Recommendation Systems with Complex Constraints: A Course Recommendation Perspective

ADITYA PARAMESWARAN, PETROS VENETIS and HECTOR GARCIA-MOLINA,
Stanford University

We study the problem of making recommendations when the objects to be recommended must also satisfy constraints or requirements. In particular, we focus on course recommendations: the courses taken by a student must satisfy requirements (e.g., take 2 out of a set of 5 math courses) in order for the student to graduate. Our work is done in the context of the CourseRank system, used by students to plan their academic program at Stanford University. Our goal is to recommend to these students courses that not only help satisfy constraints, but that are also desirable (e.g., popular or taken by similar students). We develop increasingly expressive models for course requirements, and present a variety of schemes for both checking if the requirements are satisfied, and for making recommendations that take into account the requirements. We show that some types of requirements are inherently expensive to check, and we present exact as well as heuristic techniques for those cases. Although our work is specific to course requirements, it provides insights into the design of recommendation systems in the presence of complex constraints found in other applications.

Categories and Subject Descriptors: H.4 [**Information Systems Applications**]: Miscellaneous; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Information Filtering*

General Terms: Algorithms, Experimentation, Theory

Additional Key Words and Phrases: complex constraints, package recommendations, recommender systems

ACM Reference Format:
ACM Trans. Inf. Syst. V, N, Article A (January YYYY), 32 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

In traditional applications for recommender systems, the objects that are recommended do not need to satisfy constraints or requirements. For example, customers are not obliged to read certain sets of books, and the recommended books do not need to satisfy constraints. Thus, traditional recommender systems focus on coming up with a score for each target (e.g., book or movie) independently, based on indicators such as popularity, target objects read or bought by similar people, and so on. Each target is recommended independently of other targets, and there is no need to check if the recommended targets satisfy constraints or requirements.

However, there are important applications where target objects do have constraints or requirements. For example, some medical treatments cannot be given concurrently. Or a computer can be configured with up to two optical storage units from a set of choices (CD writer, DVD reader, BluRay player, etc.). Or a holiday travel plan must take into account the total budget. Thus, when we make recommendations we would like to take into account such constraints, in addition to the traditional indicators.

Recommendation of an *individual* item to satisfy constraints is not a particularly hard problem: this boils down to an extra pre-recommendation filtering step to prune out those items that do not satisfy constraints. For example, Yelp (`www.yelp.com`), which offers restaurant recommendations, offers users the option to filter recommendations based on cuisine, distance from a given point, how expensive it is, and so on. The harder problem, instead, is that of *set recommendations*, recommending a set of items that satisfy several constraints. Apart from travel package recommendations, which are done by some travel websites such as Expedia (`www.expedia.com`), very few existing real-world systems integrate traditional scoring strategies with set recommendations and hard requirements or constraints.

In this paper we study set recommendations under constraints or requirements. We do our work in the context of the CourseRank project. As we will discuss in the next section, CourseRank is a system developed at Stanford for evaluating courses and planning academic programs, and is in use at over 170 universities today. Working on a particular application (course recommendations) grounds our research, and lets us provide solutions for real constraints and real students. Furthermore, as we will see, course requirements are especially rich, so we can explore different types of requirements and solutions. As a matter of fact, one of the main challenges we address in our paper is the modeling of complex academic requirements, a problem that as far as we know has not been formally studied.

The area we focus on, academic software, is in itself a multi-billion dollar business and a major application area for information management systems. Blackboard, just one of the many companies in the domain of education software, has a market cap of 940 million dollars (March 14, 2009). There are over 6000 universities in the USA alone, with over 15M college students, most of whom use software to track courses that they take. Several companies, including Red Lantern (DARS), Jenzabar, Datatel, Sungard (SCT Banner & CAPP), Conclusive Systems, and PeopleSoft (Oracle) have products for course planning (they perform requirement checking but not recommendations). In spite of the importance of this area, we could not find scientific literature on course and requirements tracking and recommendations.

Given our experience with CourseRank, and our own experience as students or academic advisors, we see a strong need for recommendations that take into account requirements. For instance, the deployed CourseRank currently offers a primitive requirements check for the five most popular majors (including Computer Science) at Stanford. In the January 1 to March 13 (2009) period, 33% of the Computer Science students who logged onto CourseRank checked the requirements page at least once. This activity indicates a significant interest by students in checking what portion of their requirements have been met. CourseRank already offers a separate course recommendation service which is also popular with students, but the recommendations do not take into account requirements. It is clear to all the academic advisors we work with that the system would be more useful if we recommend not just "interesting" courses, but "interesting" courses that help students graduate! The academic requirements at most universities are complex enough that students often have a hard time identifying good ways to complete their course requirements (and often just forget to explicitly check the requirements).

Our course recommendation approach is complementary to that of traditional recommendation algorithms; traditional algorithms provide a *score* that indicates how desirable the target is for a user. We use these *scores* and the requirements and constraints to create the optimal set recommendation for the user. Not only does this set satisfy requirements, but is also the best possible such set to do so. (Note that any existing recommendation algorithm can be used to provide the *score*. The focus of this work is on integrating scores provided by existing recommendation algorithms with requirements and constraints.) Even though the specific requirements and constraints may be different in other domains, we believe our approach of integrating requirements and *scores* would provide useful insights into creating recommender systems for those domains.

In summary, our contributions are as follows:

— We present the first algorithmic study of a real set recommendation scenario with complex requirements or constraints.
— We first isolate a *core requirement* model that captures the essential constraints in Section 4 and in Section 4.1 and 4.2, we describe efficient methods to check these

requirements and provide recommendations to satisfy the requirements for this core model. We extend these strategies to make recommendations while integrating traditional *scores* with requirements or constraints.
— We incorporate additional constraints occurring in practice in this core model in Sections 5 and 6. We prove that making recommendations under this model is intractable in Section 7 and we provide an exact solution based on integer linear programming in Section 8.
— We present approximate schemes for checking and recommendations, for cases where the exact schemes may be expensive in Section 9.
— We present other domains (other than course recommendations) for which our complex constraints may also be applicable in Section 10.
— We test both our approximate and exact schemes for recommendation on a dataset of student records in Section 11. Our results show that our requirements-based recommendations substantially outperform traditional recommendations.

2. THE COURSERANK FRAMEWORK

Our work on requirement checking and course recommendations is done in the context of CourseRank (`www.courserank.com`), a social tool we have developed in the InfoLab at Stanford University. CourseRank displays official university information and statistics, such as bulletin course descriptions, grade distributions, and results of official course evaluations, as well as unofficial information, such as user ratings, comments, questions and answers. Students can search for classes, rate and give comments, and receive (simple) course recommendations.

In addition, students enter manually (or load from their transcript) the courses they have taken (and grades) into a planner tool. They can also enter courses they plan to take, check for time conflicts, and see which other students are planning to take those courses.

There is also an interface for faculty and administrators (currently being tested). In particular, department administrators are able to enter requirements for their major(s). Before we developed the model for requirements and the interface, we studied the requirements of many departments (at Stanford and at other universities). The model(s) we present in the following sections are synthesized from this experience. Although we do not discuss the interface, it is based on the formal requirements model we will present.

Soon after its launch, CourseRank became quite popular at Stanford, and virtually all undergraduates (and a good number of graduate students) were using it on a regular basis. In 2009, CourseRank was spun out as a startup to service other universities, and in 2010 it was acquired by Chegg (`www.chegg.com`) (but still run as an independent unit). Today (October 2010), CourseRank services over 170 universities (see `www.courserank.com/schools.php`), and has about 100,000 registered users and growing rapidly.

3. COURSE REQUIREMENTS

For a given major (or department), e.g., Computer Science or History, the course requirements can be expressed as a conjunction of *sub-requirements*. For example, the Computer Science (CS) Department at Stanford University specifies course requirements as a conjunction of 20 sub-requirements, e.g., theory, databases and systems sub-requirements. The course requirements for a student to graduate in a given major can be abstractly expressed as a formula $\mathcal{R}$ of the form $R_1 \wedge R_2 \wedge \ldots \wedge R_m$. Clauses R_i in $\mathcal{R}$ are sub-requirements. We represent the courses the student has taken by $\mathcal{C}$, the set of all courses offered by the university by $\mathcal{U}$.

An important point to note about course requirements is that a course typically cannot be used to satisfy multiple sub-requirements at the same time. For example, CS156 (Automata) can be used to fulfill the math sub-requirement or the theoretical CS sub-requirement, but not both. (In traditional constraint satisfaction [Russell and Norvig 2003], a "taken" variable can be used to satisfy all constraints.) Thus, to check if a student has satisfied her course requirements, we take courses from $\mathcal{C}$, and *assign* each of them to exactly one sub-requirement R_i. A course assigned to a sub-requirement cannot be assigned to any other sub-requirement. By assigning sufficiently many courses to a sub-requirement, a sub-requirement can be *fulfilled* or *satisfied*.

Given a set $\mathcal{R}$ of course requirements stipulated by the University for various majors, and given the set of courses $\mathcal{C}$ taken by a student, the aim is to check if the student has satisfied the course requirements, subject to certain constraints.

In addition to simply checking requirements, we also want to provide course recommendations. Intuitively, our goal will be to recommend a "small and desirable" set of courses from $\mathcal{U} - \mathcal{C}$ that help satisfy the requirements. Desirability of a course will be determined by computing a score using traditional recommendation schemes.

4. THE CORE MODEL

We now describe a model for course requirements that captures the most common features for course requirements in universities.

Let each sub-requirement R_i be of the following form:

`take` k_i `from` $\mathcal{S}_i$,

where $\mathcal{S}_i$ is a set of courses in our context, and $k_i > 0$. Note that $\mathcal{S}_i \cap \mathcal{S}_j$ need not be empty, i.e., there could be courses that are common to two sub-requirements as well, e.g., the database systems principles course could be both in the theory and the systems sub-requirements. However, a given course can be assigned to only one sub-requirement.

Even though this model is simple, it captures a lot of the functionality of course requirements appearing in Stanford University (across all departments) and other universities. In particular, this model can handle compulsory courses and disjunction of courses. Typically, each major has some compulsory courses — for example, electrical engineering may have a compulsory electric circuits class. The requirement for taking compulsory courses can be easily expressed as a sub-requirement of the type shown above (if the compulsory courses are $c_1, c_2, \ldots, c_k$): `take` k `from` $\{c_1, c_2, \ldots, c_k\}$. We can also express disjunction of a set of courses $c_1, c_2, \ldots, c_k$ as a sub-requirement: `take` 1 `from` $\{c_1, c_2, \ldots, c_k\}$. In addition, this model is relevant for requirements of other kinds, e.g., hardware requirements for software projects: project i requires k_i dedicated machines from a set of machines.

We use this model as a starting point for requirement checking and recommendations. We discuss generalizations to this model in Section 6.

Consider the following example of the core model of course requirements: In order to graduate with a major in CS one has to fulfill several sub-requirements. We restrict ourselves to a subset of the sub-requirements: $R_1 \wedge R_2 \wedge R_3$, where R_1 stands for the theory sub-requirement, R_2 stands for the databases sub-requirement and R_3 stands for the systems sub-requirement.

Also assume that a student can take courses from the set $\{a,\ p,\ d,\ i,\ o\}$, where a stands for an introductory course in algorithms, p stands for a database system principles course, d stands for an introductory course to databases, i stands for a database system implementation course and o stands for an operating system course. Then, the sub-requirements could be the following:

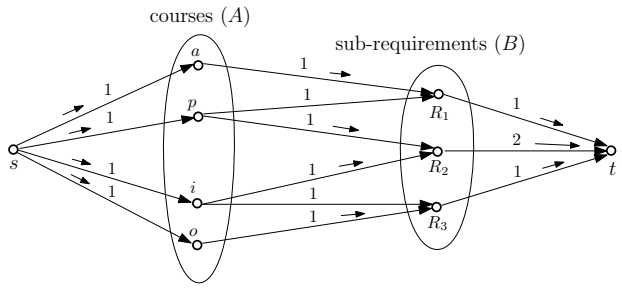


Fig. 1. Core model flow $\mathcal{G}$ for Alice, numbers indicate maximum capacities. The arrows above the edges indicate the output flows, the longer arrow from R_2 to t is of magnitude 2, rest are of magnitude 1.

R_1: `take 1 from` $\{a, p\}$
R_2: `take 2 from` $\{p, d, i\}$
R_3: `take 1 from` $\{i, o\}$

Further, assume that student Alice has taken courses $\{a, p, i, o\}$, and Bob has taken courses $\{p, d, o\}$. We use this example in subsequent sections.

4.1. Flow Algorithm for Requirement Check

We now wish to check if the courses in $\mathcal{C}$ (i.e., those courses that the student has taken already) can be assigned to various sub-requirements, thereby satisfying all the sub-requirements. We do this check using a max-flow algorithm.

We create a graph $\mathcal{G}$ as follows: Consider a source node s and a target node t. We create two sets A and B corresponding to courses and sub-requirements. We create in A a node corresponding to each course $c \in \mathcal{C}$, and a node in B corresponding to every sub-requirement $R_i \in \mathcal{R}$. We also create *directed* edges:

— We create edges from s to each c in A of maximum capacity 1.
— For each course c in A and each sub-requirement R_j in B that has course $c \in \mathcal{S}_j$, we create an edge from c to R_j, with maximum capacity 1.
— For each j, we create edges from R_j to t, with maximum capacity equal to k_j for R_j.

The corresponding graph for Alice is Figure 1.

We then run the Ford-Fulkerson max-flow algorithm [Ford and Fulkerson 1962] on this graph to find the maximum flow from s to t. Recall that the max-flow algorithm aims to find the maximum flow from a source node to a target node, respecting capacity constraints of various edges in the graph. Note that since all the capacity constraints are integers, the maximum flows are integers on all edges [Tarjan 1983]. If there is a feasible flow of magnitude $\sum_j k_j$, then there is an assignment of courses to sub-requirements such that each sub-requirement is satisfied. Conversely, if there is an assignment of courses to sub-requirements that satisfies the sub-requirements, then we can send one unit of flow across each of those "assignments" (i.e., edges corresponding to the assigning of courses to sub-requirements), giving rise to a feasible flow of magnitude $\sum_j k_j$. Thus we have:

THEOREM 4.1. *There exists an assignment of each course to a sub-requirement that satisfies all the sub-requirements iff there exists a feasible flow in graph $\mathcal{G}$ of magnitude $\sum_j k_j$.*

In the case of Alice as in Figure 1, we would have one unit of flow from s into each of a, p, i, o, and $1, 2, 1$ units of flow from R_1, R_2, R_3 to t respectively. Thus, Alice has satisfied the course requirements.

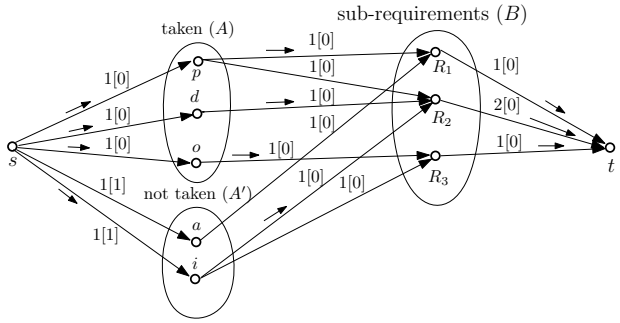


Fig. 2. Core model flow $\mathcal{G}'$ for Bob, numbers are capacities, costs are in square brackets. The arrows above the edges give the output flows, the longer arrow from R_2 to t has flow of 2, rest have a flow of 1.

We discuss complexity of max-flow algorithm in Section 4.3.

4.2. Incorporating Recommendations

Let $\mathcal{N}$, denoting the set of "Not taken" courses, be equal to $\mathcal{U} - \mathcal{C}$. We wish to recommend to the student, a subset of $\mathcal{N}$ that she can take, in order to fulfill all the sub-requirements. This subset should be as small as possible. Thus the problem of recommendations in this context is: *Pick the smallest set of courses $D \subseteq \mathcal{N}$ to recommend, such that all the sub-requirements are fulfilled using $\mathcal{C} \cup D$.*

In order to solve this problem, we modify graph $\mathcal{G}$ into a new graph $\mathcal{G}'$, by adding costs per unit flow to each edge, and by augmenting the graph with nodes and edges. First, the nodes and edges in the graph $\mathcal{G}$ are copied into $\mathcal{G}'$, with the same capacity constraints. We set cost per unit flow to be 0 for all the copied edges.

In addition to the set of courses A (copied from $\mathcal{G}$), we also add a new set of courses A' to $\mathcal{G}'$, such that each node in A' corresponds to a course in $\mathcal{N}$. We connect s to each node in A' with a directed edge of maximum capacity 1, and cost per unit flow 1. We then connect a course node c in A' to sub-requirement R_i in B if the corresponding course c is present in $\mathcal{S}_i$. This edge has maximum capacity 1, and cost 0.

The graph $\mathcal{G}'$ for Bob is given in Figure 2. Since Bob has not taken database implementation i, and algorithms a, those courses appear as nodes in A', with cost per unit flow as 1 (given in square brackets in Figure 2) for the edges from s.

We then run min-cost max-flow algorithm on the graph $\mathcal{G}'$. Recall that the min-cost max-flow algorithm [Tarjan 1983] on a graph gives us the flow of minimum cost among all max-flows. Here cost is defined as the sum over all edges of: the cost per unit flow for a given edge $\times$ the number of units of flow across that edge. It can be shown that the following theorem holds:

THEOREM 4.2. *If the min-cost max-flow algorithm on $\mathcal{G}'$ returns a flow of magnitude $\sum_j k_j$ and cost l, then l is the minimum number of additional courses that need to be taken to satisfy the course requirements. The converse is also true. In addition, the edges from s to A' that have flow > 0, correspond to one set of courses in A' that can be taken in order to satisfy the course requirements.*

Note that Bob needs to take some courses from $\mathcal{N}$ in order to fulfill his course requirements. This is easy to see because the sub-requirements need a total of $1+2+1 = 4$ courses, while Bob has taken only 3 courses. He is therefore recommended course i to fulfill his requirement R_2. Thus the flow is of cost 1.

The min-cost max-flow algorithm on $\mathcal{G}'$ gives us a method to find the value of l, the minimum number of courses that need to be taken to fulfill requirements. Now suppose

we have a *score* of taking each course $c \in \mathcal{N}$, expressed as $score(c)$, $0 \leq score(c) < 1$. The score represents the utility or usefulness of taking the course for the given student. This score could depend on many different factors, including records of previous students, popularity, ratings, prerequisites being satisfied, etc. For our purposes, we use a personalized function (according to the student's preferences) giving us how much the student values any given course. This function can use techniques common in recommendation systems [Sarwar et al. 2001; Adomavicius and Tuzhilin 2005; Burke 2002; Pazzani and Billsus 2007; Burke 2000], e.g., collaborative filtering, content based, and hybrid approaches. Since traditional recommendation systems are well understood, we do not try to re-invent good recommendation strategies. We just incorporate into our scheme, via the score function, a measure that works well for traditional recommendations.

Now from all sets $D \subseteq \mathcal{N}, |D| = l$, that can fulfill the requirements, we wish to find the one with the maximum score defined as:

$$score(D) = \sum_{c \in D} score(c) \tag{1}$$

We thus wish to find the set of courses that is the most useful, from all "smallest" sets of courses that can be used to satisfy the requirements. Thus our recommendation problem is: *From all smallest sets of courses $D \subseteq \mathcal{N}$ such that all sub-requirements are fulfilled using $\mathcal{C} \cup D$, find the D with the maximum score, as in Equation 1.*

We solve this problem by applying the min-cost max-flow algorithm on a modified graph $\mathcal{G}''$. We modify the cost of each edge from s to a course c in A' from 1 to $\delta(c)$, where $\delta(c)$ is defined as (recall that $score(\cdot)$ returns a value in $[0, 1)$): $\delta(c) = 1 - score(c)$. Note that $\delta(c) = 1, \forall c \in A'$, gives rise to the algorithm outlined earlier. We have the following theorem:

THEOREM 4.3. *If min-cost max-flow algorithm on $\mathcal{G}''$ returns a feasible flow of magnitude $\sum_j k_j$ with flow of l going into set A', then l is the minimum number of courses that need to be taken to satisfy the course requirements. In addition, the edges from s to A' that have flow > 0, correspond to the set of courses of minimum size in A' that has the largest $\sum score(c)$ that can be taken in order to satisfy the course requirements.*

Returning to our prior example, suppose we find that Bob is interested more in a than in i. Let us further assume that Bob's *score* is 0.9 for introduction to algorithms a and 0.2 for database implementation i. Recall that the min-cost max-flow on graph $\mathcal{G}'$ gave i as a recommendation for Bob. However, running min-cost max-flow algorithm on graph $\mathcal{G}''$ will change the recommendation. Course node a will get a flow of 1 from s instead of i, making a the new recommended course. p and d will be used to fulfill R_2, while a is used to fulfill R_1. This flow will have a minimum cost of $1 - 0.9 = 0.1$.

In our algorithm above, we return the set D with the highest $score$ (as in Equation 1) among all sets of smallest size. One could instead consider optimizing over a (weighted) combination of score and size, for example, $\alpha \cdot score(D) - (1 - \alpha) \cdot size(D), 0 \leq \alpha \leq 1$. However, given the nature of the requirements model, the optimal solution of the new objective function will contain, as a subset, the set D of smallest size that satisfies all the sub-requirements and has the highest score (or another subset, with same size and score). Hence the objective we maximize above subsumes other objective functions combining score and size that we could consider.

4.3. Complexity

If each course appears in r sub-requirements, and if there are m sub-requirements, then the complexity of the Ford-Fulkerson max-flow algorithm [Ford and Fulkerson 1962] for checking course requirements is: $O((|\mathcal{C}| \cdot r + m) \cdot \sum_j k_j)$.

Additionally, the complexity of the Ford-Fulkerson version of the min-cost max-flow algorithm [Tarjan 1983] for course recommendations is: $O((|\mathcal{U}| \cdot r + m)^2 \cdot \sum_j k_j)$.

4.3.1. Requirement Checking. There are two well-studied algorithms to solve the max-flow problem in PTIME, the Ford-Fulkerson [Ford and Fulkerson 1962] algorithm and the Edmonds-Karp [Edmonds and Karp 1972] algorithm. Generally Edmonds-Karp gives better asymptotic results. In Section 4.1, we wish to find a feasible flow of at most $\sum_j k_j$, which for our problem can be considered to be a small value (no more than 50). Note also that all the capacity constraints for the edges are integers. In this case, Ford-Fulkerson is a faster algorithm. Its cost will be $O(E \cdot \sum_j k_j)$, where E is the number of edges in graph $\mathcal{G}$. E is calculated as follows:

— Node s connects to all the courses in $\mathcal{C}$, giving us $|\mathcal{C}|$ edges.
— The number of edges connecting courses in $\mathcal{C}$ to sub-requirements is $|\mathcal{S}_1| + |\mathcal{S}_2| + \ldots + \left|\mathcal{S}_{|\mathcal{R}|}\right|$.
— The number of edges between sub-requirements and node t is $|\mathcal{R}|$, i.e., the number of sub-requirements we are examining.

Thus, the total asymptotic complexity of our algorithm is:

$$O\left(\left(|\mathcal{C}| + |\mathcal{S}_1| + |\mathcal{S}_2| + \ldots + \left|\mathcal{S}_{|\mathcal{R}|}\right| + |\mathcal{R}|\right) \cdot \sum_j k_j\right).$$

If each course appears in at most r sub-requirements, then $|\mathcal{S}_1|+|\mathcal{S}_2|+\ldots+\left|\mathcal{S}_{|\mathcal{R}|}\right| \leq r \cdot |\mathcal{C}|$ and thus the complexity will be $O((|\mathcal{C}| \cdot r + |\mathcal{R}|) \cdot \sum_j k_j)$.

4.3.2. Recommendations. The problem of min-cost max-flow [Tarjan 1983] is also well studied and there are a number of algorithms in PTIME. The algorithm used most often is a variation of the Ford-Fulkerson algorithm, which has a cost of $O(\sum_j k_j \cdot E \cdot V)$, where E is the number of edges in the graph and V is the number of nodes in the graph. We calculated E in Section 4.3.1. (However, we replace $\mathcal{C}$ in the expression with $\mathcal{U}$.) We also have: $V = 2 + |\mathcal{U}| + |\mathcal{R}|$. Thus, the total complexity of this algorithm is $O(\sum_j k_j \cdot E \cdot V)$, for the values of E and V calculated above.

5. VARIATIONS OF THE CORE MODEL

We now describe variations of the core model of requirements and other recommendation problems that can be solved exactly using modifications of the flow algorithm. Later on, in Section 6, we describe modifications to the core model that cannot be exactly solved using flow algorithms.

5.1. Range Requirements

So far, we have dealt with "fixed" or "strict" constraints, where we do not have any flexibility when it comes to dealing with constraints. Course requirements contain constraints of this form. However, in other domains the constraints may not need a fixed number of items and may instead be satisfied by items in a certain permissible range. To support ranges, constraints of the core model can be converted into constraints of the following form: `take` a-`to`-b `from` $\{c_1, c_2, \ldots, c_k\}$, i.e., take x items from the set, where $a \leq x \leq b$. We could also have a global constraint that the total number of items chosen in all sub-requirements is greater than or equal to r (If this constraint does not exist, then we set $r = \sum a$, sum of the lower bounds of the number of items for each sub-requirement.).

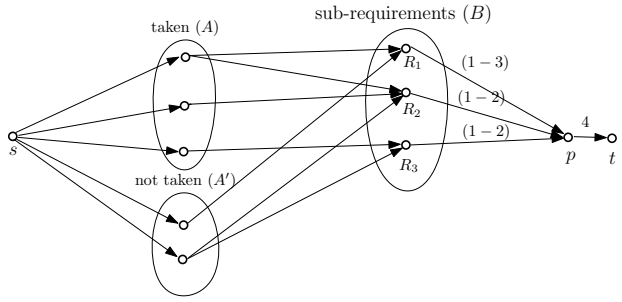


Fig. 3. An example of how range constraints can be used in the core model. All other costs/capacities are the same as before.

For the "range" version of the core model above, we have a straightforward generalization of the flow algorithm (when flow has upper and lower bounds on each edge [Tarjan 1983]) to perform recommendations. An example of how this generalization is done for 3 sub-requirements is shown in Figure 3. We create a pseudo-target node p, and connect the sub-requirement nodes R_i to p instead of t, with both upper and lower bounds on capacities. For example, if R_1 is of the form `take 1-to-3 items from a set`, then we use 1 as the lower bound and 3 as the upper bound on the flow on the edge from R_1 to p, as shown in the figure. In addition, we have a directed edge from p to t, the target node. If there is a global constraint that at least 4 items need to be taken in total (i.e., $r = 4$), then we set 4 as the upper bound on the flow along the edge p from t. Now, on running the min-cost max-flow algorithm, if we do not obtain 4 units of flow on the edge from p to t, then there is no way to recommend items to satisfy constraints. Else the set of items recommended is the best set possible. Note that all other capacities and costs in the graph $\mathcal{G}$ are as before.

5.2. Multiple Recommendations

Sometimes, we may wish to offer other sets of courses as recommendations, in order to present to the student, a variety of options that could be used to satisfy the course requirements.

The naive approach is to do the following: Consider the results of running the min-cost max-flow algorithm on $\mathcal{G}''$. Let the set D correspond to the courses from A' that have non-zero flow in $\mathcal{G}''$. We set the $score$ of a given course in D to be $-\infty$, and re-run the min-cost max-flow algorithm, giving rise to a new set D' (if there exists a feasible flow of magnitude $\sum_j k_j$). We can re-run the algorithm in the manner above for each course in D (by setting the $score$ of the corresponding edge to $-\infty$) and if more recommendations are required, for pairs or triples of courses in D. We then obtain multiple feasible sets of courses, all of which can be used to satisfy the course requirements. In order to discard some "bad" feasible sets of courses recommended by the algorithm, we set an appropriate threshold value t. If some of these sets D' of courses have $\sum_{c \in D'} score(c) < t$, then they are discarded, and not displayed to the student.

These feasible sets can then be presented to the student (or the union of the courses in the various sets can be shown) and the student can pick her best choice of courses. For any set of courses that the student picks, we re-run the above algorithm to check if her choices can satisfy the course requirements, and we can suggest additional courses.

Instead, we can use Lawler's algorithm [Lawler 1972] to generate multiple ranked sets to be presented to the user. The algorithm generates sets in decreasing order of score by adding additional constraints and solving the optimization problem in those cases.

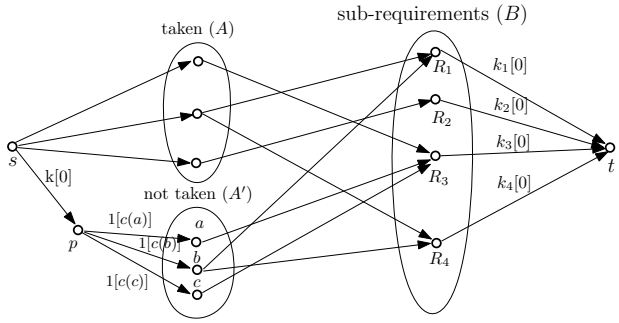


Fig. 4. An example of how course planning for a quarter would be done for a given student using a modification of graph $\mathcal{G}''$. The value k is the upper bound on the number of courses to be taken for the quarter. The node p is the pseudo-node. The cost function c returns $1 - score$. All other edges/costs are as before.

5.3. Maximum Score vs. Minimum Courses

There are cases where a student would like to be recommended a set of courses of size k, $k > l$, where l is the minimum number of courses that needs to be taken to satisfy all the sub-requirements.

We enable this by selecting the set of courses of size k, that while satisfying the sub-requirements, also has best possible score among all sets of courses of size k. We do this in two stages:

— Run the min-cost max-flow algorithm on $\mathcal{G}''$ to obtain the recommended set (of size l) of courses with the best score among all sets of minimum size that fulfill the course requirements.
— Augment this set with the best $k - l$ courses, in decreasing order of *score* that have not already been recommended, in order to get a set of size k.

This procedure enables us to pick the best k courses to recommend to a student.

5.4. Planning Over Multiple Quarters

Typically, students take courses over multiple quarters (or semesters), and would like to plan the courses that they take as spread out over these quarters. The aim in this case is to not just satisfy course requirements (eventually), but also take courses that have a good *score*. Note that some courses are offered only in some quarters. The courses may also have prerequisites, i.e., some condition that the student needs to satisfy in order to be eligible to take the course.

We consider prerequisites as an independent problem in [Parameswaran et al. 2010]. Since the problem of recommending a package of courses with prerequisites is NP-hard, we only suggest an approximate heuristic approach to course planning in this section.

Returning to our example of Section 4: let Bob not have taken course d. In this case, Bob can take any of the pairs $\{a, i\}$, $\{d, i\}$, and $\{a, d\}$ in order to fulfill the three sub-requirements. Now let d be a prerequisite of i, i.e., i cannot be taken by a student unless he has taken d in a previous quarter. In this case, Bob cannot take $\{a, i\}$. He can take $\{a, d\}$ in the same quarter, if they are offered, or he can take d followed by i in successive quarters, assuming they are offered in those quarters.

The above example makes it clear that enabling course planning is important for our application. We wish to enable course planning for students over multiple quarters by making flexible recommendations that work towards satisfaction of course requirements.

Let the current set of courses taken by the student be $\mathcal{C}$. We first run the min-cost max-flow algorithm on graph $\mathcal{G}'$ to find out the minimum number of courses required for the student to graduate = l. Let the current quarter number be q. Let the student specify the number of courses that she wishes to take per quarter[1] as k. We define graph $\mathcal{G}'''$ to be $\mathcal{G}''$ with the following modification: instead of having edges from s to A', we create a pseudo-node p, which connects to nodes in A' with the same costs/flow capacities as the edges from s to nodes in A' in graph $\mathcal{G}''$. We then connect s to p, with an edge of maximum flow capacity k, and cost 0.

For an example as to how graph $\mathcal{G}'''$ would be constructed for a student who has to fulfill 4 sub-requirements, refer Figure 4. Note that the pseudo node p is connected to the courses in A', instead of s. For course planning starting from quarter q, we do the following:

(1) We run min-cost max-flow algorithm on graph $\mathcal{G}'''$, restricting the courses in A' to those offered in quarter q, and whose prerequisites are satisfied using courses in A.
(2) We offer multiple sets of recommendations for quarter q as in Section 5.2, one of which (D) is picked by the student. If flow into node p is less than k, we augment the set of courses recommended with additional courses that have a high $score$ as in Section 5.3.
(3) Update $\mathcal{C} \leftarrow \mathcal{C} \cup D$, $q \leftarrow q + 1$, and return to step 1.

At the end of the 16 quarters (assuming 4 years of an undergraduate degree), when we run the min-cost max-flow on the new graph, we need to ensure that flow into $A' = 0$, or equivalently, cost $= 0$ (since $\forall c, score(c) < 1$, there is always a cost of flowing into the set A').

It may be the case that our recommendations may not allow the student to graduate in the 16 quarters specified, in which case we may need to backtrack our algorithm and recommend that the student take more courses or substitute some important prerequisite courses in earlier quarters. However, we can also assume that courses that are prerequisites of several courses later on will have higher scores, and will therefore be picked early on. In this manner, we can allow students to plan the courses they need to take, in advance.

5.5. Complex Scoring Functions

Equation 1 rewards each course independently of each other, which may sometimes not give rise to good recommendations. For example, if course a and course b have significant overlap in course content, which is undesirable for the student, then we may wish to penalize this in Equation 1. Additionally, if a and b are two courses that go hand in hand, we may wish to reward this in Equation 1. More generally, there may be cases where we may wish to select the smallest set of courses that satisfies the requirements and is optimal for a different function $score'$ defined over sets of courses (instead of a single course).

If instead of a function $score$ that works on a single course, we wish to use a function $score'$ that works on a set of courses, we can do so in the following way.

Firstly, we determine the minimum number of courses required for a student to graduate, i.e., l. Let the cardinality of the set of not-taken courses $\mathcal{N}$ be $|\mathcal{N}|$ and let the number of sub-requirements be m.

We create a modification of graph $\mathcal{G}'$. For each selection of l courses from $\mathcal{N}$, we do the following: we add the selection as nodes to A' and run max-flow algorithm to

[1]This value can be determined as per the students discretion, and based on the number of courses required for graduation. Typically, $k > l$/(quarters left).

see if all the sub-requirements are satisfied. From all those selections of courses that together with the already taken courses (in A) fulfill all the sub-requirements, we keep the selection of courses with the highest score.

Note, however, that the set of courses with the optimal score that fulfills all the sub-requirements might be of size greater than l, in which case our algorithm may not return that set. Normally though, we expect that the scoring function $score'$ will give preference to smaller sets of courses.

The number of different combinations of l courses out of $|\mathcal{N}|$ is $\binom{|\mathcal{N}|}{l}$. Also, running the Ford-Fulkerson max-flow algorithm costs $O((|\mathcal{C}| + l)^2 \cdot m)$. Thus, the total cost of finding the best set that will allow us to fulfill the requirements is $O((|\mathcal{C}|+l)^2 \cdot m \cdot \binom{|\mathcal{N}|}{l})$.

Note that a simplification is possible. If we first run the core model flow algorithm on $\mathcal{G}$, we will obtain the number of choices that remain for each of the sub-requirements, i.e., by how much each of the sub-requirements still need to be fulfilled. We can then run the flow algorithm only on the unfulfilled sub-requirements, a total of $\binom{|\mathcal{N}|}{l}$ times, by letting A contain the selection of l courses from $\mathcal{N}$. This approach gives us a complexity of $O(l^2 \cdot n \cdot \binom{|\mathcal{N}|}{l})$, where n is the number of unfulfilled sub-requirements. Note however, that this may not give us the set of courses with the best $score'$ (but will return a set that satisfies all the sub-requirements), and hence is approximate.

6. THE EXTENDED MODEL

While the core model in the previous sections captures a lot of functionality commonly found in course requirements, there are still other features that arise in course requirements in universities.

6.1. Simple Modifications

In this section we describe modifications to the core model that are essentially reductions, i.e., sub-requirements written using these modifications can be reduced to the core model.

6.1.1. Free Courses. A course that is used to satisfy one sub-requirement may not be used to fulfill another sub-requirement. However, the exceptions are listed in the `free` set for a given sub-requirement. For example, we could have the following sub-requirement.

`take 2 from` $\{a, b, c, d\}$
`free:` $\{a, c\}$

The courses a, c in the `free` clause may be used to satisfy the given sub-requirement, but can also be used to satisfy other sub-requirements, i.e., it does not get "used up" when satisfying the given sub-requirement.

We now reduce the sub-requirement above that contains this feature into one of the core model as in Section 4. We do this by converting the free variables a, c, into entirely new variables a' and c', which are not present in other sub-requirements:

`take 2 from` $\{a', b, c', d\}$

We also ensure that if the student has taken courses a and c, she has taken the new courses a' and c', which have the same characteristics as a and c. For instance, the number of units of a and a' (as will be defined in Section 6.2.2) are equal.

In this manner, we can reduce any sub-requirement with the `free` clause into a sub-requirement of the core model.

6.1.2. Conditions. Sometimes, departments specify course requirements in the following format:

`if [condition] then {sub-requirement}`

The `condition` clause specifies when the sub-requirement is *active*. For example, if the student has already taken a basic calculus course, then she may need to only take 2 math courses from set S to fulfill her math sub-requirement. If not, she may need to take 3 math courses from set R to fulfill her math sub-requirement. In the first case, take 2 from S is the active sub-requirement, and in the second case, take 3 from R is the active sub-requirement.

We only need to check sub-requirements that are active to see if a student has fulfilled the global course requirements $\mathcal{R}$. Typically, for a sub-requirement that has a `condition A`, we expect that there is a sub-requirement that has a `condition (not A)`, in order for all cases to be covered.

Given courses $\mathcal{C}$, it is easy to check which sub-requirements R_i are active in $\mathcal{R}$. If we then restrict our checking to active requirements, then we obtain a set of sub-requirements that can be checked as in Section 4.1.

Note, however, that while making recommendations, it may be beneficial to recommend courses that form part of the `condition` (if they are not already taken), since satisfying the condition may enable us to recommend fewer or better courses overall. In such a case, we can add the courses forming part of the `condition` to the sub-requirement following the `then`, thereby converting the `if...then` into a sub-requirement with `and` clauses (described in Section 6.2.3.)

6.2. Other Modifications

We now describe modifications to the core model that cannot be reduced to instances of the core model. We discuss NP-Hardness of these modifications in Section 7. We then discuss approximate ways of handling these modifications in Section 9, and the ILP method for requirements and recommendations in Section 8.

6.2.1. Forbidden Courses. Universities may offer courses that are either equivalent in content, or contain a large overlap. However, many universities disallow similar courses being used to fulfill the same sub-requirement. We enable this feature by allowing the sub-requirement to contain course pairs that are `forbidden`, i.e., cannot be taken together to satisfy the sub-requirement. For example, we could have a sub-requirement of the form:

`take 3 from` $\{a, b, c, d, e, f, g\}$
`forbidden:` $\{(a, c), (d, e), (f, g)\}$

If we assign the set $\{a, c, e\}$ to the above sub-requirement, we would still not satisfy the sub-requirement, because the pair (a, c) is forbidden, i.e., at most one course of the pair can be used to satisfy the sub-requirement.

We describe how even checking of a single sub-requirement of Section 4 with forbidden pairs can become NP-Hard in Section 7.1.

6.2.2. Unit Constraints. Typically universities want to quantify the amount of work done by a student for each course, and they do so using *units*. Each course $c \in \mathcal{U}$ is assumed to have a fixed integer number of units $u(c)$, corresponding to the amount of work involved, or importance. The `units` clause specifies the minimum number of units required to satisfy the sub-requirement. For example,

`take 1 from` $\{a, b, c, d\}$ `units:`5

Here, the student needs 5 units. If a, b are of 3 units each, and c, d are of 5 units each, then the student can satisfy the sub-requirement by taking both a and b, or by taking either one of c or d.

We show NP-Hardness of the units constraint added to the core model in Section 7.2.

6.2.3. And Clauses. We could also allow conjunctions of courses, in the following way: `take 2 from` $\{ade, bc, f, g\}$. Here, we can take, for example, courses $\{b, c, f\}$ to fulfill the sub-requirement, but not just courses $\{b, f\}$. We would like to enable this functionality because there are many cases where universities count taking multiple simple courses as equivalent to taking one complex course. For example, taking two courses, "data structures" and "algorithms" could be equivalent to taking one course "data structures and algorithms".

We show NP-Hardness of the `and` clauses functionality added to the core model in Section 7.3.

6.2.4. Or Clauses. We disallow disjunctions in the set of courses, because this can be captured independently using `forbidden`. We illustrate this with an example: consider the simplified sub-requirement `take 3 from` $\{a, bc, e|d|f\}$, where $e|d|f$ stands for e or d or f. We can replace this with `take 3 from` $\{a, bc, e, d, f\}$, `forbidden`: $(e, d), (d, f), (e, f)$. We can perform this replacement for any disjunction in the set of courses. We do make one assumption, however, i.e., that an `and` clause containing a course present in a disjunction is not present in the same sub-requirement. Presence of an `and` clause would imply that taking a, or a combination of courses abc is equivalent for the sub-requirement — if so, there is no reason to keep abc. Note however, that another disjunction containing the same course can be present — and we handle each of them independently.

Also note that complicated boolean formulas are not allowed in our model because it can be decomposed into basic sub-formulae. For example, $ab(c|d)$ can be reduced to two items in the set of courses, abc, abd, with forbidden: (c, d).

6.2.5. Nested Constraints. Some departments use nested sub-requirements (i.e., sub-requirements within sub-requirements). For example,

`take 3 from` $\{$`take 2 from` $\{a, d, e\}, a, f, g\}$

This feature could be useful in the following way: If a student in computer science can either pick a theory, systems or database specialization, and then take k courses from that specialization (sets S_t, S_s, S_d respectively), this could be expressed using nested constraints: `take 1 from` $\{$`take` k `from` S_d, `take` k `from` S_t, `take` k `from` $S_s\}$

Note that the same restrictions still apply — a course used to fulfill a nested constraint within a sub-requirement cannot be used to fulfill another nested constraint, or picked as a choice in the sub-requirement. In effect, a course can only be used once. As an example, course a above can either be used to fulfill the nested constraint or be picked as an alternative in the sub-requirement.

We describe how even checking of a single sub-requirement of Section 4 with nesting can become NP-Hard in Section 7.4. Subsequently, we do not consider nested constraints to be part of our general model because they can be flattened or expanded to give a requirement with only and clauses.

6.3. Putting It All Together

Thus, we have the following extended model for a sub-requirement encompassing some of the previous features:

```
condition: /*boolean condition on courses*/
take k from {/*set containing "∧" of courses*/}
forbidden: /*a set of course pairs*/
units: /*a number*/
free: /*a set of courses*/
```

We exclude nested constraints for this model. Note that each sub-requirement is thus of the following form: If some `condition` is satisfied, `take` k `from` a set of courses, where each item in the set of courses could be an `and` clause of courses, such that

— *Work Quantified*: Total units of courses assigned to the sub-requirement is $\geq$ `units`,
— *No Double Count*: A course counts towards only one sub-requirement, with exceptions listed in `free`, or
— *No Overlap*: Courses picked cannot have same content, with conflicting pairs listed in `forbidden`

7. HARDNESS OF THE EXTENDED MODEL

In this section, we provide proofs of hardness of checking of course requirements when we add the functionality of the features listed in Section 6. In each of the following subsections, we take the core model as defined in Section 4, and add a feature from the previous section, and prove its NP-Hardness.

7.1. Hardness of Forbidden Pairs

We will reduce the independent set problem to the checking of sub-requirements containing forbidden pairs of courses. Recall that the independent set problem is the following: given a graph $\mathcal{G} = (V, E)$, is there a set of vertices $\mathcal{X} \subseteq \mathcal{V}$ of size at least k ($|\mathcal{X}| \geq k$) such that no two vertices in $\mathcal{X}$ are connected to each other with an edge in E? Since the independent set problem is NP-Complete, the reduction will imply that checking whether a student satisfies sub-requirements that allow forbidden pairs is NP-Hard. Additionally, let $\mathcal{V} = \{v_1, v_2, \ldots, v_n\}$.

The reduction is the following: we create a sub-requirement R_1. Requirement $\mathcal{R}$ will contain this sub-requirement and no other. Additionally we assume that all courses mentioned in this sub-requirement are taken. This sub-requirement will be of the following form:

`take` k `from` $\{v_1, v_2, \ldots, v_n\}$
`forbidden: all` $\{v_i, v_j\}$, $\{v_i, v_j\} \in \mathcal{E}$

Clearly, this transformation can be done in polynomial time. Note that if the checking of this sub-requirement yields value true, then there is a solution to the independent set problem, by construction. Also, every solution to the independent set problem, would also be a solution to the requirement checking problem that we have. Thus, we have reduced the independent set problem to the problem of checking sub-requirements with forbidden pairs. This implies that adding the functionality of forbidden pairs to sub-requirements makes even the checking of a single sub-requirement NP-Hard.

7.2. Hardness of Minimum Units Criteria

We will prove that if we have a minimum units constraint in our sub-requirements then we obtain an NP-Hard problem. In particular, we will reduce the Partition problem to the problem of checking if sub-requirements with minimum units requirements are fulfilled.

Recall that the NP-Complete Partition problem is the following: We have a set S containing numbers and we want to split it into to sets S_1 and S_2 such that each partition S_i has the same sum of the numbers it contains, i.e. $\sum_{x \in S_1} x = \sum_{y \in S_2} y$. Every instance of the partition problem can be reduced to the checking of course requirements with sub-requirements having a minimum units criterion, in the following manner.

Assume that we are given an instance $S = \{s_1, \ldots, s_n\}$ of the partition problem, with $\sigma = \sum_{s \in S} s$. For each s_i we create a new course c_i with $u(c_i) = s_i$. Then, we create two sub-requirements: R_1 and R_2. These two sub-requirements will look like this:

R_1: `take` 0 `from` $\{c_1, \ldots, c_n\}$, `units=`$\frac{\sigma}{2}$
R_2: `take` 0 `from` $\{c_1, \ldots, c_n\}$, `units=`$\frac{\sigma}{2}$

This transformation can be done in linear time (and thus polynomial) in the size of our input S. We assume that all courses are taken. Note that every solution that we get for the course requirement checking problem is also a solution of the partition problem and the other way round. Recall that one cannot take one course to satisfy more than one sub-requirements, so each course will be assigned to, and be used to satisfy either R_1 or R_2.

Thus, the partition problem is reduced to the course requirement checking problem. Thus, the requirement checking problem becomes NP-Hard when we add the minimum units constraint.

7.3. Hardness of And Clauses

We will use a reduction from the independent set problem to prove that this variation of the problem is NP-Hard as well. We will use the same notation for the independent set problem that we used in Section 7.1.

The reduction is as follows. For an instance $\mathcal{G}$ of the independent set problem, we can create a sub-requirement R_1. This will be the only sub-requirement that $\mathcal{R}$ will contain. For each edge $\{v_i, v_j\}$ of $\mathcal{G}$, we will create a new course $c_{\{i,j\}}$. Now, we will create `and` clauses of courses as in the following:

R_1: `take` k `from` $\{\bigwedge_{i=1} c_{\{i,j\}}, \bigwedge_{i=2} c_{\{i,j\}}, \ldots, \bigwedge_{i=n} c_{\{i,j\}}\}$

This transformation once again can be done in polynomial time of the size of the graph $\mathcal{G}$. We assume that all courses are taken. Every solution to the course requirement check problem above is a solution of the independent set problem, by construction. Also, every solution of the independent set problem is a solution of the checking problem. To see this, observe that every course represents an edge in $\mathcal{G}$. Also, every `and` clause represents a vertex of the graph. The way the sub-requirement is constructed does not allow for two `and` clauses that share a course to be fulfilled at the same time. Thus, the checking of this variation of course requirements is also NP-Hard.

7.4. Hardness of Nesting

Nested constraints make the course requirement checking NP-Hard. We obtain a reduction from the NP-Complete independent set problem. Consider graph $\mathcal{G} = (V, E)$. We create a course corresponding to each node in $\mathcal{V}$, and a nested constraint `take 1 out of` $\{v_1, v_1'\}$ corresponding to each edge (v_1, v_1'). If the required size of the independent set is k, then the reduction is as follows:

`take` k `from` $\{$`take 1 from` $\{v_1, v_1'\}$, ..., `take 1 from` $\{v_k, v_k'\}\}$

Additionally, we assume that all the courses are taken. The output of the requirement check problem in this case is the same as the output of the decision version of the independent set problem. If the sub-requirement above is fulfilled, then there is an independent set of size k. The converse is also true, because if there is an independent set of size k, then the sub-requirement is fulfilled.

Thus, allowing nested sub-requirements makes the course requirement checking problem NP-Hard.

8. THE ILP ALGORITHM

We now describe an Integer Linear Program [Gomory 1958] for checking course requirements and recommending the "best" set of courses for the extended model as in Section 6. Since checking constraints the extended model is itself already NP-Hard, this shows that the making recommendations for the extended model is NP-Complete.

8.1. ILP for Requirement Check

Consider the set of all courses $\mathcal{C}$ that are taken by the student. For each course $c \in \mathcal{C}$, do the following: Let X_c^j be a boolean variable corresponding to whether we count course c towards fulfilling sub-requirement R_j (with value 1 if yes, 0 if no). Clearly, we have the following rules to begin with (apart from the obvious one $\forall i \in \mathcal{C}, R_j \in \mathcal{R} : 0 \leq X_i^j \leq 1$):

$$\forall i \in \mathcal{C} : \sum_{R_j \in \mathcal{R}} X_i^j \leq 1, \tag{2}$$

since a course can count towards only one sub-requirement. However, in the above equations, the variables X_i^j, where course i is `free` in sub-requirement R_j, are removed. We do this so that we do not "use up" a course that is `free` for a sub-requirement.

To handle the number of choices, we do the following. Let k_j denote the minimum number of choices from set $\mathcal{S}_j$ required to fulfill sub-requirement R_j. We then have:

$$\forall R_j \in \mathcal{R} : \sum_{i \in \mathcal{C}} X_i^j \geq k_j \tag{3}$$

However, for the above equation, we do not remove variables X_i^j corresponding to courses i that are `free` in sub-requirement R_j, allowing i to be counted in fulfillment of sub-requirement R_j.

To handle `units`, we do the following. Let m_j denote the minimum units required to fulfill sub-requirement R_j, and let u_i denote the number of units of course i. We have:

$$\forall R_j \in \mathcal{R} : \sum_{i \in \mathcal{C}} u_i X_i^j \geq m_j \tag{4}$$

As above, we do not remove variables corresponding to `free` courses, since such courses count towards fulfillment of `unit` constraints.

Forbidden pairs are represented in the following format — if course i and j form a `forbidden` pair for requirement R_k, then we have

$$\forall R_k \in \mathcal{R}, \forall (i,j) \in F_k, i, j \in \mathcal{C} : X_i^k + X_j^k \leq 1, \tag{5}$$

where F_k lists all the `forbidden` pairs in R_k.

`And` Clauses in the set of courses S_l for a sub-requirement R_l can once again be captured using linear constraints. For now, consider the case that a course can appear in only one `and` clause within the set S_l. Consider the following example: Let courses i, j and k be part of a 3-course `and` clause ijk in the set of courses $\mathcal{S}_l$ within the sub-requirement R_l. We then enforce that $X_i^l = X_j^l = X_k^l$. Additionally, variables X_i^l, X_j^l and X_k^l in Equation 3 are replaced by $X_i^l/3, X_j^l/3$ and $X_k^l/3$ respectively. We add the units corresponding to i, j and k to Equation 4 for R_l, multiplied by X_i^l.

The case of multiple `and` clauses containing the same course is slightly complicated. For example, consider the case where abc and ad are two `and` clauses for the same sub-requirement R_l. In this case, we create two variables corresponding to the assignment of a to l, X_a^l and $\bar{X}_a^l$, the first of which corresponds to the first clause abc and the second to clause ad. We add both of these variables to Equation 2 for course a. Additionally,

we enforce the following: $X_a^l = X_b^l = X_c^l$ and $\bar{X}_a^l = X_d^l$, making sure that if we pick one course in an and clause, we pick all the courses. Variables X_a^l, X_b^l and X_c^l in Equation 3 for R_l are replaced by $X_a^l/3, X_b^l/3$ and $X_c^l/3$ respectively. We also add variables $\bar{X}_a^l/2$ and $X_d^l/2$ in the same equation. We add the units corresponding to a, b and c to Equation 4 for R_l, multiplied by X_a^l, and also units corresponding to a and d multiplied by $\bar{X}_a^l$. In this manner, we handle multiple and clauses.

Thus the exact checking of course requirements can be expressed as an ILP (Integer Linear Program), which is NP-Complete. For this ILP, we do not try to minimize any function, i.e., we only ask if the constraints expressed as an ILP have a feasible solution. A feasible solution would imply that the requirements are satisfied.

If we discard the constraint of the variables having integer values, then the resulting LP (Linear Program) can be solved in PTIME. This solution is only an approximate solution (because non-integer values have no correspondence to real entities). However, an approximate solution can be used as a starting point for an integer solution by suitably rounding up or down the appropriate non-integer values or by suitably introducing other constraints that give rise to integer solutions as in [Gomory 1958].

As noted above, if the LP returns an infeasible value, we know that the original ILP has no solution (since the ILP has strictly more constraints). Hence there are no false negatives, but there may be false positives if we use the solution of the LP for the ILP.

8.2. ILP for Recommendations

We now introduce new variables corresponding to each course in the not-taken set $\mathcal{N}$. For all $i \in \mathcal{N}$, do the following: let X_i^j be a boolean variable corresponding to whether we count course i towards fulfilling sub-requirement R_j (with value 1 if yes, 0 if no). We enforce that $\forall i \in \mathcal{N}, R_j \in \mathcal{R} : 0 \leq X_i^j \leq 1$. In addition, we also change Equations 2, 3, 4 and 5 to include variables in $\mathcal{N}$ as well, i.e., wherever we let $i, j \in \mathcal{C}$ in the above equations, we now let $i, j \in \mathcal{C} \cup \mathcal{N}$. We encode whether a course is recommended by the ILP towards fulfillment of course requirements (or has already been taken) using a new boolean variable Y_c, $c \in \mathcal{U}$ (1 if yes, 0 if no). We have: $\forall c \in \mathcal{U}, \forall R_j \in \mathcal{R}, X_c^j \leq Y_c \leq 1$.

For core recommendations of the form in Section 4.2, we have the following objective for minimization:

$$\sum_{c \in \mathcal{N}} (1 - score(c)) \times Y_c, \tag{6}$$

subject to the same constraints listed above. Once again, solving this ILP will be expensive.

8.3. Supporting more features

There are some other features that we would like our recommendations to handle. For example, we should not recommend to a student two courses that are taught during the same time slots. We now present a list of features that are important for the recommendation phase of our algorithms and also ways in which we can model them as constraints in the ILP. In the following, we will use the boolean variable Z_c^q to denote that the student has taken (or will be recommended to take) course c in quarter q. For these variables we have the constraints for $c \in \mathcal{U} : 0 \leq Z_c^q \leq 1$ and $Y_c = \sum_q Z_c^q$. For $c \in \mathcal{C}$, we set $Z_c^q = 1$ for the appropriate quarter q.

Overlap of courses. Looking at courses, we realized that it is useless for students to take some pairs of courses together (for example, when a student takes a course from the statistics department and one from the mathematics department that have almost

the same course material). Thus, we change the minimization objective Equation 6 to:

$$\mathcal{M} = \sum_{c \in \mathcal{N}} (1 - score(c)) \times Y_c + \sum_{c_1, c_2 \in \mathcal{U}} \alpha_{c_1,c_2} \times O_{c_1,c_2},$$

where $\alpha_{c_1,c_2} \in \mathbb{R}^+$ is a weight for how bad the pair c_1 and c_2 is. Also, we impose constraints on boolean variable O_{c_1,c_2}: $O_{c_1,c_2} \geq Y_{c_1} + Y_{c_2} - 1$ and $O_{c_1,c_2} \geq 0$. This restriction guarantees that we will only penalize taking c_1 and c_2 together; in that case $O_{c_1,c_2} = 1$. In all other cases, O_{c_1,c_2} will get the value 0, since we are minimizing over the formula above and since $O_{c_1,c_2} \geq 0$.

Good pairs of courses. There are some courses that are "good pairs", i.e., the student gains more by taking both of them rather than individually. For example, a student in computer science would gain more by taking both p (principles of databases) and i (implementation of databases), than just one of them. The new objective function to minimize is: $\mathcal{M} + \sum_{c_1,c_2 \in \mathcal{U}} \beta_{c_1,c_2} \times J_{c_1,c_2}$, where $\beta_{c_1,c_2} \in \mathbb{R}^-$ is a weight for how good pair c_1 and c_2 is. Also, we impose the following restrictions on boolean variable J_{c_1,c_2}: $J_{c_1,c_2} \leq (Y_{c_1} + Y_{c_2})/2$ and $0 \leq J_{c_1,c_2} \leq 1$. This restriction guarantees that we will only reward taking c_1 and c_2 together; in that case $J_{c_1,c_2} = 1$. In all other cases, J_{c_1,c_2} will get the value 0, since J_{c_1,c_2} is also a boolean variable.

Prerequisites. In order to take some courses, you must have already taken some other courses, called prerequisites. For example, in order to be eligible to take a course c in quarter q, $(c_1 \vee c_2) \wedge c_3$ may have to be taken before c in a quarter before q. Generally, a CNF (conjunctive normal form) formula describes the prerequisites of any given course:

$$(c_{1,1} \vee c_{1,2} \vee \ldots \vee c_{1,l_1}) \wedge \ldots \wedge (c_{m,1} \vee c_{m,2} \vee \ldots \vee c_{m,l_m})$$

For the general case, we need to incorporate the following inequalities for every course c, every quarter q, and every `or` clause $1 \leq k \leq m$, $(c_{k,1} \vee c_{k,2} \vee \ldots \vee c_{k,l_k})$ in the above CNF for course c: $\sum_{i=1}^{l_k} \sum_{q'=1}^{q-1} Z^{q'}_{c_{k,i}} \geq Z^q_c$.

This equation states that one from $c_{k,1}, \ldots, c_{k,l_k}$ has to be taken in a quarter q' prior to quarter q in which c can be taken. Prerequisites are studied in greater detail in [Parameswaran et al. 2010].

Offered only in a particular quarter q. Some courses are only offered during one quarter q and not at any other time. The way to model that will be: $\sum_{q' \neq q} Z^{q'}_c = 0$.

Total units per quarter bounded. In many cases, students want to take no more than k units in one quarter q. Our model can capture this as: $\sum_{c \in \mathcal{N}} u_c \cdot Z^q_c \leq k$, assuming that u_c is the number of units for which the student can take course c.

Slot clash. There are many courses that are taught during the same time slots and a student cannot take both of them in the same quarter. Let us assume that classes c_1, c_2, …, c_m take place in the same time slots in quarter q. We can express this in our model as: $\sum_{i=1}^{m} Z^q_{c_i} \leq 1$.

8.4. Discussion

Typically, one would expect that personalized recommendations can be computed offline and maintained for each student so that they can be displayed when the student logs on to CourseRank. In this setting, the time taken by the algorithm may not be so critical, and an ILP solution may suffice.

Even though the ILP in practice is NP-Complete, our problems are usually small, and less constrained — e.g., there are few forbidden pairs, there are few instances where the units requirement is a tight constraint. So, even in some practical real-time

settings, ILPs form a viable solution. In particular, as we shall see in Section 11.2 for course recommendations, ILPs run in less than 5 times the amount it takes to run the flow algorithm.

9. FLOW ALGORITHMS FOR THE EXTENDED MODEL

Solving an ILP for checking course requirements and making recommendations as described in the previous section may be expensive. In this section, we describe how we can continue to use the core model of Section 4 to make recommendations for the Extended Model of Section 6. There are two techniques that may be used to utilize the flow algorithms on the extended model. The first involves multiple "runs" of the flow algorithm. This approach is exact, and runs in polynomial time if the number of modifications to the core model (e.g., the number of forbidden pairs, or the number of and clauses) is bounded. The second approach involves the use of heuristics and is necessarily approximate.

9.1. Exact Techniques

We now describe some exact techniques that will allow us to utilize the flow algorithms to make recommendations for the extended model.

9.1.1. Courses Appearing in One Sub-Requirement. We now discuss how to reduce the size of sub-requirement R_j when some of the courses that have already been taken by the student are present only in $\mathcal{S}_j$, and not in any other $\mathcal{S}_i$. We take care of such a course c that is present only in sub-requirement R_j by reducing the number of courses required for sub-requirement R_j by 1, and by decreasing `units` by the number of units of c. For example, consider sub-requirement R_j of the form:

`take 3 from` $\{c_1, c_2, \ldots, c_m\}$
`units:` u,

where c_1 appears only in R_j, and no other sub-requirement, one can create the corresponding sub-requirement:

`take 2 from` $\{c_2, \ldots, c_m\}$
`units:` $u - u(c_1)$

Note that this simplification may yield incorrect results if we have `forbidden` pairs in our sub-requirement. However, the approach we use in Section 9.1.2 will handle this wrinkle.

9.1.2. Bounded Forbidden Pairs. If the total number of forbidden pairs across all sub-requirements is bounded, then there exists a polynomial-time algorithm that solves the checking problem. Let (a_i, b_i) for $i = 1, 2, \ldots, m'$ be all the forbidden pairs that the student has taken. We run the max-flow algorithm using one of two alternatives for the sub-requirement R_j where (a_i, b_i) is a forbidden pair: we either let the sub-requirement R_j have an edge from course a_i but not b_i in the flow graph, or from b_i but not a_i. Each of these options do not violate the forbidden pair rule. We therefore run the flow algorithm $2^{m'}$ times, and the cost of this process is $2^{m'} \times$ the cost of the max-flow algorithm. If for some combination of courses in the graph the student satisfies the requirements, he satisfies his original requirements.

For recommendations, we will need to run the algorithms 2^m times, where m is the number of forbidden pairs appearing across all sub-requirements. If the process recommends some courses for each run, we use the recommendation that maximizes the *score* from among all set recommendations that have smallest cardinality.

9.1.3. Bounded And Clauses. If the total number of `and` clauses that appear in sub-requirements are bounded, then there exists a polynomial algorithm that solves the problem. Let the `and` clauses present in the sub-requirements be $a_i \wedge b_i \wedge c_i \wedge \ldots$ for $i = 1, 2, \ldots, m$ (where m is bounded). Similar to the case of bounded forbidden pairs, we use the following procedure to make recommendations in polynomial time: Let the i^{th} `and` clause be $a \wedge b$. If the student has taken a `and` b, we have two options (for the given `and` clause) in the flow graph: either (i) include $a \wedge b$ in the graph as one course (i.e., collapse a and b into one course) and allow it to have an edge to all the sub-requirements containing the `and` clause $a \wedge b$, and remove nodes a and b from the flow graph or (ii) not include $a \wedge b$ and delete the $a \wedge b$ node, as well as edges to all sub-requirements that mention them in the flow graph. This procedure would take $2^{m'} \times$ time to run max-flow algorithm, where m' is the number of `and` clauses that the student has taken out of the m possible.

Note that there could be multiple `and` clauses that contain the same course: even in this case, we continue to use the same procedure, except that if a node a is removed from the flow graph due to any one of the `and` clauses, then it is never added again.

While making recommendations, we recommend the set (from among all runs) that has the largest *score* from the one that has the smallest cardinality. In this case, we need to run the flow algorithm 2^m times, where m is the total number of `and` clauses.

9.1.4. Connected Components. There are typically few courses that can be assigned to multiple sub-requirements. As a result, the undirected graph of connections between courses and sub-requirements (i.e., a course c is connected to R_j if $c \in \mathcal{S}_j$) is sparse, and can be partitioned into *connected components*. Requirement checking and recommendations for each connected component can be done in parallel. Computing connected components for each major can be done offline.

9.2. Heuristic Techniques

We now describe some heuristics that we can use to check and make recommendations for the extended model.

9.2.1. Minimum Units. We find that in practice, the `units` condition is satisfied whenever the minimum choices (i.e., `take` k_j) condition is satisfied. Thus, it is usually sufficient to ignore the `units` condition until checking using the flow algorithm of Section 4.1 is complete.

At that point, if there are still some sub-requirements whose `units` condition is not satisfied, we do the following: For those sub-requirements R_j, we increase k_j by the minimum number of courses required to fulfill the remaining units[2] and re-run the min-cost max-flow algorithm. If even this approach does not suffice to satisfy the minimum units requirement, then we increase k_j for each of the sub-requirements that are still not fulfilled by 1. We then re-run the flow algorithm. We repeat these two steps until all sub-requirements are fulfilled. This approach may give rise to an approximate solution. For example, we may advise students to take more courses than required, or inform the student that she has not fulfilled the course requirements when she actually has.

9.2.2. Forbidden Pairs. If we view `forbidden` pairs of courses as those having same content, then we can effectively "eliminate" one of the courses. We "eliminate" courses as illustrated in the following example. Consider a forbidden pair (a, b). For this pairs, we construct a new course c that is taken if at least one of a or b is already taken. Also, if

[2]This can be computed by counting the number of courses with the largest number of units that add up to the remaining units.

both a and b have been taken, then $u(c) = \max\{u(a), u(b)\}$ (the idea is that the student will probably use the course with the more units to satisfy his requirements). If a has been taken and b has not been taken, then $u(c) = u(a)$. Symmetrically, if b has been taken and a has not been taken, then $u(c) = u(b)$.

We thus approximate forbidden pairs by assuming that they represent the same course in all sub-requirements.

This elimination can be done even if there are multiple courses, all pairs of which are forbidden. Here, we assume that if (a, b) is forbidden for one sub-requirement, then (a, b) is forbidden for all sub-requirements.

Another way to approximate forbidden pairs is to check, after a run of the algorithm, which sub-requirements R_j have been satisfied by forbidden pairs. We suitably increment k_j for those sub-requirements by the number of forbidden pairs assigned to the sub-requirement, in a manner similar to Section 9.2.1.

9.2.3. And Clauses. We can group a set of courses together if they form part of an `and` clause, and always appear together as one alternative `and` clause in the set of courses. This new course will be taken iff all its components have been taken. Once again, we assume that if ab forms an `and` clause in sub-requirement R_j, then ab always appears together in all sub-requirements. This approach is similar to that in Section 9.2.2.

As an example, consider `and` clause abc. We replace abc with a new course d, taken only if a, b and c are all taken. Also, we have $u(d) = u(a) + u(b) + u(c)$. In addition, we update the *score* of d to be $\min\{score(a), score(b), score(c)\}$.

10. OTHER DOMAINS

Although we focus here on course requirements, we believe that requirements appear in other recommendation scenarios and hence our solutions can be either applied directly or extended. We illustrate this by briefly describing some scenarios where recommendations are constrained by requirements.

Say we need to recommend 20 movies for screening at a movie festival. In addition, at least 3 movies of each genre, i.e., horror, comedy, romance, action, need to be screened. In this case, there is no upper bound on the number of movies of each genre that can be shown. This situation directly corresponds to a range version of the core model of requirements that is, requirements of the form `take` k_1`-to-`k_2 `items from` S with movie selection for each genre representing one sub-requirement (lower bound on number of items per sub-requirement is 3, upper bound ∞). The global requirement of 20 movies can also be captured in our model.

Additionally, we may need to use forbidden pairs to represent movies by the same director (i.e., the constraint that no director gets more than one movie that he/she directed screened at the festival). Also, we may use `and` clauses to represent groups of short films that may replace a long movie.

As another example, consider recommendation of stock purchases to a user. We assume that amounts are discretized in chunks of (say) $1000. We wish to diversify the stock portfolio by investing in stocks of multiple types — banking, I.T., oil, etc. For example, we can capture investing between $5000 and $7000 in I.T. stocks as a range sub-requirement. Some stocks (companies) are present in multiple types, because they have divisions in each type. We can capture stocks that should not be purchased together (forbidden pairs), and also the expected depreciation of each $1000 investment (as the "cost" of the investment). Risk involved can be captured using units.

As a third example, consider hardware purchases for software projects. We wish to recommend purchase of the smallest number of additional hardware (and also determine optimal allocation of already purchased hardware) for serving the needs of large scale software projects. Also, from all such allocations, we need to recommend the pur-

Table I. Statistics on the requirements of various departments and the number of students.

Department	Number of courses in requirements	Number of sub-requirements	Number of students
Biology	39	8	120
Computer Science	140	21	63
Economics	22	3	149
Electrical Engineering	66	13	44
Human Biology	12	2	182

chase that costs the least amount of money. Assume that each software project has a certain minimum number of servers required. (Note that in this case range sub-requirements need not be used.) The units constraint could be used to represent the fact that the combined processing power of the servers assigned to any given project must be greater than some amount. Many small servers (represented using an and clause) could be equivalent to a large server. We could have pairs of servers that cannot be used together for a project — for example, those servers running Windows cannot be used with those running Linux, represented in our case using forbidden pairs.

Another example is that of team formation [Lappas et al. 2009], where we wish to hire the minimum number of people to work on a project, such that certain skill-sets are met (i.e., subrequirements): e.g., 2 people who know Java, 3 who know C, etc. People possess multiple skill-sets. Also, people have different expertise in each skill-set, e.g., X knows Java better than he knows C — we capture this by extending our model. The extension of the model involves assigning a score corresponding to assigning a course to each sub-requirement. For example, if we hire a person who is not very good at Java (but is an expert at C) as a Java programmer, we get a lower score than when we hire the same person as a C programmer. Note that the flow algorithm can be extended to handle these scores[3].

11. CASE STUDY

In this section, we describe experiments we performed to test our algorithms. We measure our algorithms in terms of running time, and on the usefulness of our recommendations. We used the transcripts of the 558 undergraduate students who graduated in Fall 2008 from Stanford University with majors in biology, computer science, economics, electrical engineering, or human biology. Each transcript contains the full sequence of courses taken by the student in the order in which the student took the courses. Courses taken together in a single quarter are ordered alphabetically. These students need to fulfill a set of sub-requirements to graduate with a degree in their selected major; the number of sub-requirements for the majors we examined was between 2 (for human biology) and 21 (for computer science). Since our focus is on requirements-based recommendations, we restricted our evaluation to courses that are mentioned at least once in the course requirements for the department/major under examination, i.e., *curricular* courses. There are between 12 (for human biology) and 140 (for computer science) such courses for each major. The other courses (which we deleted) are simply "extra-curriculars", e.g., Drama, Physical Education, and play no role in satisfaction of requirements. We present some statistics about the requirements we dealt with in Table I.

Even though the number of sub-requirements is not too large, there are many ways in which students can graduate in a particular discipline, making the problem of recommendations non-trivial. For example, from among the curricular courses in the

[3]By letting the edges from the items to the sub-requirement nodes have "costs", instead of the edges from the source s to the items

Computer Science (CS) major, we found that the average Jaccard similarity between the sets of curricular courses taken by two CS students is only ~ 0.42 on average, indicating that the CS students have multiple options and they exercise those options in order to satisfy their requirements.[4]

For each student we constructed the graph $\mathcal{G}$ using approximate techniques. By excluding some of the constraints (for instance, we exclude the units and forbidden pairs constraints, we expand `and` clauses out into its alternatives, e.g., abc into a, b, c) in the extended model, we effectively convert it into the core model. Since this core model only excludes some of the constraints, we will not have any false negatives when using the flow algorithms to check and make recommendations. However, we may have false positives, since there are fewer constraints. Hence, an additional check would have to be made to any solution suggested by the core model — which may then lead to a solution of the extended model. As we shall see in the following, even these techniques are sufficient to achieve high precision while making recommendations. For additional approximate techniques that could have been used, refer to Section 9. We then checked requirements using the Ford-Fulkerson max-flow algorithm (as expected, all students met requirements). The algorithm took 0.16 seconds per student on average for the most complex set of requirements (CS) we examined; the experiment was run on a server running Linux 2.6.25 with an Intel Xeon processor (2.0 GHz) and 4 GB of RAM. The algorithm was implemented in C++.

11.1. Flow-Based Recommendations

Our recommendation techniques are provably correct, in a sense, since they return the best possible set of courses for a given function $score(\cdot)$ that operates on courses. However, we would still like to see how "good" our recommendations are. We did so by evaluating the precision of our experiments (i.e., the overlap between the courses that we recommend, and the courses actually taken by the student). We used two different functions $score(\cdot)$:

(1) The first one simply returns a number proportional to the popularity (i.e., the number of students who took the course the previous year) of the course among all students of the university; we will denote this function in the following as $score_1(\cdot)$.
(2) The second one returns a number proportional to the popularity of the course among students of the university that graduated in the same major as the student under examination; we will denote this function in the following as $score_2(\cdot)$.

Even though the second $score(\cdot)$ function is more fine-grained than the first, we observed experimentally that they lead to similar results. The experiment demonstrating this is described later on.

We implemented the following recommendation techniques:

— *Requirements*: The plain max-flow algorithm that we described earlier (or the min-cost max-flow algorithm with "equal scores" for all not taken courses).
— *Requirements with Popularity*: The min-cost max-flow algorithm we described earlier where the $score(\cdot)$ function is $score_1(\cdot)$.

[4]We report here the average Jaccard similarity between curricular courses in all the majors we have examined for completeness. For biology the value was 0.74, for computer science it was 0.42, for economics it was 0.68, for electrical engineering 0.51, and for human biology 0.98. Human biology gave very high values because it had a rather "inflexible" set of requirements. It contained two sub-requirements with non-overlapping courses. The first sub-requirement was of the form take 6 out of a set of 6 courses (basically take all 6 courses) and the second one was of the form take 4 out of a set of 6 courses. Obviously, the students could only deviate in the second sub-requirement, but still not significantly.

— *Vanilla Popularity*: Let k be the number of courses recommended by the previous two schemes. (This value is nothing but the minimum number of courses for a student to fulfill her requirements.) This scheme recommends the top-k among all courses over all departments that the student has not taken. We tried using both $score_i(\cdot)$ functions described above. However, we found that this recommendation technique performs extremely poorly, with $< 5\%$ precision against the courses actually taken by the student. Due to this reason, we do not display this technique in our figures.
— *Popularity of Curricular Courses*: (Or *Popularity* for short) If k is the minimum number of courses for a student to fulfill her requirements, this scheme recommends the top-k *curricular* courses that the student has not taken (the courses examined appear in the sub-requirements of the specific major). We used the $score_1(\cdot)$ function described above.

We perform our evaluation via two experiments, similar to each other: (1) we delete a certain portion of a transcript *from the end*; for example, we delete the 10 courses that a student has taken last in his/her transcript; (2) we delete a certain portion of a transcript *randomly*; for example, we delete 10 arbitrary courses that a student has taken. We then measured the *precision* of our experiments. As an example, if the deleted portion of the transcript was the set $\{a, b, c, d, e, f\}$, and we recommend that the student takes $\{a, b, m\}$ to fulfill requirements, then our precision is: $\frac{|\{a,b,m\} \cap \{a,b,c,d,e,f\}|}{|\{a,b,m\}|} = 66.66\%$.

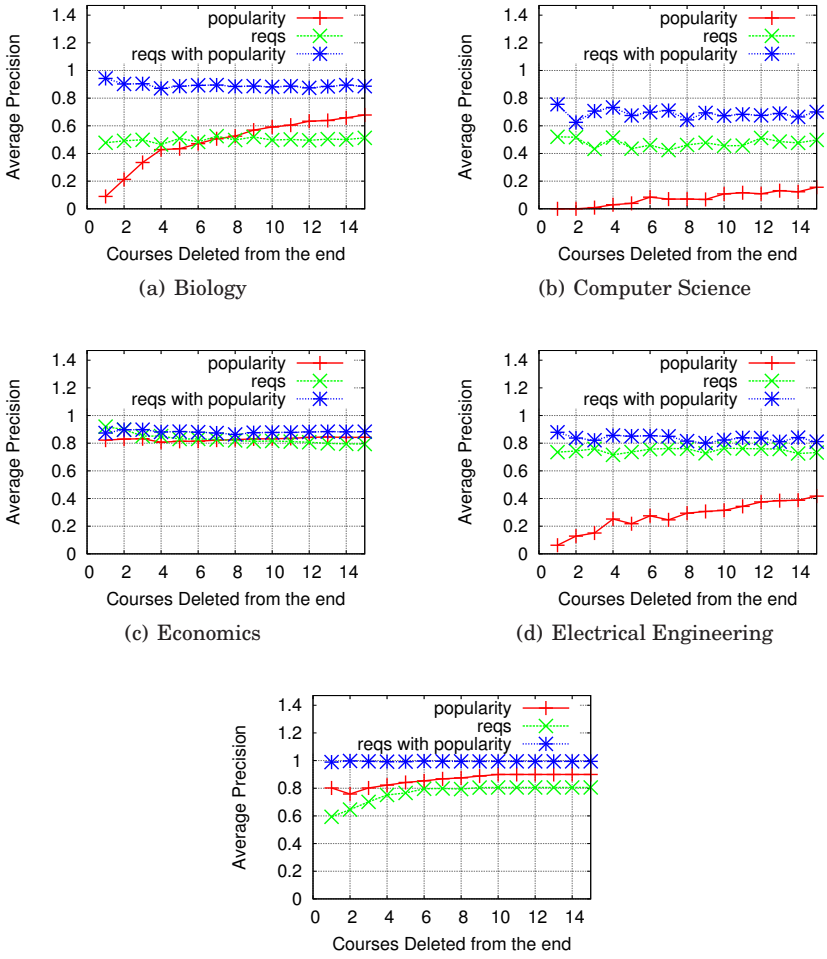
We present figures for all the majors we have examined (Figure 5 for courses deleted from the end of transcripts and Figure 6 for courses deleted at random from the transcripts). Note that there is not a large difference between the two sets of figures; practically, the only difference is that the random deletion of courses gives "smoother" graphs. The curves in each sub-figure show how the precision varies with the number of courses deleted from the transcript, averaged over all students.

Both min-cost max-flow techniques (*Requirements* and *Requirements with Popularity*) display a very high precision against the courses actually taken by students — illustrating the fact that our algorithm can be used to recommend courses that students actually end up taking. Additionally, taking into account even a simple score like popularity consistently improves precision by 5% and in some cases by even 50% (e.g., biology) between the plain *Requirements* method and the *Requirements with popularity* method. The *Popularity* technique performed poorly compared to our algorithms: Only when we deleted almost all the courses in the transcript did its precision increase. Bear in mind that this scheme only recommends the top-k courses that a student has not taken from the courses that *appear in the requirements he wants to satisfy*, and not generally from all courses that the University offers (which performs extremely poorly throughout).

Note that our methods have high precision throughout, and not only when almost all the courses are deleted from the transcript. Even when all the courses are deleted from the transcript, the *Requirements with Popularity* method performs better than *Popularity*. Interestingly, for some majors, we found that the *Popularity* method does better than the *Requirements* method, displaying that disregarding popularity can result in poor recommendations.

Note that in all disciplines, the *Requirements with Popularity* method performs exceedingly well, in all cases above a precision of 0.75, and in most cases, as high as 0.9. This result indicates that the approximate techniques that we used to convert complex constraints into constraints of the core model are sufficient in practice to provide "good" recommendations.

In order to compare the two $score(\cdot)$ functions, we repeated the experiment for all departments with both alternatives ($score_1(\cdot)$ and $score_2(\cdot)$) for the *Popularity of Cur-*



(a) Biology (b) Computer Science

(c) Economics (d) Electrical Engineering

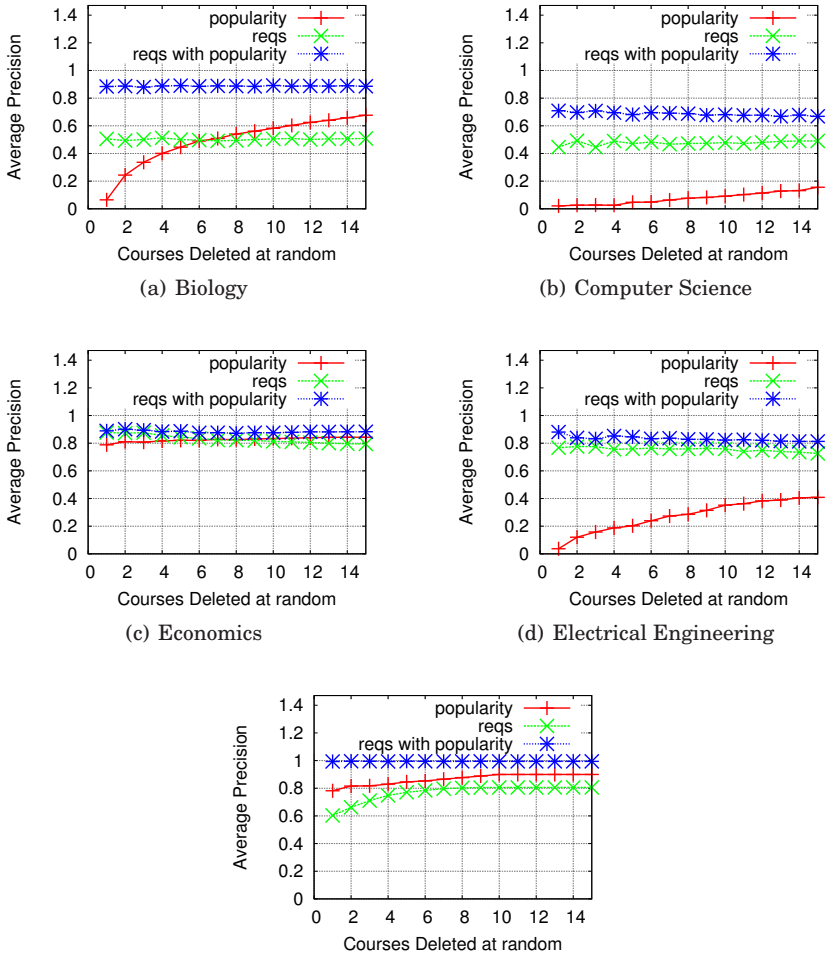
(e) Human Biology

Fig. 5. Variation of precision between recommended courses and courses actually taken vs. number of courses deleted *from the end* for various majors.

ricular Courses scheme as well as the *Requirements with Popularity* scheme. The results are displayed in Figure 7. This figure clearly shows that neither the *Popularity* scheme nor the *Requirements with popularity* scheme change significantly on changing the $score(\cdot)$ function.

For the experiments described so far, we artificially deleted courses from the end of the transcripts of students. We now explore what fraction of courses deleted were used to satisfy requirements for graduation. Discovering this fraction would enable us to decide when to recommend additional courses (beyond the ones used to satisfy requirements) to students that they may find interesting or fun. Figure 8 helps us understand the choices students make for their course work (for Computer Science students).

We deleted between 1 and 20 courses from the end of the transcripts of Computer Science students, and for the remaining portion of each transcript, we used our algorithms to discover how many additional courses were required to satisfy requirements.



(a) Biology (b) Computer Science

(c) Economics (d) Electrical Engineering

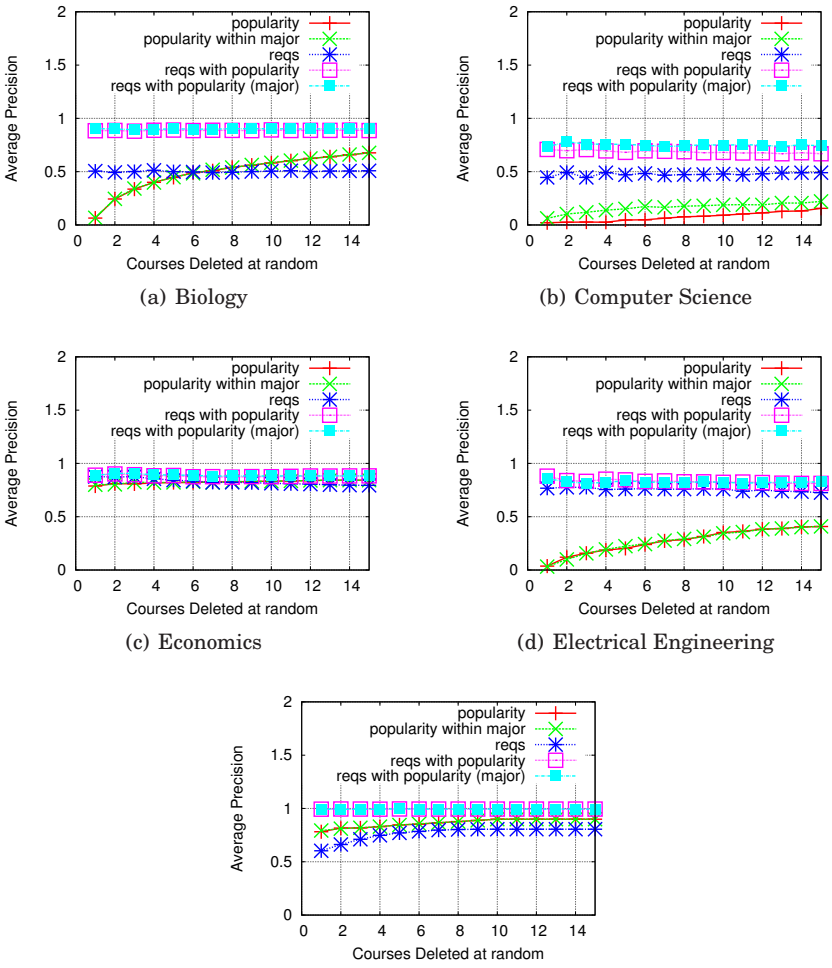
(e) Human Biology

Fig. 6. Variation of precision of recommended courses against courses actually taken vs. number of courses deleted *randomly* for various majors.

We plotted the average number of additional courses required (averaged over many student transcripts) versus the number of courses deleted in Figure 8. From the figure, we can see that practically for every two courses we delete, on average, one of them was used towards the fulfillment of some requirement. Even though students do not just take the minimum number of courses possible in order to graduate, they at least have to take some courses that will help them fulfill their major's requirements. For the remaining courses that the students take that do not help meet requirements, we can always use traditional recommendation strategies to recommend courses that the students would find interesting and/or fun.

11.2. Exact Recommendations

In order to see how an exact requirements checker would perform, we implemented the Integer Linear Program for the same problem, with the complete set of constraints found in the CS Department requirements. We used the GPLEX [Makhorin 2000]



(a) Biology (b) Computer Science

(c) Economics (d) Electrical Engineering

(e) Human Biology

Fig. 7. Comparison of all methods for all $score_i(\cdot)$ functions we have examined

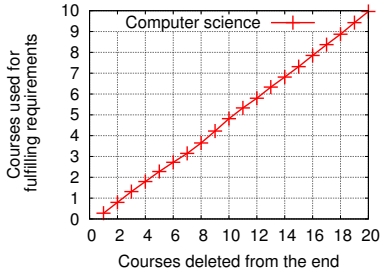


Fig. 8. Number of courses (out of the deleted ones) that were used to satisfy some requirement vs. number of courses deleted from the end of the transcripts. The y-axis values are averaged over all Computer Science student transcripts we used for our evaluations.

Mixed Integer Linear Program (MILP) solver for this purpose. We used the same popularity based $score_1(\cdot)$ function that we used above. The ILP, on average, took 0.8 seconds (around 5 times longer than the approximate flow method) to check requirements and make recommendations for a single student.

In order to compare the performance of the ILP versus the flow algorithm, we deleted 10 courses from the end of each of the CS student transcripts, and measured the number of courses recommended per student by the ILP versus the flow algorithm, and how good those recommendations were. The additional constraints in the ILP required that sometimes, more courses be recommended to the student in order to fulfill the same sub-requirements. As a result, the ILP, on average, recommended more courses (6.36 courses), while the flow algorithm recommended fewer courses (5.93 courses) for a student to complete their requirements.

The number of courses that overlapped with courses actually taken by the student was higher too — the ILP had 4.82 courses (on average) overlapping with the 10 courses deleted as against 4.65 (on average) for the approximate flow approach. These results show that if we can tolerate longer running times, then we can obtain slightly better recommendations by using the ILP.

If scores for each node are picked appropriately, course planning for any given student will automatically take into account prerequisites, in the following manner: While running the flow algorithm, per quarter, we either work towards course requirement completion, in a greedy fashion, or we take courses that work as prerequisites in order to allow the student to take better courses in subsequent quarters.

All of this is automatically done by the course planning procedure in Section 5.4, once the appropriate scores are assigned to prerequisite courses.

12. RELATED WORK

The problem of recommending items when requirements or constraints are present can be posed as one of two variants: First, that of recommendation of a single item, subject to certain constraints. For example, recommendation of a non-stop flight from point A to B, subject to constraints such as that of cost, flying time, baggage allowance etc. The best item that satisfies the constraints is presented to the user. Some work in this regard has been done previously [Felfernig and Burke 2008; Zanker and Jessenitschnig 2009]. The second problem, which we address in this paper, involves recommending a set of items that satisfies constraints. In this case, the items in the set satisfy the constraints when put together, but may not do so when one item is replaced by another item not already found in the set. Thus the problem becomes a lot harder since all combinations of items need to be considered in the worst case.

One work we are aware of in the set recommendation domain is that of [Xie et al. 2010], performed concurrently with our work. In [Xie et al. 2010] each item has a value (rating) and a cost. The recommendation is a set of items such that the total cost of the recommended items is within a budget. While, the notion of cost in [Xie et al. 2010] is analogous to the course units in our model, our problem is orthogonal to that of [Xie et al. 2010]. In our setting the user (student) must have taken a minimum of course units, while in [Xie et al. 2010] the user wants to be recommended items with a total cost less than a preset budget.

Another work in the set recommendation area that takes into account various constraints is described in [Karimzadehgan and Zhai 2009]. This work deals with the problem of assignment of papers to reviewers subject to constraints. Some of the constraints they consider include: (1) each paper is reviewed by more than a certain number of reviewers, (2) each reviewer reviews up to a certain number of papers, and (3) the reviewers assigned to review a paper have the expertise to review it. Greedy and an ILP-based solutions are given to solve the problem. Our models can express

most of the constraints discussed in [Karimzadehgan and Zhai 2009]; some extensions of our techniques can capture practically all the aspects of the constraints defined in [Karimzadehgan and Zhai 2009].

The work on collaborative constraint-based recommendations [Zanker 2008] solves a different problem — that of inferring constraints or "functional dependencies" among items consumed/purchased by users, and using those for recommending items to other users. While these techniques could be applied in our context, they capture dependencies that are less expressive than those of the core and extended model. In addition, our aim is to return the "optimal" recommended set for the set of constraints, not just items inferred using dependency rules based on consumption patterns of other users.

In this paper, we do not try to reinvent good recommendation strategies, but rather incorporate existing solutions into our algorithm. There are many excellent recommendation algorithms that have been proposed and we can utilize [Sarwar et al. 2001; Adomavicius and Tuzhilin 2005; Pazzani and Billsus 2007; Burke 2000; Burke 2002]. Most of these algorithms exploit collaborative (i.e., recommend items that other users have liked) and content (i.e., recommend items that are "good") information, as well as preference information provided by the user and prior user history to generate a score for an item for a given user. These recommendation algorithms are very popular, and are used in many shopping and media websites, such as Amazon ("Here are items related to items you viewed.." or "People who bought this item also bought.."), Netflix ("We recommend X because you enjoyed movie Y") and Pandora ("We play song X because.."). Our focus is on the combination of the constraints and the scores returned by these traditional recommendation algorithms. All of our algorithms combine both notions, returning highly recommended courses that also satisfy constraints.

Since our focus is on satisfying constraints, we may at times end up recommending courses that are not novel or interesting. However, since students end up taking more courses than is needed to meet requirements, we can provide novel and diverse courses for the remaining courses. (Our experience tells us that students want to take interesting or diverse courses, but not at the cost of delaying graduation!) For those courses it is critical to evaluate the related notions of unexpectedness, goodness, novelty and diversity [Sarwar et al. 2001; Parameswaran et al. 2010; McNee et al. 2002], typically using user-studies.

Our work draws on the work of traditional Constraint Satisfaction Problems (CSP) [Russell and Norvig 2003], a subdomain of the field of planning in Artificial Intelligence, where the extended problem is to find an assignment of values to variables to satisfy certain constraints. The extended method of solving CSPs is an exponential backtracking approach. The problem we wish to solve is a special case of this general problem. However, our work is specific to course requirements, and making "good" recommendations to satisfy them. Additionally, in our work, if a course is *assigned* to a sub-requirement, then it cannot be assigned to any other sub-requirement. This notion of assignment does not exist in traditional CSP.

Our work uses network flow algorithms of max-flow and min-cost max-flow. These algorithms have also been used in operations research problems in the past, e.g., project scheduling [Levner and Nemirovsky 1994], inventory management [Barahona and Jensen 1998], resource allocation [Xiao et al. 2004], processor partitioning [Bokhari 1988], network planning [Wu et al. 2005], but there has been no application of these algorithms in constrained recommendation systems.

A number of problems can be cast as ILPs, which are NP-Complete. However, several MILP solvers (mixed-integer-linear program) exist — e.g., CPLEX [Ceria 1996], MINTO [Nemhauser and Savelsbergh 2004], GLPK [Makhorin 2000], which perform very well in practice for small problems, and problems that have few constraints. The ILP that we create is a 0-1 ILP, i.e., the variables can only be assigned

values 0 or 1. There are heuristic approaches to solve 0-1 ILPs [Geoffrion 1967; Mahendrarajah and Fiala 1976].

13. CONCLUSIONS

Our interaction with Stanford staff and advisors has shown us that there are many cases where students forget to explicitly check requirements while taking classes. Students then end up missing deadlines, postponing graduation or rushing and taking classes that they are not interested in just to graduate. Furthermore, there are usually many ways in which students can graduate, so "personalized" recommendations offer them options they may not have thought of, are more aligned to their interests, or ones they had simply forgotten to consider. We believe the situation is the same at most universities, and hence course recommendations can play a very useful role.

In this work, we introduced two frameworks for course requirements, and considered the dual problems of checking requirements and making recommendations. For the core model, making recommendations can be done using a flow algorithm. We showed how to extend our algorithms for the core model to handle course planning, range recommendations, handling prerequisites, returning multiple options, and using complex scoring functions.

For the extended model, we provided an ILP solution. However, since the ILP may be too expensive, we provided two kinds of techniques to convert the ILP into instances of the flow algorithm. The first involves multiple runs of the flow algorithm and is exact, but assumes bounded number of extra constraints. The second is a heuristic technique.

We evaluated both the approximate flow-based techniques and the exact ILP technique for requirement checking and recommendations on a dataset of student transcripts for 5 majors. We found that both the techniques recommend "good" courses that the students actually end up taking. The ILP makes marginally better (and more accurate) recommendations but takes a longer time to execute. We also found that even a primitive recommendation function (such as one returning popularity) could improve the quality of recommendations by a significant amount.

Our recommendations are based on the students expressed preferences (their declared major) and choices they have made in the past (courses taken so far). These two components (preferences and history) have been extensively used in recommender systems. There are two additional aspects of our work that differentiate it from previous work: the constraints that exist for the items that are recommended and the recommendation of sets of items (as opposed to just items). Our models capture many different constraints found in real settings.

Our technique of integrating traditional recommendation indicators along with complex constraints or requirements may be adapted for application in other domains. In fact, our constraint models themselves may be general enough to be relevant in multiple domains. There are several domains where constraints such as "take some out of a set of items" are applicable, as seen in Section 10. The other features described in the extended model such as forbidden pairs, units, etc. make our constraint model even more powerful and applicable in more general situations where recommendations are made to satisfy constraints.

ACKNOWLEDGMENT

We would like to thank Claire Stager for providing us with the CS Students data and Benjamin Bercovitz, Henry Liou and Filip Kaliszan of the CourseRank system for their help. We also thank the reviewers for their insightful comments.

REFERENCES

ADOMAVICIUS, G. AND TUZHILIN, E. 2005. Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. *IEEE TKDE 17*, 734–749.

BARAHONA, F. AND JENSEN, D. 1998. Plant location with minimum inventory. *Math. Program. 83,* 1, 101–111.

BOKHARI, S. H. 1988. Partitioning problems in parallel, pipeline, and distributed computing. *IEEE Trans. Comput. 37,* 1, 48–57.

BURKE, R. 2000. Knowledge-based recommender systems. In *Encyclopedia of Library and Information Systems*. Vol. 69.

BURKE, R. 2002. Hybrid recommender systems: Survey and experiments. *User Modeling and User-Adapted Interaction 12*.

CERIA, S. 1996. CPLEX 3.0.

EDMONDS, J. AND KARP, R. M. 1972. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM 19*, 248–264.

FELFERNIG, A. AND BURKE, R. D. 2008. Constraint-based recommender systems: technologies and research issues. In *EC*. 1–10.

FORD, L. R. AND FULKERSON, D. R. 1962. *Flows in Networks*. Princeton University Press.

GEOFFRION, A. M. 1967. Integer programming by implicit enumeration and balas' method. *SIAM Review*, 178–190.

GOMORY, R. E. 1958. Outline of an algorithm for integer solutions to linear programs. *Bull. Amer. Math. Soc. 64,* 5, 275–278.

KARIMZADEHGAN, M. AND ZHAI, C. 2009. Constrained multi-aspect expertise matching for committee review assignment. CIKM. 1697–1700.

LAPPAS, T., LIU, K., AND TERZI, E. 2009. Finding a team of experts in social networks. In *KDD*. 467–476.

LAWLER, E. L. 1972. A procedure for computing the k best solutions to discrete optimization problems and its application to the shortest path problem. *Management Science 18,* 7, 401–405.

LEVNER, E. V. AND NEMIROVSKY, A. S. 1994. A network flow algorithm for just-in-time project scheduling. *European J. of O. R. 79,* 2, 167–175.

MAHENDRARAJAH, A. AND FIALA, F. 1976. A comparison of three algorithms for linear zero-one programs. *ACM Trans. Math. Softw. 2,* 4, 331–334.

MAKHORIN, A. 2000. Gnu linear programming toolkit.

MCNEE, S. M., ALBERT, I., COSLEY, D., GOPALKRISHNAN, P., LAM, S. K., RASHID, A. M., KONSTAN, J. A., AND RIEDL, J. 2002. On the recommending of citations for research papers. In *CSCW*. 116–125.

NEMHAUSER, G. AND SAVELSBERGH, M. 2004. Minto 3.1.

PARAMESWARAN, A., GARCIA-MOLINA, H., AND ULLMAN, J. D. 2010. Evaluating combining and generalizing recommendations with prerequisites. In *CIKM*. 919–928.

PARAMESWARAN, A. G., KOUTRIKA, G., BERCOVITZ, B., AND GARCIA-MOLINA, H. 2010. Recsplorer: recommendation algorithms based on precedence mining. In *SIGMOD*. 87–98.

PAZZANI, M. J. AND BILLSUS, D. 2007. Content-Based Recommendation Systems, The Adaptive Web. 325–341.

RUSSELL, S. AND NORVIG, P. 2003. *Artificial Intelligence: A Modern Approach* 2nd Ed. Prentice-Hall.

SARWAR, B., KARYPIS, G., KONSTAN, J., AND REIDL, J. 2001. Item-based collaborative filtering recommendation algorithms. In *WWW*. 285–295.

TARJAN, R. E. 1983. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics.

WU, Y., CHOU, P., ZHANG, Q., JAIN, K., ZHU, W., AND KUNG, S.-Y. 2005. Network planning in wireless ad hoc networks: A cross-layer approach. *Selected Areas in Communications, IEEE Journal on 23,* 1, 136–150.

XIAO, L., JOHANSSON, M., AND BOYD, S. 2004. Simultaneous routing and resource allocation via dual decomposition. *IEEE Trans. on Communication 52,* 7, 1136–1144.

XIE, M., LAKSHMANAN, L. V., AND WOOD, P. T. 2010. Breaking out of the box of recommendations: from items to packages. In *RecSys*. 151–158.

ZANKER, M. 2008. A collaborative constraint-based meta-level recommender. In *RecSys*. 139–146.

ZANKER, M. AND JESSENITSCHNIG, M. 2009. Case-studies on exploiting explicit customer requirements in recommender systems. *User Modeling and User-Adapted Interaction 19*, 133–166.

Received Month Year; revised Month Year; accepted Month Year