TopRecs$^+$: Pushing the Envelope on Recommender Systems

Mohammad Khabbaz, Min Xie, Laks V.S. Lakshmanan
Department of Computer Science, University of British Columbia
{mkhabbaz,minxie,laks}@cs.ubc.ca

1 Introduction

Spurred by the advances in collaborative filtering, by applications that form the core business of companies such as Amazon and Netflix, and indeed by incentives such as the famous Netflix Prize, research on recommender systems has become quite mature and sophisticated algorithms that enjoy high prediction accuracy have been developed [1]. Most of this research has been concerned with what we regard as *first generation recommender systems*. Ever since the database community got interested in recommender systems, people have begun asking questions related to functionality. This includes developing flexible recommender systems which can efficiently compute top-k items within their framework [18] and using recommender systems to design packages subject to user specified constraints [11].

Significant effort has been dedicated to improving accuracy of recommendations. Many of the recommendation algorithms, while highly accurate, have scalability issues. The number of items managed by modern information systems is growing rapidly. Therefore, scalability is one of the serious issues for future generation recommender systems. Recommendation methods try to capture personalized patterns in user feedback data by making assumptions and keeping dense summaries of data. User feedback is typically represented in the form of a sparse matrix that stores existing ratings of users on items. There are two groups of methods – model-based and memory-based [1]. Model-based methods assume there is a lower dimensional underlying parametric model that has generated the ratings matrix. These methods aim at finding optimal parameter values for the model, given the observed data. Memory-based methods, on the other hand, calculate similarities between users or items and use these similarities for aggregating existing ratings and predicting unknown ratings. Thus, any item recommendation process has two steps: (1) an off-line training phase that captures personalized profiles (either as a model or as a similarity matrix); (2) an online recommendation generation process that uses the latest up-to-date model or similarity matrix to return top-k recommendations for a user. Any approach for improving scalability of item recommendation needs to pay attention to both profile building and recommendation generation. In section 2, we show how better scalability can be achieved in both aspects for one of the most popular and practical recommendation algorithms.

In addition to efficiency and scalability, an important limitation of classical recommender systems is that they only provide recommendations consisting of single items, e.g., books or DVDs. It has been recognized several applications call for composite recommendations consisting of sets, lists or other collections. For example, in trip planning, a user is interested in suggestions for a set of places to visit, or points of interest (POI). If the recommender system only provides a ranked list of POIs, the user has to manually figure out the most suitable set of POIs, which is often non-trivial as there may be a cost to visiting each place (time, price, etc.), and the

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

user may want the overall cost of all POIs to be less than a cost budget. Furthermore, some additional package constraints such as “no more than 3 museums in a package”, “not more than two parks”, “the total distance covered in visiting all POIs in a package should be $\leq$ 10 km.” might render the manual configuration process even more tedious and time consuming. Another application which needs package recommendation is music list generation [17], where the system needs to recommend users with lists of songs called playlists, and users may have a constraint on the overall time for listening to these songs, and possibly constraints on the diversity of songs in the list.

So in these applications, there is a natural need for top-k package recommendations which can recommend users with high quality packages which satisfy all the constraints. Some so-called “third generation” travel planning web sites, such as NileGuide[1], YourTour[2], and some recent research works like [2, 3] are starting to consider certain of these features, although in a limited form.

1.1 Our Vision for the Next Generation Recommender System

We believe that the next generation recommender system should be efficient, scalable, and flexible enough to be tailed to different applications and users' customization requests such as the ability to compose packages and other collections and enforce user-specified constraints. In Figure 1, we show the architecture of our envisioned next generation recommender system that we call TopRecs$^+$.

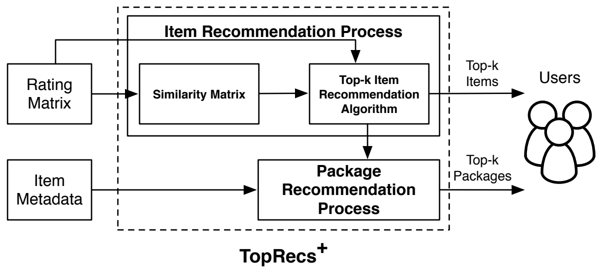
Item Recommendation Process
Rating Matrix
Similarity Matrix
Top-k Item Recommendation Algorithm
Top-k Items
Users
Item Metadata
Package Recommendation Process
Top-k Packages
TopRecs+

Figure 1: Next Generation Recommender System

In TopRecs$^+$, the recommendation engine can choose to recommend either top items or top packages depending on the application and user requests. The item recommendation engine can leverage the user item rating matrix to efficiently maintain the item-item similarity matrix, then based on this similarity matrix, an efficient and scalable top-k item recommendation algorithm can provide users with the set of k most interesting items [18]. On the other hand, based on the items generated by the item recommendation engine, combined with metadata information (such as price, type and etc.) associated with each item, the top-k package recommendation engine can return top-k packages that users will be interested in [11].

2 Top-k Item Recommendation

Predicting personalized scores of individual items for users is the core task of most recommendation algorithms. We follow item-based collaborative filtering (CF) [4], which is used widely in academia and practice [14, 5]. In CF, input data is typically represented as a sparse $n \times m$ matrix(R) of user ratings on items. An entry r_{ij} shows the existing rating of user u_i on item v_j. The main task is to predict the unknown ratings using the existing ones. Item-based CF computes and maintains an item-item similarity matrix using the existing ratings in R. Pearson correlation coefficient [12], is one of the popular choices for calculating item similarities. In item-based CF, the unknown rating $\widehat{r}_{ij}$, of u_i on v_j, is predicted by taking the weighted average of ratings of u_i on N most similar items to v_j. Equation 2 shows how existing ratings are aggregated to calculate $\widehat{r}_{ij}$ where $N(v_j, u_i)$ denotes the set of N items that are most similar to v_j and are rated by u_i.

[1]http://www.nileguide.com

[2]http://www.yourtour.com

$$\widehat{r}_{ij} = (\sum_{x=1}^{N} s_{xj} \times r_{ix}) / (\sum_{v_y \in N(v_j,u_i)} s_{yj}) \quad (2)$$

A unique challenge here in providing a score sorted list of items, is the fact that the individual scores to be aggregated to calculate $\widehat{r}_{ij}$ *come from different lists for different items*. This is because each candidate item can have a different set of N nearest neighbors among those rated by u_i. This makes it challenging to use traditional top-k algorithms. In fact, in [18], we theoretically show that adapting classic TA/NRA [13] algorithms, which are known to be instance optimal, for aggregating partial scores and returning top items leads to unpredictable performance due to this challenge. In particular, instance optimality no longer holds and there are instances where the adaptations can perform as bad as naive algorithms. Therefore, we identify the problem of discovering $N(v_j, u_i)$ for all candidate items to be the costly step in score prediction. For this purpose, we propose a novel algorithm, called the *Two Phase Algorithm (TPH)*. for finding all $N(v_j, u_i)$ in two steps.

2.1 Two Phase Algorithm

A naive approach for finding N nearest neighbors of a candidate item v_j in u_i's profile is to retrieve similarities of v_j to all (μ_i) items rated by u_i. Going over all μ_i similarity values and finding the N highest ones can be done in $O(\mu_i logN)$ for one item. t The total cost of finding nearest neighbors of all candidate items in u_i's profile is $O(m\mu_i logN)$, which can be costly in practice if μ_i is large.

In order to design a more efficient process, we propose a new global data structure, L, rather than the similarity matrix. Every column of L corresponds to one of the items. Items in each column are sorted using their similarities with respect to the item indexing the column. Thus, the j^{th} entry in the i^{th} column of L is a pair (item-id, sim), where item-id is the id of the j^{th} most similar item to the i^{th} item. The second element of the pair is the similarity between these two items.

The main intuition behind the two phase algorithm is the following. Assuming that some $N' < \mu_i$ nearest neighbors of a candidate item v_j are known, finding N nearest neighbors can be done more efficiently. This is regardless of whether N' is greater or smaller than N. Suppose we know $N' < N$ nearest neighbors of v_j, then finding the remaining ones can be done in $O(\mu_i log(N - N'))$. If $N' = N$, no further processing is needed. For $N' > N$, it again takes $O(N' logN)$ to find the N nearest neighbors.

All of the required similarity values for finding N nearest neighbors are in the μ_i columns of L that correspond to rated items. Therefore, we propose our two phase algorithm as follows. In the first phase, we choose a similarity threshold and read only those values from these columns that are above the threshold. This leads to discovering a variable number of nearest neighbors for every candidate item. Depending on the number of neighbors found for each item, in the second phase we find the exact N nearest neighbors. For those items that we have managed to find some neighbors for, the process will be more efficient as described earlier.

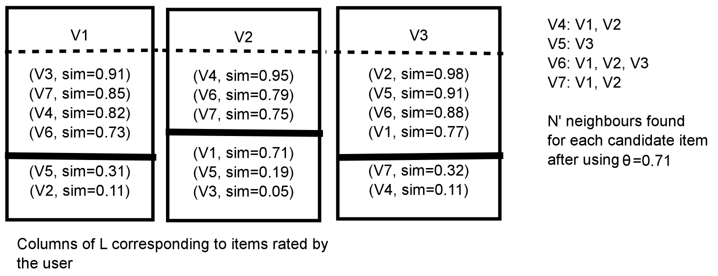

Figure 2: An example of running the two phase probe step using a prob threshold of $\theta = 0.72$ and comparing it to naive algorithms

Figure 2 illustrates the process using a threshold value $\theta = 0.72$. In the first phase all of the entries above the threshold are read as shown on the left side. In this example $\mu_i = 3$. Notice that for both cases of $N = 1$ or 2, the process can be done more efficiently for three out of four candidate items.

2.2 Optimal Threshold θ

The cost of the two phase algorithm ($C(\theta)$) can be written as the sum of three main components: (1) Cost of the first phase ($C1(\theta)$); (2) Cost of finding N nearest neighbors when $N' < N$ ($C2(\theta)$); (3) Cost of finding N nearest neighbors when $N' > N$ ($C3(\theta)$).

For instance, assuming the example in Figure 2 and $N = 2$, v_5 falls in the second category and v_6 falls in the third category. While for v_4 and v_7, we have already found their 2 nearest neighbors. Using smaller θ results in greater $C1$ and $C3$ and smaller $C2$. This is due to the fact that more similarity values will pass the filter and make it to the second phase. On the other hand, $C2$ increases if we use a larger threshold and the other two components decrease. Therefore, there is a tradeoff between $C1$ and $C3$ on one hand and $C2$ on the other. Optimal threshold value is one that minimizes the total cost of all components put together.

We perform a probabilistic cost-based analysis in [18], in order to find the optimal threshold value. In [18], we fit a Gaussian probability distribution to the similarity values in the similarity matrix. Using the cumulative density function, we calculate the probability that a particular similarity value can result in one of the N nearest neighbors of another item. Our cost function provides an upper bound on the expected cost of both phases together. We find the optimal similarity threshold that minimizes $C(\theta)$. Moreover, we theoretically prove that due to the tradeoff between cost components, $C(\theta)$ always has one and only one minimum. We refer the reader to [18] for more details, where we also empirically evaluate our algorithm. Our empirical results confirm the reliability of our theoretical probabilistic process for finding the optimal threshold value which in turn leads to a consistently efficient performance of the top-k recommender algorithm.

2.3 Updating the Similarity Matrix

Pearson correlation, Cosine and Jaccard are some of the popular examples of similarity measures used in CF. Among all of these, Pearson correlation has been applied most widely in practice. It is possible to provide guidelines in order to keep the similarity matrix updated for most of these measures. Here, we show how this is doable for Pearson correlation measure. Equation 3 shows how similarity between two items v_i and v_j is calculated using Pearson correlation coefficient. It measures the similarity between two items using only ratings of users who have rated both items (I_{ij}).

$$s(i,j) = \frac{\sum_{u \in I_{ij}} (R(u,v_i)-\bar{r}_{v_i})(R(u,v_j)-\bar{r}_{v_j})}{\sqrt{\sum_{u \in I_{ij}} (R(u,v_i)-\bar{r}_{v_i})^2 \sum_{u \in I_{ij}} (R(u,v_j)-\bar{r}_{v_j})^2}}, I_{ij} = v_i \cap v_i \tag{3}$$

Equation 4 provides the sufficient statistics that can be stored in order to be able to update the similarity matrix incrementally.

$$A_{ij} = \sum_{u \in I_{ij}} R(u,v_i)R(u,v_j), B_{ij} = \sum_{u \in I_{ij}} R(u,v_i), C_i = \bar{r}_{v_i}, D_{ij} = \sum_{u \in I_{ij}} R(u,v_i)^2, E_{ij} = |I_{ij}| \tag{4}$$

Keeping the similarity matrix updated requires storing four $m \times m$ matrices (A,B,D,E) and a vector of size m (C), containing averages of values as shown in Equation 4. When a new rating becomes available, all of the matrices in Equation 4 can be updated incrementally. Equation 5 shows how every entry of the similarity matrix can be written based on the matrices defined in Equation 4.

$$s(i,j) = \frac{A_{ij} - C_j B_{ij} - C_i B_{ji} + E_{ij} C_i C_j}{\sqrt{(D_{ij} + E_{ij} {C_i}^2 - 2C_i B_{ij})(D_{ji} + E_{ij} {C_j}^2 - 2C_j B_{ji})}} \tag{5}$$

The similarity matrix can be updated incrementally with respect to a new rating in two steps: (1) all of the entries of matrices in Equation 4 that are affected by the new rating are updated; (2) all of the entries of the similarity matrix that are affected by the changes in step 1 are updated. Since we are keeping similarities in a data structure L and columns of L are kept sorted by similarity, every column of L could be stored as a priority queue. Providing efficient strategies for performing these updates in terms of computational cost, quality of results and memory requirements is part of our ongoing work.

3 Top-*k* Package Recommendation

As mentioned in the introduction, many applications like trip planning and music list generation can benefit from having packages recommended instead of a ranked list of single items. In this section, we will discuss how the package recommendation engine can be built upon the item recommender system.

Let I be the set of all items, for each item $v \in I$, we denote the *value* of v for the current active user as $val(v)$ which can be obtained as the predicted utility or rating from the underlying item recommendation algorithms. We denote the *cost* of v as $c(v)$. The cost may correspond to time, price, etc. For a subset of items $P \subseteq I$, we define the *value* of P as $val(P) = \sum_{v \in P} val(v)$, and the *cost* of P as $c(P) = \sum_{v \in P} c(v)$. Let $\mathbf{P} = \{P \mid P \subseteq I\}$ be the set of all possible subsets of items, and given a user defined cost budget B, a subset of items $P \subseteq I$ is *feasible* if $c(P) \leq B$. We can define our top-k package recommendation problem as follows.

Definition 1: (Top-k Package Recommendation): Given a universe of items I and an underlying item recommender system for predicting values of items for the current active user, a cost budget B, find top-k packages $P_1, \ldots, P_k$ such that each P_i is feasible and $val(P) \leq val(P_i)$ for all feasible packages $P \in \mathbf{P} - \{P_1, \ldots, P_k\}$.

As discussed in [11], when $k = 1$, the top-k package recommendation problem can be viewed as a variation of the 0/1 knapsack problem [6], where we have the restriction that items can be accessed only in the non-increasing order of their value. This is because of the way recommendations are made by the underlying item recommender system. Furthermore, because solving the top-k package recommendation problem optimally is NP-hard [6], we need to consider approximate answers instead of exact ones, i.e., in Definition 1, for a package P_i in the top-k package set, instead of requiring $val(P) \leq val(P_i)$ for all feasible packages $P \in \mathbf{P} - \{P_1, \ldots, P_k\}$, we aim for $val(P) \leq \alpha \times val(P_i)$, where α is the approximation factor.

Let c_s be the access cost of retrieving the next highest-value item from the underlying item recommender system and let c_r be the access cost of obtaining the cost (time, price etc) associated with an item. It is clear that total access cost of processing n items is $n \times (c_s + c_r)$. Notice that c_s and c_r can be large compared to the cost of in-memory operations: for both accesses information may need to be transmitted through the Internet, and for the sorted access, $val(v)$ may need to be computed. So well-known algorithms for knapsack which need to access *all* items [6] may not be realistic, and instead we should minimize the total number of items accessed by the algorithm and yet ensure that high quality top-k packages are obtained.

In [11] we also show that without background knowledge about the cost distribution of items, in the worst case, we must access all items to find top-k packages. To facilitate the pruning of item accesses, we thus assume some simple background information $\mathcal{BG}$ about item costs. In [11], we assume $\mathcal{BG}$ is the minimum item cost for illustrative purposes, however, more sophisticated stats like histogram can be easily incorporated.

In the rest of this section, we will first consider an algorithm which can minimize the number of items accessed while guaranteeing the quality of the packages returned, then we will discuss a greedy algorithm which may not be optimal w.r.t. the number of items accessed but has very good empirical performance. Handling of additional constraints and highlights of the empirical results will be discussed at the end of this section.

3.1 Instance Optimal Algorithm

As discussed in the previous section, it is crucial for an algorithm to find high-quality solutions *while minimizing the number of items accessed.*

Consider the top-1 package recommendation problem. Let $S = \{v_1, ..., v_n\}$ be the set of items which have been accessed so far. It is well known from previous work [6] that a pseudo-polynomial algorithm can be utilized to get the optimal knapsack solution for S. Furthermore, as shown in [11], by utilizing the same algorithm and the background information $\mathcal{BG}$, we can also get a tight upper bound V^* on the value of the optimal solution. So we can have an iterative algorithm which retrieves a new item in each iteration, calculate the optimal solution R^o for S and stop the algorithm once $val(R^o) \geq \frac{1}{\alpha} \times V^*$.

We have shown in [11] that the above algorithm can correctly return approximate top-k package recommendations, and is *instance optimal* [13]. In particular, given any instance of the problem, and given any α-approximation algorithm A for the problem with the same background cost information $\mathcal{BG}$ and the same access constraints, A must read at least as many items as the our algorithm.

To produce top-k package recommendations, we can apply Lawler's procedure [19] to the above top-1 package recommendation algorithm. The idea is that instead of returning the α-approximation solution found in the top-1 package recommendation algorithm, we enumerate at this point all possible α-approximation solutions using Lawler's procedure. If the number of α-approximation solutions is at least k, then we can report the top-k packages found; otherwise, we continue accessing the next item. It can be shown that the above top-k package recommendation algorithm is also instance optimal.

3.2 Greedy Algorithm

Although the instance optimal algorithms presented above guarantee to return top-k packages that are α-approximations of the optimal packages, they rely on an exact algorithm for the knapsack problem, which may lead to high computational cost. To remedy this, we proposed in [11] more efficient algorithms which utilize simple greedy heuristics to form a high quality package from the accessed itemset S.

Similar to the instance optimal algorithm, this greedy algorithm will always generate a correct α-approximation to the optimal solution, however, it is not instance optimal among all α-approximation algorithms with the same constraints and background information.

3.3 Highlights of the Empirical Results

To evaluate the performance of various proposed algorithms, in [11] we tested our algorithms on four datasets: two real datasets MovieLens[3], TripAdvisor[4]; and two synthetic datasets, one with item value correlated with item cost, and another with item value and item cost uncorrelated.

We first tested the quality of packages generated by our proposed algorithms compared with optimal top-k packages generated using an offline knapsack solver. It can be verified that our proposed approximation algorithms can return top-k packages whose values are very close to the optimal solution. Furthermore, by comparing the average item value from the top-k packages generated, it can be verified that our proposed approximation algorithms often recommend packages with high average value, whereas the optimal algorithm often tries to fill the package with small cost and small value items.

For MovieLens, TripAdvisor and the uncorrelated dataset, we have verified that on average the greedy algorithm has excellent performance in terms of *both* running time and access cost. The instance optimal algorithm also has low access cost, but its running time grows very quickly with k since it needs to solve exactly many instances of knapsack, albeit restricted to the accessed items. The only dataset where both the greedy and instance optimal algorithms have a high access cost is the correlated dataset (but for this case the greedy algorithm still has good running time). The reason for this is that, for the correlated dataset, the global minimum cost corresponds only to items which also have the least value. Thus the information it provides on the unseen items is very coarse. In practice, an obvious solution to this is to obtain more precise background cost information.

[3] `http://www.movielens.org`

[4] `http://www.tripadvisor.com`

3.4 Handling Additional Package Constraints

For many applications of package recommendation, the users might have some additional constraints, e.g., for trip planning, the user may require the returned package to contain no more than 3 museums. To capture these constraints in our algorithms, we can define a Boolean *compatibility function* $\mathcal{C}$ over the packages under consideration. Given a package P, $\mathcal{C}(P) = true$ iff all constraints on items in P are satisfied. We can add a call to $\mathcal{C}$ in the proposed algorithms after each candidate package has been found. If the package fails the compatibility check, we just discard it and search for the next candidate package.

It is worth noting that many constraints studied in the previous work such as [2] and [9] are restricted classes of boolean compatibility constraints. However, depending on the application needs, for scenarios where only one specific type of constraint is considered, e.g., having one item from each of 3 predefined categories, more efficient algorithms like Rank Join [16] can be leveraged.

4 Related Work

Item-based CF was first proposed in [4]. Their solution for improving scalability is storing only N (between 10 and 30) nearest neighbors of each item regardless of user profiles. This approach improves scalability for some items but has a major drawback: for some users we may not be able to find any rated items among these N items and make predictions. Other works have tried to improve scalability by instance selection [7, 8]. The main idea is finding nearest neighbors of all items from a smaller subset of items, thus the nearest neighbors are not global nearest neighbors. It is also worth mentioning that our approach achieves scalability without deviating from the standard item-based method, while instance selection achieves scalability at the cost of sacrificing accuracy. Our approach is orthogonal to instance selection. Indeed, both methods can be combined for providing better scalability when accuracy can be traded for space needed for storing all pairwise item similarities.

For package recommendation, the closest to our work is [2], where they are interested in finding top-k tuples of entities. Examples of entities include cities, hotels and airlines, while packages are tuples of entities. A package in their framework is of fixed size, e.g., one city, one hotel and one airline, with fixed associations among the entities essentially indicating all possible valid packages. Instead, we allow for packages (composite recommendations) of variable size, subject to a budget constraint. Associations between entities can be easily captured in our framework using the notion of compatibility of sets. Another closely related work is [9] where a novel framework is proposed to automatically generate travel itineraries from online user-generated data like picture uploads. They formulate the problem of recommending travel itineraries of high quality where the travel time is under a given time budget. However, in this work, the value of each POI is determined by the number of times it is mentioned by users, whereas in our work, item value is a personalized score which comes from an underlying recommender system and unlike their work, accessing these items is constrained to be in value-sorted order. Finally, motivated by online shopping applications, [15] studies the problem of recommending "satellite items" related to a given "central item" subject to a cost budget. The resulting notion of packages is quite restricted compared to our framework, and item values are not taken into account.

5 Summary and Open Problems

In this article, we have discussed our vision for the next generation recommender system which has an efficient and scalable item recommendation engine which recommends top-k interesting items, and an efficient package recommendation engine which recommends top-k interesting packages which satisfy all the user specified constraints. While we have investigated some initial efforts in realizing such a recommender system [18, 11], much remains to be done for realizing our vision. In the following, we will point to a few interesting open problem in this direction.

- In order to handle dynamically changing user-item ratings matrix, an efficient way of maintaining the similarity matrix is critical. Preliminary ideas for doing this appear in [18]. Efficient algorithms for incremental maintenance of the similarity matrix is an important open problem.

- Instead of returning the exact top-*k* answers, for many cases, we might also be interested in providing approximate top-k recommendations with probabilistic guarantees. Some previous work like [10] has investigated probabilistic threshold algorithm, however, it cannot be directly utilized in a recommendation framework. A related point is materializing and maintaining a "key" subset of similarity entries instead of the whole similarity matrix, thus trading accuracy for speed and storage.
- For top-*k* item recommendation algorithms based on model-based methods such as matrix factorization [1] is an interesting problem.
- For top-*k* package recommendations, in [11] we have investigated the problem of recommending sets of items. Sometimes, the order of items in the recommended package might be also critical, e.g., as mentioned in [17], the order of songs in the recommended music list might be important for the users; and also for trip planning, the order of visiting different POIs can be important. Thus, recommendation of richer types of collections should be investigated.

References

[1] G. Adomavicius and A. Tuzhilin. Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. *IEEE TKDE*, 17(6):734–749, 2005.

[2] A. Angel et al. Ranking objects based on relationships and fixed associations. In *EDBT*, 2009.

[3] A. Brodsky et al. Card: A decision-guidance framework and application for recommending composite alternatives. In *ACM RecSys*, 2008.

[4] B. Sarvar et al. Item-based collaborative filtering recommendation algorithms. In *WWW*, 2001.

[5] C. Yu et al. It takes variety to make a world: Diversification in recommender systems. In *EDBT*, 2009.

[6] H. Kellerer et al. *Knapsack Problems*. Springer, 2004.

[7] K. Yu et al. Instance selection techniques for memory-based collaborative filtering. In *SDM*, 2002.

[8] K. Yu et al. Probabilistic memory-based collaborative filtering. *IEEE TKDE*, 16(1):56–69, 2004.

[9] M. De Choudhury et al. Automatic construction of travel itineraries using social breadcrumbs. In *ACM Hypertext*, 2010.

[10] M. Theobald et al. Top-k query evaluation with probabilistic guarantees. In *VLDB*, 2004.

[11] M. Xie et al. Breaking out of the box of recommendations: From items to packages. In *ACM RecSys*, 2010.

[12] P. Resnick et al. Grouplens: An open architecture for collaborative filtering of netnews. In *CSCW*, 1994.

[13] R. Fagin et al. Optimal aggregation algorithms for middleware. In *PODS*, 2001.

[14] S. Amer-Yahia et al. Group recommendation: Semantics and efficiency. In *PVLDB*, 2009.

[15] S. Basu Roy et al. Constructing and exploring composite items. In *SIGMOD*, 2010.

[16] J. Finger and N. Polyzotis. Robust and efficient algorithms for rank join evaluation. In *SIGMOD*, 2009.

[17] D. Hansen and J. Golbeck. Mixing it up recommending collections of items. In *CHI*, 2009.

[18] M. Khabbaz and L. Lakshmanan. Toprecs: Top-k algorithms for item-based collaborative filtering. In *EDBT*, 2011.

[19] E. Lawler. A procedure for computing the k best solutions to discrete optimization problems and its application to the shortest path problem. *Management Science*, 18(7):pp. 401–405, 1972.