Recommendations with Prerequisites

Aditya G. Parameswaran
Stanford University
353 Serra Mall
Stanford, CA, USA
adityagp@cs.stanford.edu

Hector Garcia-Molina
Stanford University
353 Serra Mall
Stanford, CA, USA
hector@cs.stanford.edu

ABSTRACT

We consider the problem of recommending the best set of k items when there is an inherent ordering between items, expressed as a set of prerequisites (e.g., the course 'Real Analysis' is a prerequisite of 'Complex Analysis'). Since this problem is NP-hard, we develop 3 approximate algorithms to solve this problem. We experimentally evaluate these algorithms on synthetic data.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]]: Information Search and Retrieval—*Information Filtering*

General Terms

Algorithms, Experimentation, Theory

Keywords

Recommendation Algorithms, Graph Theory

1. INTRODUCTION

Traditional recommendation systems deal with the problem of recommending items or sets of items to users by using various approaches [2, 9]. However, most of these approaches fail to take into account *prerequisites* while recommending an item: A prerequisite of an item i is another item j that must be taken or consumed (watched, read, ...) in advance of i. Thus, it makes sense to consider prerequisites when making recommendations. For example, university courses often have prerequisites. If course i cannot be taken unless course j has been completed, then it does not make sense to recommend to a student course i if j has not been taken. We could maybe recommend *both* i and j, or perhaps we could recommend some other course k that may be less desirable then i but whose prerequisites have been met.

We are interested in the problem of prerequisites in the context of our CourseRank project at Stanford University. CourseRank is a social tool developed in our InfoLab and used by students to evaluate courses and plan their academic program. CourseRank is currently used by over 9,000 Stanford students (out of 14,000); the vast majority of undergraduates use it regularly. One of the CourseRank goals is to recommend courses that are not just 'good' but also help students meet academic requirements [7]. (Academic requirements describe the constraints on the courses needed to complete a major.) In addition, we would like to take into account prerequisites, which the current production system does not take into account. Since this shortcoming is serious, we have developed a model and algorithms for recommendations constrained by prerequisites, which we describe and evaluate in this paper. Our plan is to incorporate one of our algorithms into the production system.

Although our focus is on prerequisites in an academic environment, prerequisites also arise in other recommendation contexts. Movies, for instance, often are best watched in a sequence. For example, the movie "Godfather I" should be watched before "Godfather II", and both these movies should be watched before "Godfather III". Drama TV serials tend to proceed in sequential fashion, and need to be watched in sequence. Novels tend to be sequential as well. While movies tend to have relatively few sequels, a fiction series could have several books that should be read in order.

We approach the problem of recommendations with prerequisites in the following way. We are given a set of items, each with an initial score that describes how desirable that item is for a particular user. The initial scores can be derived using traditional recommendations techniques, e.g., a movie may have a high score if people like our given user have watched that movie. We also know which items the user has consumed already. We now wish to recommend to the user a set of k desirable items, such that the set can be taken without further prerequisites. That is, the prerequisite of any item in the set has either been satisfied or is in the set itself. For example, if we wish to recommend "Lord of the Rings II" to a user as one of the k items, then we have to recommend "Lord of the Rings I" (a prequel, and therefore a prerequisite) as another one of the k items (unless the user has already watched part I).

In Section 2 we formally define the problem of set recommendations, and we show that selecting the best set is NP-hard. In Section 3 we present three heuristic algorithms that find good sets.

2. THE PROBLEM OF PREREQUISITES

We now formally define the problem of recommendation with prerequisites. We wish to recommend a set of k items from a set of items $\mathcal{V}$. We are also given a directed graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, where the vertices $v \in \mathcal{V}$ corresponds to items, and directed edges $(u, v) \in \mathcal{E}$ correspond to prerequisites, i.e.,

RecSys'09, October 22–25, 2009, New York, New York, USA.

item u needs to be taken before item v. We assume that each item in $\mathcal{V}$ has been already assigned a *score*, which corresponds to how 'good' the item is. This score could be obtained by various approaches — content-based, collaborative filtering [9, 2, 3] etc. Note that we do *not* include in $\mathcal{G}$ nor in $\mathcal{V}$ items that have already been taken or watched. That is, if item i has item j as a prerequisite, but j is already taken, then we can ignore j and its prerequisite.

Our task is to pick a set A, of size $|A| = k$, such that $score(A)$ is maximized:

$$score(A) = \sum_{a \in A} score(a) \quad (1)$$

In addition, we also have the following constraint to ensure that prerequisites are satisfied:

$$\forall u, v \in \mathcal{V} : v \in A \wedge (u, v) \in \mathcal{E} \Rightarrow u \in A \quad (2)$$

Note that these equations above inherently assume that the items being recommended are *independent* of each other, except for those that are related via set $\mathcal{E}$. That is, the scores of items (not connected through $\mathcal{E}$) do not change if we recommend them together or separately.

We define a *chain* to be a sequence of items $a_1 \rightarrow a_2 \rightarrow \ldots \rightarrow a_n$, such that there exists only the following edges involving $a_1, \ldots, a_n$: $(a_1, a_2), (a_2, a_3), \ldots, (a_{n-1}, a_n)$. Note however, that n could be 1, in which case the node has no edges either coming into or going out of it. For example, if the items we wish to recommend are movies, then nodes corresponding to movies "Godfather I", "Godfather II" and "Godfather III" would form a chain as follows: Godfather I $\rightarrow$ Godfather II $\rightarrow$ Godfather III. A graph consisting of a set of chains is called a *chain graph*. If $a_1 \rightarrow a_2 \rightarrow \ldots \rightarrow a_n$ is a chain, then $a_1 \rightarrow a_2 \rightarrow \ldots \rightarrow a_i, i \leq n$ is a *sub-chain*.

The problem of picking the best set $A, |A| = k$, satisfying prerequisites as described above is NP-Hard for directed acyclic graphs via a reduction from set cover [6]. However there is an exact but expensive dynamic programming algorithm for chain graphs but not for DAGs [6]. In this paper, we instead provide approximate but efficient algorithms that operate on chain graphs as well as DAGs.

3. THE 3 ALGORITHMS

Since the problem of recommendation with prerequisites is NP-Hard, we can only provide approximate solutions. We illustrate the three algorithms using an example, and then discuss them in more detail in subsequent sections. For ease of exposition, we will use a chain graph as our example — however, note that the pseudocode and description of the algorithms is for the more general case of DAGs.

3.1 Illustrative Example

Consider the following graph:

- $a(0.5) \rightarrow j(0.8) \rightarrow k(0.9)$
- $b(0.6) \rightarrow g(0.7)$
- $c(0.3) \rightarrow h(0.8) \rightarrow i(0.2)$
- $d(0.7)$
- $e(0.2)$

Each letter above indicates a node in the prerequisite graph, and the arrows show the prerequisites. For example, a is a prerequisite of j, which is a prerequisite of k. We include a *score* of picking each item, displayed in brackets next to the node corresponding to the item. As an example, h has a *score* of 0.8.

Our aim is to pick a set of size k, such that prerequisites are retained, and *score* of the set as defined in Eq. 1 is maximized. If say $k = 4$, then the optimal solution which does not violate prerequisites is $\{a, j, k, d\}$, with a *score* of 2.9. We leave the proof that this set is optimal as an exercise for the reader.

3.2 Definitions

We define $boundary(A)$ of a set A as the set of items in A that can be deleted, without violating the prerequisites of any other items in the set, i.e., if $x \in boundary(A)$, then there is no $y \in A$ and $x_1, x_2, \ldots, x_n$ such that there exists a sequence of edges $(x, x_1), (x_1, x_2), \ldots, (x_n, y)$ in $\mathcal{E}$. In the above example, if $A = \{a, g, c, h\}$, then $boundary(A) = \{a, g, h\}$, any of which can be discarded without affecting the prerequisites of other items in A.

We define $external(A)$ of a set A as the set of items that are not in A and can be potentially added to A without violating prerequisites, i.e., if $x \in external(A)$, then there is no $y, x_1, x_2, \ldots, x_n$ such that the edges (y, x_1), (x_1, x_2), $\ldots$, (x_n, x) exists in $\mathcal{E}$, but $y \notin A$. Note that this set also contains the items in $\mathcal{V}$ that have no edges coming into them. In the above example, if $A = \{b, j, c, h\}$, then $external(A) = \{a, i, g, d, e\}$, any of which can be added without violating any prerequisites. For example, adding k would create a new inconsistency since a is not present in A. However, all the items that have no prerequisites are present in $external(A)$, along with i, whose prerequisites c and h, are present.

3.3 Algorithm 1: Breadth-first Pickings

3.3.1 Example

We describe the execution of this algorithm on the graph in Sec. 3.1. We try to pick a set of size $k = 4$.

Step 0: We start by picking nodes whose prerequisites have been satisfied (or do not exist). Thus, the candidates are a, b, c, d, e. The best such node is d. We then add b, then g (whose prerequisite, b, is now present), and then a, until $|A| = k$. This A is $\{a, b, g, d\}$.

Step 1: Consider all nodes whose parents are in A or those who have no prerequisites. Here, we have $B = \{j, c, e\}$. In a greedy fashion, we try to see if we can replace the worst node in A (that can be removed) with the best node in B all the time maintaining prerequisites. In this case, we first examine j, the item with the highest score in B. The worst node in A is a. However, j is a child of a. We therefore pick one of $\{b, d, g\}$, instead. Since b is a prerequisite of g, we cannot pick b. Instead, we pick one of d or g, g (say). Since $score(j) > score(g)$, we replace g with j. The new $B = \{c, e, g, k\}$, and the new $A = \{a, j, b, d\}$.

Step 2: The best node in B is k (since its parent, $j \in A$). We then replace (the worst node in A that can be removed) b with k, giving us $A = \{a, j, k, d\}$, and $B = \{b, c, e\}$.

Step 3: The best node in B, b, is no longer better than any node in A, and we then terminate the algorithm, with optimal $A = \{a, j, k, d\}$

Note that in *Step 1*, if we had removed d instead of g, we would have ended up with the same A, in more iterations.

3.3.2 Description of the Algorithm

As listed in **Algorithm 1**, we initialize the set A with the best k items by picking greedily the best item from among the items whose prerequisites have been satisfied, but are not already in the set A, i.e., $external(A)$ (line 2-4).

We then greedily try to replace items from $boundary(A)$, i.e., the items that are non-essential to A, with those from

$external(A)$, those whose prerequisites have been satisfied (line 7-14). However, we make sure that we do not delete the parent of a child (line 8).

Algorithm 1 *BreadthFirstPickings*: BF Pickings

Require: $k \leftarrow$ size
Require: $G \leftarrow$ graph
1: $A \Leftarrow \emptyset$
2: **while** $size(A) < k$ **do**
3: $A \Leftarrow A \cup \{$item with largest *score* in $external(A)\}$
4: **end while**
5: $B \Leftarrow external(A)$
6: **while** there exist items in B **do**
7: pick $b \in B$ in order of decreasing *score*
8: $a \Leftarrow$ item with smallest *score* in $boundary(A)$ that is not parent of b
9: **if** a exists $\wedge$ $score(b) > score(a)$ **then**
10: $A \Leftarrow A - \{a\} \cup \{b\}$
11: $B \Leftarrow external(A)$
12: **else**
13: remove b from B
14: **end if**
15: **end while**
16: **return** A

3.4 Algorithm 2: Greedy-value pickings

We use a *max-priority-queue* for this algorithm, and insert sets of items into the queue. The max-priority-queue is sorted on the average *score* of the items in the sets, and on querying returns the set with the largest average score.

3.4.1 Example

We describe the execution of this algorithm on the graph in Sec. 3.1. For every chain in the above graph, we insert all sub-chains as sets into the max-priority-queue. For example, for chain $a \rightarrow j \rightarrow k$, we insert into the queue the following sets: $\{a\}, \{a, j\}, \{a, j, k\}$, which have average *score* $0.5, (0.5 + 0.8)/2 = 0.65, (0.5 + 0.8 + 0.9)/3 = 0.73$.

We keep popping sets with the maximum average *score* from the queue, see if the number of items in the set is greater than the remaining capacity that we can accommodate. If so, we discard it, if not, we add the set to A. In this case, we pop $B = \{a, j, k\}$ first, whose average *score* is 0.73, and whose size is 3. Let k' denote the current size of A, $k' = 0$. Since $k' + size(B) \leq 4$, we let $A \Leftarrow A \cup B$.

We then update the average *score* and size of all sets in the queue that have a nonzero intersection with chain $a \rightarrow j \rightarrow k$, assuming that the set $\{a, j, k\}$ has been picked. For example, the average score of $a \rightarrow j$ is now set to 0 (since $\{a, j\}$ is already in A). If $a \rightarrow j \rightarrow k \rightarrow y$, then the average score of $\{a, j, k, y\}$ would be set to $score(y)/1$, and size = 1 (since a, j, k have been picked).

Now $k - k' = 1$, so only sets of size 1 can be picked. Once again, in this case, $B = \{d\}$ with average score = 0.7, is added to A. Thus $A = \{a, j, k, d\}$, the optimal set.

3.4.2 Description of the Algorithm

We list the pseudocode for in **Algorithm 2**. We insert a set corresponding to each node in the graph $\mathcal{G}$ into the queue (line 3-7). This set contains the given node v, and all nodes a such that there is a path from a to v. These sets in the queue are sorted on average *score*, i.e., the sum of *score* of the items in the set, divided by the size of the set.

Now, as long as we have not picked enough items in A, we keep picking items by popping sets from the queue (line 8-9). If the popped set is small enough to be accommodated in A (line 10), we add it to A (line 11), and update the values of other sets that have a non-zero intersection with the set currently added to A, in two steps: Firstly, the number of items is reduced by the number of new items added to A that are also present in the set (line 14). Additionally, since those items no longer count towards the average score of the set, the *value* of the set is appropriately changed (line 15-19).

Algorithm 2 *GreedyValue*: GV Pickings

Require: $k \leftarrow$ size
Require: $G \leftarrow$ graph
Require: $Q \leftarrow$ max-priority-queue
1: $A \Leftarrow \emptyset$
2: $Q \Leftarrow \emptyset$
3: **for all** items $i \in G$ **do**
4: $c \Leftarrow \{i\}$
5: $c \Leftarrow c \cup$ prerequisites of i
6: insert c into Q with $size(c) =$ no. of items in c; $value(c) = \sum_{a \in c} score(a)/size(c)$
7: **end for**
8: **while** $size(A) < k \wedge Q \neq \emptyset$ **do**
9: $m \Leftarrow pop(Q)$ /* m has highest *value* in Q*/
10: **if** $size(m) \leq k - size(A)$ **then**
11: $A \Leftarrow A \cup m$
12: **for all** sets $c \in Q$ where $c \cap m \neq \emptyset$ **do**
13: sum $\Leftarrow \sum_{a \in (c-A)} score(a)$
14: $size(c) \Leftarrow$ no. of items in $c - A$
15: **if** $size(c) \neq 0$ **then**
16: $value(c) \Leftarrow$ sum$/size(c)$
17: **else**
18: $value(c) = 0$
19: **end if**
20: **end for**
21: **end if**
22: **end while**
23: **return** A

3.5 Algorithm 3: Top-down pickings

3.5.1 Example

Here we sort all nodes in decreasing order of *score*, and initially let A be the top-k, in this case (say): $\{j, k, g, h\}$. We now try to add the prerequisites of these items, starting from the item with the highest score. The set of items already considered is B, which is currently empty.

The best item in A is k, with a *score* of 0.9. Item k needs the set $C = \{a, j\}$. Since a is missing in A, we add a, and delete the node with the worst score from the *boundary* of A, but that which has not already been considered (i.e., is not in B), in this case, g. Thus we now have $A = \{a, j, k, h\}$. We keep track of the nodes already considered so far in B, which is now $\{k\}$.

Next, we try to see if j's prerequisites are present in A. They (i.e., $\{a\}$) already are. The set B now becomes $\{k, j\}$.

The next item from $A - B$ is h. Now, we try to add h's prerequisites. Deleting another node from the *boundary* of A (which contains only k) cannot be done since k has already been considered (i.e., is present in B) — we keep this constraint because we do not want worse items to override better ones. We instead try to replace h with a node that does not need any new prerequisites. Here h is replaced by one of $\{b, c, d, e\}$, in this case, d, which has the highest *score*. Set A now becomes $\{a, j, k, d\}$. Set B now becomes $\{j, k, h\}$.

We now pick the next best item from $A - B$ to check if its prerequisites are present. This item is d, whose prerequisites are present, so we do not add or delete any items from A. The set B now becomes $\{d, j, k, h\}$. Next, a is picked, and once again, there is no change to A.

Thus we once again get the optimal solution, $A = \{a, j, k, d\}$

3.5.2 Description of the Algorithm

In **Algorithm 3** we start off with the best set of items of size k(line 1), and try to incrementally add prerequisites. We keep track of items that we have already added prerequisites for in B (line 6), and never let such items be deleted.

We pick the items in order of decreasing scores from $A-B$, i.e., the items that have not been examined already (line 4). We then check if the prerequisites of the item are already present in A (line 7), if so, we examine the next item.

If there are still some s prerequisites required (line 8), we replace items from A if possible (line 11-14,18). These items are picked from $boundary(A)$, but should not be present in the items already considered B (line 12-13).

If sufficient items cannot be found we replace the item under consideration with an item from $external(A)$ (line 16), i.e., those items that can be potentially added because their prerequisites are present.

Algorithm 3 *TopDownPickings*: Top-Down Pickings

Require: $k \leftarrow$ size
Require: $G \leftarrow$ graph
1: $A \Leftarrow$ best set of size k
2: $B \Leftarrow \emptyset$
3: **while** there exists items $\in A - B$ **do**
4: $a \Leftarrow$ item with largest *score* in $A - B$
5: $C \Leftarrow$ prerequisites of a
6: $B \Leftarrow B \cup \{a\}$
7: if $(C - A == \emptyset)$ continue
8: $s \Leftarrow size(C - A)$ /* no. of missing prereqs. */
9: $A' \Leftarrow A$
10: $R \Leftarrow \emptyset$ /* deletions from A */
11: **while** $size(R) < s \wedge (boundary(A') - B) \neq \emptyset$ **do**
12: $a \Leftarrow$ item with smallest *score* in $boundary(A') - B$
13: if a exists, $R \Leftarrow R \cup a$; $A' \Leftarrow A' - a$
14: **end while**
15: **if** $size(R) < s$ **then**
16: replace a in A with item with largest score from $external(A)$
17: **else**
18: $A \Leftarrow (A - R) \cup C$
19: **end if**
20: **end while**
21: **return** A

4. RELATED WORK

We are not aware of any prior work in the area of set recommendations that take prerequisites into account.

However, there is a large body of work on traditional recommendation systems, aimed at coming up with a single 'score' for each item, combining approaches that look at using ratings given by other 'similar' users [9], other 'similar' items that the user liked [8], and other approaches [3]. All of these techniques could be used in generation of the *score* function that we use as a black box, therefore our work builds on top of other recommender systems work.

The body of work on Top-N recommendation systems [4] solve a different problem. Their aim is: given a user X item matrix of scores, and given the set of items that a given user has consumed, recommend an ordered set of up to N items that the user has not consumed. In this case there is no inherent ordering of items that needs to be respected when recommending N items, which is the case in our problem.

Ziegler et. al. [10] consider the case of recommending lists of items taking into account diversity among items in the list. Prerequisites are not considered; additionally, our algorithms can be generalized to handle the case of complex scoring functions where the score of a set is not just the sum of scores of the items contained in the set (See [6]).

Some of the recommendation questions we pose can be written in RQL (Recommendation Query Language) [1], or expressed as constraints [5], however, our aim in this paper is to consider efficient algorithms that solve those recommendation questions, and not posing those questions themselves.

5. CONCLUSIONS

In this paper, we studied how prerequisites affect the problem of recommendations. We focused on the problem of recommending a set of items with high score, while satisfying prerequisites. We proved that this problem is NP-Hard, and suggested 3 approximate algorithms to solve this problem.

For the three algorithms that we described above we analyzed *termination, worst case complexity,* and *worst case performance bounds* in the extended technical report [6].

In [6], for a synthetic chain graph dataset, we compared the three algorithms with the best set that could be returned without regard to prerequisites. We found that the greedy value pickings algorithm consistently performs better than the other two algorithms, and performs even better if we increase the number of items picked relative to the number of chains. However, this algorithm may be more expensive computationally than the other two.

We also found that there are cases where the breadth first pickings algorithm does better than the top down pickings algorithm, specifically when the number of items is small relative to the number of chains. When the number of items is large relative to the number of chains, the top down pickings algorithm tends to do better.

6. REFERENCES

[1] G. Adomavicius, A. Tuzhilin, and R. Zheng. Rql: A Query Language for Recommender Systems.

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

[3] R. Burke. Hybrid recommender systems: Survey and experiments. *User Modeling and User-Adapted Interaction*, 12(4):331–370, 2002.

[4] M. Deshpande and G. Karypis. Item-based top-n recommendation algorithms. *ACM Trans. Inf. Syst.*, 22(1):143–177, 2004.

[5] A. Felfernig and R. Burke. Constraint-based recommender systems: technologies and research issues. In *EC '08*.

[6] A. Parameswaran and H. Garcia-Molina. Evaluating and combining recommendations with prerequisites. http://ilpubs.stanford.edu:8090/939/.

[7] A. Parameswaran, P. Venetis, and H. Garcia-Molina. Recommendation systems with complex constraints: A courserank perspective. http://ilpubs.stanford.edu:8090/909/.

[8] M. Pazzani and D. Billsus. Content-based recommendation systems. In *The Adaptive Web*, pages 325–341, 2007.

[9] B. Sarwar, G. Karypis, J. Konstan, and J. Reidl. Item-based collaborative filtering recommendation algorithms. In *WWW '01*.

[10] C.-N. Ziegler, S. M. McNee, J. A. Konstan, and G. Lausen. Improving recommendation lists through topic diversification. In *WWW '05*.