

# Student Flow Simulation with Applications in Curricular Analytics

by

**Michael Hickman**

B.S., University of New Mexico 2014

THESIS

Submitted in Partial Fulfillment of the  
Requirements for the Degree of

Master of Science  
Computer Engineering

The University of New Mexico

Albuquerque, New Mexico

September, 2017

©2017, Michael Hickman

# Acknowledgments

I would like to thank ...

# Student Flow Simulation with Applications in Curricular Analytics

by

**Michael Hickman**

B.S., University of New Mexico 2014

M.S., Computer Engineering, University of New Mexico, 2017

## **Abstract**

Abstract goes here

# Contents

<b>List of Figures</b>	<b>viii</b>
<b>Glossary</b>	<b>ix</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Previous Work</b>	<b>4</b>
<b>3 Simulating Students in a Curriculum</b>	<b>5</b>
3.1 Discrete Event Simulations . . . . .	5
3.2 Defining A Curricula . . . . .	6
3.3 Assumptions . . . . .	8
3.4 Student Flow Simulation Model Logic . . . . .	10
3.4.1 Control Flow . . . . .	11
<b>4 Curricula Simulation Framework Design &amp; Implementation</b>	<b>13</b>
4.1 Julia . . . . .	13

## *Contents*

4.2	Architecture and Implementation Details . . . . .	14
4.2.1	Framework Data Types . . . . .	15
4.2.2	Performance Modules . . . . .	24
4.2.3	Simulation Method . . . . .	27
<b>5</b>	<b>Application in Curricular Analytics</b>	<b>29</b>
5.1	Curricular Complexity . . . . .	29
5.2	Correlating Complexity with Completion Rates . . . . .	31
5.3	Sensitivity Analysis of Instructional Complexity . . . . .	34
5.4	Probit Performance Model . . . . .	37
<b>6</b>	<b>Discussion</b>	<b>38</b>
<b>7</b>	<b>Future Work</b>	<b>39</b>
	<b>References</b>	<b>40</b>

# List of Figures

1.1	Comparison of two Electrical Engineering Curricula . . . . .	3
3.1	Visual representation of the electrical engineering curriculum at UNM.	7
4.1	A component diagram of the CAS framework. . . . .	15
5.1	Delay factors are given for courses A, B, C, and D. . . . .	30
5.2	Blocking Factors are given for courses A, B, C, and D. . . . .	31
5.3	Basic Curricula . . . . .	32
5.4	Simulation results of thirty five real-world curricula . . . . .	34
5.5	Graph showing how the completion rate of the Computer Engineering curriculum at UNM changes with respect to course pass-rates. . . .	35

# Glossary

Discrete-Event Simulation JSON



# Chapter 1

## Introduction

The goal of every higher-education institution is to successfully educate students with skills and knowledge needed to enter, and be successful in the workforce. This is done by offering courses with desired learning outcomes required for success in a given field of work. These courses are then structured into a curriculum in which knowledge can be acquired and built upon. Moving students through these curricula is means in which universities and institutions achieve the goal of producing a student body well equipped for success in their chosen careers. Even more, universities are not merely just interested in students completing a program, but doing so in a timely manner. Most students wish to complete a degree as quick as possible and part of their success is determined by efficiency of their time at a university. Likewise, universities are incentivized to produce high four, five, and six year graduation rates.

It is no surprise then that much time, money, and research are spent in attempts to improve timely student success and the efficiency of their offered programs. Examples include tutoring programs, better advisement, intervention programs, financial support, improving instruction quality, experimenting with classroom structures (hybrid courses, flipped classrooms etc.) and many more [5, 6, 2, 3, 4]. While these

## *Chapter 1. Introduction*

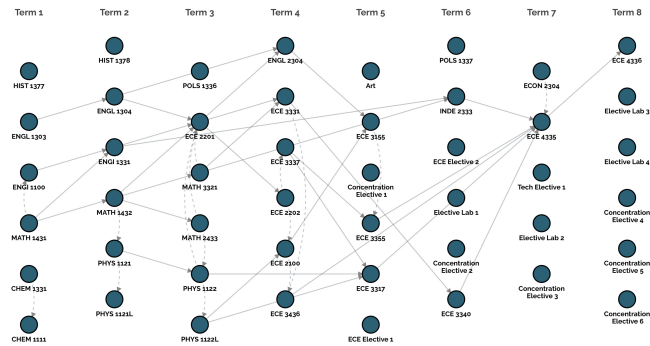
are all worthy pursuits that address many aspects of a student’s academic career, there is one area that many universities often overlook when it comes to improving a student’s journey through a curriculum: the structure of the curriculum itself.

Every university’s curricula are carefully designed and structured in such a way that students can achieve the learning outcomes defined for a given field of study. In most cases these learning outcomes are shared among most universities. The knowledge provided by a Mathematics at one university will most likely be the same as that at any other university. However, despite these shared learning outcomes every university’s curricula is quite different. This is evident when viewing a curriculum as data - or more specifically, a graph, and by doing so the way in which it’s structure can influence the efficiency of a student moving through it becomes apparent. For example see figure 1.1.

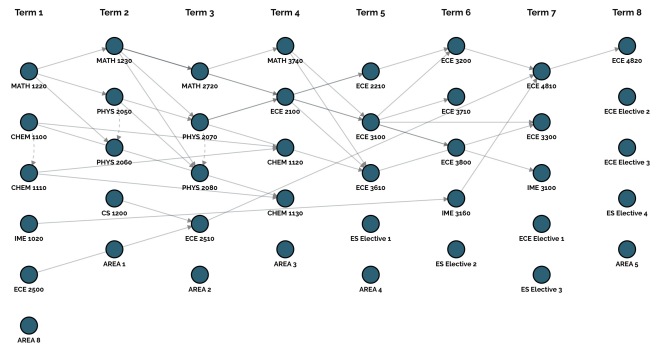
In this figure you can see a comparison of two Electrical Engineering curricula from two different institutions. Figure 1.1a shows University of Houston’s while 1.1b shows Michigan University’s. Both curricula are four year programs within four credit hours of one another and even without knowing the specific courses in each it obvious that they are both structurally different from one another. This begs the question is one, based merely on structure alone, more conducive to timely graduation than the other?

This thesis describes a software framework, the Curricular Analysis and Simulation (CAS) framework that allows for formal analytics over curricula to be done to help answer questions like the one previously posed. The framework described is an open-source Julia package that allows curricula to be represented in a programming, provides metrics for quantifying the complexity of a curriculum, both instructional and structural and, as it’s name implies, simulates of students moving through curricula via discrete-event simulations. Though there are many statistical and machine-learning models for predicting the rate at which students might grad-

## Chapter 1. Introduction



(a) University of Houston



(b) Michigan University

Figure 1.1: Comparison of two Electrical Engineering Curricula

uate, they fail to meaningfully represent the process from enrollment to graduation. This is where simulations can be a powerful tool for understanding not just how fast, but how students move through curricula and how curriculum characteristics play a role. This, rather than accurately predicting completion rates is the main focus of the CAS framework.

## Chapter 2

### Previous Work

# Chapter 3

## Simulating Students in a Curriculum

### 3.1 Discrete Event Simulations

Students moving through a curriculum can be viewed as a fixed, chronological sequence of events. Students will enter the university, enroll in classes, complete or withdraw from classes, then graduate if all requirements have been met. This process repeats every semester. Discrete-event simulation models a system by emulating a sequence of events, where every event occurs independently of the others at a particular instance in time and are characterized by having the following components:

- **Starting and Ending States** - The simulation will begin in a given state and will continue until it reaches another state that represents some pre-defined ending-condition or a point in time.
- **Clock** - The simulation must keep track of the time that has elapsed after it begins in any time unit relevant to the domain of the simulation. The clock does not run continuously as events occur instantaneously, but it does jump to certain times as events occur.

### *Chapter 3. Simulating Students in a Curriculum*

- **List of Events** - The system maintains a list of events that can occur during the simulation. These events are placed in a queue and are usually executed based on simulation time rather than the time they were queued.
- **Statistics** - As the ending state is not the only result of interest when running simulations, the system keeps track of various statistics based on the events that have occurred.

Using these components, a simulation is constructed using the basic following logic:

- 1: Initialize the system
- 2: Initialize the system clock
- 3: Schedule the initial event
- 4: **while** ending condition is false **do**
- 5:   Increment clock
- 6:   Perform next event
- 7:   Update system statistics
- 8: **end while**
- 9: Output simulation results

Given these characteristics and basic logic flow, this type of simulation lends itself well to the process of students flowing through a curriculum within an institution, much more so than continuous-time simulations and other analytical models; therefore, the student-flow simulation is modeled as a discrete-event simulation.

## **3.2 Defining A Curricula**

A curriculum is the backbone of the simulation. As the goal of the simulation is to model students moving through a curriculum, the events of the simulation depend

### Chapter 3. Simulating Students in a Curriculum

on the curriculum that is being simulated; therefore it is important to understand how they are defined. In order for a student to obtain a degree within a given academic program, a set of requirements must be met. The requirements, in most cases, are simply courses that must be passed. However a curriculum is more than a set of courses, but rather an arrangement of courses with constraints as to when they can be taken. This arrangement begins by placing courses within terms where, ideally, student would take and pass the entire set of courses in one semester. Next, relationships are created between courses in the form of prerequisite and co-requisite relationships. This structure of courses grouped in terms with defined relationships make up a curriculum. A visual, graph representation of a curriculum can be seen in Figure 3.1.

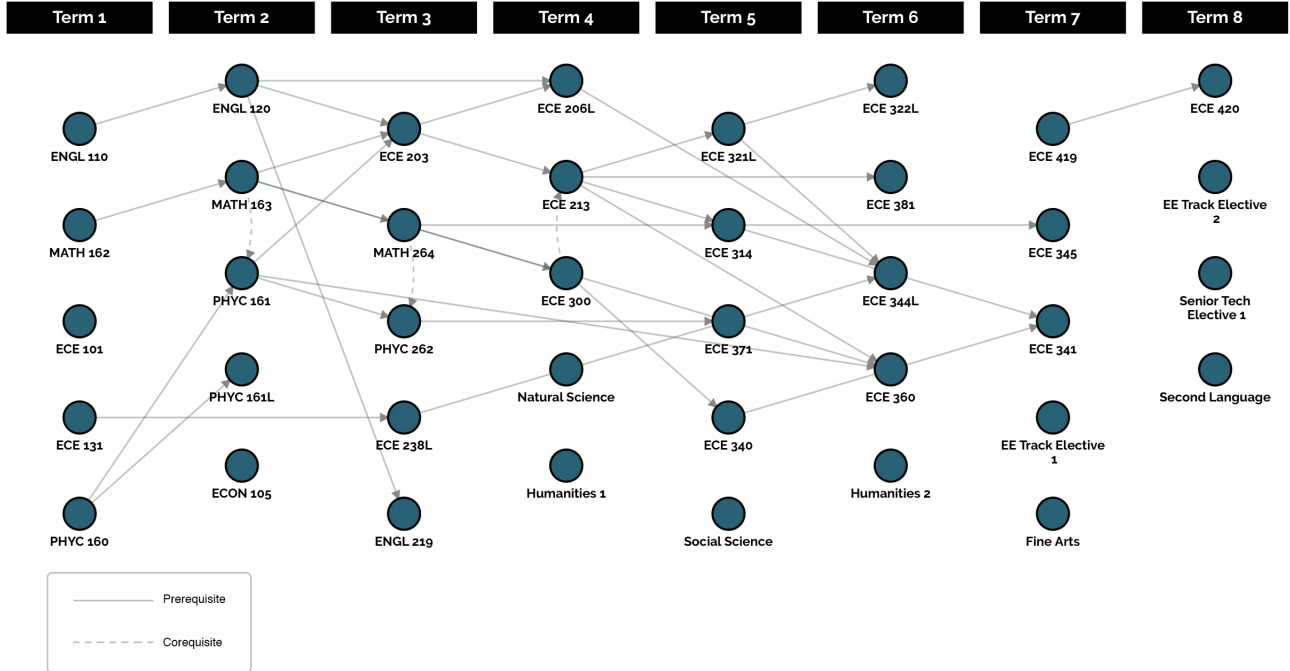


Figure 3.1: Visual representation of the electrical engineering curriculum at UNM.

### **3.3 Assumptions**

A student's academic career within an institution can potentially be a very difficult thing to model. This is primarily because student behavior can be erratic and unpredictable even given a structured curriculum. Students might not register for the courses they should, take courses that don't count, drop classes, take a semester off, get an override to take a course that they normally wouldn't have been able to, and various other actions that would otherwise not be considered 'typical' behavior. There are a vast amount of factors that can influence student behavior, many of which are not academic and would therefore be very difficult to account for. Furthermore universities can impose various policies that would also have an effect on how students register for classes and affect other academic decisions.

Trying to realistically simulate all possible actions that a student can take would be impossible, therefore several assumptions and decisions are made to simplify student behavior down to the basics, making meaningful simulation feasible without needless complexity. The most influential decisions made, was to design the student flow simulation with the primary goal of performing analytics over curricula and observing the effects their properties have on students moving through it. This is in contrast to developing a system to provide realistic simulations of student behavior with the goal of predicting graduation rates and other statistics. The simulation is potentially capable of this; however it was not the goal. Thus the other decisions made were done so to put emphasis on how curricula influence outcomes relative to one another instead of how student behavior does so. Below is a list of the other assumptions and decisions:

- All students at the beginning of the simulation are treated as first-time, full-time students. Although in many cases actual freshman might have some college credit, either through AP-tests, taking college-classes while in high-



### *Chapter 3. Simulating Students in a Curriculum*

school, or transfers, students in the simulation begin with a clean slate. They are also treated as full-time students with no notion of 'part-time' built in.

- Given that all students are full-time, each semester all students will register for as many credits as they can. In reality, most students do not hit their institutions max credit-hour limit, but the simulation will register students in as many courses as they are allowed - which is set by the user.
- The simulation only deals with a single class of admitted students. There is no continuous influx of freshman or transfer students each semester within the simulation.
- Students who stop-out are dis-enrolled permanently. They do not re-enroll.
- Students will only register for courses within the specified curriculum. The simulation is only aware of the courses specified in the curriculum, so while it is common for students to register for courses that might not count towards their major, this behavior is not part of the simulation.
- The order of courses in which students enroll in depends on their ordering within the curriculum. Students will enroll in the earliest listed course within the curriculum and will roll in each course they can as until they hit the credit-hour limit. This process is described in greater detail later.
- The minimum passing grade is universal among all courses. This grade can be set by the user, and defaults to a C-.
- All course enrollment requirements are strictly enforced. While it is common for students to obtain permission and subsequent overrides to enroll in a class they have not met the requirements for, the system does not allow for this.
- All courses within a curriculum are offered every semester.

### Chapter 3. Simulating Students in a Curriculum

- Courses have no notion of capacity. Any number of students can be enrolled in any course in a given term.

These decisions might seem very restrictive, and in many ways that is intentional. The idea wasn't to simplify the simulation for the sake of simplicity, but rather to put more emphasis on the curriculum and its structural properties. By standardizing the way students behave, it is possible to compare curricula and how changes to their structure or difficulty effect outcomes which provides insight into how they might be improved.

## 3.4 Student Flow Simulation Model Logic

The student flow simulation uses the components and basic logic of DES previously described and adapts them to fit within the domain of students flowing through a curriculum. The components are defined as follows:

*Clock.* The unit of time in which the simulation uses is a semester. All events occur within one semester, therefore the clock will on increment time by one semester.

*Starting State.* The simulation begins with a set of students which represent first-time full-time students just enrolled in a university. The clock is set to the first semester.

*Events.* The primary events in the simulation emulate basic student behavior during their academic career:

- Students enroll in the university.
- Student enrolls in the courses that they are able to take.

### *Chapter 3. Simulating Students in a Curriculum*

- Student takes the classes they enrolled in and are assigned a grade, either passing or failing.
- Student stopouts.
- Student graduates.
- Student remains enrolled.

*Ending Condition.* The simulation ends when there are no students enrolled. Therefore all students have either graduated or stopped out. The simulation could end before this however as it can also be run for a set amount of semesters.

*Statistics.*

#### **3.4.1 Control Flow**

The simulation begins at its starting state - a set of students with no previous academic experience and a clock set to semester one. Each student is initiated with defined attributes. All enrolled students then begin registering for classes using a systematic approach governed by the way the curriculum is defined. The order of the courses in the curricula governs the order in which students register. Students will always register for the earliest courses listed within the curricula and continue registering for courses until there are no more courses that they can take or they cannot register for more courses due to the given term credit hour restriction. For example, if the first term within a curriculum consists of courses A, B, and C, the student will register for those courses in that order. If the student fails course B, then the following term, this will be the first course the student enrolls in. In order for a student to register for a course, they must meet the course's enrollment requirements. These requirements can be prerequisites, co-requisites, a term restriction meaning that the course cannot be taken until the clock reaches a specified term,

### *Chapter 3. Simulating Students in a Curriculum*

and that the student has not already taken and passed the course. In addition to these requirements, a student can only register for so many courses in a given term based on a given maximum credit hour limit which is set by the user. If a student can register for a course then they are added to that course's list of enrolled students and their term credit hour count is incremented by that courses credit hour value.

Once all students have completed the registration process, every student is assigned a grade for the courses they are enrolled in. The method for doing so depends on the user and will be covered later. The simulation uses the following grades: A+, A, A-, B+, B, B-, C+, C, C-, D+, D, D-, F, and W. The system also assigns numeric values to each grade which are 4.33, 4.0, 3.77, 3.33, 3.0, 2.77, 2.33, 2.0, 1.77, 1.33, 1.0, 0.77, 0 and 0, respectively. These values are used for GPA computations, point calculations, and can possibly be used to determine grades in other courses. The passing grade can be set by the user, but defaults to a C (2.33). If a student obtains this grade then their completion of the course is recorded.

Once all students have received their grades for all courses, the students GPA's are computed and then the simulation will check to see if each student has completed all requirements in the curriculum. If they have, then they are removed from the pool of enrolled students and added to the list of graduated students. The system will then simulate students stopping out. Again, the user can determine the method used for doing this which will be explained later. If the student is chosen as a stopout, then they are removed from the pool of enrolled students and added to a list of stopped out students. The clock is then incremented by one semester and the registration process begins for the next semester. These events are then repeated until there are no students enrolled, or the simulation clock reaches a time set by the user.

## Chapter 4

# Curricula Simulation Framework Design & Implementation

### 4.1 Julia

The CAS framework is implemented in the open-source Julia programming language and made available as an open-source Julia package. Julia is described as a high-level, high-performance dynamic programming language for technical computing that "has the performance of a statically compiled language while providing interactive dynamic behavior and productivity like Python, LISP or Ruby" [1]. It is built upon an LLVM-based just-in-time (JIT) compiler which allows it to reach performance close to that of C and, in many cases, speeds faster than that of R or MATLAB [1]. Julia's defining feature is multiple dispatch but some other notable features are module support, a type system, parallelism and a built-in package manager. It also has a thriving community of developers contributing high-quality open source libraries spanning a range of applications such as machine-learning, statistical analysis, graph analysis, plotting, and data-handling.

Each of these features contributed to choosing Julia as the language of implementation. Julia’s type system, ease of use and dynamic nature made it simple to implement the framework’s various components, methods logic. Working with data formats such as JSON and CSV were especially easier than a computation language such as MATLAB. However this ease of development did not come at the expense of performance as Julia is incredibly fast and its support for parallelism allowed running many simulations much faster as they could be carried out in parallel. Topping it all off is its built in package manager with a plethora of packages that will enable users extend the framework and simulation capabilities with powerful machine learning and statistical models.

## **4.2 Architecture and Implementation Details**

The framework is comprised of three main components: (1) custom data types, (2) the simulation method, and (3) course performance prediction modules. The first component, a set of custom data types, represent all the necessary components (curricula, students, etc.) over which analysis and simulations can be performed. The second component of the framework is a method that carries out a simulation using a defined curriculum and a set of students taking into account various simulation parameters that can be set by the user. The third and final component of the framework are modules which are passed into the simulation method that determine how the virtual students ‘perform’ in each course. The framework comes with one out of the box, but users can develop their own to conduct simulations that suite their needs. A system diagram can be seen in figure 4.1. The next several sections describe each component in greater detail.

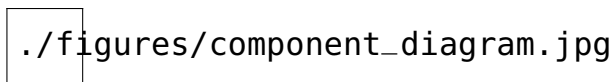


Figure 4.1: A component diagram of the CAS framework.

### 4.2.1 Framework Data Types

Custom Julia types serve as the foundation of the framework allowing curricula and related entities to be defined as mutable objects allowing analysis and simulation to be performed. These data types are: curricula, terms, courses, students, and simulations. Due to Julia not being an OO language, these entities are not true objects, but are very similar to structures in C in that they do not have associated methods, but do have associated attributes. Each type's attributes describe the characteristics of the entity it represents as well as provides a place to record statistics regarding the entity after a simulation is performed.

#### Course Type

At the heart of every curriculum are courses and every course in a curriculum is represented by its own object. These objects have the following attributes:

**name** A string that is used to identify the course. This can be anything, but the best options would be a course code, such as 'MATH 162' or the course's title such as "Calculus I".

**id** A unique integer identification number for the course.

**credits** The number of credit hours the course is worth.

**delay** The course's delay factor.

**blocking** The course's blocking factor.

#### *Chapter 4. Curricula Simulation Framework Design & Implementation*

**cruciality** The course's cruciality.

**prereqs** An array of courses that are considered prerequisites.

**coreqs** An array of courses that are considered co-requisites.

**postreqs** An array of courses that the courses is a prerequisite or co-requisite to.

**termReq** A courses minimum term requirement. For example, if this value is 2, then the course could not be taken until at least the second term.

**students** An array of students that are enrolled in the courses in a given term. The simulation populates this array each term, and then at the end of the term, empties it.

**passrate** The course's actual, real-world pass-rate. This is a percentage of students that pass the course represented as a floating point number between 0 and 1.0, where 1.0 is a 100% pass-rate.

**failures** The number of simulated failures. Every time a student within the simulation fails the course, this value is incremented.

**grades** An array of all simulated grades made by students enrolled in the course.

**enrolled** The total number of students enrolled in the course over the course of the simulation.

**termenrollment** An array of the number of students enrolled in the course each term. For example an array of [20, 30, 25] indicates that 20 students were enrolled in term one, 30 students in term two and 25 students in term three.

**termpassed** An array of the number of students who passed the course each term.

**model** A julia dictionary, which is an associative array, that is used to store any kind of data pertaining to the prediction model the user selects for making



## Chapter 4. Curricula Simulation Framework Design & Implementation

grade predictions for students enrolled in the course. A dictionary was chosen because it is extremely flexible in terms of the values it can store and therefore suitable when the types of stored information is unknown prior to runtime.

Like all types in Julia, a course object is instantiated via a constructor method that has the same name as the type. Unlike other languages, Julia allows a type to have multiple constructors and in this case, the course type has four. Each constructor has the same name, *Course()*, but the number and types are arguments differ. All constructors require a name, the number of credits the course is worth, an array of prerequisites and an array of co requisites with one of the constructors accepting only these arguments. The others constructors allow more information to be provided, specifically a pass-rate, a termReq, or both. Examples of the constructors can be seen below.

```
1 # Math 162 course with no pre or corequisite.
2 math162 = Course("Calculus I", 4, [], [])
3
4 # Physics course with a passrate of 90%, and math 162 as a prerequisite.
5 phys162 = Course("Physics II", 3, 0.9, [math162], [])
6
7 # Physics lab with Pysicis II as a corequisite
8 phys162l = Course("Physics II Lab", 1, [], [phys162])
```

### Term Type

The term type essentially represents a collection of courses that, ideally, would be taken together in the same semester. It has the following attributes:

**courses** An array of courses that make up the term. The order of the courses matters, as those that are at the beginning of the list will be registered for first.

**credits** The sum of the credit hour values of courses within the term.

## Chapter 4. Curricula Simulation Framework Design & Implementation

**totalEnrolled** The total number of students enrolled in the courses that belong to the term.

**failures** The total number of failing grades made in the courses that belong to the term.

The term type only has a single constructor that accepts an array of courses:

```
1  # Math 162 course with no pre or corequisite.
2  math162 = Course("Calculus I", 4, [], [])
3
4  # Physics course with a passrate of 90%, and math 162 as a prerequisite.
5  phys162 = Course("Physics II", 3, 0.9, [math162], [])
6
7  # Physics lab with Pysicis II as a corequisite
8  phys162l = Course("Physics II Lab", 1, [], [phys162])
9
10 term1 = Term([math162])
11 term2 = Term([phys162, phys162l])
```

### Curriculum Type

The curriculum type brings the term and course types together to define a complete academic program that contains a structured set of courses that students must progress through to obtain a degree. This is the type that the framework uses to perform a simulation over a curriculum. It has the following attributes:

**name** A string identifier for the curriculum.

**terms** An array of terms that make up the curriculum. The order should reflect the actual order of the terms.

**courses** An array of all the courses within the curriculum. This array is created from the array of terms and serves as a more convenient way to access the curriculum's courses rather than having to access them through the terms array.

**numCourses** The number of courses in the curriculum.

## Chapter 4. Curricula Simulation Framework Design & Implementation

**complexity** The curriculum's complexity.

**delay** The sum of the courses' delay factors.

**blocking** The sum of the courses' blocking factors.

**passrate** The average of the curriculum's courses' real-world pass-rates. Can be used as a naive measure of difficulty.

**stoupoutModel** A dictionary that stores the model that predicts student stopouts.

The curriculum has two constructors. The first takes a string that represents its name and an array of terms as arguments. The issue with this constructor, although it is completely usable, is that it requires the term objects, and therefore course objects to already be instantiated. This can be a tedious and time consuming and is not a very good way to store curricula in a extensible format. To address this problem, a JSON file format was developed as a straightforward way to define a curriculum that is easier to create, more portable, as well as easier for a computer to generate or read.

At the root of this definition is an object with two keys: *terms* and *courses*. The *terms* key stores the number of terms in the curriculum. The *course* key stores an array of objects that represent a course. These objects have the following keys: *name*, *credits*, *passrate*, *term* which is the term the course belongs to, *prerequisites* which is an array of strings corresponding to the names of the course's prerequisites, and *co-requisites* which is an array of strings corresponding to the names of course's co-requisites. An example can be seen below:

```
1 {  
2   "terms": 2,  
3   "courses": [  
4     {  
5       "name": "Calculus I",  
6       "credits": 4,  
7       "passrate": 0.7,  
8       "term": 1,
```

## Chapter 4. Curricula Simulation Framework Design & Implementation

```
9      "corequisites": [],
10     "prerequisites": []
11   },
12   {
13     "name": "Physics II",
14     "credits": 3,
15     "passrate": 0.8,
16     "term": 2,
17     "corequisites": [],
18     "prerequisites": ["Calculus I"]
19   },
20   {
21     "name": "Physics II Lab",
22     "credits": 1,
23     "passrate": 0.9,
24     "term": 2,
25     "corequisites": ["Physics II"],
26     "prerequisites": []
27   }
28 ]
29 }
```

The second constructor method accepts a name and another string that is expected to be a path to one of these JSON files. The method will parse the file and create course and term objects automatically. This makes it much more convenient to create a curriculum object along with objects for its associated courses and terms. Below are examples of these constructors:

```
1  # Curriculum from term objects
2  myCurriculum = Curriculum("My Curriculum", [term1 term2])
3
4  # Curriculum from JSON file
5  myIdenticalCurriculum = Curriculum("My myIdentical Curriculum", "./path/to/curriculum.
    json")
```

### Student Type

Every student in the simulation is represented by its own student object. Therefore, the student type was designed to contain all the information needed to track a student's progress as well as represent the characteristics that define the student and aid in grade predictions. It was designed to be flexible allowing the user to determine what characteristics are needed rather than have a set of hard-coded attributes. The attributes for the student type are:

## Chapter 4. Curricula Simulation Framework Design & Implementation

- id** A unique integer identifier for the student. Before a simulation is run, these are set to ensure all student ids are unique integer values starting at one.
- total\_credits** The total number of credit hours a student has earned during the simulation.
- total\_points** The total number of points a student has earned during the simulation. The number of points a student earns for a course is the product of the number of credit hours for the courses and the received grade's value. For example if a student earns a B+ in a three credit hour course, then they will receive  $(3.3 \times 3)$  or 9.9 points.
- gpa** The student's grade point average, which is computed at the end of every term. It is derived by dividing the student's *total\_points* by their *total\_credits*.
- termcredits** The number of credit hours the student registers for in a single term. At the end of every term it is reset to zero.
- stopout** A boolean value that indicates whether a student has stopped out and is no longer enrolled.
- stopsem** The term in which a student stops out.
- graduated** A boolean value that indicates whether a student has graduated.
- gradsem** The term in which the student graduates.
- performance** A dictionary that keeps track of how students perform in the classes they enroll in. The key value pair consists of the course name and letter grade made, respectively.
- attributes** A dictionary that stores the student's characteristics. What the exact attributes are is up to the user to decide and is most likely based on the variables to predict or assign grades. For example if the grade prediction model uses high

## Chapter 4. Curricula Simulation Framework Design & Implementation

school GPAs and ACT scores as features, then these attributes will be stored here.

The student type only has one constructor which simply takes a dictionary of student attributes as an argument. Below is an example:

```
1  # Attributes
2  attributes = Dict(
3      'ACT' => 36,
4      'HSGPA' => '3.45'
5  )
6
7  # Student Object
8  student = Student(attributes)
```

### Simulation Type

A simulation object is used to store the results of a simulation, as well as tie all of the individual pieces together so they can be easily accessed through a single object. The curriculum, students, and results are all contained within this object. This makes it easy to keep track of all the data used in a simulation as well as compare multiple simulations. This type has the following methods:

**curriculum** A curriculum object to be simulated.

**duration** The number of terms the simulation ran for.

**predictionModel** A Julia module that implements the model used for predicting student outcomes in each class.

**numStudents** The number of students that participate in the simulation.

**enrolledStudents** An array of students that are still enrolled when the simulation ends.

## Chapter 4. Curricula Simulation Framework Design & Implementation

**graduatedStudents** An array of students that have graduated during the simulation.

**stopoutStudents** An array of students that stopped out during the simulation.

**studentProgress** A *courses x students* matrix that is used keep track of which students have passed which classes. Each row is associated with a student and each column a course. The indices of each column and row match the id's of their associated student and course. A value of 1 in the matrix indicates that the row's associated student has passed the column's associated course, while a 0 indicates that the course has not been passed by the student.

**gradRate** The simulated graduation rate at the end of the simulation.

**termGradRate** An array that contains the simulated graduation rates at the end of each term.

**stopoutRate** The stopout rate at the end of the simulation.

**termStopoutRates** An array that contains the stopout rates at the end of each term.

Before any simulation, a simulation object is created that contains a curriculum and an optional Julia grade prediction module. If no module is passed in then a default is used. It has a single constructor shown below:

```
1 # A simulation object that uses the default grade prediction module.
2 sim_default = Simulation(myCurriculum)
3
4 # A simulation object that uses a custom grade prediction module (ProbitModel).
5 sim = Simulation(myCurriculum, model=ProbitModel)
```

## 4.2.2 Performance Modules

Performance modules are used to predict the grades that students make in their enrolled courses. These modules give users the ability to specify exactly what kind of prediction model they would like to use, although they must implement it themselves. Given the number of open-source Julia regression and machine learning libraries available, a wide variety of techniques can be easily implemented. This makes the simulation framework flexible and allows it to carry out a wide variety of simulations so that users can find the one that best fits their data.

A performance module is simply a Julia module that contains three functions: *train()*, *predict\_grade()*, and *predict\_stopout*. These methods are used during the simulation, and do not need to be called by the user manually, therefore it is important that they are implemented correctly. The *train()* method is called before the simulation begins and performs any kind of training that might be necessary for predictions, accepting a single curriculum object as a parameter. This gives the user access to all of the courses, where some sort of prediction model will be trained for each one based on features that the user has selected and will be included in the student's attributes. Of course these trained models will need to be stored somewhere and that's why each course has a *model* attribute. This attribute is a dictionary that can store any kind of data. The user can take advantage of this to store any information or objects relevant to the model for each course. Also, a model for predicting whether or not a student drops out must be trained. Like the course's *model* attribute, the curriculum's *stopoutModel* attribute will store any relevant information. Below is a template function:

```
1  # SampleModule
2
3  function train(curriculum)
4      # Loop through each course in a curriculum and train each one
5      for course in curriculum.courses
6          # Load some training data
7          data = read_data_here("./some/location/${course.name}.csv")
8      end
9  end
```



## Chapter 4. Curricula Simulation Framework Design & Implementation

```
9      # Train a model
10     # For example, obtain the params for a linear regression.
11     model_params = train_model(data)
12
13     # Store the model
14     course.model[:params] = model_params
15 end
16
17 # Model for predicting stopouts
18 # Here this could simply be a probability.
19 curriculum.stopoutModel[:rate] = 0.1
20 end
```

The next function, *predict\_grade()*, actually predicts a grade for a student. It takes in two arguments: a course object and a student object. Because the model for the passed in course object has already been trained, then the course's *model* contains the necessary information to construct the model and the student's *attributes* contain the features used in the model making it possible to predict a grade. As mentioned previously, grades are represented by floating point numbers that correspond to a letter grade, therefore a numeric value is expected to be returned. An example can be seen below:

```
1  # SampleModule
2
3  function predict_grade(course, student)
4      # Get the parameters from course
5      params = course.model[:params]
6
7      # Construct model based on the params
8      model = build_model(params)
9
10     # Construct a feature sample from student
11     sample = [student.attributes[:ACT], student.attributes[:GPA]]
12
13     # Predict grade
14     grade = predict(model, sample)
15
16     return grade
17 end
```

The last function in the performance module, *predict\_stopout()*, takes a student object, the current term, and the curriculum's *stopoutModel* attribute as arguments and predicts whether or not a student will stop out. This process is similar to the grade prediction: the model is constructed and given the student's features

## Chapter 4. Curricula Simulation Framework Design & Implementation

a prediction will be made. This prediction should return either true, meaning the student drops out, or false indicating that the student remains enrolled. See example below:

```
1  # SampleModule
2
3  function predict_stopout(student, currentTerm, model)
4  # Chance of stopping out
5  chance = model[:rate] - currentTerm*0.01
6
7  # Determine stopout
8  return rand() <= chance
9  end
```

Once a user constructs one of these modules, they will specify that they would like to use when they create a simulation object. However, if no module is passed in a default module built into the framework will be used. This default model simply uses the pass-rate of a course as a probability of passing a course. This model doesn't so much predict a grade as it does predict whether a student will pass a course by carrying out a single Bernoulli trial. A 'success' is interpreted as a student making an 'A' and a 'failure' results in the student receiving a 'F'. A Bernoulli trial is also used to determine stopouts, where the probability of stopping out is based on the current term and it's corresponding real-world retention value at UNM. Below is the code for this module:

```
1  # Predicts whether a student will pass a course using the course's passrate
2  # as a probability.
3
4  module PassRate
5  # Train the model
6  function train(curriculum)
7  for course in curriculum.courses
8  model = Dict()
9  model[:passrate] = course.passrate
10  course.model = model
11  end
12
13  curriculum.stopoutModel[:rates] = [0.0838, 0.1334, 0.0465, 0.0631, 0.0368,
14  0.0189, 0.0165] * 100
15
16  # Predict grade
17  function predict_grade(course, student)
18  roll = rand()
19
20  if roll <= course.model[:passrate]
```

```
21         return 4.0
22     else
23         return 0.0
24     end
25 end
26
27 # Predict stopout
28 function predict_stopout(student, currentTerm, model)
29     if currentTerm > 7
30         return false
31     else
32         roll = rand(1:100)
33         return roll <= model[rates][currentTerm]
34     end
35 end
36 end
```

### 4.2.3 Simulation Method

All of the components that have been described come together to perform what the Curriculum Flow framework was designed to do - simulate students flowing through a curriculum. This simulation is carried out through the framework's *simulate()* method. This method requires two arguments: a simulation object, and an array of students. It will also accept three optional arguments: *max\_credits*, *duration*, and *stopouts*. The first, *max\_credits*, allows the user to specify the maximum number of credit hours that a student can take in a given term. If no value is specified then the simulation defaults the value to 18. To specify how long the simulation will run for, the *duration* argument can be set. The simulation will run for the specified number of terms as long as there remains enrolled students. The default value is 8. The *stopouts* argument is a boolean value that tells the simulation whether to include stopout students. If false, then no students will ever stop out, while a true value will use the Performance Module's *stopout()* method to unenroll students at the end of every term. An example of how all these pieces come together to run a simulation is shown below:

```
1 # Import the framework
2 using CurriculumFlow
3
```

## Chapter 4. Curricula Simulation Framework Design & Implementation

```
4  # Load the user defined performance module
5  # in this case called 'ProbitModel'
6  require("./path/to/ProbitModel.jl")
7
8  # Load the Curriculum
9  curriculum = Curriculum("./path/to/curriculum.json")
10
11 # Create a set of students with a random ACT and
12 # high school GPA based on a normal distribution
13 students = []
14 for i=1:1000
15     attributes = Dict(
16         'ACT' => random_act(),
17         'HSGPA' => random_gpa()
18     )
19     s = Student(attributes)
20     push!(students, student)
21 end
22
23 # Create the simulation object
24 simulation = Simulation(curriculum; model=ProbitModel)
25
26 # Now a simulation can be run
27 simulate(simulation, students)
28 println(simulation.gradRate)
29 # => 0.73
30
31 # Tweak the parameters
32 simulate(simulation, students; duration=10, max_credis=19, stopouts=false)
33 println(simulation.gradRate)
34 # => 0.85
```

# Chapter 5

## Application in Curricular Analytics

### 5.1 Curricular Complexity

Before using simulation to evaluate and analyze curricula, it would be useful if there was a metric that describes how complex a curriculum is. Luckily, such a metric has been devised by researchers at UNM. This metric is called the *structural complexity* of a curriculum. As previously seen in Figure 3.1, a curriculum can be viewed as a graph, with nodes represented courses and edges representing the relationships between them. As its name implies, a curriculum's structural complexity quantifies the complexity of its graph structure using a combination of complex network analysis and graph theory.

Computing this metric begins by assigning each course a value named its *course cruciality*. This value indicates its importance in the curriculum based on its relationship to other courses. This value in turn is comprised of two other features: the course's *blocking factor* and its *delay factor*. A course's delay factor is defined as the number of nodes on the longest path that passes through the given course. That is the longest path, An example can be seen in Figure 5.1. In this figure, course A

has a delay factor of three because the longest path that passes through it has three courses: A, B, and D as opposed to the shorter path with length two which is made up of courses A and C. Courses B and D share course A's delay factor of three while course C has a delay factor of two.

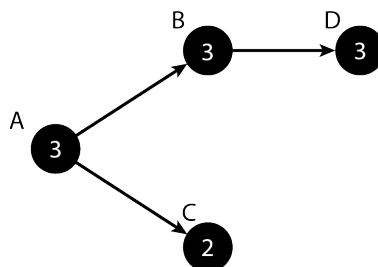


Figure 5.1: Delay factors are given for courses A, B, C, and D.

A course's blocking factor is defined as the number of courses that are blocked from being taken if the given course is not passed. This is essentially the connectivity of the course. If there is a path from  $i$  to  $j$  then  $n_{ij}$  is 1 and 0 otherwise. Then, the blocking factor of course  $i$  would be given by

$$V_i = \sum_j n_{ij} \quad (5.1)$$

An example of blocking factors can be seen in Figure 5.2. Here, course A has a blocking factor of three because it is connected (blocks) three other courses: B, C and D. Course B blocks one course, D and courses C and D have a blocking factor of zero because no course succeeds them.

Given a course's delay factor path,  $L_i$ , and blocking factor,  $V_i$ , then its cruciality,  $C_i$  is simply given by the sum of the two:

$$C_i = V_i + L_i \quad (5.2)$$

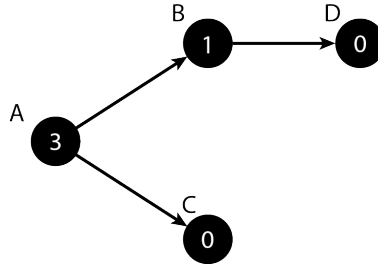


Figure 5.2: Blocking Factors are given for courses A, B, C, and D.

A curriculum's complexity,  $S$  is then given by the sum over all course crucialities:

$$S = \sum_i^n C_i \quad (5.3)$$

While it is far from the only factor for student success, it is intuitive to see that a curriculum's structural complexity can impact the time in which it takes students to move through a program. Students that fail a course with a high blocking factor will be unable to take needed courses the following semester potentially delaying them and curricula with high blocking factors present many opportunities for being delayed. Curricula with long paths, and therefore high delay factors, can also delay students and even impose minimum term completion limits. For example, a curricula with a long path of eight requires at least four years to complete and if a course on that path is failed, the student will be delayed at least one semester.

## 5.2 Correlating Complexity with Completion Rates

One of the motivations for the development of the curriculum flow simulation is to show that there is a correlation between structural complexity and the rates at which students complete the curricula. As a start, several tests were carried out over extremely simple curricula seen in 5.3. These curricula, consisting of two terms with

## Chapter 5. Application in Curricular Analytics

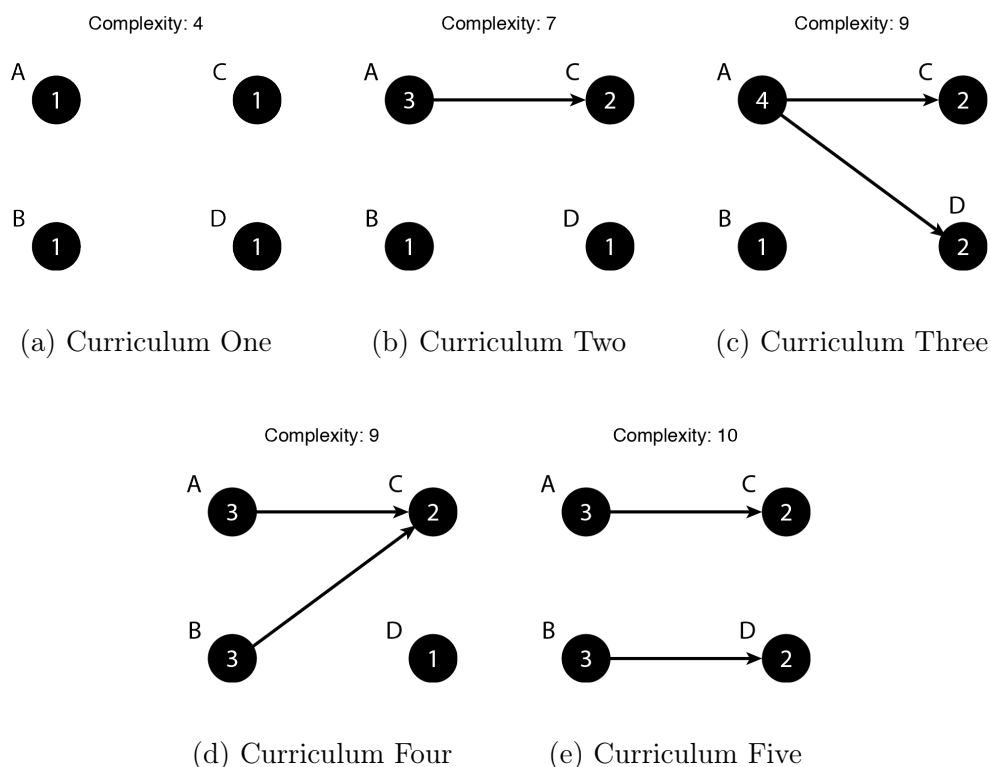


Figure 5.3: Basic Curricula

two courses each make up common patterns that can be found in actual curricula and span a range of complexity values making them good candidates to observe how structure affects the way students flow through curricula.

Each simulation carried out over the five curricula using the following configuration. The simulation's default performance module is used meaning that a student is determined to either pass or fail based on the course's pass-rate. Every course has a pass-rate of 50%, therefore each student has a one in two chance of passing the course. The simulation's duration is four semesters, and a student is allowed to take up to nine credit hours a semester with each course counting for three. Each curriculum is simulated one hundred times with one hundred students each and the results from each are then averaged. The results can be seen in Table 5.1.



## Chapter 5. Application in Curricular Analytics

Table 5.1

(a) Curriculum One					(b) Curriculum Two				
Course	Term 1	Term 2	Term 3	Term 4	Course	Term 1	Term 2	Term 3	Term 4
A	49.62	75.2	87.33	93.73	A	51.19	75.31	87.23	93.48
B	50.64	75.48	87.64	93.97	B	49.94	74.71	87.31	93.45
C	49.98	75.62	87.98	94.14	C	0.0	25.5	51.27	69.39
D	0.0	44.1	71.89	85.77	D	49.3	75.2	87.31	93.59
Completion Rate	0.0	20.72	49.5	71.63	Completion Rate	0.0	14.35	39.21	60.41

(c) Curriculum Three					(d) Curriculum Four				
Course	Term 1	Term 2	Term 3	Term 4	Course	Term 1	Term 2	Term 3	Term 4
A	49.96	74.78	87.43	93.38	A	49.98	75.12	87.93	93.92
B	49.68	74.57	87.28	93.95	B	49.76	75.44	87.37	93.73
C	0.0	25.59	50.51	69.2	C	0.0	12.35	34.7	55.21
D	0.0	24.61	49.64	68.83	D	49.98	75.4	87.79	94.33
Completion Rate	0.0	8.88	30.01	52.57	Completion Rate	0.0	9.31	30.44	52.3

(e) Curriculum Five				
Course	Term 1	Term 2	Term 3	Term 4
A	49.9	74.99	87.75	93.6
B	49.93	75.1	88.0	94.05
C	0.0	25.21	50.07	68.75
D	0.0	24.61	49.47	68.61
Completion Rate	0.0	6.07	24.5	46.93

These tables show the percentage of students that have passed each course at the end of each semester. For example in Table 5.1a, 75.2% of students had passed Course A after the second semester. The row at the bottom then shows the percentage of students that had completed all courses at the end of each term. As expected, the least complex curricula resulted in the highest completion rates after four terms with completion rates dropping as complexity increased. Its also important to note that the two curricula with the same complexity had very similar completion rates despite their structures being slightly different.

This inverse relationship continues for larger curricula as well. A similar test was conducted over thirty five curricula from Universities around the country. These curricula were pulled from [curricula.academicdashboards.org](http://curricula.academicdashboards.org), a web service that stores, visualizes, and computes the complexity of curricula. Each curriculum used in the experiment contain eight terms and at least one hundred credit hours. Each curricu-

lum was simulated for a duration of ten terms one hundred times with every course having a pass-rate of %80. The results of which can be seen in Figure 5.4.

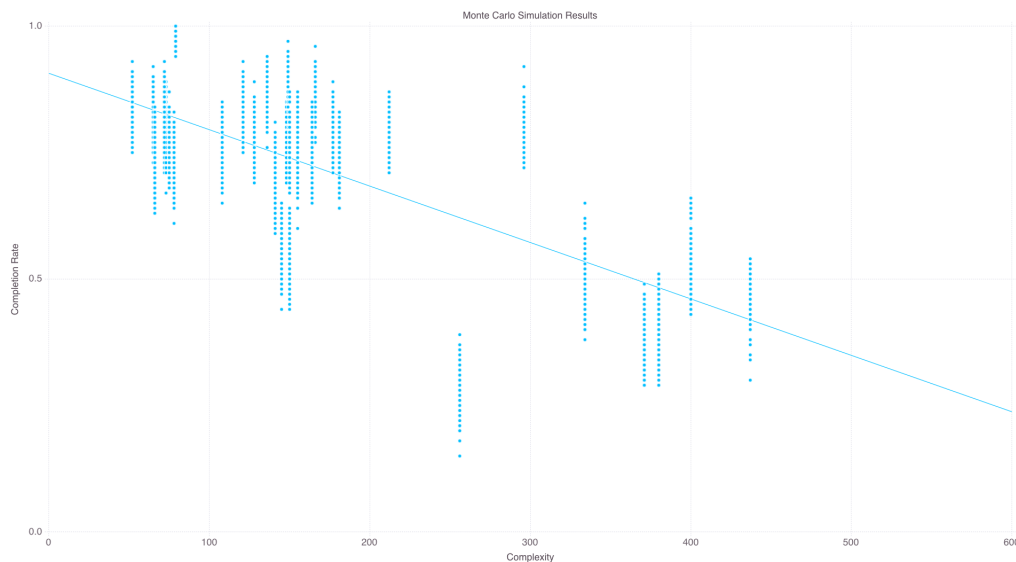


Figure 5.4: Simulation results of thirty five real-world curricula

The line seen in the figure above is the result of a linear regression over the data. It clearly shows that completion rates decrease as complexity increases. More specifically, it shows that for every point of complexity, a curriculum's completion rate after ten terms will decrease by 0.1%. This may not seem like much, but as seen in 5.4, curriculum can vary widely in complexity. Even curriculum with the same number of terms and similar credit hours can easily differ one hundred points in complexity which would equate to a 10% completion rate difference, which is significant.

### 5.3 Sensitivity Analysis of Instructional Complexity

Another metric that is useful when working with curricula is the instructional complexity of a curriculum. This can be thought of as the difficulty of the curriculum,

## Chapter 5. Application in Curricular Analytics

which in term will be defined by the difficulty of the courses that make it up. Defining how difficulty a course is not straightforward and can be done in many ways, but in the experiments described here, the naive definition of the course's pass-rate is used. Another use of the curriculum flow simulation framework is to observe the effects that instructional complexity has, as it is intuitive to believe that the difficulty of a curriculum's courses would greatly affect student performance. Although not a realistic scenario, Figure 5.5 illustrates this point.

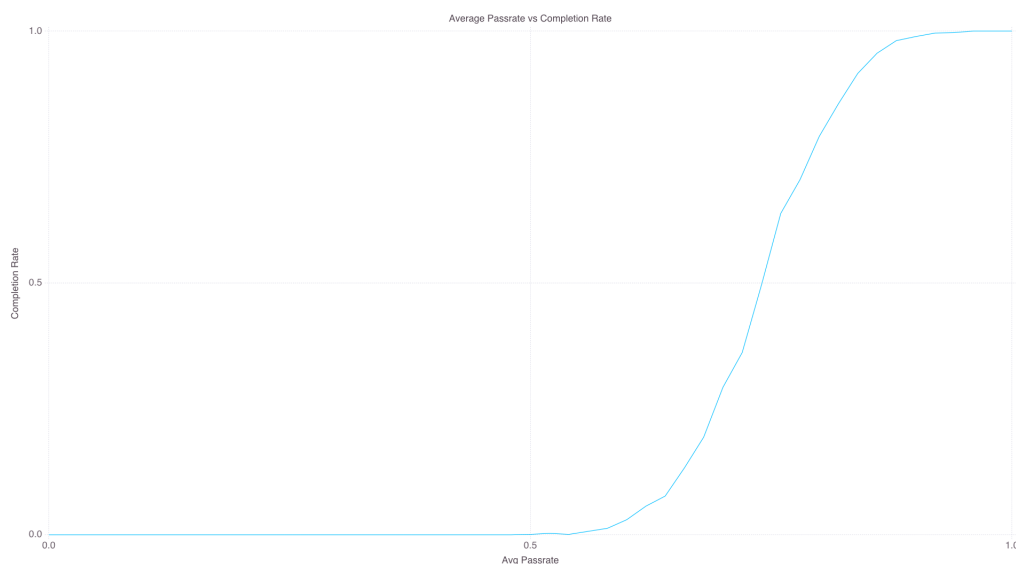


Figure 5.5: Graph showing how the completion rate of the Computer Engineering curriculum at UNM changes with respect to course pass-rates.

This figure shows the completion rate of the Computer Engineering curriculum for a given pass-rate assigned to every course. As expected, the completion rates increase as course pass-rates increase and shows how the instructional complexity of a curriculum affects student success. One of the most common methods for increasing student success is to increase student success in individual courses. Regardless of the method used, increasing the pass-rate of critical courses can increase student graduation rates. So then the question becomes which courses should receive the

## Chapter 5. Application in Curricular Analytics

most attention?

This question can be answered by performing sensitivity analysis over course pass-rates via simulation. This analysis is done by performing a baseline simulation and then increasing a course's pass-rate and then measuring the difference. This experiment was carried out over three UNM curricula, Computer Engineering, Accounting, and Mechanical Engineering, where pass-rates of every were computed using UNM data over the last ten years. To get a baseline completion rate, a simulation over 1000 students was carried out 20 times and then the eighth term completion rates were averaged. Then, one at a time, each course's pass-rate is increased up to thirty percent and then an identical simulation was carried out. Figures 5.2, 5.3, and 5.4 show the results of this analysis - specifically the top 10 courses in which an increase in pass-rate had the greatest affect on eighth-term completion rates.

Course	Original Pass-rate	Increased Pass-rate	Cruciality	Increase in Completion Rate
Elective	0.9	1	1	0.0836
MATH 180	0.7397	0.9616	4	0.0282
MGMT 202	0.8038	1	12	0.028
MATH 121	0.7389	0.9606	10	0.0274
ECON 105	0.7904	1	1	0.0227
ECON 106	0.7984	1	8	0.0224
MGMT 340	0.8943	1	6	0.0154
MGMT 326	0.8605	1	3	0.0147
Foreign Language	0.9	1	1	0.0142
Physical & Natural Science	0.9	1	1	0.0139

Table 5.2: Accounting Sensitivity Analysis

Course	Original Pass-rate	Increased Pass-rate	Cruciality	Increase in Completion Rate
MATH 162	0.7585	0.9861	29	0.0407
MATH 163	0.7502	0.9753	26	0.0404
MATH 264	0.8399	1	16	0.0265
PHYC 160	0.7926	1	25	0.0212
ECON 105	0.7904	1	1	0.0206
Tech Elective	0.9	1	1	0.0195
Tech Elective	0.9	1	1	0.0163
CHEM 121	0.8049	1	13	0.0158
Tech Elective	0.9	1	1	0.0143
MATH 316	0.8633	1	18	0.0108

Table 5.3: Mechanical Engineering Sensitivity Analysis

In each curricula, the courses with the highest cruciality had some of the greatest increases in completion rates with an increase in their pass-rates. This makes sense as these are courses are on long-paths and block other crucial courses. This analysis also

## Chapter 5. Application in Curricular Analytics

Course	Original Pass-rate	Increased Pass-rate	Cruciality	Increase in Completion Rate
Humanities	0.9	1	1	0.0661
MATH 162	0.7585	0.9861	19	0.0609
MATH 163	0.7502	0.9753	18	0.0554
Fine Arts	0.9	1	1	0.0474
Tech Elective	0.9	1	1	0.0396
PHYC 160	0.7926	1	14	0.0287
MATH 264	0.8399	1	5	0.0251
ECON 105	0.7904	1	1	0.0239
ECE 344L	0.9026	1	6	0.0237
ECE 131	0.8571	1	13	0.0218

Table 5.4: Computer Engineering Sensitivity Analysis

## 5.4 Probit Performance Model

## Chapter 6

## Discussion

## Chapter 7

### Future Work

# References

- [1] J. Bezanon, S. Karpinski, V. Shah, and A. Edelman. Julia: A fast dynamic language for technical computing. In *Lang.NEXT*, Apr. 2012.
- [2] M. S. Hunter and E. R. White. Could fixing academic advising fix higher education?. *About Campus*, 9(1):20–25, 2004.
- [3] K. P. King. Identifying success in online teacher education and professional development. *The Internet and Higher Education*, 5(3):231–246, 2002.
- [4] W. C. Lewallen. Early alert: A report on two pilot projects at antelope valley college. 1993.
- [5] J. O’Flaherty and C. Phillips. The use of flipped classrooms in higher education: A scoping review. *The Internet and Higher Education*, 25:85–95, 2015.
- [6] K. J. Topping. The effectiveness of peer tutoring in further and higher education: A typology and review of the literature. *Higher education*, 32(3):321–345, 1996.