Evaluating, Combining and Generalizing Recommendations with Prerequisites

Aditya Parameswaran
Stanford University
adityagp@cs.stanford.edu

Hector Garcia-Molina
Stanford University
hector@cs.stanford.edu

Jeffrey D. Ullman
Stanford University
ullman@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 movie 'Godfather I' is a prerequisite of 'Godfather II'). Since this general problem is computationally intractable, we develop 3 approximation algorithms to solve this problem for various prerequisite structures (e.g., chain graphs, AND graphs, AND-OR graphs). We derive worst-case bounds for these algorithms for these structures, and experimentally evaluate these algorithms on synthetic data. We also develop an algorithm to combine solutions in order to generate even better solutions, and compare the performance of this algorithm with the other three.

1. INTRODUCTION

Traditional recommendation systems deal with the problem of recommending items or sets of items to users by using various approaches [2, 17]. However, most of these approaches do not 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. For example, university courses often have prerequisites. If course i cannot be taken unless 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 recommend *both* i and j, or perhaps we could recommend some other course k that may be less desirable than 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 [15]. (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 can 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.

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 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 the ranking scenario 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.

Additionally, recommending each item that satisfies prerequisites one-by-one, instead of a "package" recommendation of size k, can prove to be sub-optimal, since at the start, we might end up not choosing items which are prerequisites for a lot of "good" items and as a result we may have to make poor recommendations later on. The challenge, therefore, is to not just recommend a package that satisfies prerequisites, but one that is the "best".

We also study various kinds of prerequisite constraints. For example, in some domains, it may be the case that in order to be able to take an item a, one needs to take either b or c. In other domains, in order to be able to take an item a, one might need to take both b and c.

This paper extends the work in our short paper [14], which contained only the three algorithms for the special case of chain graphs. (We summarize this content in two pages.) Our contributions are the following:

- We define the problem of recommendations with prerequisites for general prerequisite structures in Sec. 2.
- We prove hardness of recommendations for certain classes of prerequisite structures in Sec. 2.1.
- We provide a PTIME dynamic programming algorithm for a special prerequisite class in Sec. 3.1.
- We give approximation algorithms that can adapt to any prerequisite structure in Sec. 3.2-3.5, obtain their worst case bounds and prove incomparability in Sec. 4, and derive complexity in Sec. 5.
- We provide an algorithm that takes sets recommended by different algorithms and combine them to obtain a new set that is even better in Sec. 3.6.
- We experimentally evaluate the algorithms for two kinds of prerequisite structures in Sec. 6.

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 acyclic graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, where the vertices $v \in \mathcal{V}$ correspond to items, and directed edges $(u, v) \in \mathcal{E}$ correspond to prerequisites, i.e., 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 [17, 2, 7] 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 edge.

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

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

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

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

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 call the prerequisite graph above an *AND Graph*, because in order to be able to take any node, *all* of the parents need to be taken as well. Another graph variant is called an *OR Graph*, where in order to be able to take any node, *at least* one of the parents needs to be taken. In this case, Eq. 2 is modified to:

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

OR graphs are common in the course recommendations, since there could be many prerequisites courses with overlapping content. For example, "Programming in C" and "Programming in Java" are both parents of "Algorithms", even though only one of them needs to be taken. A third variant (and generalization of the previous two) is an *AND-OR Graph*, where at each node, either all of the parents need to be taken (i.e., the node is an *AND node*), or at least one of the parents needs to be taken (i.e., the node is an *OR node*).

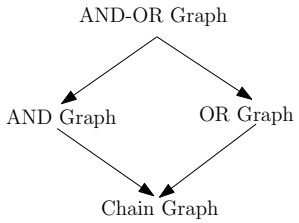

Figure 1: Hierarchy of prerequisite structures studied in this paper

We also consider *Chain Graphs*, a special case for which we can obtain exact solutions. First, 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, II and III would form a chain as follows: Godfather I $\rightarrow$ Godfather II $\rightarrow$ Godfather III. On the other hand, the movie "Shawshank Redemption" would form a singleton node with no edges either going in or coming out. 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*, while $a_1 \rightarrow a_2 \rightarrow \ldots \rightarrow a_n \rightarrow \ldots \rightarrow a_{n+m}, m \geq 0$ is a *super-chain*.

Note that Chain Graph is a special case of an AND graph as well as an OR Graph, since if every item has at most one parent, then that parent would need to be selected for both AND and OR graphs. The hierarchy of prerequisite structures is displayed in Fig. 1. We study recommendations for each of these structures in this paper.

Our algorithms can be generalized to handle fuzzy prerequisites and generalized scoring functions, with different worst case guarantees, as discussed in Appendix C. In particular, fuzzy prerequisites allow multiple scores for a given item based on whether or not the prerequisites for that item are present. Furthermore, with generalized scoring functions, item scores need not be independent. (In this paper, we assume scores are independent.)

2.1 Complexity

We find that picking the best set satisfying prerequisites is NP-Hard for OR, AND and AND-OR graphs. We first prove the hardness result for AND graphs, and then for OR graphs. Trivially, either one of these reductions would be applicable to AND-OR graphs, making them NP-Hard as well.

2.1.1 Hardness of AND Graphs

The problem of picking the best set $A, |A| = k$, satisfying prerequisites for AND Graphs is NP-Hard via a reduction from set cover. Consider a set cover instance with sets $s_1, s_2, \ldots, s_n$, covering some items from the set $\mathcal{T} = \{t_1, t_2, \ldots, t_m\}$. We aim to find a set cover of size k, i.e., a selection of k sets such that all items in $\mathcal{T}$ are covered. We reduce set cover to the decision version of the prerequisite problem: Given a graph G, is there a set satisfying prerequisites of size k^* which has score $\geq s$. We reduce the given set

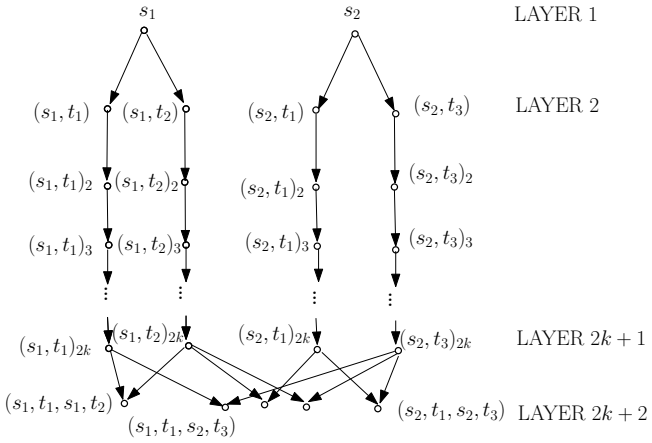

Figure 2: NP-Hardness Proof for AND Graphs

cover instance to the following decision problem: Is there a set satisfying prerequisites of size $k^* = 2mk + k + \binom{m}{2}$ that has a score $\geq \binom{m}{2}$ in a graph G that we construct as follows: First create a node corresponding to each set s_i. These nodes form the first layer. Now we create nodes corresponding to each item in s_i. Thus if t_j is in set s_i, we create a node (s_i, t_j) in the second layer. We create directed edges from s_i to each (s_i, t_j) node created. Subsequently, we form a chain of $2k-1$ nodes below each (s_i, t_j), i.e., we create a chain $(s_i, t_j) \rightarrow (s_i, t_j)_2 \rightarrow (s_i, t_j)_3 \rightarrow \ldots \rightarrow (s_i, t_j)_{2k}$. Subsequently, we create nodes for each t_i, t_j pair; i.e. for $(s_p, t_i)_{2k}$ and $(s_q, t_j)_{2k}$, where $i \neq j$, we create a new node (s_p, t_i, s_q, t_j), such that there are directed edges from $(s_p, t_i)_{2k}$ and $(s_q, t_j)_{2k}$ to (s_m, t_i, s_l, t_j). Only the nodes with four labels, i.e., those at the last layer, have a score of 1, all other nodes have score 0. This construction is depicted for an example in Fig. 2. In the figure, we show just two sets s_1, s_2 covering items t_1, t_2 and t_1, t_3 respectively, and the edges between them (some labels are omitted for clarity).

If there is a set cover of size k, then it is easy to construct a set of size k^* that satisfies prerequisites, and has score at least $\binom{m}{2}$: First, pick the nodes corresponding to the sets comprising the set cover at the first layer. Then pick m complete chains (one for each t_i) such that each chain belongs to a set s_i that has been chosen in the first layer. Such chains are always present because each item t_i is present in at least one s_j. Then, we pick all $\binom{m}{2}$ nodes at the last layer formed at the end of each pair of chains from the m chains (since each chain corresponds to a unique t_i, all pairs of chains have a node at the last layer). Thus the size of the set is $\leq k^*$ and has score $\binom{m}{2}$. We now try to prove the converse.

If there is a set of size k^* satisfying prerequisites of score $\geq \binom{m}{2}$ then the following hold:

- At least $\binom{m}{2}$ items are picked in the last layer, because only those items have non-zero score.
- At least m items are picked in the last-but-one layer, because items in the last layer are formed from pairs of items in the penultimate layer. If at least m items are not picked from the penultimate layer, then we cannot pick $\binom{m}{2}$ from the last layer, since each item at the last layer takes a distinct pair of prerequisites from the penultimate layer. Therefore, at least m chains are picked (so as to not violate prerequisites), which amounts to $2mk$ items.
- If there are more than m items picked in the penultimate layer, then there are at least $2mk + 2k$ items picked in the chains, and at least $2mk + 2k + \binom{m}{2}$ items picked, which violates the number of items picked (since the total number of items picked has to be exactly $2mk + k + \binom{m}{2}$). Thus there are exactly m chains picked, completely.
- All m chains need to correspond to different items, otherwise $\binom{m}{2}$ items cannot be picked in the last layer.
- There are at most $k^* - \binom{m}{2} - 2mk = k$ items chosen at the first layer, corresponding to sets.

Thus, the sets that are picked correspond to a set cover, since each item from $\mathcal{T}$ is found in the set that is the parent of the chain corresponding to the item. Thus, there is a set cover. Hence, the reduction holds.

2.1.2 Hardness of OR Graphs

The proof of NP-hardness for OR Graphs involves a reduction from Set Cover as well, and is much simpler. A brief outline of the proof follows: Consider a node corresponding to each set $s_i \in \{s_1, s_2, \ldots, s_n\}$ in the set cover problem, and each item $t_i \in \{t_1, t_2, \ldots t_m\}$. We connect node s_i to a node t_j via a directed edge if $t_j \in s_i$. Additionally, each t_i has score 1, while all s_i have score 0. If the set cover problem asks for a set of size k, then the decision version of the prerequisite problem for the OR Graph asks if there exists a set of size $k + m$ with score $\geq m$ satisfying prerequisites.

3. THE BASIC ALGORITHMS

Since the problem of recommendation with prerequisites is NP-Hard for AND, OR and AND-OR Graphs, we can only provide approximate solutions. Throughout this paper, the algorithms listed are for AND Graphs (and therefore for chain graphs as well), which forms a reasonable prerequisites structure for movies, books and other media. However, our algorithms, with simple modifications, work for OR and AND-OR graphs as well. We defer the modifications to them for OR and AND-OR graphs to Section 3.7. We also walk through each algorithm for an example graph in the Appendix A.

For the special case of chains graphs, there is an expensive but exact PTIME algorithm that we examine first, in Sec. 3.1. Note that all algorithms return a set of size k assuming one exists. If not, the algorithms return the entire set of items.

3.1 Exact Chains Algorithm

For the case when $\mathcal{G}$ is a forest of chains $C_1, \ldots C_n$, the problem of finding the best set satisfying prerequisites is solvable using a PTIME dynamic programming algorithm.

The algorithm *DP Chains* generates an array a of size $(n+1) \times (k+1)$. The (i, j)th entry of this array corresponds to the score of the best package that can be obtained by picking j items from the first i chains, while satisfying prerequisites. For entries in the ith row, we only need to consider the values of a for the previous row, and the scores of items in the current chain C_i. In particular, consider entry (i, j). We can pick l items where $0 \leq l \leq j$ items from the chain C_i and pick the remaining $j - l$ items from the previous $i - 1$ chains. The optimal score of picking $j - l$ items from the previous $i - 1$ chains is stored at $a[i-1][j-l]$, so it does not need to be recomputed. Hence there is an optimal substructure built into this formulation of the problem.

However, note that this algorithm cannot be generalized to the case of DAGs or even trees, since subgraphs picked in either case cannot be picked independently of each other, unlike in the chain graph case. Thus, there is no optimal substructure in the DAG case. To combat this problem, we provide three other algorithms in subsequent sections. However, these algorithms will be approximate.

Note also that the complexity of the dynamic programming algorithm, $O(nk^2)$ may be high because n may be high. The algorithms in subsequent sections describe algorithms which operate in $O(n)$

or $O(nk)$ and hence are more efficient. Note that in particular, the Greedy-value Pickings Algorithm that we describe performs almost as well as *DP Chains* Algorithm on synthetic chain graphs.

DP Chains Algorithm

Require: $k \leftarrow$ size
Require: $G \leftarrow$ graph of chains $C_1, C_2, \ldots, C_n$
1: $a \Leftarrow$ array of size $(n+1, k+1)$
2: **for all** i in $0 \ldots n$ **do**
3: **for all** j in $0 \ldots k$ **do**
4: $a[i][j] \Leftarrow 0$
5: **if** $i == 0$ **then**
6: continue
7: **end if**
8: $s \Leftarrow 0$
9: **for all** l in $0 \ldots j$ **do**
10: $s \Leftarrow s+$ score of l^{th} item in C_i
11: $a[i][j] \Leftarrow max(a[i][j], a[i-1][j-l]+s)$
12: **end for**
13: **end for**
14: **end for**
15: **return** $a[n][k]$

3.2 Definitions

We define $boundary(A)$ as the set of items in A each of which 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 $\mathcal{E}$.

We define $external(A)$ as the set of items in $\mathcal{V}$ 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.

The modifications to these definitions for OR and AND-OR graphs can be found in Sec. 3.7, along with the modifications to the algorithms described below.

3.3 Algorithm 1: Breadth-first Pickings

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.

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 *value*, i.e., the average *score* of the items in the set, and on querying returns the set with the largest *value*.

We list the pseudocode in **Algorithm 2**. We insert a set corresponding to each node in the graph $\mathcal{G}$ into the max-priority 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. On performing *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 items by popping sets from the queue (line 8-9). If the popped set is small enough to be added to 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).

The algorithm has to terminate because the number of sets in the priority queue is bounded by the total number of items.

Algorithm 1 *Breadth-first Pickings*

Require: $k \leftarrow$ size
Require: $G \leftarrow$ AND 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 $b \in B$ with largest $score$
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

Algorithm 2 *Greedy-value Pickings*

Require: $k \leftarrow$ size
Require: $G \leftarrow$ AND 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

In **Algorithm 3** we start with the best set of items of size k (line 1) by picking items with largest *score* without regard to prerequisites, and try to incrementally add prerequisites. We keep track of items that we have already added prerequisites for in B (line 6). We 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).

Algorithm 3 *Top-down Pickings*

Require: $k \leftarrow$ size
Require: $G \leftarrow$ AND 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

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 added without violating prerequisites because their prerequisites are already present. Note that such an item can always be found by simply picking the first unpicked item in a topological sort of the graph.

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.

Note that there may be cases where **Algorithm 3** does much worse than the other two algorithms, simply because we blindly add prerequisites of items with high 'score', without regard to the 'score' of those prerequisites. There are a few reasons why we consider **Algorithm 3** important: firstly, typically chains are *coherent*, meaning that there is not much variation between scores of items in a chain. Therefore, if we pick a chain containing an item with a high score, then it is likely that other items in that chain have high score as well. Secondly, **Algorithm 2** is expensive, because subchains of each chain have to be inserted into an priority queue, and values need to be updated after every addition, while **Algorithm 1** and **Algorithm 3** are less expensive. However **Algorithm 1** has no concept of a look-ahead towards items having a high score, while **Algorithm 3** has a look-ahead aspect built into it. Thirdly, we discuss later on, an approach to combine several candidate solutions, and diverse algorithms are likely to give rise to better results on combining.

3.6 Algorithm 4: Combining Solutions

We have also designed an algorithm, *Merge*, that takes as input two sample solution sets A_1 and A_2 of size k (both satisfying prerequisites) for an AND graph. *Merge* generates a new set of size k, which does not violate prerequisites, and also has a *score* greater than or equal to A_1 and A_2. This set is formed only using elements from $A_1 \cup A_2$. The algorithm starts with the set with higher score and iteratively replaces subgraphs with least average score with subgraphs with high average score from the other set, while retaining prerequisites and making sure we have k items. The description of this algorithm and pseudocode can be found in Appendix B.

3.7 Modifications for Other Structures

We now describe the modifications to definitions discussed in Sec. 3.2 for AND-OR graphs (and therefore for OR graphs as well), and then provide specifics on the modification of each algorithm for AND-OR graphs.

For AND-OR graphs, $boundary(A)$ is the set of items in A such that each of the items is not a prerequisite or a potential prerequisite of any other item in A, i.e., if $x \in boundary(A)$, then there does not exist an item $y \in A$ and items $x_1, x_2, \ldots, x_n$ such that there is a sequence of edges $(x, x_1), (x_1, x_2), \ldots, (x_n, y)$ in $\mathcal{E}$.

Similarly, $external(A)$ is the set of items that can be added such that prerequisites are not violated. For an AND-OR graph, if the node x added is an AND node, then all items y such that there is a edge from y to x in $\mathcal{E}$ must also be present in A, as well as prerequisites for all such y. If the node x is an OR node, then at least one y (whose prerequisites are already in A) such that there exists an edge from y to x should be present in A.

3.7.1 Algorithm 1

We replace line 8 in the algorithm to reflect the fact that a should not be necessary for b, i.e., b's prerequisites in $A - \{a\}$ are present.

3.7.2 Algorithm 2

For AND-OR graphs, as before, we insert sets corresponding to each item into the max-priority queue in a top-down fashion. For an AND node a, in line 5, we include the set of prerequisites that is the union of the prerequisite sets corresponding to all of a's immediate parents. For an OR node a, in line 5, the set of prerequisites corresponding to a's immediate parents with the largest *value* (as in the algorithm) is included. For OR graphs, this procedure corresponds to selecting the best path to any root (i.e., the path with greatest *value*.) Although we do not prove it here, this procedure guarantees that a set of size k respecting prerequisites, if present, is always returned.

3.7.3 Algorithm 3

Line 5 uses the "best" set of prerequisites, i.e., those used in Sec. 3.7.2 to augment the set corresponding to each item.

4. WORST CASE BOUNDS

In this section, we derive bounds on the worst case difference between the score of the optimal set and the score of the set that we return for each prerequisite structure. The worst case bounds are listed in Table 1. We prove the worst case bounds for chain graphs and some other prerequisite structures here and defer the remaining proofs to the appendix.

We define the following properties of the graph $\mathcal{G}$: The *coherence* of $\mathcal{G}$, i.e., the maximum difference between the minimum and maximum *score* of two items in any connected component in the graph is γ. (For the case of chain graphs, γ is the maximum difference between two items in any chain.) Additionally, the *depth* of a graph $\mathcal{G}$, i.e., the maximum length of any directed path, is d. (For chain graphs, the *depth* is the maximum number of items in any chain.) As before, a set of size k is desired.

4.1 Algorithm 1: Breadth-first Pickings

We consider chain graphs first. Assume $d < k$. The worst case is attained as in Fig. 3, when there are k singleton items that have

Structure	DP	Alg 1	Alg 2	Alg 3
chain graphs	0	$k\frac{\min(k,d)-1}{\min(k,d)}$	$\frac{\min(k,d)}{4}$	$k\frac{\min(k,d)-1}{\min(k,d)}$
AND	-	$k-1$	$k+2-2\sqrt{k+1}$	$k-1$
OR	-	$k-1$	$k-\frac{(\min(d,\sqrt{k+1})+1)^2+k}{\min(d,\sqrt{k+1})}$	$k\frac{\min(k,d)-1}{\min(k,d)}$
AND-OR	-	$k-1$	$\geq k+2-2\sqrt{k+1}$	$k-1$

Table 1: Worst Case Bounds for each prerequisite structure for each algorithm (all entries are multiplied by γ)

a $score$ of α, while there are several ($\geq k$, say) other items which have $score$ of $\alpha-\delta$, for very small δ, 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, Alg. 1 picks k items that have a score of α to form part

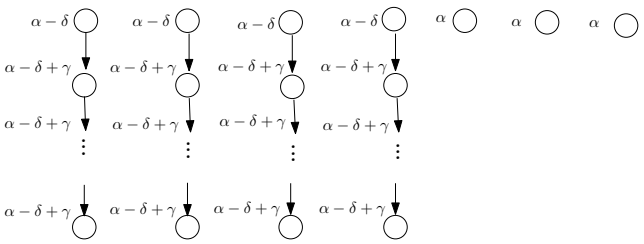

Figure 3: Worst case for Alg. 1

of A, and never discards them in favor of items with score $\alpha-\delta$, which forms part of $external(A)$. 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 δ)

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

The worst case for $d\geq k$ has a single long chain of size k as above and many ($\geq k$) singleton items. The worst case difference is then $(k-1)\gamma$.

As a justification for the fact that this is actually the worst case, consider the following: Note that Alg. 1, on termination, returns a set A such that no item in $external(A)$ is better than any item in $boundary(A)$. Also note that in each of the chains, the first item that is present in the optimal set but not in A will lie in $external(A)$. Now let there be a sub-chain in the optimal set no part of which is in A. The first item of this sub-chain (score $\alpha-\delta$, say) forms part of $external(A)$ for all iterations of Alg. 2. Thus, in the worst case, each item in A is equal to, or slightly better than $\alpha-\delta$. Thus A has k items of score α. We now try to maximize the score of the remaining items in the optimal set. The maximum score is when the first item of each of the sub-chains in the optimal set is not in A and has score $(\alpha-\delta)$. (If it had a higher score, then it would be chosen by A.) Each of these items are followed by $d-1$ 'good' items of score $\alpha+\gamma-\delta$, which is the maximum such score. This situation is precisely the one described above. Though we do not prove it here, the case when every sub-chain in the optimal set overlaps with some sub-chain in A does not change the worst case bound. (This case has lower difference because of high overlap between A and the optimal set, and the coherence constraint.) Note that if $d\geq k$, k takes the place of d above.

We prove the worst case bounds for other prerequisite structures in Appendix A.6.1.

4.2 Algorithm 2: Greedy-value Pickings

We consider chain graphs first. Assume that $k>d$ for now. Let there be 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. 4. 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 α, while the last $d/2$ have a score of $\alpha+\gamma$. (The average score of this chain is $\alpha+\gamma/2$.) In this situation, the item that is picked first by Alg. 2 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, Alg. 2 picks the first $d/2$ items of the chain, each of which have value α. The optimal algorithm picks the chain of d items instead of the singleton nodes. The difference between the optimal score and the

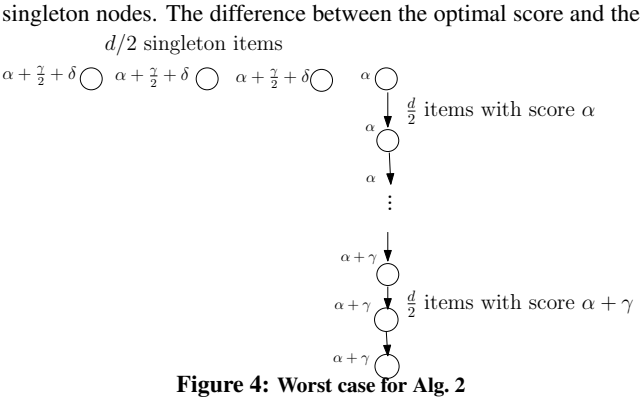

Figure 4: Worst case for Alg. 2

score of the Alg. 2 is (ignoring δ)

$$(d\alpha+\frac{d}{2}\gamma)-(d\alpha+\frac{d}{2}.\frac{\gamma}{2})=\frac{d}{4}\gamma \quad (4)$$

If $k\leq d$, then a similar construction can be used, with k taking the place of d. Here, the worst case bound is $k\gamma/4$.

For an informal proof of why this graph is the worst case when $k>d$ (the proof for $k\leq d$ is similar), consider the following: Note that if we do not discard any sets in the execution of the algorithm via the condition in line 10, then we are guaranteed an optimal solution. To see this, consider A returned by the algorithm, and an optimal A^* with higher score. Let the average score of the last set picked by the algorithm be v. Now, since A^* has a higher score, (A^*-A) must have a sub-chain c that can be added to A without violating prerequisites and has higher average score than v. (If all sub-chains have same average score v, then there is no way A^* can have higher score.) However, if c has higher average score than v, it should have been picked instead of the last set that was picked in the algorithm.

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 due to insufficient capacity. (The case where there are multiple chains that are not picked is no worse.) As a result of this insufficient capacity, a sub-standard chain with fewer items is 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 γ. Let us assume that there are $d-r$ items in C that have value α that were added, followed by r items that have score $\alpha+\gamma$ that were not added. The average of C 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 $r\gamma-r^2\gamma/d=\gamma(r)(1-r/d)$. The optimal value of r, for which the difference is greatest, is $d/2$. Thus the worst case bound is $\gamma d/4$.

4.2.1 Worst case bounds for other structures

For AND graphs, the worst case situation is displayed in Fig. 5. In the figure, there are $k-x$ singleton items with score $\alpha+\frac{\gamma}{x+1}+\delta$. Additionally, there is another connected component which has x root items with score α, and $k-x$ items with score $\alpha+\gamma$,

each of which has all x root items as prerequisites. Note that this situation is similar to the one in Fig. 4, in the sense that in both the situations there are several good items at the end of a subgraph of bad items, but Alg. 2 never gets to the good items because it makes poor greedy choices early on.

In this case, the algorithm picks $(k-x)$ items of score $\alpha + \frac{\gamma}{x+1} + \delta$ first, since those items have the highest average score. Then, the algorithm picks the x items that have a score of α. The optimal algorithm, on the other hand, picks all items from the first connected component.

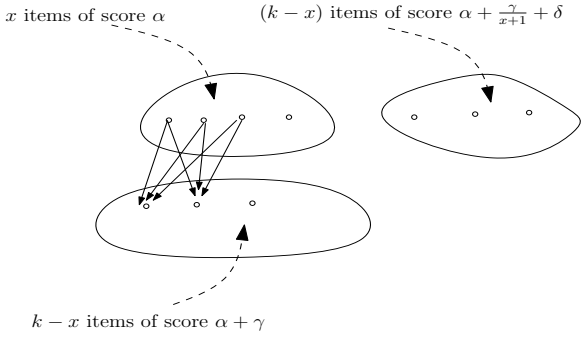

Figure 5: Worst case for Alg. 2 for and graphs

We ignore δ in the following calculation. The score of the set returned by Alg. 2 is $k\alpha + \frac{(k-x)\gamma}{x+1}$, while the score of the optimal set is $k\alpha + (k-x)\gamma$. Thus the difference is: $\gamma(k-x)\frac{x}{x+1}$. The largest value of this difference (forming the worst case) is $\gamma(k + 2 - 2\sqrt{k+1})$, with $x = \sqrt{k+1} - 1$.

We prove the worst case bounds for OR and AND-OR graphs in Appendix A.6.2.

4.3 Algorithm 3: Top-down Pickings

For chain graphs, the worst case is attained when 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., $(d-1)$ items with scores of α. This situation is given in Fig. 6.

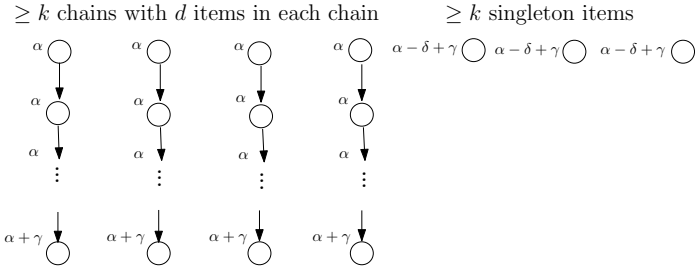

Figure 6: Worst case for Alg. 3

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 difference (ignoring δ) between the optimal solution and the solution returned by the algorithm is:

$$k(\alpha+\gamma) - \frac{k}{d}(\alpha + \gamma + (d-1)\alpha) = \frac{k\gamma(d-1)}{d} \qquad (5)$$

Note that this value is the same as the value for Alg. 1. When $k \geq d$, the worst case is still Fig. 6 but with a single chain of size k. The worst case difference is then $(k-1)\gamma$.

We prove that Fig. 6 is the worst case for chain graphs and derive bounds for other structures in Appendix A.6.3.

4.4 Summary and Incomparability

In summary, for chains, the worst case bound of Alg. 2 is better than the worst case bound of Alg. 1 and 3, both of which have worst case bounds of the same magnitude. As can be seen in Table 1, this relationship holds true for other prerequisite structures as well. The only case where Alg. 3 and Alg. 1 have different worst case bounds is for OR graphs, wherein Alg. 3 can add at most d bad elements (i.e., prerequisites) by making a poor greedy choice, while Alg. 1 adds $k-1$ bad elements.

We now prove that the three approximation algorithms are incomparable, i.e., there exist cases where each algorithm does better than the other two. The proof is for chain graphs and therefore also holds for OR, AND-OR and AND graphs.

For the case when the graph is Fig. 4 (and $k = d$), Alg. 3 would pick the optimal set, while Alg. 1 and Alg. 2 would pick sub-optimal sets. There is also a case where Alg. 1 performs better than both Alg. 2 and Alg. 3. Consider the following graph:

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

Let $k = 3$. In this case, Alg. 1 would pick c, d and then e — a score of 2.05. However, Alg. 2 and Alg. 3 would pick $\{a, b\}$ and then pick $\{c\}$, a score of 2.0.

Alg. 2 would do better than Alg. 1 and Alg 3 on the following example (with $k = 3$):

- $a(0.51) \rightarrow b(0.9)$
- $c(0.5) \rightarrow d(0.8) \rightarrow e(0.85)$

Here, Alg. 1 and Alg. 3 would pick $\{a, b, c\} = 1.91$, while Alg. 2 picks $\{c, d, e\} = 2.15$.

Thus, *the three algorithms are incomparable.* Therefore, given the resources, we might wish to implement all three algorithms in order to improve our recommendations.

5. COMPLEXITY

We now examine the worst case complexity for the exact algorithm, three heuristic algorithms and the merge algorithm for chains. Let the number of chains be n.

The exact dynamic programming algorithm maintains a matrix of size $O(nk)$, and computes each entry in time $O(k)$. Thus the overall complexity is $O(nk^2)$.

For the rest of the algorithms, we can prune the search for the best set at the start itself. We use a fibonacci heap, which takes $\Theta(n)$ to construct and $O(\log n)$ to delete any element.

For Alg. 1, we make a pass of the first item in each of the n chains, and extract at most k chains with the best first items. This operation can be done in $O(n + k \log n)$ using the fibonacci heap. Subsequently, we take at most $O(k^2)$ items (from the k chains), and then run Alg. 1 on those items. If we use two fibonacci heaps to store the items in $external$ and $boundary$, we insert each item at most once in each of the heaps, and extract each item at most once (k^2 extractions of $O(\log k^2)$). Thus, this phase is $O(k^2 \log k)$. Thus, Alg. 1 is $O(n + k \log n + k^2 \log k) \approx O(n)$ if $n >> k$.

For Alg. 2, we first extract k best sub-chains of each size from $1 \ldots k$. This operation can be done in $O(nk + k^2 \log n)$ since there are $O(nk)$ sub-chains, and since we extract k items from k heaps of size $O(n)$ (one heap corresponding to each length of sub-chain from 1 to k). Subsequently, all sub-chains of the k^2 chains corresponding to the k^2 sub-chains picked earlier (i.e., at most k^3 sub-chains) are used in Alg. 2. We use k fibonacci heaps (one corresponding to chains of each size from $1 \ldots k$), and extract at most k items in total. Each such item will then trigger the change of the average score of at most k items. Thus, we insert items in heaps in $O(k^3)$, extract items in $O(k \log k)$, and modify score in

$O(k^2 \log k)$, giving a complexity $O(nk+k^2 \log n+k^3) \approx O(nk)$, if $n >> k$.

For Alg. 3, we make a pass through all the $O(nk)$ items to extract the best k items, and also the best k items at the that do not have any prerequisites (to replace the items whose prerequisites cannot be added), in $O(nk + k \log n)$. (Note that this step is faster for Alg. 3 which simply needs to pick the k best elements than Alg. 2 which needs to pick the k^2 best subchains (by average score).) Subsequently, these $2k$ chains of size at most k are used in Alg. 3. We use two max-heaps, one to store items whose prerequisites have not been added yet, and one of the items in $boundary$, and one min-heap, to store items whose prerequisites have not been added (and therefore can be deleted). Each of these heaps contain at most k^2 items, and at most k^2 items are deleted. Thus the complexity is $O(k^2 \log k)$. The total complexity is therefore, $O(nk + k \log n + k^2 \log k) \approx O(nk)$, if $n >> k$.

Thus, we see that Alg 2 has the highest runtime complexity, followed by Alg 3 and then Alg 1. In fact, Alg 1 does not even need to look at the entire data. The relationships between the algorithms with respect to complexity (i.e., Alg 2 > Alg 3 > Alg 1) still hold true even when we consider AND, AND-OR and OR graphs. (See Appendix A.5.) This is primarily because Alg. 2 ends up modifying the scores of all other sets that have some overlap with the set under consideration (which, in the worst case, could be all the sets,) while none of the other two algorithms have such an expensive step. Alg. 1 is the least expensive because it performs a search for a local maxima near the start of the DAGs.

Alg. 4 has the same complexity as the second phase of Alg. 2, i.e., $O(k^2 \log k)$, not dependent on n (the number of chains), and thus is more efficient than any of Alg. 1, 2, or 3.

6. EXPERIMENTAL ANALYSIS

We analyze the algorithms for chain graphs in Sec. 6.1, and for other structures in Sec. 6.2. We assess the average performance of the three algorithms of Sec. 3 and the *Merge* algorithm of Sec. 3.6.

We use synthetic data because it allows us to study the behavior of the algorithms on varying parameters. Additionally, publically available datasets do not posess prerequisite information.

6.1 Experiments for Chains

We study the performance of the algorithms by running them on several random graph instances, and collecting the $score$ of the set returned by the algorithms. As we will see in the following, Alg. 2 performs the best out of the three approximation algorithms, and very close to the optimal DP Chains algorithm (from Sec. 3.1). However, we have seen in Sec. 5 that Alg. 2 has the highest complexity of the three algorithms, which makes it less desirable. We therefore implemented the *Merge* algorithm (which has lower complexity than any of the three algorithms), and used the *Merge* algorithm to merge the sets returned by Alg. 1 and Alg. 3 (we refer to this scheme as *merge2*). For comparison, we also implemented algorithm *top2* which picks the set with the higher score out Alg. 1 and Alg. 3 for each input instance. As a baseline, we also implemented the DP Chains algorithm of Sec. 3.1.

6.1.1 Experiment Design

We generated random instances of the graph $\mathcal{G}$ in the following fashion: We set the number of chains 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 desired set.

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 — Breadth-first Pickings (*bf*), Greedy-value Pickings (*greedy*) and Top-down Pickings (*td*), $score$ of *top2* and $score$ of *merge2* to the $score$ returned by the optimal *DP* algorithm.

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* tends to work better than the *Breadth-first Pickings* algorithm. For some additional experiments refer Appendix D.

6.1.2 Variation with Number of items picked

When we vary the size of the desired set k, we find:

- *Alg. 2 is always better than the other two approximation algorithms, and very close to optimal.*
- *Alg. 3 is better than Alg. 1 when k is large compared to n, the number of chains, while the reverse is true when k is small compared to n.*
- *All 3 approximation algorithms of Sec. 3 return sets whose score is at least 90% of the optimal set on average.*
- *top2 does better than both Alg. 1 and Alg. 3.*
- *merge2 does better than top2, and is almost as good as Greedy-value Pickings.*

Fig. 7(a) 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 Fig. 7(a) 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 for that graph instance (averaged over all graph instances). For example, if we set $k = 15$, we find that on average, both Alg. 1 and Alg. 3 return a set that has a score of approximately 95% of the optimal set. On the other hand, on average, Alg. 2 returns a set that has a score of nearly 100% of the optimal score. *top2*, which picks the best set from Alg. 1 and Alg. 3, does significantly better at around 97%. *merge2* performs even better than *top2*, returning a set with score of 99% of the optimal score.

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.

While *top2* has some definite gains over Alg. 1 and Alg. 3, *merge2* does even better than *top2*. This is because it does not simply pick the best set returned by the 2 algorithms of Sec. 3, but combines diverse elements picked by each of the algorithms to get an even better set. In all cases, *merge2* does not depart from more than 1% of the optimal set. *merge2* has low complexity (as seen in Sec. 5), and thus can be used as an effective replacement for Alg. 2.

In all cases, Alg. 2 beats all three algorithms, but as mentioned earlier, is more computationally expensive.

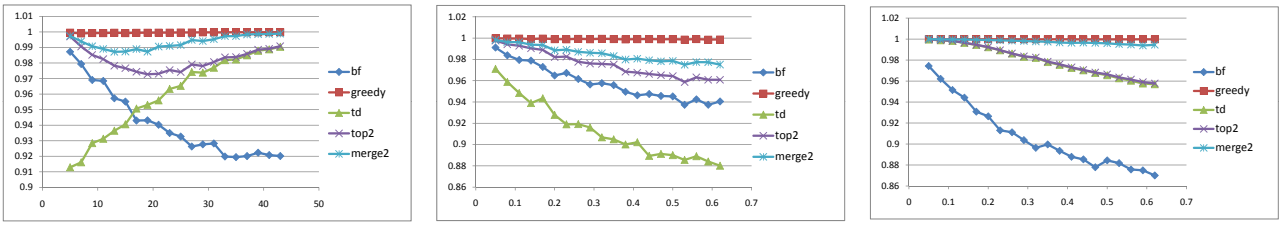

Figure 7: Variation of fraction of score of the set for the algorithms (a) with k (b) with p (c) with p when $k \approx n$

6.1.3 Variation with Number of Long Chains

As we vary p, the probability that a chain is 'long', we have the following results:

- *Alg. 2 still performs the best of all the algorithms*
- *The relative ordering between the scores of the algorithms tend to remain the same as p increases.*
- *merge2 combines sets with poor score to get very close to the optimal set, while top2 does little better than the best of Alg. 1 and Alg. 3.*

In this case, we set $n = 50, d = 5$. 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 optimal set, and plotted the values for $k = 10$ in Fig. 7(b), and for $k = 45$ (i.e., $k \approx n$) in Fig. 7(c).

We find that all algorithms except *greedy* 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. The merge algorithm once again proves invaluable in merging sets with very low scores to give a set that is within 2% of optimal.

6.2 Experiments for Other Structures

For the experiments in this section, in addition to the three algorithms, we also implemented *top2*, which returns set with the better score from the sets returned by Alg. 1 and Alg. 3, and *top3*, which returns the set with the best score from the sets returned by the three algorithms. We compare our algorithms against the score of the best set if no prerequisites are taken into account, called *NoPrereq*.

6.2.1 Experiment Design

In order to test the performance of our algorithms for other structures, we would need to generate random instances of directed AND-OR acyclic graphs. In this paper, we test the algorithms on a collection of balanced trees, which are easier to generate randomly. Testing the algorithms on truly random AND-OR DAGs is left for future work. Note that since each node has a single parent, these balanced trees are AND as well as OR graphs.

Our graph $\mathcal{G}$ therefore consists of a set of n trees, disconnected from each other. With probability p, as in Sec. 6.1.1, a tree has more than one item. We allowed the number of items in each tree to be up to m. We first let the depth d of the tree be a random number between 1 to $d_{max}(< m)$. Subsequently, the branching factor is a random number from 1 to $\lfloor m^{1/d} \rfloor$. The score of each item is exponentially distributed with mean 0.5.

As before, we took several random instances of DAGs as described above and determined the average ratio of the *score* of the set returned by each of the three algorithms (*greedy, td* and *bf*), *top2* and *top3* to the score of the algorithm *NoPrereq*. We plot these ratios on varying various parameters.

6.2.2 Results

We have the following results:

- *Alg. 2 performs better than Alg. 1 and 3, and almost the same as top3*
- *When k is small, Alg. 3 performs better than Alg. 1, with the reverse holding true when k is large.*
- *When the number of chains and the number of items to be picked are both similar, Alg. 3 performs almost as well as Alg. 2.*
- *top2 does better than Alg. 2 when the number of items and number of chains are small.*
- *On varying the number of chains, all algorithms perform similarly.*
- *All algorithms are within 20% of each other.*

Fig 8(a), Fig 8(b) and Fig. 8(c) show the variation with increasing p from 0.05 to 0.65. The vertical axis shows the score returned by each algorithm as a fraction of the best possible score, averaged over 500 graph instances. For Fig. 8(a) and Fig. 8(b), we set $n = 40$, while for Fig. 8(c), we set $n = 100$. For Fig. 8(a) and Fig. 8(c), we set $k = 20$ while $k = 40$ for Fig. 8(b) The maximum number of items in a chain, m is set to be 10 for all the algorithms. In all three plots, we find that Alg. 2 performs the best among the three algorithms, nearly indistinguishable from *top3*. However, Alg. 3 follows Alg. 2 very closely, with less than 5% difference. For example, on setting $p = 0.4$ in Fig 8(a), on average Alg. 1 returns a set with score 80% of the score of the no-prereq set, while Alg. 2 and Alg. 3 return sets with score 90% and 92% respectively.

In fact, in Fig. 8(b), Alg. 3 performs better than Alg. 2 (even though it is not clearly distinguishable in the plot) when p is small. However, note that when the number of chains are large, Alg. 2 does the best, as is evident from Fig. 8(c). (In this case, Alg. 1 does better than Alg. 3.)

We then tried to examine the variation in performance with increasing number of chains. We set $n = 100$, $m = 10$ and $p = 1$, and repeated the experiment over 500 random graph instances. Fig 9(a) shows the variation with increasing number of items being picked. As is expected, when the items being picked becomes large, Alg. 1 starts performing better than Alg. 3, with two plots intersecting when $k = 60$.

Fig 9(b) shows the variation for the extreme case when both the number of chains and the number of items to be picked are small, i.e., $k = 3, n = 3, m = 3$, repeated over 500 trials. In this case, *top3* and *top2* (indistinguishable in the graph) do better than the rest by almost 1%.

Fig 9(c) shows the variation of the algorithms on increasing the number of chains for $k = 20, p = 1, m = 10$ over 500 trials. As expected, all three algorithms improve when the number of chains increase because they have more shots at picking the best items.

While we have not implemented the merge algorithm for AND-OR graphs, we believe that it will give us a further boost compared to *top2*, thereby narrowing the gap to Alg. 2.

7. RELATED WORK

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

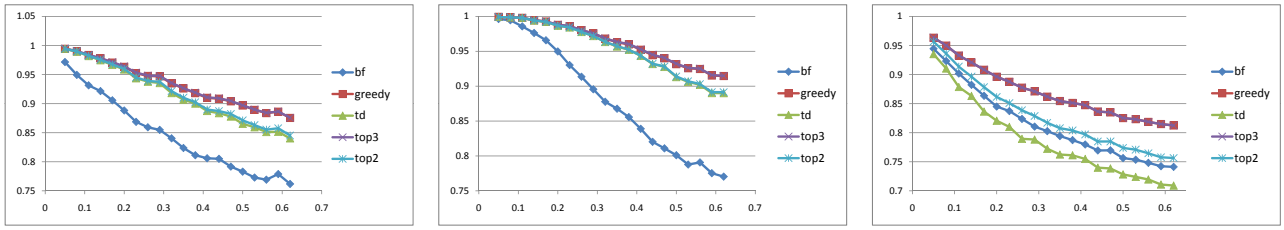

Figure 8: Experiments for AND Graphs: Variation of fraction of score of resulting set for the algorithms with p for (a) small n and small k (b) small n and larger k (c) large n and small k

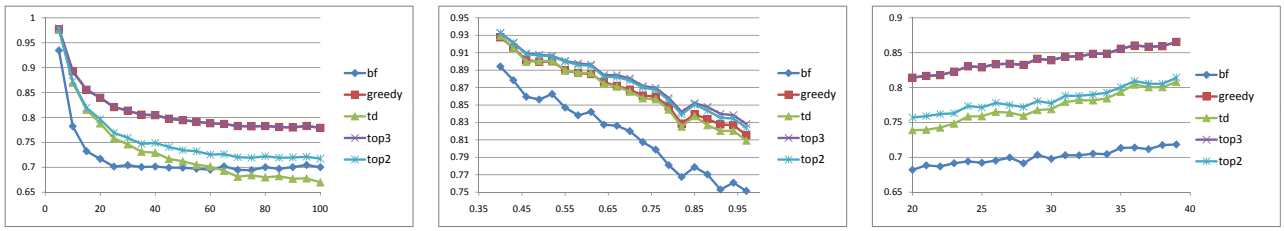

Figure 9: Experiments for AND Graphs: Variation of fraction of score of resulting set for the algorithms (a) with k (b) with p when k and n are both very small (c) with n

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 [17], other 'similar' items that the user liked [16], and other approaches [7]. 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 [9] 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. [19] consider 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.

There has been some recent work on incorporation additional constraints into the recommender systems problem, for example, group recommendations [4, 12] dealing with the problem of recommending items to a group of people with diverse interests, out-of-the-box recommendations and recommending items that are non-obvious and diverse [1, 18], and the work on recommendation of paper assignment to reviewers [8].

Some of the recommendation questions we pose can be written in RQL (Recommendation Query Language) [3], or expressed as constraints [10], however, our aim in this paper is to consider efficient algorithms that solve those recommendation questions, and not posing those questions themselves.

Our problem is an instance of preference-based optimization [6, 5], which considers preferences between subsets of items. However, we leverage the fact that the subsets that we recommend have to satisfy specific constraints, i.e., prerequisites to provide efficient algorithms. Additionally, considering all subsets of items in our case is computationally hard.

8. CONCLUSIONS

As recommender systems are applied to more and more domains [13, 11], it is important to incorporate into our recommendations the constraints introduced by the domain, or more generally, contextual information. If we do not incorporate such constraints we can end up making recommendations that do not make sense to the user, e.g., suggesting courses they are unable to take.

In our case, we considered prerequisite constraints and how they affect the problem of recommendations. We focused on the problem of recommending a set of items with high score, while satisfying prerequisites, for various prerequisite structures. We proved that this problem is NP-Hard for most prerequisite structures and suggested approximate algorithms to solve this problem. For the case of chain graphs, we presented an exact algorithm.

In our experiments, we find that *Greedy-value Pickings* performs exceedingly well. For most practical situations, say for example, we have 100,000 chains to choose 10 items from, and chains with depth around 10, we would prefer using Greedy-value Pickings, which takes $10^5 \times 10$ time steps, i.e., $\frac{1}{10}$ the time as DP, and makes nearly the same (good) recommendations. If we wish to save even more on time, say for example we need to make live recommendations on the web, we might prefer performing both *Top-down Pickings* and *Breadth-first Pickings* and doing a *Merge*. (If we are implementing *Top-down Pickings*, then there is no good reason to not use *Breadth-first Pickings* and *Merge*, since both of these have significantly lower time complexity.) If we simply need to make fast recommendations, we would opt for *Breadth-first Pickings*.

9. REFERENCES

[1] Z. Abbassi, S. Amer-Yahia, L. V. Lakshmanan, S. Vassilvitskii, and C. Yu. Getting recommender systems to think outside the box. In *RecSys '09*.

[2] 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, 2005.

[3] G. Adomavicius, A. Tuzhilin, and R. Zheng. Rql: A Query Language for Recommender Systems. *Technical Report, NYU*.

[4] S. Amer-Yahia, S. B. Roy, A. Chawla, G. Das, and C. Yu. Group recommendation: Semantics and efficiency. *VLDB '09*.

[5] R. I. Brafman and C. Domshlak. Graphically structured value-function compilation. *Artif. Intell.*, 172(2-3):325–349, 2008.

[6] R. I. Brafman, C. Domshlak, S. E. Shimony, and Y. Silver. Preferences over sets. In *In AAAI*, 2006.

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

[8] D. Conry, Y. Koren, and N. Ramakrishnan. Recommender systems for the conference paper assignment problem. In *RecSys '09*.

[9] M. Deshpande and G. Karypis. Item-based top-n recommendation algorithms. *ACM Trans. Inf. Syst.*, 22(1):143–177, 2004.
[10] A. Felfernig and R. Burke. Constraint-based recommender systems: technologies and research issues. In *EC '08*.
[11] L. Grossman. How computers know what we want before we do. *Time Magazine*, July 7, 2010.
[12] A. Jameson and B. Smyth. Recommendation to groups. In *The Adaptive Web: Methods and Strategies of Web Personalization*, 2007.
[13] D. Monroe. Just for you. *Commun. of the ACM*, 52(8), 2009.
[14] A. Parameswaran and H. Garcia-Molina. Recommendations with prerequisites (short paper). In *RecSys '09*.
[15] A. Parameswaran, P. Venetis, and H. Garcia-Molina. Recommendation systems with complex constraints: A courserank perspective. http://ilpubs.stanford.edu:8090/909/.
[16] M. Pazzani and D. Billsus. Content-based recommendation systems. In *The Adaptive Web*, 2007.
[17] B. Sarwar, G. Karypis, J. Konstan, and J. Reidl. Item-based collaborative filtering recommendation algorithms. In *WWW '01*.
[18] M. Zhang and N. Hurley. Avoiding monotony: improving the diversity of recommendation lists. In *RecSys '08*.
[19] C.-N. Ziegler, S. M. McNee, J. A. Konstan, and G. Lausen. Improving recommendation lists through topic diversification. In *WWW '05*.

APPENDIX

A. ALGORITHMS: DETAILS

We consider each algorithm in turn and describe the execution of the algorithm on the example below.

A.1 Illustrative Example

For ease of exposition, we use a chain graph as our 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.

We try to pick a set that satisfies prerequisites of size $k = 4$ using each algorithm.

A.2 Algorithm 1

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.

A.3 Algorithm 2

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 the set corresponding to $a \rightarrow j \rightarrow k$, 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.

A.4 Algorithm 3

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, $A = \{a, j, k, d\}$

A.5 Complexity, revisited

We now examine the worst case complexity for the three heuristic algorithms for general prerequisite structures. Since it is tricky to come up with exact complexity values as was done for chain graphs in Sec. 5, we express complexity in this section in terms of the number of transformations to priority queues. This calculation is general enough to accommodate OR, AND-OR and AND graphs.

All algorithms use at least one priority queue structure: the set $external(A)$ for Alg. 1 and 3, and the set containing subgraphs for Alg. 2. Note that there is another priority queue structure in Alg. 1 and 3, i.e., the $boundary(A)$ 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 $external$ set.

We express our complexity values in terms of three functions $insert(n)$: cost of examining/inserting n individual items into a priority queue and $mov(n)$: cost of a value modification in a priority queue containing n items. Let the total number of items be n in $\mathcal{G}$.

In Alg. 1, each item is inserted and deleted once. Deletion counts as a value modification to 0. Note that at any point, we can restrict $external$ to contain just k items (the others can be ignored, since they will not be picked anyway). Thus modification happens only from a queue of size k. Thus the worst case complexity is $O(insert(n) + n \cdot mov(k))$.

Alg. 2 requires all items with their prerequisites ("best" in the case of OR and AND-OR graphs) to be inserted into the priority queue at the start itself. We cannot prune any part of this set, since it can be potentially part of an optimal solution (after some other sets are picked). Subsequently, we modify (by extracting) up to n sets. And every time we extract a set, we could modify up to n other sets. Thus we have n^2 modifications. Thus, the worst case complexity is $O(insert(n) + n^2 \cdot mov(n))$.

Alg. 3 can maintain two $external$ sets, one corresponding to the top k items, and another corresponding to the items from the start of the graph (whose prerequisites are present). Each item may be inserted once to each set, and deleted from each set. Both the $external$ sets can be restricted to be of k items at all times. Thus the worst case complexity is $O(insert(n)+n\cdot mov(k))$. However, note that checking if an item's prerequisites is present is not an unit operation, and this could be especially costly in the case of OR graphs when any one of many paths could be picked. In order to check prerequisites, we could perform a BFS starting from the item in question, following edges in reverse. This BFS will stop in at most k steps. Thus, we check prerequisites in k steps for at most k items. Since this check is not dependent on n, we ignore it.

Thus, asymptotically, Alg. 1 and Alg. 3 have similar worst case complexity. Both of these have a better worst case complexity than Alg. 2. In practice, Alg. 1 is never likely to examine all n items, and hence runs faster than Alg. 3.

A.6 Worst Case Bounds

We now prove worst case bounds for AND, OR and AND-OR graphs for the three approximation algorithms.

A.6.1 Algorithm 1

For AND, OR and AND-OR graphs, the worst case occurs when we miss out on exploring more of the graph since we only look at $external(A)$. Since at every iteration the algorithm picks the best item from $external(A)$, it must be the case that the first item/s of the "good" connected component that forms part of the optimal solution (that is not picked) must have same or smaller value. However, since the connected component could have an arbitrary number of items (due to an arbitrary branching factor), we lose out on at most γ for each of those $(k-1)$ remaining items in the "good" connected component. Thus the worst case is $(k-1)\gamma$ for AND, OR and AND-OR graphs.

A.6.2 Algorithm 2

For OR graphs, the worst case situation is similar to the one for AND graphs, except that the x items with score α form a chain which then forms the prerequisites for each of the $k-x$ items of score $\alpha+\gamma$. If $d \geq \sqrt{k+1}$, the same bound holds. If $d < \sqrt{k+1}$, then the worst case difference is $\gamma(k+\frac{(d-1)^2-k}{d})$, smaller than that for AND graphs. Note that the value above is equal to that for AND graphs when $d = \sqrt{k+1}$.

For AND-OR graphs, the worst case is at least as bad as that for AND and OR graphs. We list this in Table 1.

A.6.3 Algorithm 3

Recall that Fig. 6 was the worst case situation for chain graphs. To see why, 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 α, which is the smallest such score. This situation is the one described.

We now describe worst case bounds for other prerequisite structures. The worst case arises when the algorithm picks good items that are at the end of "bad" trees. For the case of AND and AND-OR graphs, this could mean that $(k-1)$ bad parents (prerequisites) need to be selected (since a child could have $\geq (k-1)$ parents). Thus for AND and AND-OR graphs, we have a worst case of $(k-1)\gamma$. However, for OR graphs, each good item thus picked could have at most $d-1$ prerequisites, and thus k/d good items will need to be picked, and we lose out on $(d-1)k/d$ "good" items (a total loss of $(d-1)k\gamma/d$).

B. COMBINING ALGORITHM

We now describe the algorithm to combine two sets that satisfy prerequisites in order to give a set that satisfies prerequisites and has score greater than or equal to the score of each of the sets.

The pseudocode listed in **Algorithm 4** and the description below is for AND graphs. We first describe the execution of the algorithm on the example in Appendix A.1.

B.1 Example

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, which 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 $\{j\}$, whose score is updated to 0, since its super-chain has already been selected and removed. Again, the sub-chain in Q_2 is $\{g\}$, whose score is updated to 0 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 $\{a,j,k,d\}$.

B.2 Description of the Combining Algorithm

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 sub-graph m from the min-priority queue (the sub-graph with the smallest average score) (line 15), and try to replace it with a number of sub-graphs from the max-priority queue (line 19-28). The sub-graphs that are used in this process are saved in $used$, while the sub-graphs that are too large are saved in $unused$ (line 22, 24).

If the sum of the scores of the sub-graphs from the max-priority queue exceeds the score of the sub-graph from the min-priority queue, then we replace the sub-graph in A'_1 with the sub-graphs from the max-priority queue (line 30). We also add the items that are $unused$ back to the max-priority queue(line 31)[1], and update the value of the other sets as in algorithm 2 (line 32).

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

Note that if the algorithm returns a set that is different from A_1 and A_2, then the set returned has a larger score than either of A_1 or A_2.

C. EXTENSIONS

In this section, we describe some extensions to our prerequisite structures that can be handled with small modifications to our algorithms. We only describe modifications for the case of chain graphs, but we believe that the modifications for other prerequisite structures will be similar.

[1]The scores of these items may need to be adjusted based on whether or not some portion of them is present in $used$(line 33).

Algorithm 4 $Merge$

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** items $x \in A'_1$ **do**
5: $Y \Leftarrow$ items in A'_1 for whom x is a prerequisite
6: $c \Leftarrow Y \cup \{x\}$
7: insert c into Q_1 with $value : \sum_{a \in c} score(a)/size(c)$
8: **end for**
9: **for all** items $x \in A'_2$ **do**
10: $Y \Leftarrow$ items in A'_2 who are prerequisites of x
11: $c \Leftarrow Y \cup \{x\}$
12: insert c into Q_1 with $value : \sum_{a \in c} score(a)/size(c)$
13: **end for**
14: **while** $Q_1 \neq \emptyset$ **do**
15: $m \Leftarrow pop(Q_1)$ /*subgraph in A'_1 with min avg. score*/
16: $unused \Leftarrow \emptyset; used \Leftarrow \emptyset$
17: $r \Leftarrow size(m)$
18: $Q'_2 \Leftarrow Q_2$
19: **while** $Q'_2 \neq \emptyset \wedge r > 0$ **do**
20: $n \Leftarrow pop(Q'_2)$ /*subgraph in A'_2 with max avg. score*/
21: **if** $size(n) > r$ **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: $r \Leftarrow r - size(n)$
27: **end if**
28: **end while**
29: **if** $size(used) = size(m) \wedge score(used) > score(m)$ **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 subgraph in $used$.
34: **end if**
35: **end while**
36: **return** $A_c \cup A'_1$

Fuzzy Prerequisites

In our current model, if a is a prerequisite of b, and we recommend a and b both, then the contribution of b to the score of the package is $score(b)$. However, if a is not present in the package, then b's contribution is 0, i.e., we do not value b recommended without a. However, there may be cases where we would like to give b a score, between 0 and $score(b)$, even if a is not recommended. Thus, in this case, the prerequisite is not "strict", it is "fuzzy". Thus each item has multiple scores, reflecting whether the item's prerequisites are present or absent.

We let a_m be the item under consideration, and let $a_1 \rightarrow a_2 \rightarrow \ldots \rightarrow a_{m-1}$ be the prerequisite chain of a_m. Since a_m has $m-1$ prerequisites, there would be 2^{m-1} scores for a_m, corresponding to all combinations of its prerequisites being present or absent. However, since $O(2^d)$ (where d is the length of the longest chain) scores per chain would be hard to collect and maintain, we restrict ourselves to items a_i having two scores, α_i and $\alpha_i - \beta_i$ (corresponding to with or without "some" prerequisites respectively — we will elaborate on this shortly). If $\beta_i = 0$, then the item does not need any prerequisites, while if $\beta_i = \alpha_i$ then it has "strict" prerequisites. The value β_i is the amount we penalize the item for not having some prerequisites.

We consider three realistic scenarios: (1) We penalize an item for not having its *immediate* parent in the same recommended set,

i.e., a_m has a score of α_m if a_{m-1} is present in the recommended set, and $\alpha_m - \beta_m$ if not. This scenario occurs in (say) a Drama TV series, when consuming the prerequisite item (episode) lets you know the background in the TV series, and about what happens immediately before the current item (episode). (2) We penalize an item for not having *all* its ancestors in the same recommended set, i.e., if $\{a_1, a_2, ..., a_{m-1}\}$ is present in the recommended set then a_m's score is α_m else it is $\alpha_m - \beta_m$. This situation occurs in (say) a book series (say, the Lord of the Rings or Asimov's Foundation series) where one needs to follow/consume all the previous items to appreciate the current item. (3) We penalize an item for not having *some* of its ancestors present in the recommended set, i.e., if any of $a_1, a_2, ..., a_{m-1}$ are present in the recommended set, then a_m's score is α_m, else it is $\alpha_m - \beta_m$. This situation occurs in (say) a TV Comedy Series (say, Friends or Seinfeld) where one needs to consume/watch at least one episode prior to appreciate the current item.

We now describe the modification to the *Greedy-value Pickings* algorithm (which performs the best out of the three algorithms for the case of chains).

In situation (1), we insert items a_i by themselves (with average score $\alpha_i - \beta_i$) and also all sub-chains $a_j \rightarrow a_{j+1} \rightarrow \ldots \rightarrow a_i, 1 \leq j < i \leq d$ into the max-priority queue (a sub-chain $a_j \rightarrow \ldots \rightarrow a_{i-1} \rightarrow a_i$ has an average score of $(\alpha_i + \alpha_{i-1} + \ldots + \alpha_j - \beta_j)/(i-j+1)$). If the length of the longest chain is d, then there are $O(d^2)$ sub-chains inserted into the max-priority queue per chain. In addition, when a sub-chain (or a singleton item) is selected by the *Greedy-value* Algorithm, all other sets that contain elements from the selected set will need to have their scores updated (at most $O(d^2)$ such sets are present in the queue).

In situation (2), we insert items a_i by themselves (with average score $\alpha_i - \beta_i$) and also all sub-chains containing the item a_i and all of its prerequisites $a_1, \ldots, a_{i-1}$ (with average score $(\alpha_1 + \alpha_2 + \ldots + \alpha_i)/i$) into the max-priority queue. In addition, when a sub-chain (or a singleton item) is selected, all other sets that are part of the same chain will need to have their scores updated. There are at most $2d$ such sets.

In situation (3), we insert items a_i by themselves (with average score $\alpha_i - \beta_i$) and also all sub-chains containing the item a_i and one of its prerequisites a_j (with average score $(\alpha_i + \alpha_j - \beta_j)/2$) into the max-priority queue, a total of $O(d^2)$ elements per chain. In addition, when a sub-chain (or a singleton item) is selected, all other sets that are part of the same chain will have their scores updated.

General Scoring functions

The algorithms given in Sections 3 and 4 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., $score(a, A) < score(b, A)$ (here $score$ takes two arguments, an item and a set). We could also define $score$ to operate on a set, in which case $score(\{a\} \cup A) < score(\{b\} \cup A)$.

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.

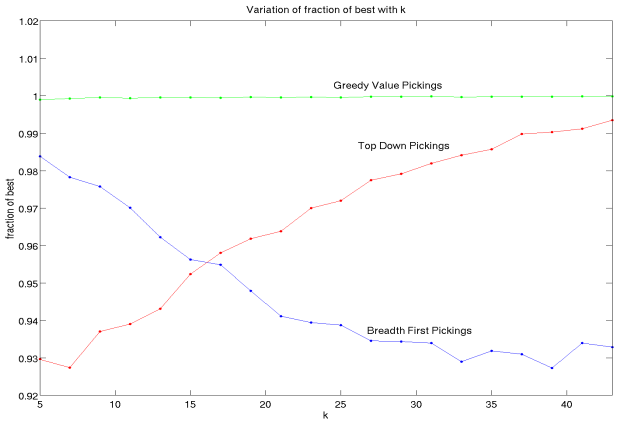

Figure 10: Variation of Fraction of score of the best set for the 3 algorithms with k for the case when the length of the chains is sampled from an exponential distribution

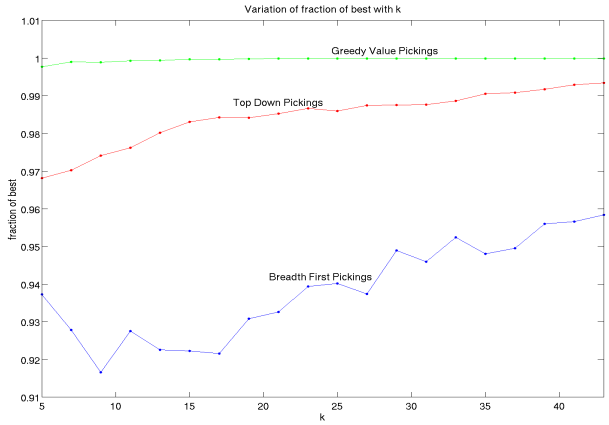

Figure 11: Variation of Fraction of score of the best set for the 3 algorithms with k for the zipf distribution

For Alg. 1, we would recompute $score$ for each item a (i.e., $score(\{a\} \cup A)$) after line 3, for the addition of each item. The pair a, b, where b replaces a in A is chosen such that $score(A \cup \{b\} - \{a\})$ is maximum. For Alg. 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 $(score(A \cup C) - score(A))/|C|$. For Alg. 3, the item a from A whose prerequisites are added first is that for which $score(A - \{a\})$ is smallest. The item in line 12 is the one such that $score(A' - \{a\})$ is the maximum. The item a chosen in line 16 is the i for which $score(A - \{a\} \cup \{i\})$ is maximum. Scores will have to be recomputed after line 18 as well.

D. EXPERIMENTS

We now describe some additional experiments for Chain graphs. For the subsequent experiments, we plot the results of the 3 algorithms of Sec. 3 on varying various parameters.

We obtain similar results when repeating the experiment in Sec. 6.1.2 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 Fig. 10 for details.

In Fig. 11, 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). Since the Zipfian distribution usually has a few outliers with large *score*, Alg. 3 tends to work better than Alg. 1.