Recommendations with Prerequisites

Aditya Parameswaran  
Hector Garcia-Molina

May 11, 2009

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 movie ‘Godfather I’ is a prerequisite of ‘Godfather II’). Since this problem is NP-hard, we develop 3 approximate algorithms to solve this problem. We derive worst-case bounds for these algorithms, and experimentally evaluate these algorithms on synthetic data. We also develop an algorithm to combine solutions in order to generate even better solutions.

1 Introduction

Traditional recommendation systems deal with the problem of recommending items or sets of items to users by using various approaches [2, 10]. 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 [8]. (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”. The problem is even more acute when it comes to television serials and novels. TV serials, especially those of the drama genre, 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 as many as 30 books that should be read in order.

There are at least two ways to approach the problem of recommendations with prerequisites:

• Ranking. 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. Next we compute new scores, based on the old scores, the prerequisite constraints, and knowledge of what items the user has already taken or watched. The idea is that an item’s “desirability” score can increase or decrease
depending on the prerequisites, e.g., an item that has many unfulfilled prerequisites is less desirable, especially if those prerequisites have low initial scores. Note that a high score does not guarantee that prerequisites have been met. That is, the user still needs to check if he can take or watch a highly recommended item.

- **Set Recommendation.** In the second case, we again have a set of items with initial scores, and knowledge of what items have been taken. 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). As discussed later, we can also return several sets of \( k \) items, each representing a different “package” recommendation.

In this paper we only study set recommendations. Although we have not yet carefully compared both options, initially set recommendations seem more attractive and easier for the user to interpret. If a user wishes to take or watch a single item, then a set recommendation with \( k = 1 \) yields the best item that can currently be taken. (As stated earlier, the top ranked item in Case I may not be yet watchable.) If the user is planning ahead and wants to take \( k \) courses or watch \( k \) episodes, set recommendations provide one or more “package” recommendations that make sense as a unit.

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, and we analyze their worst-case bounds in Section 4, and worst-case complexity in Section 5. It is actually possible to take sets recommended by different algorithms and combine them to obtain a new set that is even better. The algorithm for such fusion of sets is described in Section 6. Finally, in Section 7 we experimentally evaluate the algorithms and identify the one that performs the best in the average case.

### 2 The Problem of Prerequisites

We now formally define the problem of recommendation with prerequisites. (We will briefly discuss extensions to the problem in Sec. 3.6). We wish to recommend a set of \( k \) items from a set of items \( V \). We are also given a directed graph \( G(V,E) \), where the vertices \( v \in V \) corresponds to items, and directed edges \((u,v) \in E\) correspond to prerequisites, i.e., item \( u \) needs to be taken before item \( v \). We assume that each item in \( V \) 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 \([10, 2, 3]\) etc. Note that we do not include in \( G \) nor in \( V \) 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 \( \text{score}(A) \) is maximized:

\[
\text{score}(A) = \sum_{a \in A} \text{score}(a)
\]

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

\[
\forall u, v \in V : v \in A \land (u,v) \in E \Rightarrow u \in A
\]

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 \( E \). That is, the scores of items (not connected through \( E \)) do not change if we recommend them together or separately.

We assume that the directed graph \( G \) consists of a forest \( C \) of chains, \( C_i \), where each \( C_i \) consists of items in sequence. A chain consists 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 → Godfather II → Godfather III. On the other hand, the movie “Shawshank Redemption” would form a singleton node with no edges either going in or coming out.

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, while \( a_1 \rightarrow a_2 \rightarrow \ldots \rightarrow a_n \rightarrow \ldots \rightarrow a_{n+m}, m \geq 0 \) is a super-chain.

We discuss extensions to this model in Sec. 3.6.

NP-Hardness

The problem of picking the best set \( A, |A| = k \), satisfying prerequisites as described above is NP-Hard. We prove hardness via a reduction from the 0-1 Knapsack problem.

Recall that the 0-1 Knapsack problem is the following. Consider a set of items \( 1, 2, \ldots, n \). Each item \( i \) has a certain (integer) weight \( w_i \) and value \( v_i \). We wish to pick a subset of these items such that the total weight does not exceed \( w \), and \( \sum v_i \) is greater than a certain value \( v \).

We transform an instance of the knapsack problem into that of the recommendations with prerequisites problem in the following fashion: For each item \( i \) in the knapsack problem, we create a chain \( C_i \), with \( w_i \) items, where each item has score = 0 except the last item, which has score = \( v_i \). The value \( k \) is set as \( w \).

It is clear that a solution of the prerequisite problem yields a solution for the knapsack problem: if the prerequisite problem picks some complete chains \( C_i \) and some incomplete chains (portions of \( C_i \)), then the solution to the knapsack problem corresponds to the complete chains that are chosen in the prerequisite problem. Conversely, every solution of the knapsack problem is a solution of the prerequisite problem: The items \( i \) picked in the knapsack problem are nothing but the complete chains \( C_i \) that are chosen in the prerequisite problem. We can embellish this set of items with incomplete chains such that a total of \( k \) items are chosen. Thus, there is a reduction from the knapsack problem to the prerequisite problem, proving that the prerequisite problem is NP-Hard.

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.

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 ∈ boundary(A), then there is no y ∈ A and x1, x2, ..., xn such that there exists a sequence of edges (x, x1), (x1, x2), ..., (xn, y) in 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 ∈ external(A), then there is no y, x1, x2, ..., xn such that the edges (y, x1), (x1, x2), ..., (xn, x) exists in E, but y ∉ A. Note that this set also contains the items in V that have no edges coming into them. In the above example, if A = {b, j, c, h}, then external(A) = {a, 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 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 ∈ 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).

Note that at each iteration (line 6-15), we either increase the score of A, or we delete an item from B. Since, beyond a point, the score cannot grow, and since B is finite, we are guaranteed termination.

Also note that at most k chains are considered completely — these are the chains with the best k starting values.

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
BreadthFirstPickings: BF Pickings

Require: \( k \leftarrow \text{size} \)

Require: \( G \leftarrow \text{graph} \)

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

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 \( (0.5 + 0.8)/2 = 0.65, (0.5 + 0.8 + 0.9)/3 = 0.73 \).

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 add the set to \( A \), if not, we discard it. 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 \( k' + \text{size}(B) \leq 4 \), we let \( A \leftarrow A \cup B \).

We then update the average score and size of all sets in the queue that correspond to super-chains and sub-chains of 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 \( \text{score}(y)/1 \), 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 use a max-priority queue for this algorithm. We insert all sub-chains present in graph \( G \) (as sets) into the queue (line 3-7), sorted on average score, i.e., the sum of score of the items in the chain, divided by the size of the chain. On performing \textbf{pop} on the queue, the item with the largest average score is removed from the queue.

Now, as long as we have not picked enough items in \( A \), we keep picking chains 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 correspond to super-chains (line 12-16) and sub-chains (line 17-19). Super-chains will now get truncated given that the set corresponding to the current chain has been picked. Sub-chains can no longer be picked, since the current chain has been picked.

The algorithm has to terminate because the number of items (sets) in the max-priority queue is bounded by (number of items in each chain \( \times \) number of chains).
**Algorithm 2 GreedyValue: GV Pickings**

Require: \( k \leftarrow \text{size} \)

Require: \( G \leftarrow \text{graph} \)

Require: \( Q \leftarrow \text{max-priority-queue} \)

1: \( A \leftarrow \emptyset \)
2: \( Q \leftarrow \emptyset \)
3: for all chains \( C_i \in G \) do
4: for all sub-chains \( c \) of \( C_i \) do
5: insert \( c \) into \( Q \) with \( \text{size}(c) = \) no. of items in \( c \); \( \text{value}(c) = \sum_{a \in c} \text{score}(a)/\text{size}(c) \)
6: end for
7: end for
8: while \( \text{size}(A) < k \land Q \neq \emptyset \) do
9: \( m \leftarrow \text{pop}(Q) \) /* \( m \) has highest value in \( Q \) */
10: if \( \text{size}(m) \leq k - \text{size}(A) \) then
11: \( A \leftarrow A \cup m \)
12: for all super-chains \( c \in Q \) where \( c \) contains \( m \) do
13: \( \text{old} \leftarrow \text{size}(c) \)
14: \( \text{size}(c) \leftarrow \text{size}(c) - \text{size}(m) \)
15: \( \text{value}(c) \leftarrow (\text{value}(c) \times \text{old} - \text{value}(m))/\text{size}(c) \)
16: end for
17: for all sub-chains \( c \in Q \) where \( m \) contains \( c \) do
18: remove \( c \) from \( Q \)
19: end for
20: end if
21: end while
22: return \( A \)

3.5 Algorithm 3: Top-down pickings

3.5.1 Example

Here we sort all nodes in decreasing order of \( \text{score} \), and initially let \( A \) be the top-\( k \), in this case \( \{j, k, g, h\} \). 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 \( \text{score} \) 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 \( \text{score} \). 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 \( \text{boundary}(A) \), 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 \( \text{external}(A) \) (line 16), i.e., those items that can be potentially added because their prerequisites are present. Note that such an item can always be found, just by picking the first unpicked item in the current chain. For example, if we wish to replace \( k \), and \( a \) is present in \( A \), but \( j \) is not present in \( A \), then we can replace \( k \) with \( j \) without violating prerequisites.

We are guaranteed termination, because there are a finite number of items, and every item that is considered is added to \( B \), and an item that is considered cannot be re-considered.

**Algorithm 3 TopDownPickings: Top-Down Pickings**

Require: \( k \leftarrow \text{size} \)

Require: \( G \leftarrow \text{graph} \)

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

### 3.6 Extensions

Note that all algorithms return a set of size \( k \) assuming one exists. If not, the algorithms return the entire set of items.

**Directed Acyclic Graphs:** The algorithms described above can be adapted to the case when the graph \( G \) is a forest of trees, or more generally, for directed acyclic graphs.

For the case of a forest of directed acyclic graphs, Algorithm 1 and Algorithm 3 are easily adapted. The definitions of \( \text{external} \) and \( \text{boundary} \) can be modified to work on sets that are actually sub-trees.

However, for a forest of trees, Algorithm 2 would then require all possible sub-trees (exponential in size) to be inserted into the max-priority queue. To speed up the algorithm, instead of inserting all possible sub-trees, we can instead insert just paths from every node to the root of the sub-tree. The max-priority queue is sorted on the average score for each such path. Thus, in this case, the number of inserts into the
max-priority queue is simply equal to total number of items. When a path is ‘picked’, we need to adjust the score of every path that contains a portion of this path as a sub-path, or a super-path, similar to the case with chains.

On a DAG, Algorithm 2, for every item, instead of just inserting the path to one root, we insert all paths to any root, as a DAG itself. Thus, each item is inserted along with all its prerequisites in the DAG (which may be a chain, a path or a sub-tree). Even though the number of entries in the max-priority queue is only linear in the size of the number of items, updates are likely to be expensive (because any node which has a path to a root that has a nonzero intersection with the selected set would have to have the value of its set changed), unless \( k \) is small.

**Multiple recommended sets:** One trivial way to return multiple ‘package’ recommended sets is to return the outcomes of the three algorithms (if different). However, we can also use Lawler’s algorithm [6] to return multiple recommended sets that do not violate prerequisites, ranked in order of decreasing score.

Lawler’s algorithm [6] can be used to return top-n ranked solutions for a discrete optimization problem. Multiple solutions are returned by solving the optimization problem under additional constraints, and maintaining a list of candidate ‘top’ solutions for the problem under various constraints.

**General Scoring functions:** The three algorithms given above can also be adapted to the case when the items already chosen affect the score of the item to be added. To see why this feature is useful, consider the following situation: Assume that we have to pick 5 movies for a movie marathon, and we wish to pick a diverse set (\( A \)) of movies. Furthermore, if we have picked 3 movies already (in \( A \)) all three of which are action movies, then the score for a ‘good’ unpicked action movie \( a \) is probably less than that of a ‘good’ comedy movie \( b \), i.e., \( \text{score}(a, A) < \text{score}(b, A) \) (here \( \text{score} \) takes two arguments, an item and a set). We could also define \( \text{score} \) to operate on a set, in which case \( \text{score}(\{a\} \cup A) < \text{score}(\{b\} \cup A) \).

We can adapt the algorithms by letting the score of the items be determined by the remaining items present in the set \( A \) at any point, and since all the algorithms are ‘greedy’, we can do so trivially. However, note that the score for each item would need to be recomputed once the set \( A \) is changed. This operation could be extremely expensive, for example for Algorithm 2, where the values of all sub-chains will need to be updated based on the current choice of \( A \).

For Algorithm 1, we would recompute \( \text{score} \) for each item \( a \) (i.e., \( \text{score}(\{a\} \cup A) \)) after line 3, for the addition of each item. The pair \( a, b \), where \( b \) replaces \( a \) in \( A \) is chosen such that \( \text{score}(A \cup \{b\} - \{a\}) \) is maximum. For Algorithm 2, recomputing of scores of all chains in \( Q \) will have to be done post line 11. In this case, the average score of a sub-chain \( C \) is \( \frac{(\text{score}(A \cup C) - \text{score}(A))/|C|}{|C|} \). For Algorithm 3, the item \( a \) from \( A \) whose prerequisites are added first is that for which \( \text{score}(A - \{a\}) \) is smallest. The item in line 12 is the one such that \( \text{score}(A' - \{a\}) \) is the maximum. The item \( a \) chosen in line 16 is the \( i \) for which \( \text{score}(A - \{a\} \cup \{i\}) \) is maximum. Scores will have to be recomputed after line 18 as well.

## 4 Worst Case Bounds for Chains

We now obtain the worst case difference between the optimal score and the score of the set that we return.

We define the following properties of the graph \( G \): The depth of the chains, i.e., the maximum number of items in any chain is \( \leq d \). The coherence of the chains, i.e., the difference between the minimum and maximum \( \text{score} \) of two items in any chain is \( \leq \gamma \). As before, a set of size \( k \) is desired.

### 4.1 Algorithm 1

The worst case is attained as in Figure 1, when there are \( k \) singleton items (with no prerequisites or children) that have a \( \text{score} \) of \( \alpha \), while there are several \( (\geq k, \text{say}) \) other items which have \( \text{score} \) of \( \alpha - \delta \), for very small \( \delta \), but those items are each the start of a chain of \( (d - 1) \) items that have a score of \( \alpha - \delta + \gamma \) each.

In this case, Algorithm 1 picks \( k \) items that have a score of \( \alpha \) to form part of \( A \), and never discards them in favor of items that have score \( \alpha - \delta \), which forms part of \( \text{extreme}(A) \). However, the optimal algorithm would pick \( k/d \) such (complete) chains, because after the first item in the chain, all the other items have a value of \( \alpha - \delta + \gamma \).
Thus the difference between the optimal score and the score of our algorithm would be: (ignoring $\delta$

\[ \frac{k}{d}(d\alpha + (d-1)\gamma) - k\alpha = \frac{k\gamma(d-1)}{d} \quad (3) \]

As an informal justification of the fact that this is actually the worst case, consider the following: Note that Algorithm 1, on termination, returns a set $A$ such that no item in $\text{external}(A)$ is better than any item in $A$. However, items in $\text{external}(A)$, might be at some point along chains which contain ‘good’ items later on, which are better than items in $\text{external}(A)$. The worst possible such case (containing maximum good items) is when the start of the chain is not in $A$ (and say has score $\alpha - \delta$), followed by several (i.e., $d-1$) ‘good’ items of score $\alpha + \gamma - \delta$, which is the maximum such score. Also note that the worst possible $A$ in this case would contain all items equal to the start of these chains $\alpha - \delta$. However, with a different initial choice of items, we may end up picking the items which are at the beginning of ‘good’ chains. Instead, we decide to set these items to a score of $\alpha$, since we anyway consider $\delta$ to be very small. This situation is precisely the one described above.

### 4.2 Algorithm 2

Assume that $k > d$ for now. Let us consider the case when there are $d$ remaining items to be picked. Let the optimal algorithm and Alg. 2 return the same score until this point. The only unpicked items are shown in Fig. 2. Let there be only $d/2$ singleton items with a score of $\alpha + \gamma/2 + \delta$. However, there is a chain of $d$ items where the first $d/2$ items in the chain, have a score $\alpha$, while the last $d/2$ have a score of $\alpha + \gamma$ (the average score of this chain is $\alpha + \gamma/2$). In this case, the item that is picked first by Algorithm 2 (say) is a singleton node with score $\alpha + \gamma/2 + \delta$. We keep picking $d/2$ such singleton items. Later, since there are only $d/2$ items left, Algorithm 2 picks the first $d/2$ items of the chain, each of which have value $\alpha$. The optimal algorithm, instead, picks the complete chain of $d$ items instead of the singleton nodes.

Here, the difference between the optimal score and the score of the algorithm is (ignoring $\delta$

\[ (d\alpha + \frac{d}{2}\gamma) - (d\alpha + \frac{d}{2}\gamma) = \frac{d}{4}\gamma \quad (4) \]
If \( k < d \), then a similar construction can be used, with \( k \) taking the place of \( d \). Here, the worst case bound is \( k\gamma/4 \).

Any other situation, for example, that involving a chain of length \( c < d \), would involve a worst case bound of \( \gamma c/4 \), which is not as big.

For an informal proof of why this graph is the worst case when \( k > d \) (the proof for \( k < d \) is similar), consider the following: The set \( A \) returned by Algorithm 2 is such that there is no sub-chain in the sub-graph formed by \( V - A \) that has a higher average score than a sub-chain of the reverse of any chain found in \( A \). In other words, if \( a \rightarrow b \rightarrow c \) is in \( A \), there is no sub-chain \( d \rightarrow e \) (that is not in \( A \)) that can have a higher average score than \( c \) or \( b \rightarrow c \) nor is there a sub-chain \( d \), that can have an average score greater than that of \( c \).

This invariant has the following implications. The only situation when the optimal set would contain a different set of items is when there is a chain (or sub-chain), \( C \), whose average score is smaller than the average score \( v \) of the last chain added, but cannot be added because of insufficient capacity. As a result of this insufficient capacity, a sub-standard chain with fewer items will be added. The worst possible sub-standard chain is a sub-chain of \( C \) of the remaining capacity itself. The worst possible difference between the items that were added and those that were not added is \( \gamma \). Let us assume that there are \( r \) items that have score \( \alpha + \gamma \), and \( d - r \) that have value \( \alpha \). The average of this is \( r\gamma/d + \alpha \), each of the \( r \) items that are preferentially chosen have a score of at least \( \alpha + r\gamma/d \). The difference in the scores is \( r\gamma - r^2\gamma/d \). This evaluates to \( \gamma(1 - r/d) \).

Note that in this particular case, Alg. 3 would have picked the optimal set. There are also cases where Alg. 1 performs better than Alg. 2.

Consider the following example:

- \( a(0.5) \rightarrow b(0.9) \)
- \( c(0.6) \rightarrow d(0.6) \rightarrow e(0.85) \)

Let \( k = 3 \). In this case, breadth first search would pick \( c, d \) and then \( e \) — a score of 2.05. However, the greedy algorithm would pick \( \{a, b\} \) and then pick \( \{c\} \), a score of 2.0.

### 4.3 Algorithm 3

The worst case is attained when (say), there are \( \geq k \) singleton items with score of \( \alpha + \gamma - \delta \), but there are \( \geq k \) items which have scores of \( \alpha + \gamma \). However, let these \( k \) items be at the end of ‘bad’ chains, i.e., that of \( (d - 1) \) items with scores of \( \alpha \). This situation is given in Figure 3.

![Figure 3: Worst case for Alg. 3](image)

Algorithm 3 picks \( k \) items with score \( \alpha + \gamma \), and then tries to include the prerequisites for those items. As a result, the algorithm terminates with \( k/d \) complete chains (each of size \( d \)) that end with items with score \( \alpha + \gamma \). The optimal solution in this graph is to pick \( k \) items with score \( \alpha + \gamma - \delta \).

The worst case bound of (ignoring \( \delta \)) the difference between the optimal solution and the solution returned by the algorithm is:

\[
k(\alpha + \gamma) - \frac{k}{d}(\alpha + \gamma + (d - 1)\alpha) = \frac{k\gamma(d - 1)}{d}
\]
Note that this value is the same as the value obtained for Algorithm 1.

For an informal proof of why this graph is the worst case, consider the following: Note that at least \( k/d \) of the items with the best score and their prerequisites are always present in the solution. For the worst case, only \( k/d \) items with the best score will be present in the solution (so that the set returned by our algorithm has smallest possible score). Also, let the score of each of these top \( k/d \) items be \( \alpha + \gamma \). The worst case arises when there are several other items that have almost same score \( \alpha + \gamma - \delta \), but are not chosen. Instead, we constrain that each of these top \( k/d \) items are at the end of long chains with items of score \( \alpha \), which is the smallest such score. This situation is the one described above.

### Algorithm 4 Merge: Merge

**Require:** \( Q_1 \leftarrow \) empty min-priority-queue  
**Require:** \( Q_2 \leftarrow \) empty max-priority-queue  
**Require:** \( A_1 \leftarrow \) first set (with greater score)  
**Require:** \( A_2 \leftarrow \) second set

1. \( A_c \leftarrow A_1 \cap A_2 \)
2. \( A_1' \leftarrow A_1 - A_c \)
3. \( A_2' \leftarrow A_2 - A_c \)
4. for all chains \( c \in A_1' \) do
   5.   for all sub-chains \( c \) of reverse\((C)\) do
   6.       insert \( c \) into \( Q_1 \) with value: \( \sum_{a \in c} \text{score}(a) / \text{size}(c) \)
   7. end for
   8. end for
9. for all chains \( C \in A_2' \) do
   10. for all sub-chains \( c \) of \( C \) do
   11.       insert \( c \) into \( Q_2 \) with value: \( \sum_{a \in c} \text{score}(a) / \text{size}(c) \)
   12. end for
13. end for
14. while \( Q_1 \neq \emptyset \) do
   15.     \( m \leftarrow \text{pop}(Q_1) \) /*chain in \( A_1' \) with min avg. score*/
   16.     \( unused \leftarrow \emptyset; used \leftarrow \emptyset \)
   17.     \( r \leftarrow \text{size}(m) \)
   18.     \( Q_2' \leftarrow Q_2 \)
   19.     while \( Q_2' \neq \emptyset \wedge r > 0 \) do
   20.         \( n \leftarrow \text{pop}(Q_2') \) /*chain in \( A_2' \) with max avg. score*/
   21.         if \( \text{size}(n) > r \) then
   22.             \( unused \leftarrow unused \cup \{n\} \)
   23.         else
   24.             \( used \leftarrow used \cup \{n\} \)
   25.             update sets in \( Q_2 \) as in line 12-19 of Alg. 2 for \( n \).
   26.             \( r \leftarrow r - \text{size}(n) \)
   27.         end if
   28.     end while
   29.     if \( \text{size}(used) = \text{size}(m) \wedge \text{score}(used) > \text{score}(m) \) then
   30.         \( A_1' \leftarrow A_1' - \{m\} \cup used \)
   31.         \( Q_2 \leftarrow Q_2 \cup unused \)
   32.         update sets in \( Q_1 \) as in line 12-19 of Alg. 2 for \( m \).
   33.         update sets in \( Q_2 \) as in line 12-19 of Alg. 2 for each chain in \( used \).
   34. end if
35. end while
36. return \( A_c \cup A_1' \)

### 4.4 Summary

Thus, the worst case bound of Alg. 2 is better than the worst case bound of Alg. 1 and 3, both of which have the worst case of the same magnitude.
5 Complexity

We now examine the worst case complexity for the three algorithms. Let the number of chains be \( n \), and the maximum depth of any chain be \( d \). The exact complexity of the three algorithms depends on the implementation used. However, all three algorithms use at least one priority queue structure: the set \( \text{external}(A) \) for Alg. 1 and 3, and the set containing sub-chains for Alg. 2. Note that there is another priority queue structure in Alg. 1 and 3, i.e., the \( \text{boundary}(A) \) set, but that set is bounded by a maximum size of \( k \), and hence is dominated (for order of magnitude considerations) by the priority queue for the \( \text{external} \) set.

We express our complexity values in terms of three functions \( \text{sort}(n) \): cost of sorting \( n \) items, \( \text{insert}(n) \): cost of inserting \( n \) items into a priority queue and \( \text{mov}(n) \): cost of a value modification in a priority queue containing \( n \) items.

Alg. 1. needs the items at the start of the chains to be sorted and inserted into a priority queue, plus subsequently requires a modification (i.e., replacement of items in the \( \text{external} \) set) of at most \( kd \) values (the items in the chains that have the best \( k \) starting scores) within a priority queue structure (which contains at most \( n \) items at any point). Thus, the worst case complexity is \( O(\text{sort}(n) + \text{insert}(n) + kd \cdot \text{mov}(n)) \).

Alg. 2. requires all \( nd \) sub-chains to be sorted by average score and inserted into a priority queue. Subsequently, at most \( k \) chains will get their values modified (i.e., replaced) — resulting in \( kd \) modifications. Thus the complexity is \( O(\text{sort}(nd) + \text{insert}(nd) + kd \cdot \text{mov}(nd)) \).

Alg. 3. requires all items to be sorted. However, the items inserted into the priority queue are only the \( n \) items at the start of chains. Note that in this case at most \( k \) values get modified (at each iteration, either a prerequisite gets fixed, or a new item whose prerequisites are present is added). Thus, the complexity is \( O(\text{sort}(nd) + \text{insert}(n) + k \cdot \text{mov}(n)) \).

Thus, depending on the implementation of the priority queue and the sorting algorithm, one of Alg. 1 or Alg. 3 would have the best worst case complexity. However, both of these have a better complexity than Alg. 2.

6 Combining Solutions

We now describe a procedure to take two sample solution sets of size \( k \), \( A_1 \) and \( A_2 \), both of which do not violate prerequisites, and generate a set of size \( k \), \( A \), which does not violate prerequisites, and also has a score that is greater than or equal to that of \( A_1 \) and \( A_2 \).

We shall use as our example, set \( \{b, g, a, d\} \) as \( A_1 \) (with a score of 2.5) and set \( \{a, j, k, e\} \) as \( A_2 \) (with a score of 2.4).

Let \( A_c = A_1 \cap A_2 \). This set consists of the common portions of the chains found in \( A_1 \) and \( A_2 \). For the above example, \( A_c = \{a\} \). \( A_c \) will form part of the final solution. We also define \( A'_1 = A_1 - A_c \), i.e., the portions of chains that are unique to \( A_1 \). \( A'_2 \) is defined similarly. For the above example, \( A'_1 = \{b, g, d\} \), while \( A'_2 = \{j, k, e\} \). We wish to answer the question of which portions of \( A'_1 \) and \( A'_2 \) to include in the final set that we return, that will be of total size \( k \).

Let score of \( A'_1 \) be greater than that of \( A'_2 \). We now try to transform \( A'_1 \) by replacing items from it with items from \( A'_2 \). The final solution we return will include the union of the transformed \( A'_1 \) and \( A_c \).

We design two priority queues, a max-priority queue \( Q_2 \) consisting of sub-chains of \( A'_2 \) (sorted on average score), and a min-priority queue \( Q_1 \) consisting of sub-chains from \( A'_1 \) (sorted on average score). However, for the min-priority queue, our sub-chains are from the end of the chain, i.e., we use the sub-chains of reverse of the given chains. In the above example, the chains we add to \( Q_2 \) (with the average score) are: \( \{j, k\} : 0.85, \{j\} : 0.8, \{e\} : 0.2 \), and the chains we add to \( Q_1 \) are: \( \{b, g\} : 0.65, \{g\} : 0.7, \{d\} : 0.7 \).

We try to delete the worst sub-chains of the reverse of chains \( A'_1 \), replacing each of them with a number of sub-chains from \( A'_2 \) that have a larger total score, but the same size in total as the sub-chain from \( A'_1 \).
Thus we incrementally increase the score of $A'_1$, resulting in a better overall score when we take the union of this modified set with $A_c$.

The worst sub-chain in $Q_1$ is $\{b, g\}$, with a score of 0.65, which is popped first. We now try to extract the best items from $Q_2$ to replace this sub-chain. Firstly, we extract $\{j, k\}$, which has average score 0.85, the largest in $Q_2$. Since $\{j, k\}$ has a larger average score, and the same size, we replace $\{b, g\}$ with $\{j, k\}$ in $A'_1$. Next, we need to update the super-chains and sub-chains of the items replaced in $Q_1$ and $Q_2$. Here, the sub-chain in $Q_1$ is $\{j\}$, which has to be removed, since its super-chain has already been selected and removed. Again, the sub-chain in $Q_2$ is $\{g\}$, which has to be removed as well. There are no super-chains, hence we do not update any more items.

Next, we pop from $Q_1$, $\{d\}$. We try to replace this with the only sub-chain left in $Q_2$, $\{e\}$. In this case, $\{d\}$ has a higher score, so we retain it in $A'_1$. Thus, the final value of $A'_1$ is $\{j, k, d\}$, giving rise to the optimal set $A$ of $\{a, j, k, d\}$.

The pseudocode listing is provided in Algorithm 4. We initialize the min-priority and max-priority queues as described above (line 4-13). We then keep removing the worst reverse sub-chain $m$ from the min-priority queue (the sub-chain with the smallest average score) (line 15), and try to replace it with a number of chains from the max-priority queue (line 19-28). The sub-chains that are used in this process are saved in $used$, while the sub-chains that are too large are saved in $unused$ (line 22, 24). If the sum of the scores of the sub-chains from the max-priority queue exceeds the score of the reverse sub-chain from the min-priority queue, then we replace the chain in $A'_1$ with the sub-chains from the max-priority queue (line 30). We also add the items that are $unused$ back to the max-priority queue (line 31), and update the value of the sub-chains and super-chains as in algorithm 2 (line 32).

If better chains are not found, we restore the max-priority queue to its original form and try replacing a new item from the min-priority queue.

7 Experimental Analysis

We aim to assess the expected performance of the three algorithms of Sec. 3. We also implemented an algorithm that returns the top-$k$ items without regard to prerequisites. This algorithm thus provides an upper-bound on the score of the optimal set.

We generated random instances of the graph $G$ in the following fashion: We set the number of chains $C_i$ in the graph to be $n$. Each chain, with probability $p$, is a long chain, i.e., has length greater than or equal to 2. Thus a chain is a singleton item with probability $1 - p$. Now, given that a chain is a long chain, we let the size of the chain be a discrete random variable, uniformly distributed among integers between 2 and $d$, the maximum depth.

We let the score of each item be a continuous random variable, exponentially distributed, with mean 0.5. As before, $k$ represents the size of the set of items we wish to extract from graph $G$.

For each experiment described in the following sections, we took several random instances of graphs generated as described above and determined the average ratio of the score of the set returned by each of the 3 algorithms to the score returned by the algorithm that does not take prerequisites into account.

Due to space limitation, we only provide here a sample of our results. We have experimented with other parameter settings and distributions, and the conclusions are not that different from what we show here. In particular, we have experimented with scores that follow a Zipfian distribution. Since the Zipfian distribution usually has a few outliers with large score, the top-down pickings algorithm tends to work better than the breadth first pickings algorithm.

7.1 Variation with Number of items picked

When we vary the target number of items, $k$, we find the following:

- Alg. 2: Greedy Value Pickings is always better than the other two algorithms.

1The scores of these items may need to be adjusted based on whether or not their sub-chain is present in $used$(line 33).
Variation of fraction of score of the best set with $k$:

- Alg. 3: Top Down Pickings is better than Alg. 1: Breadth First Pickings when $k \approx n$, the number of chains, while the reverse is true when $k$ is small compared to $n$.
- All 3 algorithms return sets whose score is at least 75% of the optimal set on average if there were no prerequisites.

Figure 4 illustrates some of our results. For this experiment we set $n = 50$, $p = 0.2$ and $d = 5$ and generated 500 random graph instances as described above. In Figure 4 the horizontal axis shows $k$ varying from 5 (small compared to $n$) to 45 (large compared to $n$). The vertical axis shows the score of the set returned by each algorithm, as a fraction of the best possible score (averaged over all graph instances). For example, if we set $k = 20$, we find that on average, both Alg. 1 and Alg. 3 return a set that has a score of 87% of the best score (when prerequisites are not taken into account). On the other hand, on average, Alg. 2 returns a set that has a score of 92% of the best score.

Alg. 2 does well for all cases, above 90% of the optimal. As we increase $k$, Alg. 2 tends to do even better, because Alg. 2 is less likely to miss out on any ‘good’ items when it is less constrained by $k$.

We find that when $k$ is small, Alg. 1 does better than Alg. 3, probably because Alg. 1 does a better job of exploring items that are close to the start of chains (and since $k$ is small, we can only include items that are very close to the start of chains). We find that around $k = 20$, Alg. 3 starts doing better than Alg. 1. As $k$ becomes $\approx n$, Alg. 3 is much better than Alg. 1, probably because Alg. 1 tends to do a local search close to the start of chains, while Alg. 3 actively tries to include the top items.

Variation of Fraction of score of the best set with $p$:

- Alg. 3: Top Down Pickings is better than Alg. 1: Breadth First Pickings when $k \approx n$, the number of chains, while the reverse is true when $k$ is small compared to $n$.
- All 3 algorithms return sets whose score is at least 75% of the optimal set on average if there were no prerequisites.

Figure 5 illustrates some of our results. For this experiment we set $n = 50$, $p = 0.2$ and $d = 5$ and generated 500 random graph instances as described above. In Figure 5 the horizontal axis shows $p$ varying from 0 to 0.7. The vertical axis shows the score of the set returned by each algorithm, as a fraction of the best possible score (averaged over all graph instances). For example, if we set $p = 0.4$, we find that on average, both Alg. 1 and Alg. 3 return a set that has a score of 80% of the best score (when prerequisites are not taken into account). On the other hand, on average, Alg. 2 returns a set that has a score of 90% of the best score.

Alg. 2 does well for all cases, above 90% of the optimal. As we increase $p$, Alg. 2 tends to do even better, because Alg. 2 is less likely to miss out on any ‘good’ items when it is less constrained by $p$.

We find that when $p$ is small, Alg. 1 does better than Alg. 3, probably because Alg. 1 does a better job of exploring items that are close to the start of chains (and since $p$ is small, we can only include items that are very close to the start of chains). We find that around $p = 0.2$, Alg. 3 starts doing better than Alg. 1. As $p$ becomes $\approx n$, Alg. 3 is much better than Alg. 1, probably because Alg. 1 tends to do a local search close to the start of chains, while Alg. 3 actively tries to include the top items.
7.2 Variation with Number of Long Chains

As we vary \( p \), the probability that a chain is ‘long’, we have the following results:

- *All algorithms tend to perform poorly on increasing \( p \) beyond a certain value.*
- *However, the relative ordering between the scores of the algorithms tend to remain the same.*

In this case, we set \( n = 50, d = 5 \) and \( k = 10 \). For each value of \( p \) ranging from 0.05 to 0.60, we generated 500 random graph instances as described above. We then ran the three algorithms on those instances and measured the average ratio of score versus the best set, and plotted the values in Fig 5.

We find that all three algorithms tend to do badly if \( p \) is increased beyond a certain point, probably because the best items tend to be buried in big chains instead of being singletons. However we find that the behavior of the three algorithms tend to be same, and the relative ordering is maintained. Since \( k \) is small relative to \( n \), Alg. 1 is always better than Alg. 3. If we set \( k \approx n \) and repeat the same experiment, we find that Alg. 3 is better than Alg. 1 throughout.

Figure 6 shows the variation for \( p \) when \( k = 45, n = 50 \). Here, the top-down approach beats the breadth first pickings algorithm.

![Figure 6: Variation of Fraction of score of the best set for the 3 algorithms with \( p \) for the case when \( k \approx n \)](image)

7.3 Additional experiments

We obtain similar results when repeating the experiment in Sec. 7.1 with an exponential distribution on the size of the ‘long’ chain, instead of the uniform distribution. We round down the exponential random variable into an integer (which is exponentially distributed with mean \( d/2 \), if it is part of a long chain). Refer Figure 7 for details.

In figure 8, we repeat the experiment on varying \( k \) when the score is sampled from a zipfian distribution (with the parameter of the curve set to 3). As described earlier, a top down approach works much better in this case, since the zipfian distribution tends to have a few outliers. Here, the intersection between the curve for the breadth first pickings curve and the top down pickings curve happens much earlier (around \( k = 7 \)).

8 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 [10], other ‘similar’ items that the user liked [9], 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.

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.

9 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. We compared these 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, as described
in Sec. 5.

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.

We proved worst-case bounds for the three algorithms, and found that the worst case for the greedy
value pickings algorithm is better than the worst case for the other two algorithms. However there may be
instances on which this algorithm performs worse than the other two algorithms (as seen in Sec. 4.2).

Since there may be specific instances on which each algorithm above does better than the rest, we
developed an algorithm that takes two sets and combines them to create a new set that satisfies prerequisites
and has a higher score. This algorithm could potentially be used to create a set that combines the “best
elements” of the sets returned by the three algorithms.

References


In EC ’08.


[8] A. Parameswaran, P. Venetis, and H. Garcia-Molina. Recommendation systems with complex con-


algorithms. In WWW ’01.