq 0.5$$

$$0 \leq i_v \leq 0.5$$

$$0.2 \leq (i_v - i_u) \leq 1$$

Solution to problem 6.1 is the maximum of the objective values that linear optimization problems (a) and (b) take. In general, consensus function involving n variables requires $n!$ linear reformulations and solving each of them individually for obtaining the correct optimization value. However, at the same time the sizes of the problems themselves are very small, consisting of only a few variables and constraints (assuming user group sizes are small), and thus are likely to be efficiently solvable by reformulating the problem into multiple linear optimization problems with practically no overhead per iteration. Note that this reformulation only works for the absolute operator in the consensus function (pairwise disagreement model), and not for the quadratic operator (variance based disagreement model).

6.4 Behavior Factoring

In this section, we explore our first space saving strategy which relies on factoring shared behavior from disagreement lists. The intuition is that if two users have the same rating on a subset of the items, they can be treated as a single virtual user whose disagreement lists with other users should only be stored once. More precisely, if two users u and v agree on a set of items $\mathcal{S}$, their disagreement lists $\mathcal{DL}_{(u,w)}$ and $\mathcal{DL}_{(v,w)}$ with any other user v share the same disagreement values for items in $\mathcal{S}$. An extreme case is when u_1 and u_2 agree on every single item, the two lists $\mathcal{DL}_{(u,w)}$ and $\mathcal{DL}_{(v,w)}$ are the same. We begin by defining the *factoring set* of a pair of users.

Definition 19 (Factoring Set)**.** *A factoring set for a pair of users u and v is the largest set of items in which u and v agree. This set is referred to as $\mathcal{S}_{(u,v)} \subseteq \mathcal{I}$ and is defined as $\forall i \in \mathcal{S}_{(u,v)}, \Delta^i_{u,v} = 0$, where $\Delta^i_{u,v} = |\texttt{predictedrating}(u,i) - \texttt{predictedrating}(v,i)|$*

Given a pair of users, (u, v), $\forall w \in \mathcal{U}$ s.t., w is different from u and v, the disagreement lists $\mathcal{DL}_{(u,w)}$ and $\mathcal{DL}_{(v,w)}$, share the same values for items in $\mathcal{S}_{(u,v)}$.

We define a configuration $\mathcal{C}$ as the set of disagreement lists materialized for a user base $\mathcal{U}$. The algorithms developed in Section 6.3 admit different configurations as input. `FM` accepts a configuration where a disagreement list is created for every user pair in $\mathcal{U}$. `RO` accepts an empty configuration (since it only processes predicted rating lists).

Given a space constraint m (number of entries for storing materialized disagreement lists) and a configuration $\mathcal{C}$, factoring aims to output a configuration $(\mathcal{C}')$ such that $\texttt{size}(\mathcal{C}') \leq m$, where $\texttt{size}(\mathcal{C}') = \sum_{\mathcal{DL}_{(u,v)} \in \mathcal{C}'} (|\mathcal{DL}_{(u,v)}|)$. The size of predicted rating lists $\mathcal{IL}_{(u,v)}$ is ignored since they are not affected by factoring. We next describe the factoring algorithm in Section 6.4.1 and the modification to query processing in the presence of factored lists in Section 6.4.2.

6.4.1 Factoring Algorithm

The outline of the algorithm is as follows: factoring begins by deciding the user-pair which has the largest factored set (say user pair (u, v)). The factored set is removed from the original disagreement list of (u, v). That set is also removed from every other original disagreement list that is shared by either u (or v) and a third user (say w) and their disagreements on those items are stored only once in a common list (note that in the original case, items of the factored set are present in both (u, w) and (v, w)'s disagreement lists). This step overall achieves a space reduction. However, if the space budget (m) is not satisfied yet, same factoring strategy is repeated on the user base which has all other users except u and v. This factoring process is reiterated unless one of these two conditions are satisfied: a) overall space is reduced under m or b) no more factoring is possible.

Consider Figure 6.2 that illustrates one complete run of the proposed factoring algorithm on an example user base of size 5, $\{u, v, w, x, y\}$. Inputs to the factoring algorithm are a space budget (m, total no of entries in all pair-wise disagreement lists in the user base) and the set of all possible pair-wise disagreement lists of the user base. Figure 6.2(a) models the user base in form of a 5-node clique, where each user contributes one node in that clique. An edge between a pair of nodes is the pair-wise disagreement list between them. Note that, initially, the presence of all pair-wise disagreement lists make this graph complete, as shown in Figure 6.2(a). Next, it aims to compute factoring sets for every user pair and identify the user pair which has the largest factoring set. (Since all disagreement lists are of same size, largest factored set attains the highest space reduction.) Let that user pair be (u, v), as shown in Figure 6.2(b). Once (u, v) is identified, the disagreement between u and any other user, and disagreement between v and the same user, over items in their factoring set, are factored out and only stored once. The core primitive in the algorithm is to consider one triangle of users at a time involving edge (u, v) and perform factoring. Note that Figure 6.2(b) explains this step where $\mathcal{S}_{(u,v)}$ is factored out from disagreement list $\mathcal{DL}_{(u,v)}$. Next, $\mathcal{S}_{(u,v)}$ is factored out from $\mathcal{DL}_{(u,w)}$ and $\mathcal{DL}_{(v,w)}$ and is stored only once in $\mathcal{DL}_{\mathcal{S}_{(u,v)},w}$. Similarly, $\mathcal{S}_{(u,v)}$ is factored out from $\mathcal{DL}_{(u,x)}$, $\mathcal{DL}_{(v,x)}$ and $\mathcal{DL}_{(u,y)}$,$\mathcal{DL}_{(v,y)}$ and stored once in $\mathcal{DL}_{\mathcal{S}_{(u,v)},x}$ and $\mathcal{DL}_{\mathcal{S}_{(u,v)},z}$, respectively. Conceptually this step involves modifications of 3 triangles involving edge (u, v) for the given user base that consists of 5 users . For each triangle, overall space reduction is $\{2 \times |\mathcal{S}_{(u,v)}|\}$ after factoring. Note that, user pairs that do not involve either u or v are not affected so far in this factoring step. We show (w, x), (w, y) and (x, y) in solid lines in Figure 6.2(b) which remain unaffected after factoring w.r.t. user pair (u, v).

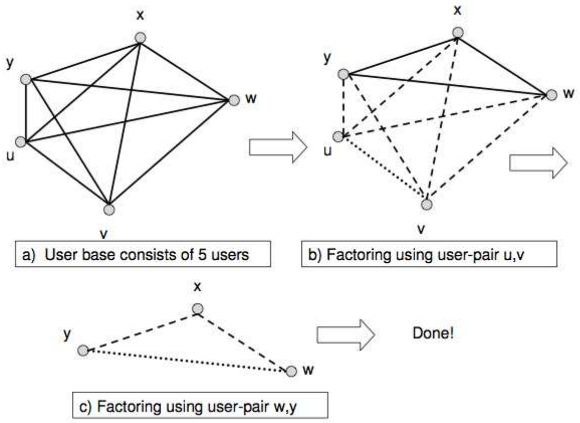

x
y
w
u
v
a) User base consists of 5 users
x
y
w
u
v
b) Factoring using user-pair u,v
x
y
w
c) Factoring using user-pair w,y
Done!

Figure 6.2. Factoring Steps.

Next, the factoring algorithm checks if the overall space now satisfies the specified space budget. It stops immediately if that condition is satisfied. Otherwise, it continues to the next step where it considers the largest complete graph (of size $>= 3$) which is not yet affected by factoring (the size 3 clique in the example). Note that the clique size gets reduced by 2 in two successive steps. Therefore, the algorithm computes the factored sets of the user pairs (w, x), (w, y), (x, y) and selects the one which has the largest factored set (say (w, y) as shown in Figure 6.2(c)). It adheres to the same factoring strategy as earlier by factoring out $\mathcal{S}_{(w,y)}$ from $\mathcal{DL}_{(w,y)}$, $\mathcal{DL}_{(w,x)}$, $\mathcal{DL}_{(y,x)}$ and storing it only once in $\mathcal{DL}_{\mathcal{S}_{(w,y)},x}$. The overall space reduction in this step is $\{2 \times |\mathcal{S}_{(w,y)|}\}$. Note that after this step, all disagreement lists are affected by factoring. Hence, the algorithm stops and outputs the factored disagreement lists.

Algorithm 19 summarizes the factoring strategy. One artifact of this factoring algorithm is it requires at least 3 user pairs to be effective. Note that, any disagreement list is factored out into at most two parts using our factoring strategy. We intend to explore more complex factoring techniques in the future.

Algorithm 19: Factoring

```
Require: Configuration $\mathcal{C}$, space budget $m$
Compute space requirement of userbase ProcessSpace($\mathcal{G}$) as $StorageR$.
ProcessedPair = null;
if (($|\mathcal{C}|$ $mod$ 2 = 0) and ($|ProcessedPair| = |\mathcal{G}| - 2$)) then
    Exit;
if (($|\mathcal{G}|$ $mod$ 2 $\neq$ 0) and ($|ProcessedPair| = |\mathcal{G}| - 1$)) then
    Exit;
while StorageR $> m$ do
    for each user pair ($(u, v) \in \mathcal{G}$) do
        if (u $\notin$ ProcessedPair) and (v $\notin$ ProcessedPair) then
            Configuration $\mathcal{C}_{(u,v)}$ = Factor($\mathcal{G}$, u, v);
            Compute storage requirement of $\mathcal{C}_{(u,v)}$ as ProcessSpace($u, v$);
            Compute $\Delta S_{(u,v)} = StorageR - \mathcal{C}_{(u,v)}$;
            Store Configuration $\mathcal{C}_{(u,v)}$, $\Delta S$ and ProcessSpace($u, v$) in CompProcessList;
    Select Configuration$(\mathcal{C}_{(x,y)})$ such that $\Delta S_{(x,y)}$ is maximum.
    Set StorageR = ProcessSpace($x, y$);
    Set ProcessedPair = $\{x, y\}$;
return $\mathcal{C}_{ProcessedPair}(StorageR)$;
```

Algorithm 20: Subroutine - Factor

```
Require: Configuration $\mathcal{C}$, user pair $u, v$
{Perform factoring of a Configuration wrt a particular user pair.}
Modify $\mathcal{DL}_{(u,v)}$ into $\mathcal{DL}_{\mathcal{S}_{(u,v)}}$ such that any item $\in \mathcal{DL}_{\mathcal{S}_{(u,v)}}$ is sorted in increasing disagreement value and $> 0$;
Add $\mathcal{DL}_{\mathcal{S}_{(u,v)}}$ in the Configuration($\mathcal{C}_{(u,v)}$);
Create list $\mathcal{DL}_{\mathcal{C}_{(u,v)}}$ from $\mathcal{DL}_{(u,v)}$ such that all items in $\mathcal{DL}_{\mathcal{C}_{(u,v)}}$ are 0.
for each $x \in \mathcal{G}$ and ($x \neq u, v$) do
    Decompose $\mathcal{DL}_{(x,u)}$ and $\mathcal{DL}_{(x,v)}$ in three lists
    Create disagreement list $\mathcal{DL}_{\mathcal{S}_{(u,v)},x}$ for items present in $\mathcal{DL}_{\mathcal{C}_{(u,v)}}$
    Create $\mathcal{DL}_{\mathcal{S}_{(x,u)}}$ from $\mathcal{DL}_{(x,u)}$ , $\mathcal{DL}_{\mathcal{S}_{(x,v)}}$ from $\mathcal{DL}_{(x,v)}$ such that an item $\in$ ($\mathcal{DL}_{\mathcal{S}_{(x,u)}}$ or $\mathcal{DL}_{\mathcal{S}_{(x,v)}}$) is not in $\mathcal{DL}_{\mathcal{S}_{(u,v)},x)}$
    Add $\mathcal{DL}_{\mathcal{S}_{(u,v)},x)}$, $\mathcal{DL}_{\mathcal{S}_{(x,u)}}$ and $\mathcal{DL}_{\mathcal{S}_{(x,v)}}$ in Configuration($\mathcal{C}_{uv}$);
return Configuration($\mathcal{C}_{(u,v)}$);
```

Algorithm 21: Subroutine - ProcessSpace

```
{Computes the space (number of entries) required to store a particular configuration}
Require: Configuration $\mathcal{C}$;
for each list $\mathcal{DL}_{\mathcal{S}_{(i)}} \in \mathcal{C}$ do
    Compute $TotalSpace = TotalSpace + \mathcal{DL}_{\mathcal{S}_{(i)}}$;
return $TotalSpace$;
```

Factoring $\mathcal{S}_{(u,v)}$ from the list $\mathcal{DL}_{(u,w)}$ (resp., $\mathcal{DL}_{(v,w)}$) results in converting $\mathcal{DL}_{(u,w)}$ (resp., $\mathcal{DL}_{(v,w)}$) into two lists: a *factored list* $\mathcal{DL}_{(u-\mathcal{S}_{(u,v)},w)}$ (resp., $\mathcal{DL}_{(v-\mathcal{S}_{(u,v)},w)}$), and a *common list,* $\mathcal{DL}_{\mathcal{S}_{(u,v)},w}$. In this case, the space saving is proportional to the size of the factoring set, $|\mathcal{S}_{(u,v)}|$. Hence, the larger the factoring set the higher the saving. Note that, factoring may fail to reach the specified budget (m) if factored sets are not large enough to reduce the overall space consumption to that extent. In fact, at the worst case, factoring fails to reduce any space if all factored sets are of length 0. However, factoring preserves all information of the original pair-wise disagreement lists and thus achieves space reduction without impacting performance.

6.4.2 Impact of Factoring on Query Processing

Algorithm 18 (`FM`) in Section 6.3.3 admits a group and a configuration containing all disagreement lists and outputs the best recommendations to the group given a consensus function. Here, we discuss how to adapt the algorithm to the case of a factored configuration where at least one disagreement list is factored out.

It turns out all is needed is to redefine `getNext`() to adapt query processing to work on factored disagreement lists. The main algorithm (Algorithm 18) does not need to be aware of such lists. Given a disagreement list $\mathcal{DL}_{(v,w)}$ which has been factored into two lists $\mathcal{DL}_{(v-\mathcal{S}_{(u,v)},w)}$ and $\mathcal{DL}_{\mathcal{S}_{(u,v)},w)}$, the `getNext`() routine on $\mathcal{DL}_{(v,w)}$ decides whether to advance the cursor on one list or the other. The decision is simply based on choosing the entry with the highest agreement value (lowest disagreement) among those two lists.

A consequence of confining the implementation to `getNext`() is that factoring does not modify the number of I/Os which makes it an appealing space saving strategy.

6.5 Partial Materialization

In the previous section we discussed the pre-processing technique of factoring that reduces the space required to store all the pairwise disagreement lists between users. However, if the set of n users is large, since the number of user pairs is quadratic in n, factoring alone may not be enough to reduce the space to manageable proportions. In such cases, it is more practical to materialize (i.e., retain) only a small but effective subset of the disagreement lists. The central problem that we consider in this section is thus: given a fixed space constraint m, to determine (after factoring) which lists to materialize such that the total space consumed by these lists is at most m, and these lists are of "maximum benefit" during recommendation processing[3].

Intuitively, a (factored) disagreement list should be materialized if (a) the corresponding users together are more likely to be a part of the same group, and (b) materializing the list significantly improves the running time of top-k recommendation algorithms. In the following subsections we formalize this problem and develops algorithms to address it.

Our discussion will proceed in two stages. We shall first consider the simple scenario, where factoring has not been applied to a configuration C. In this scenario, all disagreement lists in C are equal in size, and each contains r entries where r is the total number of items. In Section 6.5.1 we discuss a simple algorithm that materializes a subset (at most m/r) of these disagreement lists that are of maximum benefit during recommendation processing, i.e., such that the average processing time is least affected.

However, once factoring has been performed, each original disagreement list may be composed of up to two factored lists of varying sizes. For example, using the factoring set $\mathcal{S}_{(u,v)}$, a disagreement list $\mathcal{DL}_{(v,w)}$ will be decomposed into two lists $\mathcal{DL}_{(v-\mathcal{S}_{(u,v)},w)}$ and $\mathcal{DL}_{\mathcal{S}_{(u,v)},w)}$. The sum of those lists' sizes is the same as that of $\mathcal{S}_{(u,v)}$. We discuss this more general situation in Section 6.5.2, where the task is to materializes a subset of the (factored) disagreement lists such that the total number of entries in all the materialized lists is m, and the average processing time is least affected. We formalize this new problem as an adaptation of the well-known NP-hard Knapsack Problem [101] and develop an approximation algorithm to address it.

[3] We assume that m represents a user-specified threshold on the total number of entries in all the materialized disagreement lists.

6.5.1 Partial Materialization Without Factoring

Let the set of users be $\mathcal{U} = u_1, \ldots, u_n$. Recall that $\mathcal{IL}_u$ is the predicted rating list for user u, and $\mathcal{DL}_{(u,v)}$ is the disagreement list of user pair u and v. Let the set of all possible user pairs in $\mathcal{U}$ be $S = \{(u,v)|u, v \in \mathcal{U}\}$. Let $M \subset S$ be the (unknown) subset of user pairs whose corresponding disagreement lists we wish to materialize (i.e., $|M| = m/r$). Let $\mathcal{G} \subseteq \mathcal{U}$ be any user group. Let $p(\mathcal{G})$ be the probability (or likelihood) that $\mathcal{G}$ will be the next "query", i.e., the next group that will seek item recommendations. Let $t_M(\mathcal{G})$ be the execution time of the top-k algorithm on user group $\mathcal{G}$ when run using the predicted rating lists $\mathcal{IL}_u$ (for all $u \in \mathcal{G}$) *as well as* the disagreement lists $\mathcal{DL}_{(u,v)}$ (for all $u, v \in \mathcal{G}$) that have been materialized in M, i.e., using algorithm `FM`. (Note that therefore $t_\phi(\mathcal{G})$ denotes the execution time of the top-k algorithm on user group $\mathcal{G}$ when run using only the predicted rating lists $\mathcal{IL}_u$ (for all $u \in \mathcal{G}$), i.e., using algorithm `RO`.

Our objective is to minimize the expected cost of executing the top-k algorithm on any user group query, using the predicted rating lists as well the disagreement lists. Let the expected cost be denoted as t_M. The partial materialization of disagreements list problem may now be formally defined as follows.

PROBLEM (Partial Materialization Without Factoring). *Determine the subset of pairs $M \subseteq S$ s.t. $|M| = m/r$ and $t_M = \sum_{\mathcal{G} \subseteq \mathcal{U}} p(\mathcal{G}) t_M(\mathcal{G})$ is minimized.*

Although clearly very important and practical, the partial materialization problem is unfortunately quite hard to solve optimally. There are several reasons for this. First, it is very difficult to get reliable and accurate estimates for the distribution $p(\mathcal{G})$, i.e., the probability that a given user group $\mathcal{G}$ will be queried next. Moreover, the set of possible user groups is exponential in n, so it is not clear how such information can be compactly represented, even if it were reliably available. Next, due to the complex dependencies involved, it is very hard to estimate the impact of a materialized disagreement list in improving the running time of a top-k algorithm, without actually materializing candidate disagreement lists and running the top-k algorithms with and without the lists to determine their benefit. Finally, an important parameter of a top-k algorithm is the value of k, which is usually unknown at pre-processing time. As a first step toward addressing these challenges, we propose several principled and practical solutions.

6.5.1.1 A Simplifying Assumption, and a Simple Lists Materialization Algorithm

In order to make the problem more tractable, we make the following simplifying assumption. We assume that each future user group query $\mathcal{G}$ will only contain exactly two users, and moreover, $p(\mathcal{G})$ is reliably known for all pairs of users $\mathcal{G}$. This assumption is of course patently false, but we emphasize here that we use it only for simplifying the computation of M. Once M has been computed and the corresponding disagreement lists materialized, we shall later show that they can be used at query time for answering any user group $\mathcal{G}$, even groups containing more than two users.

This assumption considerably simplifies the computation of M, which can now proceed as follows. Recall that S is the set of all $n(n-1)/2$ pairs of users. For every pair of users u and v, we temporarily materialize the disagreement list $\mathcal{DL}_{(u,v)}$, and compute $t_{\{(u,v)\}}(\{u,v\})$ as well as $t_{\phi}(\{u,v\})$ by running the top-k algorithm twice, once with the disagreement list, and once without the disagreement list, respectively.[4]

We can then eliminate from S those pairs $\{u,v\}$ where $t_{\{(u,v)\}}(\{u,v\}) \geq t_{\phi}((u,v))$

Although situations where the additional use of a disagreement list actually hurts the top-k execution may appear counter-intuitive, they can occur. For example, consider two users that are very similar to each other (e.g., they agree on most items) or are very dissimilar to each other (e.g., they disagree on most items). In both cases, their disagreement

[4] Performance numbers are obtained for a fixed k, specifically set for each application. E.g., in a movie recommendation, 10 movies is typical

list contains very similar disagreement values (mostly 0's, or mostly 1's, respectively), and consequently is of no help in forcing early termination of the top-k algorithm, and in fact hurts the execution because of the extra sequential list accesses incurred. A disagreement list is useful for forcing early termination *only if there is significant skew in its disagreement scores*, i..e, at the top of the list the users agree on most items, whereas their disagreement is more pronounced as we go deeper into the list.

Let the remaining set of pairs be S'. Then, we should select M from S' such that following expression is maximized:

$$\sum_{(u,v)\in M} p(\{u,v\}) \cdot (t_{\phi}(\{u,v\}) - t_{\{(u,v)\}}(\{u,v\}))$$

Algorithm 22: Partial Materialization Without Factoring

```
Require: User pairs in $S'$;
1: Sort the pairs $(u,v) \in S'$ by decreasing $p(\{u,v\}) \cdot (t_{\phi}(\{u,v\}) - t_{\{(u,v)\}}(\{u,v\}))$;
2: Return the $m/r$ pairs with the largest values.
```

Algorithm 22 shows a very simple approach to compute M optimally. The algorithm requires $O(n^2)$ executions of the top-k algorithm. Even though this is a pre-processing step, it may nevertheless be very time consuming. We discuss in Section 6.5.1.2 additional techniques by which this can be reduced.

The disagreement lists materialization procedure discussed above assumed that the user groups are restricted to two members only. However, once the m/r lists have been materialized, they can be used at query processing time for user groups of any size in a straightforward manner. Consider any arbitrary user group $\mathcal{G}$. In executing the top-k recommendation algorithm for this group, we use the predicted rating lists $\mathcal{IL}_u$ (for all $u \in \mathcal{G}$) as well as all disagreement lists $\mathcal{DL}_{(u,v)}$ (for all $u, v \in \mathcal{G}$) that have been materialized in M.

6.5.1.2 Avoiding Examining all User Pairs

In a large user base, it is very likely that many user pairs are almost never going to occur in query groups. In order to reduce pre-processing costs, it is critical that we identify only those user pairs that have significant likelihood of occurring together, and only consider such pairs in the above algorithm.

If we have a rich *query log* (or workload) of past user groups, then it is possible to analyze the query log in determining this information. For example, let $\mathcal{G}_1, \ldots, \mathcal{G}_q$ be a query log of q user groups. Then for any user pair (u, v), we can compute

$$p(\{u, v\}) = \frac{|\{\mathcal{G}_i | u, v \in \mathcal{G}_i\}|}{q}$$

This computation can be carefully done to ensure that we only compute the probabilities for those user pairs that occur in the query log, thus avoiding having to examine a vast majority of the user pairs that never occur together. Moreover, even for user pairs that occur together in the query log, we can eliminate those that have extremely low probabilities.

6.5.2 Partial Materialization after Factoring

We next consider the more complex case when the disagreement lists have already been factored. Recall that given a factoring set $\mathcal{S}_{(u,v)}$, each original disagreement list $\mathcal{DL}_{(u,w)}$ is now factored into a possibly smaller list $\mathcal{DL}_{u-\mathcal{S}_(u,v),w}$ such that the original list is the union of the factored list $\mathcal{DL}_{u-\mathcal{S}(u,v),w}$ and a common list $\mathcal{DL}_{\mathcal{S}_{(u,v)},w)}$ for some other user w.

Our partial materialization goal will be to identify the subset of pairs $M \subseteq S$ such that both the factored as well as common component of the original disagreement list for each such pair is materialized. Using notation similar to Section 6.5.1, let $t_M(\mathcal{G})$ be the execution time of the top-k algorithm on user group $\mathcal{G}$ when run using the predicted rating lists $\mathcal{IL}_u$ (for all $u \in \mathcal{G}$) *as well as* the materialized (factored as well as common) disagreement lists corresponding to all user pairs (u, v) that appear in both M and $\mathcal{G}$. Our objective is to minimize the expected cost of executing the top-k algorithm on any user group query, using the predicted rating lists as well the materialized factored and common disagreement lists. Let the expected cost be denoted as t_M. Given a space budget m, the partial materialization problem after factoring problem may be formally defined as follows.

PROBLEM (Partial Materialization After Factoring). *Determine the subset of pairs $M \subseteq S$ s.t. the space required by all factored and common lists corresponding to all pairs in M is at most m, and $t_M = \sum_{\mathcal{G} \subseteq \mathcal{U}} p(\mathcal{G}) t_M(\mathcal{G})$ is minimized.*

As before, we will make the simplifying assumption that each future user group query $\mathcal{G}$ will only contain exactly two users, and $p(\mathcal{G})$ is reliably known for all pairs of users $\mathcal{G}$. We also reduce the set of user pairs from S to S', eliminating those pairs for which the availability of the disagreement list does not improve the query processing time.

Let $\mathcal{DL}_{\mathcal{S}(P_i)}$ be the factored list corresponding to any user pair $P_i \in S'$. Since common lists are shared, let $C(S')$ represent the set of all common lists corresponding to S'. Then the space consumed by all factored as well as common lists is

$$Space(S') = \sum_{P_i \in S'} |\mathcal{DL}_{\mathcal{S}(P_i)}| + \sum_{\mathcal{DL}_C \in C(S')} |\mathcal{DL}_C|$$

It may be that this space is still greater than the space constraint m. In this case, we will have to remove a few more user pairs from S', eliminating those pairs for which the availability of the disagreement list adversely impacts query processing time the least.

For user pair $(u, v) = P_i$, let the *benefit* B_i be defined as

$$B_i = p(\{u, v\}) \cdot (t_\phi(\{u, v\}) - t_{\{(u,v)\}}(\{u, v\}))$$

The residual problem can be formally defined as follows.

PROBLEM (0/1 Knapsack-Based Formulation of Partial Materialization After Factoring). *Determine the subset of pairs $M \subseteq S'$ s.t.*

$$\sum_{P_i \in M} B_i$$

is maximized, subject to

$$Space(M) = \sum_{P_i \in M} |\mathcal{DL}_{\mathcal{S}(P_i)}| + \sum_{\mathcal{DL}_C \in C(M)} |\mathcal{DL}_C| \leq m$$

We note that this problem is similar, but not identical, to the classical NP-Hard 0/1 Knapsack Problem [101]. This is because the space constraint contains a term that represents the space consumed by the common lists of M. If this term were not there, then the formulation can be easily seen to be identical to 0/1 Knapsack.

In solving this problem, we leverage the well-known greedy 1/2-approx algorithm for 0/1 Knapsack, suitably modified to account for the extra complexity of having to consider the materialization of common lists.

Algorithm 23: Partial Materialization After Factoring

```
Require: User pairs in $S'$;
Sort the pairs $P_i \in S'$ by decreasing $B_i/|\mathcal{DL}_{\mathcal{S}(P_i)}|$
$M = \{\}$
$i = 1$
while $Space(M) + |\mathcal{DL}_{\mathcal{S}(P_i)}| \leq m$ do
    $M = M \cup P_i$; $i++$;
if $\sum_{P_j \in M} B_j \geq B_i$ then
    return $M$
else
    return $P_i$
return
```

Algorithm 23 essentially orders the pairs in S' by decreasing "benefit density", except that in the calculation of this density, the common lists are not considered. The common lists are only considered in the space calculation of M. The returned user pairs are either (a) the largest prefix of this ordered list that can fit within the space budget, or (b) the very last user pair that causes the space to exceed the budget.

While Algorithm 23 is not an optimal algorithm for the problem, it is adapted along the lines of the 1/2-approx algorithm for the classical 0/1 Knapsack problem, and our experiments indicate it is both efficient and provides solutions of good quality. More interestingly, when run on un-factored disagreement lists, it is *identical* to Algorithm 22 which is optimal for that case. As shown in our experiments, for the same space constraint, factoring followed by partial materialization is always better than partial materialization alone.

6.6 Experiments

We evaluate our group recommendation system from three major angles. First, from the *quality* perspective, we conduct an extensive user study through Amazon Mechanical Turk[5] to demonstrate that group recommendations with the consideration of disagreements are superior to those relying on aggregating individual predicted rating scores alone (Section 6.6.1). Second, from the *performance* perspective, we conduct a comprehensive set of experiments to show that our materialization algorithms can achieve better pruning than alternative algorithms (Section 6.6.2). Third, we investigate the performance of our space saving strategies with respect to both space and time.

We implemented our prototype system using JDK 5.0. All performance experiments were conducted on an Intel machine with dual-core 3.2GHz CPUs, 4GB Memory, and

[5] https://www.mturk.com/

Table 6.1. Statistics about the MovieLens Data Set.

# users	# movies	# ratings
71,567	10,681	10,000,054

500GB HDD, running Windows XP. The Java Virtual Memory size is set to 256MB. All numbers are obtained as the average of three runs.

Data Set: We use the MovieLens [102] 10M ratings data set for evaluation purposes. The statistics of this data set is shown in Table 6.1.

Individual Predicted Ratings: We adopt collaborative filtering [98] for generating individual predicted ratings as described in Section 6.2.1.2, where the user-user similarity, $\texttt{UserSim}(u, u')$, is computed as follows:

$$\texttt{sim}(u, u') = \frac{|\{i | i \in \mathcal{I}_u \wedge i \in \mathcal{I}_{u'} \wedge |\texttt{rating}(u,i) - \texttt{rating}(u,i)| \leq 2\}|}{|\{i | i \in \mathcal{I}_u \vee i \in \mathcal{I}_{u'}\}|}$$

where $\mathcal{I}_u$ denotes the set of items u has rated. We consider a movie to be shared between two users if they both rated it within 2 of each other on the scale of 0 to 5.

6.6.1 User Study

We conduct an extensive user study through Amazon Mechanical Turk to compare our proposed group recommendation consensus functions with prior group recommendation mechanisms, which rely solely on rating aggregations. In particular, we compare four group recommendation mechanisms:

Average Rating (AR), which computes the group recommendation score as the average of individual predicted ratings. The disagreement weight is set to zero.

Least-Misery Only (MO), which computes the group recommendation score as the minimum individual predicted rating among all group members. Again, the disagreement weight is set to zero.

Consensus with Pair-wise Disagreement (RP), which computes the group recommendation score as a weighted summation of the average predicted rating and the average pair-wise disagreements between all group members.

Consensus with Disagreement Variance (RV), which computes the group recommendation score as a weighted summation of the average predicted rating and the variance of individual predicted ratings among all group members.

The user study is conducted in two phases: *User Collection Phase* and *Group Judgment Phase*. At each phase, a series of HITs (Human Intelligence Tasks) are generated and posted on Mechanical Turk, Amazon users are invited to complete those tasks.

6.6.1.1 User Collection Phase

The goal of the User Collection Phase is to recruit users and obtain their movie preferences. Those users will later form groups and perform judgments on group recommendations.

Preferences Collection: Asking a user to go through all ten thousand movies in our system and give ratings as they go is clearly not practical. Therefore, we selected a subset of the movies for users to provide their preferences. We considered two factors in selecting those movies: *familiarity* and *diversity*. On one hand, we want to present users with a set of movies that they do know about and therefore can provide ratings. On the other hand, we want to maximize our chances of capturing the different tastes among moviegoers. Towards these two goals, we select two sets of movies. The first set is called the *popular set*, which contains the top-40 movies in MovieLens in terms of popularity (i.e., the number of users who rated a movie in the set). The second set is called the *diversity* *set*, which contains the 20 movies in MovieLens that have the highest variance among their user ratings and that are ranked in the top-200 in terms of popularity. We created two HITs with 40 movies each. The **Similar HIT** consisted entirely of the movies within the *popular set* and the **Dissimilar HIT** consisted of the top-20 movies from the *popular set* and the 20 movies from the *diversity set*. Fifty users were recruited to participate in each HIT. Users are instructed to provide a rating between 0 and 5 (5 being the best) for at least 30 of the 40 movies listed (in random order) according to their preferences. In addition to their ratings, we also record their Mechanical Turk IDs for future reference.

Group Formation: We consider two main factors in forming user groups: *group size* and *group cohesiveness*. We hypothesize that varying group sizes will impact the difficulties in reaching consensus among the members and therefore affect to which degree members are satisfied with the group recommendation. We chose two group sizes, 3 and 8, representing small and large groups, respectively. Similarly, we hypothesize that group cohesiveness (i.e., how similar are group members in their movie tastes) is also a significant factor in the satisfaction with group recommendation. As a result, we chose to form three kinds of groups: *similar, dissimilar, random.* A similar group is formed by selecting users who: 1) have completed the **Similar HIT** described above; 2) combined with having the maximum summation of pair-wise similarities (between group members) among all groups of the same size. A dissimilar group is formed by selecting users who: 1) have completed the **Dissimilar HIT** described above; 2) combined with having the minimum summation of pair-wise similarities (between group members - based on the provided ratings) among all groups of the same size. Finally, a random group is formed by randomly selecting users from all the pool of available users. Table 6.2 illustrates the average similarity between group members of the six groups formed.

Table 6.2. Similarities of User Study Groups.

	Size = 3	Size = 8
Similar Group	0.89	0.90
Dissimilar Group	0.29	0.27
Random Group	0.69	0.73

6.6.1.2 Group Judgment Phase

The goal of the Group Judgment Phase is to obtain ground truth judgments on movies by users in a group setting. Those judgments can then be used to compare group recommendation generated by the four different mechanisms **AR**, **MO**, **RP** and **RV**.

Individual Recommendation: For each user in one of the six groups in table 6.2, we generated and materialized a list of individual recommendations against the MovieLens database using collaborative filtering.

Group Recommendation Candidates: For each group, we generated group recommendations using all of our four strategies. The resulting recommendation lists were combined into a single set of distinct movies, called *group candidate set*. This ensures that we obtain ground truth judgments on all the movies we will encounter using any of the four strategies.

For each group, a **Group HIT** was generated and contained the following group context: for each movie in the group candidate set, the individual recommendation score of each member. The users are then instructed to decide *whether a movie in the group candidate set is suitable for recommendation given its group context*. Users from the previous phase were invited back (with a higher payout) to participate in the HITs which correspond to a group to which they belong. Additional users were also recruited to participate in the HITs to complement the set of prior users, and they were instructed to pretend themselves to be one of the group members in the HIT. At the conclusion of the user study, on average 5 users participated in the three 3-member-group HITs and 10 users participated in the three 8-member-group HITs, for a total of 45 users.

6.6.1.3 Result Interpretation

Given a Mechanical Turk user's ground truth evaluation of the candidate movies, we adopt the Discounted Cumulative Gain (DCG) [103] measure to evaluate each of the following six group recommendation strategies (note that the least-misery model by definition considers only one member of the group and therefore can not be combined with either of the disagreement models):

AR, MO: these two are group recommendation lists generated based on average and least-misery models, respectively, without the disagreement component.

RP20, RP80: these two are group recommendation lists generated by combining the average predicted ratings model with the pair-wise disagreement model. RP20 sets w_2 in Definition 18 to 0.2, while RP80 sets it to 0.8.

RV20, RV80: these two are group recommendation lists generated by combining the average predicted ratings model and the disagreement variance model. RV20 sets w_2 in Definition 18 to 0.2, while RV80 sets it to 0.8.

Each strategy generates a 10-movie recommendation list and for a given list, its DCG value is calculated as follows:

$$DCG_{10} = rating_1 + \sum_{i=2}^{10} \frac{rating_i}{\log_2(i)}$$

where $rating_i$ is the ground truth (provided by the Mechanical Turk user) of the movie at position i, and is either 1 (the user considers this movie suitable for the group setting) or 0 (otherwise). We further normalize the DCG value into a range between 0 and 1 by dividing it by the DCG value of the ideal list to produce the nDCG value. (The ideal list is obtained by re-sorting the movies in the list in the order of their predicted ratings.)

For each group with a given size and cohesiveness, the nDCG values of each recommendation list are computed as the average of all the users who participated in the group HIT. The results are shown in Figure 6.6.1.3.

The top-left chart in Figure 6.6.1.3 reports the nDCG for small and large groups of similar users. In a real world setting, a group of friends can be thought of as such a group. According to this chart, **MO** results in the best performance for both small and large groups. This can be explained as a group activity of similar users, where the objective is to agree with the person who has the harshest opinion. **MO** is most practical for this setting since agreeing upon the worst opinion results in the least disagreement from a user's personal opinion. It is also interesting to notice, that for large groups, **MO** performs very well. The next best strategy is **AR**, which is intuitively true for any set of similar users - people with very high similarity have no difference in their opinion. **RV80** and **RP80** perform worst since there is hardly any scope of difference in opinion in a group of similar users.

The top-right chart in Figure 6.6.1.3 reports the nDCG for small and large groups of dissimilar users. In a practical setting, a group of family members, whose tastes typically differ is a good example here. For dissimilar users, differences in opinion is conspicuous hence needs to be captured carefully. Indeed, we can see that, our disagreement based models **RV80, RP80** start performing better than other two models. Specifically, for large groups, **RV80** results in the best value of nDCG while the predicted rating based models are useless. This observation corroborates our initial claim that formalizing disagreement as a component of the consensus function is important for group recommendation.

The bottom-left chart in Figure 6.6.1.3 reports the nDCG for small and large groups of random users. A random group can consist of both similar and dissimilar users. For

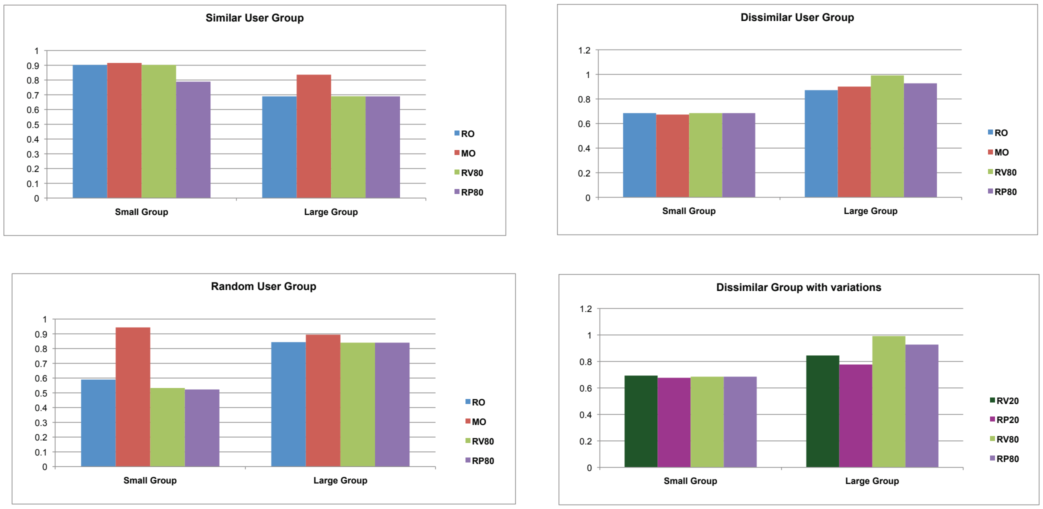


Figure 6.3. Comparison of User predicted ratings (using NDCG) among Different Group Recommendation Lists.

small groups, **MO** works best, whereas, for large groups, there is no significant difference between all four strategies.

The bottom-right chart in Figure 6.6.1.3 reports the differences in our disagreement models (notice the different weights) for dissimilar user groups. It is interesting to notice that, for small groups, all four disagreement models perform equally well in general. However, for large groups, disagreement becomes a conspicuous part in decision making. Consequently, the disagreement strategies **RV80, RP80** outweigh the other two models **RV20, RP20**.

To summarize, we can say that user similarity in a group as well as group size should be accounted in modeling disagreement in the consensus function. One of our planned experiment is to involve users more actively in the final judgment by letting group members consult with each other and reach consensus in an iterative manner as described in [91]. Such feedback would help draw a stronger connection between group size and overall group dynamics in group recommendation.

6.6.1.4 Effectiveness of Group Ratings

We next perform user studies to validate the effectiveness of the group ratings. More precisely, we ask users to compare group ratings generated by our group recommendation strategies with the individual ratings obtained directly from the underlying recommendation system. Again, **Group HITs** are generated based on similar and dissimilar groups. Additional users were recruited to participate in the HITs to complement the set of prior users, and they were instructed to pretend themselves to be one of the group members in the HIT. Within each HIT, a 10-movie recommendation list is presented to each user within the context of a group. Each movie comes with the individual predicted rating and the group rating generated by one of our group recommendations strategies (**RP80** and **RV80**). The users were then instructed to give their *preference for either the group or the individual rating for each movie*, although the explicit model name was kept hidden from them. Additionally, they were also required to *describe their satisfaction level for the group ratings overall*, in the scale of 1-5. In this new user study, on average 15 users participated in the each of the two (similar and dissimilar) 3-member-group HITs and 8 users participated in each of the two (similar and dissimilar) 8-member-group HITs, for a total of 218 users.

Result Interpretation: For each group with a given size and cohesiveness, we calculate the percentage of user's preference for group ratings corresponding to a strategy and compare that with the percentage of user's preference for individual ratings. The results are listed in Figure 6.6.1.4. In all cases, group ratings are preferred by more than 50% of users. It is also easy to observe that the group ratings are more preferred over individual ratings for similar user groups as compared to dissimilar user groups. The reason is intuitive and can be explained as follows: similar users have similar ratings for movies; hence with a small compensation, they can match their individual preference with the group preference. However, for dissimilar user groups, preference varies widely among group members - hence

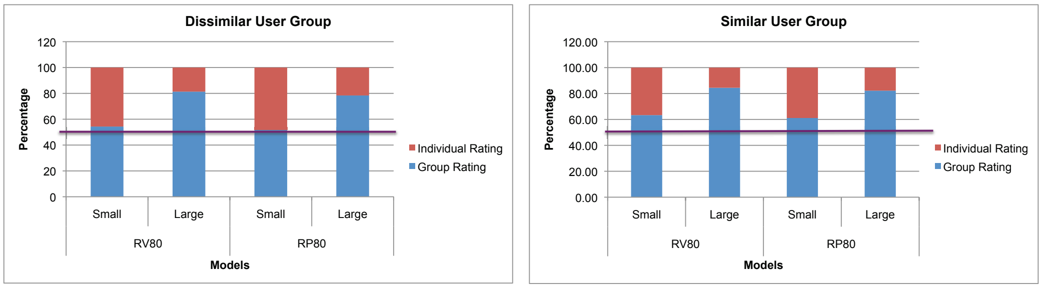


Figure 6.4. Comparison of Percentage of User Preference for Group Ratings and Individual Predicted Ratings among Different Group Recommendation Lists.

Table 6.3. Dissimilar User Group - Overall model ratings

Rating	**RP80**		**RV80**	
	small	*large*	*small*	*large*
1	0%	0%	0%	0%
2	5%	0%	8%	3%
3	31%	20%	28%	17%
4	42%	44%	60%	36%
5	22%	36%	4%	44%

dissimilar users are more reluctant to adopt group ratings. Another interesting observation is, irrespective of group cohesiveness, members in large groups prefer group ratings more than members in small groups do. This corroborates the efficacy of our group recommendation strategies which are designed to minimize the difference in opinions between group members individual preference and are more conspicuous for larger groups.

Table 6.3 and Table 6.4 record the percentage of overall group ratings (in the scale of $1 - 5$) of different group recommendation strategies for different group cohesiveness and group size. It can be easily observed from the tables that proposed group recommendation strategies are highly rated (mostly 3 and above) always, irrespective of the size and cohesiveness of the group under consideration.

Table 6.4. Similar User Group - Overall model ratings

Rating	**RP80**		**RV80**	
	small	*large*	*small*	*large*
1	3%	0%	5%	0%
2	14%	0%	8%	0%
3	14%	14%	20%	11%
4	52%	30%	40%	41%
5	17%	56%	27%	48%

6.6.2 Performance Evaluation

In this section, we analyze the performance of the three group recommendation algorithms described in Section 6.3: Dynamic Computation with Predicted Rating List Only (`RO`), Full Materialization (`FM`), and Partial Materialization with a given budget on number of lists (`PM`). At the core of all three algorithms is the top-k TA algorithm [61], which scans down the input lists and stops processing when score bounds indicate that no more items qualify. The cost of `TA` is determined by two factors: the number of *sequential accesses*, which corresponds to the number of `next()` calls made during the scan of each list, and the number of *random accesses*, which corresponds to the number of calls made to each list for score retrieval given an item. During the processing, when the buffer is bounded and only the top-k items are kept, the number of random accesses is proportional to the number of sequential accesses. When the buffer is unbounded, the number of random accesses is proportional to the *number of distinct items processed*. We adopt the bounded buffer version of the TA algorithm and therefore mostly measure the number of sequential accesses to compare the performance between various algorithms.

In addition to that, we also compare our proposed group recommendation algorithms with a very simple baseline approach- `Without-Fagin RO`. This algorithm works as follows: It works only with the set of lists relevant to a specific group. This algorithm doesn't work in Fagin(top-k) style; i.e., it can not acquire any early stopping using upper bound value of thresholds. In order to compute the top-k group ratings, it maintains a heap and stores the top-k ratings encountered thus far. However, the algorithm can only terminate once the entire database is scanned and outputs the top-k best ratings thereafter.

Group Formation: Groups are formed by selecting users from the MovieLens database. The key factor we consider is group cohesiveness (or similarity). We defined four group similarity levels: $0.3, 0.5, 0.7, 0.9$, with a margin of ± 0.05. To form a group of 3 with similarity 0.3, we select three users u_1, u_2, u_3 from the database, such that $\forall i, j, 0.25 < \mathtt{sim}(u_i, u_j) < 0.35$, where $1 \leq i, j \leq 3, i \neq j$. The other factors we consider are number of recommendations being produced (small = 5, medium = 10, large = 30) and the size of groups (small = 3, medium = 5, large = 8).

Summary of Results: Our first observation is that group similarity has a direct impact on the number of sequential accesses (SAs). This is not surprising: the predicted rating lists of similar users tend to contain similar items at similar positions, including those with high predicted ratings. Our second observation is that some Disagreement Lists ($\mathcal{DL}$s) almost always guarantee earlier stopping. Hence, RO wins in very few cases. However, the presence of $\mathcal{DL}$s is not always beneficial and can sometimes become *redundant*. In fact, the results show that for different user groups, different strategies (RO, FM or PM) will win. In particular, a higher number of $\mathcal{DL}$s does not guarantee earlier stopping. The proliferation of lists may increase the number of SAs and also the number of distinct items seen unnecessarily, thereby hurting the performance in the end. In addition to that, we compare our three algorithms, with the baseline approach Without-Fagin RO. Eventually, as shown in Figure 6.5, This algorithm always scans the entire database and encounters all items in the database before producing the output. Consequently, it attains the worst performance among all. We provide detailed descriptions on our experiments below.

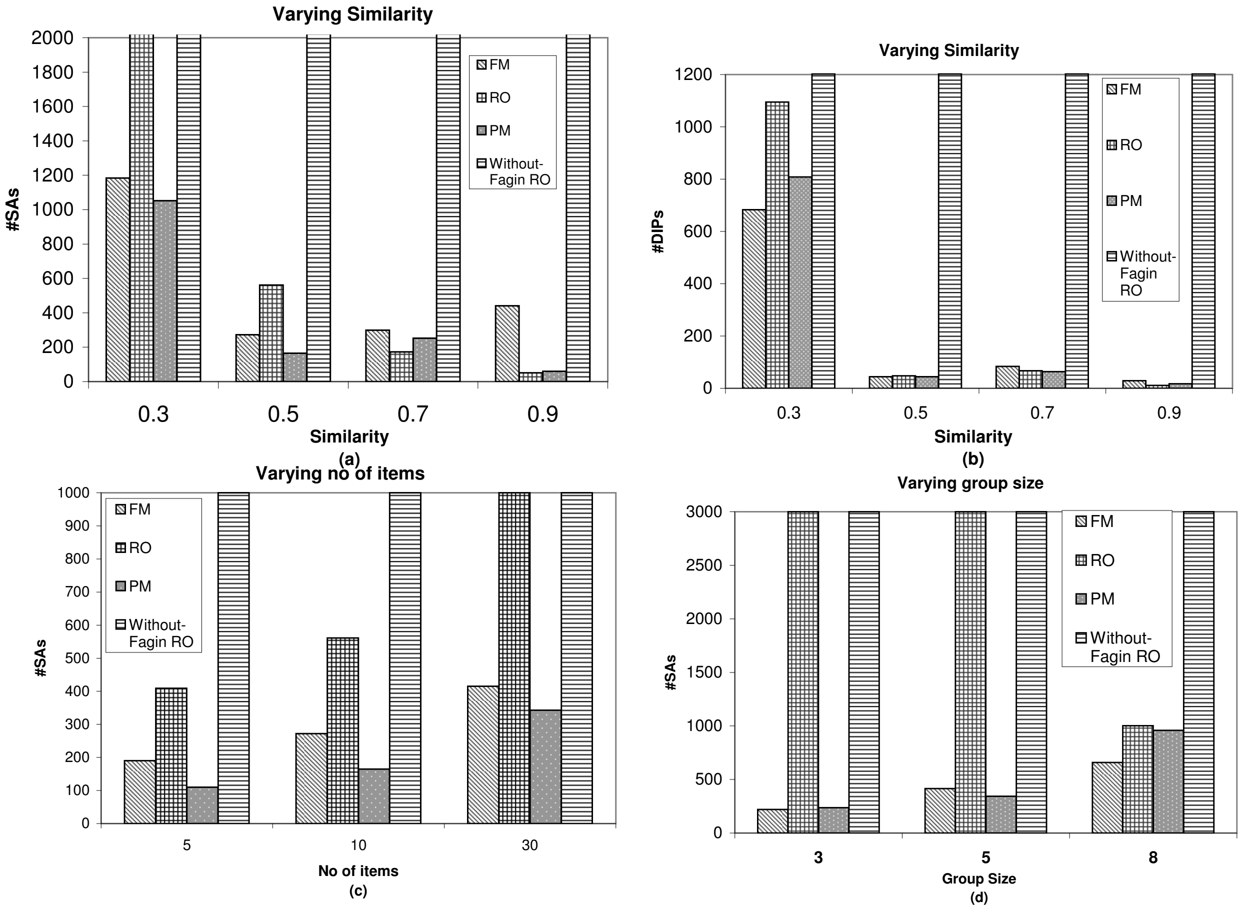


Figure 6.5. Performance Comparison among Algorithms RO, FM, PM and Without-Fagin RO. a)Measures #SA-s varying Similarity, b)Measures #DIP-s varying Similarity, c)Measures #SA-s varying k, and d)Measures #SA-s varying Group Size.

6.6.2.1 Varying Group Similarity

Figure 6.5(a)(b) illustrate the performance of RO, FM and PM with different group similarities in terms of both SAs and DIP. The group size is fixed at 5 and the number of recommended items is 10. For PM, the number of materialized lists is 3. As the group similarity increases, the effectiveness of our materialization algorithms gradually decrease. This is not surprising since the more similar the members are with each other, the more likely their agreements on the top items are close to the upper bounds that are estimated in the RO algorithms. As a result, RO can reach stopping conditions as early as PM and FM do. This observation is also corroborated by the similar numbers of DIP between RO and the other two algorithms for high similarity values. Furthermore, FM forces the system to scan unnecessarily large number of lists and results in poor performances instead. In fact, it can be easily observed from Figure 6.5(a)(b), for very high similarity, RO results in the best performances, whereas, for very low similarity, FM is the winner in most of the cases. The performance of PM can be observed to be in between. An interesting observation in this case is, for average similarity, PM results in the best performances for both SAs and DIP. This corroborates the fact that in certain cases partial materialization can be the best option.

6.6.2.2 Varying K

Figure 6.5(c) illustrates the performance comparison of RO, FM and PM with different numbers of items recommended. The group size is fixed at 5 and the group similarity is fixed at 0.5. Algorithm PM uses three materialized lists for $k = 5, 10$ and five lists for $k = 30$. As expected, the number of SAs increases with the increasing number of recommended items. For all three cases, algorithm PM out-performs both RO and FM significantly.

6.6.2.3 Varying Group Size

We examine the effect of different group sizes in Figure 6.5(d). The group similarity is fixed at 0.5 and the number of recommended items is 10. For PM, the number of materialized lists is 3. As expected, the number of SAs increases as the group size increases. When the group sizes are small and medium, both materialization algorithms significantly out-perform RO. It is counter-intuitive to see that when the group size is large, the benefit of materialization decreases. After some investigation, we discovered that when the group is large, it is easy to have a predicted rating list that can provide enough pruning power to

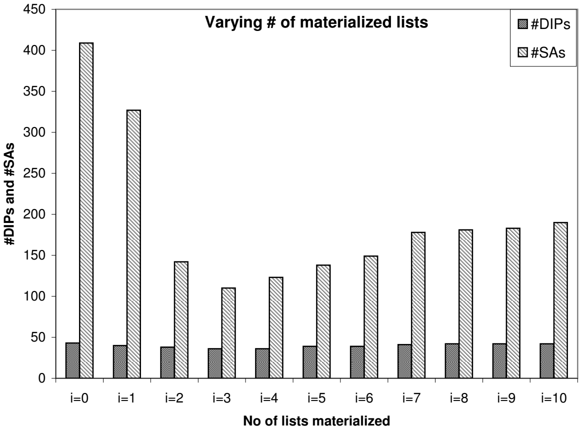


Figure 6.6. Effect of the number of $\mathcal{DL}$s.

trigger the early stopping conditions. As a result, pruning through the disagreement lists is no longer as effective.

6.6.2.4 Effect of disagreement lists in query processing

We study the impact of materializing different numbers of disagreement lists ($\mathcal{DL}$s). The group size is fixed at 5 and its similarity is fixed at 0.5, the number of recommended items is 5. We report SAs and DIP by varying the number of materialized disagreement lists. As shown in Figure 6.6, the performance is at its worst when the number of $\mathcal{DL}$s is 0, which corresponds to RO. It starts getting better as more $\mathcal{DL}$s are added and the performance is best when the number of $\mathcal{DL}$s reaches 3. Then, it starts degrading and never gets better. However, from the 4^{th} to the 10^{th} list, the number of DIP remains almost the same. By examining the 4^{th} list, we noticed that many top items in that list are not present in the final result, and, as a result, the number of SAs increases unnecessarily. We also noticed that the top items in the 4^{th} list are shared by all subsequent lists (which explains the close-to-constant performance). This situation can arise when a subset of the group dislikes the same set of movies equally.

6.6.3 Space reduction techniques and their impact on query processing

The main focus of this subsection is to analyze and compare the query processing performance of PM algorithm under space constraints. Recall that the PM algorithm is designed when a space budget is enforced and a subset of possible set of pair-wise disagreement lists can be materialized. We proposed to combine factoring and disagreement lists materialization to satisfy such hard space constraints. Here, we experimentally evaluate query processing performance attained by PM using configurations offered by these different space reduction techniques.

Summary of Results: Our first observation is PM is never worse than RO. For some groups, the best performance can be attained by using PM algorithm. In general, FM gets better with bigger group sizes. However, the difference in performance between FM and PM is not noteworthy as the group size is increased. Hence, under a space constraint, PM is an acceptable solution. Next, we observe that our proposed behavior factoring algorithm performs well in reducing space. Finally, we experimentally demonstrate that factoring is always beneficial from performance perspective since it aids to preserve more disagreement lists in a lossless way. Consequentially, Factoring followed by Knapsack based PM is better than PM-Only even when a small fraction of space is offered to materialize disagreement lists.

In these experiments, the space required by a configuration is interpreted as the total number of entries in the disagreement lists (as defined in Section 6.4.) That is because predicted rating lists being necessary, they are not affected by our space reduction strategies, factoring and partial materialization.

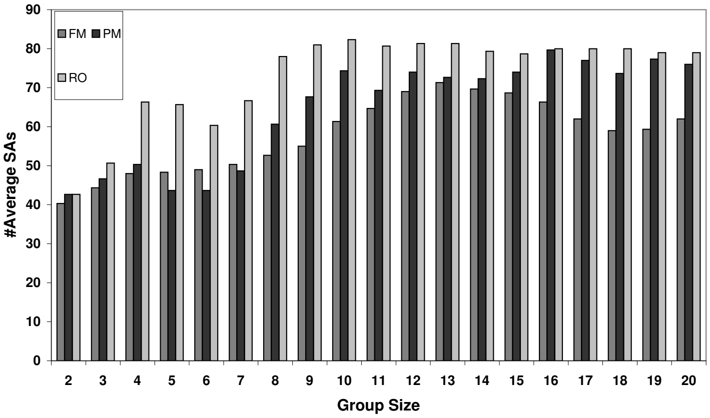


Figure 6.7. Query Processing Performance of Group Recommendation Algorithms.

6.6.3.1 Effect of partial materialization (PM-Only) on query processing

First, we perform a comparative performance study of query processing of different group recommendation algorithms (RO, FM and PM). In these experiments, we set the available space to materialize disagreement lists to 50% of the total space consumed by all possible pair-wise disagreement lists in the user base. We vary the query size from 2 to 20 (recall that a query is a group that is seeking recommendations) and measure the number of sequential accesses (SAs) required to compute top-k ($k = 30$) recommended items to the group. Each performance number of a particular query size is obtained by averaging the number of sequential accesses required to compute top-k recommendations of three different groups of that particular size. For a particular query, its size is increased by adding one random new user from the user base.

Figure 6.7 illustrates the performance comparison of different group recommendation algorithms. As expected, the average number of SAs increases with the increasing group size in general. In general, FM gets better as group size is increased. RO performs the worst among all three in all cases. For groups 5, 6 and 7, PM is the best solution. By examining group 5 in one individual run, we noticed that PM uses only 4 disagreement lists at that step, whereas, FM uses all 10 disagreement lists. These extra disagreement lists in-

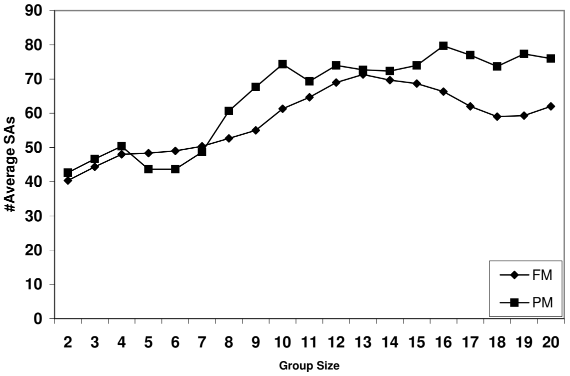


Figure 6.8. Difference in Performance between PM and FM.

cur unnecessary sorted accesses in FM. Also, PM gets better from group 4 to group 5. Our analysis reveals that PM uses only 1 disagreement list in group 4, whereas in group 5, it uses 3 new disagreement lists. We further investigate that behavior and notice that the new disagreement lists play crucial role in reaching the threshold fast during top-k computation. Consequently, the overall number of accesses drops from group 4 to group 5. This experiment also reinforces the intuition that different disagreement lists have varying impacts on performance.

Next, we study the difference in performance between PM and FM (the better one between FM and RO) in the same settings in Figure 6.8. Although, FM outperforms PM with the increase in group size, however, the difference is not significant. These two experiments corroborate our initial claims: even when full materialization is acceptable, partial materialization is important since that can attain the best performance sometime. Also, under a space constraint, PM is a satisfactory solution since its performance is reasonably close to the best solution.

6.6.3.2 Benefit of factoring algorithm in space saving

Next, we evaluate the space saving benefit of behavior factoring. We increase the size of the user base (from 3 to 20) and measure the space requirement (i.e., no of entries)

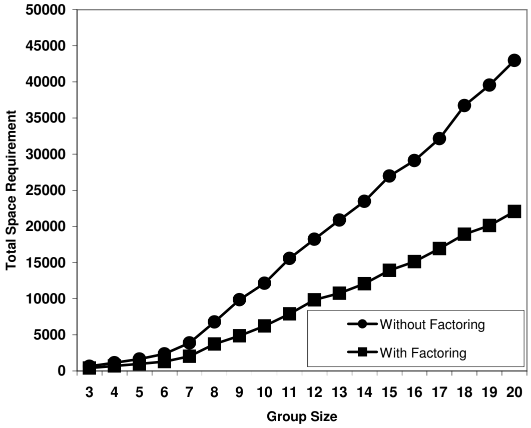


Figure 6.9. Space Savings of Factoring Algorithm for Similar Userbase.

to store all pair-wise disagreement lists for that user base *with and* without factoring. Recall that, the benefit of factoring can only be achieved for groups with size 3 and beyond. In particular, we consider two different cases: in one case, a new user is added into the existing user base at random, whereas, in other case, a new user is only selected for addition into the existing user base when it is highly similar (50% or more) to at least one existing user (henceforth referred to as Random Userbase and Similar Userbase respectively in this section.)

Figure 6.9 demonstrates the benefit of space saving for Similar Userbase. The user group of size 20 has 190 disagreement lists that contain 42978 entries (space) originally. Upon factoring, the total size of these lists is reduced to 22063 entries, thus achieving a space saving of 48.66% in a lossless manner.

Figure 6.10 demonstrates the benefit of space saving for Random Userbase. The user group of size 20 has 190 disagreement lists that consumes 42012 entries (space) originally. Upon factoring, the total size of these lists is reduced to 31023 entries, thus achieving a space saving of 26.15% in a lossless manner. It is easy to observe that space reduction is very significant for Similar Userbase, however, even for Random Userbase the reduction achieves good performance. This demonstrates that the factoring algorithm is effective and

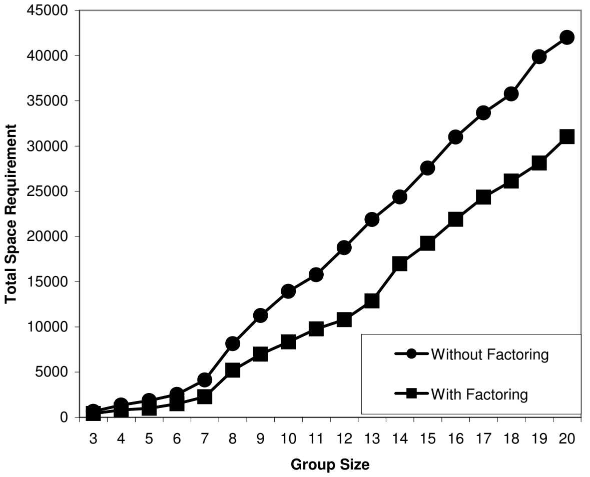


Figure 6.10. Space Savings of Factoring Algorithm for Random Userbase.

performs well in practice, thereby, reinforcing the idea that unless no two users agree on any item, factoring is always beneficial.

6.6.3.3 Impact of different space reduction strategies on query processing

Finally, we investigate different space reduction strategies and their comparative effectiveness in query processing. Recall that, given a space budget (i.e., number of entries) for materializing disagreement lists, behavior factoring may fail to reduce the original pair-wise disagreement lists of the user base to that extent. Effectiveness of space saving solely depends on similarity between users in the user base under consideration. Therefore, it may be necessary to apply techniques to drop some disagreement lists on the factored user base (refer to Algorithm in Section 6.5) to satisfy the hard space constraint. On the other hand, the hard space constraint can also be guaranteed by applying partial materialization only (refer to Section 6.5) on the original (not factored) pair-wise disagreement lists.

In these experiments, we intend to evaluate the impact of space reduction techniques from two major angles: first, given different space budgets, we evaluate the impact on query processing of factoring followed by disagreement lists materialization (henceforth referred

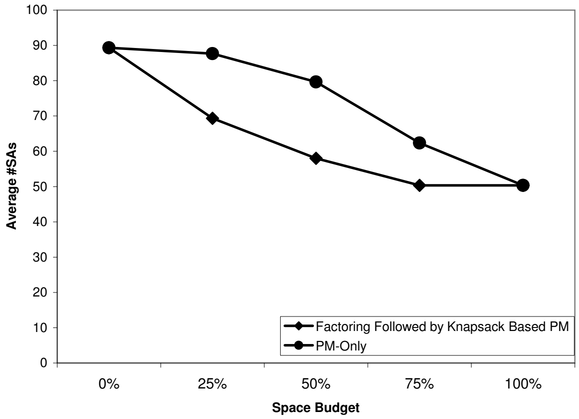


Figure 6.11. Performance of Space Saving Strategies Under Different Space Budgets.

to as `Factoring followed by Knapsack based PM` in this section) and compare that with the performance attained by applying partial materialization only (henceforth referred to as `PM-Only` in this section).

Figure 6.11 shows the comparative study of the query processing performance of `Factoring followed by Knapsack based PM` and `PM-Only` on Random Userbase. The group size is fixed at 10. Performance numbers are obtained by averaging the number of sequential accesses required to compute top-k ($k = 30$) recommendations of three different groups of size 10 chosen randomly from the Random Userbase. Space constraints are varied by 5 different numbers, in an equi-spaced manner, ranging from $(0\% - 100\%)$. Recall that this hard space constraint allows only certain amount of space ($\#$ of entries) for materializing disagreement lists. Note that 0% space means no disagreement list can be materialized (Algorithm `RO`) and 100% space budget allows all disagreement lists to be materialized (Algorithm `FM`.)

Figure 6.11 demonstrates one such case, where a higher space budget results in better performance. Therefore, performance is the worst for 0% space and the best for 100% space. It also corroborates the fact that factoring is always beneficial, since, it conserves more information in a lossless way under the same space constraint. Consequently,

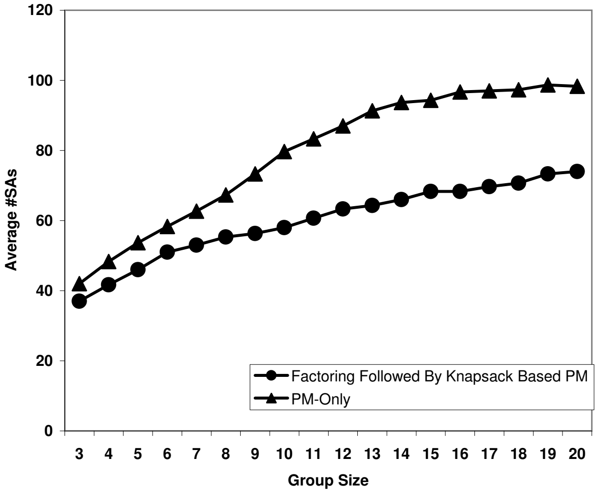


Figure 6.12. Performance of Space Saving Strategies with a Space Budget of 50%.

`Factoring followed by Knapsack based PM` performs better than `PM-Only` in all three intermediate space constraints, 25% , 50% and 75%. The most interesting observation is `Factoring followed by Knapsack based PM` attains the same performance in 75% and 100% space constraints. Recall that we use Random Userbase in this experiment which achieves a 26.15% overall space saving, i.e., factoring stores all disagreement lists in 26.15% less space, while guaranteeing the same processing performance as `FM`.

Finally, we investigate the comparative performance of the two space reduction strategies discussed above at a fixed space constraint (50%). We profile the performance of query processing by varying query size (i.e., group size from 3 to 20) there. Each performance number is presented after averaging the individual performance numbers as discussed earlier.

Figure 6.12 summarizes the result of this experiment. As expected, the average number of `SA`s increases with the increasing group size in general. However, `Factoring followed by Knapsack based PM` outperforms `PM-Only` significantly in all group sizes. This result corroborates the effectiveness of the proposed `Factoring followed by Knapsack based PM` algorithm.

6.7 Related Work

We organized our related work section into two subsections: recommendations and query processing.

6.7.1 Recommendations

Two good surveys of recommendations can be found in [98] and [99]. Briefly, the goal of a recommendation strategy is to estimate a user's rating for items he has not rated before, and return k items with highest estimated ratings. The two most popular families of recommendation strategies are item-based and collaborative filtering. The former leverages items similar to the user's previously highly rated items and the latter leverages users who share the user's interests. In this paper, we use collaborative filtering to generate individual recommendations.

A survey on group recommendations is given in [91]. It describes the two prevalent approaches: *virtual user* and *recommendation aggregation*. The former combines existing ratings of each group member to create a virtual user to whom conventional recommendation strategies are applied, whereas, the latter creates individual recommendation lists for each member and consolidates those lists to form the group's list. In this paper, we adopt the latter approach for its flexibility as described in [91].

Existing research on group recommendations mainly focuses upon group formation and evolution, privacy concerns and interfaces for supporting group recommendations. To the best of our knowledge, we have not encountered any related work that emphasizes on performance aspect of group recommendation computation, nor do they provide a theoretical and empirical study of different consensus functions, as we have done in this work. A few conducted user studies to evaluate the benefits of group recommendations. Those are summarized below.

PolyLens [93], is a group recommender extension to the MovieLens recommender system. The authors report a user study where existing MovieLens users were allowed to form groups of their preference(e.g., by inviting each other) and the system studies the impact of group behavior on the recommender system MovieLens. In order to produce group recommendations, individual groups members' recommendations were merged using the least misery model. User satisfaction was measured using following different criteria: how easy the process of creating groups was; how easy it was to add members into a group; how useful group recommendations were; and the overall satisfaction. The study concluded, among other findings, that users in a group prefer group recommendations than individual ones. This inspired our group vs individual recommendation comparison in Subsection 6.6.1.4.

In [95], the authors develop a genetic algorithm based collaborative filtering strategy to infer interactions between group members to compute the predicted rating of an item for a group. Even here, their experimental evaluation validates the quality of group recommendations and users satisfaction.

In [97], the authors distinguish between group recommendations in online communities and in non-online ones. They propose a two-phase approach, where, first a set of recommendations are generated for a group using collaborative filtering, and then items are filtered from that set in order to improve satisfaction of individual members preferences. Their experiments show that the proposed method has consistently higher precision and individual members are more satisfied.

AHP (Analytic Hierarchy Process) of multi-criteria decision making is used in [94] to model group preferences using the preferences of individuals. The authors also use a Bayesian network to model uncertainty in an individual user's preference. Their evaluation on 10 different situations assesses the high usability of their system and a comparison with both random and rule-based recommendation is also provided.

The authors in [92] develop 3 different aggregation policies of individual user models into a group model and for the purpose of biasing recommendations in a critiquing-based, case-based recommender. They conduct experiments to highlight the benefits of group recommendation using live-user preference data.

Finally, in [96], the authors use hierarchical clustering and decision trees to generate recommendations of user groups in Facebook. This work differs from ours because it focuses on recommending friends groups instead of recommending items to groups. The experiments show that a large number of groups in Facebook (73%) are accurately predicted using members's profiles.

6.7.2 Query Processing

Factoring Lists: In [104], the authors developed space-saving strategies on keyword inverted lists using shared user behavior. Their approach is based on clustering users first and then building per-cluster keyword indices instead of individual users' indices. The experiments show that such clustering saves space and that processing keyword queries on cluster-based indices has acceptable time overheads. There are two key differences between our factoring strategy and this work. First, factoring is explored in a pair-wise fashion (and not for an entire user cluster). Second, facoring does not incur additional I/O. One extension of our work is to explore factoring for a cluster of users.

Top-K Processing: The family of top-k threshold algorithms [105, 100] aim to reduce the amount of processing required to compute top-ranked answers, and have been used in the relational [106], XML [107], and many other settings. Monotonic score aggregation functions, which operate on sorted input, enable the early pruning of low-rank answers. In this work, we apply these algorithms on user's predicted rating lists and introduce pair-wise disagreement lists to improve performance.

Knapsack Problem: This combinatorial optimization problem [60, 108] arises whenever resource allocation is required between many contenders under budgetary constraints. Each resource has a cost and a value and the total allocated resource cost is restricted under a hard constraint, so it aims to allocate resources such that it gathers maximum value for a given cost. Two main variants of this problem are: *Bounded Knapsack* and *Unbounded Knapsack*. Bounded Knapsack assumes limited availability of each resource type, whereas, each resource may have infinite no of copies in Unbounded Knapsack problem. We adapt a *special* case of Bounded Knapsack known as $0/1$ *Knapsack* for modeling disagreement lists materialization problem. Each disagreement list is selected for materialization under overall space constraints (space budget) based on how much benefit it offers in speeding up query processing (value) by consuming how much space (cost).

6.8 Conclusion

Group recommendations are becoming of central importance as people engage in online social activities together. In this thesis, we define the semantics and study the efficiency of delivering recommendations to groups of users. We introduce the notion of a consensus function which aims to achieve a balance between an item's aggregate predicted rating in the group and individual member's disagreements over the item. We design and implement efficient threshold algorithms to compute group recommendations. We report on a user study conducted on the MovieLens data sets using Amazon's Mechanical Turk and a comprehensive performance study of our algorithms. We established that similarity between group members impacts both quality and efficiency.

In the absence of any information about what groups could be formed, pair-wise user disagreement lists need to be maintained in order to efficiently process recommendations to randomly formed groups. Hence, we developed two complementary space reduction strategies and studied their impact on space and time. In particular, our experiments showed that behavior factoring, a space saving strategy where items two users agree on are stored only once, achieves considerable space reduction. That strategy combined with selectively materializing disagreement lists successfully addresses applications where a space budget is enforced.

There are many avenues we would like to explore in the future. One extension to this work is to devise a query optimization algorithm which takes a group and a configuration (a set of materialized and possibly factored disagreement lists) and determines which lists to use for that group. The experiment in Section 6.6.2.4 showed that it is sometimes beneficial to merge a subset of the disagreement lists for some groups, even if they are materialized. Another avenue for improvement is the implementation of threshold sharpening as described in Section 6.3.4 for the pair-wise disagreement model. We believe this will have drastic improvements on processing recommendations.

CHAPTER 7

CONCLUSION

This dissertation focusses in designing novel online data exploration techniques from underlying large data repositories (structured data and web), that extend existing ranked retrieval based query-answering paradigm. In particular, the results in this dissertation widen the scope of existing *faceted search*, and online *recommendation systems* - two upcoming fields in online data exploration which are still in their infancy. To that end, we propose *dynamic faceted search systems* in conjunction with structured and unstructured data, based on a *navigational effort based model*. Furthermore, we augment the existing online recommender systems with novel functionalities that enables the system to recommend *composite items* to a user *interactively*, or to recommend items to a *group of users*. We investigate technical and algorithmic challenges involved in enabling efficient computation in these online problems. In this section, we briefly discuss other promising problems with future opportunities in this field.

7.1 Ongoing Work

As an ongoing work, we investigate how to enable the functionalities of the Star composite items onto the Chain composite items, and vice versa. For example, we aim at understanding how to *summarize* top-r *diversified* itineraries, or how to enable *interaction* in package construction problem. Furthermore, we wish to investigate how we can establish the connection between the interaction in composite item recommendation and an *effort based model*, where the objective is to suggest composite items based on user feedback such that the interaction completes in a *minimum number of iteration*. We also aim at exploring other composite item models; especially, we are keen to propose a solution framework for composite item recommendations that is appropriate for any arbitrary composite relationship.

Observe that, so far we have explored problems such as how to recommend *composite items*, or how to recommend top-k *individual* items to a *group of users*? A natural extension of those problems may be *how to recommend composite items to a group of users?*. Consider the interesting example scenario, where a traveling agency is required to design a vacation travel itinerary for a group of travelers (who would be traveling together), subject to some budgetary constraints (time, money, etc). Observe that, the challenges are, each traveler in the group may have different preferences, may have different budget; the itinerary recommendation system for the group must consider these constraints. Thus the corresponding optimization problem requires substantially different modeling, and substantially different solutions.

7.2 Future Work

Data is one of the primary assets to any organization, and has exhibited an extraordinary growth rate off late. An incredible amount of knowledge can be harvested by analyzing and exploring this ever-expanding large volume of data. Data exploration is still an emerging research area, and a tremendous scope of research lies here. In this section, we summarize some of my future research plans.

Analysis and Management of Structured and Unstructured Data

My immediate research interests lie in large-scale data exploration, touching upon diverse areas: web search, information retrieval, data mining, databases, recommender systems, and so on. we plan to build information systems with novel query-answering capabilities. we would like to focus on a good mix of futuristic research problems, and in building real-world systems that impart immediate higher impacts in the society. For example, we would like to investigate how to design exploratory search interfaces for large organizations which require in-depth understanding and detail modeling of the underlying complex schema, or how to leverage collaborative tagging information for designing faceted interface, or how to augment existing recommendation systems with additional functionality, considering additional contexts. In addition, most of my current research considers the case where the query is *under specified*. we would also like to study how data exploration techniques can be enabled for the *over-specified* queries that do not return any results initially.

Analysis and Management of Social Data

New research suggests that every digital comment made by users anywhere - a product review, social book-marking, tweets, blogs, activities on a social network site, e-mails can be mined for hints as to emotions and other thoughts. we intend to tap into these latent information sources and leverage that in a principled way to enhance query answering tasks, and analyze that information for future learning and opportunities. For example, we would like to investigate how tweets, blogs impact market trends, or social outcomes in advance.

Observe that, there are diverse challenges involved. The immediate challenge lies in the automatic extraction and cleaning of the real world noisy data. we intend to investigate a principled and domain independent framework to accomplish that task. The next challenge is how to leverage those information in a principled manner to enhance underlying analysis or query-answering tasks, and finally, how to design efficient scalable solutions that needs to handle the analysis of this data deluge during query time.

we am also interested in studying the security and privacy aspects of social web, that can potentially be leveraged to build dossiers on users. Social network application providers benefit from the increasing amount of personally identifiable information available on social sites through blogs, Twitter, Facebook, or simply through idle chatter and casual conversation, but, at the same time, risks of data misuse threaten the information privacy of individual users as well as the providers business model.

We are interested in studying and analyzing how online social activities can be used to provide a supportive and assistive environment to users that can foster behavioral development and further learning. Especially, we intend to study the psychological aspects of social activity, and leverage that for social and personality development, health promotion, and so on. For example, what kind of social activities can be useful to educate people of different ages, or how social and health awareness can be promoted through social activity. we intend to collaborate with experts in the area of psychology, sociology to harness the needful expertise and knowledge for that.

REFERENCES

[1] M. A. Hearst, "Clustering versus faceted categories for information exploration," *Commun. ACM*, vol. 49, no. 4, pp. 59–61, 2006.

[2] E. Stoica, M. A. Hearst, and M. Richardson, "Automating creation of hierarchical faceted metadata structures," in *HLT-NAACL*, 2007, pp. 244–251.

[3] W. Dakka and et al., "Automatic discovery of useful facet terms," 2006.

[4] S. B. Roy, H. Wang, G. Das, U. Nambiar, and M. K. Mohania, "Minimum-effort driven dynamic faceted search in structured databases," in *CIKM*, 2008, pp. 13–22.

[5] S. B. Roy, H. Wang, U. Nambiar, G. Das, and M. K. Mohania, "Dynacet: Building dynamic faceted search systems over databases," in *ICDE*, 2009, pp. 1463–1466.

[6] S. B. Roy and G. Das, "Top-k implementation techniques of minimum effort driven faceted search for databases," in *COMAD*, 2009.

[7] C. Li, N. Yan, S. B. Roy, L. Singh, and G. Das, "Facetedpedia: Dynamic generation of query-dependent faceted interfaces for wikipedia," in *World Wide Web Conf.*, 2010.

[8] N. Yan, C. Li, S. B. Roy, R. Ramegowda, and G. Das, "Facetedpedia: enabling query-dependent faceted search for wikipedia," in *CIKM*, 2010, pp. 1927–1928.

[9] S. Amer-Yahia, L. Lakshmanan, and C. Yu, "SocialScope: Enabling Information Discovery on Social Content Sites," in *CIDR*, 2009.

[10] S. Amer-Yahia, S. B. Roy, A. Chawla, G. Das, and C. Yu, "Group Recommendation: Semantics and Efficiency," in *VLDB*, 2009.

[11] S. B. Roy, S. Amer-Yahia, A. Chawla, G. Das, and C. Yu, "Constructing and Exploring Composite Items," in *SIGMOD*, 2009.

[12] G. Das, V. Hristidis, N. Kapoor, and S. Sudarshan, “Ordering the attributes of query results,” in *SIGMOD Conference*, 2006, pp. 395–406.

[13] R. Fagin, “Combining fuzzy information from multiple systems,” in *Proceedings of the Fifteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 3-5, 1996, Montreal, Canada*. ACM Press, 1996, pp. 216–226.

[14] U. Güntzer, W.-T. Balke, and W. Kießling, “Optimizing multi-feature queries for image databases,” in *VLDB*, 2000, pp. 419–428.

[15] R. Fagin, A. Lotem, and M. Naor, “Optimal aggregation algorithms for middleware,” in *PODS*, 2001.

[16] S. Chaudhuri, G. Das, V. Hristidis, and G. Weikum, “Probabilistic information retrieval approach for ranking of database query results,” *ACM Trans. Database Syst.*, vol. 31, no. 3, pp. 1134–1168, 2006.

[17] V. Hristidis and Y. Papakonstantinou, “Discover: Keyword search in relational databases,” in *VLDB*, 2002, pp. 670–681.

[18] B. Aditya, G. Bhalotia, S. Chakrabarti, A. Hulgeri, C. Nakhe, Parag, and S. Sudarshan, “Banks: Browsing and keyword searching in relational databases,” in *VLDB*, 2002, pp. 1083–1086.

[19] J. English, M. A. Hearst, R. R. Sinha, K. Swearingen, and K.-P. Yee, “Hierarchical faceted metadata in site search interfaces,” in *CHI Extended Abstracts*, 2002, pp. 628–639.

[20] V. T. Chakaravarthy, V. Pandit, S. Roy, P. Awasthi, and M. K. Mohania, “Decision trees for entity identification: approximation algorithms and hardness results,” in *PODS*, 2007, pp. 53–62.

[21] J. Gehrke, R. Ramakrishnan, and V. Ganti, “Rainforest - a framework for fast decision tree construction of large datasets,” *Data Min. Knowl. Discov.*, vol. 4, no. 2/3, pp. 127–162, 2000.

[22] T. Mitchell, *Machine Learning*. McGraw Hill, 1997.

[23] J. Shlens, “A tutorial on principal component analysis,” *Institute for Nonlinear Science*, no. UCSD, 2005.

[24] Y. Huhtala, J. Kärkkäinen, P. Porkka, and H. Toivonen, “Efficient discovery of functional and approximate dependencies using partitions,” in *ICDE*. IEEE Computer Society, 1998, pp. 392–401.

[25] S. Agrawal, S. Chaudhuri, G. Das, and A. Gionis, “Automated ranking of database query results,” in *CIDR*, 2003.

[26] W. Dakka, P. G. Ipeirotis, and K. R. Wood, “Faceted browsing over large databases of text-annotated objects,” in *ICDE*, 2007, pp. 1489–1490.

[27] ——, “Automatic construction of multifaceted browsing interfaces,” in *CIKM*, 2005, pp. 768–775.

[28] I. Martin and J. M. Jose, “Fetch: A personalised information retrieval tool,” in *RIAO*, 2004, pp. 405–419.

[29] FacetedDBLP, “Faceted dblp,” http://www.l3s.de/growbag/demonstrators.php.

[30] O. Ben-Yitzhak, N. Golbandi, N. Har’El, R. Lempel, A. Neumann, S. Ofek-Koifman, D. Sheinwald, E. J. Shekita, B. Sznajder, and S. Yogev, “Beyond basic faceted search,” in *WSDM*, 2008, pp. 33–44.

[31] K. Chakrabarti, S. Chaudhuri, and S. won Hwang, “Automatic categorization of query results,” in *SIGMOD Conference*, 2004, pp. 755–766.

[32] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Prentice Hall Series, 2003.

[33] S. Agrawal, S. Chaudhuri, and G. Das, “Dbxplorer: enabling keyword search over relational databases,” in *SIGMOD Conference*, 2002, p. 627.

[34] P. Wu, Y. Sismanis, and B. Reinwald, “Towards keyword-driven analytical processing,” in *SIGMOD Conference*, 2007, pp. 617–628.

[35] R. A. Baeza-Yates and B. A. Ribeiro-Neto, *Modern Information Retrieval*. ACM Press / Addison-Wesley, 1999.

[36] E. N. Efthimiadis, “User choices: A new yardstick for the evaluation of ranking algorithms for interactive query expansion,” *Inf. Process. Manage.*, vol. 31, no. 4, pp. 605–620, 1995.

[37] C. L. Lucchesi and S. L. Osborn, “Candidate keys for relations,” *J. Comput. Syst. Sci.*, vol. 17, no. 2, pp. 270–279, 1978.

[38] A. S. Pollitt, “The key role of classification and indexing in view-based searching,” in *IFLA*, 1997.

[39] K.-P. Yee, K. Swearingen, K. Li, and M. Hearst, “Faceted metadata for image search and browsing,” in *CHI ’03*, 2003.

[40] E. Stoica, M. A. Hearst, and M. Richardson, “Automating creation of hierarchical faceted metadata structures,” in *Proc. NAACL-HLT 2007*, 2007, pp. 244–251.

[41] W. Dakka, P. G. Ipeirotis, and K. R. Wood, “Automatic construction of multifaceted browsing interfaces,” in *CIKM*, 2005, pp. 768–775.

[42] W. Dakka and P. Ipeirotis, “Automatic extraction of useful facet hierarchies from text databases,” *ICDE*, 2008.

[43] K. A. Ross and A. Janevski, “Querying faceted databases,” in *the Second Workshop on Semantic Web and Databases*, 2004.

[44] S. B. Roy, H. Wang, G. Das, U. Nambiar, and M. Mohania, “Minimum effort driven dynamic faceted search in structured databases,” in *CIKM*, 2008.

[45] J. Diederich and W.-T. Balke, “FacetedDBLP - navigational access for digital libraries,” *Bulletin of IEEE Technical Committee on Digital Libraries*, vol. 4, Spring 2008.

[46] D. Debabrata, R. Jun, N. Megiddo, A. Ailamaki, and G. Lohman, “Dynamic faceted search for discovery-driven analysis,” in *CIKM*, 2008, pp. 3–12.

[47] O. Ben-Yitzhak, N. Golbandi, N. Har’El, R. Lempel, A. Neumann, S. Ofek-Koifman, D. Sheinwald, E. Shekita, B. Sznajder, and S. Yogev, “Beyond basic faceted search,” in *WSDM*, 2008, pp. 33–44.

[48] W. Pratt, M. A. Hearst, and L. M. Fagan, “A knowledge-based approach to organizing retrieved documents,” in *AAAI ’99/IAAI ’99*, 1999, pp. 80–85.

[49] M. Käki, “Findex: search result categories help users when document ranking fails,” in *CHI ’05*, 2005, pp. 131–140.

[50] K. Rodden, W. Basalaj, D. Sinclair, and K. Wood, “Does organisation by similarity assist image browsing?” in *CHI*, 2001, pp. 190–197.

[51] D. R. Cutting, D. R. Karger, J. O. Pedersen, and J. W. Tukey, “Scatter/gather: a cluster-based approach to browsing large document collections,” in *SIGIR ’92*, 1992, pp. 318–329.

[52] O. Zamir and O. Etzioni, “Grouper: a dynamic clustering interface to web search results,” in *WWW*, 1999.

[53] H. Bast and I. Weber, “The CompleteSearch engine: Interactive, efficient, and towards IR & DB integration,” in *CIDR*, 2007, pp. 88–95.

[54] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives, “DBpedia: A nucleus for a web of open data,” in *6th Int.l Semantic Web Conf.*, 2007.

[55] R. Agrawal and R. Srikant, “Fast algorithms for mining association rules in large databases,” in *VLDB ’94: Proceedings of the 20th International Conference on Very*

Large Data Bases. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994, pp. 487–499.

[56] D. Burdick, M. Calimlim, J. Flannick, J. Gehrke, and T. Yiu, "Mafia: A maximal frequent itemset algorithm," *IEEE Trans. Knowl. Data Eng.*, vol. 17, no. 11, pp. 1490–1504, 2005.

[57] D. Gunopulos, H. Mannila, and S. Saluja, "Discovering all most specific sentences by randomized algorithms," in *ICDT*, ser. Lecture Notes in Computer Science, F. N. Afrati and P. G. Kolaitis, Eds., vol. 1186. Springer, 1997, pp. 215–229.

[58] W. A. Gale and G. Sampson, "Good-turing frequency estimation without tears," *Journal of Quantitative Linguistics*, vol. 2, no. 3, pp. 217–237, 1995.

[59] R. Motowani and P. Raghavan, *Randomized Algorithms*. Cambridge University Press, 1995.

[60] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[61] R. Fagin and et. al., "Optimal Aggregation Algorithms for Middleware," in *PODS*, 2001.

[62] R. Fagin, A. Lotem, and M. Naor, "Optimal aggregation algorithms for middleware," *JCSS*, vol. 66, no. 4, pp. 614–656, 2003.

[63] M. Miah, G. Das, V. Hristidis, and H. Mannila, "Standing out in a crowd: Selecting attributes for maximum visibility," in *ICDE*, 2008, pp. 356–365.

[64] R. J. B. Jr., "Efficiently mining long patterns from databases," in *SIGMOD Conference*, L. M. Haas and A. Tiwary, Eds. ACM Press, 1998, pp. 85–93.

[65] X. Yang, C. M. Procopiuc, and D. Srivastava, "Summarizing relational databases," *PVLDB*, vol. 2, no. 1, pp. 634–645, 2009.

[66] C. Yu and H. V. Jagadish, "Schema summarization," in *VLDB*, U. Dayal, K.-Y. Whang, D. B. Lomet, G. Alonso, G. M. Lohman, M. L. Kersten, S. K. Cha, and Y.-K. Kim, Eds. ACM, 2006, pp. 319–330.

[67] X. Lin, Y. Yuan, Q. Zhang, and Y. Zhang, "Selecting stars: The k most representative skyline operator," in *ICDE*, 2007, pp. 86–95.

[68] E. Vee, U. Srivastava, J. Shanmugasundaram, P. Bhat, and S. Amer-Yahia, "Efficient computation of diverse query results," in *ICDE*, 2008, pp. 228–236.

[69] J. Whittaker, *Graphical Models in Applied Multivariate Statistics*. Wiley, 1990.

[70] T. M. Mitchell, *Machine Learning*. New York: McGraw-Hill, 1997.

[71] K. Chen and S. Har-Peled, "The Euclidean orienteering problem revisited," *SICOMP*, vol. 38, no. 1, pp. 385–397, 2008.

[72] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Second Edition*. The MIT Press and McGraw-Hill Book Company, 2001.

[73] S. B. Roy, S. Amer-Yahia, A. Chawla, G. Das, and C. Yu, "Constructing and exploring composite items," in *SIGMOD Conference*, A. K. Elmagarmid and D. Agrawal, Eds. ACM, 2010, pp. 843–854.

[74] K.-P. Yee, K. Swearingen, K. Li, and M. A. Hearst, "Faceted metadata for image search and browsing," in *CHI*, G. Cockton and P. Korhonen, Eds. ACM, 2003, pp. 401–408.

[75] S. Ahern, M. Naaman, R. Nair, and J. Yang, "World explorer: Visualizing aggregate data from unstructured text in geo-referenced collections," in *Proc. Joint Conference on Digital Libraries (JCDL'07)*, June 2007, pp. 1–10.

[76] D. Crandall, L. Backstrom, D. Huttenlocher, and J. Kleinberg, "Mapping the world's photos," in *Proc. 18th International World Wide Web Conference (WWW'2009)*, April 2009, pp. 761–770.

[77] F. Girardin, “Aspects of implicit and explicit human interactions with ubiquitous geographic information,” Ph.D. dissertation, Universitat Pompeu Fabra, Barcelona, Spain, 2009.

[78] A. Popescu and G. Grefenstette, “Deducing trip related information from flickr,” in *Proc. 18th International World Wide Web Conference (WWW’2009)*, April 2009, pp. 1183–1184.

[79] T. Rattenbury, N. Good, and M. Naaman, “Toward automatic extraction of event and place semantics from flickr tags,” in *Proc. 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR’07)*, July 2007, pp. 103–110.

[80] L. Ardissono, A. Goy, G. Petrone, M. Segnan, and P. Torasso, “Intrigue: Personalized recommendation of tourist attractions for desktop and handset devices,” *Applied Artificial Intelligence*, vol. 17, no. 8-9, pp. 687–714, 2003.

[81] K. Cheverst, N. Davies, K. Mitchell, A. Friday, and C. Efstratiou, “Developing a context-aware electronic tourist guide: some issues and experiences,” in *CHI ’00: Proceedings of the SIGCHI conference on Human factors in computing systems*. New York, NY, USA: ACM, 2000, pp. 17–24.

[82] S. Dunstall, M. E. T. Horn, P. Kilby, M. Krishnamoorthy, B. Owens, D. Sier, and S. Thiebaux, “An automated itinerary planning system for holiday travel,” *Information Technology and Tourism*, vol. 6, no. 3, 2004.

[83] D. Leake and J. Powell, “Mining large-scale knowledge sources for case adaptation knowledge,” in *Proc. ICCBR 2007*, 2007, pp. 209–223.

[84] C. H. Tai, D. N. Yang, L. T. Lin, and M. S. Chen, “Recommending personalized scenic itinerary with geo-tagged photos,” in *Proc. IEEE International Conference on Multimedia and Expo (ICME’2008)*, 2008, pp. 1209–1212.

[85] M. D. Choudhury, M. Feldman, S. Amer-Yahia, N. Golbandi, R. Lempel, and C. Yu, “Automatic construction of travel itineraries using social breadcrumbs,” in *HT*, M. H. Chignell and E. Toms, Eds. ACM, 2010, pp. 35–44.

[86] R. Levin, Y. Kanza, E. Safra, and Y. Sagiv, “Interactive route search in the presence of order constraints,” *PVLDB*, vol. 3, no. 1, pp. 117–128, 2010.

[87] G. Dantzig and J. Ramser, “The truck dispatching problem,” in *Operations Research*, 1959, pp. 80–91.

[88] K. Chen and S. Har-Peled, “The euclidean orienteering problem revisited,” *SIAM J. Comput.*, vol. 38, no. 1, pp. 385–397, 2008.

[89] A. Blum, S. Chawla, D. R. Karger, T. Lane, A. Meyerson, and M. Minkoff, “Approximation algorithms for orienteering and discounted-reward tsp,” *SIAM J. Comput.*, vol. 37, no. 2, pp. 653–670, 2007.

[90] C. Chekuri, N. Korula, and M. Pál, “Improved algorithms for orienteering and related problems,” in *SODA ’08: Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2008, pp. 661–670.

[91] A. Jameson and B. Smyth, *The Adaptive Web*, ser. LNCS. Springer-Verlag, 2007, vol. 4321, ch. Recommendation to Groups, p. 596.

[92] K. McCarthy, L. McGinty, and B. Smyth, “Case-based group recommendation: Compromising for success,” in *ICCBR*, ser. Lecture Notes in Computer Science, R. Weber and M. M. Richter, Eds., vol. 4626. Springer, 2007, pp. 299–313.

[93] M. O’Connor, D. Cosley, J. A. Konstan, and J. Riedl, “Polylens: A recommender system for groups of user,” in *ECSCW*, 2001, pp. 199–218.

[94] M.-H. Park, H.-S. Park, and S.-B. Cho, “Restaurant recommendation for group of people in mobile environments using probabilistic multi-criteria decision making,”

in *APCHI*, ser. Lecture Notes in Computer Science, S. Lee, H. Choo, S. Ha, and I. C. Shin, Eds., vol. 5068. Springer, 2008, pp. 114–122.

[95] Y.-L. Chen, L.-C. Cheng, and C.-N. Chuang, "A group recommendation system with consideration of interactions among group members," *Expert Syst. Appl.*, vol. 34, no. 3, pp. 2082–2090, 2008.

[96] E.-A. Baatarjav, S. Phithakkitnukoon, and R. Dantu, "Group recommendation system for facebook," in *OTM Workshops*, ser. Lecture Notes in Computer Science, R. Meersman, Z. Tari, and P. Herrero, Eds., vol. 5333. Springer, 2008, pp. 211–219.

[97] H. O. J.K. Kim, H.K. Kim and Y. Ryu, "A group recommendation system for online communities," in *International Journal of Information Management*, Oct. 2009.

[98] G. Adomavicius and A. Tuzhilin, "Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions," *IEEE Trans. Knowl. Data Eng.*, vol. 17, no. 6, 2005.

[99] J. A. Konstan, "Introduction to recommender systems," in *SIGIR*, 2007.

[100] R. Fagin, A. Lotem, and M. Naor, "Optimal aggregation algorithms for middleware," in *PODS*, 2001.

[101] S. Martello and P. Toth, *Knapsack Problems: Algorithms and Computer Implementations (Wiley-Interscience Series in Discrete Mathematics and Optimization)*. John Wiley & Sons, 1990.

[102] GroupLens at University of Minnesota, "http://www.grouplens.org/node/73."

[103] K. Jarvelin and K. Kekalainen, "Cumulated gain-based evaluation of ir techniques," *ACM TOIS*, vol. 20, no. 4, 2002.

[104] S. Amer-Yahia, M. Benedikt, L. Lakshmanan, and J. Stoyanovich, "Efficient Network-Aware Search in Collaborative Tagging Sites," in *VLDB*, 2008.

[105] R. Fagin, “Combining fuzzy information: an overview,” *SIGMOD Record*, vol. 32, no. 2, pp. 109–118, 2002.

[106] M. J. Carey and D. Kossmann, “On saying ”enough already!” in sql,” in *SIGMOD*, 1997.

[107] A. Marian, S. Amer-Yahia, N. Koudas, and D. Srivastava, “Adaptive processing of top-k queries in xml,” in *ICDE*, 2005.

[108] S. Sahni, “Approximate algorithms for the 01 knapsack problem,” in *Journal of the ACM*. ACM, 1975.

BIOGRAPHICAL STATEMENT

Senjuti Basu Roy was born in Calcutta, India. She received her B.Tech. degree in Computer Science and Engineering from the University of Calcutta, India, in 2004. She has received her M.S. in Computer Science from The University of Texas at Arlington in 2007. Her current research interests include data and social content management and exploration, information retrieval, data mining techniques on databases and social networks, algorithms etc.