Constructing and Exploring Composite Items *

Senjuti Basu Roy†, Sihem Amer-Yahia‡, Ashish Chawla†‡, Gautam Das†, Cong Yu‡
‡Yahoo! Research, †Univ. of Texas Arlington,
‡{sihem,achawla,congyu}@yahoo-inc.com, †senjuti.basuroy@mavs.uta.edu,
{gdas,achawla}@uta.edu

ABSTRACT

Nowadays, online shopping has become a daily activity. Web users purchase a variety of items ranging from books to electronics. The large supply of online products calls for sophisticated techniques to help users explore available items. We propose to build *composite items* which associate a *central item* with *a set of packages*, formed by *satellite items*, and help users explore them. For example, a user shopping for an *iPhone* (i.e., the central item) with a price budget can be presented with both the iPhone 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. We define and study the problem of 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 find diverse composite items quickly by minimizing overlap between packages presented to the user in a ranked order. We conduct an extensive set of experiments on Yahoo! Shopping[1] data sets to demonstrate the efficiency and effectiveness of our algorithms.

Categories and Subject Descriptors

H [**Information Systems**]: INFORMATION STORAGE AND RETRIEVAL—*Information Search and Retrieval*

General Terms

Algorithms, Performance

*The work of Senjuti Basu Roy and Gautam Das was supported in part by the US National Science Foundation under grants 0916277, 0845644 and 0812601, a grant from the Department of Education, and unrestricted gifts from Microsoft Research and Nokia Research.

[1]http://shopping.yahoo.com/

SIGMOD'10, June 6–11, 2010, Indianapolis, Indiana, USA.

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 2).
- We design and implement a random walk algorithm to efficiently construct all valid and maximal packages (Section 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).
- 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 (Section 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 6). Finally, we discuss related works and conclude in Sections 7 and 8, respectively.

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 $[\mathcal{C}, \mathcal{S}_1, \ldots, \mathcal{S}_n]$.

2.1 Valid and Maximal Packages

DEFINITION 1 (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 2 (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.

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

Table 1: Central Items

Also, consider the satellite items in Table 2, grouped by type for ease of exposition. There are 7 types in the table. Assume, for simplicity, that all satellite items in Table 2 are compatible with all available iPhones in Table 1. Table 3

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	... ...	...
...	... ...	...

Table 3: Examples of Valid Satellite Packages

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 2: Satellite Items and their Price

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*.

DEFINITION 3 (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 3.

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 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.

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.

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

Table 4: Two Sets of Summary Packages for Central Item *iPhone 3G/8GB*

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 4 (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. 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 5.

3. MAXIMAL PACKAGE CONSTRUCTION

Recall from Section 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) [1, 2], 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 [1]. 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 [7]), `MaxCompositeItemSet`, that computes an approximate $\mathcal{M}_c$ based on random walks.

3.1 Algorithm MaxCompositeItemSet

Algorithm 1 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 1) accomplishes the random walk procedure. It

Algorithm 1 `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

1: $\mathcal{M}_c = \{\}$
2: **repeat**
3: $p =$ `MaxCompositeItem`$(c, \mathcal{A}, b)$
4: **if** $p \notin \mathcal{M}_c$ **then**
5: $\mathcal{M}_c = \mathcal{M}_c \cup \{p\}$
6: $count(p) = 1$
7: **else**
8: $count(p) + +$;
9: **end if**
10: **until** $\{\forall p \in \mathcal{M}_c, count(p) \geq 2\}$
11: **return** $\mathcal{M}_c$;

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

1: $p = \{\}$
2: pick a random $a \in \mathcal{A}$, add a to p
3: **repeat**
4: 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) $a.\texttt{cost} + \sum_{s_i \in \mathcal{A}}(s_i.\texttt{cost}) \leq b$
5: add a to p
6: **until** {no new item can be added}
7: **return** p

Figure 1: Function `MaxCompositeItem`

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.

We illustrate this algorithm with the running iPhone example from Section 2.

EXAMPLE 1. *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 2 depicts the random walk process as selecting random paths in the package lattice. Algorithm 1 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.

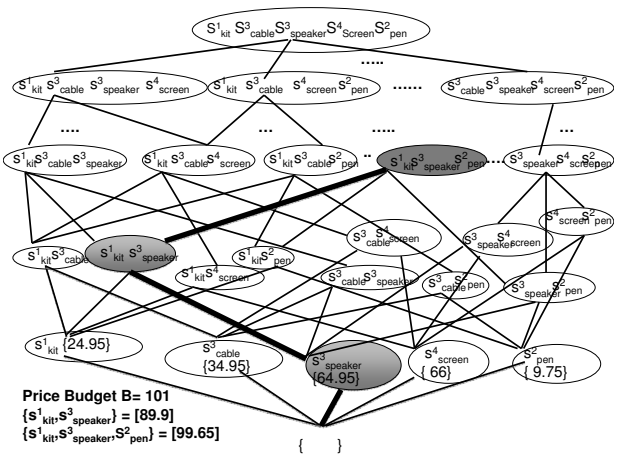


Figure 2: Random Walk on Item Lattice

3.2 Termination Condition

The termination condition used in Algorithm 1 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 [5]. 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 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 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 1, P_0 can be estimated to be 0, which means it is highly likely that all maximal packages have been discovered.

4. SUMMARIZATION

Presenting the full set of maximal packages to the user directly has two main challenges as discussed in Section 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.

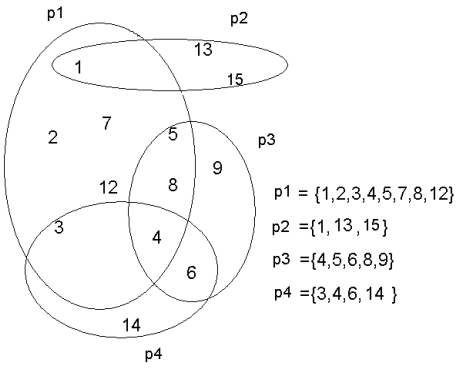


Figure 3: Example Maximal Packages to be Summarized.

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

Definition 5 (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 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 double-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.

Intuitively, the coverage of a set of packages can be computed based on the *Inclusion-Exclusion Principle* [11] (a standard technique for deriving the cardinality of the union of a set of sets) using the procedure described in Figure 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.

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

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

Figure 4: Function `ComputeCoverage`

To address this performance challenge, in Section 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 4.2, 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 3.

4.1 Greedy Summarization Algorithms with Bounded Approximation Factors

We first present the baseline `GreedySummarySet`, which is shown in Algorithm 2. 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). The algorithm stops after k packages have been chosen. Consider again the example in Figure 3: when $k = 2$, Algorithm 2 produces the summary $\{p_1, p_3\}$, and when $k = 3$, it produces the summary $\{p_1, p_3, p_4\}$.

Algorithm 2 `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 $iteration \le k$ **do**
  $p = argmax_{p \in \mathcal{M}_c}$(`ComputeCoverage`($\mathcal{I}_c \bigcup \{p\}$))
  remove p from $\mathcal{M}_c$
  add p to $\mathcal{I}_c$
  $iteration++$
end while
return $\mathcal{I}_c$

This baseline algorithm is directly adapted from a greedy approximate algorithm designed for the *Maximum k-Set Cover* problem [6], 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* prob-

lem is known to have a $(1 - 1/e)$ approximation ratio [6], therefore we have:

LEMMA 2. *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 2 can still be quite expensive in practice. We describe `FastGreedySummarySet` (Algorithm 3) 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 [11] 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 3 `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

```
1: I_c = {}
2: iteration = 1
3: while iteration ≤ k do
4:    r = −1
5:    repeat
6:       r = r + 2
7:       for p ∈ M_c do
8:          p.lower = BonferroniLower(I_c ⋃ {p}, r)
9:          p.upper = BonferroniUpper(I_c ⋃ {p}, r)
10:      end for
11:      p_1 = argmax_{p' ∈ M_c}(p'.lower)
12:      p_2 = argmax_{p' ∈ M_c, p' ≠ p_1}(p'.upper)
13:   until (p_1.lower ≥ p_2.upper)
14:   remove p_1 from M_c
15:   add p_1 to I_c
16:   iteration + +
17: end while
18: return I_c
```

The algorithm estimates the coverage using the *Bonferroni Inequalities* [11] 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 [3], since the function `ComputeCoverage` exhibits a monotonic behavior with increasing r.

Algorithm 4 `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

```
1: I_c = {}
2: i = 1
3: for a ∈ V_c do
4:    a.seenCnt = 1
5: end for
6: while i ≤ k do
7:    p = SelectRepresentative(V_c, b)
8:    if p ∉ I_c then
9:       add p to I_c
10:      for a ∈ p do
11:         a.seenCnt + +
12:      end for
13:   end if
14:   i + +
15: end while
16: return I_c ;
```

4.2 Randomized Summarization Algorithm

Both greedy algorithms described in Section 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 1). 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.

As shown in Algorithm 4, `ProbSummarySet` has the same overall structure as `MaxCompositeItemSet` (Algorithm 1), i.e., it makes similar random walks to generate a set of maximal packages. There are two main differences. First, Algorithm 4 stops as soon as k packages are generated. Second, more importantly, each random walk (Function `Select Representative` in Figure 5) invoked from within Algorithm 4 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.`seenCnt`). The algorithm then selects the next item with probability inversely proportional to its a.`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 1 and the corresponding item lattice in Figure 2 and assume that Algorithm 4 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

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 `seenCnt`s
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**
  $a.\texttt{probiblity} = \frac{1}{a.\texttt{seenCnt} \times a.\texttt{cost} \times P}$
end for
$p = \{\}$
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 5: Function `SelectRepresentative`

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 6, we found this randomized summarization algorithm to work very well in practice.

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 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:

EXAMPLE 2. *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 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 5.2 and show that it is optimal when there is only one satellite type.

5.1 Visual Effect Optimization is NP-Complete

LEMMA 3. *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.

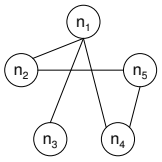
Given a graph G, we can transform it into a set of packages S in polynomial time and show that an optimal ordering of the packages with an all-zero penalty vector exists if and only if a Hamiltonian path exists for G.

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 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. □

5.2 Heuristic Visual Effect Optimization

In this section, we introduce a heuristic algorithm (Algorithm 5) 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



	S1	S2	...
p1	s¹1	s²2	
p2	s²1	s³2	
p3	s²1	s¹2	
p4	s²1	s⁴2	
p5	s¹1	s¹2	

Figure 6: Transforming a graph into packages.

Algorithm 5 `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
    $D = D_{s_1^y}(\mathcal{P})$
  end if
  $p =$ `PickBestCandidate`$(p_o, D, \mathcal{O})$
  add p to PO
  remove p from $\mathcal{P}$
end while
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 $\mathcal{P}$
end while
return PO

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.

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:

Example 3. *Given the following four packages:*
$p_1 = (s^1_{case}, s^1_{charger}, s^1_{kit}, s^1_{cable}, s^1_{speaker}, s^2_{screen}, s^1_{pen})$,
$p_2 = (s^1_{case}, s^1_{charger}, -, s^3_{cable}, -, s^1_{screen}, s^1_{pen})$,
$p_3 = (s^1_{case}, s^4_{charger}, -, s^2_{cable}, s^3_{speaker}, -, s^1_{pen})$,
$p_4 = (s^2_{case}, s^4_{charger}, -, s^2_{cable}, s^3_{speaker}, s^1_{screen}, s^1_{pen})$,

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
    end if
  end for
  if $|D| == 1$ **then**
    return $p \in D$
  end if
  if $|D| == 0$ **then**
    return random $p \in C$
  end if
end for
if $|D| > 1$ **then**
  return random $p \in D$
end if

Figure 7: Function `PickBestCandidate`

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 6, this heuristic algorithm efficiently produces package orderings with close to optimal quality. Further, we prove that when $m = 1$, Algorithm 5 does produce the optimal ordering.

Lemma 4. *Algorithm 5 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 5 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.□

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 [4]) 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.

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 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 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.

#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

Table 5: # Maximal Packages Generated

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. We compare them against three baseline algorithms:

`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 2), where the coverage of a candidate set of summary packages are computed using the *Inclusion-Exclusion Principle* [11].

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.

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 1). 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 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).

6.2.2 Summarization: Performance

Figure 8 shows the performance comparison between our

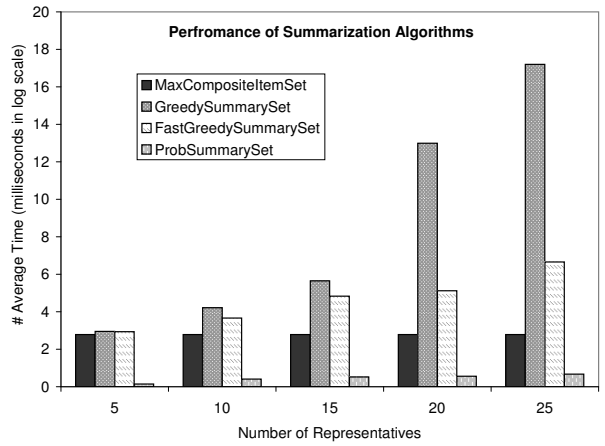


Figure 8: Summarization Algorithms Performance

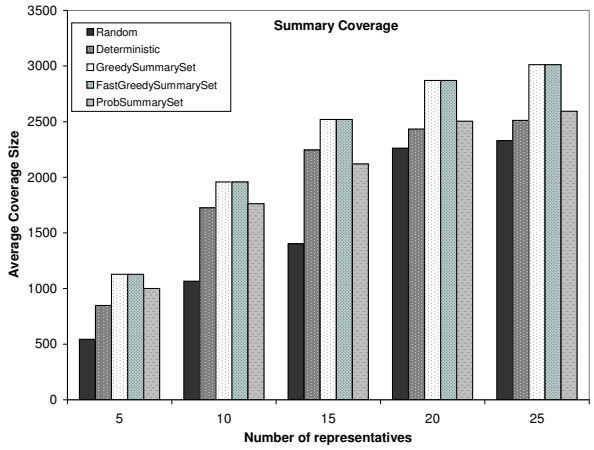


Figure 9: Summarization Algorithms Coverage

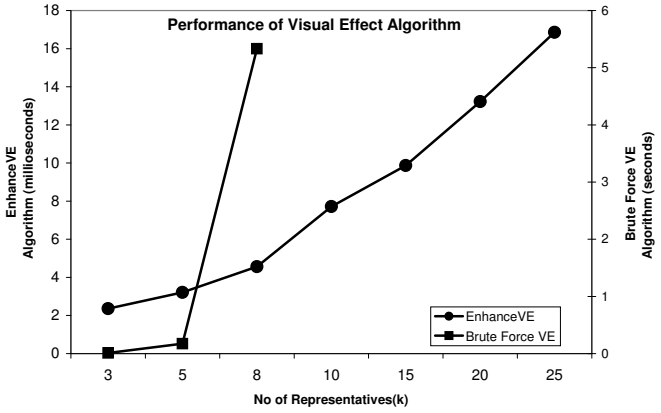


Figure 10: Performance of Visual Effect Algorithms

two proposed algorithms, `FastGreedySummarySet` and `ProbSummarySet`, 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 8, where the cost 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.

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 9, `FastGreedySummarySet` achieves exactly the same coverage as the baseline `GreedySummarySet`, which confirms our theoretical analysis in Section 4.1 that the former 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 8, we believe it is the best choice for summarization.

6.3 Visual Effect Optimization

In this section, we evaluate the quality and performance of the heuristics visual effect optimization algorithm `EnhanceVE` in Section 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.

Performance: Figure 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 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 num-

bers 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.

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

Table 6: Comparison of Penalty Vectors

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 paper 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 [7, 10] that are primarily designed for computing maximal frequent itemsets. Several other works have also investigated this problem [1, 8, 2]. 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* [5] 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* [6]. 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 [13] and XML databases [14].

Our proposed modeling of summarization bears resemblance to existing work on ranking skyline points based on dominance [9]. 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 [12]. 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* [6] can be reduced to an instance of our visual effect optimization problem as discussed in Section 5.

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.

9. REFERENCES

[1] 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*, pages 487–499, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[2] D. Burdick, M. Calimlim, J. Flannick, J. Gehrke, and T. Yiu. Mafia: A maximal frequent itemset algorithm. *IEEE Trans. Knowl. Data Eng.*, 17(11):1490–1504, 2005.

[3] R. Fagin and et. al. Optimal Aggregation Algorithms for Middleware. In *PODS*, 2001.

[4] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *JCSS*, 66(4):614–656, 2003.

[5] W. A. Gale and G. Sampson. Good-turing frequency estimation without tears. *Journal of Quantitative Linguistics*, 2(3):217–237, 1995.

[6] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[7] D. Gunopulos, H. Mannila, and S. Saluja. Discovering all most specific sentences by randomized algorithms. In F. N. Afrati and P. G. Kolaitis, editors, *ICDT*, volume 1186 of *Lecture Notes in Computer Science*, pages 215–229. Springer, 1997.

[8] R. J. B. Jr. Efficiently mining long patterns from databases. In L. M. Haas and A. Tiwary, editors, *SIGMOD Conference*, pages 85–93. ACM Press, 1998.

[9] X. Lin, Y. Yuan, Q. Zhang, and Y. Zhang. Selecting stars: The k most representative skyline operator. In *ICDE*, pages 86–95, 2007.

[10] M. Miah, G. Das, V. Hristidis, and H. Mannila. Standing out in a crowd: Selecting attributes for maximum visibility. In *ICDE*, pages 356–365, 2008.

[11] R. Motowani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[12] E. Vee, U. Srivastava, J. Shanmugasundaram, P. Bhat, and S. Amer-Yahia. Efficient computation of diverse query results. In *ICDE*, pages 228–236, 2008.

[13] X. Yang, C. M. Procopiuc, and D. Srivastava. Summarizing relational databases. *PVLDB*, 2(1):634–645, 2009.

[14] C. Yu and H. V. Jagadish. Schema summarization. In U. Dayal, K.-Y. Whang, D. B. Lomet, G. Alonso, G. M. Lohman, M. L. Kersten, S. K. Cha, and Y.-K. Kim, editors, *VLDB*, pages 319–330. ACM, 2006.