Analyzing Student Credits

Don Hush, Elias Lopez, Kevin Warne, Wisam Al-Doroubi, Tushar Ojha, Brianne Santos
Institute of Design and Innovation (IDI)
University of New Mexico, Albuquerque, NM 87106
idi.unm.edu

April 25, 2022

Contents

1 Introduction

This report describes methods for analyzing the efficiency of a student's progress towards a degree at an educational institution. A degree is granted when the student satisfies all degree requirements. Thus, progress is measured as headway towards requirement satisfaction. Requirements are generally satisfied by earning credits towards a specific subject area, and nearly all credits are earned by successfully completing classes in the subject area. Thus, student progress can be quantified by comparing the total credits earned towards the degree to the total credits required by the degree. We note that simply comparing the student's total accumulated credits to the total required credits is not a suitable progress measure because some of the accumulated credits may be *unusable*, i.e. they don't count towards any of the degree requirements. In addition, some credits may actually be in *excess* of specific degree requirements and therefore do not indicate progress towards degree completion. Thus progress is determined by the number of credits that count towards the degree requirements without excess, i.e. the *applied* credits. We also seek to quantify the *efficiency* of progress which is determined by comparing the total *applied* credits to the total accumulated credits ($accumulated = applied + unusable + excess$). In addition to determining the *number* of credits we seek to determine the *types* of credits in each category. For example we seek to determine whether the credits are transfer or non-transfer, lower or upper division, elective or non-elective, and so on. We note that attributing credit *types* to the *applied*, *unusable*, and *excess* categories requires the assignment of each individual class to one of these three categories. This level of assignment might not be necessary if we were only interested in the *number* of credits in each category.

There are often many ways to attribute a student's classes to the degree requirements, and some ways may suggest more progress than others. This report seeks to determine the maximum number of applied credits. This is accomplished by solving an optimization problem that matches classes to requirements in a way that maximizes the number of applied credits subject to the constraints that each class is matched to at most one requirement and no excess credits are attributed to any requirement. The solution to this optimization problem is not necessarily unique, and so the attribution of classes (and class types) to the unusable, excess, and applied categories is not necessarily unique. To obtain a unique solution we introduce an algorithm that swaps the initial optimal solution with an alternative optimal solution that satisfies a set of user-defined prioritization rules. For example, the earliest completed classes may be given the highest priority for the applied category. Alternatively we may want to give a higher priority to institutional courses over transfer courses, or vice versa. The application of different prioritization rules allows us to perform a detailed analysis of unusable, excess, and applied credits under different scenarios.

2 Separate-class Requirements

The methods described in this report are designed to work with a set of *separate-class requirements* where each individual class is only allowed to match one requirement, and separate requirements can only be matched by separate classes. But the full set of requirements for a university degree typically do not take this form. This can be illustrated with a simple toy example. Figure 1 shows a stereotypical collection of requirements for a hypothetical Bachelor's degree in Computer Engineering. These requirements are split into three groups: state-wide core requirements which are designated by the state department of higher education, institution-wide requirements which are common to all degree programs at the institution, and degree-specific requirements which are designated by a college or department at the institution. This collection contains numerous overlapping

State-wide Core Requirements (9 CR total)	
Requirement	Requirement Description
S1 (3 CR)	Core Area 1 (Communications)
S2 (3 CR)	Core Area 2 (Humanities)
S3 (3 CR)	Core Area 3 (Math)

Institution-wide Requirements	
Requirement	Requirement Description
I1	Total Credits must be $\geq$: 30
I2	Student must be accepted into the Comp Engg Degree Program
I3	All classes must be completed, i.e. no incompletes
I4	Total number of "passed classes without a grade" credits ≤ 6

Degree-specific Requirements (40 CR total)	
Requirement	Requirement Description
R1 (3 CR)	English I
R2 (3 CR)	Communications I
R3 (4 CR)	Math I
R4 (4 CR)	Physics I
R5 (4 CR)	Computer Hardware I
R6 (3 CR)	Computer Software I
R7 (6 CR)	Concentration (Hardware or Software) R7a or R7b
R7a (6 CR)	Hardware Electives
R7b (6 CR)	Software Electives
R8 (6 CR)	Technical Electives
R9 (4 CR)	Science Electives
R10 (3 CR)	Humanities Electives

Figure 1: Requirements for a hypothetical Bachelor's degree in Computer Engineering.

requirements. For example the degree-specific requirements have been designed to accommodate all the state-wide core requirements so that satisfaction of R2, R10, and R3 guarantees satisfaction of S1, S2 and S3 respectively. Thus, the state-wide core requirements are redundant. In addition, the degree-specific requirements have been designed to accommodate some of the institution-wide requirements. For example completion of the degree-specific requirements guarantees the satisfaction of requirement I1, and so I1 is redundant. We refer to requirements like S1, S2, S3, and I1 as *shared-class requirements* because classes attributed to them can also be attributed to other requirements. On the other hand, a class that is attributed to one of the degree-specific requirements in the third block cannot be attributed to another requirement in this block, e.g. Physics I cannot count for both R4 and R9. Thus the degree-specific requirements in the third block are *separate-class requirements*. In this report we measure student progress against a set of requirements that are all *separate-class requirements*. These requirements are formed from degree-specific requirements that are designed to accommodate most (if not all) of the other requirement specifications. With this approach the number of credits associated with the matched classes forms the basis for a suitable measure of progress.

3 Institutional Audit Tools (e.g. Lobo Trax)

Institutions are generally required to employ an audit system that compares student's classes to a list of degree requirements that includes both shared-class and separate-class requirements. At the University of New Mexico (UNM) the official audit system is called *Lobo Trax*. Lobo Trax uses the *DegreeWorks* software produced by the company *Ellucian*. The DegreeWorks software can accommodate a mixture of requirement types and provides the institution with a great deal of flexibility when it comes to requirement specification and evaluation. Lobo Trax is a realization of the DegreeWorks software that has been programmed by UNM personnel to produce student audits for the degree programs at UNM. Thus in principle, Lobo Trax could be used to produce the audit results needed in our analysis of student progress. However, there are a number of logistic, institutional, and practical barriers to this path. Some of these barriers are described below.

1. UNM personnel have designed the Lobo Trax system with a fixed set of prioritization rules that may vary from one college to another and are unknown to the end user. This makes it difficult for the end user to explore different prioritization schemes and to provide the correct interpretation of results obtained from the Lobo Trax data.

2. The methods described in this paper are designed to solve an optimization problem that determines the maximum number of *applied* credits, but the DegreeWorks software appears to take a different path[1] . Although the exact matching algorithm employed by DegreeWorks is unclear, their documentation describes a greedy matching approach whose optimality properties appear to be unknown.

3. It is unclear how to extract the analysis information from the current Lobo Trax output. The audit output has been designed for other purposes, and is not well suited to the analysis methods described in this report. For example, Lobo Trax produces results for the full set of degree requirements, and its not immediately clear which requirements are shared-class and which are separate-class. Also, while the Lobo Trax output produces a match between classes and requirements, it does not provide information about which other classes could have been matched to specific requirements, making it difficult for the end user to correctly identify *excess* credits.

4. The Lobo Trax system performs an audit based on the student's current class list, but some of our analysis will require an audit against a restricted class list. Consider the following example. The residency requirement states that students must complete at least 30 credits at UNM. So students enrolled in a 120 credit degree program can expect at most 90 transfer credits to be applied to their UNM degree. However many transfer students enter UNM with more than 90 transfer credits[2]. In these cases we would like to determine the maximum number of transfer credits that the students could actually expect to contribute towards the degree. This can be determined by running an audit against the transfer classes only.

4 One-of Class Requirements (OCRs)

The problem of computing an optimal match between a student's classes and a list of separate-class requirements is complicated by the fact that the requirements can typically be satisfied by

[1] DegreeWorks Technical Guide, Release 5.0.1, March 2018

[2] Indeed, some transfer students enter UNM with more than 300 transfer credits.

a large number of different class combinations. We refer to an individual class combination that matches all the separate-class requirements in a degree program as a *class-level realization* of the degree program. One way to determine student progress is to enumerate all possible class-level realizations, and then find a realization that best matches the student's classes. But this brute force approach is often impractical because of the extremely large number of class-level realizations. For example, one type of requirement that can lead to a large number of class-level realizations is a *one-of class requirement* (OCR) which states that the student must complete one class from a list of classes. Each OCR requirement increases the number of class-level realizations by a multiplication factor equal to the size of the OCR list, and so OCR requirements can increase the total number of class-level realizations very rapidly.

Example 1. Consider a degree program that has many requirements, three of which are OCRs with class list sizes of 20 each. The number of class combinations that satisfy these three requirements alone is $20^3 = 8000$, and so these three requirements increase the number of class-level realizations by a multiplication factor of 8000.

A substantial reduction in the number of combinations can be obtained by replacing the enumeration of *class combinations* by the enumeration of OCR *combinations*. That is, the computational requirements can be greatly reduced by representing each degree program realization as an OCR-list instead of a class-list, and then finding the best match between the student's class-list and all OCR-list realizations. We refer to an OCR-list realization as a *degree path*. While a modest amount of extra computation is required to match a class-list to an OCR-list (instead of matching to another class-list), this approach drastically reduces the number of match computations because OCR requirements do not contribute a multiplication factor to the number of OCR-list realizations. In the example above the three OCRs would simply appear once in each degree path without increasing the number of paths. Thus, from these three requirements alone, the number of OCR-list realizations would be nearly 4 orders of magnitude less than the number of class-level realizations, i.e. from 8000 to 1.

This combinatorial reduction comes with a modest increase in the computation required to compute a match. Indeed we must now match a class-list to an OCR-list (instead of another class-list). This problem is complicated by the fact that multiple classes can satisfy a single OCR, and multiple OCRs can be satisfied by a single class, but in the end we seek a one-to-one match. That is, we seek a pairing between OCRs and classes where each class is matched to at most one OCR and each OCR is matched to at most one class. In particular we seek a one-to-one match that maximizes the number of credit hours contributed by the matched classes. Section 8 shows how this can be accomplished by solving a *maximum weighted bipartite matching* problem.

Matching classes to OCRs not only allows us to quantify progress towards degree completion, but also enables numerous additional types of analysis that were not previously possible. These include:

1. **Student Progress Assessment:** The goal here is to use the student's accumulated credits to quantify progress towards the degree, and then to use this quantification to determine whether the pace of progress is adequate. Not all credits accumulated by a student are *applied* credits, i.e. credits that are essential to completion of the degree program. Indeed, some credits may be *unusable*, i.e. not applicable to the degree program, while others may be *excess*, i.e. more than what's needed by the degree program. Thus, the key to determining student progress is to identify the student's *applied* credits, and to be able to distinguish them from the *unusable* and *excess* credits. These three types of credits can be derived as a by-

product of the the optimal match solution. Indeed, the applied credits are derived from the matched classes and the *unusable* and *excess* credits are derived from the unmatched classes.

2. **Student Planning:** The goal here is to help the student determine the best path forward. One aspect of this planning is to identify the *unsatisfied* requirements in the student's chosen degree program, and then make an individualized plan to help the student meet these requirements in a timely manner. Unsatisfied requirements are easily derived as a by-product of the the optimal match solution. Another aspect of student planning is to assess the student's progress towards *other* degree programs. For example, this will allow the student to determine the effect of switching to another degree program of interest. More broadly this will allow the student to identify all other degree programs where significant progress has been made so that all viable options are known.

3. **Credit Efficiency Analysis:** The goal here is to develop an institution-wide understanding of credit hour efficiency. This is accomplished by decomposing student credits into *applied*, *unusable*, and *excess* categories, and then examining the magnitude and cause of the *unusable* and *excess* credits. The optimal match solution produces a unique number of applied, unusable, and excess credits, but the specific course-to-requirement assignments may not be unique. Indeed, there may be many optimal solutions that assign classes to requirements in different ways but produce the same number of applied credits. Thus, the determination of which courses contribute to the applied and excess categories may be ambiguous. This ambiguity can be resolved by prioritizing the attributed classes. For example classes that are completed earlier might be given higher priority. This report uses optimal matching and class prioritization to create a list of *applied*, *unusable*, and *excess* classes (credits) for each student.

4. **Transfer Credit Analysis:** The goal here is to develop an institution-wide understanding of transfer credit efficiency. This is accomplished by applying the credit efficiency analysis separately to transfer and non-transfer credits, and then comparing the results.

5. **Requirement Analysis:** The goal here is to develop an institution-wide understanding of degree requirements. Some requirements are more easily met than others, and some are routinely met with an excess number of credits, i.e. they are over-satisfied. Unsatisfied and over-satisfied requirements are easily derived as a by-product of the the optimal match solution. This analysis seeks to identify these different requirement types, and to understand the factors that influence their magnitude.

The remainder of this report is organized as follows.

- Section 5 describes the *All Paths* algorithms that convert degree program requirements to degree paths and then find the best match to the student's classes.

- Section 6 describes the *copy-and-replace* operations required by the *All Paths* algorithms to convert degree program requirements into a list of degree paths.

- Section 7 provides examples of the process that converts degree requirements into degree paths for specific degree programs at UNM.

- Section 8 describes the algorithm that matches student classes to degree path OCRs in a way that maximizes the number of credit hours contributed by the matched classes.

- Section 9 describes approximation algorithms that require substantially less computation than the *All Paths* algorithms, but may produce sub-optimal solutions.
- Section 10 shows how to use the best matching degree path to compute an effective measure of student progress, and how to decompose the the best matching degree path into components that support requirement fulfillment assessment and excess credit analysis.
- Section 11 addresses possible inaccuracies in the results produced by the methods in this report. It describes numerous practical reasons for these inaccuracies, including a lack of access to complete student class lists, or incomplete requirement specifications. It also derives quantitative accuracy measures and uses them to analyze the accuracy of the methods developed in this report.

5 The *All Paths* Algorithms

This section presents an overview of two different algorithms for converting degree program requirements into degree paths, and then find the best match to the student's classes. Details of these algorithms are described in later sections.

We start with the following definitions.

- c = a individual class
  - If two (or more) classes should be taken simultaneously, such as a class and its corresponding lab, then these classes are treated as a single class. In particular, if non-zero degree credit is given only when both classes are completed then these classes are treated as a single class.
  - Co-requisites are generally *not* treated as a single class.
- C = *one-of* course requirement (OCR) (also called a *one-course* requirement)
  - An OCR is represented by a *set* of individual classes, i.e. $C = \{c_1, c_2, ...\}$. Note that an OCR cannot contain repeated classes, i.e. a specific class cannot occur more than once in an OCR. Note also that an OCR may contain classes with different numbers of credit hours. For example some requirements can be satisfied by the completion of either a 3-credit or a 4-credit class. It turns out however, that a vast majority of the OCRs created by the methods in this report contain only classes with the same number of credits hours. OCRs that contain classes with different numbers of credit hours are created only to accommodate certain types of *variable credit hour* requirement.
  - An OCR is satisfied when there is a match to at least one of the classes in the set.
  - *Special Case:* As a special case, an OCR may contain an individual class, i.e. $C = \{c\}$. In this case the student is required to complete the class c.
- p = a *path* (also called a *product requirement*)
  - A path is represented by a list of OCRs, i.e. $p = (C_1, C_2, ...)$. Note that a path can contain repeated OCRs, i.e. a specific OCR can occur more than once in a path. For example a student may be required to complete two (or more) technical electives, i.e. two (or more) classes from the same class list.

- A path is satisfied when there is a distinct match to each of its OCRs. A path is *partially* satisfied when some, but not all, of its OCRs are matched.
- *Special Case (path = OCR):* As a special case, a path may contain a single OCR, i.e. $p = (C)$. In this case the student must satisfy C.
- *Special Case (path = class):* As a special case, a path may contain a single OCR which contains an individual class, i.e. $p = (\{c\})$. In this case the student must complete the class c.
- A *degree path*, denoted p_D, is a special type of path that, when satisfied, constitutes the completion of a degree.

- $P =$ a *one-of* path requirement (OPR) (also called a *one-path* requirement)
  - An OPR is represented by a set of paths, i.e. $P = \{p_1, p_2, ...\}$. An OPR cannot contain repeated paths, i.e. a specific path cannot occur more than once in an OPR.
  - An OPR is satisfied when at least one of its paths is satisfied. An OPR is *partially* satisfied when some of its paths are partially satisfied.
  - *Special Case:* As a special case, an OPR may contain a single path, i.e. $P = \{p\}$. In this case the student must satisfy the path p.
  - A *degree* OPR, denoted P_D, is a special type of OPR that contains a list of all degree paths for a particular degree program.

- $R =$ a *general requirement* can be either
  1. a path,
  2. a logical combination of other requirements (combined using ANDs and ORs),
  3. a combinatorial combination of other requirements (i.e. m-of-n other requirements),
  4. a credit hour requirement, or
  5. a credit hour and number of courses requirement

  These general requirement types will be described in more detail below.

- $D =$ degree program,
  - A degree program is defined by a list of general requirements, i.e. $D = (R_1, R_2, ..., R_M)$. Note that a degree program can contain repeated requirements, i.e. a specific requirement can occur more than once (e.g. *Technical Electives* often occur more than once).
  - A degree program is satisfied when each of its general requirements is satisfied. A degree program is *partially* satisfied when some, but not all, of its general requirements are satisfied.
  - *Special Case:* Because a general requirement can be a logical combination of other requirements a degree program can always be represented by a single general requirement, i.e. $D = (R)$ where $R = R_1 \wedge R_2 \wedge ... \wedge R_M$.

Consider a degree program with M general requirements, i.e. $D = (R_1, R_2, ..., R_M)$. If all M of these requirements are path requirements, i.e. $R_i = p_i$ where the p_i are either individual class

requirements, individual OCR requirements, or individual OCR-lists, then the degree program can be expressed as a single degree path of the form

$$\begin{aligned} p_{degree} &= R_1 \wedge R_2 \wedge ... \wedge R_M \\ &= p_1 \odot p_2 \odot ... \odot p_M \end{aligned} \tag{1}$$

where $\odot$ represents the *concatenation* operation so that

$$p_{degree} = \left(\underbrace{C_{1,1}, C_{1,2}, ...}_{p_1}, \ \underbrace{C_{2,1}, C_{2,2}, ...}_{p_2}, \ ..., \ \underbrace{C_{M,1}, C_{M,2}, ...}_{p_M} \right)$$

where $C_{i,j}$ is the j^{th} OCR for the i^{th} requirement. In this case student progress is determined by computing the best match between this degree path and the student class-list as described in Section 8.

But most degree programs contain some non-path requirements. In this case the non-path requirements must be converted to path requirements which are then concatenated to form degree paths. More specifically, each non-path requirement is converted into a list of path requirements which correspond to different realizations of the non-path requirement. Then multiple degree paths are formed by concatenating all possible combinations of path requirements (realizations) from the original non-path requirements. This process can be articulated and visualized through the use of *requirement trees* and *product trees*. A *requirement tree* is simply a tree whose internal nodes are either conjunctions (logical AND) or disjunctions (logical OR) of other requirements. For example, the requirement tree for the degree program $D = (R_1, R_2, R_3, R_4)$ is shown in Figure 2 below. A rectangle node is used to indicate that the requirement is a conjunction of other requirements and a diamond node is used to indicate a leaf whose requirement type is open.

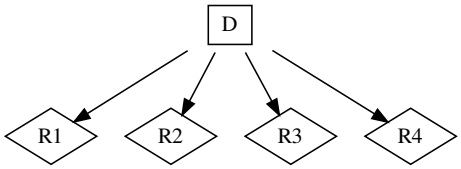


Figure 2: Requirement tree for degree program $D = (R_1, R_2, R_3, R_4)$.

A more elaborate example of a requirement tree is shown in Figure 3 below for the following

requirement.

$$R = R1 \wedge ((R5 \wedge (R9 \vee R10)) \vee (R7 \wedge R8))$$

$$= R1 \wedge \Bigg(\underbrace{\Bigg(\underbrace{R5 \wedge \Big(\underbrace{R9 \vee R10}_{R6} \Big)}_{R3} \Bigg) \vee \Big(\underbrace{R7 \wedge R8}_{R4} \Big)}_{R2} \Bigg) \tag{2}$$

A red ellipse node is used to indicate that the requirement is a disjunction of other requirements.

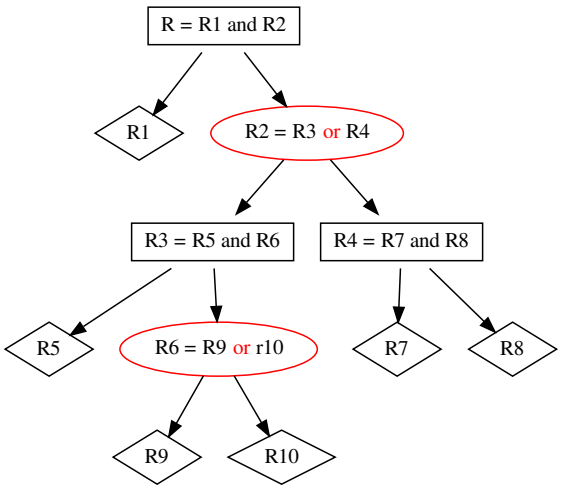


Figure 3: The Requirement Tree for an LC-Requirement.

A *product tree* is simply a requirement tree whose internal nodes are all conjunctions.

The process of converting a degree program into list of degree paths can be viewed as the process of converting a requirement tree into a forest of product trees. The next two sections describe two slightly different ways of implementing this process.

5.1 ADP: The All Degree Paths Algorithm

The process of converting a degree program into list of degree paths, and then finding the degree path that best matches the student's classes, can be accomplished by carrying out the steps in the *All Degree Paths (ADP)* algorithm in Figure 4. Section 6 develops *copy-and-replace* operations for Step 2b of the ADP algorithm. The ADP Algorithm is guaranteed to produce an optimal result, but the number of trees (degree paths) generated by this algorithm may be excessive.

All Degree Paths (ADP) Algorithm

1. Given a degree program $D = (R_1, R_2, ..., R_M)$, the first step is to synthesize a product tree representation like the one in Figure 2.

2. Initialize the first *working* tree to be the product tree from the first step. Then repeat the following *copy-and-replace* operation until all the leaves of all the working trees contain path requirements only.

   (a) Choose R to be any remaining non-path leaf requirement.

   (b) *Copy-and-Replace:* Make m copies of all working trees that contain R and replace the node R with a different product tree in each copy. The value of m and the content of the m product trees are determined by the methods described in Section 6. They correspond to m different realizations of R.

3. Once this process is completed there will be a forest of working product trees whose leaves contain path requirements only. Each working product tree is converted into a degree path by forming a product of its leaf paths (similar to the operation in (1) above).

4. Once the degree paths have been created the next step is to compute the best match between the degree program and the student's classes. This is accomplished by computing the max-match between each degree path and the student's classes, and then tracking the best max-match(es). More specifically, a match is a pairing between the degree path OCRs and the student's classes where each class is matched to at most one OCR and each OCR is matched to at most one class. A max-match is a pairing that maximizes the number of credit hours contributed by the matched classes. Section 8 shows how to compute the max-match by solving a *maximum weighted bipartite matching* problem. A max-match is computed between each degree path and the student's classes, and the degree path(s) with the best max-match value is retained. Then the student's progress is determined by the the number of credit hours in the best max-match as described in Section 10.1.

Figure 4: The All Degree Paths (ADP) Algorithm

5.2 ARP: The All Requirements Paths Algorithm

The ADP algorithm in Section 5.1 guarantees an optimal solution, i.e. it is guaranteed to produce the maximum number of credit hours that count towards a degree program, but it can be computationally prohibitive because of the extremely large number of degree paths. In this section we describe a slightly different algorithm called the *All Requirements Paths* (ARP) algorithm which also guarantees an optimal solution and may be more efficient than the ADP algorithm. Perhaps more importantly the ARP algorithm has the following advantages.

- The ARP algorithm computes an alternate representation of degree requirements that requires only a modest amount of storage and can be used to enumerate all degree paths in a very simple manner. Thus the ARP algorithm provides an improved degree program representation that has a simpler computational format than the original degree requirements and requires far less storage that the *all degree paths* representation.
- The ARP algorithm computes a separate OPR for each degree requirement. This makes it easier to track the progress against these requirements, i.e. it is easier to determine which requirements are satisfied, unsatisfied, and partially satisfied.
- A simple modification to the ARP algorithm produces the very effective approximation algorithm described in Section 9.1.

The *All Requirements Paths (ARP)* algorithm is shown in Figure 5. This algorithm works by converting individual degree program requirements into OPRs, and then forming degree paths by computing all combinations of the paths from these OPRs. This algorithm creates an alternate representation of degree requirements that consists of the core path p_{core} plus the OPRs created for each of the non-path requirements, i.e. the alternate representation is $(p_{core}, P_1, P_2, ..., P_K)$. While the potential exists for an individual OPR P_i to have an excessively large number of paths, experience has shown the number of paths to be quite manageable in practice, even for the most combinatorially explosive requirements. Thus, this alternate representation requires only a modest amount of storage. Note also that each degree path contains a clearly identifiable path segment from each of the original requirements so that once the best matching degree path has been identified it is easy to assess progress towards the individual requirements.

All Requirement Paths (ARP) Algorithm

1. Given a degree program $D = (R_1, R_2, ..., R_M)$, the first step is to separate these requirements into two groups. The first group consists of all path requirements, i.e. individual class requirements, individual OCR requirements, and individual OCR-lists, and the second group consists of all remaining requirements. Without loss of generality let $(R_a, R_b, R_c, ...)$ be the first group and $(R_1, R_2, ..., R_K)$ be the second group.

2. Form a single core path p_{core} from the requirements in the first group.

$$\begin{aligned} p_{core} &= R_a \wedge R_b \wedge R_c \wedge ... \\ &= p_a \odot p_b \odot p_c \odot ... \end{aligned}$$

   where $\odot$ represents the *concatenation* operation so that

$$p_{core} = \left(\underbrace{C_{a,1}, C_{a,2}, ...}_{p_a}, \ \underbrace{C_{b,1}, C_{b,2}, ...}_{p_b}, \ \underbrace{C_{c,1}, C_{c,2}, ...}_{p_c}, \ ... \right)$$

   where $C_{x,i}$ is the i^{th} OCR from the path p_x.

3. Convert each requirement from the second group $(R_1, R_2, ..., R_K)$ into an OPR. This can be accomplished by applying Steps 2 & 3 of the ADP algorithm to each of these requirements separately. Let these OPRs be called $P_1, P_2, ..., P_K$.

4. An individual degree path will now take the form

$$p_D = p_{core} \odot p_1 \odot p_2 \ ... \odot p_K$$

   where p_i is one of the paths from P_i. Form the *degree* OPR P_D by enumerating degree paths formed from p_{core} and all possible combinations of paths from the OPRs $(P_1, P_2, ..., P_K)$.

5. Once the degree paths have been created the next step is to compute the best match between the degree program and the student's classes as described in Step 4 of the ADP algorithm.

Figure 5: The All Requirements Paths (ARP) Algorithm

Note that the total number of degree paths is equal to the product of the OPR sizes, i.e.

$$\text{number of degree paths } = |P_1| \cdot |P_2| \cdot ... \cdot |P_K|.$$

Thus, a modest reduction in OPR size can translate into a large reduction in the number of degree paths. To this end we can often improve the runtime of the ARP algorithm without sacrificing its optimality by simply removing all requirement paths that do not match any of the student's completed classes. This can be accomplished by inserting the following operation after Step 3. For each path in each P_i, a check is made to see if at least one of the student's classes match at least

one of the path OCRs, and if there is no match then the path is removed. Section 9.1 explores an extension of this idea, where only the best matching requirement paths are retained.

6 Copy-and-Replace Operations

This section shows how to perform the *copy-and-replace* operation for the following types of non-path requirements.

1. An *LC* requirement is a *logical combination* of other requirements (i.e. combined using ANDs and ORs). This includes nested combinations.

2. A *CC* requirement is a *combinatorial combination* of other requirements (i.e. m-of-n other requirements).

3. A *CH* requirement is a *credit hour* requirement that states that the student must complete (at least) N credit hours from a specific category, where the category is witnessed by a list of classes.

4. A *CHC* requirement is a *credit hour and number of courses* requirement that states that the student must complete M courses that contribute (at least) N credit hours from a specific category, where the category is witnessed by a list of classes.

The last two types can be converted directly into OPRs as shown in Sections 6.3 and 6.4, but the first two types are combinations of other general requirements which themselves can be combinations of yet other general requirements, and so on. This process can lead to a deep nesting of general requirements that terminates at the level where requirements can be converted directly into OPRs. In practice it is extremely rare to encounter more than one or two levels of nesting, but with the *copy-and-replace* operations in Sections 6.1-6.4 below the All Paths algorithms are able to handle the most general case.

Sections 6.1-6.4 treat the most common types of requirements that appear in a degree program. Section 6.5 treats the special situation where an individual course can contribute a variable number of credit hours towards the degree. Less common requirement types are treated in the Appendices. For example Appendix A considers the less common *variable credit hour requirement* (VCR), and Appendix B treats the special case where one course can satisfy two (or more) requirements.

The sections below contain a number of practical examples based on non-path requirement specifications from actual UNM degree programs. It is important to point out that in many of these examples the original non-path requirement specification can be simplified, and that these simplifications can lead to more efficient path representations and simpler requirement specifications for the end-user to understand. This is illustrated in the sections below.

6.1 Logical Combinations of Other Requirements

This section describes the *copy-and-replace* operation for non-path *LC* requirements, i.e. requirements that are a logical combination of other requirements. We assume that the other requirements can be a mixture of paths, *CC* requirements, *CH* requirements, and *CHC* requirements. Also, although the methods described below will work when some of the other requirements are also *LC* requirements, we generally expect "*LC* requirements of *LC* requirements" to be combined into a single (nested) *LC* requirement.

The next two sections consider the two simple cases where the LC requirement consists entirely of conjunctions, or entirely of disjunctions. Then we consider the more general case where the requirement may contain nested logical combinations.

6.1.1 Conjunctive Logical Combinations

This section describes *copy-and-replace* operations for an LC requirement that consists entirely of conjunctions, i.e.

$$R_{cc} = R_1 \wedge R_2 \wedge ... \wedge R_K$$

Note that this requirement is already in the form of a product tree, so strictly speaking no action is required in Step 2b of the ADP algorithm. Nevertheless, this type of requirement can sometimes be simplified. For example, if the requirements $(R_1, R_2, ..., R_K)$ are all paths, then these paths can be concatenated into a single path and R_{cc} can be replaced by a single path requirement[3]. If *some* of the requirements R_i are paths, then these paths should be replaced by a single concatenated path. Then, once the path requirements have been concatenated, the remaining requirements are either *CC* requirements, *CH* requirements, or *CHC* requirements. The ADP algorithm will then process these remaining requirements according to their requirement type using the *copy-and-replace* operations described in the sections below.

6.1.2 Disjunctive Logical Combinations

This section describes *copy-and-replace* operations for an LC requirement that consists entirely of disjunctions, i.e.

$$R = R_1 \vee R_2 \vee ... \vee R_K. \tag{3}$$

It is important to recognize the an LC requirement consisting entirely of *nested* disjunctions can be replaced by the simpler (un-nested) form in (3). This is illustrated in the following example.

Example 2. Consider the *First-Year Learning Communities* requirement shown below. This requirement states that the student must satisfy 1-of-19 other requirements, i.e. it is a disjunction of 19 other requirements

$$R = R_1 \vee R_2 \vee ... \vee R_{19}.$$

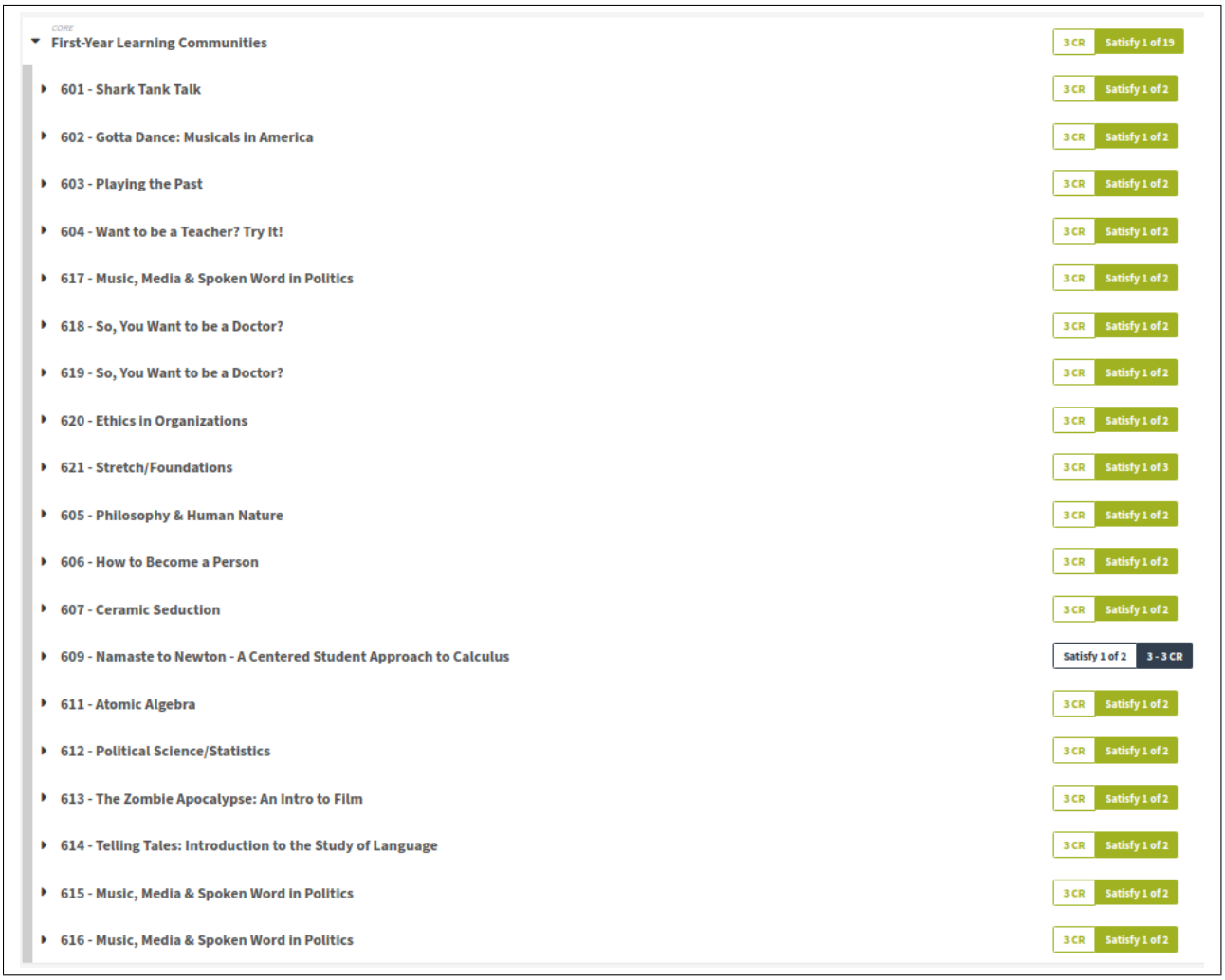
In addition, each of the 19 other requirements are disjunctions of 2 (or 3) other requirements, e.g.

$$R_i = R_{i,1} \vee R_{i,2}$$

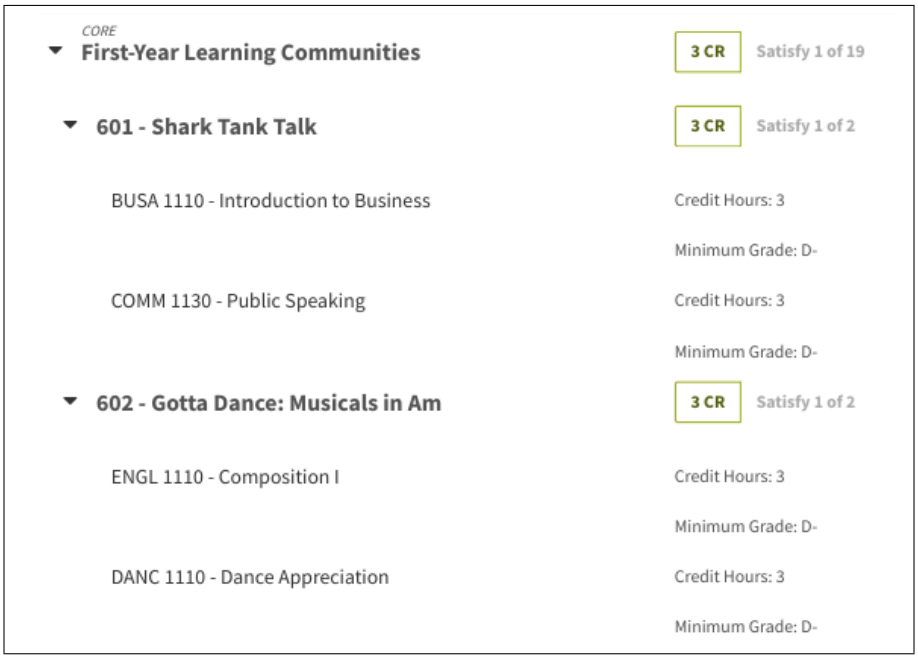
These nested conjunctions can be replaced by a single level of (un-nested) conjunctions, i.e. R can be replaced by

$$R = R_{1,1} \vee R_{1,2} \vee R_{2,1} \vee R_{2,2} \vee ... \vee R_{19,1} \vee R_{19,2}$$

[3]Strictly speaking this is not an example of a non-path requirement and should not be labeled as such.



Now we describe *copy-and-replace* operations for disjunctive LC requirements of the form in (3). First we consider the special case where all the requirements $(R_1, R_2, ..., R_K)$ are individual course requirements. In this case R can be replaced by an OCR containing the individual courses, and no further action is required. This is precisely the case for the *First-Year Learning Communities* requirement in the example above, since all of its second level requirements turn out to be individual class requirements. The first two are shown in the diagram below.



Next we consider the case where some, but not all, of the requirements $(R_1, R_2, ..., R_K)$ are individual course requirements. Without loss of generality let $R_1, R_2, ..., R_m$ correspond to the individual course requirements $c_1, c_2, ..., c_m$. In this case the individual course requirements can be replaced by an OCR containing the courses, i.e. $C = \{c_1, c_2, ..., c_m\}$, and R is replaced by

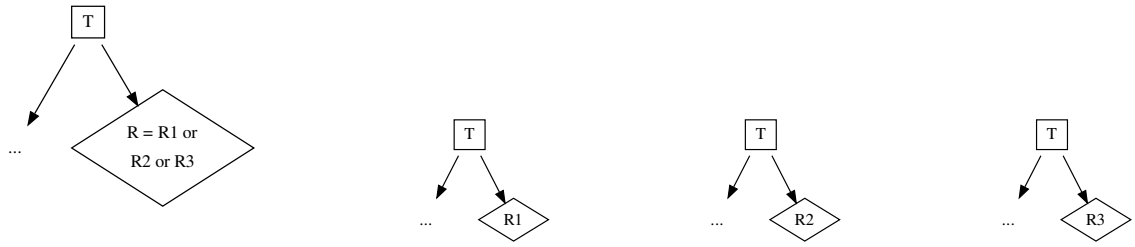
$$R = C \vee R_{m+1} \vee ... \vee R_K$$

which is a disjunction of non-class requirements.

Now we consider the more general case where none of the requirements $(R_1, R_2, ..., R_K)$ are individual course requirements. In this case we apply the following procedure.

Procedure 1. *Copy-and-replace for disjunctions* (CRD): For each working tree containing $R = R_1 \vee R_2 \vee ... \vee R_K$, copy the tree K times, and then replace R by R_k in the k^{th} tree.

An example with $k = 3$ is shown in Figure 6 below.



(a) Tree with conjunctive leaf requirement R (b) Copy #1 (c) Copy #2 (d) Copy #3

Figure 6: Copy-and-Replace example for conjunctive leaf requirement R.

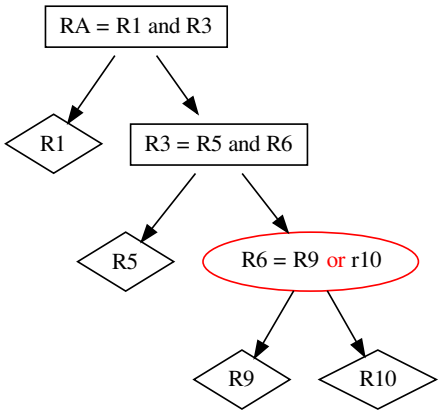
6.1.3 Nested Logical Combinations

This section describes the *copy-and-replace* operation for a non-path LC requirement R that may contain nested logical combinations as illustrated in Equation (2) and Figure 3. In this case we apply the following procedure.

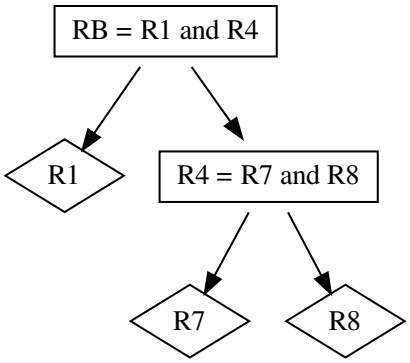
Procedure 2. *Copy-and-replace for nested logic combinations* (CRNLC): For each working tree containing the nested logic leaf R, copy the tree m times and replace R by a different product tree in each copy, i.e. attach a different product tree as a sub-tree to each copied tree. The value of m and the content of the m product trees are determined as follows.

1. Start at the root of R's requirement tree and traverse the tree in either *depth-first-search* or *breadth-first-search* order.
2. Each time the traversal encounters a conjunctive requirement of the form $\bar{R} = R_1 \vee R_2 \vee ... \vee R_K$,
   (a) Detach the sub-tree rooted at $\bar{R}$
   (b) Copy the remaining tree K times
   (c) Then, for each copied tree, re-attach the sub-tree rooted at R_k to the position where $\bar{R}$ was previously located, and continue the traversal from this position.

Example 3. As an example consider the application of Procedure 2 to the requirement tree in Figure 3. The first conjunctive requirement encountered during the traversal is $R2$. Application of Step 2 to $R2$ yields the two new trees RA and RB in Figure 7. Continued traversal of these two trees encounters no more conjunctive requirements in RB, and one more conjunctive requirement RA. Application of Step 2 to $R6$ in RA yields the two new trees in Figure 8. Continued traversal of these two trees yields no more conjunctive requirements, so we end up with the three product trees in Figures 7b, 8a, and 8b. To complete the application of Procedure 2 we would make three copies of each working tree that contains R, and replace R in the three copies with the three product trees. (The working trees are not shown in this example.)

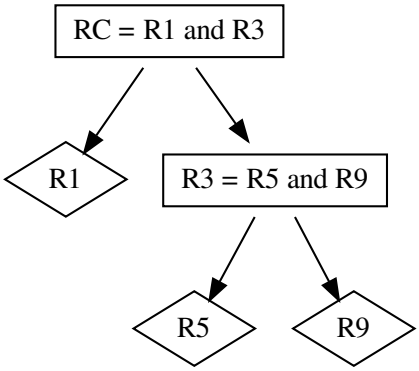


(a) Replace R_2 with R_3 in Figure 3.

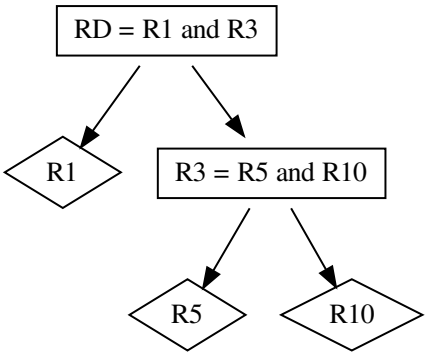


(b) Replace R_2 with R_4 in Figure 3.

Figure 7: First step in the decomposition of the requirement tree in Figure 3. Split the original requirement tree into two by replacing the OR node containing requirement R_2 with its left and right subtrees.



(a) Replace R_6 with R_9 in Figure 7a.



(b) Replace R_6 with R_{10} in Figure 7a.

Figure 8: Second step in the decomposition of the requirement tree in Figure 3. Split the tree in Figure 7a into two by replacing the OR node containing requirement R_6 with its left and right subtrees.

Procedure 2 creates product trees by detaching, copying, and manipulating sub-trees, but the product trees can also be synthesized from scratch using Procedure 3 below.

Procedure 3. *Copy-and-replace for nested logic combinations (version 2)*: This procedure synthesizes product trees by starting with the top level logic requirements for R, i.e. the logic in the root

node of R's requirement tree, and recursively parsing requirements, level by level, carrying out the following operations for each requirement:

1. If the requirement is a conjunction, i.e. $R_{current} = R_1 \wedge R_2 \wedge ... \wedge R_K$, then
   for $(k = 1, 2, ..., K)$ **do**
   attach a child node R_k to the current node $R_{current}$
   make a recursive call to continue the synthesis process from this child node
   end for

2. If the requirement is a disjunction, i.e. $R_{current} = R_1 \vee R_2 \vee ... \vee R_K$, then
   for $(k = 1, 2, ..., K)$ **do**
   make a copy of the existing tree (i.e. the tree that has been synthesized so far)
   replace the current requirement $R_{current}$ by R_k in the copied tree
   add a pointer to the copied tree to the output list (this tree may continue to grow)
   make a recursive call to continue the synthesis process from the R_k node in the copied tree
   end for

   After the **for** loop above, the pointer for the existing tree is removed from the output list (because the existing tree is replaced by the copies).

A more detailed description is provided in Algorithm 1 below. This algorithm assumes that the nested logic requirements are arranged so that each internal node of the requirement tree contains a list of requirements that are either ORed together or ANDed together[4]. With this assumption internal nodes are represented by a two-tuple (R_{list}, Op) where R_{list} is a list of requirements and Op is either AND or OR. Associated with each requirement R in R_{list} is a requirement list $R.R_{list}$ for the requirements that contribute to R at the next lower level and a corresponding operation for $R.Op$. Leaf nodes are represented as $(R, leaf)$, i.e. a two-tuple where R is a non-LC requirement and and $Op = leaf$.

Application of this algorithm to the requirement tree in Figure 3 yields the same product trees as Example 3 above.

6.2 Combinatorial Combinations of Other Requirements (*CC*)

This section describes the *copy-and-replace* operation for a CC requirement which takes the form

$$R_{cc} = m\text{-of-}n \text{ other requirements}$$

This type of requirement is represented by a three-tuple $R_{cc} = (m, n, \mathbf{R})$ where $\mathbf{R} = (R_1, R_2, ..., R_n)$ is a list of the n other requirements. The *copy-and-replace* operation for this type of requirement depends on the nature of the other requirements and is described by the three cases in the next three sections.

6.2.1 Case 1: *CC* with All Individual Class Requirements

If all of the n "other requirements" in $\mathbf{R}$ are individual classes then R_{cc} can be simplified to a product of m identical OCRs composed of the n individual classes. For example if $R_{cc} =$

[4]Any logical combination of requirements can be expressed in this manner.

Algorithm 1 Algorithm for synthesizing tree realizations from a nested logic requirement.

{Main Routine}
Inputs: collection of nested logic requirements for R of the form (R_{list}, Op)
$T = (R_{list}, Op)$ {*initialize the root node with the top level requirement list*}
initialize the output list of product trees to contain a pointer to T
`SynthesizeProductTrees`(T, T)

{Recursive Subroutine}
`SynthesizeProductTrees`(T, N) {T = root of the current tree, N = current node in T}

if $(N.Op = leaf)$ **then**
  return
end if

if $(N.Op = AND)$ **then**
  for (each requirement $R \in N.R_{list}$) **do**
    $N_I \leftarrow (R.R_{list}, R.Op)$ {create new internal node (one level down)}
    $N.child \leftarrow N_I$ {attach N_I as a child of N}
    `SynthesizeProductTrees`(T, N_I)
  end for
else
  {handle case where $N.Op = OR$}
  for (each requirement $R \in N.R_{list}$) **do**
    $\acute{T} \leftarrow$ copy of T
    $\acute{T}.N \leftarrow (R.R_{list}, R.Op)$ {move requirements from the level below $T.N$ up to node $\acute{T}.N$}
    place a pointer to $\acute{T}$ in the output list of product trees
    `SynthesizeProductTrees`($\acute{T}$, $\acute{T}.N$)
  end for
  remove the pointer to T from the output list of product trees
end if
return

3-of-7 other requirements, and the 7 other requirements are all individual classes $(c_1, c_2, ..., c_7)$ then R_{cc} can be simplified to the product of 3 OCRs that take the form $C = (c_1, c_2, ..., c_7)$, i.e. $R_{cc} = C \wedge C \wedge C$. In this case the *copy-and-replace* operation makes *zero* copies of working trees, and for all working trees that contain R_{cc} it replaces R_{cc} with a path $p_{cc} = C \wedge C \wedge ... \wedge C$ of size m.

6.2.2 Case 2: *CC* with No Individual Class Requirements

In this case none of the n "other requirements" in $\mathbf{R}$ are individual classes. This case is processed using the following three steps.

1. Enumerate all $q = \binom{n}{m}$ combinations of the "other requirements" in $\mathbf{R}$.
2. For each combination, form a new requirement R' as a conjunction of the requirements in the combination.
3. Replace R_{cc} by the disjunction of these q new requirements, i.e.

$$R_{cc} = R'_1 \vee R'_2 \vee ... \vee R'_q$$

   Now R_{cc} has the form of a disjunctive LC requirement so that the *copy-and-replace* operation described previously in Procedure 1 can be applied.

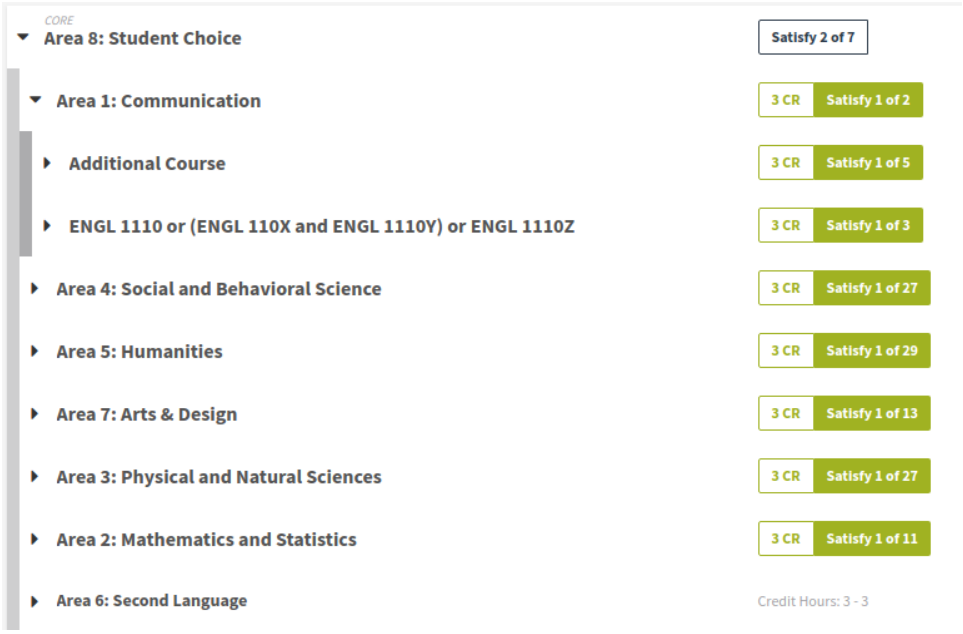
Example 4. Let R_{cc} be the requirement that the student must complete 3-of-4 requirements from (R_1, R_2, R_3, R_4), i.e. $R_{cc} = (3, 4, \mathbf{R})$ where $\mathbf{R} = (R_1, R_2, R_3, R_4)$. Then there are a total of $q = \binom{4}{3} = 4$ combinations and therefore 4 new requirements of the form

$$\begin{aligned} R'_1 &= R_1 \wedge R_2 \wedge R_3 \\ R'_2 &= R_1 \wedge R_2 \wedge R_4 \\ R'_3 &= R_1 \wedge R_3 \wedge R_4 \\ R'_4 &= R_2 \wedge R_3 \wedge R_4 \end{aligned}$$

so that R_{cc} now takes the form of an LC requirement

$$R_{cc} = R'_1 \vee R'_2 \vee R'_3 \vee R'_4.$$

Example 5. *Example 2:* Consider the *Area 8: Student Choice* requirement below.



This requirement is satisfied by completing 2 of 7 requirements, i.e. $R_{cc} = (2, 7, \mathbf{R})$, where the 7 requirements in $\mathbf{R}$ are shown below.

$$\begin{aligned} R_1 = &\quad \text{Area 1: Communication} \\ R_2 = &\quad \text{Area 4: Social and Behavioral Science} \\ R_3 = &\quad \text{Area 5: Humanities} \\ R_4 = &\quad \text{Area 7: Arts \& Design} \\ R_5 = &\quad \text{Area 3: Physical and Natural Sciences} \\ R_6 = &\quad \text{Area 2: Mathematics and Statistics} \\ R_7 = &\quad \text{Area 6: Second Language} \end{aligned}$$

There are a total of $q = \binom{7}{2} = 21$ combinations and therefore 21 new requirements. A few of these are shown below.

$$\begin{aligned} R'_1 &= R_1 \wedge R_2 \\ R'_2 &= R_1 \wedge R_3 \\ R'_3 &= R_1 \wedge R_4 \\ &\vdots \\ R'_7 &= R_2 \wedge R_3 \\ R'_8 &= R_2 \wedge R_4 \\ &\vdots \end{aligned}$$

Thus, the R_{cc} requirement is converted to the following disjunctive LC requirement

$$R_{cc} = R'_1 \vee R'_2 \vee R'_3 \vee \ldots \vee R'_{21}.$$

Application of the *copy-and-replace* operation in Procedure 1 requires every working tree that contains R_{cc} to be copied 21 times, and that R_{cc} be replaced with one of these new requirements in each copy. (The working trees are not shown in this example.)

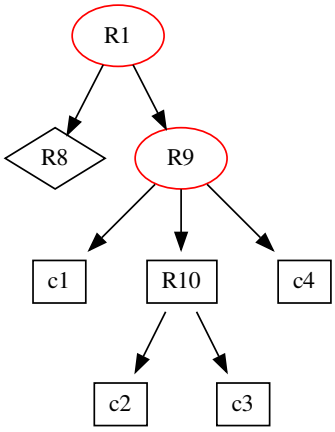
Note that the "other requirement" labeled R_1 in this example is an LC requirement of the form

$$R_1 = R_8 \vee \underbrace{\left(c1 \vee \underbrace{(c2 \wedge c3)}_{R_{10}} \vee c4 \right)}_{R_9}$$

where

$$\begin{aligned} R_8 = &\quad \text{Additional Course} \\ c1 = &\quad \text{ENGL 1110} \\ c2 = &\quad \text{ENGL 110X} \\ c3 = &\quad \text{ENGL 1110Y} \\ c4 = &\quad \text{ENGL 1110Z} \end{aligned}$$

This can be expressed by the following requirement tree.



Application of the *copy-and-replace* operations in Procedure 2 yields the following 4 product trees.

However, it is clear that R_1 is a "one-class" requirement, and so with a little pre-processing R_1 can be simplified to a single OCR as follows. First we treat the combination of ENGL 110X and ENGL 1110Y as a single class, which we denote by

$$c23 = \text{ (ENGL 110X:ENGL 1110Y)}$$

Next, let the 5 classes in $R_8 =$ `Additional Course` be denoted by

$$R_8 = (c5, c6, c7, c8, c9)$$

Now R_1 can be replaced by the 1-of-8 OCR

$$C = (c1, c23, c4, c5, c6, c7, c8, c9)$$

6.2.3 Case 3: *CC* with Some Individual Class Requirements

This section describes the *copy-and-replace* operation for a *CC* requirement

$$R_{cc} = m\text{-of-}n \text{ other requirements}$$

that takes the form $R_{cc} = (m, n, \mathbf{R})$ where $\mathbf{R}$ has a mixture of individual class requirements and non-individual class requirements. Specifically let us assume that n_1 of the "other requirements" are individual classes, n_2 are non-individual class requirements, and $n = n_1 + n_2$. We describe two solutions for this case.

- *Solution #1:* This case can be treated exactly like Case 2 where we replace R_{cc} with a disjunction of the $\binom{n}{m}$ requirement combinations.
- *Solution #2:* The total number of combinations produced by Solution #1 can sometimes be reduced as follows.
  - Let the n requirements in $\mathbf{R}$ be denoted

    $$R_1 = c_1,\ R_2 = c_2,\ ...,\ R_{n_1} = c_{n_1}, \qquad R_{n_1+1},\ ...,\ R_n$$

  - Form an OCR $C = (c_1, c_2, ..., c_{n_1})$ from the first n_1 individual class requirements.
  - Replace the first n_1 requirements with m copies of the OCR requirement C, so that the new list of "other requirements" becomes

    $$\mathbf{R}' = \underbrace{C,\ C,\ ...,\ C}_{m},\ R_{n_1+1},\ ...,\ R_n$$

    The number of requirements in the list is now $n' = m + (n - n_1)$, and the new *CC* is

    $$R'_{cc} = m\text{-of-}n' \text{ other requirements}$$

    or simply $R'_{cc} = (m, n', \mathbf{R}')$.
  - Now apply the Case 2 method to R'_{cc}.

6.3 Credit Hour Requirements (*CH*)

This section describes the *copy-and-replace* operation for a *credit hour* (*CH*) requirement that states

$$R_{ch} = \text{ a student must complete (at least) } N \text{ credit hours from a specific category}$$

where the specific category is witnessed by a list of classes

$$\mathbf{C} = (c_1, c_2, ...)$$

This type of requirement is represented by a two-tuple $R_{ch} = (N, \mathbf{C})$ where N is the minimum number of credit hours and $\mathbf{C} = (c_1, c_2, ..., c_n)$ is a list of classes that count towards the specific category. Examples of specific categories include *Technical Elective*, *Area Elective*, *Upper Division Elective*, and *Free Elective*. Note that this type of requirement will always appear at the lowest level of a requirement hierarchy because it does not contain a combination of other (non-path) requirements. The *copy-and-replace* operation for this type of requirement is achieved using the following two steps.

1. Convert R_{ch} into a disjunctive LC requirement that takes the form of an OPR. That is, R_{ch} is converted into a list of paths and a requirement that (at least) one of the paths must be satisfied.

2. Then apply the *copy-and-replace* operation for disjunctive LC requirements in Procedure 1.

One way to perform the conversion in Step 1 is to enumerate all course combinations that achieve (at least) N credit hours, and then form a disjunction of these combinations. That is, a path is formed from the classes in each combination, and then the credit hour requirement R_{ch} is replaced by a disjunction of these paths. But this number of combinations can be too large to be practical. Instead of enumerating *course combinations*, we can obtain a much more efficient method by enumerating all *credit hour combinations* that add up to (at least) N. Since the number of distinct course credit hour sizes is small (e.g. typically 1-4) compared to the number of courses, this approach often produces a much smaller number of combinations.

We now show how to convert the credit hour requirement R_{ch} into a conjunction of paths $p_1, p_2, ..., p_Q$, where each path takes the form

$$\begin{aligned} p = \quad & \text{student must take } M \text{ classes from a specific category,} \\ & \text{where the } i^{th} \text{ class must contribute } h_i \text{ credit hours, } i = 1, ..., M \end{aligned}$$

so that the general credit hour requirement can be represented as an OPR,

$$R_{ch} = p_1 \vee p_2 \vee ... \vee p_Q$$

For example p_1 and p_2 might be something like

$$p_1 = \textit{two 3-credit hour classes, one 2-credit hour class, and two 1-credit hour classes}$$

$$p_2 = \textit{three 3-credit hour classes, and one 1-credit hour class}$$

Both of these add up to $N = 10$ total credit hours. Let us start with some definitions.

- Let $\mathbf{C} = \{c_1, c_2, ...\}$ = set of all individual classes that count towards the specific category.
- Let $|c|$ = number of credits hours for course c.
- Let $\bar{C}_h = \{c_i : c_i \in \mathbf{C}, |c_i| = h\}$ be the OCR containing all h-credit hour classes that count towards the specific category. With this notation the paths that we seek take the form

  $$p = \bar{C}_{H_1} \wedge \bar{C}_{H_2} \wedge ... \wedge \bar{C}_{H_M}$$

  where H is a list of credit hour sizes, e.g. $H = (3, 3, 1, 2, 4, 2)$, and H_i is the i^{th} member of the list.
- Let $n_h = |\bar{C}_h|$ = number of h-credit hour classes from $\mathbf{C}$
- Let $A = \{(h, n_h)\}$ be the set of (number of credit hours, number of courses) tuples for the classes in $\mathbf{C}$.

We seek to enumerate all course credit hour combinations (i.e. combinations of h values) that yield a total of at least N credit hours without including unnecessary courses. Consider the following example where $N = 10$. Viable combinations include

three courses with 3 credit hours, and one course with 1 credit hour
three courses with 3 credit hours, and one course with 2 credit hours
two courses with 4 credit hours, and one course with 3 credit hours

but the combination

three courses with 3 credit hours, and two courses with 1 credit hour

is not viable because one of the 1 credit hour classes is unnecessary. Let h_{min} be the smallest number of credit hours in a combination. Then we seek to enumerate all course credit hour combinations that yield a total of $N' \geq N$ credit hours where $(N' - h_{min}) < N$. This can be accomplished with the *Enumerate Course Credit Hour Combinations* (`EnumerateCCHC`) algorithm below.

{Main Routine}
Inputs: A, N
L = empty {*the initial list of credit hour combinations is empty*}
$H = []$ {*the initial credit hour combination is blank*}
`EnumerateCCHC`(A, N, H)

{Recursive Subroutine}
`EnumerateCCHC`(A, N, H)
$N' \leftarrow$ sum of credit hour values in the current list H
if $(N' \geq N)$ **then**
  $h_{min} \leftarrow$ smallest credit hour value in current list H
  if $((N' - h_{min}) < N)$ **then**
    add the combination H to the list L of output combinations
  end if
  return
end if

for (each $(h, n_h) \in A$, in order from smallest h to largest h) **do**
  if ($(n_h > 0)$ and ($h >$ most recent addition to H)) **then**
    $H' \leftarrow$ add h to the credit hour combination in H
    $A' \leftarrow A \setminus (h, n_h) \cup (h, n_h - 1)$ {*adjust the count so that* $(h, n_h) \leftarrow (h, n_h - 1)$}
    `EnumerateCCHC`*(A', N, H')*
  end if
end for
return

This routine outputs a list of credit hour combinations $L = (H_1, H_2, ..., H_Q)$ that are used to create the paths $p_1, p_2, ..., p_Q$. For example the credit hour combination

$$H = (1, 1, 1, 2, 2, 3)$$

is used to create the path

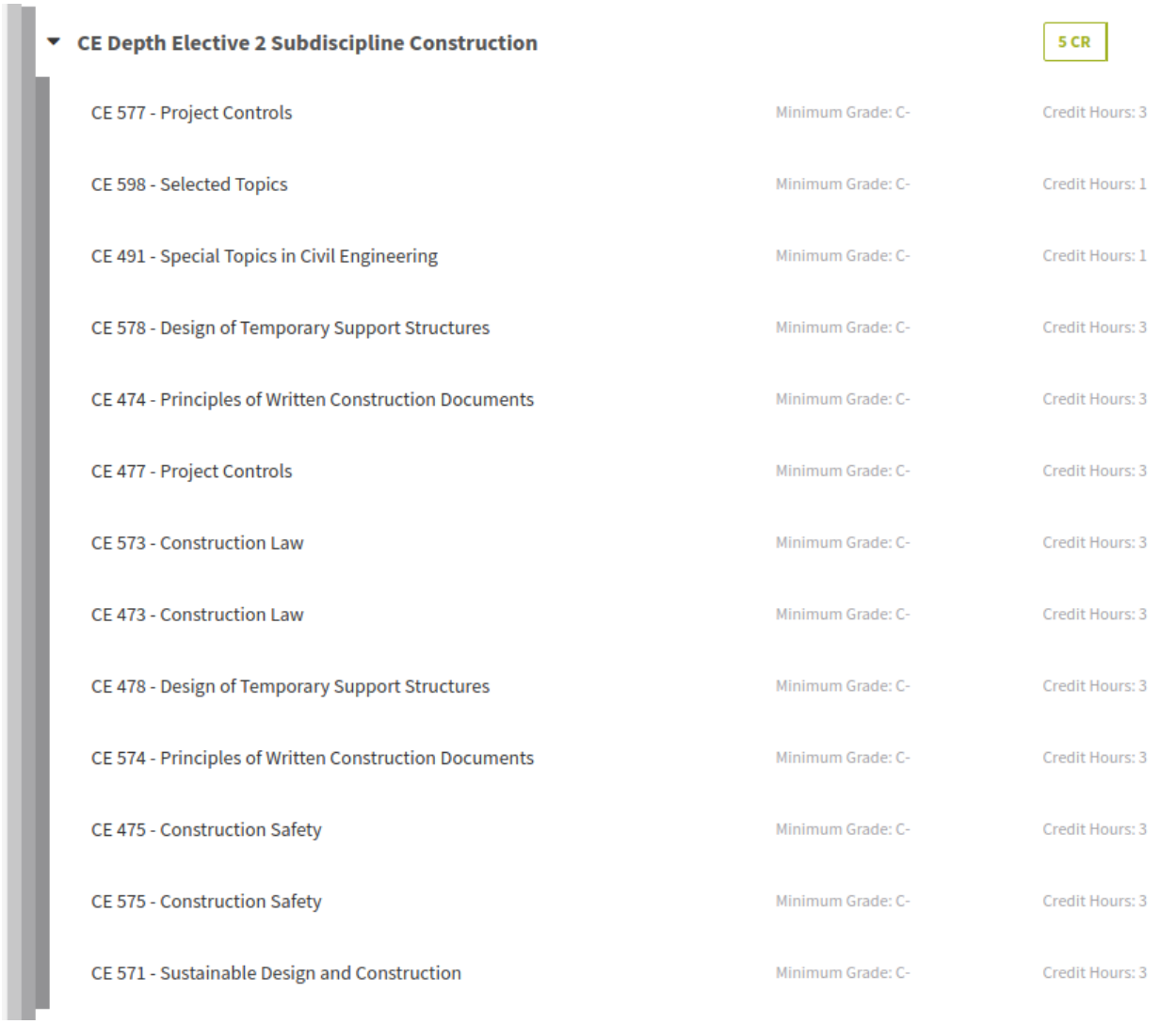
$$p = (\bar{C}_1, \bar{C}_1, \bar{C}_1, \bar{C}_2, \bar{C}_2, \bar{C}_3).$$

Once the paths have been created the credit hour requirement R_{ch} is replaced by

$$R_{ch} = p_1 \vee p_2 \vee ... \vee p_Q$$

and the *copy-and-replace* operation for disjunctive LC requirements in Procedure 1 can be applied.

Example 6. Consider the *CE Depth Elective 2 Subdiscipline Construction (5 CR)* requirement below.



In this case

$$R_{ch} = \text{ at least 5 credit hours from } Construction$$

In this example $\bar{C}_1$ has two 1-credit classes and $\bar{C}_3$ has eleven 3-credit classes. The only credit hour combinations that yield a total of (at least) 5-credit hours without including unnecessary classes are

$$H_1 = (3, 3)$$
$$H_2 = (3, 1, 1)$$

Thus, R_{ch} is replaced by

$$R_{ch} = p_1 \vee p_2$$

where

$$p_1 = \bar{C}_3 \wedge \bar{C}_3$$
$$p_2 = \bar{C}_3 \wedge \bar{C}_1 \wedge \bar{C}_1$$

6.4 Credit Hour and Number of Classes Requirements (*CHC*)

This section describes the *copy-and-replace* operation for a *credit hour and number of courses* requirement (*CHC*) that states

$$R_{chc} = \begin{array}{l}\text{the student must complete } M \text{ courses that contribute} \\ \text{(at least) } N \text{ credit hours from a specific category}\end{array}$$

where the category is witnessed by a list of classes. This type of requirement is represented by the three-tuple $R_{chc} = (M, N, \mathbf{C})$ where M is the number of classes, N is the minimum number of credit hours, and $\mathbf{C} = (c_1, c_2, ..., c_n)$ is a list of classes that count towards the specific category. We assume that this requirement means that the student must take exactly M courses, no more and no less. The *copy-and-replace* operation for this type of requirement is achieved using the following two steps.

1. Convert R_{chc} into a disjunctive *LC* requirement that takes the form of an OPR. That is, R_{chc} is converted into a list of paths and a requirement that (at least) one of the paths must be satisfied.

2. Then apply the *copy-and-replace* operation for disjunctive *LC* requirements in Procedure 1.

The conversion in Step 1 can be accomplished with the following steps.

1. Form the OCRs $\bar{C}_h = \{c_i : c_i \in \mathbf{C}, |c_i| = h\}$, that contain all h-credit hour classes that count towards the specific category.

2. Form $A = \{(h, n_h)\}$, i.e. the set of (number of credit hours, number of courses) tuples for the classes in $\mathbf{C}$.

3. Enumerate all credit hour combinations that occur in groups of M, excluding combinations where h appears more that n_h times. For example if $M = 2$ and

   $$A = \{(1, 10), (2, 10), (3, 10), (4, 10)\}$$

   then the enumeration is

   $$(1,1), (1,2), (1,3), (1,4), (2,2), (2,3), (2,4), (3,3), (3,4), (4,4) \tag{4}$$

   Note that if $(2, h_2) = (2, 1)$ instead of $(2, 10)$ then $(2, 2)$ is left out of the enumeration.

4. Retain only the combinations that provide at least N credit hours. For example if $N = 6$ then retain only

   $$(2,4), (3,3), (3,4), (4,4) \tag{5}$$

   from the combinations in Equation (4).

5. Use the OCRs $\bar{C}_h$ to create one path from each retained combination. For example the retained combinations in Equation (5) lead to the four paths

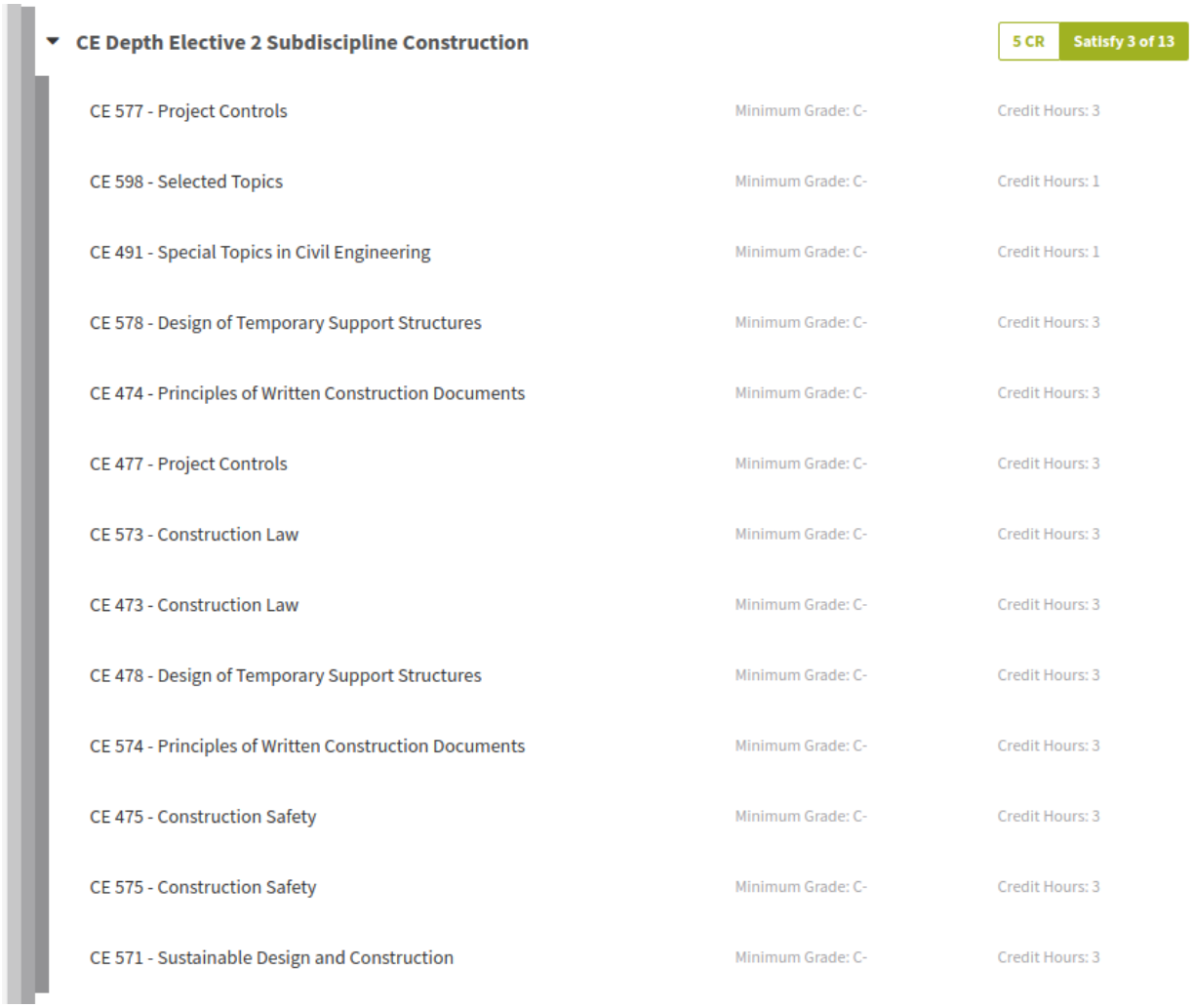
   $$\begin{aligned} p_1 &= \bar{C}_2 \wedge \bar{C}_4 \\ p_2 &= \bar{C}_3 \wedge \bar{C}_3 \\ p_3 &= \bar{C}_3 \wedge \bar{C}_4 \\ p_4 &= \bar{C}_4 \wedge \bar{C}_4 \end{aligned}$$

6. Once the paths have been created the credit hour requirement R_{chc} is replaced by the OPR

$$R_{chc} = p_1 \vee p_2 \vee ...$$

and the *copy-and-replace* operation for disjunctive LC requirements in Procedure 1 can be applied.

Example 7. Consider the *CE Depth Elective 2 Subdiscipline Construction (3 of 13) (5 CR)* requirement below.



In this case $R_{chc} = (3, 5, \mathbf{C})$ where $\mathbf{C}$ contains two 1-credit hour classes and eleven 3-credit hour classes. The credit hour combinations that occur in groups of $M = 3$, but exclude combinations where $h = 1$ appears more that $n_h = 2$ times, are

$$(1, 1, 3)$$
$$(1, 3, 3)$$
$$(3, 3, 3)$$

and all of these contribute at least $N = 5$ credit hours. Thus, we create three paths

$$p_1 = \bar{C}_1 \wedge \bar{C}_1 \wedge \bar{C}_3$$
$$p_2 = \bar{C}_1 \wedge \bar{C}_3 \wedge \bar{C}_3$$
$$p_3 = \bar{C}_3 \wedge \bar{C}_3 \wedge \bar{C}_3$$

and R_{chn} is replaced by the OPR

$$R_{chn} = p_1 \vee p_2 \vee p_3$$

6.5 Variable Credit Classes (*VC*)

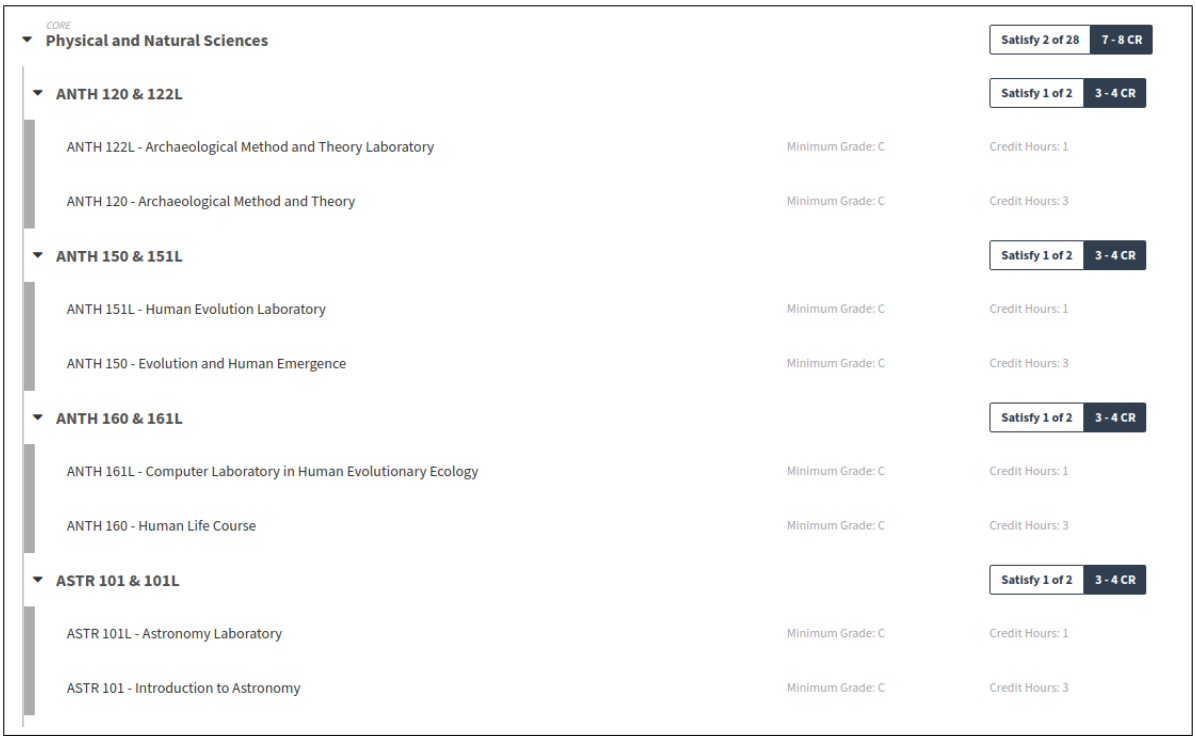
This section considers a special situation where an individual course can contribute a variable number of credit hours towards the degree. This situation may seem unlikely, or even illegal, but neither is true, and so we start by looking at a practical example. Consider the *Physical and Natural Sciences* requirement below. Only 13 of the 28 sub-requirements are shown (to save space), but all 28 take one of the forms shown, i.e. they are either an individual 3-credit class, an individual 4-credit class, or a 1-of-2 requirement. Here is a summary of the 28 sub-requirement types.

Requirement Type	number of instances
1-of-2	17
3-credit class	4
4-credit class	7

Normally a 1-of-2 requirement that provides 3-4 credits would imply that there are two classes to choose from, one 3-credit class and one 4-credit class. But in this example each of the 17 1-of-2 requirements involves a 3-credit class and its 1-credit lab (e.g. the first few are shown in the figure below). Thus, the two choices are obviously

- c = the 3-credit class alone, or
- cl = the 3-credit class with its 1-credit lab.

Note that in this situation the 1-credit lab counts *nothing* by itself, i.e. it must be paired with a specific 3-credit course before it can contribute to the satisfaction of a requirement[5]. So this situation is exactly as if the class itself can contribute a variable number of credits, either 3 or 4.



6.5.1 (Class, Lab) Replacement

When a class and its lab appears in the manner above, the first step is to replace the two original classes (3-credit class, 1-credit lab) with the two classes (3-credit class, 4-credit class+lab) $= (c, cl)$. Then c and cl are treated as two different classes. In doing so however, we must make sure that a student who has completed both the class and the lab does not get credit for both c and cl (i.e. 7 credits). This is accomplished by converting the student's class list as follows. If the student's class list contains both the class and the lab then these two classes are replaced by one completed class of the form (4-credit class+lab)[6]. This creates equivalent class types on both sides of the bipartite graph, and ensures that the student's class list will never match both c and cl. Thus, the individual class credits will never be counted more than once in a match.

There is one more situation that must be considered here. If the student's class list contains both the class and the lab, and there is another requirement, separate from the (Class, Lab) requirement, that can be satisfied by the class alone, then the student's new (4-credit class+lab) will be blocked from a possible match to this other requirement. The solution is to allow the (4-credit class+lab) to count towards both c and cl such that when the (4-credit class+lab) is matched to c it contributes 3 credits and when it is matched to cl it contributes 4 credits. Since the student's class list contains only one instantiation of this variable credit class, and the matching routine in Section 8 only allows (at most) one match for each class in the student's class list, there will never be a match to both c and cl.

[5]Indeed, the 1-credit lab cannot be paired with any other 3-credit classes. In addition, multiple 1-credit labs cannot be combined (by themselves) to satisfy multiple credits of the 7 credit requirement.

[6]If the student's class list contains only the class or the lab then no change is required.

6.5.2 *CC + VC*

Now let us return to the *Physical and Natural Sciences* requirement. At first glance this might look like a *CC* requirement of the form 2-of-28 that can be converted into a disjunction of $q = \binom{28}{2} = 378$ other requirements as described in Section 6.2.2. But this conversion is flawed because some of the paired combinations fail to meet the 7 credit hour requirement (i.e. some would contribute only $3 + 3 = 6$ credits). To remedy this we can enumerate all 378 cases and then reject the ones that fail to meet the 7 hour requirement. In this case however, we must be able to determine whether a 3-credit class paired with a 1-of-2 requirement meets the 7 hour requirement. Clearly this will only be true if the 1-of-2 requirement contributes 4 credits, i.e. it becomes a 1-of-1 requirement. Thus, this approach requires modification of the 1-of-2 requirement to a 1-of-1 requirement in some cases. Furthermore, we must be able to determine whether a pair of 1-of-2 requirements meets the 7 credit hour requirement. Clearly this will only be true if at least one of the 1-of-2 requirements contributes 4 credits. This can be enforced by replacing the pair of 1-of-2 requirements with a disjunction of the three (out of four) individual class parings that yield at least 7 credits. By rejecting combinations that fail to meet the 7 credit hour requirement, and modifying some of the 1-of-2 requirements, this approach yields a valid *copy-and-replace* operation. Consider the simple example below.

Example 8. Consider a (2-of-6, 7 CR) requirement where the 6 sub-requirements are

c_1 = 3-credit class
c_2 = 3-credit class
c_3 = 4-credit class
c_4 = 4-credit class
A = 1-of-2 where the two classes are a 3-credit class c_5 and its 1-credit lab l_5
B = 1-of-2 where the two classes are a 3-credit class c_6 and its 1-credit lab l_6

The first step is to replace classes in the 1-of-2 requirements as follows.

$$(c_5, l_5) \text{ is replaced by } (c_5, cl_5) \text{ where } cl_5 = c_5 + l_5$$
$$(c_6, l_6) \text{ is replaced by } (c_6, cl_6) \text{ where } cl_6 = c_6 + l_6$$

Now A and B become the OCRs

$$A = \{c_5, cl_5\}$$
$$B = \{c_6, cl_6\}$$

Next we enumerate all $\binom{6}{2} = 15$ combinations, and then reject or modify some as shown below.

Combination	Reject/Modify
$c_1 \wedge c_2$	reject
$c_1 \wedge c_3$	
$c_1 \wedge c_4$	
$c_1 \wedge A$	modify to $c_1 \wedge cl_5$
$c_1 \wedge B$	modify to $c_1 \wedge cl_6$
$c_2 \wedge c_3$	
$c_2 \wedge c_4$	
$c_2 \wedge A$	modify to $c_2 \wedge cl_5$
$c_2 \wedge B$	modify to $c_2 \wedge cl_6$
$c_3 \wedge c_4$	
$c_3 \wedge A$	
$c_3 \wedge B$	
$c_4 \wedge A$	
$c_4 \wedge B$	
$A \wedge B$	modify to $(c_5 \wedge cl_6) \vee (c_6 \wedge cl_5) \vee (cl_5 \wedge cl_6)$

In this example the (2-of-6, 7 CR) requirement has been converted into a disjunction of 14 un-rejected (and possibly modified) requirements shown above. Note that 13 of 14 requirements are simple conjunctions of two classes, or a class and an OCR. In addition, the non-simple replacement for $A \wedge B$ creates a nested disjunction that can be replaced by an un-nested disjunction by adding the 3 individual class parings (conjunctions) from $A \wedge B$ to the other 13 conjunctions. This yields an un-nested disjunction of $13 + 3 = 16$ conjunctions. Thus, in this example the *copy-and-replace* operation replaces the (2-of-6, 7 CR) requirement with the following disjunction of 16 conjunctions.

$$
\begin{array}{l}
(c_1 \wedge c_3) \ \vee \ (c_1 \wedge c_4) \ \vee \ (c_1 \wedge cl_5) \ \vee \ (c_1 \wedge cl_6) \ \vee \\
(c_2 \wedge c_3) \ \vee \ (c_2 \wedge c_4) \ \vee \ (c_2 \wedge cl_5) \ \vee \ (c_2 \wedge cl_6) \ \vee \\
(c_3 \wedge c_4) \ \vee \ (c_3 \wedge A) \ \vee \ (c_3 \wedge B) \ \vee \ (c_4 \wedge A) \ \vee \\
(c_4 \wedge B) \ \vee \ (c_5 \wedge cl_6) \ \vee \ (c_6 \wedge cl_5) \ \vee \ (cl_5 \wedge cl_6)
\end{array}
$$

Applying this approach to the *Physical and Natural Sciences* requirement yields the following statistics.

- A total of $\binom{28}{2} = 378$ combinations, with $\binom{4}{2} = 6$ rejections, leaving 372 combinations.
- A total of $4 \cdot 17 = 68$ modifications from 1-of-2 to 1-of-1.
- A total of $\binom{17}{2} = 136$ modifications that produce a disjunction of three 2-class conjunctions. Each of these 136 cases expands the number of combinations by 2 yielding a total of $372 + 2 \cdot 136 = 644$.

Thus, the *copy-and-replace* operation for the *Physical and Natural Sciences* requirement produces a disjunction of 644 conjunctions.

In this section we have shown how to incorporate one type of variable credit course into the *copy-and-replace* operation for *CC* requirements. This approach can produce a large number of combinations as illustrated in the *Physical and Natural Sciences* example. The next section describes a much more efficient approach.

6.5.3 *CHC + VC*

In this section we show how to incorporate (Class, Lab) variable credit courses into the *copy-and-replace* operation for *CHC* requirements. Note that the *Physical and Natural Sciences* requirement specifies both a number of courses and a number of credit hours, much like the *CHC* requirement described in Section 6.4. Indeed, if we assume that the (Class, Lab) replacement operations described above in Section 6.5.1 have already been performed then the 2-of-28 *Physical and Natural Sciences* requirement can be converted to a 2-of-45 *CHC* requirement with $(M, N, \mathbf{C}) = (2, 7, \mathbf{C})$, where the class list $\mathbf{C}$ consists of all $4 + 7 = 11$ original 3 and 4-credit classes plus all $17 \cdot 2 = 34$ of the 3 and 4-credit classes formed from the (Class, Lab) replacement operation. Note that this conversion requires the subtle replacement of nested disjunctions with un-nested disjunctions, e.g. $c_i \vee (c_j \vee cl_j)$ is replaced by $c_i \vee c_j \vee cl_j$. Then a direct application of the *copy-and-replace* operation in Section 6.4 replaces the *Physical and Natural Sciences* requirement with the disjunction

$$p_1 \vee p_2$$

where the 2 paths are

$$p_1 = \bar{C}_3 \wedge \bar{C}_4$$
$$p_2 = \bar{C}_4 \wedge \bar{C}_4$$

and $\bar{C}_3$ and $\bar{C}_4$ are the 3 and 4- credit OCRs formed from the individual classes in the list $\mathbf{C}$. Note that any class list that includes both c_j and cl_j will provide a 7-credit match to $p_1 = \bar{C}_3 \wedge \bar{C}_4$, but this will never happen when matching to a student class list because the (Class, Lab) replacement operations will only place c_j or cl_j, not both, in the student class list.

Note that the *copy-and-replace* operation described in this section is much simpler than the one in the previous section. It is also much more efficient. Indeed, in this section the *Physical and Natural Sciences* requirement is converted to a disjunction of only 2 conjunctions, while the conversion in the previous section yielded 644 conjunctions[7].

6.5.4 General *VC* Classes

In the previous sections we examined a special type of variable credit class. This section attempts to generalize to a broader group of variable credit classes. This broader group includes both individual classes that can contribute a variable number of credits, e.g. where the instructor includes more class content for more class credits, or classes whose total credit contribution depends on the optional completion of other classes, e.g. the (Class, Lab) example above. The general procedure for processing this group consists of the following four steps.

1. For requirements that include a variable credit class: replace the variable credit class c_v by an OCR $C_v = \{c_1, c_2, ...\}$ that contains multiple new classes, one for each credit-hour size.

2. For the student class list: modify the class list so that it contains (at most) one member of the variable credit OCR formed in Step 1. This member, c^*, should be the one with the largest number of credits supported by the student's class list. This may require combining multiple classes into one (as in the (Class, Lab) example above), or it may require re-formatting an individual member of the class list so that it matches the format in the variable credit OCR, or it may require no action at all.

[7]The conversion in the previous section yielded conjunctions of *smaller* OCRs, but the number of conjunctions has a much larger impact on the computational efficiency than the OCR sizes.

3. In many cases the *copy-and-replace* operations can now be carried out as before. However in some cases the requirements may need to be modified so that their action on the variable credit OCR has the same meaning as the original requirement. These modifications tend to be simple (and local). In fact the most common modifications have already been described in the previous two sections.

4. Let C_v^* be the subset of C_v that contains all members of C_v with $|c^*|$ credits or less. When creating the bipartite graph in Section 8 an edge is created between the student class c^* and all requirement path OCRs C_i that contain members of C_v^*. This allows c^* to count towards requirements that have instantiations of the variable credit class with fewer credits than c^*. In each case the class edge weight is set equal to

$$edge\ weight = \min(|c^*|, \max_{c \in C_i} |c|)$$

7 Requirements-to-Paths Examples for Select UNM Programs

This section describes the process of converting degree requirements to degree paths for a few programs at UNM.

Example 9. *Africana Studies:* For students entering in Fall 2018 this program has the following degree requirements.

- *Individual Class Requirements (24 CR):* The following 8 individual class requirements contribute a total of 24 credit hours.

AFST 102 (3 CR)	AFST 250 (3 CR)	AFST 284 (3 CR)	AFST 285 (3 CR)
AFST 329 (3 CR)	AFST 333 (3 CR)	AFST 388 (3 CR)	ENGL 120 (3 CR)

- OCR *Requirements (21 CR):* The OCR requirements for this program are listed below. They contribute a minimum of 21 credit hours. Note that the Mathematics OCR contains classes with 3, 4 and 5 credit hours while the requirement specifies a minimum of 3 credits.

OCR Requirement	Credits
AFST 299 or AFST 309 (1-of-2)	3
AFST 396 or HIST 474 (1-of-2)	3
Fine Arts (1-of-12)	3
Language (1-of-3)	3
Mathematics (1-of-11)	3
Writing and Speaking, Additional Course (1-of-5)	3
ENGL 110 or (111 and 112) or 113 (1-of-3)	3

- *Credit Hour Requirements (26 CR):* The *CH* requirements for this program are listed in the table below. They contribute a minimum of 26 credit hours. The list of courses that count toward each category is not specified in the supporting documentation so we have computed a maximum number of paths assuming the availability of an abundance of 1,2,3, and 4 credit courses. The actual number of paths for the Upper Division Electives is likely to be smaller.

CH Requirement	Credits	Number of paths
Elective (any course at any level)	2	4
Freshman Academic Choice (any course of any level)	3	5
Upper Division Elective(s)	21	139

- *Combinatorial Requirements (25 CR):* The *CC* requirements for this program are listed in the table below. They contribute a minimum of 26 credit hours.

CC Requirement	Credits	Number of paths
Humanities (2-of-32)	6	1 (496)
Literature and Culture (2-of-5)	6	1 (10)
Social and Behavioral Sciences (2-of-31)	6	1 (465)
Physical and Natural Sciences (2-of-28)	7	2 (644)

  The number of paths produced by the *copy-and-replace* operation for each of these requirements is shown in the right hand column. In the first three cases the number of paths in parentheses are obtained using the method in Section 6.2.2, but since the combined requirements are all individual classes the number of paths can be drastically reduced to 1 by applying the method in Section 6.2.1. The Physical and Natural Sciences is discussed at length in Section 6.5. The number of paths in parentheses are obtained using the method in Section 6.5.2, but the method in Section 6.5.3 produces only two paths.

- *Minor or Second Major (24 CR):* This requirement contributes a minimum of 24 credit hours. The specific requirements depend on the student's choice. In general, a majority of the credit hour requirements for a minor or second major come from individual class requirements. Thus, the *copy-and-replace* operations for these requirements usually do not contribute a large number of paths.

Using the ARP algorithm to enumerate all possible degree paths starts by creating a core path p_{core} that contains all of the individual class and OCR requirements, and then enumerating all combinations of *CH* and *CC* requirements. This gives a total of no more than $4 \cdot 5 \cdot 139 \cdot 2 = 5,560$ degree paths (before including the paths for the Minor or Second Major). Note that if the *CC* requirements are converted using the less efficient method then the total number of degree paths would be more like

$$4 \cdot 5 \cdot 139 \cdot 496 \cdot 10 \cdot 465 \cdot 644 = 4,129,194,048,000$$

i.e. approximately 4 trillion.

Example 10. *Psychology:* For students entering in Fall 2018 this program has the following degree requirements.

- *Individual Class Requirements (9 CR):* The following 3 individual class requirements contribute a total of 9 credit hours.

  ENGL 120 (3) PSY 302 (3)
  PSY 200 (3)

- OCR *Requirements (15 CR):* The OCR requirements for this program are listed below. They contribute a total of 15 credit hours.

OCR Requirement	Credits
Fine Arts (1-of-12)	3
Foreign Language (1-of-16)	3
Mathematics (1-of-11)	3
Writing and Speaking, Additional Course (1-of-5)	3
ENGL 110 or (111 and 112) or 113 (1-of-3)	3

- *Credit Hour Requirements (32 CR):* The *CH* requirements for this program are listed in the table below. They contribute a total of 32 credit hours. The list of courses that count toward each category is not specified in the supporting documentation so we have computed a maximum number of paths assuming the availability of an abundance of 1,2,3, and 4 credit hour courses. The actual number of paths is likely to be smaller in some cases.

CH Requirement	Max number of paths
Any Level Elective (8)	19
PSY upper division (12)	40
Upper Level Elective (6)	12
Any Level Psychology Course (3)	5
Freshman Academic Choice (3)	5

- *Combinatorial Requirements (31 CR):* The *CC* requirements for this program are listed in the table below. They contribute a minimum of 31 credit hours.

CC Requirement	Credits	Number of paths
Humanities (2-of-32)	6	1 (496)
Social and Behavioral Sciences (2-of-31)	6	1 (465)
PSY (220/240/260/265/271/280) (4-of-6)	12	1 (15)
Physical and Natural Sciences (2-of-28)	7	2 (644)

  The number of paths produced by the *copy-and-replace* operation for each of these requirements is shown in the right hand column. In the first three cases the number of paths in parentheses are obtained using the method in Section 6.2.2, but since the combined requirements are all individual classes the number of paths can be drastically reduced to 1 by applying the method in Section 6.2.1. The Physical and Natural Sciences is discussed at length in Section 6.5. The number of paths in parentheses are obtained using the method in Section 6.5.2, but the method in Section 6.5.3 produces only two paths.

- *Minor or Second Major (24 CR):* This requirement contributes a minimum of 24 credit hours. The specific requirements depend on the student's choice. In general, a majority of the credit hour requirements for a minor or second major come from individual class requirements. Thus, the *copy-and-replace* operations for these requirements usually do not contribute a large number of paths.

Using the ARP algorithm to enumerate all possible degree paths starts by creating a core path p_{core} that contains all of the individual class and OCR requirements, and then enumerating all combinations of *CH* and *CC* requirements. This gives a total of no more than $2 \cdot 5 \cdot 5 \cdot 12 \cdot 40 \cdot 19 = 456,000$ degree paths (before including the paths for the Minor or Second Major). Note that if the *CC* requirements are converted using the less efficient method then the total number of degree paths would be more like

$$5 \cdot 5 \cdot 12 \cdot 40 \cdot 19 \cdot 496 \cdot 15 \cdot 465 \cdot 644 = 507,979,987,200,000$$

i.e. approximately 507 trillion.

Example 11. *Computer Engineering:* For students entering in Fall 2018 this program has the following degree requirements.

- *Individual Class Requirements (73 CR):* The following individual class requirements contribute a total of 73 credit hours.

ECE 101 (1)	ECE 131 (3)	ECE 203 (3)	ECE 206L (2)
ECE 213 (3)	ECE 231 (3)	ECE 238L (4)	ECE 300 (4)
ECE 314 (4)	ECE 321L (4)	ECE 330 (3)	ECE 331 (3)
ECE 340 (3)	ECE 344L (4)	ECE 419 (3)	ECE 420 (3)
ECE 437 (3)	ECE 440 (3)	MATH 163 (4)	MATH 264 (4)
MATH 327 (3)	PHYS 160 (3)	ENGL 120 (3)	

- OCR *Requirements (15 CR):* The OCR requirements for this program are listed below. They contribute a total of 15 credit hours.

Fine Arts 1-of-12	3
Foreign Language 1-of-16	3
Mathematics 1-of-11	3
Writing and Speaking: ENGL 110 or (111 and 112) or 113	3
Writing and Speaking: Additional Course 1-of-5	3

- *Credit Hour Requirements (6 CR):* There is one *CH* requirement that contributes a total of 6 credit hours.

```
Tech Elective (6)
  Any MATH course numbered between 300 and 499.
  Any PHYS course numbered between 300 and 499.
  Any CS course numbered between 300 and 499.
  Any ECE course numbered between 300 and 499.
```

  The list of courses that count toward each category is not specified in the supporting documentation, but the maximum number of paths assuming the availability of an abundance of 1,2,3, and 4 credit hour courses is 12.

- *Combinatorial Requirements (25 CR):* The *CC* requirements for this program are listed in the table below. They contribute a minimum of 25 credit hours.

CC Requirement	Credits	Number of paths
Humanities (2-of-32)	6	1 (496)
Social and Behavioral Sciences (2-of-31)	6	1 (465)
Physical and Natural Sciences (2-of-28)	7	2 (644)
Track Elective (1-of-2)	6	2

The number of paths produced by the *copy-and-replace* operation for each of these requirements is shown in the right hand column. In the first two cases the number of paths in parentheses are obtained using the method in Section 6.2.2, but since the combined requirements are all individual classes the number of paths can be drastically reduced to 1 by applying the method in Section 6.2.1. The Physical and Natural Sciences is discussed at length in Section 6.5. The number of paths in parentheses are obtained using the method in Section 6.5.2, but the method in Section 6.5.3 produces only two paths.

Using the ARP algorithm to enumerate all possible degree paths starts by creating a core path p_{core} that contains all of the individual class and OCR requirements, and then enumerating all combinations of *CH* and *CC* requirements. This gives a total of no more than $2 \cdot 2 \cdot 12 = 48$ degree paths. Note that if the *CC* requirements are converted using the less efficient method then the total number of degree paths would be more like

$$2 \cdot 644 \cdot 496 \cdot 465 \cdot 12 = 3,564,771,840$$

i.e. approximately 3.6 billion.

8 Matching

This section shows how to compute the *max-match* between a path and a student's class-list. A *max-match* is a one-to-one pairing between OCRs and classes that maximizes the number of credit hours contributed by the paired classes. The first section below treats the special case where the path is a list of individual classes. The next section treats the more general (and more common) case where the path is an OCR-list.

8.1 Paths of Individual Courses

In this case the path is the conjunction of individual classes, i.e.

$$p = (c_1, c_2, ..., c_K)$$

Given p and the student class list $L = \{c_1, c_2, ..., c_M\}$ the max-match value is obtained by simply summing the credit hours associated with the courses they have in common.

8.2 Paths of OCRs

In this section we consider the case where the path is an OCR-list, i.e.

$$p = (C_1, C_2, ..., C_K)$$

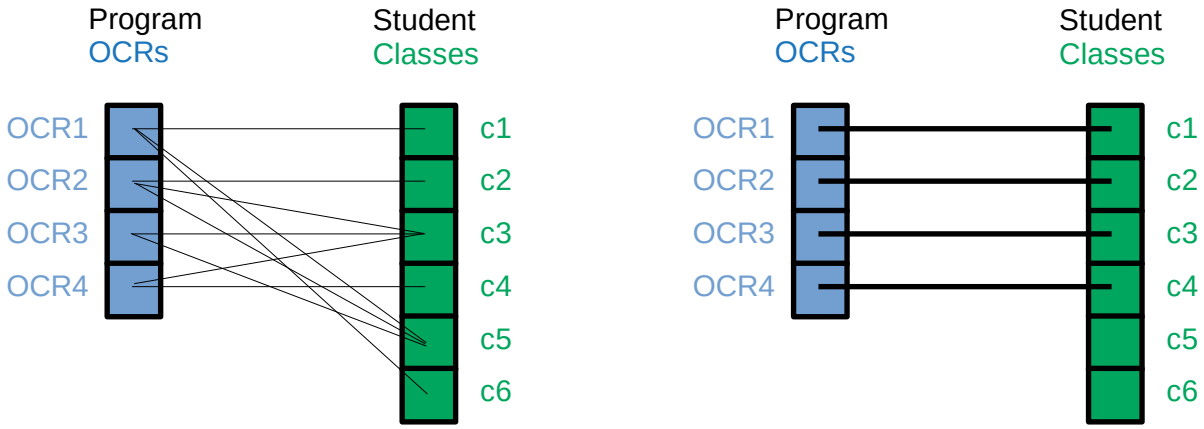
We seek the max-match between p and the student class list $L = \{c_1, c_2, ..., c_M\}$. This problem is complicated by the fact that multiple classes can satisfy a single OCR, and multiple OCRs can

be satisfied by a single class, as illustrated by the bipartite graph in Figure 9a. Finding a one-to-one match that maximizes the number of credit hours contributed by matched classes can be accomplished by solving the *maximum weighted bipartite matching* problem described below.

- First form a bipartite graph where each node on the left corresponds to an OCR from the path $p = (C_1, C_2, ..., C_K)$ and each node on the right corresponds to a class from the list $L = \{c_1, c_2, ..., c_M\}$. An edge is placed between each C_i on the left, and each c_j on the right that is a member of C_i. That is, for each C_i we compute $L_{C_i} = C_i \cap L$ and place an edge between C_i and each class $c \in L_{C_i}$. Then we assign an edge weight equal to the number of credit hours in the class c.

  Note that if the path OCRs contain classes that have been derived from *variable credit classes* (e.g. see Section 6.5) then the student class list should be pre-processed as described in Section 6.5.4 and the edge weights should be set according to the method described in that section. In particular, if the variable credit class stems from a (Class) or (Class, Lab) requirement, then the student class list should be pre-processed as described in Section 6.5.1.

- Once the bipartite graph is formed we solve the *maximum weighted bipartite matching* problem. An exact solution to this problem can be obtained by a well-known computationally efficient algorithm such as Edmond's Algorithm[8]. Then the max-match value is equal to the optimal solution value, i.e. the sum of the edge weights in the optimal match. An example is shown in Figure 9b.



(a) Bipartite Graph with OCRs and Classes.

(b) Max-match solution.

Figure 9: Bipartite Matching Example

Often the *max-match* solution is not unique, and in practice some optimal solutions are preferred over others. Furthermore, there is often a strong desire to prioritize the requirements to which individual classes can be matched. For example, a core class can often satisfy both an elective requirement and a core requirement, but we generally prefer that it be matched to the core requirement. Giving preferences to tied solutions, and prioritizing requirement matches, can be accomplished by adjusting the edge weights of the bipartite graph. In particular the list of requirements from Step 3 of the ARP algorithm $(p_{core}, R_1, R_2, ..., R_K)$ can be ordered from highest priority

[8]Edmonds, Jack (1965). "Paths, trees, and flowers". Can. J. Math. 17: 449467. doi:10.4153/CJM-1965-045-4.

to lowest priority, and then decreasing weights can be assigned to the ordered requirements[9]. This ordering can be defined by the user, or it can be automated. One ordering that may automatically reflect the end user's desire is to assign the highest priority to requirements contain the least number of paths. Another option is to assign the highest priority to requirements that can be satisfied by the least number of classes. Once the weights have been assigned to the requirements the edge weights of the bipartite graph are adjusted as follows. Each edge weight is now the product of the class credit hour weight and the requirement priority weight. This will encourage classes to be matched to higher priority requirements whenever possible.

9 Approximation Algorithms

The ARP Algorithm is guaranteed to produce an optimal result, but the number of degree paths generated by this algorithm may be excessive. In this case we consider the following options.

1. *Parallel Processing:* The ARP algorithm is easily parallelized, e.g. by partitioning the requirement path combinations into distinct groups and computing the max-match for each group separately. This approach may reduce computation time by multiple orders of magnitude.

2. *Approximation Algorithms:* Approximation algorithms employ heuristics that can substantially reduce the amount of computation, but may produce sub-optimal solutions. The goal is to develop heuristics that yield an efficient algorithm with a high probability of producing an optimal (or near-optimal) solution. Some examples are described below.

   - *Most Likely Heuristic:* This heuristic reduces the number of paths in the requirement OPRs by eliminating paths that have been rarely used by students in the past. For example, it may be extremely rare for a *CH* requirement that requires 20 credits to be satisfied by a student that has completed 20 1-hour classes, and so this path may be removed from the corresponding OPR. This is a data driven approach that requires the accumulation of requirement paths that have been completed by students in the past, and then a statistical analysis of this data.
   - *Path Elimination Heuristic:* This heuristic reduces the number of paths in the requirement OPRs by eliminating paths that are not well matched to the student's courses. The details of this approach are described in Section 9.1
   - *Iterative Match Heuristic:* This heuristic attempts to synthesize the best degree path from scratch by starting with a small (incomplete) path and iteratively adding OCRs to the path until the max-match value between the student's classes and the current path can no longer be improved. This approach avoids the combinatorial enumeration of paths carried out by the ADP and ARP algorithms and therefore yields in a substantial reduction in computation. But because it is greedy it can produce sub-optimal solutions. A detailed description of this method is provided in Section 9.2.

9.1 Path Elimination

The *path elimination* algorithm inserts the following procedure between Steps 3 and 4 of the ARP algorithm. This procedure limits the number of paths in each OPR to M (a user specified value), thereby giving the user direct control over the total number degree path combinations in Step 4.

[9] It is probably best if all these weights are close to 1, i.e. between 0.9 and 1.0.

M = max number of paths to keep in each OPR {user defined parameter}
for (each P_i from $(P_1, P_2, ..., P_K)$) **do**
  for (each path $p_j \in P_i$) **do**
    $m_j \leftarrow$ max-match value between p_j and the student's class list
  end for
  remove all (m_j, p_j) pairs where $m_j = 0$
  sort the remaining (m_j, p_j) pairs by the max-match values m_j from largest to smallest
  $P_i \leftarrow$ the first M paths in the sorted list
end for

9.2 The Iterative Match Algorithm

This section describes the *Iterative Match* algorithm (IM) which attempts to discover the best degree path by starting with a small (incomplete) path and iteratively adding OCRs until the max-match value between the student's classes and the current path can no longer be improved. We start with the following definitions.

- Let $L = \{c_1, c_2, ..., c_M\}$ be the list of classes completed by the student.
- Let $R_1, R_2, ..., R_K$ be the non-path requirements determined in Step 1 of the ARP algorithm and let p_{core} be the core path formed in Step 2 of the ARP algorithm.
- Let h_{req_i} be the minimum number of credit hours associated with requirement R_i.
- Let $\mathbf{C}_i$ be the list of classes that count towards the non-path requirement R_i.
- Let $\bar{C}_{i,h} = \{c_j : c_j \in \mathbf{C}_i, |c_j| = h\}$, be the OCR containing all h-credit classes that count towards requirement R_i.
- Let p_i be a *partial path* for requirement R_i consisting of a combination of the OCRs $(\bar{C}_{i,1}, \bar{C}_{i,2}, \bar{C}_{i,3}, \bar{C}_{i,4})$. This partial path may contain any combination of $(\bar{C}_{i,1}, \bar{C}_{i,2}, \bar{C}_{i,3}, \bar{C}_{i,4})$, including multiple instances of some or all of them.
- Let the partial degree path $p_D = p_{core} \odot p_1 \odot p_2 \odot ... \odot p_K$ be a concatenation of p_{core} and the partial paths p_i.
- Let `MaxMatch`(p_D, L) be the routine that computes the max-match between a partial degree path p_D and the class list L. We assume that this routine returns the following information.
  - the maximum match value (i.e. number of credit hours in the max-match)
  - a sub-path $p_m = (p_{m,core}, p_{m,1}, p_{m,2}, ..., p_{m,K})$ of matched paths where
    - $p_{m,core}$ is a sub-path of matched OCRs from p_{core}
    - $p_{m,i}$ is a sub-path of matched OCRs from partial path p_i
  - a list $L_m = (L_{m,core}, L_{m,1}, L_{m,2}, ..., L_{m,K})$ of matched class lists where
    - $L_{m,core}$ is a list of classes matched to the OCRs from $p_{m,core}$
    - $L_{m,i}$ is a list of classes matched to the OCRs from $p_{m,i}$

The basic strategy is to apply the max-match algorithm to an iteratively generated collection of partial paths. Each partial path is initialized with one OCR for each credit hour size, e.g.

$$p_i = (\bar{C}_{i,1}, \bar{C}_{i,2}, \bar{C}_{i,3}, \bar{C}_{i,4})$$

and then an OCR is iteratively added to each partial path for each h-credit size that is *in-demand*, and this process continues until the max-match value no longer increases. An *in-demand* credit hour size is defined as follows. Let $p_{u,i}$ be the list of unmatched OCRs for requirement R_i, i.e. OCRs that are in p_i but not in $p_{m,i}$. An OCR is *in-demand* if does *not* the appear in the unmatched list. The fact that $\bar{C}_{i,h}$ does not appear in $p_{u,i}$ suggests that there is a chance that, if we add it to the path p_i, it could be matched at the next iteration. Thus, the list of in-demand OCRs is given by

$$p_{demand,i} = \text{ list of OCRs from } (\bar{C}_{i,1}, \bar{C}_{i,2}, \bar{C}_{i,3}, \bar{C}_{i,4}) \text{ that are not in } p_{u,i}$$

Algorithm 2 Iterative Match Algorithm

```
1: {initialize the partial paths}
2: for (i = 1, 2, ..., K) do
3:    p_i ← (C̄_{i,1}, C̄_{i,2}, C̄_{i,3}, C̄_{i,4})
4: end for
5:
6: {compute the initial match}
7: p_D ← (p_core, p_1, p_2, ..., p_K)
8: (match_value, p_m, L_m) ← MaxMatch(p_D, L)
9: (p_D, match_value, p_m, L_m) ← RemoveExcessOCRs(p_D, p_m, L_m)
10:
11: {add OCRs to the partial paths until the match value no longer increases}
12: repeat
13:    prev_match_value ← match_value
14:    for (i = 1, 2, ..., K) do
15:       h_i ← number of credit hours in the p_{m,i}
16:       if (h_i < h_{req_i}) then
17:          p_{demand,i} ← list of OCRs from p_i that are in demand
18:          for (each C̄_{i,h} ∈ p_{demand,i}) do
19:             p_i ← (p_i, C̄_{i,h})    {add OCR to partial path, give it a chance to increase its match}
20:          end for
21:       end if
22:    end for
23:    p_D ← (p_core, p_1, p_2, ..., p_K)
24:    (match_value, p_m, L_m) ← MaxMatch(p_D, L)
25:    (p_D, match_value, p_m, L_m) ← RemoveExcessOCRs(p_D, p_m, L_m)
26: until (match_value ≤ prev_match_value)
27:
28: return(p_m, L_m)
```

The *Iterative Match Algorithm* is shown in Algorithm 2. After the initialization steps the algorithm enters a repeat-until loop that adds OCRs to the partial paths until the match value

no longer increases. Note that every call to the `MaxMatch` algorithm is followed by a call to the `RemoveExcessOCRs` algorithm[10]. This is necessary because the addition of OCRs is done without regard for the number of credits specified by the requirements, and therefore the `MaxMatch` algorithm may produce *over-satisfied* requirements, i.e. it may produce matched paths that contain excess OCRs. For example consider a 6-credit requirement whose initial partial path is $p_i = (\bar{C}_{i,1}, \bar{C}_{i,2}, \bar{C}_{i,3}, \bar{C}_{i,4})$. A student's classes may match all 4 OCRs for a total of 10 credits, but only 6 of these credits are essential, and 4 credits are in excess. The `RemoveExcessOCRs` algorithm removes excess OCRs, and then computes a new match that allows classes that were previously matched to excess OCRs to match elsewhere. This process of removing excess OCRs and re-computing the best match is repeated until there are no remaining excess OCRs.

Note that there may be multiple options when removing excess OCRs. For example, with the 6-credit requirement where all 4 OCRs in the partial path $p_i = (\bar{C}_{i,1}, \bar{C}_{i,2}, \bar{C}_{i,3}, \bar{C}_{i,4})$ are matched, we could remove $\bar{C}_{i,4}$, or $(C_{i,1}, \bar{C}_{i,2})$, or $(C_{i,1}, \bar{C}_{i,3})$. Since OCR removal frees up classes to match elsewhere it is important to choose a removal option that frees the classes most likely to match elsewhere. The `RemoveExcessOCRs` algorithm in Algorithm 3 accomplishes this by giving removal priority to classes that are *linked* to the largest number of unmatched OCRs[11].

Algorithm 3 Remove Excess `OCR`s

```
1: RemoveExcessOCRs(p_D, p_m, L_m)
2: repeat
3:    {Initialization: determine the number of alternative matches for each matched class}
4:    p_u ← the list of unmatched OCRs, i.e. OCRs that are in p_D but not in p_m
5:    for (each matched class c from the lists in L_m) do
6:       n_c ← number of OCRs from p_u that class c is linked to
7:    end for
8:    {now remove excess OCRs from any over-matched requirements}
9:    excess ← FALSE
10:   for (each requirement R_i, i = 1, 2, ..., K) do
11:      h_i ← number of credit hours in p_{m,i}
12:      {Note: (C̄, c) denotes a matched OCR-class pair from (p_{m,i},L_{m,i})}
13:      for (each (C̄, c) pair from (p_{m,i},L_{m,i}) in order from largest n_c value to smallest) do
14:         if ((h_i− number of credits in C̄) ≥ h_{req_i}) then
15:            remove C̄ from p_i
16:            h_i ← h_i− number of credits in C̄
17:            excess ← TRUE
18:         end if
19:      end for
20:   end for
21:   p_D ← (p_core, p_1, p_2, ..., p_K)
22:   (match_value, p_m, L_m) ← MaxMatch(p_D, L_m)
23: until (excess = FALSE)
24: return(p_D, match_value, p_m, L_m)
```

[10] This includes the very first call to `MaxMatch` during the initialization steps.

[11] Here a *link* corresponds to an edge that is present in the input graph to the `MaxMatch` algorithm but not present in the output (match) graph.

10 Analyzing the Best Max-Match Solutions

Sections 5-9 have described a complete algorithm that accepts a student class list and a set of degree requirements as input, and produces a best max-match (BMM) solution as output. The sections below describe subsequent analyses that are a by-product of the BMM solution. Section 10.1 defines various student progress measures that can be used to quantify student progress, and Section 10.2 describes methods for decomposing the BMM solution into components that support the credit efficiency analysis, transfer credit analysis, and requirement analysis described in Section 4.

10.1 Quantifying Student Progress

There are (at least) two different types of student assessments that may be of interest.

1. The first assessment quantifies the student's *progress towards completion* of the degree program. To define a viable progress measure we start with the following definitions.

   - Let $R_1, R_2, ..., R_K$ be the non-path requirements determined in Step 1 of the ARP algorithm and let $R_0 = p_{core}$ be the core path formed in Step 2 of the ARP algorithm.
   - Let h_{req_i} be the minimum number of credit hours required to satisfy requirement R_i, and $h_{total} = h_{req_0} + h_{req_1} + h_{req_2} + ... + h_{req_K}$.
   - Let $p_{m,i}$ be the list of OCRs from the best max-match solution that count towards requirement R_i, and let $h_{m,i}$ be the total number of credit hours in $p_{m,i}$.

   It is tempting to define progress as the ratio of the number of matched credit hours to the minimum number of credit hours for the degree program, i.e. $\sum_{i=0}^{K} h_{m,i}/h_{total}$. But this ratio can exceed 1, even when some of the requirements are not yet satisfied. This can happen because a requirement may be satisfied by a collection of matched courses whose total credit hours exceed the minimum, i.e. $h_{m,i} > h_{req_i}$. A better measure of progress is obtained as follows. Define the progress towards requirement R_i as follows

   $$\rho_i = \frac{\min(h_{m,i}, h_{req_i})}{h_{req_i}}$$

   and then define the overall student progress

   $$\rho = \frac{\sum_{i=0}^{K} h_{req_i}\rho_i}{h_{total}} = \frac{\sum_{i=0}^{K} \min(h_{m,i}, h_{req_i})}{h_{total}}$$

   This measure always takes a value between 0 and 1, and is equal to 1 if and only if all requirements are satisfied.

2. The second assessment quantifies the *efficiency* of the student's path at the institution. The *graduation efficiency index* (GEI)[12] is a common way to quantify efficiency, but it only applies to *graduates* and therefore does not require the methods developed in this report. An efficiency measure that is more relevant to current students is

   $$\text{efficiency} = \frac{\text{number of completed credit hours that count towards the degree}}{\text{number of completed credit hours}}$$

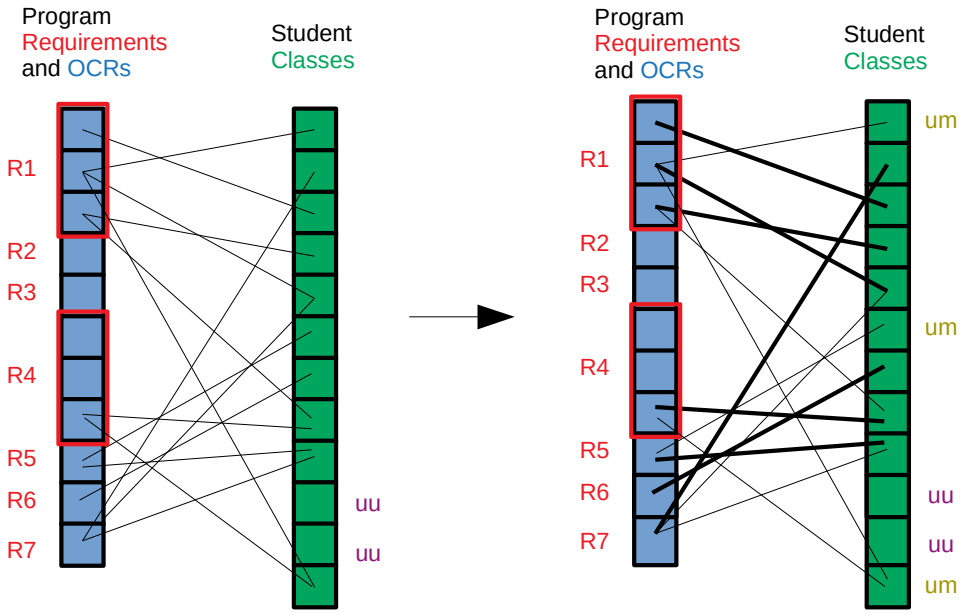
[12] G.M. Gillmore and P.H. Hoffman, "The Graduation Efficiency Index: Validity and Use as an Accountability and Research Measure", *Research in Higher Education*, Vol. 38, No. 6, pp. 677–697, 1997.

where the denominator might also included attempted (but not completed) credit hours. The determination of this measure *does* require the methods developed in this report.

10.2 Decomposing the Best Max-match Solution

This section describes three BMM decompositions that support the credit efficiency analysis, transfer credit analysis, and requirement analysis described in Section 1. The first decomposes the requirements into the four categories: unsatisfied, partially satisfied, satisfied and over-satisfied. This decomposition is described in Section 10.2.1. The second decomposes the credits into the three categories: applied, unusable, and excess. This decomposition determines only the *number* of credits in each category, and is described in Section 10.2.2. The third decomposition attributes specific classes to each of the three credit categories, and therefore allows us to determine the characteristics of each category. For example the third decomposition allows us to compute the fraction of unusable credits that are transfer credits. While the BMM solution produces a unique *number* of applied, unusable, and excess credits, the specific course-to-requirement assignments may not be unique. Indeed, there may be many BMM solutions that assign classes to requirements in different ways but produce the same number of applied, unusable, and excess credits. To obtain a specific type of BMM solution we swap the original BMM solution for a different BMM solution that attributes classes according to a prioritization rule. This prioritized decomposition is described in Section 10.2.3.

The BMM solution is derived from a bipartite graph that has OCRs on one side and completed classes on the other. An example bipartite graph is shown on the left below. OCRs are colored blue, student classes are colored green, and program requirements[13] are outlined in red. The graph contains an edge between each class and every OCR that the class could possibly satisfy. Note that some of the classes cannot satisfy any OCR, i.e. they are *unusable*. This is the *input graph* to the *maximum weighted bipartite matching* algorithm.



[13]Program requirements are represented as OCR lists (i.e. paths).

A BMM solution is a *one-to-one match* that determines the maximum number of completed credits towards the degree program *without excess*. An example is shown on the right above. The darker edges represent the (OCR, class) matches that form the BMM solution. We refer to the sub-graph that contains only the darker edges as the *output graph*. The lighter edges (on the right) represent the input graph edges that are unused by the BMM solution. We refer to these unused edges as *links*. There are three classes that possess only *linked* edges after the match. We call these the *unmatched* classes. The next three sections refer back to the input and output graphs above to help illustrate their decompositions.

10.2.1 Decomposition into Requirement Types

Once a BMM solution has been computed the following three types of OCRs can be determined directly from the input and output graphs.

1. **unsatisfied OCR:** An OCR is unsatisfied if it is unmatched in the output graph, e.g. R2:OCR1 and R3:OCR1 in the example above.

2. **satisfied OCR:** An OCR is satisfied (without excess) if it is matched in the output graph and contains no links to unmatched classes, e.g. R6:OCR1 and R7:OCR1 in the example above. Note that R7:OCR1 contains two links, but both are to matched classes and therefore do not contribute to an over-satisfaction of R7:OCR1.

3. **over-satisfied OCR:** An OCR is *over-satisfied* if it is matched in the output graph and contains links to unmatched classes, e.g. R5:OCR1 and R1:OCR2 in the example above.

In addition, the original requirements can be placed into four categories as follows.

1. **unsatisfied requirement:** A requirement is unsatisfied if all of its OCRs are unsatisfied, e.g. R2 and R3 in the example above.

2. **partially satisfied requirement:** A requirement is partially satisfied if some of its OCRs are unsatisfied and some are either satisfied or over-satisfied, e.g. R4 in the example above.

3. **satisfied requirement:** A requirement is satisfied (without excess) is all of its OCRs are satisfied (and none are over-satisfied), e.g. R6 and R7 in the solution above.

4. **over-satisfied requirement:** A requirement is over-satisfied if all of its OCRs are either satisfied or over-satisfied with at least one of its OCRs being over-satisfied. e.g. R1 and R5 in the example above.

10.2.2 Decomposition into Credit Type Sizes

A BMM solution can be used to determine the total number of applied, unusable, and excess credits as follows.

1. **number of unusable credits:** The total number of unusable credits is obtained by summing the credits associated with the unusable classes, i.e. summing the credits associated with classes that have no edges in the input graph.

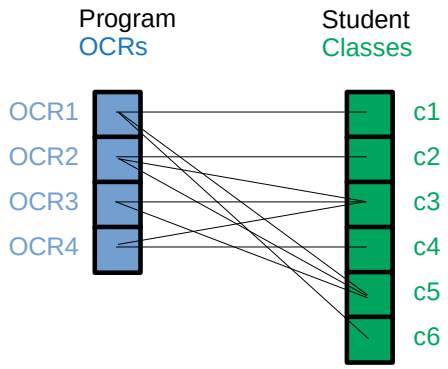
2. **number of applied credits:** The total number of applied credits is obtained by summing the credits associated with the matched classes in the output graph.

3. **number of excess credits:** The total number of excess credits is obtained by summing the credits associated with the unmatched classes, i.e. by summing the credits that have at least one link to an OCR but are unmatched in the output graph.

10.2.3 Decomposition into Prioritized Credit Types

This section shows how to attribute specific classes (or class types) to the three credit categories: unusable, applied, and excess. This enables a more detailed assessment of the student's credit hours by exploring the characteristics of the classes placed in each category. The BMM solution provides a default assignment of classes to these categories, but there may be other optimal solutions that are better suited to the credit analysis goal. For example, we may prefer that the earliest completed classes are given the highest match priority, i.e they are given the best chance to end up in the applied group. Alternatively we may want to prioritize transfer classes, or institutional classes, or some other class type. This will be accomplished by replacing the original BMM solution with a different BMM solution whose matched classes are determined according to a prioritization rule. This solution replacement will be accomplished by a sequence of exchanges (or swaps) between matched and unmatched classes, where each swap produces a different BMM solution with the same optimal match score. We develop this solution replacement process in three steps.

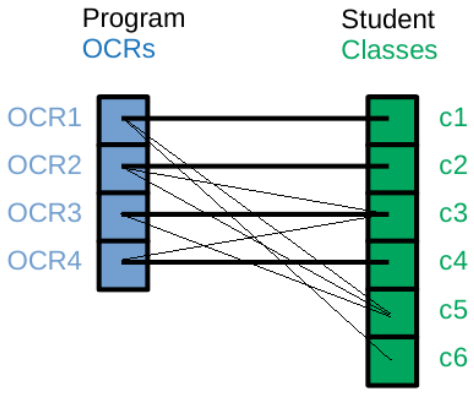
1. The first step is to place the matched and unmatched classes for each OCR into an *exchangeable class group*. The exchangeable class group is defined to be the set of classes that are linked to the OCR in the input graph. Consider the input graph shown below.



The exchangeable class groups in this graph are shown in the table below.

Requirement	Exchangeable Class Group
OCR1	c1,c5,c6
OCR2	c2,c3,c5
OCR3	c3,c5
OCR4	c3,c4

2. The second step is to describe the process of performing a swap. Starting with a BMM solution, we must first identify the classes that are eligible for an immediate swap. This is accomplished by partitioning the exchangeable class group into three types of classes: *matched-here*, *excess*, and *matched-elsewhere*. For example, consider the BMM solution shown below. The table below the graph shows how to partition the classes.



Req	Matched Here	Excess	Matched Elsewhere
OCR1	**c1**	c5 c6	
OCR2	**c2**	c5	c3
OCR3	**c3**	c5	
OCR4	**c4**		c3

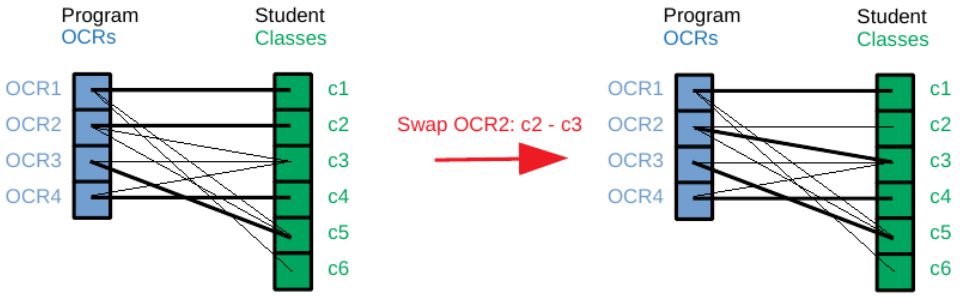
Swaps are performed between two *matched-here* classes, or between a *matched-here* class and an *excess* class. After the swap the exchangeable class groups must be re-partitioned. An example of swapping c3 and c5 in OCR3 is shown below. Note that after the swap c3 must be moved into the *excess* category for OCR2-OCR4, and c5 must be moved into the *matched-elsewhere* category for OCR1-OCR2.

OCR1	**c1**	c5 c6	
OCR2	**c2**	c5	c3
OCR3	**c3**	c5	
OCR4	**c4**		c3

→

OCR1	**c1**	c6	c5
OCR2	**c2**	c3	c5
OCR3	**c5**	c3	
OCR4	**c4**	c3	

An additional swap of c2 and c3 in OCR2 is shown below. After this swap c3 must removed from the *excess* category for OCR2-OCR4, and added to the *matched-elsewhere* category for OCR3-OCR4.



OCR1	**c1**	c6	c5
OCR2	**c2**	c3	c5
OCR3	**c5**	c3	
OCR4	**c4**	c3	

→

OCR1	**c1**	c6	c5
OCR2	**c3**	c2	c5
OCR3	**c5**		c3
OCR4	**c4**		c3

3. The third step in our development is to derive an algorithm that performs these swaps according to a user-specified prioritization rule. Algorithm 4 is one possible solution. This algorithm swaps eligible classes in order from highest to lowest priority, and each swap is with the highest priority *matched* class from the eligible OCRs. Line 4 places all classes to be considered for exchange into the set C_{exch}. C_{exch} may consist of all linked classes (including *matched* classes), i.e. all classes that could possibly participate in an exchange. However in some cases we may wish to restrict the membership of C_{exch}. For example, the number of *matched* transfer classes can be maximized by restricting C_{exch} to transfer classes only. A similar approach can be used to maximize the number of *matched* UNM classes. Line 5 uses a prioritization rule to determine the order in which the classes are considered for swap. Example prioritization rules include:

   - *First-come First-serve (FCFS):* order by the date the class was completed from earliest to latest
   - *Last-come First-serve (LCFS):* order by the date the class was completed from latest to earliest
   - *Lower-to-Upper (L2U):* order by the classes level from lower division to upper division
   - *Upper-to-Lower (U2L):* order by the classes level from upper division to lower division

   Line 5 may also use a multi-level prioritization rule. For example the L2U+FCFS rule would first order the classes from lower to upper division, and then each same-division group would be ordered from earliest to latest. Algorithm 4 is a greedy algorithm, and its ability to produce a (new) BMM solution with the absolute highest priority remains open.

Algorithm 4 Algorithm for producing a prioritized BMM solution.

1: Form the exchangeable class group for each OCR
2: Given the initial BMM solution, partition the exchangeable class groups into
3: *matched-here*, *excess*, and *matched-elsewhere*
4: $C_{exch} \leftarrow$ set of classes to be considered for exchange
5: Order the classes in C_{exch} by the prioritization rule
6: **for** (each class $c \in C_{exch}$ in order) **do**
7: extract OCRs linked to c and order them according to prioritization of their *matched* class
8: **for** (each OCR in order) **do**
9: $c_{match} \leftarrow$ the OCR's *matched* class
10: **if** (c has a higher priority than c_{match}) **then**
11: swap c and c_{match} and re-partition all exchangeable class groups
12: exit the inner **for** loop (and move to the next class in C_{exch})
13: **end if**
14: **end for**
15: **end for**

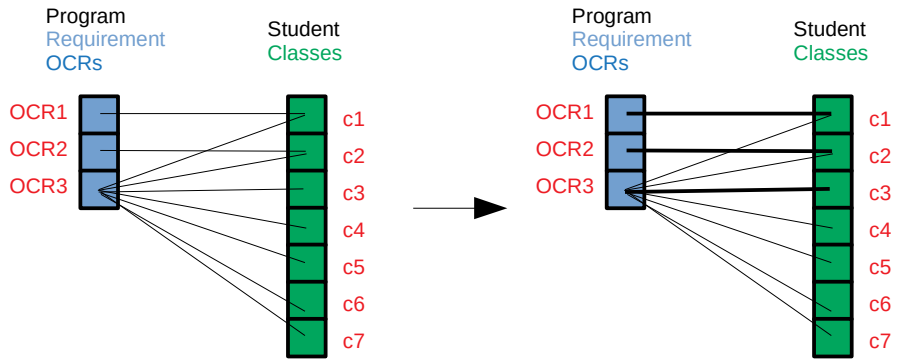
It is clear that Algorithm 4 retains the same matched OCRs as the original BMM solution. But the matched classes may be different, and so we still need to verify that the new solution has the same optimal match score. This will be accomplished by proving that each individual swap maintains the optimal match score.

Recall that the match score is equal to the sum of the matched edge weights, and that each edge weight is equal to the product of the corresponding OCR and class weights. Since a swap only changes the matched class (not the matched OCR), the match score will only change if the class weight changes. Furthermore, since the class weight is equal to the number of credits in the class, the match score will only change if the two swapped classes have different numbers of credits. But the only time that swapped classes can have different numbers of credits is when the OCR class list contains classes with different numbers of credits, and this only happens in some very special cases, e.g. VC classes. Nearly all the requirement-to-OCR methods described in this report produce *single credit size* OCRs and so, for a vast majority of cases, the swaps are guaranteed to maintain the optimal match score. In the very rare case where a potential swap between classes of different sizes appears in step 11 of Algorithm 4, this swap can simply be skipped.

This section has shown how to form a different BMM solution by computing a different set of matched classes. But it may also be possible to form a different BMM solution by computing a different degree path, and this degree path may in turn yield a different set of matched classes. Ideally we would like to use the prioritization rule to explore optimal degree paths as well as matched-class-sets. But this type of exploration is beyond the scope of this report.

10.2.4 Flex Match Classes

When analyzing institutional data it may be useful to determine which classes are most flexible, i.e. which classes can most often be matched to different degree requirements. Although it may be tempting to define flexible classes to be the classes in the exchangeable class groups, the result can be misleading. Consider the following example.



In this example OCR1 and OCR2 can only be satisfied by one specific class each, but OCR3 can be satisfied by any class (like a free elective). Thus, the optimal match will always match (OCR1,c1) and (OCR2,c2), while OCR3 will be matched to any one of the remaining classes c3-c7. The exchangeable class group for OCR3 is (c1,c2,c3,...,c7), but neither c1 or c2 can actually be exchanged to match OCR3 without reducing the match score. Thus c1 and c2 are not truly "exchangeable" and should not be included in the flexible class group.

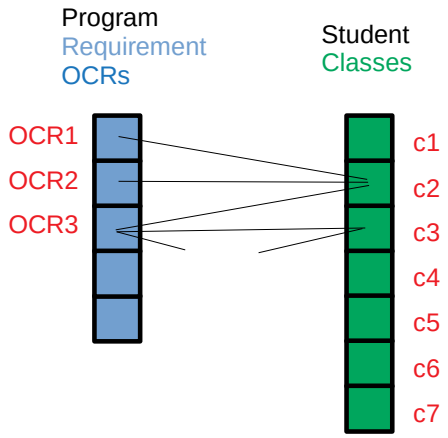
We now attempt to generalize the observations from the above example. We start with the following definitions.

- An OCR that is linked to only one class is called an *exclusive* OCR.
- A class that is linked to only one OCR is called an *exclusive* class.
- An OCR that is linked to at least one exclusive class is called an *independent* OCR.

The following two theorems can be used to identify classes that should be omitted from the flexible class group.

Theorem 1. *Consider a class c that is linked to both exclusive and non-exclusive* OCR*s. If the largest exclusive* OCR *weight is greater than the largest non-exclusive* OCR *weight, then all BMM solutions will match the class c to one of the exclusive* OCR*s (and the other exclusive* OCR*s will be unmatched).*

Proof. This theorem can be proved using proof-by-contradiction. We provide only a sketch of the proof. Assume that OCR1 and OCR2 are exclusive OCRs linked to the class c1, and that OCR3 is a non-exclusive with exchangeable class group (c1, c2, ...). An example is shown in the figure below. Assume the OCR1 and OCR2 weights are larger than the OCR3 weight. This establishes the conditions of Theorem 1. Now assume the BMM solution contains the match (OCR3,c1). This implies that OCR1 and OCR2 are unmatched. If we replace the (OCR3,c1) match with the (OCR1, c1) match then the weighted match score will increase (because the OCR1 weight is larger than the OCR3 weight), and so the assumed solution (with (OCR3,c1) match) cannot be optimal. The same is true of we replace the (OCR3,c1) match with the (OCR2, c1) match. □



Theorem 2. *Consider a class c that is only linked to exclusive and independent* OCR*s. All BMM solutions will match the class c to one of the exclusive* OCR*s (and the other exclusive* OCR*s will be unmatched).*

Proof. This proof is similar to the proof of Theorem 1 and is omitted here. □

Theorem 3. *Consider an exclusive class c that is only linked to an exclusive* OCR*. All BMM solutions will match this class c to this exclusive* OCR*.*

Proof. The proof of this theorem is trivial. □

Theorem 1 says that classes that are linked to exclusive OCRs with dominate weights are not actually exchangeable[14]. Theorem 2 makes a similar statement about classes that are only linked to exclusive and independent OCRs (as in the first example above). Furthermore, since these classes will always match one of the exclusive OCRs, the link between these classes and all other OCRs can be removed. These edges can be removed at any time without affecting the optimal solution, i.e. they can be removed before the BMM solution has been computed, after the BMM solution has been computed, or after the BMM solution has been modified by Algorithm 4. If they are removed before the BMM solution has been computed then the *maximum weighted bipartite matching* algorithm may be more efficient. If they are removed after the BMM solution has been computed then Algorithm 4 may be more efficient.

The theorems and observations above suggest the following steps for computing the flexible class group.

1. Start with the input graph and modify it as follows.
2. Identify all exclusive and independent OCRs.
3. Identify all classes that are linked to both exclusive and non-exclusive OCRs. For each of these classes, if the largest exclusive OCR weight is greater than the largest non-exclusive OCR weight then remove all edges to the non-exclusive OCRs.
4. Identify all remaining classes that are only linked to exclusive and independent OCRs, and remove all edges to the non-exclusive OCRs.

[14] In addition, it is not hard to prove that even when the OCR weights are all the same, at least some of the optimal solutions will contain the exclusive OCR-class match.

5. Now use the modified graph to form the exchangeable class groups.

6. Discard all exchangeable class groups that contain only one class.

7. Form the flexible class group by combining all the classes from the remaining exchangeable class groups.

11 Assessing the Accuracy of the Reasoner

The end-to-end process that collects student class lists and degree requirements from institutional databases, represents this information in a form that can be processed on a digital computer, and then applies one of the algorithms described earlier (e.g. the ARP algorithm or one of the approximation algorithms) is called the *reasoner*. This section describes methods for assessing the accuracy of the reasoner.

If the ARP algorithm is presented with a complete input (i.e. the class and requirement lists are complete) then it will provide a correct optimal solution whose accuracy is perfect[15]. On the other hand, if the input is incomplete and/or an approximation algorithm is used to compute an (approximate) best max-match solution then the accuracy may be less than perfect. The goal of this section is to quantify the accuracy of approximate solutions.

Section 11.1 describes methods for assessing the accuracy of approximation algorithms when the inputs are complete and the ARP algorithm is available to establish an optimal solution. In practice however it may not be possible to obtain complete inputs for all cases (see Section 11.2), and/or the ARP algorithm may not be available because of its excessive computational requirements. Thus, Section 11.2 describes methods for assessing the accuracy when inputs are incomplete and/or the ARP algorithm is not available. These methods are used to produce the results for UNM data in Section 11.3.

The following definitions (similar to Section 10.1) will be used in all three Sections 11.1-11.3.

- Let $R_1, R_2, ..., R_K$ be the non-path requirements determined in Step 1 of the ARP algorithm and let $R_0 = p_{core}$ be the core path formed in Step 2 of the ARP algorithm.

- Let h_{req_i} be the minimum number of credit hours required to satisfy requirement R_i, and $h_{total} = h_{req_0} + h_{req_1} + h_{req_2} + ... + h_{req_K}$ be the minimum number of credit hours required to satisfy the degree program.

- Consider a solution that may not be optimal because either the input is incomplete and/or the solution has been produced by an approximation algorithm. Let $\hat{p}_{m,i}$ be the list of OCRs from the approximate best max-match solution that count towards requirement R_i, and let $\hat{h}_{m,i}$ be the total number of credit hours in $\hat{p}_{m,i}$.

- Define the *estimated* progress towards requirement R_i as follows

$$\hat{\rho}_i = \frac{\min(\hat{h}_{m,i}, h_{req_i})}{h_{req_i}}$$

  and then define the *estimated* overall student progress

$$\hat{\rho} = \frac{\sum_{i=1}^{K} h_{req_i}\hat{\rho}_i}{h_{total}} = \frac{\sum_{i=1}^{K} \min(\hat{h}_{m,i}, h_{req_i})}{\sum_{i=1}^{K} h_{req_i}}$$

[15]This must be true because the ARP algorithm considers all possible solutions.

Note that all progress measures ρ_i, ρ, $\hat{\rho}_i$, $\hat{\rho}$ take values between 0 and 1, and are equal to 1 if and only if the requirement(s) is(are) completely satisfied.

11.1 Accuracy in the Ideal Case

The path elimination and iterative match algorithms in Section 9 produce an approximate best max-match solution. This section describes methods for quantifying the accuracy of these algorithms when the input is complete and the ARP algorithm is available to provide an optimal best max-match solution.

The ARP algorithm provides a correct optimal solution. When there is more than one optimal solution we assume that the ARP algorithm returns all optimal solutions, and we compute the measures below against all of these solutions and keep the best result.

Consider the solution provided by the ARP algorithm. Let $p_{m,i}$ be the list of OCRs from the best max-match solution that count towards requirement R_i, and let $h_{m,i}$ be the total number of credit hours in $p_{m,i}$. Now consider the following *error measures*.

1. **Credit Hour Error:** The *total credit hour error* is defined
$$e_{tcr} = \left(\sum_{i=0}^{K} h_{m,i}\right) - \left(\sum_{i=0}^{K} \hat{h}_{m,i}\right)$$
We say that the approximation algorithm has produced a correct result when $e_{tcr} = 0$. Note that this does not mean that $\hat{h}_{m,i} = h_{m,i}$ for all i. Indeed, a more sensitive error measure is the *individual credit hour error*
$$e_{cr} = \sum_{i=0}^{K} |h_{m,i} - \hat{h}_{m,i}|,$$
Note that if $e_{cr} = 0$ then $e_{tcr} = 0$, but the reverse is not true. Our primary interest is in the *total credit hour error* e_{tcr}, but e_{cr} can be used to help diagnose specific differences between the matches made by the ARP and approximation algorithms.

2. **Progress Error:** To assess the accuracy of approximation algorithms in computing student progress we use the *total progress error*
$$e_{tpr} = |\rho - \hat{\rho}|$$
Again we say that the approximation algorithm has produced a correct result when $e_{tpr} = 0$, but this does not mean that $\hat{p}_i = p_i$ for all i. Indeed, a more sensitive error measure is the *individual progress error*
$$e_{pr} = \sum_{i=0}^{K} |\rho_i - \hat{\rho}_i|,$$
which can be used to help diagnose specific differences between the matches made by the ARP and approximation algorithms. Note that if $e_{pr} = 0$ then $e_{tpr} = 0$, but the reverse is not true.

3. **Requirement Error:** To assess the ability of approximation algorithms to correctly match classes to individual requirements we define the *total requirement error*
$$e_{req} = \sum_{i=0}^{K} I(\hat{\rho}_i \neq \rho_i)$$

where $I(\cdot)$ is the identity function that is equal to 1 when its argument is true and 0 otherwise. This error is simply a count of the number of mismatched requirements.

Finally, to quantify the *accuracy* of the approximation algorithms we simply need to normalize the error values.

- **Credit Hour Accuracy:** The *total credit hour accuracy* is

$$a_{tcr} = \left(1 - \frac{e_{tcr}}{\max\left(\sum_{i=0}^{K} h_{m,i}, \sum_{i=0}^{K} \hat{h}_{m,i}\right)}\right) \times 100$$

  This value is equal to 100% when $e_{tcr} = 0$.

- **Progress Accuracy:** The *total progress accuracy* is

$$a_{tpr} = \left(1 - \frac{e_{tpr}}{\max\left(\rho, \hat{\rho}\right)}\right) \times 100$$

  This value is equal to 100% when $e_{tpr} = 0$.

- **Requirement Accuracy:** The *total requirement accuracy* is

$$a_{req} = \left(1 - \frac{e_{req}}{K}\right) \times 100$$

  This value is equal to 100% when $e_{req} = 0$.

11.2 Accuracy in Practice

There are numerous practical issues that may prevent a complete input to the reasoner. Examples of issues that arose during IDI's development for UNM students and degree programs are described in the list below.

- This report has assumed that the only way to satisfy a degree requirement is by successfully completing certain classes. But there are at least two exceptions to this rule.

  1. The English 101 (or 110?) requirement is automatically satisfied for incoming students whose ACT or SAT score is sufficiently high. However these students do not get the 3 credits towards their degree program that they would if they had successfully completed the course. Instead these students may be required to complete up to 3 additional credits of Free Electives. The actual number of additional free elective credits required, h_{free}, depends on the (minimum) total credits in the student's degree program, $h_{program}$, and is given by the following formula

$$h_{free} = \max\left(0, 120 - (h_{program} - 3)\right)$$

     This guarantees that all UNM students graduate with at least 120 credit hours. A simple way to handle this situation is to modify the degree requirements for the student so that the English requirement is removed and h_{free} credits of free electives are added.

2. Students can *test out* of some courses at UNM. That is, they can get credit for a course even though they do not actually take the course (in the normal way). A simple way to handle this situation is to map each student's list of *successful tests* to a list of *completed courses*, and then augment the student's class list with these *completed courses*. Then the analysis can proceed as before.

- UNM has been translating all of its lower division courses from a UNM-specific 3-digit numbering system to a state-wide 4-digit common course numbering system. This process has created a mixture of 3-digit and 4-digit course specifications in UNM databases, and it is not clear that the 3,4-digit equivalence has been correctly articulated in all the right places. For example, the official class list for a student that started at UNM before the conversion is likely to contain a combination of both 3-digit and 4-digit courses. But the degree requirements for this student, which are tied to a specific year, will only be specified in terms of either 3-digit or 4-digit courses. This means that some courses in the student's class list will not automatically match the degree requirement specification. The solution is to specify both the degree requirements and the student class list in terms of a *canonical course numbering system* before applying the algorithms in this report.

- The requirements for a specific degree program can change each year. This means that the IDI algorithms must keep track of the requirements for each degree program for each year dating back to the earliest year of interest. In addition, a student's classes must be matched against the requirement list for the year that the student first entered the degree program, and this may not be the first year that the student entered UNM. For example, many students switch degree programs during their time at UNM.

- The list of classes that count towards a particular requirement is sometimes incomplete. This seems to happen most often with electives that are tied to a specific degree program (e.g. depth electives). The problem seems to be that the list of courses extracted from the university catalog is incomplete. A complete list of courses can possibly be obtained from the Lobo Trax system.

- The data entry process, which enters degree requirements for every program at UNM, is subject to human error. These errors often go unnoticed until these requirements are used for some type of analysis (e.g. the analysis performed by the algorithms developed in this report). The process of locating and correcting these errors is tedious and labor intensive, and exasperated by the fact that an error discovered for a degree requirement in one year must often be corrected for the same degree requirement across all other years.

- There are numerous anomalies, i.e. unusual cases, that occur in the student and/or institutional data. For example, a recent student's class list included a very large number of courses that were completed more than 20 years ago.

- As of late November 2021 the reasoner did not yet fully accommodate the following cases.
  - *variable credit classes*
  - *exceptions:* These are special cases where the student is given credit for a requirement or class based on some type of completed work that is outside the normal scope. For example, a student may be given credit for a UNM course based on one or more courses that were completed at another institution but were not successfully transferred to UNM under the normal process.

– Transfer courses that count at UNM but are not articulated to a specific UNM course are not yet fully incorporated into the IDI algorithms.

Most of the issues above can be resolved with sufficient time and effort, but it is unlikely that all of them can be resolved with perfect accuracy. Thus, the reasoner used in practice will likely always have incomplete inputs for some students, and therefore provide only approximate solutions. Consequently the ARP algorithm cannot be used to establish the *ground truth* for an optimal solution, and therefore the accuracy measures in the previous section cannot be used to assess the accuracy of the reasoner.

In this section we describe accuracy assessment methods that do not rely on ARP *ground truth* values, and can therefore be used to assess any approximate solution. In particular, the next three sub-sections describe three different methods. Section 11.2.1 describes a method that uses only students that have graduated. Ground truth is easy to established in this case, i.e. credit hours should meet or exceed 120, student progress should be 100%, and all requirements should be satisfied. The accuracy values produced by this method do not apply directly to active students (that have not graduated) and are therefore not directly relevant to the accuracy of the *student progress* assessment. But the inaccuracies revealed by this method are likely to be the same (or very similar to the) inaccuracies that occur when the reasoner is applied to active students. In addition, the reasoner accuracy for graduated students is directly relevant to other types of analysis such as over-satisfied requirement analysis and/or excess credit analysis (i.e. the analysis of quantities that are derived from the decomposition in Section 10.2). Section 11.2.2 attempts to more directly assess the accuracy of the reasoner when it is applied to active students (that have not yet graduated). Ground truth is more difficult to establish in this case. But we can establish a surrogate ground truth by applying the ARP algorithm to the student's class list, and then retaining a subset of the matched classes. Any reasoner algorithm that is applied to this subset of classes should match all of these classes. This method does not necessarily address all aspects of the *student progress accuracy*, but does provide a useful assessment of an important component of this accuracy. Finally, instead of analyzing the accuracy of the overall reasoner, Section 11.2.3 describes a method that analyzes the accuracy of the individual reasoner components.

11.2.1 Accuracy for Students that have Graduated

This method applies the reasoner to a collection of students that have graduated and then, using the definitions of K (the number of requirements), h_{req_i}, h_{total}, and $\hat{h}_{m,i}$ provided in the first part of Section 11, performs the following computations.

- Let h_0 = number of *freed* credits. Freed credits occur when a requirement is automatically satisfied by a student activity that contributes no credits. For example the English 101 (or 110?) requirement is automatically satisfied for incoming students whose ACT or SAT score is sufficiently high. In this case the English requirement is satisfied but the students do not get 3 credits towards their degree program and so the number of freed credits is 3.
- Let $h^*_{total} = \max(120, h_{total} - h_0)$ = minimum number of UNM credits the student must complete.
- Then the *progress accuracy* is

$$a_{pr} = \left(\frac{\sum_{i=0}^{K} \min(\hat{h}_{m,i}, h_{req_i})}{h^*_{total}} \right) \times 100$$

- Let $\hat{h} = \sum_{i=0}^{K} \hat{h}_{m,i}$ = total number of credits produced by the reasoner.
- Then the *credit hour accuracy* is

$$a_{cr} = \left(\frac{\min(\hat{h}, h^*_{total})}{h^*_{total}} \right) \times 100$$

- Let $\hat{K}$ = number of satisfied (or over-satisfied) requirements produced by the reasoner.
- Then the *requirement accuracy* is

$$a_{req} = \left(\frac{\hat{K}}{K} \right) \times 100$$

11.2.2 Accuracy for Students that have not Graduated

This section describes methods that attempt to assess the accuracy of the reasoner when it is applied to active students (that have not yet graduated). One approach is to have a human examine the reasoner output in detail, and with the assistance of the official university audit system (e.g. Lobo Trax), determine the accuracy. But this approach is extremely labor intensive and therefore can only be applied to a small number of students. We now describe a less comprehensive method that can be automated and applied to a large number of students.

This method establishes a surrogate ground truth by applying the ARP algorithm to a student's class list, and then retaining a subset of the matched classes. Although the accuracy of the ARP match may be unknown, it is safe to assume that all matched classes can be counted towards the degree program, and therefore any reasoner algorithm that is applied to the matched classes alone should match all of these classes. The same is true for any subset of the matched classes. Furthermore, this truth should hold regardless of whether the initial student class list came from a student that has graduated or from a student that have not yet graduated. Note that even when the reasoner produces a match for all classes in the subset, the number of satisfied requirements will depend on how the match is made, and since the reasoner algorithms make no direct attempt to optimize the number of satisfied requirements there is little value in computing a *requirement accuracy* for this situation.

A summary of the steps performed by this method are shown below.

1. Choose a collection of students (both active and graduated) and apply the ARP algorithm to each student's class list. Then retain a subset of the matched classes for each student.
2. Let h^* = number of matched credits that are retained.
3. Let $\hat{h} = \sum_{i=0}^{K} \hat{h}_{m,i}$ = total number of credits produced by the reasoner when presented with the subset of matched classes.
4. Then the *credit hour accuracy* is

$$a_{cr} = \frac{\hat{h}}{h^*}$$

5. Let h_i = the number of credits from the subset of matched classes that count towards requirement R_i. Define the ground truth progress to be

$$\rho^* = \left(\frac{\sum_{i=0}^{K} \min(h_i, h_{req_i})}{h^*_{total}} \right)$$

and the estimated progress to be

$$\hat{\rho} = \left(\frac{\sum_{i=0}^{K} \min(h_{m,i}, h_{req_i})}{h^*_{total}} \right)$$

where $h_{m,i}$ is the number of credits counted towards requirement R_i by the reasoner when presented with the subset of matched classes.

6. Then the *progress accuracy* is

$$a_{pr} = \frac{\hat{\rho}}{\rho^*}$$

11.2.3 Accuracy of Reasoner Components

In this section we propose dividing the reasoner into the following three components and then analyzing the accuracy of each component separately.

1. student class list generation
2. degree path generation
3. computing the max match

Results from the Lobo Trax audit system can be used to help analyze the accuracy of the first two components as described below. Analyzing the accuracy of the third component amounts to validating our implementation of the *maximum weighted bipartite matching* algorithm and should be relatively straightforward.

The accuracy of the reasoner's student class list can be assessed as follows. The Lobo Trax system always contains at least one requirement that is verified by examining the complete list of the student's classes. Thus, the Lobo Trax output always includes a complete list of the student classes that can be compared against the student class list formed by the reasoner.

There are two parts to the analysis of the reasoner's degree path generation. The first is the synthesis of the individual requirement paths that are combined to form the degree paths, and this can be validated by carefully debugging the code that implements the methods in Sections 5 and 6. The second is the synthesis of the OCR *class lists*, i.e. the list of classes that count toward each OCR. This appears to be one of the main sources of reasoner inaccuracy. First note that every requirement in a Lobo Trax audit contains a complete list of requisite classes that can count towards the requirement's satisfaction. Furthermore, even though the Lobo Trax system uses a collection of shared class requirements, a subset of these requirements are often nearly identical to the separate class requirements employed by the reasoner. Thus, the requisite classes for the nearly identical Lobo Trax requirements can be used to validate the OCR class lists produced by the reasoner.

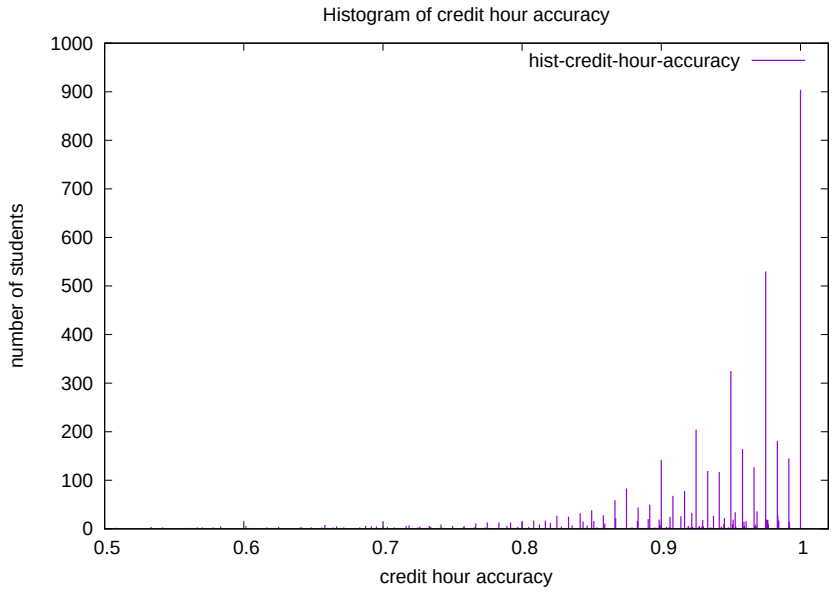
11.3 Accuracy Results for Current Reasoner with UNM Data

An *IDI reasoner* has been developed using the methods described in this report. This section reports preliminary accuracy results for this reasoner. These results were obtained by applying the current IDI reasoner[16] to a cohort of 4684 students that have all graduated under a 2016 degree program. These students may have started and finished at different times, but as of the Fall 2021

[16]This IDI reasoner is the one that was current at the end of November 2021.

semester they have all graduated under a degree program that was prescribed in 2016. The 2016 degree programs were the first to be converted to degree paths by the IDI reasoner. Conversion of degree programs for other years is currently under way. The 4684 students in this cohort graduated from at total of 91 different degree programs.

Since all of these students have graduated we restrict to the accuracy measures in Section 11.2.1. The results in the section were obtained using the iterative match algorithm. A histogram of the credit hour accuracy values for these 4684 students is shown below. A perfect accuracy of 1.0 is achieved for 904 (19%) of these students. An accuracy of .95 or better is achieved for 2711 (58%) students, and an accuracy of .9 or better is achieved for 3685 (79%) students. The average accuracy of the complete group of 4684 students is 92.7%.



A Variable Credit Hour Requirements (VCRs)

This appendix describes the *copy-and-replace* operation for a *variable credit hour* requirement of the form

$R_{vcr} =$ student must complete N credit hours from a specific category where $N_{lower} \leq N \leq N_{upper}$.

This type of requirement is represented by the three-tuple $R_{vcr} = (N_{lower}, N_{upper}, \mathbf{C})$ where $\mathbf{C} = (c_1, c_2, ..., c_n)$ is a list of classes that count towards the specific category. Note that if this requirement is isolated, i.e. if it is not linked with one (or more) other variable credit hour requirements (VCRs), then the upper bound is vacuous and the VCR requirement can be replaced by the simpler *credit hour* requirement

$$R_{ch} = \text{ student must complete (at least) } N_{lower} \text{ credit hours from a specific category.}$$

Now consider a group of non-isolated VCRs $(R_1, R_2, ...)$ that are linked through a parent requirement R_{parent} that specifies the total number of hours contributed by the group. For example suppose the two VCRs

$R_1 =$ student must complete N credit hours from category $\mathbf{C}_1$ where $N_{1,lower} \leq N \leq N_{1,upper}$.

$R_2 =$ student must complete N credit hours from category $\mathbf{C}_2$ where $N_{2,lower} \leq N \leq N_{2,upper}$.

are linked by the parent requirement

$$R_{parent} = \text{ student must satisfy } R_1 \text{ and } R_2 \text{ with a total of (at least) } N_{parent} \text{ credit hours}$$

In this case N_{parent} must satisfy $N_{parent} \geq (N_{1,lower} + N_{2,lower})$, otherwise the two VCRs can be simplified in the same way as the isolated VCR above. Let

$$N_+ = N_{parent} - (N_{1,lower} + N_{2,lower})$$

When $N_+ > 0$ the requirement R_{parent} can be satisfied by taking the extra N_+ credit hours from either category $\mathbf{C}_1$, or category $\mathbf{C}_2$, or both. At the same time however, there is an upper limit on the total number of credit hours that can come from each category.

We will show how to convert a parent requirement that has multiple child VCR requirements into a collection of paths (i.e. an OPR) by extending the `EnumerateCCHC` algorithm in Section 6.3. We start with some definitions.

- Let R_{parent} be a parent requirement of the form

  $$R_{parent} = \begin{array}{l}\text{student must satisfy requirements } R_1, R_2, ..., R_M \\ \text{with a total of (at least) } N \text{ credit hours}\end{array}$$

  Let the children requirements $R_1, R_2, ..., R_M$ be VCRs with categories $\mathbf{C}_i$, $i = 1, 2, ..., M$ and with lower and upper credit hour requirements $(N_{i,lower}, N_{i,upper})$, $i = 1, 2, ..., M$.

- Let $A_i = \{(h_i, n_{h_i})\}$ be the set of (number of credit hours, number of courses) tuples for category $\mathbf{C}_i$.

- Let $\mathcal{A} = (A_1, A_2, ..., A_M)$ be the complete collection of (number of credit hours, number of courses) from the M categories.

We seek to enumerate all course credit hour combinations from the M categories that yield a total of at least N credit hours and satisfy the credit hour bounds for each category. The `EnumerateCCHC-VCR` algorithm below accomplishes this by extending the `EnumerateCCHC` algorithm in Section 6.3. The variables N_i are used to keep track of the total number of credit hours from categories $\mathbf{C}_i$ as the algorithm unfolds. In addition $\mathcal{N} = (N_1, N_2, ..., N_M)$ is the complete collection of N_i variables. The output of this algorithm is L, a collection of credit hour requirement lists where each list H is a list of two-tuples (h, i) that represent a requirement for a course with h credit hours from category $\mathbf{C}_i$.

{Main Routine}
Inputs: $\mathcal{A}$, $\mathcal{N}$, $(N_{i,lower}, N_{i,upper})$ for $i = 1, 2, ..., M$
L = empty {*the initial list of credit hour combinations is empty*}
$H = []$ {*the initial credit hour combination is blank*}
$\mathcal{N} = (0, 0, ..., 0)$ {*number of credit hours from each category is initially 0*}
`EnumerateCCHC-VCR`($\mathcal{A}$, $\mathcal{N}$, N, H)

{Recursive Subroutine}
`EnumerateCCHC-VCR`(A, $\mathcal{N}$, N, H)
$N' \leftarrow$ sum of credit hour values in the current list H
if $(N' \geq N)$ **then**
  $h_{min} \leftarrow$ smallest credit hour value in current list H
  if $((N' - h_{min}) < N)$ **then**
    if $((N_{i,lower} \leq N_i \leq N_{i,upper})$ for all $i = 1, 2, ..., M)$ **then**
      add H to the output list L
    end if
  end if
  return
end if

for (each $i = 1, 2, ..., M$) **do**
  for (each $(h_i, n_{h_i}) \in A_i$, in order from smallest h_i to largest h_i) **do**
    if ($(n_{h_i} > 0)$ and $(h_i >$ most recent addition to H from $A_i)$) **then**
      $H' \leftarrow$ add (h_i, i) to the credit hour combination in H
      $A'_i \leftarrow (A_i \setminus (h_i, n_{h_i})) \cup (h_i, n_{h_i} - 1)$ {*adjust the count so that* $(h_i, n_{h_i}) \leftarrow (h_i, n_{h_i} - 1)$}
      $N'_i \leftarrow N_i + h_i$
      `EnumerateCCHC-VCR`($\mathcal{A}'$, $\mathcal{N}'$, N, H')
    end if
  end for
end for
return

This routine outputs a list of credit hour combinations $L = (H_1, H_2, ..., H_Q)$. Each member of H_i takes the form (h, i) where h is the number of credit hours and $i \in \{1, 2, ..., M\}$ indicates the category. These credit hour combinations are used to create the paths $p_1, p_2, ..., p_Q$. For example

the credit hour combination

$$H = ((1,2),(1,1),(1,3),(2,1),(2,3),(3,1))$$

is used to create the path

$$p = ([\bar{C}_1]_2, [\bar{C}_1]_1, [\bar{C}_1]_3, [\bar{C}_2]_1, [\bar{C}_2]_3, [\bar{C}_3)]_1.$$

where $[\bar{C}_h]_i$ is the OCR containing h-credit hour courses from category i. Once the paths have been created the credit hour requirement R_{ch} is replaced by

$$R_{ch} = p_1 \vee p_2 \vee ... \vee p_Q$$

and the *copy-and-replace* operation for disjunctive LC requirements in Procedure 1 can be applied.

B One Class Satisfies Two (or more) Requirements

Apparently there are rare times when a degree program will contain a group of 2 (or more) OCRs that can be simultaneously satisfied by completing a single course. While it is possible to satisfy these OCRs by completing 2 (or more) different classes, the student can get a 2-for-1 (or many-to-one) return by taking a single course. In this section we show how to treat this situation when performing student assessments.

Let c^* be the course that satisfies multiple OCRs. First consider the case where one of these OCRs is an *individual course requirement* i.e. $C = \{c^*\}$. In this case the other requirements are *unnecessary*, since they are automatically satisfied once C is satisfied. So in this case we simply discard the other requirements and retain only the requirement $C = \{c^*\}$, and then proceed as before.

Next we consider the case where c^* is just one of multiple courses in each of the OCRs $C_1, C_2, ..., C_Q$. In this case we create a new OCR

$$C^* = \{c^*\}$$

and we remove c^* from the OCRs $C_1, C_2, ..., C_Q$ to give

$$C_i' \leftarrow C_i \setminus \{c^*\}, \qquad i = 1, 2, ..., Q$$

Then we replace the path

$$p = C_1 \wedge C_2 \wedge ... \wedge C_Q$$

with the requirement

$$R = (C_1' \wedge C_2' \wedge ... \wedge C_Q') \vee C^*$$

and then proceed as before.